

Hans Berger

# Automating with SIMATIC S7-1200

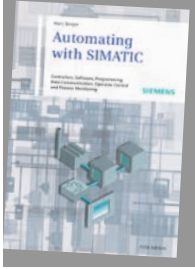
Configuring, Programming and Testing  
with STEP 7 Basic  
Visualization with WinCC Basic

**SIEMENS**



Second Edition

Berger Automating with SIMATIC S7-1200

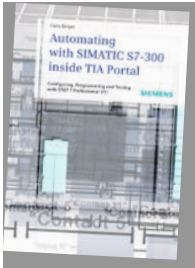


Hans Berger

## Automating with SIMATIC

**Controllers, Software, Programming,  
Data Communication, Operator Control  
and Process Monitoring**

5<sup>th</sup> revised and enlarged edition, 2012,  
284 pages, 140 illustrations, 49 tables, hardcover  
ISBN 978-3-89578-387-6, € 44.90



Hans Berger

## Automating with SIMATIC S7-300 inside TIA Portal

**Configuring, Programming and Testing  
with STEP 7 Professional V11**

2012, 709 pages, 429 illustrations,  
85 tables, hardcover  
ISBN 978-3-89578-382-1, € 69.90

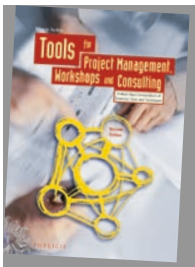


Hans Berger

## Automating with SIMATIC S7-400 inside TIA Portal

**Configuring, Programming and Testing  
with STEP 7 Professional**

June 2013, ca. 760 pages,  
441 illustrations, 94 tables, hardcover  
ISBN 978-3-89578-383-8, € 69.90



Nicolai Andler

## Tools for Project Management, Workshops and Consulting

**A Must-Have Compendium of  
Essential Tools and Techniques**

2<sup>nd</sup> revised and enlarged edition, 2011,  
382 pages, 136 illustrations, 55 tables, hardcover  
ISBN 978-3-89578-370-8, € 39.90



# Automating with SIMATIC S7-1200

Configuring, Programming and  
Testing with STEP 7 Basic  
Visualization with HMI Basic

by Hans Berger

2<sup>nd</sup> enlarged and revised edition, 2013

Publicis Publishing

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet at <http://dnb.d-nb.de>.

The author, translators and publisher have taken great care with all texts and illustrations in this book. Nevertheless, errors can never be completely avoided. The publisher, author and translators accept no liability, for whatever legal reasons, for any damage resulting from the use of the programming examples.

[www.publicis-books.de](http://www.publicis-books.de)

**Print ISBN: 978-3-89578-385-2**

**ePDF ISBN: 978-3-89578-901-4**

2<sup>nd</sup> edition, 2013

Editor: Siemens Aktiengesellschaft, Berlin and Munich

Publisher: Publicis Publishing, Erlangen

© 2013 by Publicis Erlangen, Zweigniederlassung der PWW GmbH

This publication and all parts thereof are protected by copyright. Any use of it outside the strict provisions of the copyright law without the consent of the publisher is forbidden and will incur penalties. This applies particularly to reproduction, translation, microfilming or other processing, and to storage or processing in electronic systems. It also applies to the use of individual illustrations or extracts from the text.

Printed in Germany

## Preface

The SIMATIC automation system unites all the subsystems of an automation solution under a uniform system architecture to form a homogenous whole from the field level right up to process control.

The *Totally Integrated Automation* concept permits uniform handling of all automation components using a single system platform and tools with uniform operator interfaces. These requirements are fulfilled by the SIMATIC automation system which provides uniformity for configuration, programming, data management and communication.

This book describes the newly developed SIMATIC S7-1200 automation system. The S7-1200 programmable controllers are of compact design and allow modular expansion. Many small applications can be solved using the CPU module with on-board I/O. The technological functions integrated in the CPU module mean that extremely versatile use of the device is possible. Two established programming languages are available for solving automation tasks: ladder logic (LAD) and function block diagram (FBD).

New SIMATIC HMI Basic Panels have been designed for operator control and monitoring appropriate to the S7-1200 programmable controllers, and provide a performance and functionality optimized for small applications. A touch screen with various monitor sizes and coordinated communication over Industrial Ethernet are ideal prerequisites for interaction with S7-1200.

The STEP 7 Basic engineering software makes it possible to use all S7-1200 controller options. STEP 7 Basic is the common tool for hardware configuration, generation of the control program, and for debugging and diagnostics. The SIMATIC WinCC Basic configuration software included in STEP 7 Basic is used to configure the Basic Panels. Modern and intuitive user guidance allows efficient and task-oriented engineering of control and visualization devices.

This book describes the S7-1200 automation system with S7-1200 programmable controllers and HMI Basic Panels. The description focuses on the generation of the control program using STEP 7 Basic engineering software Version 11 SP2.

Nuremberg, February 2013

Hans Berger

# The contents of the book at a glance

## Start

### Introduction

**SIMATIC S7-1200:** Overview of the SIMATIC S7-1200 automation system.

**STEP 7 Basic:** Introduction to the engineering software for SIMATIC S7-1200.

**SIMATIC project:** Basic functions for the automation solution.

## Devices & networks

### The hardware components of S7-1200

**Modules:** Overview of the SIMATIC S7-1200 modules.

### Device configuration

**Hardware configuration:** Configuration of the hardware design.

**Network configuration:** Configuration of a communication network.

## PLC programming

### The control program

**Operating modes:** How the CPU module responds with STARTUP, RUN and STOP.

**Processing modes:** Restart characteristics, main program, interrupt processing, and error handling define the processing of the control program.

**Blocks:** Organization blocks, function blocks, functions, and data blocks structure the control program.

### The program editor

**Programming:** How the control program is produced.

**Program information:** Tools for supporting programming.

### Ladder logic and function block diagram as programming languages

**Program elements:** The characteristics of LAD and FBD programming; the use of contacts, coils, standard boxes, Q boxes and EN/ENO boxes.

### Tags and data types

**Tags:** Operand areas, project-wide and block-local tags, addressing.

**Data types:** Description of elementary and compound data types.

### Description of the control functions

**Basic functions:** Binary operations, memory functions, edge evaluation, timer and counter functions.

**Digital functions:** Move, comparison, arithmetic, math, conversion, shift, and logic functions.

**Program flow control:** Jump functions, block end function, block calls.

## Online & diagnostics

### Connection of programming device to PLC station

**Online operation:** Establish connection to PLC station.

**Status LEDs:** The modules signal an error.

**Diagnostics information:** Find the error using the diagnostics information.

**Online tools:** Control the CPU module using the online tools.

### Online & offline project data

**Download:** Download control program into CPU memory.

**Blocks:** Edit and compare the blocks offline/online.

**Test:** Test the control function using program status and monitoring tables.

## Data communication

### Open user communication

**Data transmission:** Data exchange from PLC to PLC over Ethernet.

### Point-to-point connection

**PtP:** Data transmission with CM modules via RS232 and RS485.

## Visualization

### Configuration of Basic Panels

**Introduction:** Overview of Basic Panels.

**Start:** Create an HMI project, the HMI device wizard.

**Connection to the PLC:** Create HMI tags and area pointers.

**Create screens:** Configuration of process screens – templates, layers and screen changeover.

**Working with image elements:** Arrange and edit operator control and display elements, configure a message system, create recipes, transfer data records, configure user management.

### Complete the HMI program

**Simulation:** Simulate the HMI program with PLC station or with tag table.

**Connection:** Transfer the HMI program to the HMI station.

## Appendix

### Integral and technological functions

**Functions:** High-speed counter, pulse generator, motion control, PID controller.

### Global libraries

**Overview:** USS drive control, MODBUS blocks.

---



# Table of contents

<b>1 Introduction</b> .....	21
1.1 Overview of the S7-1200 automation system .....	21
1.1.1 SIMATIC S7-1200 .....	22
1.1.2 Overview of STEP 7 Basic .....	24
1.1.3 Three programming languages .....	25
1.1.4 Execution of the user program .....	27
1.1.5 Data management in the SIMATIC automation system .....	29
1.1.6 Operator control and monitoring with process images .....	30
1.2 Introduction to STEP 7 Basic for S7-1200 .....	31
1.2.1 Installing STEP 7 .....	31
1.2.2 Automation License Manager .....	31
1.2.3 Starting STEP 7 Basic .....	32
1.2.4 Portal view .....	32
1.2.5 Help Information system .....	33
1.2.6 The windows of the project view .....	34
1.2.7 Adapting the user interface .....	36
1.3 Editing a SIMATIC project .....	37
1.3.1 Structured representation of project data .....	38
1.3.2 Project data and editors for a PLC station .....	39
1.3.3 Creating and editing a project .....	41
1.3.4 Creating and editing libraries .....	42
<b>2 SIMATIC S7-1200 automation system</b> .....	43
2.1 S7-1200 station components .....	43
2.2 S7-1200 CPU modules .....	44
2.2.1 Integrated I/O .....	44
2.2.2 PROFINET connection .....	46
2.2.3 Status LEDs .....	47
2.2.4 SIMATIC Memory Card .....	47
2.2.5 Expansions of the CPU .....	47
2.3 Signal modules (SM) .....	49
2.3.1 Digital I/O modules .....	49
2.3.2 Analog input/output modules .....	50
2.3.3 Properties of the I/O connections .....	50
2.4 Communication modules (CM) .....	52
2.4.1 Point-to-point communication .....	52
2.4.2 PROFIBUS DP .....	52
2.4.3 Actuator/sensor interface .....	53
2.4.4 GPRS transmission .....	53
2.5 Further modules .....	54
2.5.1 Compact switch module (CSM) .....	54

2.5.2 Power module (PM)	54
2.5.3 TS Adapter IE Basic	54
2.5.4 SIM 1274 simulator	55
2.6 SIPLUS S7-1200	55
<b>3 Device configuration</b>	<b>57</b>
3.1 Introduction	57
3.2 Configuring a station	60
3.2.1 Adding a PLC station	60
3.2.2 Arranging modules	61
3.2.3 Adding an HMI station	61
3.3 Assigning module parameters	61
3.3.1 Parameterization of CPU properties	61
3.3.2 Addressing input and output signals	64
3.3.3 Parameterization of digital inputs	65
3.3.4 Parameterization of digital outputs	65
3.3.5 Parameterization of analog inputs	66
3.3.6 Parameterization of analog outputs	66
3.4 Configuring the network	67
3.4.1 Introduction	67
3.4.2 Networking stations	68
3.4.3 Node addresses in a subnet	69
3.4.4 Connectors	70
3.4.5 Configuring a PROFINET subnet	73
3.4.6 Configuring a PROFIBUS subnet	75
3.4.7 Configuring an AS-i subnet	77
<b>4 Variables and data types</b>	<b>79</b>
4.1 Operands and tags	79
4.1.1 Introduction, overview	79
4.1.2 Operand areas: inputs and outputs	80
4.1.3 Operand area bit memory	82
4.1.4 Operand area data	84
4.1.5 Operand area temporary local data	85
4.2 Addressing	85
4.2.1 Signal path	85
4.2.2 Absolute addressing of an operand	86
4.2.3 Absolute addressing of an operand area	86
4.2.4 Symbolic addressing	88
4.2.5 Addressing a tag part	89
4.2.6 Addressing constants	89
4.2.7 Indirect addressing	89
4.3 General information on data types	92
4.3.1 Overview of data types	92
4.3.2 Implicit data type conversion	93
4.3.3 Overlying tags (data type views)	93
4.4 Elementary data types	95
4.4.1 Bit-serial data types BOOL, BYTE, WORD and DWORD	95

4.4.2	BCD-coded numbers BCD16 and BCD32	95
4.4.3	Unsigned fixed-point data types USINT, UINT and UDINT	97
4.4.4	Fixed-point data types with sign SINT, INT and DINT	98
4.4.5	Floating-point data types REAL and LREAL	98
4.4.6	Data type CHAR	100
4.4.7	Data type DATE	100
4.4.8	Data type TIME	100
4.4.9	TIME_OF_DAY (TOD) data type	101
4.5	Structured data types	101
4.5.1	Data type DTL	101
4.5.2	Data type STRING	102
4.5.3	Data type ARRAY	104
4.5.4	Data type STRUCT	104
4.6	Parameter types	107
4.6.1	Parameter types for IEC timer functions	107
4.6.2	Parameter types for IEC counter functions	108
4.6.3	Parameter type VARIANT	108
4.6.4	Parameter type VOID	109
4.7	PLC data types	109
4.8	System data types	110
4.8.1	IEC_TIMER system data type	110
4.8.2	IEC_COUNTER system data type	112
4.8.3	TCON_Param data type	112
4.8.4	TADDR_Param data type	112
4.8.5	Data type ErrorStruct	112
4.8.6	TimeTransformationRule data type	115
4.9	Hardware data types	115
<b>5</b>	<b>Edit user program</b>	<b>117</b>
5.1	Operating modes	117
5.1.1	STOP mode	118
5.1.2	STARTUP mode	118
5.1.3	RUN mode	119
5.1.4	Retentive behavior of operands	121
5.2	Creating a user program	122
5.2.1	Program draft	122
5.2.2	Program execution	123
5.2.3	Nesting depth	125
5.3	Programming blocks	125
5.3.1	Block types	125
5.3.2	Editing block properties	128
5.3.3	Configuring know-how protection	132
5.3.4	Copy protection	132
5.3.5	Block interface	133
5.3.6	Programming block parameters	136
5.4	Calling blocks	137
5.4.1	General information on calling logic blocks	137
5.4.2	Calling a function (FC)	139

---

5.4.3	Calling a function block (FB)	140
5.4.4	“Passing on” of block parameters	142
5.5	Start-up routine	142
5.6	Main program	143
5.6.1	Organization blocks for the main program	143
5.6.2	Process image update	143
5.6.3	Cycle time	144
5.6.4	Reaction time	146
5.6.5	Stop program execution	147
5.6.6	Time	148
5.6.7	Runtime meter	151
5.7	Interrupt processing	153
5.7.1	Introduction to interrupt processing	153
5.7.2	Time-delay interrupts	155
5.7.3	Cyclic interrupts	159
5.7.4	Process interrupts	163
5.7.5	Assigning interrupts during runtime	164
5.7.6	Delay and enable interrupts	166
5.8	Troubleshooting, diagnostics	167
5.8.1	Causes of errors and responses	167
5.8.2	Error display with the ENO output	168
5.8.3	Time error OB 80	168
5.8.4	Local error handling	169
5.8.5	Diagnostic functions in the user program	172
5.8.6	Diagnostics interrupt OB 82	176
<b>6</b>	<b>Program editor</b>	<b>178</b>
6.1	Introduction	178
6.2	PLC tag table	178
6.2.1	Creating and editing the PLC tag table	179
6.2.2	Defining PLC tags	179
6.2.3	Editing a PLC tag table	181
6.2.4	Exporting and importing a PLC tag table	181
6.2.5	Constants tables	182
6.3	Programming a code block	183
6.3.1	Creating a new code block	183
6.3.2	Working area of program editor for code blocks	184
6.3.3	Specifying code block properties	186
6.3.4	Programming a block interface	186
6.3.5	Programming control functions	188
6.3.6	Editing tags	192
6.3.7	Working with program comments	193
6.4	Programming a data block	194
6.4.1	Creating a new data block	194
6.4.2	Working area of program editor for data blocks	195
6.4.3	Defining properties for data blocks	196
6.4.4	Declaring data tags	196
6.4.5	Entering data tags in global data blocks	198

6.5	Compiling blocks	198
6.5.1	Starting the compilation	198
6.5.2	Compiling SCL blocks	199
6.5.3	Eliminating errors following compilation	200
6.6	Program information	201
6.6.1	Cross-reference list	201
6.6.2	Assignment list	203
6.6.3	Call structure	204
6.6.4	Dependency structure	205
6.6.5	Consistency check	206
6.6.6	CPU resources	206
6.7	Language setting	207
<b>7</b>	<b>Ladder logic LAD</b>	<b>209</b>
7.1	Introduction	209
7.1.1	Programming with LAD in general	209
7.1.2	Program elements of ladder logic	211
7.2	Programming with contacts	212
7.2.1	NO and NC contacts	212
7.2.2	Consideration of sensor type in ladder logic	213
7.2.3	Series connection of contacts	215
7.2.4	Parallel connection of contacts	215
7.2.5	Mixed series and parallel connections	216
7.2.6	T branch, open parallel branch in the ladder logic	217
7.2.7	Negating result of logic operation in the ladder logic	218
7.2.8	Edge evaluation of a binary tag in ladder logic	218
7.2.9	OK contact	219
7.2.10	Comparison contacts	219
7.3	Programming with coils	221
7.3.1	Simple and negated coils	222
7.3.2	Set and reset coil	223
7.3.3	Retentive response due to latching	223
7.3.4	Edge evaluation with pulse output in the ladder logic	224
7.3.5	Multiple setting and resetting (filling of bit field) in the ladder logic	225
7.3.6	Starting IEC timer functions in the ladder logic with coils	225
7.4	Programming with Q boxes in the ladder logic	226
7.4.1	Arrangement of Q boxes in the ladder logic	226
7.4.2	Memory boxes in the ladder logic	227
7.4.3	Edge evaluation of current flow	229
7.4.4	Example of binary scaler in the ladder logic	229
7.4.5	Controlling IEC timer functions in the ladder logic with Q boxes	230
7.4.6	Controlling IEC counter functions in the ladder logic with Q boxes	231
7.5	Programming with EN/ENO boxes in the ladder logic	233
7.5.1	Positioning of EN/ENO boxes in the ladder logic	234
7.5.2	Transfer functions in the ladder logic	235
7.5.3	Arithmetic functions for numerical values in the ladder logic	236
7.5.4	Arithmetic functions for time values in the ladder logic	236
7.5.5	Math functions in the ladder logic	237

7.5.6	Conversion functions in the ladder logic	238
7.5.7	Shift functions in the ladder logic	239
7.5.8	Logic functions in the ladder logic	240
7.5.9	Functions for strings in the ladder logic	240
7.6	Functions for program flow control (LAD)	241
7.6.1	Jump functions in the ladder logic	242
7.6.2	Jump list in the ladder logic	243
7.6.3	Jump distributor in the ladder logic	244
7.6.4	Block end function in the ladder logic	244
7.6.5	Block call functions in the ladder logic	245
<b>8</b>	<b>Function block diagram FBD</b>	<b>246</b>
8.1	Introduction	246
8.1.1	Programming with function block diagram in general	246
8.1.2	Program elements of the function block diagram	248
8.2	Programming of binary logic operations (FBD)	249
8.2.1	Scanning for signal states “1” and “0”	250
8.2.2	Taking account of the sensor type in the function block diagram	251
8.2.3	AND function	252
8.2.4	OR function	253
8.2.5	Exclusive OR function	254
8.2.6	Mixed binary logic operations	254
8.2.7	T branch in the function block diagram	255
8.2.8	Negate result of logic operation in the function block diagram	255
8.2.9	Edge evaluation of binary tags in the function block diagram	256
8.2.10	Validity checking of floating-point numbers in the function block diagram	257
8.2.11	Comparison functions in the function block diagram	258
8.3	Programming with standard boxes (FBD)	258
8.3.1	Assignment and negated assignment	259
8.3.2	Set and reset boxes	260
8.3.3	Edge evaluation with pulse output in the function block diagram	261
8.3.4	Multiple setting and resetting (filling of bit field) in the function block diagram	262
8.3.5	Starting IEC timer functions in the function block diagram with standard boxes	262
8.4	Programming with Q boxes (FBD)	264
8.4.1	Arrangement of Q boxes in the function block diagram	264
8.4.2	Memory boxes in the function block diagram	265
8.4.3	Edge evaluation of logic operation result in the function block diagram	266
8.4.4	Example of binary scaler in the function block diagram	267
8.4.5	Controlling IEC timer functions in the function block diagram with Q boxes	267
8.4.6	IEC counter functions in the function block diagram	268
8.5	Programming with EN/ENO boxes (FBD)	270
8.5.1	Positioning of EN/ENO boxes in the function block diagram	270
8.5.2	Transfer functions in the function block diagram	271

8.5.3	Arithmetic functions for numerical values in the function block diagram . . . . .	273
8.5.4	Arithmetic functions with time values in the function block diagram . . . . .	273
8.5.5	Math functions in the function block diagram . . . . .	274
8.5.6	Conversion functions in the function block diagram . . . . .	275
8.5.7	Shift functions in the function block diagram . . . . .	276
8.5.8	Logic functions in the function block diagram . . . . .	277
8.5.9	Functions for strings in the function block diagram . . . . .	278
8.6	Functions for program flow control (FBD) . . . . .	279
8.6.1	Jump functions in the function block diagram . . . . .	280
8.6.2	Jump list in the function block diagram . . . . .	281
8.6.3	Jump distributor in the function block diagram . . . . .	281
8.6.4	Block end function in the function block diagram . . . . .	282
8.6.5	Block call functions in the function block diagram . . . . .	282
<b>9</b>	<b>Structured Control Language SCL . . . . .</b>	<b>284</b>
9.1	Introduction to programming with SCL . . . . .	284
9.1.1	Programming with SCL in general . . . . .	284
9.1.2	SCL statements and operators . . . . .	286
9.2	Programming binary logic operations with SCL . . . . .	288
9.2.1	Scanning for signal states “1” and “0” . . . . .	288
9.2.2	Taking account of the sensor type for SCL . . . . .	289
9.2.3	AND function . . . . .	291
9.2.4	OR function . . . . .	291
9.2.5	Exclusive OR function . . . . .	292
9.2.6	Combined binary logic operations . . . . .	292
9.2.7	Negating the result of logic operation . . . . .	293
9.3	Programming memory functions with SCL . . . . .	294
9.3.1	Value assignment of a binary tag . . . . .	294
9.3.2	Setting and resetting . . . . .	294
9.3.3	Edge evaluation . . . . .	295
9.4	Programming timer and counter functions with SCL . . . . .	296
9.4.1	IEC timer functions . . . . .	296
9.4.2	IEC counter functions . . . . .	297
9.5	Programming digital functions with SCL . . . . .	298
9.5.1	Transfer function, value assignment of a digital tag . . . . .	298
9.5.2	Conversion functions . . . . .	299
9.5.3	Comparison functions . . . . .	301
9.5.4	Arithmetic functions . . . . .	301
9.5.5	Mathematical functions . . . . .	303
9.5.6	Word logic operations . . . . .	303
9.5.7	Shift functions . . . . .	304
9.6	Controlling the program flow with SCL . . . . .	305
9.6.1	Working with the ENO tag . . . . .	305
9.6.2	EN/ENO mechanism with SCL . . . . .	306
9.6.3	Control statements . . . . .	307
9.6.4	Block functions . . . . .	316
9.7	Working with source files . . . . .	319

---

9.7.1	General procedure	319
9.7.2	Programming a logic block in the source file	321
9.7.3	Programming a data block in the source file	325
9.7.4	Programming a PLC data type in the source file	327
<b>10</b>	<b>Basic functions</b>	<b>328</b>
10.1	Binary logic operations	328
10.1.1	Introduction	328
10.1.2	Scanning for signal states “1” and “0”, result of the scan	329
10.1.3	Negating the result of the logic operation, NOT contact	329
10.1.4	Testing floating-point tag, OK contact, OK box	330
10.1.5	AND function, series connection	331
10.1.6	OR function, parallel connection	332
10.1.7	Exclusive OR function, non-equivalence function	333
10.2	Memory functions	334
10.2.1	Introduction	334
10.2.2	Simple and negated coil, assignment	334
10.2.3	Single set and reset	335
10.2.4	Multiple setting and resetting	336
10.2.5	Dominant setting and resetting, memory boxes	337
10.3	Edge evaluation	338
10.3.1	Functional principle of an edge evaluation	338
10.3.2	Edge evaluation of the result of the logic operation	340
10.3.3	Edge evaluation of a binary tag	341
10.3.4	Edge evaluation with pulse output	342
10.4	Time functions	344
10.4.1	Introduction	344
10.4.2	Pulse generation TP	346
10.4.3	On-delay TON	347
10.4.4	OFF delay TOF	347
10.4.5	Accumulating ON delay TONR	348
10.5	Counter functions	349
10.5.1	Introduction	349
10.5.2	Up counter CTU	351
10.5.3	Down counter CTD	352
10.5.4	Up-down counter CTUD	353
<b>11</b>	<b>Digital functions</b>	<b>355</b>
11.1	Transfer functions	356
11.1.1	Introduction	356
11.1.2	Copy tag, MOVE box for LAD and FBD	356
11.1.3	Copy string, S_MOVE box for LAD and FBD	357
11.1.4	Value assignments with SCL	358
11.1.5	Copy data area (MOVE_BLK, UMOVE_BLK)	360
11.1.6	Filling the data area (FILL_BLK, UFILL_BLK)	361
11.1.7	Read and write the load memory (READ_DBL, WRIT_DBL)	362
11.1.8	Swap bytes (SWAP)	363
11.2	Comparison functions	364



11.2.1	Overview	364
11.2.2	Comparison of two tag values	364
11.2.3	Range comparison	365
11.3	Arithmetic functions for numerical values	366
11.3.1	Introduction	366
11.3.2	Addition ADD	367
11.3.3	Subtraction SUB	367
11.3.4	Multiplication MUL	367
11.3.5	Division DIV	367
11.3.6	Division with remainder as result MOD	368
11.3.7	Generation of absolute value ABS	368
11.3.8	Negation NEG	369
11.3.9	Decrement DEC, increment INC	369
11.4	Arithmetic functions for time values	369
11.4.1	Introduction	369
11.4.2	Addition T_ADD	371
11.4.3	Subtraction T_SUB	371
11.4.4	Difference T_DIFF	371
11.4.5	Combine T_COMBINE	371
11.5	Mathematical functions	372
11.5.1	Introduction	372
11.5.2	Trigonometric functions SIN, COS, TAN	373
11.5.3	Arc functions ASIN, ACOS, ATAN	373
11.5.4	Formation of square SQR	374
11.5.5	Extraction of square root SQRT	374
11.5.6	Exponentiate to base e EXP	374
11.5.7	Calculation of Napierian logarithm LN	374
11.5.8	Extracting decimal places FRAC	375
11.5.9	Exponentiation to any base EXPT	375
11.6	Conversion functions (Conversion of data type)	376
11.6.1	Introduction	376
11.6.2	Conversion function CONV	377
11.6.3	Conversion functions for floating-point numbers	378
11.6.4	Conversion functions SCALE_X and NORM_X	381
11.6.5	Conversion function T_CONV	383
11.6.6	Conversion function S_CONV	383
11.6.7	Conversion functions STRG_VAL and VAL_STRG	385
11.6.8	Conversion functions STRG_TO_CHARS and CHARS_TO_STRG	387
11.6.9	Conversion functions ATH and HTA	389
11.7	Shift functions	389
11.7.1	Introduction	389
11.7.2	Shift to right (SHR)	389
11.7.3	Shift to left (SHL)	391
11.7.4	Rotate to right (ROR)	391
11.7.5	Rotate to left (ROL)	392
11.8	Logic functions	392
11.8.1	Introduction	392
11.8.2	Word logic operations (AND, OR, XOR)	392
11.8.3	Invert (INV)	394

---

11.8.4	Coding functions DECO and ENCO	394
11.8.5	Selection functions SEL, MUX, and DEMUX	395
11.8.6	Minimum selection MIN, Maximum selection MAX	397
11.8.7	Limiter LIMIT	398
11.9	Processing of strings (Data type STRING)	398
11.9.1	Output length of a string LEN	399
11.9.2	Combine strings CONCAT	400
11.9.3	Output left part of string LEFT	400
11.9.4	Output right part of string RIGHT	401
11.9.5	Output middle part of string MID	401
11.9.6	Delete part of a string DELETE	401
11.9.7	Insert string INSERT	402
11.9.8	Replace part of string REPLACE	403
11.9.9	Find part of string FIND	403
11.10	Calculating with the CALCULATE box in LAD and FBD	404
<b>12</b>	<b>Program flow control</b>	<b>406</b>
12.1	Jump functions	406
12.1.1	Overview	406
12.1.2	Absolute jump	407
12.1.3	Conditional jump	408
12.1.4	Jump list JMP_LIST	409
12.1.5	Jump distributor SWITCH	410
12.2	Block end function	412
12.3	Calling of code blocks	413
12.3.1	Introduction	413
12.3.2	Calling a function FC	413
12.3.3	Calling a function block (FB)	415
12.4	EN/ENO mechanism	417
12.4.1	EN/ENO mechanism with LAD and FBD	418
12.4.2	EN/ENO mechanism with SCL	418
12.4.3	EN/ENO for user blocks	419
<b>13</b>	<b>Online operation, diagnostics and debugging</b>	<b>420</b>
13.1	Connecting a programming device to the PLC station	421
13.1.1	IP addresses of the programming device	421
13.1.2	Connecting the programming device to the PLC station	422
13.1.3	Assigning an IP address to the CPU module	424
13.1.4	Switching on the online mode	424
13.2	Transferring project data	425
13.2.1	Loading project data for the first time	425
13.2.2	Delta downloading of project data	427
13.2.3	Error message following downloading	428
13.2.4	Working with the memory card	428
13.2.5	Processing blocks offline/online	431
13.2.6	Comparing blocks offline/online	432
13.2.7	Editing online project without offline project	433
13.2.8	Uploading project data from the CPU	434

13.3 Hardware diagnostics .....	436
13.3.1 Status displays on the modules .....	436
13.3.2 Diagnostics information .....	437
13.3.3 Diagnostics buffer .....	437
13.3.4 Diagnostics functions .....	439
13.3.5 Online tools .....	439
13.3.6 Further diagnostics information via the programming device .....	440
13.4 Testing the user program .....	441
13.4.1 Introduction to testing with program status .....	441
13.4.2 Program status with LAD and FBD .....	442
13.4.3 Program status in SCL .....	444
13.4.4 Monitoring with the PLC tag table .....	445
13.4.5 Monitoring of data tags .....	446
13.4.6 Testing with watch tables .....	447
13.4.7 Monitoring tags using watch tables .....	449
13.4.8 Modifying tags using watch tables .....	450
13.4.9 Enable peripheral outputs and “Modify now” .....	451
13.4.10 Forcing tags .....	452
<b>14 Distributed I/O .....</b>	<b>455</b>
14.1 Introduction, overview .....	455
14.2 PROFINET IO .....	456
14.2.1 PROFINET IO components .....	456
14.2.2 Addresses with PROFINET IO .....	457
14.2.3 Configuring PROFINET IO .....	459
14.2.4 Real-time communication with PROFINET IO .....	461
14.3 PROFIBUS DP .....	462
14.3.1 PROFIBUS DP components .....	462
14.3.2 Addresses with PROFIBUS DP .....	465
14.3.3 Configuring PROFIBUS DP .....	467
14.3.4 System functions for PROFINET IO and PROFIBUS DP .....	470
14.4 Actuator/sensor interface .....	473
14.4.1 Components of actuator/sensor interface .....	473
14.4.2 Configuring an AS-i master CM 1243-2 .....	475
14.4.3 Configuring an AS-Interface .....	476
14.4.4 Interface to user program .....	477
14.5 Communication via Modbus .....	477
14.5.1 Modbus RTU .....	477
14.5.2 Modbus TCP .....	480
<b>15 Communication .....</b>	<b>482</b>
15.1 Overview .....	482
15.2 Open user communication .....	484
15.2.1 Basics .....	484
15.2.2 Open user communication with TCP and ISO-on-TCP .....	485
15.2.3 Open user communication with the UDP protocol .....	487
15.2.4 Communication functions for open user communication .....	489
15.2.5 Configuring open user communication .....	493

---

15.2.6	Configuring a PN interface with T_CONFIG	495
15.3	S7 communication	496
15.3.1	Basics	496
15.3.2	Data structure for one-way data exchange	496
15.3.3	Communication functions for one-way data exchange	497
15.3.4	Configuring S7 communication	498
15.4	Point-to-point communication	499
15.4.1	Introduction to point-to-point communication	499
15.4.2	Configuring the CM 1241 communication module	500
15.4.3	Point-to-point communication functions	501
15.4.4	USS protocol for drives	504
<b>16</b>	<b>Visualization</b>	<b>507</b>
16.1	Introduction to visualization	507
16.1.1	Overview of HMI Panels in STEP 7 Basic	508
16.1.2	Creating a project with an HMI station	510
16.1.3	Cross-references for HMI objects	512
16.2	Creating HMI tags and area pointers	513
16.2.1	Introduction to HMI tags	513
16.2.2	Creating an HMI tag	514
16.2.3	Creating an area pointer	515
16.3	Configuring process screens	517
16.3.1	Introduction to configuring process screens	517
16.3.2	Working window for process screens	518
16.3.3	Working with screen layers	519
16.3.4	Working with templates	519
16.3.5	Working with function keys	520
16.3.6	Creating a new screen	521
16.3.7	Configuring a screen change	522
16.3.8	Working with objects in process screens	522
16.3.9	Changing screen objects during runtime	524
16.3.10	Basic objects for screen configuration	524
16.4	HMI functions	525
16.4.1	Input and display of process values	525
16.4.2	Working with alarms	528
16.4.3	Working with recipes	535
16.4.4	Working with the user administration	539
16.5	Completing HMI configuration	542
16.5.1	Compiling the HMI configuration (Consistency test)	542
16.5.2	Simulation of HMI configuration	542
16.5.3	Downloading configuration to the HMI station	543
16.5.4	Maintenance of the HMI station	546
<b>17</b>	<b>Appendix</b>	<b>548</b>
17.1	Integral and technological functions	548
17.1.1	High-speed counter (HSC)	548
17.1.2	Pulse generator	554
17.1.3	Technology objects for motion control	557

17.1.4 Technology objects for PID control .....	561
17.2 Telephone network connections with TeleService .....	564
17.3 Telecontrol with CP 1242-7 .....	565
17.4 Web server .....	567
17.4.1 Enable web server .....	567
17.4.2 Reading out web information .....	567
17.4.3 Standard web pages .....	567
17.5 Data logging .....	569
17.5.1 Introduction .....	569
17.5.2 Using data logging .....	569
17.5.3 Functions for data logging .....	570
<b>Index</b> .....	<b>572</b>

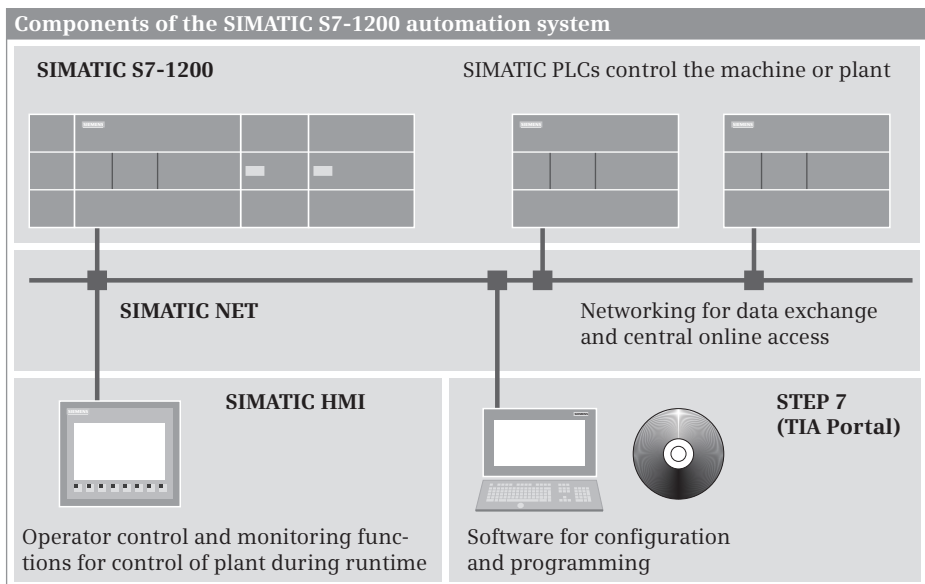
# 1 Introduction

## 1.1 Overview of the S7-1200 automation system

The SIMATIC S7-1200 automation system consists of the four controllers S7-1211C, S7-1212C, S7-1214C, and S7-1215C, which can exchange data with each other, with SIMATIC HMI Basic Panels, or with other programmable controllers over SIMATIC NET. STEP 7 (TIA Portal) is used to configure and program the devices (Fig. 1.1).

The **SIMATIC S7-1200 controllers** are programmable logic controllers (PLC) and constitute the basis of the automation system. Four different controllers with graded performances cover the low-end range of industrial controls.

**SIMATIC HMI** refers to the Human Machine Interface for operator control and monitoring. The Basic Panels are designed such that they interact optimally with SIMATIC S7-1200. The devices are available with display dimensions of 3.8, 5.7, 10.4 and 15 inches, and are operated using the touch screen. Except for the 15-inch device, they have additional function keys.



**Fig. 1.1** Components of the SIMATIC S7-1200 automation system

**SIMATIC NET** links all SIMATIC stations, and allows trouble-free data exchange. SIMATIC S7-1200 with PROFINET interface uses the Industrial Ethernet network to exchange data with other PLC stations, HMI stations, and programming devices. Communication modules expand the communication capabilities to other networks such as PROFIBUS DP, AS-Interface, or point-to-point coupling based on RS232 or RS485.

The **STEP 7** programming software provides the nesting function for Totally Integrated Automation (TIA), the automation system with uniform configuration and programming, data management, and data transfer. STEP 7 is used to configure and parameterize the SIMATIC components, and STEP 7 is also used to generate and debug the user program. The *TIA Portal* is the central user interface for management of the tools and automation data. STEP 7 in the TIA Portal is available in the versions STEP 7 Professional and STEP 7 Basic. Both versions can be used to configure and program an S7-1200 station. This book describes the use of STEP 7 Basic.

### 1.1.1 SIMATIC S7-1200

SIMATIC S7-1200 is the modular microsystem for the lower and medium performance range. The central processing unit (**CPU**) contains the operating system and the user program. The user program is located in the load memory and is power failure-proof. The parts of the user program relevant to execution are processed in a work memory with fast access. Tags whose values are to be retained in the event of a power failure or when switching off/on are stored in the retentive memory (Fig. 1.2).

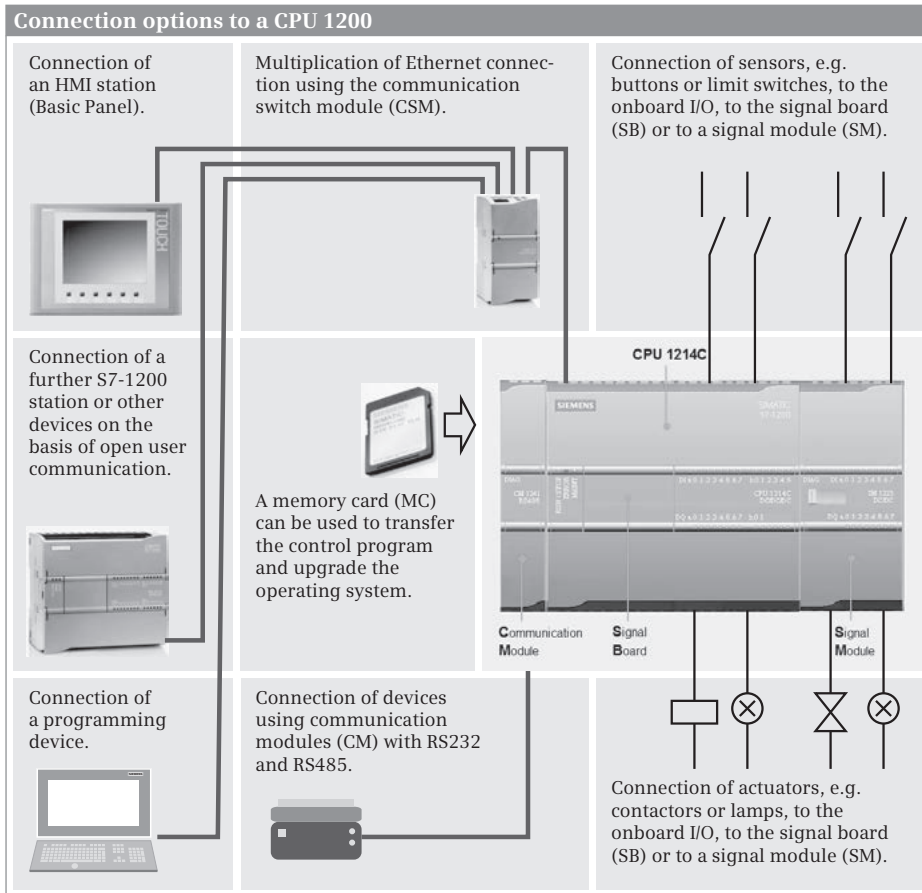
The user program can be transferred to the CPU using a plug-in memory card (**MC**) – as an alternative to transfer via an online connection to the programming device. The memory card can also be used as an external load memory or for updating the firmware.

The connections to the plant or process are made by **onboard inputs and outputs**, their number being determined by the CPU version. The onboard inputs and outputs are designed especially for operation of the integral high-speed counters (HSC). The operating system additionally includes pulse generators with a pulse-width modulated output and also the technology objects *Axis* for controlling stepper motors and servo motors with pulse interface and *PID Compact*, a PID controller with optimized self-tuning.

A signal board (SB) can be used to expand the onboard inputs and outputs. The communication board (CB) creates a point-to-point connection for the CPU and the battery board (BB) increases the power reserve of the integrated hardware clock to about one year.

If further inputs and outputs are required, signal modules (**SM**) can be plugged onto the CPU depending on its version. These are available for digital and analog signals.

The **PROFINET interface** connects the CPU to the Industrial Ethernet subnet. The programming device is connected to this interface if, for example, the user pro-



**Fig. 1.2** Connection options to a PLC station with CPU 1200

gram is to be transferred online to the CPU and tested on the machine. Data is exchanged with HMI stations and other automation devices via this interface.

If the CPU is only connected to one device over Ethernet, a standard or crossover cable can be used. If more than two devices that only have a PROFINET interface are networked, the connecting cables must be routed via a multiplier, e.g. the **communication switch module (CSM)**. A CPU 1215 has two ports connected with a switch so that they can be networked with the next programmable controller without an interposed connection multiplier.

Communication modules (**CM**) permit the operation on further bus systems such as PROFIBUS DP. Here, an S7-1200 station in a DP master system can be both DP master and DP slave. An S7-1200 station can be the AS-Interface master on AS-Interface and can control up to 62 AS-Interface field devices. The communication module for the point-to-point connection is available with RS232 or RS485 interface, to which, for example, a barcode or RFID reader can be connected.



### 1.1.2 Overview of STEP 7 Basic

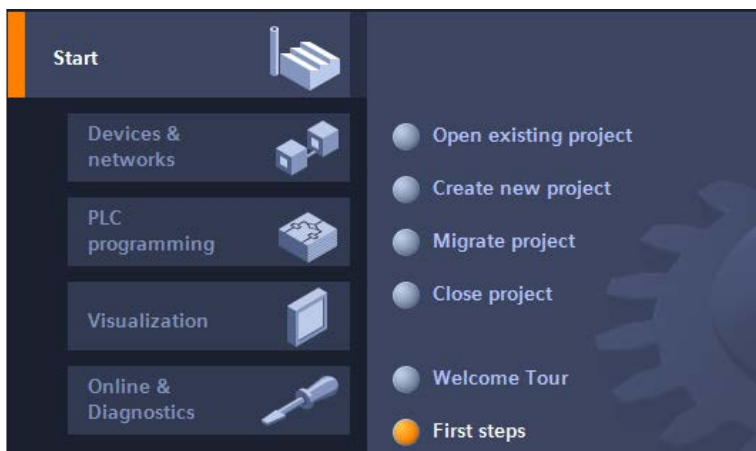
STEP 7 is the central automation tool for SIMATIC. STEP 7 requires authorization (licensing), and is executed on the current Microsoft Windows operating systems. STEP 7 Basic can be used to configure the S7-1200 controllers and – with WinCC Basic – the Basic Panels. Configuration is carried out in two views: the Portal view and the Project view.

The **Portal view** is task-oriented.

In the Start portal you can open an existing project, create a new project, or migrate an (HMI) project. A “project” is a data structure containing all the programs and data required for your automation task. The most important STEP 7 tools and functions can be accessed from here via further portals (Fig. 1.3):

- ▷ In the *Devices & networks* portal you configure the programmable controllers, i.e. you position the modules in a rack and assign them parameters.
- ▷ In the *PLC programming* portal you create the user program in the form of individual sections referred to as “blocks”.
- ▷ The *Visualization* portal provides the most important tools for configuration and simulation of Basic Panels.
- ▷ The *Online & Diagnostics* portal allows you to connect the programming device online to a CPU. You can control the CPU's operating modes, and transfer and test the user program.

The **Project view** is an object-oriented view with several windows whose contents change depending on the current activity. In the *Device configuration*, the focal point is the working area with the device to be configured. The Device view includes the rack and the modules which have already been positioned (Fig. 1.4). A further

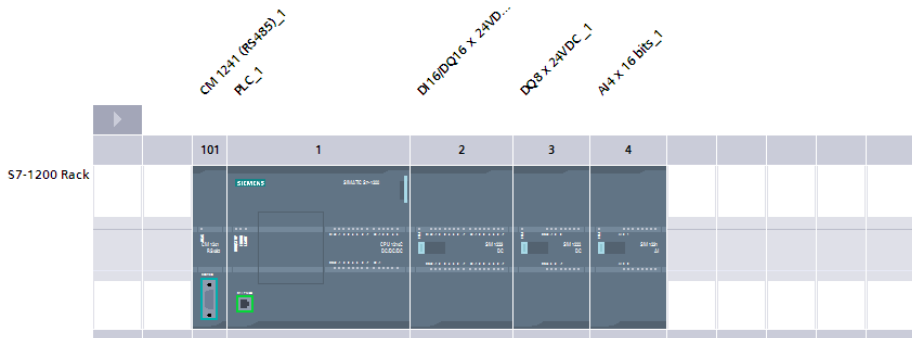


**Fig. 1.3** Tools in the Start portal of STEP 7 Basic

window – the inspector window – displays the properties of the selected module, and the task window provides support by means of the hardware catalog with the available modules. The Network view shows the networking between the devices and permits the configuration of communication connections.

When carrying out *PLC programming* you edit the selected block in the working area. You are again shown the properties of the selected object in the inspector window where you can adjust them. In this case, the task window contains the catalog of statements with the available program elements and functions. The same applies to the processing of PLC tags, to the online program test using watch tables, or to configuration of an HMI device.

And you always have a view of the *project tree*. This contains all objects of the STEP 7 project. You can therefore select an object at any time, for example a program block or watch table, and edit this object using the corresponding editors which start automatically when the object is opened.



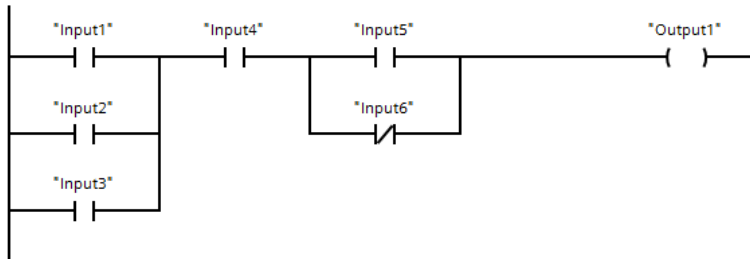
**Fig. 1.4** Example of working area of device configuration

### 1.1.3 Three programming languages

You can select between three programming languages for the user program: ladder logic (LAD), function block diagram (FBD), and structured control language (SCL). The user program can be structured into individual parts known as “blocks”. The programming language is a property of a block, which means you can use the programming language that is best suited to resolve the block function for every block in the user program.

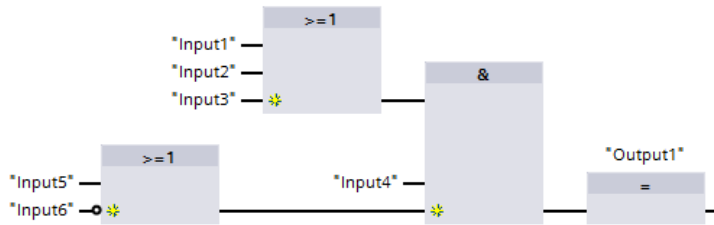
Using the **ladder logic**, you program the control task based on the circuit diagram. Operations on binary signal states are represented by serial or parallel arrangement of contacts (Fig. 1.5). A current path is terminated by a coil. Complex functions are represented by boxes which you handle like contacts or coils. Examples of boxes are mathematical functions or functions for processing strings.

Using the **function block diagram**, you program the control task based on electronic circuitry systems. Binary operations are implemented by linking AND and OR



**Fig. 1.5** Example of binary operations in ladder logic representation

functions and terminated by simple boxes (Fig. 1.6). Complex boxes are used to handle the operations on digital tags, for example with mathematical functions or functions for strings.



**Fig. 1.6** Example of binary operations in function block diagram representation

**Structured control language** is particularly suitable for programming complex algorithms or for tasks in the area of data management. The program is made up of SCL statements which, for example, can be value assignments, comparisons, or control statements (Fig. 1.7).

```

18 (*****)
19 Write_register:
20 IF #Level = #Register_length - 1
21     THEN #Full := TRUE;
22     ELSE #Register[#Write_pointer] := #Input_value;
23         #Level := #Level + 1;
24     IF #Write_pointer = #Register_length
25         THEN #Write_pointer := 0;
26         ELSE #Write_pointer := #Write_pointer + 1;
27     END_IF;
28     #Empty := FALSE;
29 END_IF; RETURN;

```

**Fig. 1.7** Example of SCL statements

### 1.1.4 Execution of the user program

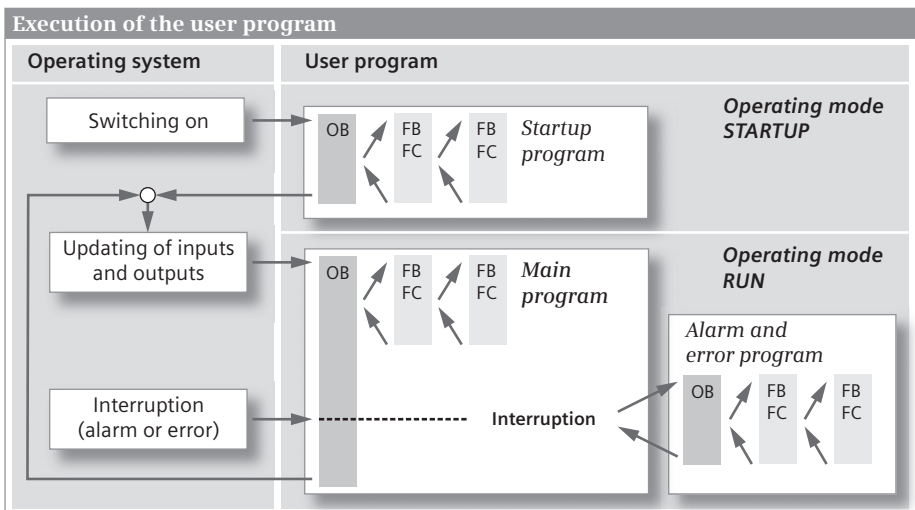
After the power supply has been switched on, the control processor checks the consistency of the hardware and parameterizes the modules. A startup program is then executed once, if present. The startup program belongs to the user program that you program. Settings and initialization operations for the user program can be present here.

The user program is usually divided into individual sections called “blocks”. The organization blocks (OB) represent the interface between operating system and user program. The operating system calls an organization block for specific events, and the user program is then processed in it (Fig. 1.8).

Function blocks (FB) and functions (FC) are available for structuring the program. Function blocks have a memory in which local tags are saved permanently, functions do not have this memory.

Program statements are available for calling function blocks and functions (start of execution). Each block call can be assigned inputs and outputs, referred to as “block parameters”. During calling, tags can be transferred with which the program in the block is to work. In this manner, a block can be repeatedly called with a certain function (e.g. addition of three tags) but with different parameters sets (e.g. for different calculations) (Fig. 1.9).

The data of the user program is saved in data blocks (DB). Instance data blocks have a fixed assignment to a call of a function block; they are the tag memory of the function block. Global data blocks contain data which is not assigned to any block.



**Fig. 1.8** Execution of the user program

Following a restart, the control processor updates the input and output signals in the process images and calls the organization block OB 1. The main program is present here. Structuring is also possible (and recommended) in the main program. Once the main program has been processed, the control processor returns to the operating system, retains (for example) communication with the programming device, updates the input and output signals, and then recommences with execution of the main program.

Cyclic program execution is a feature of programmable controllers. The user program is also executed if no actions are requested “from outside”, such as if the controlled machine is not running. This provides advantages when programming: For example, you program the ladder logic as if you were drawing a circuit diagram, or program the function block diagram as if you were connecting electronic components. Roughly speaking, a programmable logic controller has characteristics like those of a contactor or relay control: The many programmed operations are effective quasi simultaneously “in parallel”.

In addition to the cyclically executed main program it is possible to carry out interrupt-controlled program execution. You must enable the corresponding interrupt event for this. This can be a hardware interrupt, such as a request from the controlled machine for a fast response, or a cyclic interrupt, in other words an event which takes place at defined intervals.

The control processor interrupts execution of the main program when an event occurs, and calls the assigned interrupt program. You can assign organization blocks to certain events, and these blocks are then processed in such a case. Once the interrupt program has been executed, the control processor continues execution of the main program from the point of interruption.

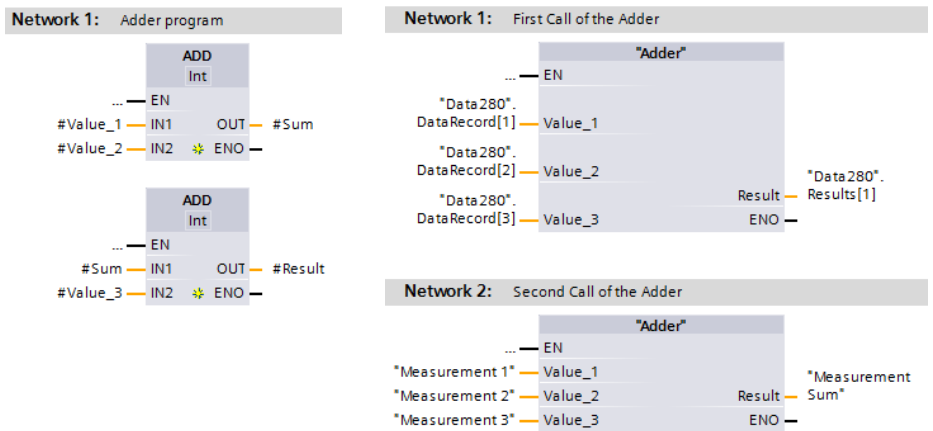


Fig. 1.9 Example of two block calls with different tags in each case

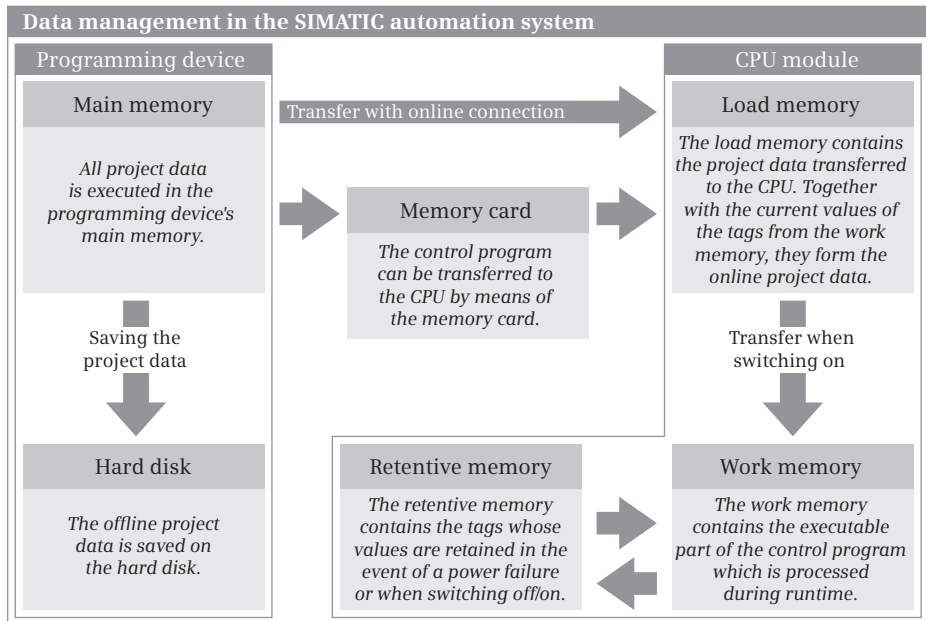
### 1.1.5 Data management in the SIMATIC automation system

The automation data is present in various memory locations in the automation system. Initially there is the programming device, referred to generally as the generation or engineering system. All automation data of a STEP 7 project is saved on its hard disk. Configuration and programming of the project data with STEP 7 is carried out in the main memory of the programming device (Fig. 1.10).

The automation data on the hard disk is also referred to as the *offline project data*. Once STEP 7 has appropriately compiled the automation data, this can be down-loaded to a programmable controller. The data downloaded into the user memory of the CPU module are known as the *online project data*.

The user memory on the CPU is divided into three components: The *load memory* contains the complete user program including the configuration data, the *work memory* contains the executable user program with the current control data, and the *retentive memory* contains the tags whose current values are saved power-failure-proof.

The memory card as a *transfer card* can transfer the user program to the CPU memory, or as a *program card* expand the CPU's internal load memory. When used as a program card, the memory card remains inserted in the CPU during runtime.



**Fig. 1.10** Data management in the SIMATIC automation system

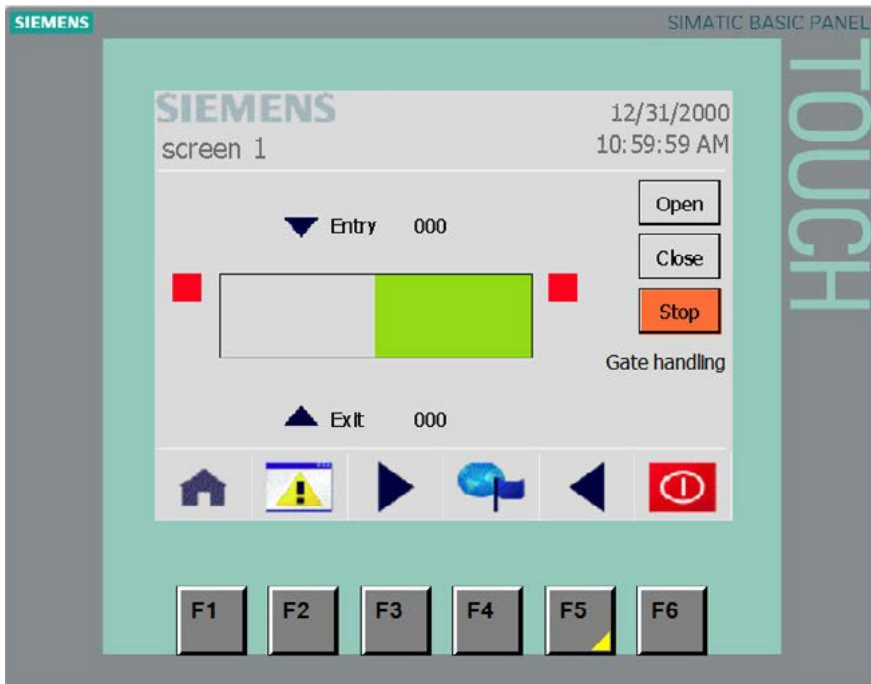
### 1.1.6 Operator control and monitoring with process images

Procedures in the process (on the controlled machine) are manually controlled and monitored using an HMI device. With the Basic Panels, a touch screen permits access using control elements represented on the monitor. Control and display elements are combined in process images. A process image can map a plant, display process sequences, output process values, or permit operator actions (Fig. 1.11).

The image sequence has a hierarchical structure. Commencing with a start screen which is displayed when the HMI device is switched on, it is possible to select the screens of the next level, from where the screens of the following level can be selected, and so on. Displays can be changed manually using key or touch inputs, or triggered by the user program.

Predefined objects are available for creating a screen, and can be inserted and adapted according to your requirements. These can be static objects such as text or graphics which do not change during process operation, or dynamic objects such as texts, numerical values, trends and bar charts which change depending on process values.

The functional scope of the Basic Panels also includes message control with bit and analog messages, management of recipes, and user administration.



**Fig. 1.11** Example of a process image in the configuration stage

## 1.2 Introduction to STEP 7 Basic for S7-1200

### 1.2.1 Installing STEP 7

STEP 7 Basic V11 is a 32-bit application, which executes with MS Windows XP (Home with SP3 or Professional with SP3) or MS Windows 7 (Home Premium, Professional, Enterprise, or Ultimate, 32/64-bit). You require administration rights in order to install STEP 7, and to work with STEP 7 you must at least be logged-on as a main user.



The processor should at least be a Pentium 4 with 1.7 GHz or a comparable type. A main memory of 1 GB is required for working with Windows XP, and should be 2 GB for Windows Vista. STEP 7 Basic requires approx. 2 GB on the hard disk.

An Ethernet interface (LAN adapter) is required on the programming device for the online connection to a programmable controller or Basic Panel. If you wish to work with a SIMATIC memory card, you require an SD card reader.

Installation is carried out using the setup program *start.exe* on the DVD. Deinstallation of STEP 7 Basic is carried out as usual in MS Windows using the *Software* program in the Windows Control Panel. Parallel online use of STEP 7 Basic and STEP 7 has not been enabled.

### 1.2.2 Automation License Manager

You require a license (user authorization) in order to use STEP 7. Licenses are managed by the Automation License Manager which is installed together with STEP 7 Basic. The license key for STEP 7 Basic is transferred to the hard disk during the installation, and removed again from the hard disk during deinstallation.



The license key is stored on the hard disk in specially identified blocks. To avoid unintentional destruction of the license key, you should observe the information for handling license keys in the Help text of the Automation License Manager.

The Automation License Manager also manages the license keys of other SIMATIC products, e.g. STEP 7 V5.4 and WinCC.



### 1.2.3 Starting STEP 7 Basic

You start STEP 7 Basic either using the Start button of Windows and *Programs > Siemens Automation > TIA Portal V11*, or by double-clicking the icon on the Windows desktop. The *Totally Integrated Automation Portal* is the user interface for STEP 7 and may also contain other applications which use the same database. For example, STEP 7 Basic V11 includes the WinCC Basic configuration software which is displayed as an integrated “Visualization” editor in the Start portal.



### 1.2.4 Portal view

Following starting-up, STEP 7 Basic displays the Start portal. A *portal* makes available all functions and tools required for the respective range of tasks in the *portal view*. The scope of the portals as well as the range of functions and tools depends on the installed applications. The *Start portal* of STEP 7 Basic permit selection of the following portals (Fig. 1.12):

- ▷ In the *Devices & networks* portal you can configure the hardware of the programmable controller, i.e. you select the hardware components, position them, and set their properties. If several devices are networked, you can define the connections here.
- ▷ The *PLC programming* portal contains all the tools required for generating the user program for a PLC station (programmable logic controller).

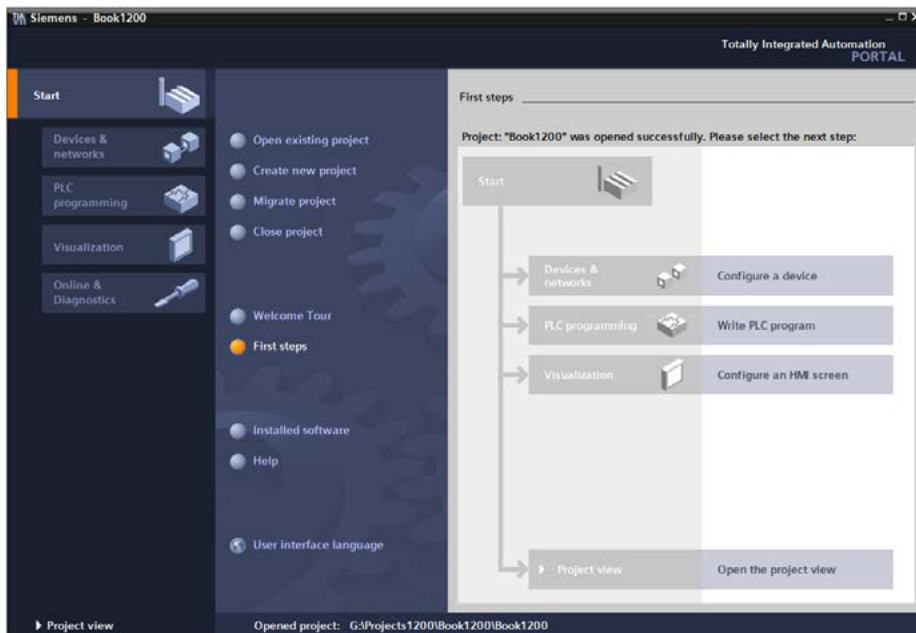


Fig. 1.12 Portal view: First steps after opening a project

- ▷ In the *Visualization* portal you generate the operator control and monitoring desktop for HMI stations. Here you can configure, for example, the process images, the control elements, and messages.
- ▷ Using the *Online & Diagnostics* portal you can connect the programming device to a programmable controller, transfer and debug programs, and detect faults in the automation system.

Additional functions included in the Start portal allow you to *Create new project* or *Open existing project*. *First steps* informs you of what possibilities you have to continue the configuration after creating a project. *Installed products* provides an overview of further SIMATIC applications currently on the computer. You can call *Help* in every portal.

### 1.2.5 Information system

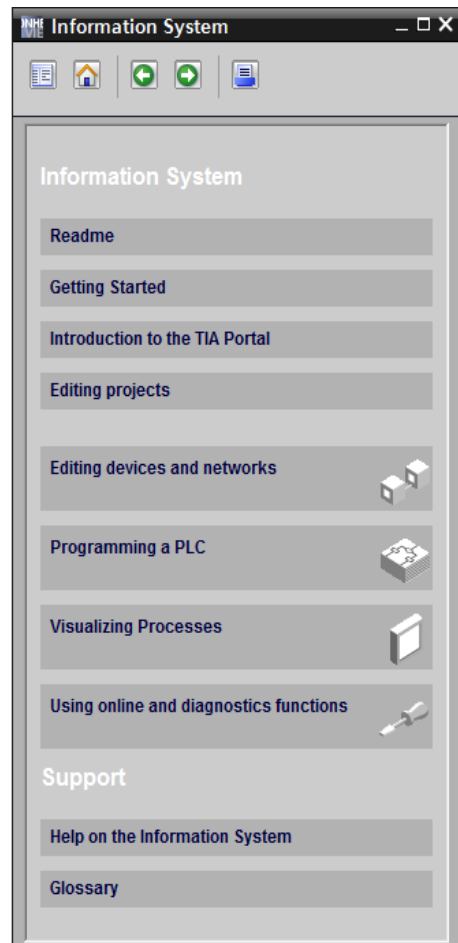
The help function of STEP 7 provides you when programming with comprehensive support for solving your automation task.

To call the help function, click on *Help* in the Portal view or select the *Help > Display help* command in the main menu in the Project view. A window appears with the Siemens Information System (Fig. 1.13).

The online help is roughly divided according to the project processing steps: Configuration, parameterization, and networking of devices, structuring and programming of the user program, visualization of processes, and utilization of the online and diagnostics functions.

*Readme* provides general information on STEP 7 and further information which could no longer be included in the online help.

A comprehensive description of all available statements, including extended statements, can be found under *PLC programming > References*.



**Fig. 1.13** Information system of STEP 7 Basic

### 1.2.6 The windows of the project view

The project view shows all elements of a project in structured form in various processing windows. You can move from the Portal view to the Project view using the *Project view* link at the bottom left of the screen, or STEP 7 automatically switches to the Project view depending on the selected tool.

Fig. 1.14 shows the windows of the Project view in an example of block programming. Different window contents are displayed depending on the currently used editor.

#### ① Main menu and toolbar, shortcut menu

Underneath the title bar is the *main menu* with all menu commands. The menu commands available for selection depend on the currently marked object; menu commands which cannot be selected are displayed in gray. The same functionality is available – somewhat user-friendlier – with the *shortcut menu*: if you click on an object with the right mouse button, a window is opened with the currently selectable menu commands. Underneath the main menu is the *toolbar* with the graphically represented “main functions”. The main menu and the toolbar are always present in all editors.

Using *Options > Customize* in the main menu you can adapt the user interface. For example, under “General” you can define the interface language in which STEP 7 is used, and the mnemonics (the representation of the operands: “I” for input (international), or “E” in German).

#### ② Working window

In the center of the screen is the working window. The contents of the working window depend on the editor currently being used. In the case of device configuration, the working window is divided in two: the objects (modules and stations) are displayed in graphic form in the top part, and in tabular form in the bottom part. When programming the PLC, the top part of the working window contains the interface description of the block, and the bottom part the program represented in LAD, FBD, or SCL. You use the working window to configure the hardware of the automation system, generate the user program, or configure the process screens for HMI devices.

You can separate the working window completely from the project view so that it is displayed as a separate window (“Release” icon in the title bar of the working window), and also insert it again (“Embed” icon). The “Maximize” icon closes all other windows and displays the working window in maximum size.

#### ③ Inspector window

The inspector window underneath the working window shows the properties of the objects marked in the latter, records the sequence of actions, and provides an overview of the diagnostics status of the connected devices.

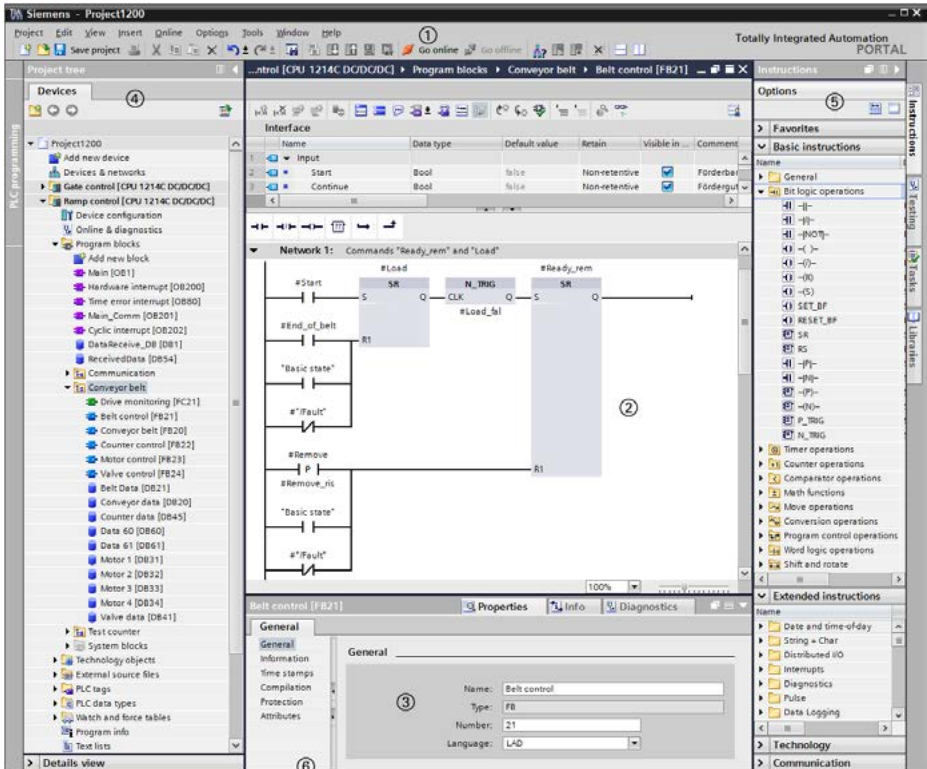


Fig. 1.14 Components of Project view using example of block programming

During configuration or programming you set the object properties in the inspector window, for example the addresses and symbol names of inputs and outputs, the properties of the PROFINET interface, tag data types, or block attributes.

#### ④ Project tree

The project tree window is displayed with the same content for all editors. Its hierarchical structure contains all project data and the required editors. With the project open, it shows the folders for the PLC and HMI stations included in the project, and further subfolders within these folders, e.g. for program blocks, PLC tags and watch tables with a PLC station or e.g. the process images and the HMI tags in the case of an HMI station.

A double-click on an object with project data automatically starts the associated editor. The project tree also includes editors such as “Add new device”, “Device configuration” or “Online & diagnostics” which you can start directly using a double-click.

The lower section of the project tree contains a details view of those objects which are present in the hierarchy underneath the object marked in the project tree.

### ⑤ Task window

To the right of the working window is the task window with the task cards. This contains further objects for processing in the working window. The contents of the task window depend on the currently active editor. In the case of the hardware configuration, for example the hardware catalog with the available components is shown here, in the case of PLC programming the program elements catalog appears, with online & diagnostics the online tools, and with the visualization the library for the process image control and display elements.

You can also call the libraries in this window: global libraries supplied with STEP 7, or the project library in which you can save reusable objects such as program blocks, templates for process images, or control elements with special configurations.

### ⑥ Editor and status bar

At the bottom left of the Project view you can change to the Portal view. In the middle you can see the tabs of the open windows. Clicking on a tab results in its contents being displayed in the top level of the working window. This makes it easy to change quickly between different window contents. The far right of the status bar indicates the current status of project processing.

## 1.2.7 Adapting the user interface

The language of the user interface can be changed. In the main menu select *Options > Settings* and the “General” section. In the “Interface language” drop-down list you can select the desired language from the installed languages. The texts of the user interface are then immediately displayed in the new language. You can also define here how the TIA Portal is to be displayed following the next restart.

You can show or hide the displayed window using the menu item *View*. You can always change the size of windows by dragging on the edge with the mouse. Windows can be minimized into a symbol which appears in one of the navigation bars in the left, bottom or right margin of the screen.

You can maximize the working window, or release it from the windows group and display it as a separate window. The working window can be divided vertically or horizontally, permitting you to view two working areas simultaneously.

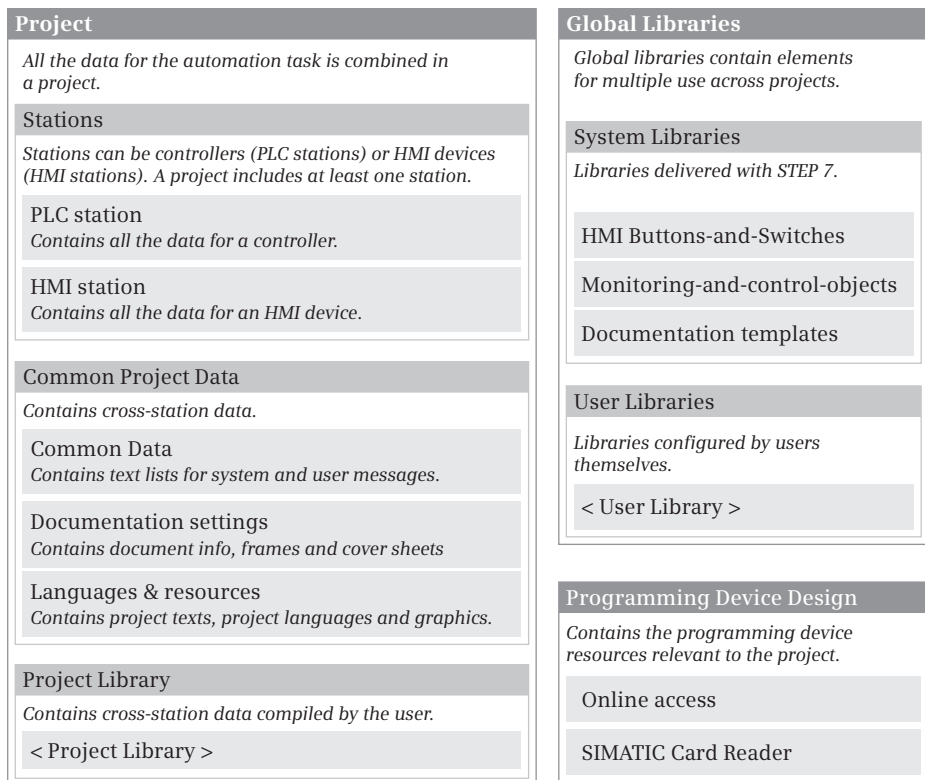
You can change the width of table columns by dragging with the cursor in the table header. In the case of columns that are too narrow, the entire content of the individual cells will appear as a tooltip when the cursor is briefly hovered over the relevant field.

## 1.3 Editing a SIMATIC project

Fig. 1.15 shows all tools and data which can be of importance in an automation task. Of prime importance is the *Project* which contains all the automation data required for control and operation of the machine or plant. The project data is roughly divided into the data for the individual stations and the common project data which applies to all stations in the project.

A *station* can be a controller (PLC station) or an HMI device (HMI station). A project can include several stations, but at least one station must be present. The data present in a station is described further below. *Common Project Data* includes, for example, centrally managed message texts or texts for multilingual projects.

A *project library* is created for each project. Objects which are used in several projects are combined in *global libraries*. Also relevant to a project is the *programming device design* with interface modules (LAN adapters) and SD card readers.



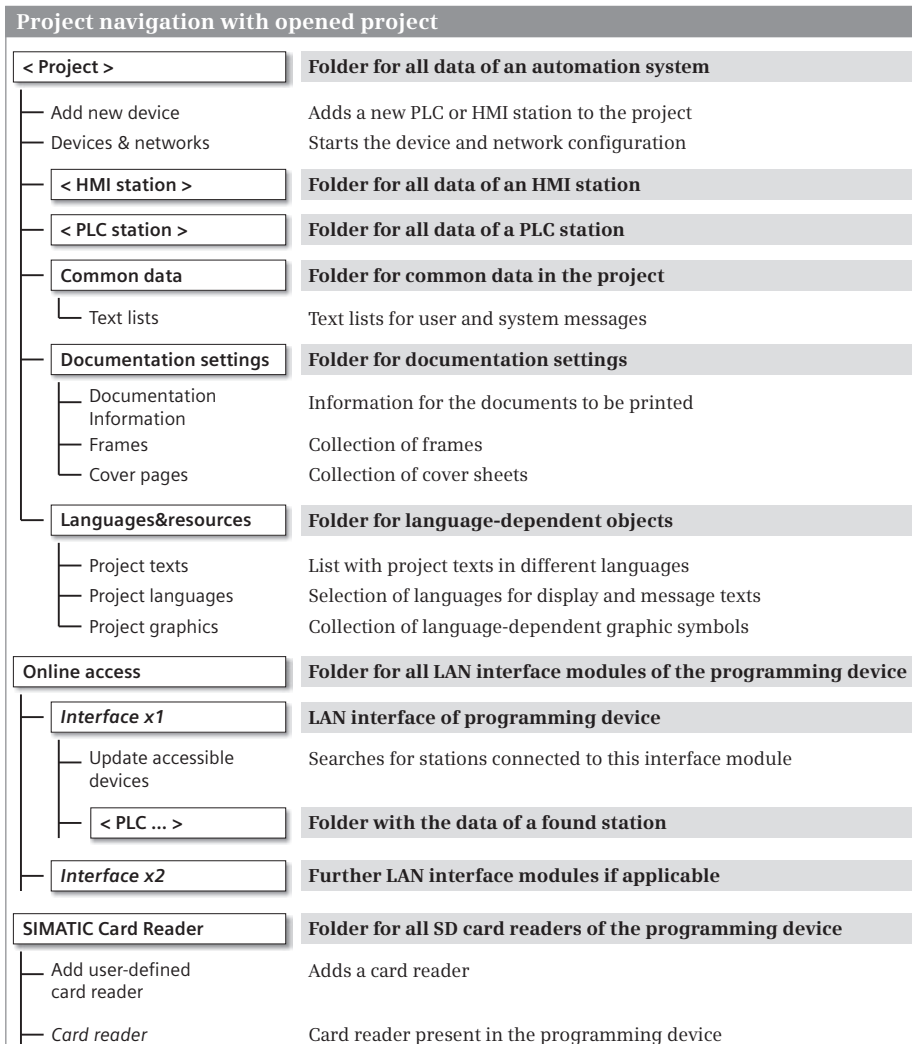
**Fig. 1.15** Project components, libraries and programming device design

### 1.3.1 Structured representation of project data

The project tree in the Project view displays the project data and the programming device configuration in a tree structure (Fig. 1.16).

The structure also includes the editors (tools) required for generating and editing the data. The project tree does not include the project library. This is represented in a task card together with the global libraries in the task window under “Libraries”.

You can replace the names shown in angle brackets by names more appropriate to your automation task.



**Fig. 1.16** Project structure in the project tree

### 1.3.2 Project data and editors for a PLC station

If you add a PLC station (an S7-1200 controller) to the project, STEP 7 creates the corresponding structure in the project data (Fig. 1.17). A PLC or HMI station is always required for editing in a project so that STEP 7 can create the data structures required for the PLC programming or HMI configuration. If you wish to write a user program without previously selecting a specific CPU, you can select an “unspecified CPU” from the hardware catalog and replace it later with a “real” CPU 1200 if necessary.

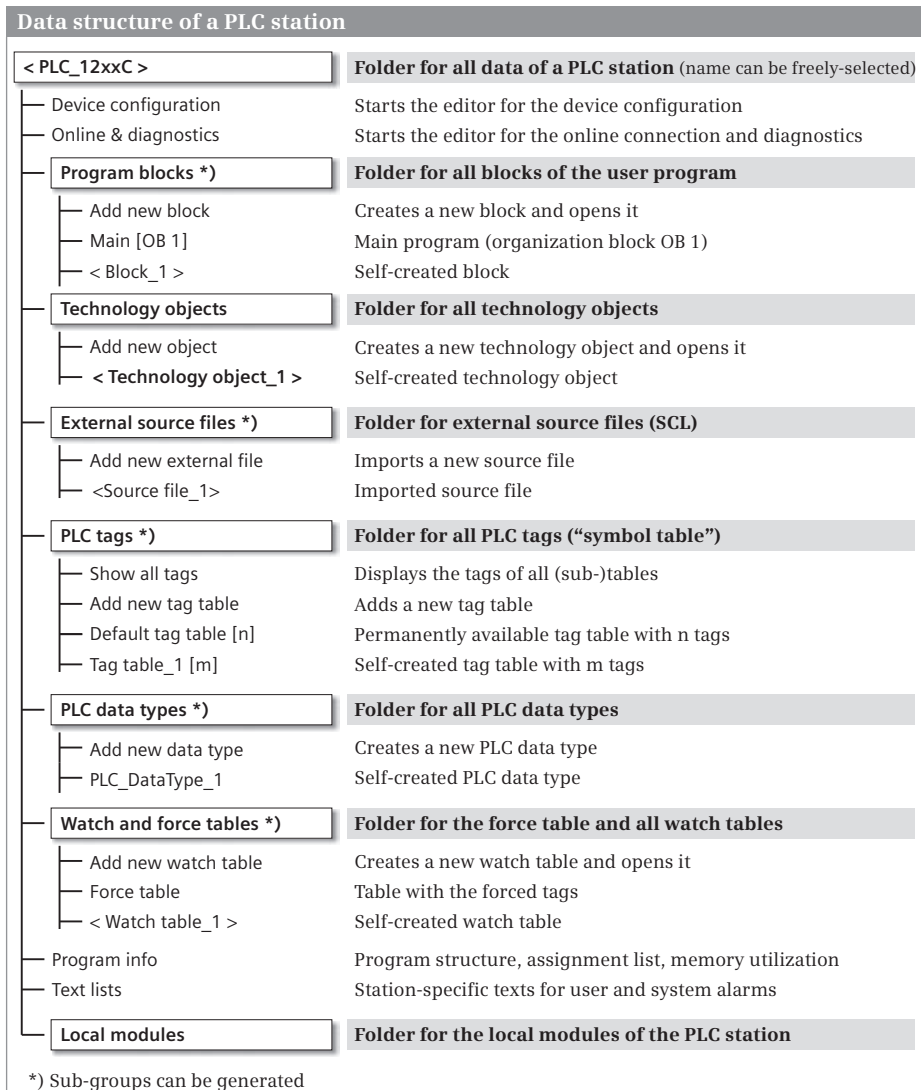


Fig. 1.17 Structure of the project data for a PLC station



The user program which controls the machine or process is located in the *Program blocks* folder. The program comprises *blocks* (separate program components) which are either stored directly in the *Program blocks* folder or – if there is a large number – in subfolders which you can create and configure yourself. The “Main” block (the name is the symbol for the block and can be changed) is the organization block OB 1 and is created automatically. The processing sequence of the blocks is defined in the user program by “block calls” and is displayed by opening the *Program information* object under *Call structure*.

The *Technology objects* folder contains the configuration data for the motion control objects (*axes*) and control loop objects (*PID controllers*).

SCL source files can be saved in the *External sources files* folder.

The *PLC tags* folder contains the assignment of the absolute address to the symbolic address of global tags that are valid throughout the PLC station (inputs, outputs, and bit memories).

The *PLC data types* folder contains user-defined, structured data types.

The force table and all created watch tables can be found in the *Watch and force tables* folder. A watch table is used during testing of the user program. It contains tags whose current value can be monitored and also changed during runtime. The force tables can be used to set tags to fixed values that cannot be changed by the program.

The *program information* shows the program structure (sequence and dependencies of the block calls), the assignment list for inputs, outputs and bit memories, and the memory usage.

Message texts are stored under *Text lists*. In the case of the user-defined text list, you can specify the value ranges which trigger the messages and the associated texts; with a system-defined text list, the contents are specified by STEP 7. Text lists created under a PLC station contain station-specific texts, those created under a project contain cross-station texts.

The *Local modules* folder contains all configured modules of the PLC station. Opening a module initiates device configuration. The module properties are displayed in the inspector window.

You start configuration of a station using the *Device configuration* editor which is located in the first position in the project structure. There is no corresponding folder for the data of the device configuration in the project tree. The configuration data is located “behind” the *Device configuration* editor.

*Online & diagnostics* starts the editor for the online connection and online functions. For example, you can use a (software) control panel to control the operating modes of the CPU, to set the CPU's IP address and time, or read the CPU's diagnostics buffer.

### 1.3.3 Creating and editing a project

#### Creating a new project

You can create a new project in the Portal view if you click in the Start portal on *Create new project*. Assign a name to the project and set a path in which the project is to be saved. After clicking the *Create* button, any project which is open is closed, the new project is created, and the next steps displayed in the Start portal for selection:

- ▷ Configure a device  
STEP 7 changes to the *Devices & networks* portal in which you can insert a new CPU 1200 (a PLC station) into the project and open it for editing.
- ▷ Write PLC program  
STEP 7 changes to the *PLC programming* portal in which you can edit the “Main” block (organization block OB 1) or insert a new block and open it for editing. A PLC station must first have been added to the project.
- ▷ Configure an HMI screen  
STEP 7 changes to the *Visualization* portal in which you can create a new HMI station or configure an already existing one. From this portal you start configuration of the process images, editing of HMI tags and messages, and the simulator.
- ▷ Open the project view  
STEP 7 changes to the Project view in which you can carry out the next steps (insert and configure PLC stations, insert and program blocks, or insert and configure HMI stations).

In the project tree you can create a new project using the *Project > New* menu command. Assign a name to the project in the dialog window, set the path in which the project is to be saved, and click on the “Create” button.

#### Editing an existing project

You can open an existing project in either the Portal view or the Project view. Either activate *Open existing project* in the Portal view or *Project > Open* in the Project view. Select the desired project from the list of projects last used. Any project which is open is closed, and the selected project is opened.

During editing in the Project view, you can save the entered project data using the *Project > Save* or *Project > Save as* menu command. You can close the project using *Project > Close* – following confirmation of whether changes are to be saved – without exiting STEP 7.

You can delete a (closed) project from the hard disk – following confirmation – using *Project > Delete*.

#### Compiling and downloading project data

Before project data can be downloaded to a station, it must be made readable for the processor: it must be “compiled”. The project data is compiled station-by-station.

The scope of the compilation can be varied depending on the type of station. For example, the command from the *Compile > Software* pop-up menu only compiles those software components which have been changed since the last compilation.

### Printing project data

The project data can be printed in the form of a circuit manual. You can use the documentation function to set the layout of the printout. Under *Document settings* in the project tree, you adapt the existing template or create a new template, for example in the sizes A3 or A4 in portrait or landscape format. You complete the text fields under *Document information* (e.g. the title block). You can also position new text fields or graphic symbols in the layout.

By means of the print preview or print output, you can specify the output with or without cover page and as compact (abbreviated) or complete.

### Converting a project from STEP 7 V10.5 to STEP V11

To convert a project created with STEP 7 V10.5, start STEP 7 V11 and select *Project > Open* from the main menu. In the dialog window, click on the *Browse* button and select the desired project from the file directory. In the open project folder, select the file *<Project\_name>.ap10* and click the *Open* button. Confirm the conversion messages with *OK*. The project is converted to Version V11, saved under the name *<Project\_name>\_<STEP7-Version>*, and opened. You then compile every station and clear any displayed faults and warnings.

#### 1.3.4 Creating and editing libraries

Libraries are used to save reusable program components. These could include stations, blocks, PLC tag tables, process images or picture elements, for example. A project library and global libraries are available. The libraries are displayed in a task card of the task window.

A *project library* which you can fill with objects is automatically created when you create a project. You can structure the contents of the project library using folders. A project library is always opened, saved, and closed together with the project.

Components which can be used in multiple projects are saved in *global libraries*. There are global system libraries which are supplied with STEP 7, and global user libraries which you create yourself. A global library is opened, saved, and closed independent of the project. If you wish to use a global library simultaneously with other users, the library must be opened as read-only.

## 2 SIMATIC S7-1200 automation system

### 2.1 S7-1200 station components



**Fig. 2.1** S7-1200 station with CPU S7-1214, two SMs (right) and one CM (left)

A complete programmable controller including all I/O modules is referred to as a “station”. This also includes distributed I/O modules connected to the CPU via a bus system. An S7-1200 station comprises at least the CPU. Depending on the version, it has digital and analog input/output channels and can be fitted with additional input/output channels using a signal board (SB).

Depending on the type of CPU, up to eight signal modules can be plugged in which expand the station by digital and analog input/output channels. A two-tier design is possible using a 2-meter long extension cable.

The programming device is connected over Industrial Ethernet. Industrial Ethernet can also be used to connect further SIMATIC stations or HMI devices to the CPU. Up to three communication modules take over the connection to additional bus systems or to a point-to-point link.

Integral *technological functions* for measuring and counting tasks, closed-loop control, and motion control allow the CPU 1200 to be used in many complex machine controls.

The *SIMATIC Memory Card* can be used to download configuration data, as an external load memory, or for a firmware update.

Mounting is on a standard 35 mm DIN *rail* either horizontally or vertically. Installation without a mounting rail is also possible. An extension cable (2 m long) enables a two-tier design without changing the number of connectable signal modules.

Available accessories include an external *power supply module*, a *connection multiplier* (Ethernet switch), a *TS Adapter*, and two *simulator modules*. SIMATIC S7-1200 is also available as a *SIPLUS* version for particularly harsh environmental conditions.

## 2.2 S7-1200 CPU modules

There are four types of CPU (CPU 1211, CPU 1212, CPU 1214, and CPU 1215), available in each case in the versions DC/DC/DC, AC/DC/RLY, DC/DC/RLY. The first item of data refers to the module power supply (DC = 24 V direct current, AC = 120/230 V alternating current). The middle item of data refers to the operating voltage of the onboard digital inputs (DC = 24 V direct current). The last item of data refers to the type of digital outputs (DC = 24 V direct current electronic, RLY = up to 30 V direct current or up to 250 V alternating current with relay).



Fig. 2.2 CPU 1214C DC/DC/DC

The four CPU versions mainly differ in the supply voltage, the number of onboard inputs and outputs, the memory size, and the expansion capability with signal modules (Table 2.1).

The CPU 1215C can be integrated in STEP 7 Basic V11 with a hardware support package (HSP).

### 2.2.1 Integrated I/O

The *digital inputs* (DI) on the CPU module work with an operating voltage of 24 V DC. Different numbers are available depending on the CPU version. The status of the input signals is displayed by means of LEDs.

The *digital outputs* (DQ or DO) are available in electronic form (24 V DC and 0.5 A output current with a resistive load of 5 W) and as relay outputs (up to 30 V DC and 2 A output current with a resistive load of 30 W or up to 250 V AC and 2 A output current with a resistive load of 200 W). Different numbers of digital outputs are available depending on the CPU version. The status of the output signals is displayed by means of LEDs.

**Table 2.1** Selected data of a CPU 1200 with Firmware V3.0

	CPU 1211C	CPU 1212C	CPU 1214C	CPU 1215C
<b>User memory</b>				
Internal load memory *)	1 MB	1 MB	4 MB	4 MB
RAM	30 KB	50 KB	75 KB	100 KB
Retentive memory	10 KB	10 KB	10 KB	10 KB
<b>Integrated I/Os</b>				
Digital inputs (DI)	6 DI, 24 V DC	8 DI, 24 V DC	14 DI, 24 V DC	14 DI, 24 V DC
Digital outputs (DO)	4 DO, 24 V DC or relay	6 DO, 24 V DC or relay	10 DO, 24 V DC or relay	10 DO, 24 V DC or relay
Analog inputs (AI)	2 AI (10 bit)	2 AI (10 bit)	2 AI (10 bit)	2 AI (10 bit)
Analog outputs (AO)	–	–	–	2 AO (10 bit)
<b>Process images</b>	1024 bytes inputs, 1024 bytes outputs			
<b>Bit memory</b>	4096 bytes		8192 bytes	
<b>Expansion with</b>				
A board (SB, CB, BB)	1	1	1	1
Signal modules (SM)	None	2	8	8
Communication modules (CM)	3	3	3	3
<b>High-speed counter</b>				
as single-phase counter				
with integrated I/O	3 with 100 kHz	3 with 100 kHz	3 with 100 kHz	3 with 100 kHz
with standard SB	2 with 30 kHz	1 with 30 kHz	3 with 30 kHz	3 with 30 kHz
with high-speed SB	2 with 200 kHz	2 with 30 kHz	–	–
as A/B counter				
with integrated I/O	3 with 80 kHz	2 with 200 kHz	–	–
with standard SB	3 with 80 kHz	3 with 80 kHz	3 with 80 kHz	3 with 80 kHz
with high-speed SB	2 with 20 kHz	1 with 20 kHz	3 with 20 kHz	3 with 20 kHz
with high-speed SB	2 with 160 kHz	2 with 20 kHz	–	–
with high-speed SB	2 with 160 kHz	2 with 160 kHz	–	–
<b>Pulse generators</b>	4			
with integrated I/O	100 kHz			
with standard SB	20 kHz			
with high-speed SB	200 kHz			
Pulse outputs **)	4	4	4	4
Inputs for pulse catch	6	8	14	14
Real-time clock buffer	Typically 10 days, can be increased up to one year with a battery board			
PROFINET connection	1			2 with switch
<b>Temporary local data / nesting depth</b>				
Startup + main program	16 KB / 16 levels			
Interrupt/error processing	4 KB / 4 levels			
<b>Execution time</b>				
for binary functions	0.085 µs/instruction			
for digital functions	1.7 µs/instruction			
for floating-point functions	2.3 µs/instruction			

\*) Expandable up to SD card size

\*\*) A digital signal board is required for CPU versions with relay outputs

Each CPU has two *analog input channels* (AI) for 0 to 10 V. The resolution is 10 bits. The analog value can be processed in the user program in the numerical range from 0 to 27 648. The CPU 1215 has two additional analog output channels (AO) for 0 to 20 mA. The resolution is 10 bits. The analog value can be processed in the user program in the numerical range from 0 to 27 648.

Further details can be found in Section 2.3.3 “Properties of the I/O connections” on page 50.

The terminal blocks for the inputs and outputs can be removed from all modules without having to disconnect the wiring.

The CPU module does not have a mode selector for switching on/off. The operating modes (RUN, STOP) are set online using the programming device.

### 2.2.2 PROFINET connection

The CPU is connected to an Ethernet network over the PROFINET interface. The connection (port) takes the form of an RJ45 socket. The protocols Transmission Control Protocol (TCP) in accordance with RFC 793, ISO Transport over TCP (ISO-on-TCP) in accordance with RFC 1006, and User Datagram Protocol (UDP) in accordance with RFC 768 are supported. The connection is able to automatically recognize a transmission rate of 10 or 100 MBit/s (autosensing). Either a standard Ethernet cable or a crossover cable can be used for the network.

The CPU can be connected, for example, to a programming device, an HMI device, or other SIMATIC stations over the PROFINET connection. The CPU 1215C has two RJ45 sockets, which are connected with a switch. The next device can therefore be connected directly to the Ethernet network from the second connection. The other CPUs have only one RJ45 socket. Here an external switch (connection multiplier) such as the CSM 1277 Compact Switch Module is required when networking several devices.

As of firmware version 2.0, a PROFINET IO controller is integrated in the CPU operating system. The CPU can thus control distributed I/O via Industrial Ethernet with PROFINET IO. More detailed information can be found in Chapter 14.2 “PROFINET IO” on page 456.



**Fig. 2.3**  
PROFINET connection

### 2.2.3 Status LEDs

The current operating mode of the CPU is indicated by LEDs on the front of the module:

RUN/STOP	Continuous yellow light in STOP mode
	Continuous green light in RUN mode
	Flashing light in STARTUP mode
ERROR	Flashing red light in event of error
	Continuous red light if hardware is faulty
MAINT	Continuous yellow light with maintenance request

After switching on, the CPU is in STARTUP mode. It runs through test routines, carries out parameter settings, and executes the startup program. The CPU then changes to the RUN status and executes the user (main) program – this is the “normal” operating status. The CPU returns to the STOP mode if it detects a “serious” error, if it executes a corresponding program statement, or if it is specifically set to this state e.g. by the programming device. The user program is not executed in the STOP mode, but the CPU is still able to communicate, facilitating downloading of parts of the user program, for example.

The ERROR LED flashes when an error has been detected. It lights up permanently if the hardware is faulty. The MAINT LED lights up continuously to indicate that a previously configured maintenance request is now present. All LEDs flash if the firmware of the CPU module is faulty.



**Fig. 2.4** LED displays on the CPU

### 2.2.4 SIMATIC Memory Card

The SIMATIC Memory Card can be used as a program card, a card for transferring data, or as a data medium for firmware updates. As a program card it is required for runtime operation of the CPU, in the other cases it is not required for operation.

There are two versions of the SIMATIC Memory Card: with a storage capacity of 2 MB or 24 MB. It has a special ID which is necessary for use in a CPU 1200.



**Fig. 2.5** SIMATIC Memory Card

### 2.2.5 Expansions of the CPU

#### Signal board (SB)

A signal board (SB) expands the onboard I/O without changing the dimensions of the CPU. The associated slot is located on the front of the CPU.

Signal boards are available with 24 V and 5 V digital inputs and outputs, which can be operated at a frequency of up to 200 kHz (Table 2.2). The counting frequency of



the high-speed counters can thus be increased. The analog inputs allow the measurement of a voltage or current with a resolution of 11 bits + sign. With a Pt 100/200/500/1000 (RTD) resistance temperature sensor or with Type J and K thermocouples (TC), temperatures can be measured with a resolution of 15 bits + sign. The list of signal boards is rounded off by one with an analog output channel (12 bit,  $\pm 10$  V or 0 to 20 mA).



**Fig. 2.6** Signal Board 1223

**Table 2.2** Selection of signal boards

Digital inputs SB 1221 DC SB 1221 DC	6ES7 221-3AD30 6ES7 221-3BD30	DI 4 × 5 V DC, 200 kHz DI 4 × 24 V DC, 200 kHz
Digital outputs SB 1222 DC SB 1222 DC	6ES7 222-1AD30 6ES7 222-1BD30	DO 4 × 5 V DC / 0.1 A, 200 kHz DO 4 × 24 V DC / 0.1 A, 200 kHz
Digital inputs and outputs SB 1223 DC/DC SB 1223 DC/DC SB 1223 DC/DC	6ES7 223-0BD30 6ES7 223-3AD30 6ES7 223-3BD30	DI 2 × 24 V DC, DO 2 × 24 V DC / 0.5 A, 30 kHz DI 2 × 5 V DC, DO 2 × 5 V DC / 0.1 A, 200 kHz DI 2 × 24 V DC, DO 2 × 24 V DC / 0.1 A, 200 kHz
Analog inputs SB 1231 AI SB 1231 RTD SB 1231 TC	6ES7 231-4HA30 6ES7 231-5PA30 6ES7 231-5QA30	AI 1 × 12 bit, $\pm 10/\pm 5/\pm 2.5$ V or 0 to 20 mA RTD 1 × 16 bit, type: Platinum (Pt) TC 1 × 16 bit, types: J, K, $\pm 80$ mV
Analog output SB 1232 AO	6ES7 232-4HA30	AO 1 × 12 bit, $\pm 10$ V or 0 to 20 mA

### Communication Board (CB)

A Communication Board (CB) expands the communication connections without changing the dimensions of the CPU. The associated slot is located on the front of the CPU.

The CB 1241 RS485 is available for serial data exchange via a point-to-point connection. The protocols for ASCII, USS drives, and Modbus RTU are already implemented; additional protocols can be subsequently loaded.

### Battery board (BB)

With the BB 1297 battery board, the buffered runtime of the real-time clock can be extended from a typical 10 days to up to one year, without changing the dimensions of the CPU. The associated slot is located on the front of the CPU.

A commercially available CR1025 battery (not included) is used for voltage buffering. The maintenance LED on the board indicates when it is necessary to change the button cell.

## 2.3 Signal modules (SM)

Signal modules are input/output modules which extend the integrated I/O. Depending on the CPU version, either none, two or eight modules can be plugged onto the right of the CPU.

A two-tier design is possible using a 2-meter long extension cable. But the number of modules which can be used is not changed as a result.

### 2.3.1 Digital I/O modules

Digital modules are signal converters for binary process signals. The CPU can use input modules to scan operating modes of the machine or plant, and output modules for intervention.

Input modules convert external signals of 24 V DC into signals with an internal level. Output modules convert the internal signal level into 24 V DC (electronic outputs) or are designed as relay outputs. Using direct current, a relay can switch maximum 30 W, while with alternating current it can switch 200 W. There are signal modules with one or two bytes corresponding to 8 or 16 signals (Table 2.3).



**Fig. 2.7** SM 1221 DI16 signal module

**Table 2.3** Selection of digital I/O modules

Digital inputs SM 1221 DC SM 1221 DC	6ES7 221-1BF30 6ES7 221-1BH30	DI 8 × 24VDC DI 16 × 24VDC
Digital outputs SM 1222 DC SM 1222 DC SM 1222 RLY SM 1222 RLY SM 1222 RLY	6ES7 222-1BF30 6ES7 222-1BH30 6ES7 222-1HF30 6ES7 222-1HH30 6ES7 222 1XF30	DO 8 × 24VDC / 0.5 A DO 16 × 24VDC / 0.5 A DO 8 × relay 30 V DC, 250 V AC / 2 A DO 16 × relay 30 V DC, 250 V AC / 2 A DO 8 × relay 30 V DC, 250 V AC / 2 A (change-over contact)
Digital inputs/outputs SM 1223 DC/DC SM 1223 DC/DC SM 1223 DC/RLY SM 1223 DC/RLY SM 1223 AC/RLY	6ES7 223-1BH30 6ES7 223-1BL30 6ES7 223-1PH30 6ES7 223-1PL30 6ES7 223-1QH30	DI 8 × 24VDC, DO 8 × 24VDC / 0.5 A DI 16 × 24VDC, DO 16 × 24VDC / 0.5 A DI 8 × 24VDC, DO 8 × relay 30VDC, 250 V AC / 2 A DI 16 × 24VDC, DO 16 × relay 30VDC, 250 V AC / 2 A DI 8 × 125/250 V AC, DO 8 × relay 30 V DC, 250 V AC / 2 A

### 2.3.2 Analog input/output modules

Analog modules are signal transducers for analog process signals. The CPU can use these modules to process analog variables when these have been converted by analog input modules into digital values. The CPU can also continuously supply actuators with analog setpoints which have been generated from the specified digital values by analog output modules.

One “channel” is occupied on the module by each analog value (e.g. measured value or setpoint). Analog modules are available with 2, 4 or 8 channels corresponding to 4, 8 or 16 bytes (Table 2.4). A digitized analog value is represented internally as a 16-bit fixed-point number (data type INT). Analog modules can output diagnostic data concerning the module status or when limit values are reached.



**Fig. 2.8** SM 1234 AI/AO signal module

**Table 2.4** Selection of analog I/O modules

Analog inputs SM 1231 AI SM 1231 AI SM 1231 AI SM 1231 RTD SM 1231 RTD SM 1231 TC SM 1231 TC	6ES7 231-4HD30 6ES7 231-4HF30 6ES7 231 5ND30 6ES7 231-5PD30 6ES7 231-5PF30 6ES7 231-5QD30 6ES7 231-5QF30	AI 4 × 13 bit, $\pm 10/\pm 5/\pm 2.5$ V or 0 to 20 mA AI 8 × 13 bit, $\pm 10/\pm 5/\pm 2.5$ V or 0 to 20 mA AI 4 × 16 bit, $\pm 10/\pm 5/\pm 2.5/\pm 1.25$ V or 0 to 20/4 to 20 mA RTD 4 × 16 bit, 0 to 150/300/600 $\Omega$ ; (Pt, Ni, Cu) RTD 8 × 16 bit, 0 to 150/300/600 $\Omega$ ; (Pt, Ni, Cu) TC 4 × 16 bit, $\pm 80$ mV; types J, K, T, E, R, S, N, C, TXK/XX(L) TC 8 × 16 bit, $\pm 80$ mV; types J, K, T, E, R, S, N, C, TXK/XX(L)
Analog outputs SM 1232 AO SM 1232 AO	6ES7 232-4HB30 6ES7 232-4HD30	AO 2 × 14 bit, $\pm 10$ V or 0 to 20 mA AO 4 × 14 bit, $\pm 10$ V or 0 to 20 mA
Analog inputs/outputs SM 1234 AI/AO	6ES7 234-4HE30	AI 4 × 13 bit, $\pm 10/\pm 5/\pm 2.5$ V or 0 to 20 mA, AO 2 × 14 bit, $\pm 10$ V or 0 to 20 mA

### 2.3.3 Properties of the I/O connections

**Digital input modules** convert the external signal voltage (24 V DC or 125/230 V AC) into the internal signal level. The connected sensor must be within the permissible voltage range and provide the required input current with the signal status “1” so that the module can switch reliably.

The input signals are filtered in addition, i.e. interferences on the lines are suppressed and glitches eliminated. The filtering results in the input signals being delayed. The input delay can be set. A compromise must be found here between interference resistance (long delay) and fast signal recording (short delay).

Depending on the design, a digital input channel on the CPU module can trigger a process interrupt when there is a change in signal status, or be activated as a

“pulse pickup”. The latter case means that a signal pulse is detected on such an input channel which is shorter than the program execution time (cycle time).

Some of the input channels on the CPU module can be used as high-speed inputs which control the integral high-speed counter (HSC). Frequencies are possible up to 100 kHz (onboard on CPU module) or up to 200 kHz (on the signal board).

**Digital output modules** enable the CPU to intervene in the controlled process. They are signal transducers which convert the internal signal statuses into the voltages and currents used in the process. The digital output modules contain a data memory which receives the signals sent to the module and passes them on to an amplifier. The latter then provides the required switching capacity.

When selecting the digital modules it is necessary to take into consideration the operating frequency (the operating delay), the load rating per channel, the total load rating, and – with electronic output channels – the residual current. It is not permissible to fall below the residual current with a signal status “0” since otherwise the controlled device will no longer respond to a switch-off signal.

The digital output modules are disabled when in the STOP and STARTUP statuses. In this case they deliver either a configured substitute value, e.g. signal status “0” or retain the last set value.

Some of the electronic output channels on the CPU module can be controlled as “pulse outputs” by the integral pulse generators (pulse train output PTO or pulse-width modulation PWM). Frequencies are possible up to 100 kHz (on the CPU) or up to 200 kHz (on the signal board). The pulse generators are also used by the technology object *Axis*.

**Analog input modules** convert analog variables occurring in the process into a digital value by means of an integration procedure. The measuring ranges  $\pm 10$  V,  $\pm 5$  V,  $\pm 2.5$  V,  $\pm 1.25$  V, and 0 to 20 mA or 4 to 20 mA correspond with a resolution of 12 bits + sign in the internal representation to a numerical value from von  $-27\,648$  to  $+27\,648$ . In addition, there is overshoot or undershoot range and the overflow or underflow. Overflow and underflow can trigger a diagnostics event.

Depending on the interference voltage suppression for 400, 60, 50 or 10 Hz, the response times without smoothing are 4, 18, 22 or 100 ms. The measured values can be smoothed by digital filtering. Smoothing of the analog signal can be set to “strong” (over 32 cycles), “medium” (16 cycles), “weak” (4 cycles) or “none” (1 cycle).

**Analog output modules** convert the internal digital values into the analog variables required in the process. In the voltage range  $\pm 10$  V the resolution is 14 bits with a rated range from  $-27\,648$  to  $+27\,648$ . The permissible load impedance is  $\geq 1000\ \Omega$ . In the current range from 0 to 20 mA the resolution is 13 bits with a rated range from 0 to 27 648. The permissible load impedance in this case is  $\leq 600\ \Omega$ . The analog output modules are disabled when in the STOP and STARTUP statuses. In this case they deliver either a configured substitute value or retain the last set value.

## 2.4 Communication modules (CM)

The communication modules (CM) support the CPU in communication tasks. They establish the physical connection to a communication partner, take over establishment of the connection and data transport on this, and provide the required communications services for the operating system of the CPU and the user program.

The communication modules are plugged onto the CPU from the left, seen from the front. Operation of up to three communication modules is possible for all CPUs (Table 2.5).



**Fig. 2.9** CM 1241 RS485 communication module

**Table 2.5** Communication modules with functions and properties

CM 1241 RS232	6ES7 241-1AH30	Point-to-point connection via RS 232
CM 1241 RS422/485	6ES7 241-1CH31	Point-to-point connection via RS 422/485
CM 1242 PROFIBUS	6GK7 242-5DX30	PROFIBUS DP slave
CM 1243 PROFIBUS	6GK7 243-5DX30	PROFIBUS DP master
CM 1243 AS-i	3RK7 243-2AA30	AS-Interface master
DCM 1271 AS-i	3RK7 271-1AA30	AS-i data decoupling module
CP 1242 GPRS	6GK7 242-7KX30	Connection to a GSM/GPRS radio network

### 2.4.1 Point-to-point communication

The CM 1241 communication modules allow fast serial data transfer via a point-to-point connection. The modules are available in two designs, depending on the physical transmission properties: with RS232 interface and with RS422/485 interface. There are standard protocols for point-to-point communication (ASCII), Modbus communication, and the universal serial interface (USS drive protocol).

The point-to-point communication is suitable for connecting, for example, printers, modems, or barcode readers to the programmable controller.

### 2.4.2 PROFIBUS DP

The CM 1243-5 communication module connects an S7-1200 station to PROFIBUS DP as DPV1 master according to IEC 61158. The module can control up to 16 PROFIBUS DP slaves. Transmission rates of up to 12 Mbit/s are supported. The CM 1243-5 communication module also allows a programming device or an operator station to be connected to the S7-1200 station. Data can be exchanged with other S7 stations via the S7 communication using the system functions GET and PUT.

The CM 1242-5 communication module connects an S7-1200 station to PROFIBUS DP as DPV1 slave according to IEC 61158. Transmission rates of up to 12 Mbit/s are supported. The connection to PROFIBUS DP takes place as an “intelligent” DP slave that can exchange data with the DP master via a configurable user data interface.

### 2.4.3 Actuator/sensor interface

The CM 1243-2 communication module connects an S7-1200 station with the AS-Interface cable as AS-i master according to AS-Interface specification V3.0. The module can control up to 62 AS-i slaves. A standard power supply with 24 V is used in conjunction with the DCM 1271 data decoupling module.

By connecting to the AS-Interface, the available inputs and outputs for an S7-1200 station can be significantly increased (per CM module a maximum of 496 digital inputs and 496 digital outputs or 31 standard analog slaves with up to 4 channels or 62 A/B analog slaves with up to 2 channels).

The CM 1243-2 communication module is included with a hardware support package (HSP) in the STEP 7 V11 hardware catalog.

### 2.4.4 GPRS transmission

The CP 1242-7 communication module connects an S7-1200 station with the GSM/GPRS mobile network (Global System for Mobile Communications/General Packet Radio Service, a packet-oriented service for data transmission in digital mobile networks). Data can be exchanged wirelessly with a transmission rate of 86 Kbit/s downlink and 43 Kbit/s uplink. The CP1242-7 module can receive and send text messages (Short Message Service). The partner can be a mobile phone or another S7-1200 station. Operation is possible with a standard mobile phone contract.

In conjunction with the *Telecontrol Server Basic* software, the GPRS module forms a telecontrol system on the basis of mobile wireless communication. Data transmission is possible between the remote control center and an S7-1200 station or between two S7-1200 stations with a “detour” via the remote control center. For direct data transmission between two S7-1200 stations, the module must be assigned a fixed IP address.

The CP 1242-7 communications processor can also set up a TeleService connection. This allows the loading of project data and the retrieval of diagnostic data via the GPRS network. The connection between the programming device and the GPRS network can be established via a telecontrol server or a TeleService gateway. The CP 1242-7 communication module is included with a hardware support package (HSP) in the STEP 7 V11 hardware catalog.



**Fig. 2.10** CP 1242-7 with antenna

## 2.5 Further modules

### 2.5.1 Compact switch module (CSM)

The connection multiplier *CSM 1277 unmanaged* allows simple design of an Ethernet network. It includes four RJ45 sockets with which an S7-1200 station can be connected to other devices over Industrial Ethernet, for example to other programmable controllers or to HMI stations.

The connection multiplexer is designed for 10/100 MBit/s. It automatically detects the data transfer rate (autosensing) and can be used with standard or crossover cables (autocrossover function).

Status LEDs indicate the existing power supply, the port status, and the current data traffic. The interface multiplier does not require configuration.



**Fig. 2.11** CSM 1277 compact switch module

### 2.5.2 Power module (PM)

The power supply for the CPU module is either 120/230 V AC with 50/60 Hz or 24 V DC. The CPU derives the required internal voltages from this voltage.

If the onboard power supply is insufficient, the external PM 1207 power supply can be used. The input voltage is 120/230 V AC with a current consumption of 1.2/0.7 A. 24 V DC with 2.5 A is available at the output.

A status LED on the front indicates that the 24 V output voltage is present. The PM 1207 does not require configuration.



**Fig. 2.12** PM 1207 power module

### 2.5.3 TS Adapter IE Basic

With TeleService, remote servicing of SIMATIC S7 automation systems or HMI devices is possible using the programming device over a fixed line or wireless network. TeleService extends the connection from the programming device via the telephone network to the programmable controllers. The functions to be performed, such as programming, are completed using the same tools and functionality as if the job was being done locally.

The TS Adapter IE Basic consists of the basic unit and a TS module with the modem or an interface for connecting to an external modem. The basic unit has an Ethernet interface for connecting to a programming device or programmable controller. The available TS modules are listed in the Table 2.6. The TS-Adapter IE Basic is parameterized with TeleService in the TIA Portal.

**Table 2.6** Basic unit and connection modules for teleservice

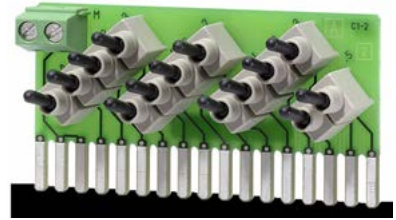
TS Adapter IE Basic	6ES7 972-0EB00	Basic unit
TS module modem	6ES7 972-0MM00	Analog modem for connection to the analog telephone network
TS module ISDN	6ES7 972-0MD00	Terminal adapter for connection to the ISDN network
TS module RS232	6ES7 972-0MS00	RS232 interface for connecting an external modem
TS module GSM	6GK7 972-0MG00 6NH9 860-1AA00	Wireless modem for connection to the GSM/GPRS network Quadband GSM antenna

The TS Adapter IE Basic can establish a modem connection to a dial-up server, for example, an Internet service provider. The system function TM\_MAIL enables sending an email from the user program.

### 2.5.4 SIM 1274 simulator

The simulator module is used to debug the user program. It is connected to the terminal block of the CPU instead of the input signals from the machine.

The simulator module is available with 8 switches (for CPU 1211 and CPU 1212) and with 14 switches (for CPU 1214 and CPU 1215). The power supply is 24 V DC.

**Fig. 2.13** SIM 1274 simulator

## 2.6 SIPLUS S7-1200

SIPLUS is the product range with hardened components for use in harsh environments. The SIPLUS S7-1200 range is offered in addition to the standard SIMATIC S7-1200 range.

Standard devices which have been specially converted and “refined” for the respective application are used as the basis for the SIPLUS components. Possible refined features include:

- ▷ Extended ambient temperature range from  $-25\text{ °C}$  to  $+70\text{ °C}$
- ▷ Condensation, increased humidity, increased degree of protection (dust, water)
- ▷ Extreme loading by media, e.g. toxic atmospheres
- ▷ Increased mechanical load, increased electrical immunity

**Fig. 2.14** SIPLUS CPU 1211C AC/DC/RLY



- ▷ Voltage ranges deviating from the standard
- ▷ Ambient conditions on track vehicles
- ▷ Specific sector solutions

SIPLUS modules are manufactured on request for the desired environmental conditions. Please therefore note the technical specifications for the module concerned.

### **Configuration of SIPLUS modules**

The functionality of a SIPLUS module is the same as that of the corresponding standard module; the Order No. (MLFB, machine-readable product designation) commences with 6AG1... SIPLUS modules are not included with their Order Nos. in the hardware catalog of the STEP 7 Basis programming software.

Since the SIPLUS modules have the same functions as existing modules, you can use the corresponding equivalent type (the standard module) when configuring. This equivalent type can be found on the device's nameplate, in the SIPLUS data sheets, and on the Internet in the Siemens A&D Mall. For example, the SIPLUS CPU 1211C shown in Fig. 2.14 with order number 6AG1211-1BD30-2XB0 is based on the S7-CPU 1211C with order number 6ES7 211-1BD30-0XB0.

## 3 Device configuration

### 3.1 Introduction

Device configuration entails planning the hardware design of your automation system. Configuration is carried out offline without a connection to the CPU. You can use this tool to add PLC stations to a project and equip these with modules which you then address and parameterize. You also use this tool to carry out the networking of PLC stations or the creation of distributed I/O stations.

This chapter primarily describes the configuration of an individual PLC station with a CPU 1200 controller and provides an overview of the networking options. Configuration of the distributed I/O is described in Chapters 14.2 “PROFINET IO” on page 456 and 14.3 “PROFIBUS DP” on page 462.

#### Start

You can start the device configuration in the Portal view when setting up a new project if the *Open device view* checkbox is activated following addition of a CPU. When opening an existing project, start the device configuration by selecting *Configure a device*.

In the Project view, you can start the device configuration in the project tree by double-clicking on the *Devices & networks* editor under the project or on the *Device configuration* editor under the PLC station.

The *Device view* tab shows the occupation of the station rack. You configure the station in this view. The *Network view* tab shows the networking between multiple stations. In this view you can add further subnets and stations, and configure their networking (described in Chapter 3.4 “Configuring the network” on page 67). In the *Topology view* tab you configure the geographic arrangement of the Ethernet network.

#### Working in the Device view

The device configuration shows a rack with the current modules. If several PLCs are present in the project, select the one you wish to edit in the toolbar of the working window. You can now drag new modules with the mouse from the hardware catalog to the rack, or remove existing ones. In the inspector window you can set the properties of the selected module such as the technological functions of the CPU or the addresses of the input/output modules.

### Working in the Network view

The Network view shows the stations present in the project and their networking. With the *Network* button activated, you connect two devices into a network by selecting an interface in a station and dragging it with the mouse to another station. A subnet is then created automatically. You can connect a station to an existing subnet by dragging the interface with the mouse to the subnet. With the *Connections* button activated, you define a connection by selecting a subnet and then the type of connection from the drop-down list; alternatively, the type of connection is determined during programming of the communication functions, for example with open user communication.

You set the properties of the selected objects in the inspector window, e.g. the line configuration and the bus parameters, when networking with PROFIBUS.

### Working in the Topology view

The Topology view shows the networking between the stations on the Ethernet bus system. The connections between the device ports are shown. The connections can be created, changed, or deleted.

In online mode, the Topology view shows the differences between the reference and actual topologies. A topology present online can be adopted offline as configuration information.

### Save, compile and download

You save the entered data on the hard disk by saving the complete project (using the *Project > Save* command in the main menu). In order to download the configuration data to a CPU module, it must first be compiled in a form understandable to the CPU (using *Edit > Compile*). Any errors occurring during compilation are indicated in the inspector window under *Info*. Only error-free (consistent) compilations can be downloaded to the CPU module using *Online > Download to device*.

### Working area of the device configuration

The device configuration shows the project in the Project view. Fig. 3.1 shows the working area of the device configuration without project tree.

Three views are available in the **Working window**:

- ▷ The *Device view* shows the current configuration of the PLC station. The configuration is shown as a graphic in the top part of the window, and as a table in the bottom part.
- ▷ In the *Network view* you can see – if more than one station is present in the project – the connections between the stations, also as a graphic in the top part of the window and with the existing stations and their interconnections as a table in the bottom part.

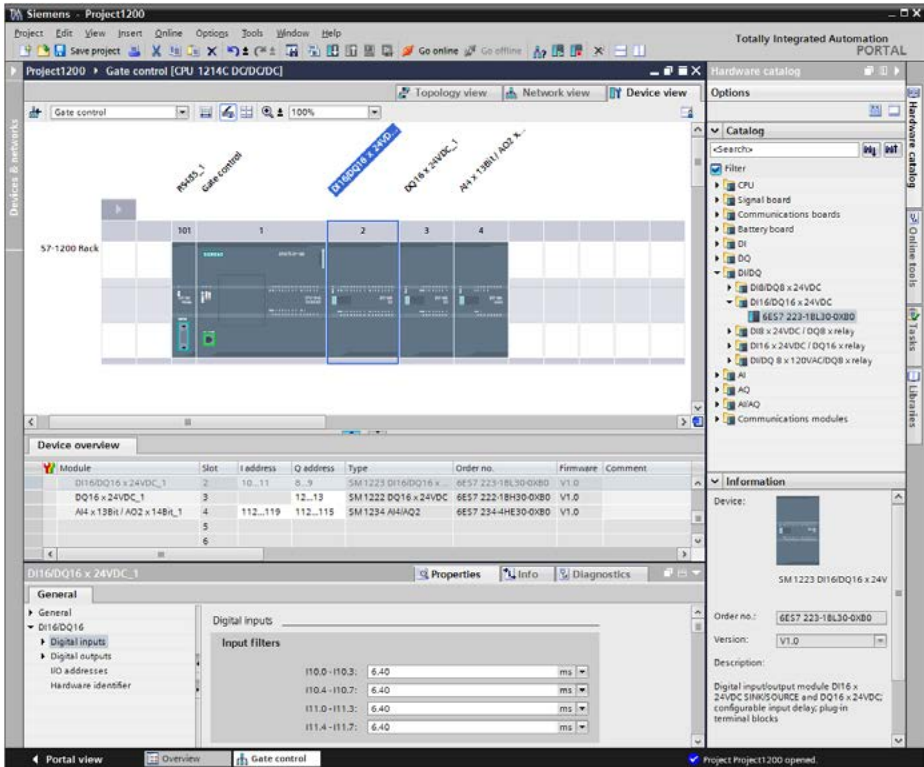


Fig. 3.1 Example of working area of device configuration (Device view)

You can use the *Topology view* to display and configure the port connections with an Ethernet network as a graphic in the top part of the window and as a table in the bottom part.

In all cases, you can “fold shut” the bottom part of the working window.

The **Inspector window** is positioned below the working window. In the *Properties* tab, this shows the properties of the object selected in the working window. The *Info* tab contains general information on the configuration session and the compilation, and the cross-reference list. The *Diagnostics* tab shows the operating mode of the stations and the message display.

The **Hardware catalog** is available on the right in the task window. It shows all hardware components which can be configured with the current version of STEP 7. If you select a component in the lowest level of the hardware catalog, a brief description of the most important properties is shown in the information area of the hardware catalog.

You can change the size of all windows. You can close all windows except the working window, thus providing more space for the working area. The working window can also be maximized and displayed as a separate window.

## Expanding the hardware catalog

If you want to configure a PROFIBUS DP slave that does not appear in the hardware catalog, you must install the GSD file provided by the manufacturer. A GSD file (General Station Description file) contains all the PROFIBUS DP slave properties as text with keywords. If you want to configure a PROFINET IO device that does not appear in the hardware catalog, you must install the GSDML file provided by the manufacturer. This is an XML file containing all properties of a PROFINET IO device as GSDML (General Station Description Markup Language).

To subsequently install GSD and GSDML files, select the command *Options > Install general station description file (GSD)* from the main menu. Enter the source path in the subsequent dialog, and select the file to be installed.

If, after the publication date of STEP 7, new components such as modules come on the market, these components can be added later to the hardware catalog with hardware support packages (HSP).

To subsequently install support packages, select the command *Options > Support Packages* from the main menu. The *Detailed information* window displays the installed products and components as well as operating system information. Under *Installation of Support Packages*, you can select whether you wish to download the update from the Internet or add existing support packages from the file system.

## 3.2 Configuring a station

“Configuring” is understood to be the addition of a station to the project or, with a PLC station, the arranging of the modules in a rack and the fitting of modules with submodules.

### 3.2.1 Adding a PLC station

When creating a new project, you normally add a PLC station at the same time. You can add further PLC and HMI stations in both the Portal view and the Project view. In the Portal view, you can add a new station in the *Devices & networks* portal using the *Add new device* command. In the Project view, double-click on *Add new device* in the project tree.

Select the desired CPU in the selection window, and assign it a meaningful name. Before clicking on the *OK* button, make sure that the *Open device view* check box is activated in the window at the bottom left.

You have now configured a rack with a CPU module inserted in slot 1. All slots possible for a CPU 1200 are shown on the right of this. The number of slots depends on the selected CPU. You can open the display to the left of the CPU and thus expand the rack for inserting communication modules.

### 3.2.2 Arranging modules

If not already done, start the *Device configuration* editor in the project tree underneath the PLC station by double-clicking. To insert a further module, select it in the hardware catalog (the symbol of the module in the lowest catalog level). You are then provided with a description of the selected module in the information window of the hardware catalog. The permissible slots in the rack are displayed. Position the new module by double-clicking on the module symbol or by dragging it with the mouse to the subrack. You can position a signal board directly on the CPU module in the same manner.

You can either delete an inserted module again (remove it from the rack) or replace it by a different, equivalent one.

### 3.2.3 Adding an HMI station

Using the *Add new device* command you can add an HMI station to the project either in the *Devices & networks* portal or in the project tree. Select the desired HMI device in the selection window, and assign it a meaningful name. Before clicking on the *OK* button, make sure that the *Use device wizard* check box is activated in the window at the bottom left if you wish for guidance when starting the configuration. The further procedure is described in Chapter 4.1.3 “Operand area bit memory” on page 82.

## 3.3 Assigning module parameters

“Parameterization” is understood to be the setting of module properties. These include, for example, setting addresses, enabling interrupts, or defining communication properties.

Module parameterization is carried out for a selected module in the inspector window in the *Properties* tab. Select the properties group on the left side, and set the values in interactive mode on the right. You can stop the setting of properties at any time and continue later. Only a portion of the total parameters described below can be assigned to individual modules.

### 3.3.1 Parameterization of CPU properties

The CPU's operating system operates with the default settings for program execution. You can change these default settings in the hardware configuration during parameterization of the CPU and match them to your specific requirements. Subsequent modification is possible at any time.

When starting up, the CPU adopts the settings deviating from the default settings in STARTUP mode. These settings then apply to further operation.

To parameterize the CPU properties, select the CPU in the working window of the device configuration. If the project contains several stations, select the desired station in the menu bar of the working window.

You set the name of the PLC station in the **General** section. Catalog information about the CPU is also shown.

In the **PROFINET interface** section under *Ethernet addresses*, you set the connection to an Ethernet subnet and define the IP address and the subnet mask. For more information on the format of the IP address, refer to Section “IP address and subnet mask” on page 73. Under *Advanced* you can parameterize the real-time properties of the PROFINET IO communication and the port interconnection (networking of the connection).

The parameterization of the **digital and analog inputs/outputs** is described in 3.3.2 “Addressing input and output signals” on page 64.

A **high-speed counter** (HSC) can count at a frequency of up to 200 kHz when using an appropriate signal board, and thus much faster than a software counter. You must activate the high-speed counter before you can use it. A high-speed counter requires defined inputs which are then no longer available for other purposes. Further information on the application and parameterization of high-speed counters can be found in Chapter 17.1.1 “High-speed counter (HSC)” on page 548.

The **pulse generators** provide pulses of up to 200 kHz when using an appropriate signal board. A pulse generator has two modes of operation: PTO (pulse train output) and PWM (pulse-width modulation). Pulses are output via a special onboard output. Further information on the application and parameterization of pulse generators can be found in Chapter 17.1.2 “Pulse generator” on page 554.

You can set the **startup characteristics of the CPU** under *Startup*. This is where you determine how the CPU module is to react when the power supply is switched on:

- ▷ No startup (CPU remains in STOP mode)
- ▷ Warm restart – RUN (the CPU executes the user program)
- ▷ Warm restart – operating mode prior to POWER OFF (either STOP or RUN)

The duration for module parameterization for the distributed I/O is monitored during a startup. You can set the monitoring time here. A module is considered to be absent if the monitoring time for it expires. Further information on the CPU startup can be found in Chapter 5.1.2 “STARTUP mode” on page 118.

In the **Cycle** section, you set the *scan cycle monitoring time* with which the processing of the main program is monitored. If the cycle monitoring time is exceeded, this is reported and can lead to the STOP mode. Furthermore, you can activate a fixed minimum cycle time, i.e. the next (cyclic) execution of the main program is only started after this time has elapsed. Further details can be found in Chapter 5.6.3 “Cycle time” on page 144.

In the **Communication load** section you define the time share for communication. In addition to execution of the user program, the CPU also carries out communication tasks, for example data transmission to another PLC station or downloading of blocks from a programming device. This communication requires time, some of which has to be added to the execution time of the main program. Specification of the communication load can be used to control influencing of the cycle time to a

certain extent. The time available for communication is entered as a percentage in the *Communication load* parameter. The cycle time is then extended by the factor  $100 / (100 - \text{communication load})$ .

**System and clock memory** are operands controlled by the operating system which can be queried in the user program. For example, there is a bit memory which indicates the occurrence of a diagnostics event, and also a bit memory which changes its signal status at a frequency of 2 Hz. During parameterization of the CPU you activate the system memories and/or the clock memories and assign addresses to them. Further information on bit memories in general and on system and clock memories can be found in Chapter 4.1.3 “Operand area bit memory” on page 82.

In the **Web server** section you can activate the Web server and set its properties. Further details can be found in Chapter 17.4 “Web server” on page 567.

In the **Time of day** section you set the time zone for the integral real-time clock and activate the daylight-saving time setting (difference between daylight-saving and standard time, start and end of daylight-saving time).

In the **Protection** section you can protect the program in the CPU from unauthorized access. Here you select whether access protection is to be switched on, and whether this is to be just write protection or combined read/write protection. Assign a password if the read or write protection is activated. Anyone in possession of the password has unlimited access to the CPU.

The **Connection resources** section shows the distribution of available connections to the programming device and other networked stations.

Gate control [CPU 1214C DC/DC/DC] Properties Info Diagnostics

General

- General
- PROFINET interface
- DI14/DQ10
- AI2
- High speed counters (HSC)
- Pulse generators (PTO/PWM)
- Startup
- Cycle
- Communication load
- System and clock memory
- Time of day
- Protection
- Overview of addresses

Overview of addresses

Filter:  Inputs  Outputs  Address gaps  Slot

Type	Addr. fr...	Addr. to	Module	PIP	DP	PN	Rack
I	64	67	AI2	OB1-PI	-	-	0
I	0	1	DI14/DO10	OB1-PI	-	-	0
I	1000	1003	HSC_1	OB1-PI	-	-	0
I	1004	1007	HSC_2	OB1-PI	-	-	0
I	1008	1011	HSC_3	OB1-PI	-	-	0
I	1012	1015	HSC_4	OB1-PI	-	-	0
I	1016	1019	HSC_5	OB1-PI	-	-	0
I	1020	1023	HSC_6	OB1-PI	-	-	0
I	112	119	AI4/AO2	OB1-PI	-	-	0
I	10	11	DI16/DQ16	OB1-PI	-	-	0
Q	0	1	DI14/DO10	OB1-PI	-	-	0
Q	1000	1001	Pulse_1	OB1-PI	-	-	0

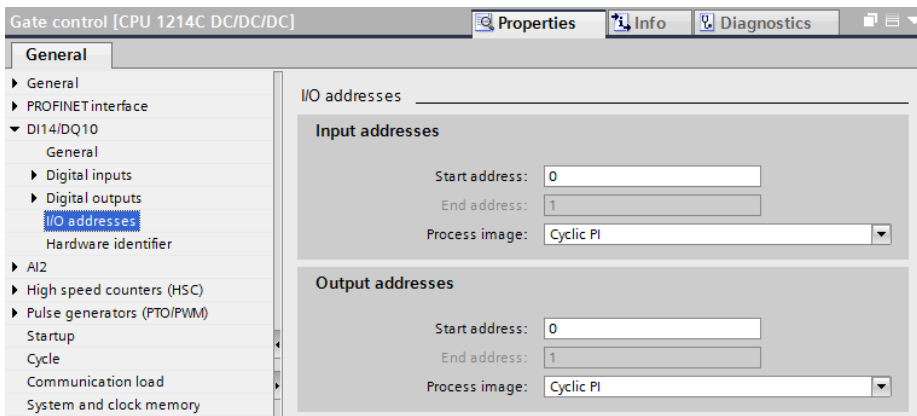
Fig. 3.2 Example of the address overview of a CPU 1200



The assigned inputs and outputs are shown in the **Overview of addresses**. The addresses of the configured modules, the slots, and – if applicable – the number of the PROFIBUS master system or PROFINET IO system used are displayed (Fig. 3.2).

### 3.3.2 Addressing input and output signals

When configuring the modules, the hardware editor automatically assigns a module start address. You can see this address in the configuration table in the bottom part of the working window or in the properties of the selected module in the inspector window in the section *I/O addresses*, specifying the input/output type: DI (= Digital Input), DQ (= Digital Output), AI (= Analog Input), and AQ (= Analog Output). Fig. 3.3 shows an example for the integrated inputs/outputs on a CPU 1214.



**Fig. 3.3** Example of parameterization of I/O addresses of a CPU 1200

You can change the automatically assigned addresses. Since the address area is separate for inputs and outputs, these can have the same addresses (see Chapter 4.2 “Addressing” on page 85). Each input and each output requires an unambiguous address.

The module start address is the first byte (the first 8 bits) of the module. If a module has several bytes, these occupy the next addresses. When assigning the address you can set whether the inputs and outputs are to be updated cyclically in the process image. If you deselect this option, you must access the peripheral I/Os directly in the program (see Chapter 5.6.2 “Process image update” on page 143).

The address dialog also shows the hardware identification (HW-ID) of the module or component. The internal number is displayed. This number is assigned by the hardware editor when configuring and cannot be changed.

### 3.3.3 Parameterization of digital inputs

Set the following parameters in addition to the module address:

Use the **input filter** to set the input delay time. This defines the resistance of an input channel to high-frequency noise signals. The longer the delay time, the greater the interference resistance. However, this also increases the detection period until a change in the input signal is recognized by the controller.

With appropriately designed input channels, e.g. onboard input channels, you can assign a **hardware interrupt** to each change in input signal. The hardware interrupt is enabled if you activate the corresponding check box for rising and/or falling edges. Subsequently provide the process interrupt with a name and assign an organization block to it. You can select the organization block from a list if it has already been created, or create a new block with a number starting from 200.

If you activate an integrated input channel for **Pulse catch**, it remains at signal status “1” in the process image for a complete program cycle, even if the pulse was so short that it would not “normally” have been detected by the user program.

### 3.3.4 Parameterization of digital outputs

Set the following parameters in addition to the module address:

With the digital output modules you can set the **Reaction to CPU STOP**. You can choose between *Use substitute value* and *Keep last value* (Fig. 3.4).

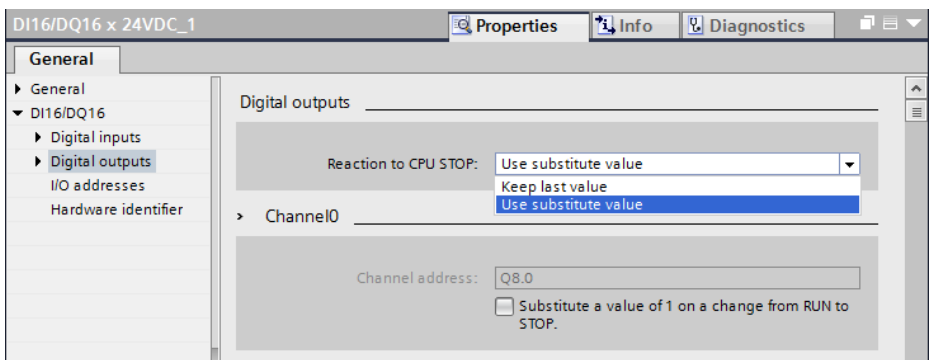


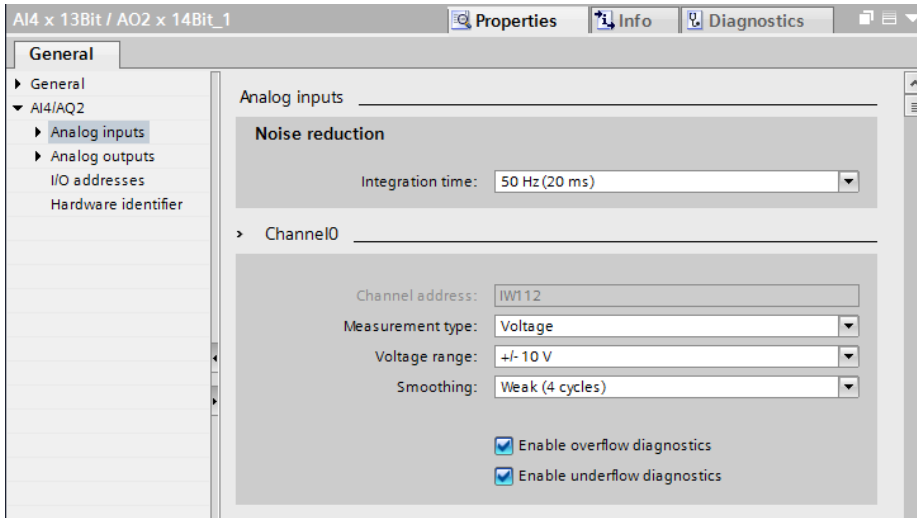
Fig. 3.4 Example of parameterization of a digital output channel

You set the **Substitute value** in the properties of an output channel. The substitute value “0” is output as standard with digital outputs. Signal status “1” is output if the *Substitute a value of 1 on a change from RUN to STOP* check box is activated.

### 3.3.5 Parameterization of analog inputs

Set the following parameters in addition to the module address:

In **Module diagnostics** you can activate monitoring of the power supply so that a diagnostics interrupt is generated in the event of a failure.



**Fig. 3.5** Example of parameterization of an analog input channel

Use **Noise reduction** to set the integration time. Because they frequently have a low signal level, analog signals can be “noisy” due to the line frequency. By selecting an integration time which differs from the line frequency, crosstalk of the line frequency is reduced (Fig. 3.5).

For the **Measurement type** you can set *Voltage* with the ranges  $\pm 2.5$  V,  $\pm 5$  V and  $\pm 10$  V or *Current* (0 to 20 mA). With smoothing in the steps *None* (1 cycle), *Weak* (4 cycles), *Medium* (16 cycles) and *Strong* (32 cycles) you can prevent excessive variation of the recorded analog signal.

In **Channel diagnostics** you can activate the *Enable overflow diagnostics* and *Enable underflow diagnostics* check boxes to trigger a diagnostics interrupt when the measured value exceeds or falls below the standard range.

### 3.3.6 Parameterization of analog outputs

Set the following parameters in addition to the module address:

In **Module diagnostics** you can activate monitoring of the power supply so that a diagnostics interrupt is generated in the event of a failure.

With the analog output modules you can set the **Reaction to CPU STOP**. You can choose between *Use substitute value* and *Keep last value*.

You set the **Substitute value** in the properties of an output channel. The substitute value 0 (zero) is output as standard with analog outputs. You can enter a different value in the input box *Substitute value for channel on a change from RUN to STOP*.

Under **Analog output type** you can set the *Voltage* ( $\pm 10$  V) or *Current* (0 to 20 mA). For the **Channel diagnostics** you can activate the check box for short-circuit diagnostics for *Voltage*, or the check box for open-circuit diagnostics for *Current*. Overshoot and undershoot diagnostics are activated as standard. A diagnostics interrupt is triggered if a corresponding event occurs.

## 3.4 Configuring the network

### 3.4.1 Introduction

The network configuration permits the graphic display and documentation of the configured networks and their stations. Configuration of the networking is part of the device configuration. If a PLC station is operated on its own – without an HMI station and without data communication to other PLC stations – you do not require network configuration. Connection of a programming device for transfer of the user program and for program debugging does not require network configuration either.

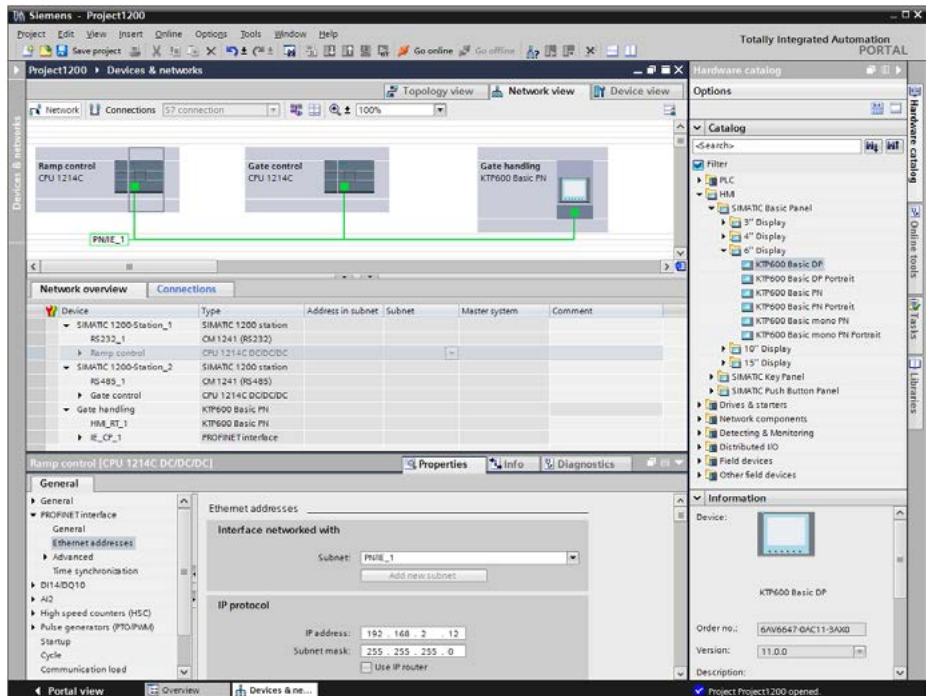


Fig. 3.6 Example of working area of network configuration (network view)

You can access the network configuration with the project opened in the Portal view via *Devices & networks* and *Configure networks* or in the Project view with the *Devices & networks* editor, which is positioned in the project tree underneath the project. In the working window of the device configuration, change to the *Network view* tab (Fig. 3.6).

In the top part of the working window, the Network view graphically displays all PLC, PC, and HMI stations present in the project as well as the networking, provided this has already been configured during device configuration. The bottom part of the working window contains the *Network overview*, *Connections* and *I/O communication* tabs. You can drag further stations with the mouse from the hardware catalog into the working area and thus add them to the project. The inspector window shows the properties of the selected object.

#### 3.4.2 Networking stations

“Networking” of stations corresponds to the wiring of modules with communication capability, i.e. a *mechanical* connection is established. A *logical* connection is additionally required in order to transfer data on the cable. The logical connection defines the transmission parameters between the modules.

The working window of the hardware editor shows the existing stations with the modules with communication capability. The interfaces for the PROFIBUS, PROFINET, and AS-Interface subnets are highlighted. Communication modules that may be present are arranged on the left next to the CPU.

##### Adding a station in the network configuration

In the hardware catalog under *PLC > SIMATIC S7-1200 > CPU > [folder: CPU 12xx...] > [CPU]*, select the desired CPU and drag it with the mouse into the working area. The graphic shows the CPU as a representative for the complete PLC station with the existing bus interfaces.

If you drag the CPU to an existing subnet and if the CPU has an interface matching the subnet, the interface is directly connected to the subnet when adding.

##### Adding a communication module in the network configuration

In the hardware catalog under *PLC > SIMATIC S7-1200 > Communication modules > [folder: Subnet] > [folder: Modules] > [Module]*, select the desired communication module and drag it with the mouse into the station graphic on the working area. The module is shown with the existing bus interfaces in the PLC station next to the CPU.

##### Adding a subnet

Select the desired bus interface in the station graphic and then select the *Add subnet* command from the shortcut menu. A subnet corresponding to the bus interface is added.

## Networking a station

To network stations, click on the *Network* button in the toolbar of the working window.

If a subnet has not yet been created, select the bus interface in one of the stations and drag it to a bus interface of the other station which matches the subnet. The subnet is then added; the interfaces are connected by a colored line.

If the matching subnet is already present, select the bus interface in the station and drag it to the subnet. The interface is connected to the subnet by a colored line.

## Properties of the Ethernet network

The network configuration shows the Ethernet connections between several stations as a linear bus connection: all stations are hanging quasi on one line. Actually, an Ethernet connection is a point-to-point connection between the stations: each station is connected to exactly one partner station. The PROFINET interface of a CPU 1215 has two ports which are connected together by an integrated switch. A linear network can thus quasi be set up.

Modules without this integrated switch must be networked together via an external “distributor” with several connections. You can find these devices in the hardware catalog under *Network components > IE switches > [Group] > [Device type] > [Device]*.

In the S7-1200 automation system, a CSM 1277 Ethernet switch is available with four ports and in S7-1200 design. The switch is not displayed in the network view since it does not require any parameters. In the topology view, if all ports are connected individually, the switch can be found in the hardware catalog under *Network components > IE switches > Compact switch modules > CSM 1277 unmanaged > [Module]*.

## Disconnecting a module from the subnet or assigning it to a different subnet

If you wish to disconnect a module from the subnet, select the bus interface and then the *Disconnect from subnet* command in the shortcut menu. If all modules have been disconnected from a subnet, it is shown as an isolated subnet at the top left in the working area.

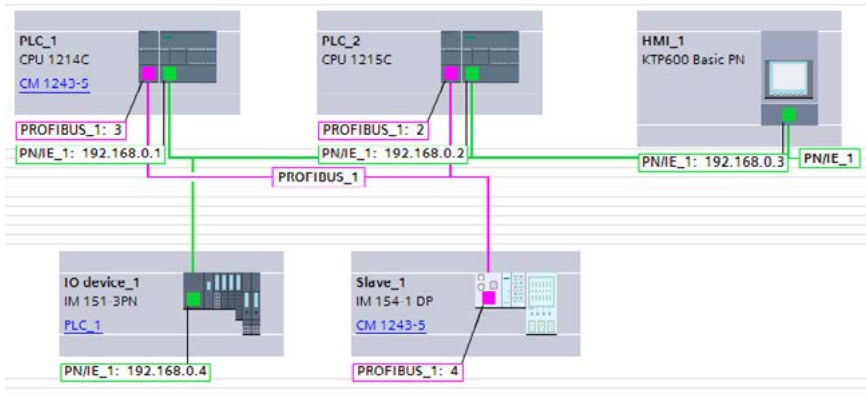
If you wish to assign a module to a new subnet, select the bus interface and then the *Assign to a new subnet* command in the shortcut menu. If several suitable subnets are available, select the appropriate one from the displayed list.

### 3.4.3 Node addresses in a subnet

Each module – each “node” – connected to a subnet requires an unambiguous address on the subnet (the “node address”) with which the module can be addressed within the subnet. When assigning node addresses, attention must be paid to the particular properties of the associated subnet.

## Display of node addresses

To display the node addresses in the Network view, click in the toolbar of the working window on the *Show address labels* icon. The Network view shows the name of the subnet and the node address. If the bus interface is not connected to a subnet, only the node address is displayed (Fig. 3.7).



**Fig. 3.7** Display of node addresses in the Network view

## Setting node addresses

When networking a module, the editor automatically occupies the next unused node address for the bus interface. You can change this automatically assigned address in the module properties in the inspector window with the bus interface selected.

### 3.4.4 Connectors

#### Introduction

A physical connection and a communication connection (logical connection) are required for communication between two devices.

The physical connection is established by “networking” during the configuration. The networking represents the cabling, even in the case of wireless transmission. The networking is represented graphically by the subnets.

Data transmission over the network requires a communication connection (logical connection). The logical connection defines the transmission parameters between the stations, such as the communication partners involved or the type of connection. The configured (logical) connections are listed in the connection table.

A connection is defined unequivocally by means of the “local (connection) ID”. In the communication functions program, this local ID specifies the connection via which the data is to be transmitted.

A connection is either dynamic or static depending on the communication service selected. Dynamic connections are not configured whose establishing and clearing down take place depending on events. Only one non-configured connection to a communication partner can exist at any time.

Static connections are configured in the connection table; they are established during the startup and are retained throughout the complete program execution. Several connections can be established in parallel to one communication partner. Under “Connection type” in the network configuration you can select the desired communication service.

### **Connection types**

Select the connection type depending on the subnet and the transmission protocol. In most cases this is the *S7 connection*. You can then exchange data between all S7 devices on all subnets. The programming device also uses this connection type for programming the PLC and HMI stations.

You use the HMI connection for communication with an HMI station. You use the other connection types, for example, if you wish to transmit data to third-party devices. This usually takes place by means of a communication module.

### **Connection resources**

Every connection requires connection resources on the communication partners involved for the end point of the connection and for the transition point in a communication module. Each CPU has a specific number of possible connections. Restrictions and rules apply to use of the connection resources. For example, not every connection resource can be used for every connection type.

The operating system of a CPU 1200 simultaneously supports the following asynchronous communication connections:

- ▷ 1 connection to a programming device
- ▷ 3 connections to an HMI device (HMI station)
- ▷ 8 connections for Open User Communication
- ▷ 3 connections for the S7 communication (PUT/GET) as server
- ▷ 8 connections for the S7 communication (PUT/GET) as client
- ▷ 1 connection to the Web server (HTTP)

One connection is always reserved for a programming device, and another one for an HMI station (these cannot be used for anything else).

### **Configuring connections**

In order to configure a connection in the network view, click on the *Connections* button in the toolbar of the working window, and select the connection type in the adjacent list. The devices suitable for this connection type are then displayed and highlighted (Fig. 3.8).



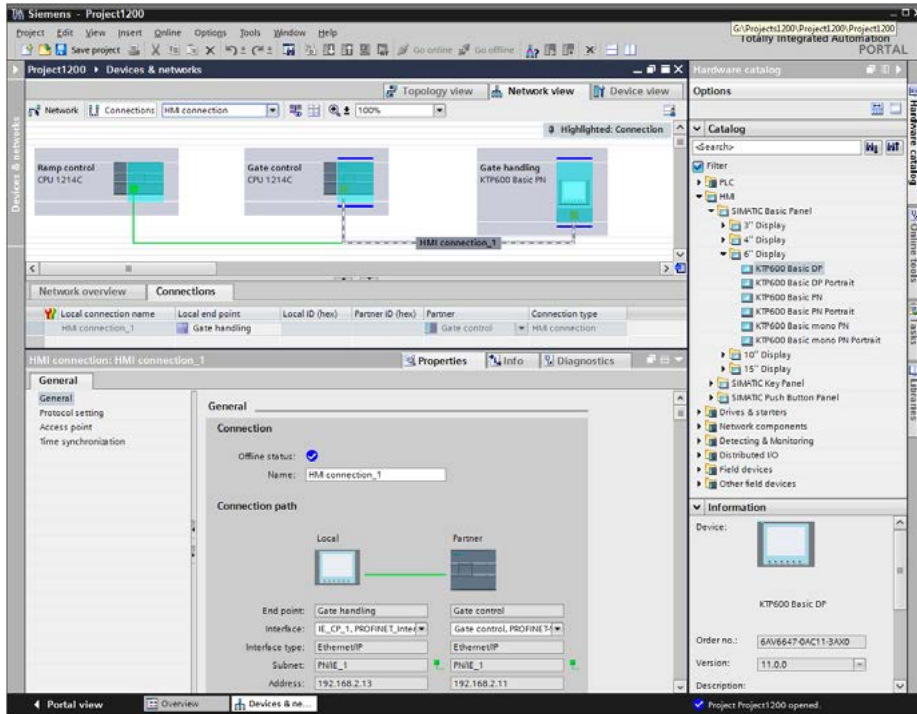


Fig. 3.8 Representation of an HMI connection in the network configuration

Click with the left mouse button on a station, drag the connection line with the mouse button pressed to the other station, and release the button. A connection with the connection name is displayed as a blue/white patterned line. Several logical connections can be created on one cable. These connections are then also present in the connection table in the *Connections* tab in the bottom part of the working window.

If you wish to determine which connections have been created in a subnet, click the *Connections* button and move the cursor to the subnet in the graphic display. If you click on one of the connections listed in the tooltip window, this connection is displayed highlighted in the Network view.

### Connection properties

Under *General* in the *Properties* tab, the inspector window shows the connection partners, the connection path, and the node addresses. If a station has several suitable interfaces, you can select the appropriate one from a drop-down list. You can set further connection properties in the bottom part of the Properties window, e.g. which partner is responsible for active establishment of the connection, and the local connection ID.

### 3.4.5 Configuring a PROFINET subnet

To configure a PROFINET subnet, drag the PN interface of one station to the PN interface of the other station with the mouse. A PROFINET subnet will be created automatically. You can also drag a PN interface to an existing PROFINET subnet.

#### Setting the properties of a PROFINET subnet

To set the properties, select the PROFINET subnet and then the *Properties* tab in the inspector window. Under *General* you can assign a different name to the subnet and also change the subnet ID if appropriate.

#### Setting the properties of a PN interface

To set the properties, select the PN interface and then the *Properties* tab in the inspector window. Under *General* you can set a different name for the interface. Under *Ethernet addresses* you set the IP address and the subnet mask of the CPU.

#### Ethernet address (MAC address)

The MAC (Media Access Control) address is an unambiguous address assigned to the device and defined by the manufacturer. It consists of three bytes with the manufacturer ID and three bytes with the device ID. The MAC address is usually printed on the device and is assigned to the latter during the configuration – if this has not already been carried out in the factory. The bytes are assigned in hexadecimal form (symbols 0 to F); the individual bytes are separated by colons. Example: 01:23:45:67:89:AB.

#### IP address and subnet mask

Each station on the Industrial Ethernet subnet which uses the TCP/IP protocol requires an IP (Internet Protocol) address. SIMATIC S7 supports Version 4 of the Internet protocol (IPv4). The IP address must be unique on the subnet. The IP address consists of four bytes, each separated by a dot. Each byte is represented as a decimal number from 0 to 255.

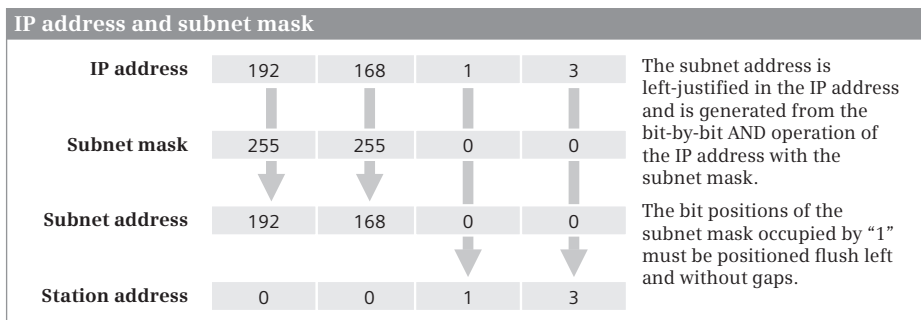


Fig. 3.9 Example of the structure of an IPv4 address

The IP address consists of the subnet address and the station address. The contribution made by the network address to the IP address is determined by the subnet mask. This consists – like the IP address – of four bytes which normally have a value of 255 or 0. Those bytes with a value of 255 in the subnet mask determine the subnet address, those bytes with a value of 0 determine the node address (Fig. 3.9).

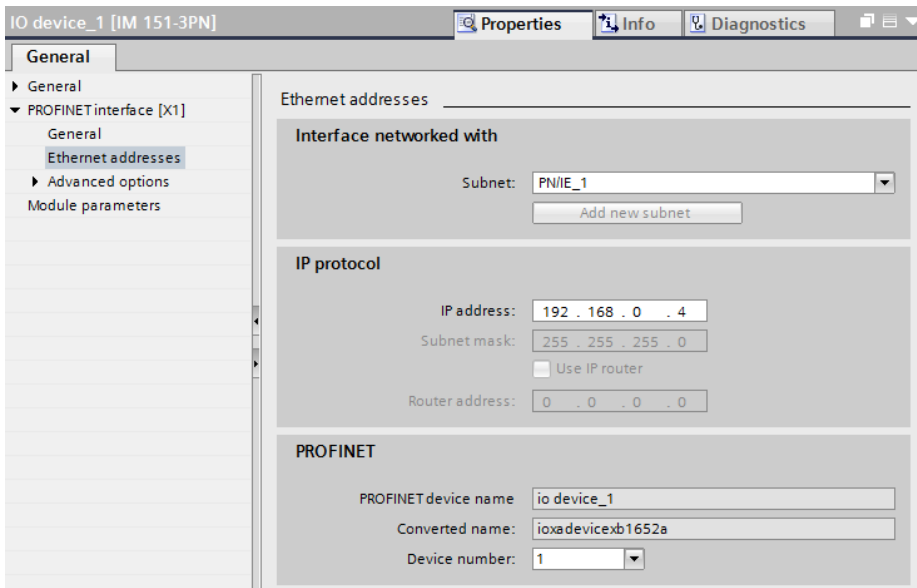
Values other than 0 and 255 can also be assigned in a subnet mask, thereby dividing up the address volume even further. The bits with “1” must be occupied beginning from the left without gaps.

The IP address is assigned one time for the IO controller when configuring with the hardware configuration for the nodes of a PROFINET IO system. Starting from this, the hardware configuration assigns the IP addresses to the IO devices in ascending order.

### Device name, device number

Every IO controller and every IO device has a device name. The device name is made up as standard from the name of the CPU used, the interface number, and the name of the PROFINET IO system: <CPU>.<Interface>.<IO system>. You can change the name of the respective component in its properties.

The interface number is only used if the CPU has more than one PN interface. The name of the IO system can be automatically appended to the device name, separated by a dot. To do this, activate the *Use name as expansion for PROFINET device name* checkbox in the properties of the PROFINET IO system.



**Fig. 3.10** Example of Ethernet addresses for an IO device

If the names used do not correspond to the conventions of IEC 61158-6-10 (name components basically consisting of lower-case letters, numbers, and hyphens separated by a dot), STEP 7 generates a so-called “converted” name which is then downloaded to the device (see Fig. 3.10).

As a supplement to the device name, the hardware configuration assigns a device number to each IO device which is independent of the IP address and which you can change. Using this device number (station number) you can address the IO device from the user program, e.g. as a current parameter on a system block.

### **IP address of the router**

A router establishes the connection between two subnets. If the target of a device connection is in a different subnet, the IP address of the corresponding router must also be specified. The connections of the router belong to two different subnets, and the IP addresses must also be selected accordingly.

### **Setting the interface parameters**

If the parameters of the PROFINET interface have not already been set during the hardware configuration, they can be defined during the network configuration.

Prerequisite: a project with two or more stations is open, the device configuration shows the stations in the network view.

- ▷ Select the PROFINET interface, e.g. by clicking with the mouse in the graphic display or on the corresponding line in the tabular device or network overview.
- ▷ In the *Properties* tab of the Inspector window, select the *Ethernet addresses* section under *General*.
- ▷ If the subnet has not yet been created, click on the *Add new subnet* button to connect the interface to a subnet.
- ▷ Enter the IP address and the subnet mask.
- ▷ Enter whether an IP router is used, and then the router address if applicable.

You can display the addresses of the interfaces using the *Show address labels* symbol in the toolbar of the network view.

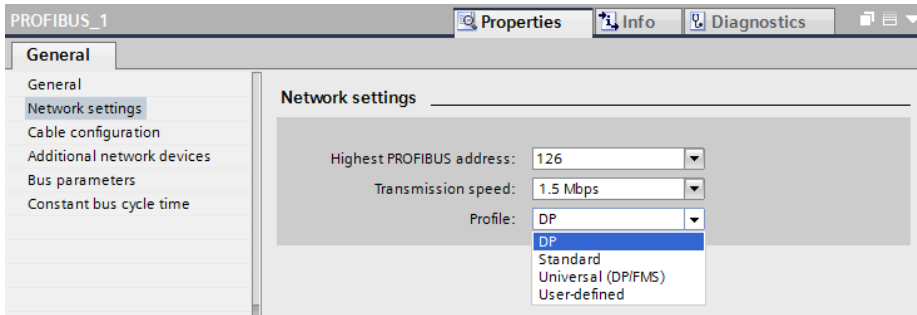
### **3.4.6 Configuring a PROFIBUS subnet**

A CPU 1200, together with a CM 1243-5 module as DP master and with a CM 1242-5 module as DP slave, can be connected to a PROFIBUS subnet.

To configure a PROFIBUS subnet, drag the DP interface of one station to the DP interface of the other station. A PROFIBUS subnet will be created automatically. You can also drag a DP interface to an existing PROFIBUS subnet.

### Setting the properties of a PROFIBUS subnet

To set the properties, select the PROFIBUS subnet and then the *Properties* tab in the inspector window. Under *General* you can assign a different name to the subnet and also change the subnet ID if appropriate. Under *Network settings* you set the highest node address, the transmission speed, and the profile in this subnet. You must observe the technical specifications of the involved modules when doing this (Fig. 3.11).



**Fig. 3.11** Example of network settings on the PROFIBUS

The selectable bus profiles have the following properties:

- ▷ The *DP* bus profile contains the optimized settings of the bus parameters for devices which comply with the requirements of the EN 50170 Volume 2/3, Part 8-2 PROFIBUS standard, for example all SIMATIC S7 DP masters and DP slaves.
- ▷ Compared to the *DP* bus profile, the *Standard* bus profile additionally contains the option for considering non-configured nodes during calculation of the bus parameters, for example nodes from other projects.
- ▷ Select the *Universal* bus profile if the PROFIBUS FMS service is to be used in the PROFIBUS subnet.
- ▷ When using the *User-defined* bus profile, you can set the parameters of the PROFIBUS subnet yourself in the subnet properties. Correct functioning is only guaranteed if the bus parameters are matched to one another. You should only change the default values if you are familiar with how to configure the bus profile for PROFIBUS.

### Setting the properties of a DP interface

To set the properties, select the DP interface and then the *Properties* tab in the inspector window. Under *General* you can set a different name for the interface. Under *PROFIBUS address* you set the node address of the CPU.

Every station on the PROFIBUS DP has a node address (station number) with which it can be addressed unequivocally on the bus. The addresses in a PROFIBUS subnet

can be freely assigned in the range from 1 to 126. The node address 0 is reserved as standard for a programming device, which can be connected temporarily to the PROFIBUS subnet for servicing purposes.

STEP 7 assigns node addresses from 2 upwards as standard in the hardware configuration. It is recommendable to assign the addresses without gaps.

Under *Operating mode* – depending on the CM module – the operating mode is pre-set as DP master or DP slave and cannot be changed. The DP slave is an “intelligent” DP slave, for which the user data interface to the DP master must be configured under *I-slave communication*. Further details can be found in the Section 14.3.3 “Configuring PROFIBUS DP” on page 467 of Chapter 14.3 “PROFIBUS DP”.

### 3.4.7 Configuring an AS-i subnet

A CPU 1200, together with the CM 1243-2 module as AS-Interface master, can be connected to an AS-Interface subnet (Fig. 3.12).

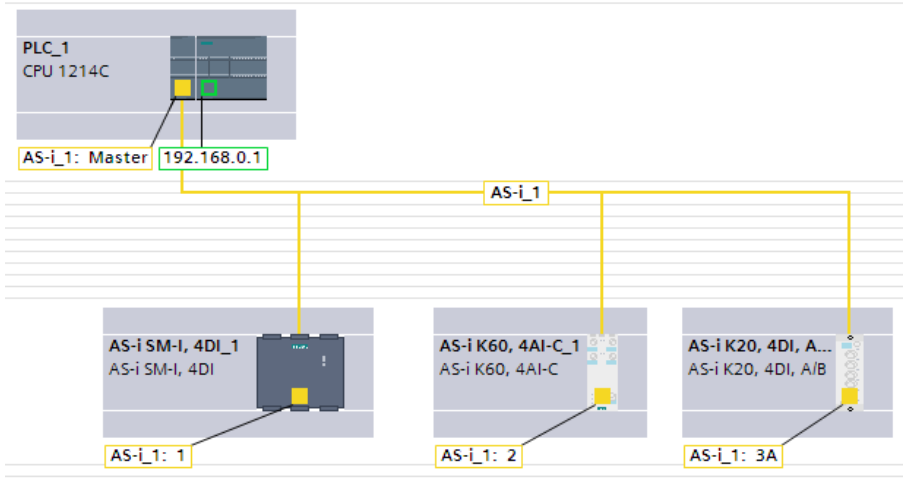


Fig. 3.12 Example of an AS-i master system with CM 1243-2

#### Setting the properties of an AS-i subnet

To set the properties, select the AS-i subnet and then the *Properties* tab in the inspector window. Under *General* you can set a different name for the subnet.

#### Setting the properties of an AS-i master interface

To set the properties, select the AS-i interface in the master and then the *Properties* tab in the inspector window. Under *General*, you can give the interface another name and the connection to the AS-Interface subnet is established under *AS-Interface*.

### **Setting the properties of an AS-i slave interface**

To set the properties, select the AS-i interface in the field device and then the *Properties* tab in the inspector window. Under *General* you can set a different name for the interface. Under *AS-Interface* you set the AS-i subnet used and the node address. The other settings specify operation on the AS-i subnet (see Chapter 14.4 “Actuator/sensor interface” on page 473).

## 4 Variables and data types

### 4.1 Operands and tags

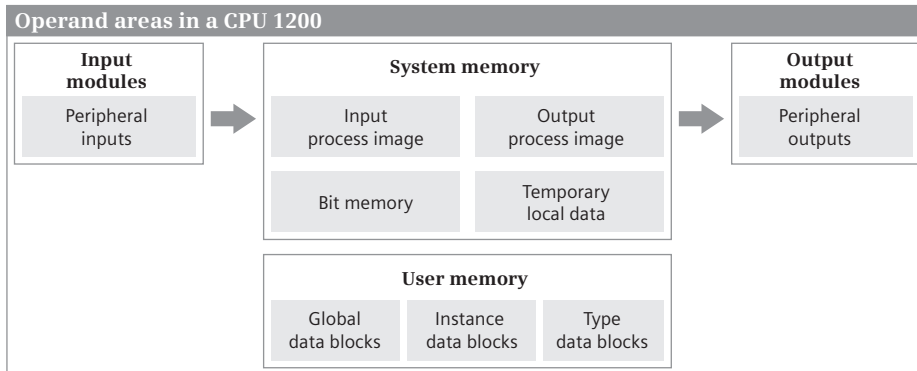
#### 4.1.1 Introduction, overview

In order to control a machine or process, signal states and numerical values are processed. Inputs are scanned, and their signal states linked together in accordance with the control task; the results then control the outputs. It is similar with the numerical values; these are selected, calculated, compared, and saved. The PLC station provides the following memory areas for these variable values (Fig. 4.1):

- ▷ *Peripheral inputs* are the memory areas on the input modules. They constitute the direct interface to the controlled machine, e.g. in order to scan the settings of control elements or sensors.
- ▷ *Inputs* are an image of the peripheral inputs in the CPU's system memory. These are normally processed by the user program when signal states of the machine are to be scanned and linked. The totality of the inputs is the process image input.
- ▷ *Peripheral outputs* are the memory areas on the output modules. They constitute the direct interface to the controlled machine, e.g. in order to control displays, valves or contactors.
- ▷ *Outputs* are an image of the peripheral outputs in the CPU's system memory. These are normally processed by the user program if the results of the control functions are to be output. The totality of the outputs is the process image output.
- ▷ *Bit memories* are a memory area in the CPU's system memory, and are used as a global intermediate memory preferably for binary signals.
- ▷ *Data* refers to memory areas in the user memory for binary signals and numerical values. Data is organized in *data blocks*, which can either be addressed globally from all parts of the user program or which locally manage the data of a function block. They are then called *static local data*.
- ▷ *Temporary local data* refers to memory areas assigned by the CPU to a logic block during processing. The program can temporarily store signal states and numerical values in the block; these lose their validity when processing of the block has been completed.

Access to the signal states and numerical values (the addressing) can be absolute or symbolic. Absolute addressing uses operands such as %I2.5, for example, which comprise the operand ID (I for input in this case) and the memory address (byte 2 bit 5 in this case). If a name and a data type are assigned to an operand (symbolic





**Fig. 4.1** Operand areas in a PLC station

addressing), this is known as a tag. For example, the operand %I2.5 could have the name “Switch on machine” and the data type BOOL.

The *data type* of an operand or tag defines which values the individual bits of the operand or tag have. An individual bit has the data type BOOL, and one refers to a *binary operand* or *binary tag*. Operands and tags with a data width of one byte (8 bits), one word (16 bits) or one doubleword (32 bits) are referred to as *digital operands* or *digital tags*. The data types for digital tags are extremely diverse. For example, the data type INT (integer) refers to a 16-bit wide fixed-point number, the data type CHAR to a character in ASCII code, and the data type ARRAY to a combination of several tags with the same type of data under one tag name.

#### 4.1.2 Operand areas: inputs and outputs

##### Inputs, peripheral inputs

The *peripheral inputs* are the operands on the input modules. They contain the signal states delivered by the machine or process to the programmable controller via the wiring. These signal states are automatically copied by the CPU's operating system into the process image input prior to each processing cycle of the user program (see Chapter 5.6.2 “Process image update” on page 143).

The process image input is located in the CPU's system memory. It contains the operand area *Inputs*. The inputs are used to scan binary signals in the user program and to link their signal states. This means that the input modules are not directly scanned in the normal case, it is the process image input which is scanned.

Access to the peripheral inputs is read-only. Inputs can be read and written. Inputs not occupied by peripheral inputs (the process image is designed for the maximum configuration) can be used as additional intermediate memories like the bit memories.

## Outputs, peripheral outputs

The *peripheral outputs* are the operands on the output modules. They contain the signal states with which the machine or process is controlled via the wiring. The CPU's operating system automatically transfers the signal states of the process image output to the peripheral outputs prior to each processing cycle of the user program (see Chapter 5.6.2 "Process image update" on page 143).

The process image output is located in the CPU's system memory. It contains the operand area *Outputs*. The outputs are used to save the results of the control functions in the user program and to output these to the machine. This means that the output modules are not directly written in the normal case, it is the process image output which is written.

Outputs can be read and written. Outputs not occupied by peripheral outputs (the process image is designed for the maximum configuration) can be used as additional intermediate memories like the bit memories.

Access to the peripheral outputs is write-only. Writing of the peripheral outputs is automatically tracked by the output process image, and therefore there is no difference in the signal states of the outputs and the peripheral outputs with the same address.

## User data area

With SIMATIC S7, every module can have two data areas: A user data area containing input and output data, and a system data area to transfer datasets with diagnostic and parameter assignment data.

When the modules are addressed it is irrelevant whether they are located in central racks or are used as distributed I/O. All modules are arranged equally in the (logical) address volume.

The user data properties of a module depend on the module type. These are digital or analog I/O signals for signal modules or, for example, control and status information for communication modules. The amount of user data is module-specific. There are modules which occupy one, two, or more bytes in this area. Occupation always commences at the relative byte 0. The (logical) address of the relative byte 0 is the module start address, which is defined by the hardware configuration.

The user data represent the peripheral operand area, divided into peripheral inputs and peripheral outputs depending on the transfer direction. If the user data is present in the area of the process images, the CPU automatically takes over data exchange when updating the process images.

## Consistent user data areas

Data consistency means that data is handled together. Transfer of the data block must not be interrupted, and the data source and destination must not be changed during the transfer either. A CPU 1200 receives the data consistency for tags with all elementary data types and system data types. This means a read or write opera-

tion for a tag with one of these data types cannot be interrupted.

For example, if in the main program a write operation is presently occurring and an alarm is triggered that interrupts the main program, the CPU finishes the complete write operation in the main program before the interrupt routine is started.

For a data block consisting of several tags, the interrupt routine can interrupt the transmission of the data block after each tag. Simultaneous access from the main program and the interrupt routine can thus destroy the consistency of the data. The system functions DIS\_AIRT before and EN\_AIRT after the transfer of the data block in the main program can be used to delay starting the interrupt routine until after the data block is transferred. With an ARRAY data field, you can also use the system function UMOVE\_BLK.

A CPU 1200 supports a data consistency of 64 bytes for data transfer via PROFINET IO and PROFIBUS DP. If a data field with more than 64 bytes should be transferred consistently, use the system functions DPRD\_DAT and DPWR\_DAT.

Note that the “normal” updating of process images can be interrupted following each transmitted doubleword.

Diagnostic and parameter data are always transferred consistently in data records, for example diagnostic data with RALRM or parameter data transferred to and from modules with RDREC and WRREC.

### **4.1.3 Operand area bit memory**

The bit memories are, as it were, the “auxiliary contactors” of the controller. They mainly serve to save binary signal states. They can be treated like outputs, but are not connected “to the outside”. The bit memories are located in the CPU's system memory; they are thus always available.

The bit memories are used if intermediate results are to be valid beyond block limits and are to be processed in several blocks.

Bit memories can be read and written without limitation.

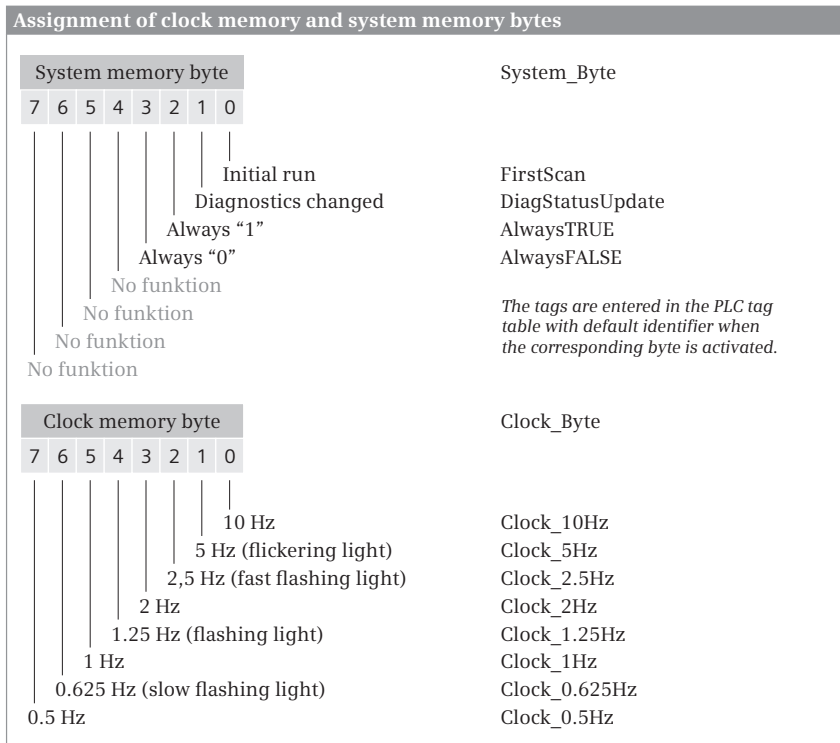
### **Retentive bit memories**

Some of the bit memories can be set “retentive”, i.e. this part retains its signal state even when deenergized. Retentivity always starts at memory byte 0 and ends at the set upper limit. You can set the retentivity in the PLC tag table or in the assignment list. Further information can be found in the Chapter 5.1.4 “Retentive behavior of operands” on page 121.

### **System memory bits**

A CPU 1200 makes a memory byte available whose signal state is controlled by the CPU's operating system. Fig. 4.2 shows the structure of this system memory byte. You define the number of the system memory byte when assigning the CPU parameters. The tags with default identifiers are entered in the PLC tag table when the sys-

tem memory byte is activated. You can change the default identifiers. The individual bits have the following meanings:



**Fig. 4.2** Assignment of the system and clock memory byte

- ▷ Bit 0: Is set to signal state "1" when the main program is processed for the first time following switching-on of the CPU. It has the signal state "0" in all other processing cycles.
- ▷ Bit 1: Is set to signal state "1" if the diagnostics state changes compared to the previous program cycle; otherwise it has signal state "0". During STARTUP and in the first RUN cycle, bit 1 has signal state "0".
- ▷ Bit 2: Is always set to signal state "1" (TRUE); can be used in the program as a binary constant.
- ▷ Bit 3: Is always set to signal state "0" (FALSE); can be used in the program as a binary constant.

*Please note that the system memory byte must not be overwritten by the user program since this could result in incorrect responses in the user program and operating system.*

## Clock memory bits

Many processes in the PLC require a periodic signal. These can be implemented using timer functions (clock generator), cyclic interrupts (time-based program execution) or in a particularly simple manner with clock memory bits.

Clock memory bits are memories whose signal state changes periodically with a pulse-to-pause ratio of 1:1. The clock memory bits are combined in one byte whose individual bits correspond to fixed frequencies (Fig. 4.2). You define the number of the clock memory byte when assigning the CPU parameters. The tags with default identifiers are entered in the PLC tag table when the clock memory byte is activated. You can change the default identifiers.

Note that the clock memories are updated asynchronous to processing of the main program. The clock memories are also updated in the startup program.

*Please note that the clock memory byte must not be overwritten by the user program since this could result in incorrect responses in the user program and operating system.*

### 4.1.4 Operand area data

The operand area *Data* is organized in data blocks which are present in the user memory. Data blocks are available in three versions:

- ▷ *Global data blocks* have a data structure which is defined when configuring the data block.
- ▷ *Instance data blocks* are derived from function blocks. The data structure of an instance data block is defined in the function block. An instance data block contains the values of the block parameters and static local data for calling the function block, for an “instance”. The instance data is local data for the program in the function block.
- ▷ *Type data blocks* are derived from data types. The data structure of a type data block is based on a PLC data type or system data type.

Data blocks are global objects which can be addressed in absolute mode using their number, or symbolically using a name. The name of the data block must be unique on the CPU. The data tags (data operands) within a data block are local data; they are declared when creating the data block (global data blocks), the function block (instance data blocks), or the data type (type data blocks). The name of a data tag must be unique in the data block. In association with the data block, a data tag has the characteristic of a global tag.

Data tags can basically be read and written without limitation; limitations may exist with certain (system) data types. The data tags of a data block with the activated attribute *Data block read-only in device* cannot be overwritten by the program.

The data present in data blocks can be retentive, i.e. it retains its value even when deenergized. Further information can be found in the Chapter 5.1.4 “Retentive behavior of operands” on page 121.

### 4.1.5 Operand area temporary local data

Temporary local data includes operands present in the local data stack (L stack) in the CPU's system memory. Temporary local data is available in each logic block. It serves as a buffer for results that are produced during block processing. Its contents are lost at the end of block processing. The data cannot be accessed by other blocks.

Within the block, the temporary local data can be read and written without limitations. Temporary local data cannot be preallocated with a defined value. In order to use temporary local data for meaningful purposes, it must be written before being read. Temporary local data is addressed symbolically.

The CPU's operating system provides 16 KB of temporary local data for block processing in the startup and in the main program, and 4 KB each for processing of an interrupt program and the error program. When certain organization blocks are called, the operating system transfers start information in the temporary local data. This start information is displayed in the block interface as input parameters.

The number of temporary local data required by a block can be seen in the call structure of the user program. With the project open, select the *Program blocks* folder in the project tree, and then select the *Call structure* command from the shortcut menu. The occupied temporary local data is displayed in the call path and per block in the table which is then output.

## 4.2 Addressing

### 4.2.1 Signal path

By wiring the machine or plant you define which signals are connected to the PLC station, and where (Fig. 4.3). An input signal, e.g. the signal from pushbutton +HP01-S10 with the significance "Switch on motor", is connected to a specific terminal on an input module. You configure the slot in which the module is inserted in the hardware configuration using STEP 7. You use the hardware configuration to set the module start address, from which the addresses of the module channels are derived. This address on the module is simultaneously the address in the process image. This (logical) address is used to address the signals in the user program.

The CPU automatically copies the signal from the input module into the process image input every time before cyclic program processing is started, where it is then addressed as the operand "Input" (e.g. %I5.2). The expression "%I5.2" is the *absolute address*.

You can now give this input a name in that you assign a name corresponding to the significance of this input signal (e.g. "Switch on motor") to the absolute address in the PLC tag table. The expression "Switch on motor" is the *symbolic address*.

The same applies analogously to the output signals. In the hardware configuration you define the slot for the output module and also the module start address. This is then also the address in the process image output. You can also assign a name to this address in the PLC tag table.

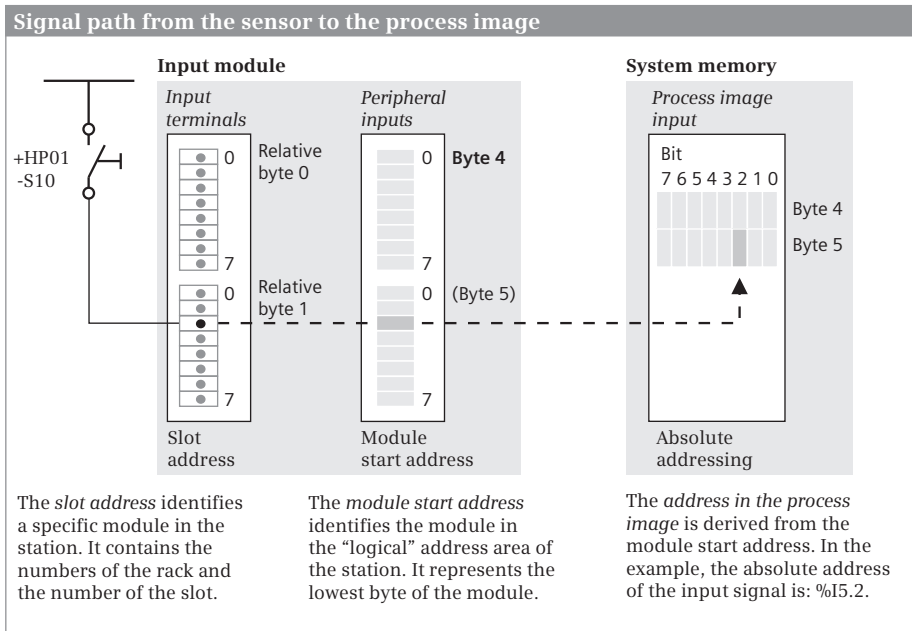


Fig. 4.3 Signal path from sensor to process image

### 4.2.2 Absolute addressing of an operand

During absolute addressing, a signal state or a numerical value is addressed directly using the address in the operand area. The operand, for example %I2.5, contains the operand ID, the byte address and – with binary operands – the bit address separated by a dot. The operand ID contains the operand area and specification of the operand width (Table 4.1). An absolute address is displayed with a preceding percent sign (%).

Data operands can only be addressed in absolute mode if the *Optimized block access* block attribute is not activated in the data block.

The bits in a byte are counted from right to left, starting with zero. Counting is started from the beginning for each byte. Each operand area is organized in bytes. The bytes are counted commencing at the start of the area with zero. With an operand of byte width, the number of the byte is specified as the byte address; with an operand of word width, the number of the least significant byte; and with an operand of doubleword width, the least significant byte number in the doubleword. This principle is explained in Fig. 4.4 using the memory bytes MB 24 to MB 27 as example.

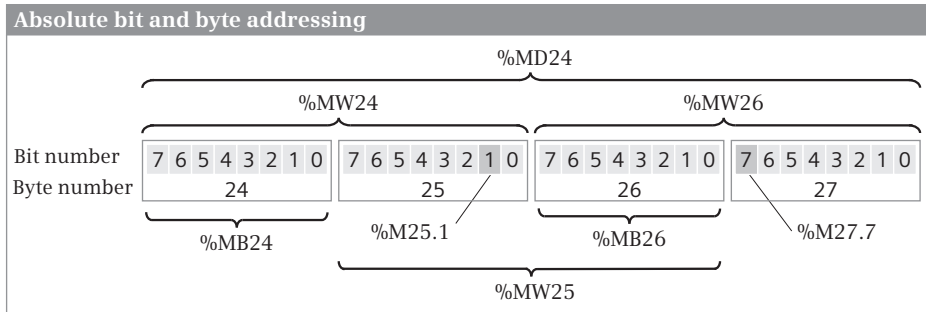
### 4.2.3 Absolute addressing of an operand area

A block parameter with the VARIANT parameter type can also be supplied with an absolutely addressed operand area. A pointer is used here containing the operand area, the start address, the data type, and the number of components (Fig. 4.5).

**Table 4.1** Operand IDs and absolute addressing

Operand area	Operand ID	Bit (1 bit)	Byte (8 bits)	Word (16 bits)	Doubleword (32 bits)
Input	I	%Iy.x	%IBy	%IWy	%IDy
Peripheral input	:P "appended" to the input operand	%Iy.x:P	%IBy:P	%IWy:P	%IDy:P
Output	Q	%Qy.x	%QBy	%QWy	%QDy
Peripheral output	:P "appended" to the output operand	%Qy.x:P	%QBy:P	%QWy:P	%QDy:P
Bit memory	M	%My.x	%MBy	%MWy	%MDy
Data	DB	%DBz.DBXy.x	%DBz.DBBy	%DBz.DBWy	%DBz.DBBy
Temporary local data	Absolute addressing of temporary local data is not possible				

z = data block number, y = byte address, x = bit address



**Fig. 4.4** Bit and byte assignments

**Absolute addressing of operand areas**

Structure of the pointer:

**# Initial operand [space] Data type [space] Number**

Initial operand:	Data type:	Number:
Input: Iy.x Output: Qy.x Bit memory: My.x Data: DBz.DBXy.x x = bit address y = byte address z = data block number	BYTE, WORD, DWORD INT, DINT, REAL, TIME, CHAR	Number of components with the specified data type. The area can comprise a maximum of 65 536 bytes.

**Fig. 4.5** Pointer for the absolute address of an operand area



The start address must be a bit address. The permissible data types are an intersection between the data types of a CPU 1200 and a CPU 300/400, because the pointer is derived from the ANY pointer of a CPU 300/400. The number defines the size of the area; it corresponds to the number of components with the specified data type.

Examples: The pointer *P#M100.0 DWORD 4* comprises an area of four doublewords, beginning with the memory byte MB 100, and with a size of 16 bytes. The pointer *P#DB10.DBX0.0 INT 16* comprises an area of 16 INT tags within data block DB 10, beginning with data byte DB 0, and with a size of 32 bytes.

If a data area is addressed in absolute mode, the *Optimized block access* block attribute must not be activated in the data block. The use of an absolutely addressed operand area makes sense if there is no tag defined for this area.

#### 4.2.4 Symbolic addressing

During symbolic addressing, an operand is assigned an alphanumeric ID (name, symbol) and a data type. This is called a *tag*. For example, the operand %I2.5 could have the name “Switch on machine” and the data type BOOL. The tag “Switch on machine” can then be used in the program instead of the operand %I2.5.

Tag names can be made up of letters, digits, and special characters (except double quotes). No distinction is made between upper and lower case when checking the name.

##### Symbolic addressing of global tags

Global tags can be addressed by any block in the entire program. They are declared in the PLC tag table, and have a unique name within the user program. Global tags are located in the following operand areas: inputs, peripheral inputs, outputs, peripheral outputs, and bit memories.

Global tags must not have a name which has already been assigned to a constant, PLC data type or block. The program editor indicates the name of a global tag in quotation marks.

##### Symbolic addressing of block-local tags

Block-local tags are declared within a block in the interface definition. They have a unique name within the block. The same tag name can be used in another block with another meaning. The operand areas of the block-local tags are

- ▷ the temporary local data in the system memory for all logic blocks,
- ▷ the block parameters for functions (FC) and function blocks (FB),
- ▷ the static local data in the instance data block for function blocks (FB), and
- ▷ the data operands for data blocks (DB).

The program editor indicates the name of a block-local tag with a preceding number sign (#). If the name includes special characters, it is additionally indicated in quotation marks.

### Symbolic addressing of data tags

A data tag is always addressed together with the data block in which it is located. The data tag is thus given the characteristic of a global tag. An example: The tag name “Switch on motor” can be present in both data blocks “Motor 1” and “Motor 2”. The address “Motor 1”.“Switch on motor” addresses a different tag than the address “Motor 2”.“Switch on motor”. The general symbolic address of a data tag is: “*Data block name*”.*Tag name*. All data tags can thus be addressed, even those in an instance data block.

If the instance data of a function block must be addressed, i.e. the block parameters and static local data, only the tag name, along with a numerical prefix, is specified: #*Local data*. For a function block, the instance data are local tags. Further details on static local data can be found in the Section “Static local data” on page 136.

#### 4.2.5 Addressing a tag part

It is possible to address an area within a tag. This area can be a bit, byte, or word.

With the block attribute *IEC check* active, the tag must have data type BYTE, WORD, or DWORD; if the *IEC check* block attribute is deactivated, this can also be a fixed-point data type (USINT, UINT, UDINT, SINT, INT, DINT).

To address a bit in a tag, program *tag name.x<bit number>*, for a byte, program *tag name.b<byte number>*, and for a word, program *tag name.w<word number>*. The numbering begins with zero in each case and must remain within the tag length.

Example: A tag with the name *Temperature* and data type INT is stored in the data block *Store*. The highest-value bit (the sign bit of data type INT) is addressed with “*Store*”.*Temperature.x15*.

#### 4.2.6 Addressing constants

A constant is a fixed numerical value. The notation for a directly entered constant and the value range depend on the required data type (see Table 4.4 on page 96). Constants in floating-point format can be entered in exponential format (e.g. +1.234567E+02) or in decimal format (e.g. -123.4567).

Globally valid constants can be assigned a name in the PLC tag table in the *User constants* tab. Letters, digits, and special characters – except double quotes – are permissible for the name. All elementary data types are permissible.

The name of a constant is unique on the CPU. A name with which a PLC tag, PLC data type, or block has already been identified cannot be assigned to a constant. No distinction is made between upper and lower case when checking the name. The program editor represents a symbolically addressed constant in quotation marks.

#### 4.2.7 Indirect addressing

Indirect addressing allows you to address operands whose address is only determined during runtime. You can thus, for example, let program parts be processed several times in a loop and use a different operand each time.

Since with indirect addressing the addresses are only calculated during runtime, the danger exists that memory areas can be overwritten unintentionally. *The automation system could then react in an unexpected manner! Therefore be extremely careful when using indirect addressing!*

### Indirect addressing of field components

The index of a field component can be a tag whose value is only specified at runtime and which can be changed. Fixed-point data types are permitted as data type of the index tags (SINT, INT, DINT, USINT, UINT, and UDINT). The value of the index tags may range only within the defined limits of the field tags.

Example: If *MeasuredValues* is the name of an ARRAY tag and *Index* is the name of an INT tag, a field component can be addressed with *#MeasuredValues[#Index]*.

Examples of the indirect addressing of a data field can be found in the Chapters 7.6.1 “Jump functions in the ladder logic” on page 242, 8.6.1 “Jump functions in the function block diagram” on page 280, and 9.6.3 “Control statements” in Section “FOR statement” on page 311.

### Indirect addressing with PEEK and POKE (SCL)

PEEK and POKE address a value in an operand area whose address (memory location) can be set during runtime. PEEK reads the value of an operand, POKE writes a value to an operand. POKE\_BLK transfers an indirectly addressed operand area (Fig. 4.6).

The operand areas addressed with PEEK and POKE are inputs, outputs, memory bits, and data blocks. The parameter AREA with the data type BYTE defines the operand area together with the parameter DBNUMBER (Table 4.2). The byte address is at the parameter BYTEOFFSET. For a binary operand, the bit number is added at the parameter BITOFFSET. DBNUMBER, BYTEOFFSET, and BITOFFSET have data type DINT. In the framework of the implicit data type conversion, these parameters can also be supplied with tags that have other fixed-point data types.

PEEK reads the value of a digital operand and makes it available as a function value. The preallocated data type is BYTE; it is used to read one byte. If two bytes should be read, note the instruction PEEK\_WORD; for four bytes, PEEK\_DWORD.

PEEK\_BOOL reads the value of a binary operand and makes it available as a function value.

POKE writes the value specified at the parameter VALUE with data type BYTE, WORD, or DWORD (corresponding to one, two, or four bytes) to the specified operand area.

POKE\_BOOL writes the value (data type BOOL) specified at the parameter VALUE to the specified binary operand.

POKE\_BLK transfers a source operand area, defined with the parameters AREA\_SRC, DBNUMBER\_SRC, and BYTEOFFSET\_SRC, to a operand area defined with the param-

### Indirect addressing of an operand with SCL

PEEK and POKE address an operand whose address is only defined during runtime (indirect addressing).

SCL

```
#variable      := PEEK_Datentyp(
  AREA         := ... ,
  DBNUMBER     := ... ,
  BYTEOFFSET  := ... );

#bitvariable   := PEEK_BOOL(
  AREA         := ... ,
  DBNUMBER     := ... ,
  BYTEOFFSET  := ... ,
  BITOFFSET   := ... );
```

#### Function:

PEEK reads the value of an indirectly addressed digital operand and makes it available as a function value with the specified data type (BYTE, WORD, DWORD).

PEEK\_BOOL reads the value of an indirectly addressed binary operand and makes it available as a function value.

```
POKE (
  AREA         := ... ,
  DBNUMBER     := ... ,
  BYTEOFFSET  := ... ,
  VALUE       := ... );
```

#### Function:

POKE writes the value of the tag specified at the VALUE parameter to an indirectly addressed digital operand.

```
POKE_BOOL (
  AREA         := ... ,
  DBNUMBER     := ... ,
  BYTEOFFSET  := ... ,
  BITOFFSET   := ... ,
  VALUE       := ... );
```

POKE\_BOOL writes the value of the bit tag specified at the VALUE parameter to an indirectly addressed bit operand.

#### Data types:

The operand area is defined at the AREA parameter with the data type BYTE: B#16#81 for inputs, B#16#82 for outputs, B#16#83 for bit memories, and B#16#84 for data.

The value zero is assigned to the DBNUMBER parameter at inputs, outputs, and bit memories, at the operand area data with the data block number.

DBNUMBER, BYTEOFFSET, and BITOFFSET have a fixed-point data type, VALUE has a bit string data type.

### Indirect addressing of a memory area with SCL

POKE\_BLK transfers an indirectly addressed memory area to another indirectly addressed memory area.

SCL

```
POKE_BLK (
  AREA_SRC     := ... ,
  DBNUMBER_SRC := ... ,
  BYTEOFFSET_SRC := ... ,
  AREA_DEST    := ... ,
  DBNUMBER_DEST := ... ,
  BYTEOFFSET_DEST := ... ,
  COUNT       := ... );
```

#### Function:

POKE\_BLK reads the number of bytes specified at the COUNT parameter from the source memory area (SRC) and writes them to the destination memory area (DEST).

#### Data types:

The operand area is defined at the AREA parameter with the data type BYTE: B#16#81 for inputs, B#16#82 for outputs, B#16#83 for bit memories, and B#16#84 for data.

The value zero is assigned to the DBNUMBER\_xxx parameter at inputs, outputs, and bit memories, at the operand area data with the data block number.

DBNUMBER\_xxx, BYTEOFFSET\_xxx, and COUNT have a fixed-point data type.

Fig. 4.6 Indirect addressing of operands with SCL

**Table 4.2** Assignment of the AREA and DBNUMBER parameters

Operand area	Assignment of the AREA parameter	Assignment of the DBNUMBER parameter
Inputs	B#16#81	0
Outputs	B#16#82	0
Bit memory	B#16#83	0
Data	B#16#84	Data block number

eters AREA\_DEST, DBNUMBER\_DEST, and BYTEOFFSET\_DEST. The number of bytes transferred is specified in the COUNT parameter.

Example: The values in a bit memory address area should be deleted. The memory range begins at the address #M\_adr and is #M\_anz bytes long. Both tags are declared with the INT data type.

```
FOR #i := #M_adr TO #M_adr + #M_anz - 1 DO
    POKE(area      := 16#83,
          dbnumber  := 0,
          byteOffset := #i,
          value     := 16#00);
END_FOR;
```

The tag #i with data type INT is used as a loop-control tag in the FOR statement and contains the address of the memory byte that will now be overwritten with 16#00.

## 4.3 General information on data types

### 4.3.1 Overview of data types

Data types define the properties of tags, basically the representation of the contents and the permissible ranges. STEP 7 provides predefined data types. The data types are globally available and can be used in any block. A distinction is made between:

- ▷ Elementary data types which are predefined and can have a width of up to one long word (64 bits)
- ▷ Structured data types, comprising a combination of elementary data types
- ▷ Parameter types for transfer of block parameters to functions and function blocks
- ▷ PLC data types whose structure is defined by the user,
- ▷ System data types with a fixed structure and defined in STEP 7, and
- ▷ Hardware data types defined by the hardware configuration

When linking tags, e.g. when comparing or adding, or when supplying block parameters, the tags involved must have the same or a comparable data type. The block attribute *IEC check* governs the test for a comparable data type: If it is activated, the test is stricter. The block attribute *Optimized block access* can also play a

role in the application of data types. Further details can be found in the section “Attributes” on page 129 of Chapter 5.3.2 “Editing block properties”.

The data types of tags can be converted. This may happen automatically with the implicit data type conversion or with functions for (explicit) data type conversion (see Chapter 11.6 “Conversion functions (Conversion of data type)” on page 376).

### 4.3.2 Implicit data type conversion

The implicit data type conversion occurs automatically when a function is executed if the data types of the involved tags are compatible. It always applies during implicit data type conversion that the bit length of the source data type must not exceed that of the destination data type. For example, a tag with data format DWORD (source data type) cannot be applied to a block parameter which expects data type WORD (destination data type). The programmed bit length must agree with the expected bit length for an in-out parameter. The scope of implicit data type conversion depends on the block attribute *IEC check* (see Table 4.3).

For LAD and FBD, the implicit conversion from a smaller data width to a larger data width is indicated with a gray symbol at the function inputs and outputs. To improve clarity, implicit data type conversion can also be programmed with SCL. The statement is *Source data type\_TO\_Destination data type*, for example

```
#var_word := BYTE_TO_WORD(#var_byte);
```

Where an implicit conversion is possible, the bit pattern of the source tags is transferred unchanged and right-justified to the target tag. Any free bit positions are filled with zeros. For a possible implicit conversion to a floating-point data type, the format is transformed (from “-1” to “-1.0”).

### 4.3.3 Overlaying tags (data type views)

A tag can be “overlaid” by a further data type. It is then possible to address the contents of tags completely or partially using various data types.

You overlay a tag with a further data type using the keyword *AT*. You can overlay several data type definitions over a tag which are differentiated by different names. A default setting with fixed values (initialization) is not possible.

Example: You declare an input parameter in the block interface with *Station* as the name and *STRING[12]* as the data type. You can overlay this input parameter with an additional *STRUCT* data type with the name *Length* and the components *maximal* and *actual*, each with the data type *USINT* (Fig. 4.7). You can now address the current length of the tags *#Station* with *#Length.actual* in the block program.

You can only program the overlaying with further data types in the interface of logic blocks. In addition, the block attribute *Optimized block access* must be deactivated. The memory requirements of the overlaying data type definition must not be greater than the “original” tag (the new data type must “fit” into the tag).

You use a overlaying data type definition like any other tag, but only locally in the block. In the example, the calling block writes a string into the input parameter

**Table 4.3** Implicit data type conversion

to from	BOOL	BYTE	WORD	DWORD	USINT	UINT	UDINT	SINT	INT	DINT	REAL	LREAL	TIME	TOD	DATE	DTL	CHAR	STRING
BOOL																		
BYTE		x	x		O	O	O	KF	O	O								O
WORD				x		O	O		KF	O					O			
DWORD							O			KF			O	O				
USINT		O	O	O		x	x		x	x	x	x						
UINT			O	O			x			x	x	x			O			
UDINT				O								x		O				
SINT		KF							x	x	x	x						
INT		S	KF							x	x	x						
DINT		S	S	KF								x	O					
REAL												x						
LREAL																		
TIME				O						O								
TOD				O			O											
DATE			O			O												
DTL																		
CHAR		O																x
STRING																		

Implicit data type conversion is possible: X Independent of attribute *IEC check*  
 O With deactivated attribute *IEC check*  
 KF only with LAD and FBD with deactivated attribute *IEC check*  
 S only with SCL with deactivated attribute *IEC check*

Interface				
	Name	Data type	Offset	Comment
1	Input			
2	Station	String[12]	0.0	Character string with the name #Station
3	Length	Struct	0.0	
4	maximal	USInt	0.0	maximum length of the string (#Length.maximal)
5	actual	USInt	1.0	actual length of the string (#Length.actual)
6	Character	array[1..12] of Char	2.0	
7	<Add new>			
8	Start	Bool	14.0	(next input parameter)
9	<Add new>			

**Fig. 4.7** Example of declaration of a “overlaid” data type

#Station; the superimposing data type definition as a USINT structure is not accessible to it.

No overlaying with data types is possible for an organization block, because the *Optimized block access* attribute is always active.

Only the tags in the temporary local data can be overlaid with additional data types in an FC block.

In a function block, the tags with additional data types can be overlaid in all declaration subsections of the block interface. If the in-out parameter has an elementary data type, overlaying is only possible with an elementary data type. If the in-out parameter has a structured data type, overlaying is only possible with a structured data type.

## 4.4 Elementary data types

Elementary data types have a width of 1, 8, 16, 32, or 64 bits (Table 4.4). The data types BCD16 and BCD 32 are not data types in the closer sense – they cannot be assigned to a tag; they are only relevant to data type conversion.

### 4.4.1 Bit-serial data types BOOL, BYTE, WORD and DWORD

A tag with data type BOOL represents a bit value (e.g. input %I1.0). The tag can have the value “0” or “1”, or FALSE or TRUE (Fig. 4.8).

A tag with data type BYTE occupies 8 bits. The individual bits have no significance. The notation for constants is 16#00 to 16#FF.

A tag with data type WORD occupies 16 bits. The individual bits have no positional significance. The notation for constants is 16#0000 to 16#FFFF. A constant of word width can also be written as a 16-bit binary number (2#0000\_...\_0000 to 2#1111\_...\_1111) or as a 2×8-bit unsigned decimal number B#(0,0) to B#(255,255).

A tag with data type DWORD occupies 32 bits. The individual bits have no positional significance. The notation for constants is 16#0000\_0000 to 16#FFFF\_FFFF. A constant of doubleword width can also be written as a 32-bit binary number (2#0000\_...\_0000 to 2#1111\_...\_1111) or as a 4×8-bit unsigned decimal number B#(0,0,0,0) to B#(255,255,255,255).

### 4.4.2 BCD-coded numbers BCD16 and BCD32

BCD-coded numbers do not have their own data type. For a BCD number, use the data type WORD or DWORD and enter only the numbers 0 to 9 or 0 and F for the sign in hexadecimal form (16#xxxx or 16#xxxx\_xxxx) (Fig. 4.9).

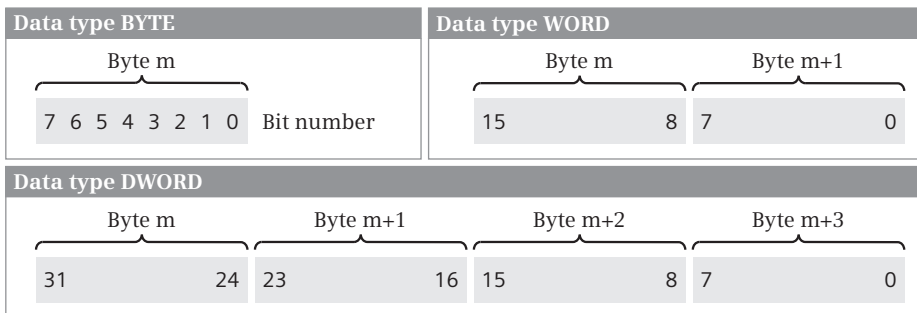
BCD-code numbers are used, for example, in association with the conversion functions. The sign of a BCD-coded number is located in the left-justified (highest) decade. Thus one decade is lost for the number range.



**Table 4.4** Overview of elementary data types

Bit string data types			BCD and floating-point numbers		
BOOL	1 bit	1-bit binary value (0, 1, FALSE, TRUE)	BCD16 <sup>1)</sup>	16 bits	3 decades with sign (-999 ... +999)
BYTE	8 bits	8-bit binary value (B#16#00 ... B#16#FF)	BCD32 <sup>1)</sup>	32 bits	7 decades with sign (-9 999 999 ... +9 999 999)
WORD	16 bits	16-bit binary value (W#16#0000 ... W#16#FFFF)	REAL	32 bits	32-bit floating-point number ( $\pm 1.18 \times 10^{-38}$ ... $\pm 3.40 \times 10^{38}$ )
DWORD	32 bits	32-bit binary value (DW#16#0000 0000 ... DW#16#FFFF FFFF)	LREAL	64 bits	64-bit floating-point number ( $\pm 2.23 \times 10^{-308}$ ... $\pm 1.80 \times 10^{308}$ )
Unsigned fixed-point numbers			Points in time and durations		
USINT	8 bits	Unsigned 8-bit fixed-point number (0 ... 255)	DATE	16 bits	Date of day (D#1990-01-01, ...)
UINT	16 bits	Unsigned 16-bit fixed-point number (0 ... 65 535)	TOD	32 bits	Time of day (TOD#00:00:00.000, ...)
UDINT	32 bits	Unsigned 32-bit fixed-point number (0 ... 4 294 967 296)	TIME	32 bits	Duration in IEC format (T#4h30m, 12 000ms, ...)
Fixed-point numbers with sign			Character		
SINT	8 bits	8-bit fixed-point number (-128 ... +127)	CHAR	8 bits	A character in ASCII code ('a', 'A', '1', ...)
INT	16 bits	16-bit fixed-point number (-32 768 ... +32 767)			
DINT	32 bits	32-bit fixed-point number (-2 147 483 648 ... +2 147 483 647)			

<sup>1)</sup> Not a data type in a narrower sense; only relevant to data type conversion



**Fig. 4.8** Bit assignments of data types BYTE, WORD, and DWORD

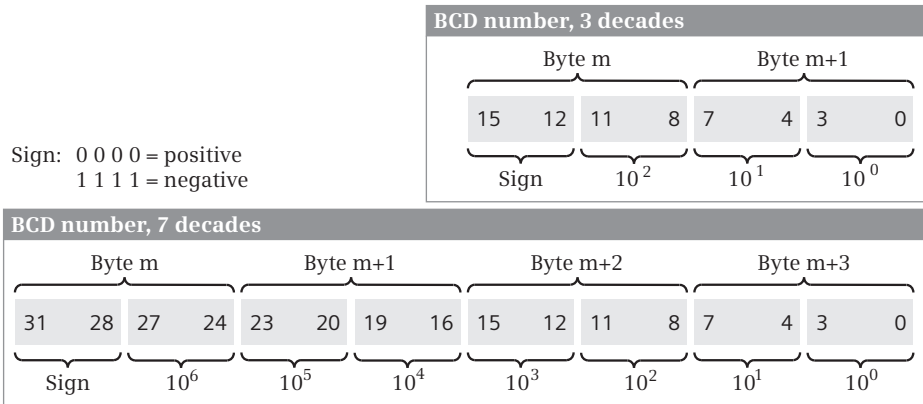


Fig. 4.9 Bit assignments of data types BCD16 and BCD32

In the case of a BCD-coded number present in a 16-bit word, the sign is in the highest decade, where only bit position 15 is relevant. Signal state “0” means that the number is positive. Signal state “1” represents a negative number. The sign does not influence the assignment of the individual decades. An equivalent assignment applies to 32-bit words.

The numerical range available for 16-bit BCD numbers is 0 to ±999, and for 32-bit BCD numbers 0 to ± 9 999 999.

#### 4.4.3 Unsigned fixed-point data types USINT, UINT and UDINT

The data type USINT (unsigned short integer or fixed-point number) occupies one byte. The numerical range extends from  $2^0$  to  $2^8-1$ , i.e. from 0 to 255 or in hexadecimal notation from 00<sub>hex</sub> to FF<sub>hex</sub> (Fig. 4.10).

The data type UINT (unsigned integer or fixed-point number) occupies one word. The numerical range extends from  $2^0$  to  $2^{16}-1$ , i.e. from 0 to 65 535 or in hexadecimal notation from 0000<sub>hex</sub> to FFFF<sub>hex</sub>.

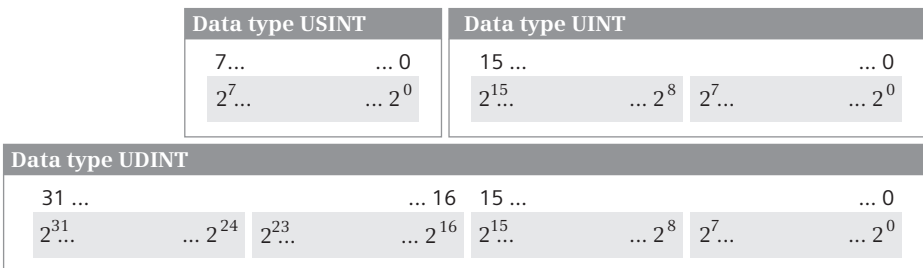


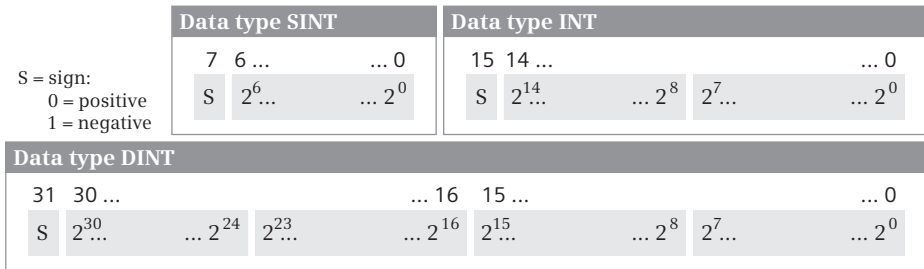
Fig. 4.10 Bit assignments of data types USINT, UINT and UDINT

The data type UDINT (unsigned double integer or fixed-point number) occupies one doubleword. The numerical range extends from  $2^0$  to  $2^{32}-1$ , i.e. from 0 to 4 294 967 295 or in hexadecimal notation from  $0000\ 0000_{\text{hex}}$  to  $FFFF\ FFFF_{\text{hex}}$ .

**4.4.4 Fixed-point data types with sign SINT, INT and DINT**

With the fixed-point data types with sign, the signal state of the highest bit represents the sign (V). Signal state “0” means that the number is positive. Signal state “1” represents a negative number. The representation of a negative number is as a two’s complement.

The data type SINT (short integer or fixed-point number) occupies one byte. The numerical range extends from  $-2^7$  to  $+2^7-1$ , i.e. from -256 to +255 or in hexadecimal notation from  $80_{\text{hex}}$  to  $7F_{\text{hex}}$  (Fig. 4.11).



**Fig. 4.11** Bit assignments of data types SINT, INT and DINT

The data type INT (integer or fixed-point number) occupies one word. The numerical range extends from  $-2^{15}$  to  $+2^{15}-1$ , i.e. from -32 768 to +32 767 or in hexadecimal notation from  $8000_{\text{hex}}$  to  $7FFF_{\text{hex}}$ .

The data type DINT (double integer or fixed-point number) occupies one doubleword. The numerical range extends from  $-2^{31}$  to  $+2^{31}-1$ , i.e. from -2 147 483 648 to +2 147 483 647 or in hexadecimal notation from  $8000\ 0000_{\text{hex}}$  to  $7FFF\ FFFF_{\text{hex}}$ .

**4.4.5 Floating-point data types REAL and LREAL**

A tag with data type REAL or LREAL represents a fractional number which is saved as a floating-point number. A fractional number is entered either as a decimal fraction (e.g. 123.45 or 600.0) or in exponential form (e.g. 12.34e12 corresponding to  $12.34 \cdot 10^{12}$ ). The representation comprises 7 or 17 relevant positions (digits) which are positioned in exponential form in front of the “e” or “E”. The data following “e” or “E” is the exponent to base 10. Conversion of the REAL or LREAL tags into the internal representation of a floating-point number is carried out by the program editor. Table 4.5 shows the internal range limits of a floating-point number.

The CPUs calculate with the full accuracy of the floating-point numbers. The display on the programming device may deviate from the theoretically exact representation as a result of rounding-off errors during the conversion.

**Table 4.5** Internal range limits of a floating-point number

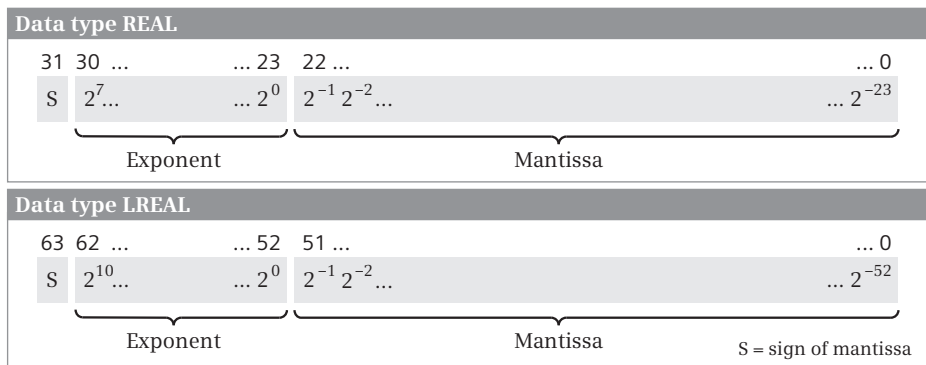
Sign	Exponent with REAL	Exponent with LREAL	Mantissa	Meaning
0	255	2047	Not equal to 0	Not a valid floating-point number (+NaN, Not a Number)
0	255	2047	0	+Inf, Infinity
0	1 ... 254	1 ... 2046	Any	Positive, normalized floating-point number
0	0	0	Not equal to 0	Positive, denormalized floating-point number
0	0	0	0	+ Zero
1	0	0	0	- Zero
1	0	0	Not equal to 0	Negative, denormalized floating-point number
1	1 ... 254	1 ... 2046	Any	Negative, normalized floating-point number
1	255	2047	0	- Inf, Infinity
1	255	2047	Not equal to 0	Not a valid floating-point number (-NaN, Not a Number)

### Data type REAL

The valid range of values of a REAL tag (normalized 32-bit floating-point number) is between the limits:

$$\begin{aligned}
 & -3,402\,823 \times 10^{+38} \text{ to } -1.175\,494 \times 10^{-38} \\
 & \pm 0 \\
 & +1.175\,494 \times 10^{-38} \text{ to } +3.402\,823 \times 10^{+38}
 \end{aligned}$$

A tag with data type REAL consists internally of three components: the sign, the 8-bit exponent to base 2, and the 23-bit mantissa. The sign can have the values “0” (positive) or “1” (negative). The exponent is saved increased by a constant (bias, +127) so that it has a range of values from 0 to 255. The mantissa represents the fractional part. The whole number part of the mantissa is not stored, because it is always equal to 1 within the valid range of values (Fig. 4.12).

**Fig. 4.12** Bit assignments of data types REAL and LREAL

### Data type LREAL

The data type LREAL cannot be used in the PLC tag table, and thus not in association with global tags in the following operand areas: inputs (I), outputs (Q) and bit memories (M).

The valid range of values of a LREAL tag (normalized 64-bit floating-point number) is between the limits:

$$\begin{aligned} & -1,797\,693\,134\,862\,3158 \times 10^{+308} \text{ to } -2.225\,073\,858\,507\,2014 \times 10^{-308} \\ & \pm 0 \\ & +2.225\,073\,858\,507\,2014 \times 10^{-308} \text{ to } +1.797\,693\,134\,862\,3158 \times 10^{+308} \end{aligned}$$

A tag with data type LREAL consists internally of three components: the sign, the 11-bit exponent to base 2 and the 52-bit mantissa. The sign can have the values “0” (positive) or “1” (negative).

The exponent is saved increased by a constant (bias, +1023) so that it has a range of values from 0 to 2047. The mantissa represents the fractional part. The whole number part of the mantissa is not stored, because it is always equal to 1 within the valid range of values.

#### 4.4.6 Data type CHAR

A tag with data type CHAR (character) occupies one byte. The data type CHAR represents a single character which is saved in ASCII format. Example: 'A'. A single character of a tag with the data type STRING has the data type CHAR and can also be used accordingly. Example: If *Author* is the name of the string with the content 'Berger', then the tag *Author[1]* has the value 'B' and the data type CHAR (Fig. 4.13).

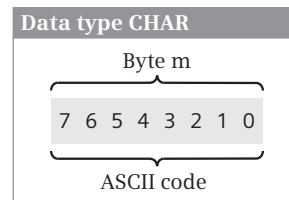


Fig. 4.13 Data type CHAR

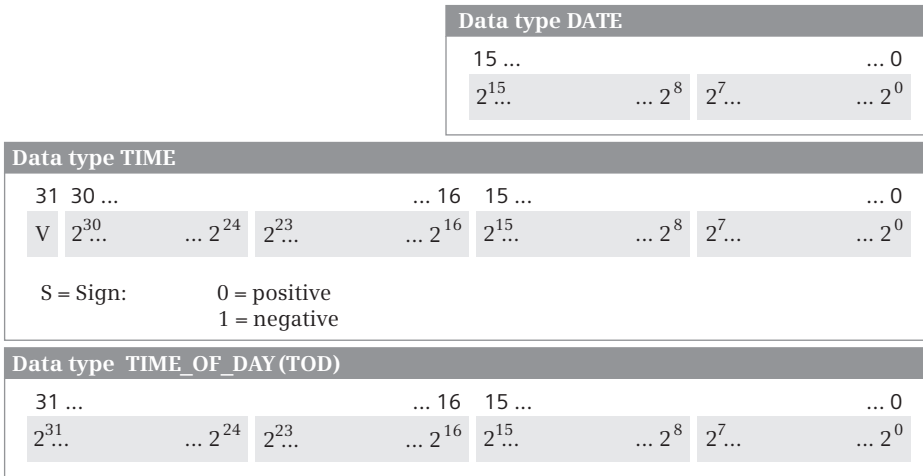
#### 4.4.7 Data type DATE

A tag with data type DATE is saved in a word as an unsigned fixed-point number. The content of the tag corresponds to the number of days since 01.01.1990. The representation contains the year, month, and day, each separated by a dash (Fig. 4.14). Examples:

```
DATE#1990-01-01  (= W#16#0000)
D#2168-12-31    (= W#16#FF62)
```

#### 4.4.8 Data type TIME

A tag with data type TIME occupies a doubleword. The representation contains the data for days (d), hours (h), minutes (m), seconds (s) and milliseconds (ms). Individual time units can be omitted. If more than one time unit is specified, the values for the time units are limited: Days from 0 to 24, hours from 0 to 23, minutes and seconds from 0 to 59, and milliseconds from 0 to 999.



**Fig. 4.14** Bit assignment of data types DATE, TIME, and TIME\_OF\_DAY (TOD)

The content of the tag is interpreted as milliseconds (ms) and saved as a 32-bit fixed-point number with sign. The range of values extends from TIME#-24d20h31m23s648ms (T#-24d20h31m23s648ms) to TIME#24d20h31m23s647ms (T#24d20h31m23s647ms).

#### 4.4.9 TIME\_OF\_DAY (TOD) data type

A tag with data type TIME\_OF\_DAY occupies a doubleword. It contains the number of milliseconds since the beginning of the day (0:00) as an unsigned fixed-point number. The representation contains the data for hours, minutes, and seconds, each separated by a colon. The specification of milliseconds, which follows the seconds and is separated by a dot, can be omitted (Fig. 4.14). Examples:

TIME\_OF\_DAY#00:00:00 (= DW#16#0000\_0000)  
 TOD#23:59:59.999 (= DW#16#0526\_5BFF)

## 4.5 Structured data types

Structured data types consist of a combination of elementary data types under one name (Table 4.6). These data types can only be used locally in the interface of logic blocks and in data blocks; they are not approved for the following operand areas: inputs (I), outputs (Q), and bit memories (M) in the PLC tag table.

### 4.5.1 Data type DTL

The data type DTL (date and time long) contains the date (year, month, day, week-day) and the time (hour, minute, second, nanosecond). Saving in the memory commences at a word limit (at a byte with even address).

**Table 4.6** Overview of structured data types

Data type	Length	Description, example
DTL	12 bytes	Date and time Example: DTL#2009-10-01-11:55:00 (October 1, 2009, five to twelve)
STRING	2+n bytes	A string with n characters. Examples: 'Hans', 'Motor switched off'
ARRAY	Variable	A combination of several equivalent data types. Example: The tag <i>Setpoint</i> has the data type ARRAY[1..32] of INT. The individual components are then: Setpoint[1]; Setpoint[2]; ... ; Setpoint[32]
STRUCT	Variable	A combination of several different data types. Example: The tag <i>Valve</i> has the data type STRUCT. It can then contain the components: Valve.Switch_on; Valve.Switch_off; Valve.Fault; etc.

The components of the data type DTL can also be addressed individually (Fig. 4.15). Example: In a data block “Warehouse”, a DTL tag with the name *Removal* is declared. If you wish to scan the weekday, use the component name “Warehouse”.*Removal.WEEKDAY*.

Data type DTL				
Address	Assignment	Component	Data type	Range
Byte n <sup>1)</sup>	Year	YEAR	UINT	1970 to 2554
Byte n+1				
Byte n+2	Month	MONTH	USINT	1 to 12
Byte n+3	Day	DAY	USINT	1 to 31
Byte n+4	Weekday	WEEKDAY	USINT	1 = Sunday to 7 = Saturday
Byte n+5	Hours	HOUR	USINT	0 to 23
Byte n+6	Minutes	MINUTE	USINT	0 to 59
Byte n+7	Seconds	SECOND	USINT	0 to 59
Byte n+8	Nanoseconds	NANOSECOND	UDINT	0 to 999 999 999
Byte n+9				
Byte n+10				
Byte n+11				

<sup>1)</sup> n = even

**Fig. 4.15** Structure of data type DTL

#### 4.5.2 Data type STRING

The data type STRING represents a string consisting of two bytes for the length data and up to 254 bytes for the characters in ASCII code. Saving in the memory com-

Data type STRING			
Byte no.		Data type	Range
n <sup>1)</sup>	Maximum length	USINT	0 to 254 (k)
n+1	Actual length	USINT	0 to 254 (m, m ≤ k)
n+2	1st character	CHAR	} Actual length (m) } } Maximum length (k)
n+3	2nd character	CHAR	
...	...	CHAR	
n+m+1	m-th character	CHAR	
...	...	CHAR	
n+k+1	...	CHAR	

<sup>1)</sup> n = even

**Fig. 4.16** Structure of STRING data type

mences at a byte with even address. The program editor reserves an even number of bytes for a string.

When creating a STRING tag, its maximum length is defined in square brackets. The current length is entered for the default setting or when processing the string (the actually used length of the string = number of valid characters). The maximum length is present in the first byte of the string, the second byte contains the actual length; this is followed by the characters in ASCII format (Fig. 4.16).

Example: The tag *Machine* is to be defined with a maximum length of 12 characters and should have 'Drill' as the default setting.

Name	Data type	Default value
Machine	STRING[12]	'Drill'

The first byte of the tag then has the value 12, the second byte the value 6, the third byte the character 'B' etc.

When declaring a STRING tag as the block parameter of a function (FC), only a length of 254 can be assigned. If no length is specified in the declaration of a STRING tag, the program editor applies the standard length of 254 characters.

A STRING tag cannot be assigned a default value when declared in the temporary local data. The content of the tag is then undefined. Prior to use, the tag must first be assigned a valid content (per program), for example with the function S\_CONV.

A constant with data type STRING is written with single quotation marks, for example 'Hans Berger'. If the single quotation mark is to be a character of the tag, it must be preceded by a dollar sign (\$) .

The characters in a STRING tag can also be addressed individually. The first character (the third byte) is accessed using *Tag\_name[1]*, the n-th character using



*Tag\_name[n]*. The individual components have the data type CHAR. In the above example, the tag *Machine[3]* has the character 'h'.

Special functions are available for processing STRING tags, for example to separate a partial string or to combine two STRING tags into a single one (see Chapter 11.9 “Processing of strings (Data type STRING)” on page 398).

### 4.5.3 Data type ARRAY

The data type ARRAY represents a data structure comprising a fixed number of components with the same data type (Fig. 4.17). For the components, all data types except ARRAY are permissible.

A tag with data type ARRAY commences at a word limit (at a byte with even address). Components with data type BOOL commence in the least significant bit; components with data type BYTE and CHAR in the right byte. The individual components are listed consecutively. The program editor reserves an even number of bytes for a field tag.

The data type ARRAY can have up to 65 536 components. When creating an ARRAY tag, the number range of the components is specified in square brackets, and the data type following the keyword OF. The number range extends from -32 768 to 32 767. The lower range value must be smaller than the upper value.

Example: A tag with the name *Measured\_value* is to have 16 components of data type INT which are numbered commencing with 1.

Name	Data type	Start value
Measured_value	ARRAY[1..16] OF INT	
Measured_value[1]	INT	0
Measured_value[2]	INT	0
...		
Measured_value[16]	INT	0

The components of an ARRAY tag can also be addressed individually, and can be handled like tags with the same data type. For example, the component *Measured\_value[10]* on a block parameter can be created with the data type INT. Addressing with a variable index is also possible: *Measured\_value[#var\_index]* (see Chapter 4.2.7 “Indirect addressing” Section “Indirect addressing of field components” on page 90).

### 4.5.4 Data type STRUCT

The STRUCT data type represents a data structure comprising a fixed number of components with different data types (Fig. 4.18). For the individual components, all data types are permissible.

A tag with data type STRUCT commences at a word limit (at a byte with even address), followed by the components in the declared sequence. Components with the

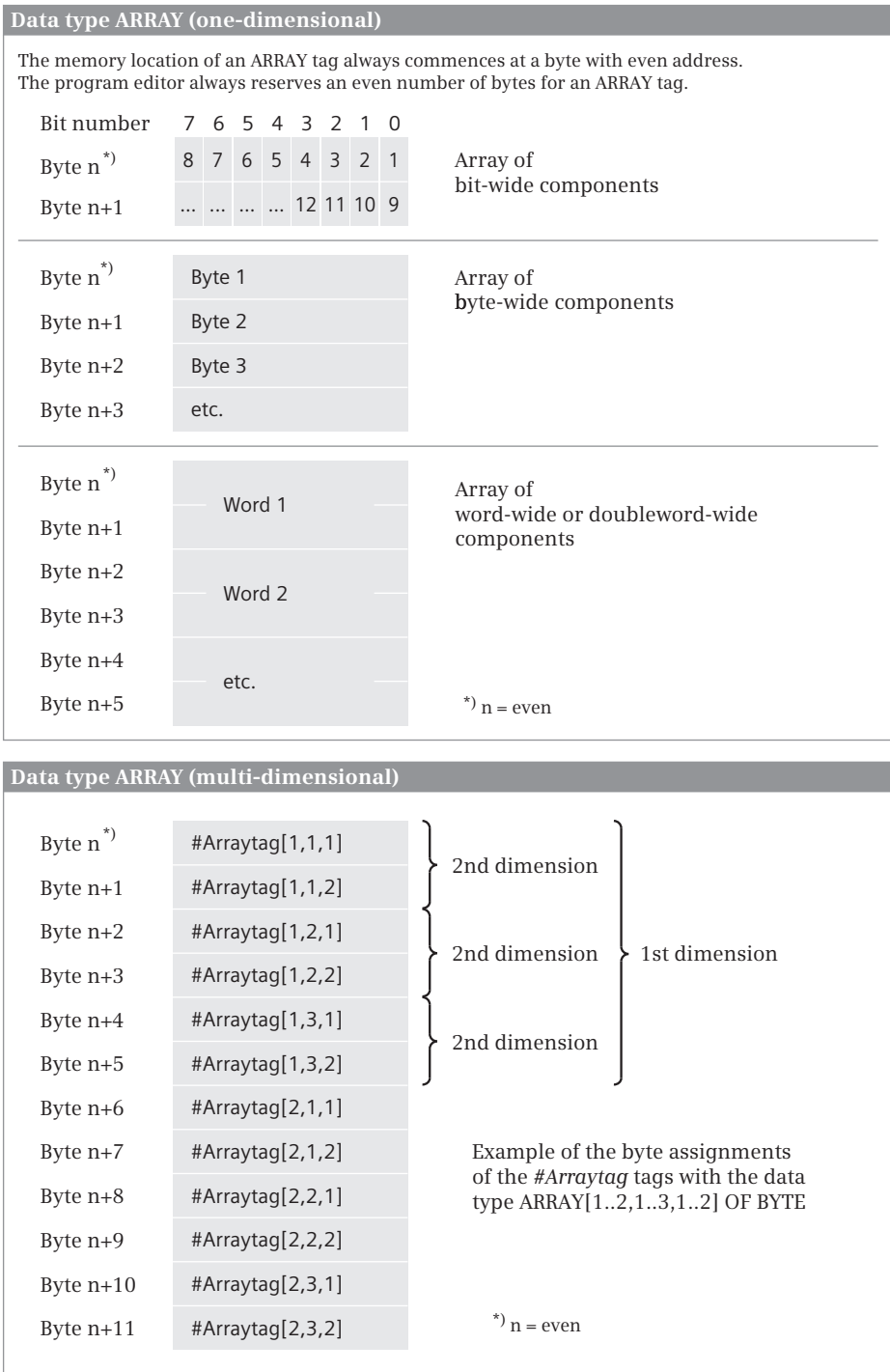
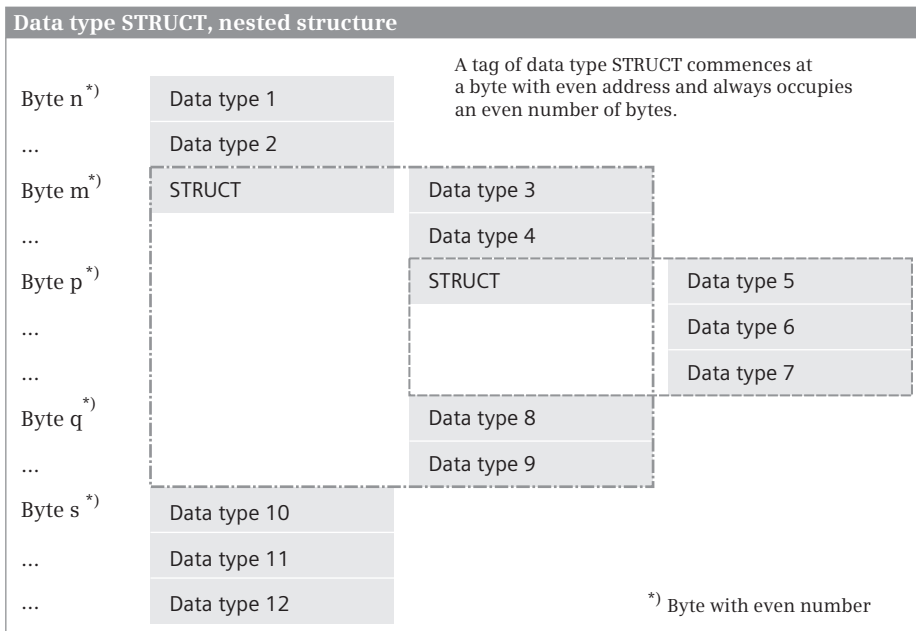
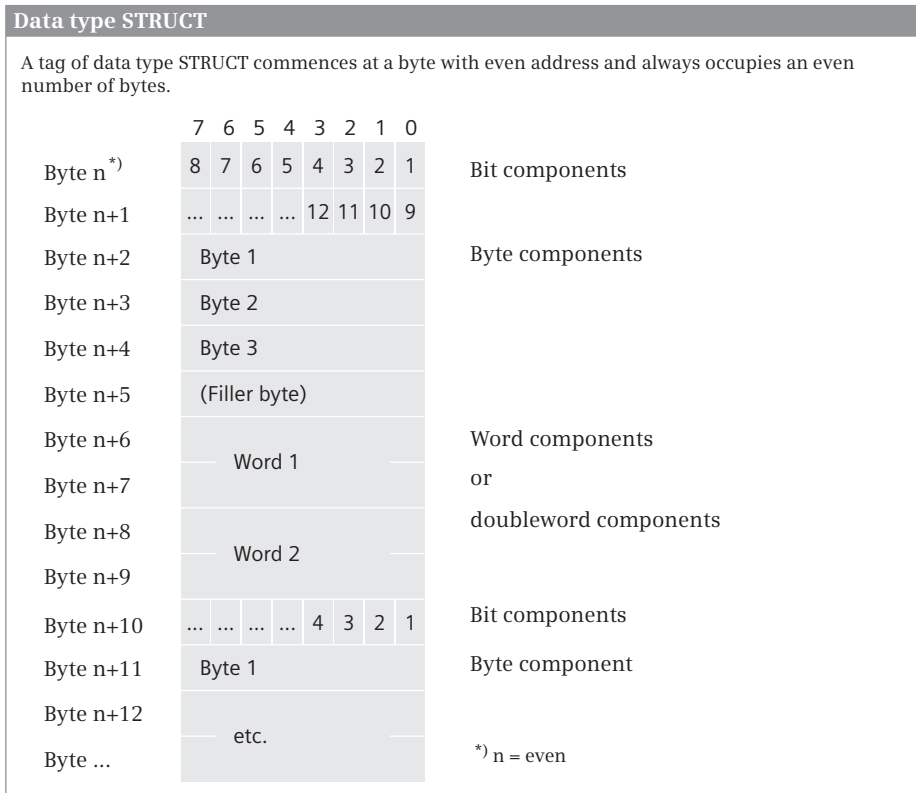


Fig. 4.17 Structure of data type ARRAY



**Fig. 4.18** Structure of data type STRUCT

data type BOOL commence in the least significant bit of the next vacant byte, components with the data type BYTE or CHAR in the next vacant byte. Components with other data types commence at a word limit. The program editor reserves an even number of bytes for a STRUCT tag.

When declaring a STRUCT tag, the tag name with the STRUCT data type is specified first, followed underneath by the individual components with their own data type.

Example: A tag with the name *Fan* is to comprise four components: “Switch on fan” (BOOL), “Switch off fan” (BOOL), “Speed” (INT) and “Delay” (TIME).

Name	Data type	Start value
Fan	STRUCT	
Switch on fan	BOOL	false
Switch off fan	BOOL	false
Speed	INT	0
Delay	TIME	T#0ms

A component of a STRUCT tag can also be addressed individually by positioning the name of the structure, separated by a dot, in front of the component name. A STRUCT component can be handled like a tag with the same data type. For example, the component *Fan.Speed* on a block parameter can be created with the data type INT.

## 4.6 Parameter types

The parameter types are additional data types for block parameters. The parameter types comprise, for example, the data types for the transfer of IEC timer functions and IEC counter functions, the transfer of any tags, and the data type for the function value of a function. PLC data types, system data types, and hardware data types can be used for the block parameters.

### 4.6.1 Parameter types for IEC timer functions

The following data types are available for the transfer of IEC timer functions to the block interface:

Timer function	Data type of the duration	IEC data type
Pulse generation	TIME	TP_TIME
ON delay	TIME	TON_TIME
OFF delay	TIME	TOF_TIME
Accumulating ON delay	TIME	TONR_TIME

The structure of the data types corresponds to the structure of the system data type IEC\_TIMER (see Chapter 4.8.1 “IEC\_TIMER system data type” on page 110).

The data types can be used in the declaration sections Input (input parameters), InOut (in-out parameters), and Static (static local data). If an IEC timer function is transferred as input parameter, its components can only be scanned. You supply a block parameter with the data type of an IEC timer function with the name of the instance data, either with the data block if the call is created as a single instance, or with the instance name if the call is created as a local instance in a function block.

The data types for IEC timer functions can also be used in PLC data types.

### 4.6.2 Parameter types for IEC counter functions

Depending on the counter type and the data type of the count value, there are the following data types for the transfer of IEC counter functions to the block interface:

Counter type	Data type of the count value	IEC data type
Up counter	SINT INT DINT	CTU_SINT CTU_INT CTU_DINT
Down counter	SINT INT DINT	CTD_SINT CTD_INT CTD_DINT
Up/down counter	SINT INT DINT	CTUD_SINT CTUD_INT CTUD_DINT

The structure of the data types corresponds to the structure of the system data type IEC\_xCOUNTER (see Chapter 4.8.2 “IEC\_COUNTER system data type” on page 112).

The data types can be used in the declaration sections Input (input parameters), InOut (in-out parameters), and Static (static local data). If an IEC counter function is transferred as input parameter, its components can only be scanned. You supply a block parameter with the data type of an IEC counter function with the name of the instance data, either with the data block if the call is created as a single instance, or with the instance name if the call is created as a local instance in a function block.

The data types for IEC counter functions can also be used in PLC data types.

### 4.6.3 Parameter type VARIANT

A block parameter with data type VARIANT contains a pointer to a tag or a data area. Tags of all data types are approved for a block parameter of type VARIANT. The tags (operands or data types) which can be connected to the block parameter or which are meaningful are defined by the program within the called block.

In the programming languages LAD and FBD, a block parameter can be assigned this parameter type. The only type of further processing possible is the “passing-on” to block parameters of called blocks.

#### 4.6.4 Parameter type VOID

The VOID parameter type (= without type) is used for the value of functions FC if the function value is not to be displayed. Further details can be found in Section 5.3.5 “Block interface” on page 133.

## 4.7 PLC data types

A PLC data type is one with its own name. It is structured as the STRUCT data type, i.e. it consists of individual components, usually with different data types. You can use a PLC data type if you wish to assign a name to a data structure, for example because you frequently use the data structure in your program. A PLC data type is valid throughout the CPU (global).

### Programming a PLC data type

All PLC data types are combined in the project tree under a PLC station in the *PLC data types* folder. To create a PLC data type, double-click on *Add new data type* in the *PLC data types* folder. Enter the individual components of the PLC data type in sequence in the declaration table with name, data type, default value, and comment (Fig. 4.19).

You can change the standard name *User data type\_n*, where *n* is a consecutive number: Select the PLC data type in the project tree with the right mouse button, select the *Properties* command from the shortcut menu, and enter the new name under *General*. The name must not already be assigned to a PLC tag, a constant, or a block. The operand ID is UDT (user-defined data type), the number is assigned by the program editor.

The screenshot shows a window titled 'Book1200 > Examples [CPU 1214C DC/DC/DC] > PLC data types > User\_data\_type\_1'. The main area displays a table for 'User\_data\_type\_1' with the following columns: Name, Data type, Default value, and Comment. The table contains five rows of data and a sixth row for adding new elements.

	Name	Data type	Default value	Comment
1	Name	String[12]		Name of the station
2	Number	Int	0	Number of the station
3	On	Bool	false	Switch-on signal
4	Off	Bool	false	Switch-off signal
5	Stop	Bool	false	Halt signal
6	<Add new>			

Fig. 4.19 Example of programming a PLC data type

## Using a PLC data type

A PLC data type can be assigned to any tag which is present in a global data block or in the interface of a logic block. The default setting for the PLC data type can be changed. You then address the individual components of the tag using `#var_name.comp_name` (Fig. 4.20).

The screenshot shows the 'Interface' table in the STEP 7 software. The table lists the components of a PLC data type named 'Station\_1'. Each component has a name, a data type, an offset, and a comment.

	Name	Data type	Offset	Comment
15	Station_1	*User_data_type_1*	16.0	Example of a PLC data type
16	Name	String[12]	0.0	Name of the station
17	Number	Int	14.0	Number of the station
18	On	Bool	16.0	Switch-on signal
19	Off	Bool	16.1	Switch-off signal
20	Stop	Bool	16.2	Halt signal
21	Start_1	Bool	34.0	(next input parameter)
22	<Add new>			

Fig. 4.20 Example of application of a PLC data type

With a PLC data type as the basis, you can also generate a data block: In the project tree, double-click on *Add new block* in the *Program blocks* folder. Click on the *Data block* button in the *Add new block* window, and select the PLC data type from the *Type* drop-down list. The data structure of this type data block is then defined by the PLC data type and can no longer be changed. The default setting is imported by the PLC data type and can be changed.

## 4.8 System data types

System data types (SDT) are predefined data types which – like the data type `STRUCT` – consist of a fixed number of components each with different elementary data types. System data types are delivered together with STEP 7 and cannot be modified.

The system data types can only be used together with certain functions or statements. Table 4.7 shows a selection of system data types.

### 4.8.1 IEC\_TIMER system data type

The instance data of an IEC timer function is structured according to the system data type `IEC_TIMER`. If you use the instructions `TP`, `TON`, `TOF`, or `TONR`, the program editor – depending on the specification *Single instance* or *Multi-instance* –

**Table 4.7** Selection of system data types (SDT)

Name	SDT number	Contains the data structure for:	Is used by:
IEC_TIMER	SDT 31	A time function	Time functions TP, TON, TONR, TOF
IEC_SCOUNTER	SDT 69	A counter with data type SINT	Counter functions CU, CD, CUD
IEC_COUNTER	SDT 30	A counter with data type INT	Counter functions CU, CD, CUD
IEC_DCOUNTER	SDT 70	A counter with data type DINT	Counter functions CU, CD, CUD
IEC_USCOUNTER	SDT 73	A counter with data type USINT	Counter functions CU, CD, CUD
IEC_UCOUNTER	SDT 72	A counter with data type UINT	Counter functions CU, CD, CUD
IEC_UDCOUNTER	SDT 74	A counter with data type UDINT	Counter functions CU, CD, CUD
Conditions	SDT 513	The data transfer	PtP function RCV_GFG
TADDR_Param	SDT 514	The addressing of the communication partner in UDP	Open User Communication
TCON_Param	SDT 515	The connection descriptions	Open User Communication
ErrorStruct		The local error handling	GetError and GetErrorID

creates a data block or a local instance with the data type IEC\_TIMER. You can also create a type data block or a local instance with the data type IEC\_TIMER yourself. IEC\_TIMER consists of the components shown in Table 4.8.

**Table 4.8** Structure of the system data types IEC\_TIMER and IEC\_xCounter

IEC_TIMER			IEC_xCOUNTER		
Name	Data type	Designation	Name	Data type	Designation
ST	TIME	(Internal)	CU	BOOL	Up counter input
PT	TIME	Preset time	CD	BOOL	Down counter input
			D	BOOL	Reset input
ET	TIME	Elapsed time	LD	BOOL	Load input
			QU	BOOL	Status up
RU	BOOL	(Internal)	QD	BOOL	Status down
IN	BOOL	Start input	PV	*)	Default value
Q	BOOL	Time status	CV	*)	Count value

\*) Depends on system data type (SINT, INT, DINT, USINT, UINT, UDINT)

You can address the individual components of the data type as usual as the data tag *"Data block".component* or as the local tag *#LocalInstance.component*. Example: You create a local instance with the name *#Duration* and the data type IEC\_TIMER. You can then scan the time status with *#Duration.Q*.



### 4.8.2 IEC\_COUNTER system data type

The instance data of an IEC counter function is structured depending on the data type of the counter value according to the system data types C\_SCOUNTER (SINT data type), IEC\_COUNTER (INT), IEC\_DCOUNTER (DINT), IEC\_US\_COUNTER (USINT), IEC\_UCOUNTER (UINT), and IEC\_UDCOUNTER (UDINT).

If you use one of the statements CTU, CTD, or CTUD, the program editor – depending on the specification *Single instance* or *Multi-instance* – creates a data block or a local instance with the data type IEC\_xCOUNTER. You can also create a type data block or a local instance with the data type IEC\_xCOUNTER yourself. IEC\_xCOUNTER consists of the components shown in Table 4.8.

You can address the individual components of the data type as usual as the data tag “*Data block*”.*component* or as the local tag *#LocalInstance.component*. Example: You create a local instance with the name *Number* and the data type IEC\_COUNTER. You can then scan the count value with *#Number.CV*.

### 4.8.3 TCON\_Param data type

The data type TCON\_Param contains the structure of the connection data either for the communication connection to the partner device (TCP native and ISO-on-TCP protocols) or for the connection to the communication access point of the local operating system (UDP protocol). You require a data block with this structure for each connection (Table 4.9).

### 4.8.4 TADDR\_Param data type

The data type TADDR\_Param contains the structure of the remote partner's address information when using the UDP protocol. With this data structure you configure a data area in a data block which contains the addresses of the receiver stations and parameterize this data area at the ADDR parameter of the send block TUSEND. At the receive block TURCV you parameterize a data area with this structure at the ADDR parameter which accommodates the addresses of the transmitting station (Table 4.10).

### 4.8.5 Data type ErrorStruct

The data type ErrorStruct is a data structure with predefined assignment. The data type is used by the functions for error evaluation GetError and GetErrorID. Information concerning errors is output with this structure (Table 4.11). A tag with data type ErrorStruct commences at a word limit (at a byte with even address).

Additional information is output depending on the assignment of the structure component MODE (Table 4.12). When declaring an ErrorStruct tag, the data type is selected from the drop-down list. The components can also be addressed individually: *Tag\_name.Component\_name*.

**Table 4.9** Structure of data type TCON\_Param

Byte	Parameter	Data type	Description
0 and 1	block_length	WORD	Length of TCON_PAR (fixed at 64 bytes)
2 and 3	id	WORD	Connection ID
4	connection_type	BYTE	Protocol variant B#16#11: TCP B#16#12: ISO-on-TCP B#16#13: TCP (compatibility mode)  With UDP: B#16#13
5	active_est	BOOL	Type of connection establishment FALSE: Passive connection establishment TRUE: Active connection establishment  With UDP: FALSE
6	local_device_id	BYTE	Communication module ID
7	local_tsap_id_len	BYTE	Length of parameter local_tsap_id
8	rem_subnet_id_len	BYTE	B#16#00
9	rem_staddr_len	BYTE	Length of parameter rem_staddr  With UDP: B#16#00
10	rem_tsap_id_len	BYTE	Length of parameter rem_tsap_id  With UDP: B#16#00
11	next_staddr_len	BYTE	Length of parameter next_staddr  With UDP: B#16#00
12to27	local_tsap_id	ARRAY [1..16] OF BYTE	Depending on connection: local port number or local TSAP ID  With UDP: local port number
28to33	rem_subnet_id	ARRAY [1..6] OF BYTE	B#16#00
34to39	rem_staddr	ARRAY [1..6] OF BYTE	IP address of remote partner  With UDP: B#16#00
40to55	rem_tsap_id	ARRAY [1..16] OF BYTE	Depending on connection: remote port number or remote TSAP ID  With UDP: B#16#00
56to61	rem_staddr	ARRAY [1..6] OF BYTE	CP slot  With UDP: B#16#00
62 and 63	spare	WORD	W#16#0000

**Table 4.10** Structure of data type TADDR\_Param

Byte	Parameter	Data type	Description
0to3	rem_ip_addr	ARRAY [1..4] OF BYTE	IP address of remote partner
4 and 5	rem_port_nr	ARRAY [1..2] OF BYTE	Port no. of remote partner
6 and 7	spare	ARRAY [1..2] OF BYTE	B#16#00

**Table 4.11** Structure of ErrorStruct data type

Name	Data type	Note, assignment
ERROR_ID	WORD	Error ID (see text)
FLAGS	BYTE	16#00
REACTION	BYTE	Reactions to error 16#00: none, no writing (write error) 16#01: replace, read a zero (read error) 16#02: skip statement (system error)
CODE_ADDRESS	CREF	Type of block in which the error occurred 16#01: OB, 16#02: FC, 16#03: FB Number of block in which the error occurred Internal memory address at which the error occurred
BLOCK_TYPE	BYTE	
CODE_BLOCK_NUMBER	UINT	
OFFSET	UDINT	
MODE	BYTE	Assignment for the significance of the supplied data (A) to (E) (see text)
OPERAND_NUMBER	UINT	Internal operand number of operation
POINTER_NUMBER_LOCATION	UINT	Internal pointer address of operation (A) (see text)
SLOT_NUMBER_SCOPE	UINT	Internal address in memory (B) (see text)
DATA_ADDRESS	NREF	Addressed memory area on occurrence of error (C) (see text) Number of data block on occurrence of error, otherwise zero (D) (see text) Bit offset on occurrence of error (E) (see text)
AREA	BYTE	
DB_NUMBER	UINT	
OFFSET	UDINT	

**Table 4.12** Information output depending on access type MODE

MODE	(A)	(B)	(C)	(D)	(I)
16#00	–	–	–	–	–
16#01	–	–	–	–	OFFSET
16#02	–	–	AREA	–	–
16#03	LOCATION	SCOPE	–	NUMBER	–
16#04	–	–	AREA	–	OFFSET
16#05	–	–	AREA	DB_NUMBER	OFFSET
16#06	POINTER_NUMBER_LOCATION	SLOT_NUMBER_SCOPE	AREA	DB_NUMBER	OFFSET
16#07	POINTER_NUMBER_LOCATION	SLOT_NUMBER_SCOPE	AREA	DB_NUMBER	OFFSET
<b>Memory area</b>		<b>Assignment of AREA component</b>			
System memory (temporary local data)		16#40...4E, 16#86, 16#87, 16#8E, 16#8F, 16#C0...CE			
Process image input (I)		16#81			
Process image output (Q)		16#82			
Bit memory (M)		16#83			
Data operand (DB)		16#84, 16#85, 16#8A, 16#8B			

The assignment of the `ERROR_ID` and handling of the error evaluation is described in Section 5.8.4 “Local error handling” on page 169.

#### 4.8.6 TimeTransformationRule data type

The data type *TimeTransformationRule* contains the correction values for setting the local time on the CPU (see Chapter 5.6.6 “Time” on page 148). It is required in conjunction with the `SET_TIMEZONE` function to calculate the local time from the module time (time data of the real-time clock) and to make the switch between daylight saving and standard time (Table 4.13).

**Table 4.13** Data structure for TimeTransformationRule

Byte	Name	Data type	Note
0	Bias	INT	Correction value for the time zone to UTC in minutes
2	DaylightBias	INT	Correction value daylight saving/standard time in minutes
4	DaylightStartMonth	USINT	Beginning of daylight saving time : Month Week ( 1 = first in month) Weekday (1 = Sunday) Hour Minute
5	DaylightStartWeek	USINT	
6	DaylightStartWeekday	USINT	
7	DaylightStartHour	USINT	
8	DaylightStartMinute	USINT	
9	StandardStartMonth	USINT	Beginning of standard time : Month Week ( 1 = first in month) Weekday (1 = Sunday) Hour Minute
10	StandardStartWeek	USINT	
11	StandardStartWeekday	USINT	
12	StandardStartHour	USINT	
13	StandardStartMinute	USINT	
14	TimeZoneName	STRING[80]	Name of time zone, e.g. '(GMT +01:00) Amsterdam, Berlin, Bern, Rome, ...'

## 4.9 Hardware data types

Hardware data types refer to all those which can accept the constants in the default tag table in the *System constants* tab. These constants are used to address hardware and software objects in the program. The data type and the value are predefined, the name can be changed in the object properties.

Fig. 4.21 shows the *System constants* tab with a selection of hardware data types.

Book1200 ▸ Examples [CPU 1214C DC/DC/DC] ▸ PLC tags ▸ Default tag table [14]				
Tags   User constants   System constants				
Default tag table				
	Name	Data type	Value	Comment
1	OB_Main	OB_PCYLE	1	
2	PROFINET-Schnittstelle[PN]	Hw_Interface	64	
3	HSC_1[HSC]	Hw_Hsc	1	
4	HSC_2[HSC]	Hw_Hsc	2	
5	HSC_3[HSC]	Hw_Hsc	3	
6	HSC_4[HSC]	Hw_Hsc	4	
7	HSC_5[HSC]	Hw_Hsc	5	
8	HSC_6[HSC]	Hw_Hsc	6	
9	AI2[AI]	Hw_SubModule	7	
10	DI14/DO10[DI/DO]	Hw_SubModule	8	
11	Pulse_1[PTO/PWM]	Hw_Pwm	9	
12	Pulse_2[PTO/PWM]	Hw_Pwm	10	
13	Pulse_3[PTO/PWM]	Hw_Pwm	257	
14	Pulse_4[PTO/PWM]	Hw_Pwm	258	

Fig. 4.21 Examples of hardware data types

## 5 Edit user program

### 5.1 Operating modes

A CPU 1200 recognizes the following operating modes:

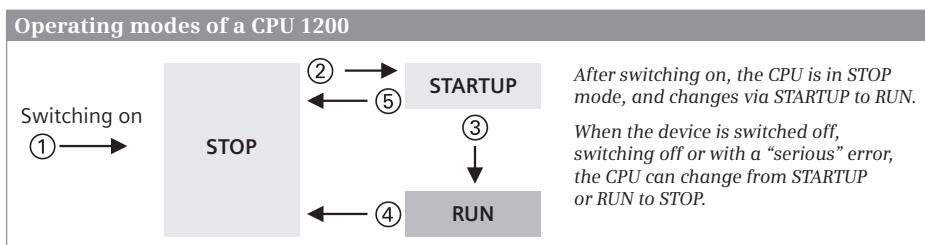
- ▷ Deenergized, when the power supply is switched off
- ▷ STOP if the user program is not being executed
- ▷ STARTUP, when the startup program is being executed
- ▷ RUN, when the main program and the interrupt program are being executed
- ▷ Faulty, if an internal error prevents further execution

Fig. 5.1 illustrates the operating mode transitions: ① After switching on, the CPU is in the STOP mode. If the corresponding conditions are fulfilled, the CPU changes to the STARTUP mode ② and subsequently to the RUN mode ③. If a “serious” error occurs during execution in STARTUP or RUN, or if the CPU is stopped by an operation on the programming device, the CPU returns to the STOP mode ④ ⑤.

The RUN and STOP modes are indicated on the CPU module by the RUN/STOP LED: the LED lights up yellow continuously in the STOP mode, and green in the RUN mode.

You define the CPU's response after the power supply has been switched on by setting the CPU parameters: you can specify whether the CPU is to remain in the STOP mode or immediately commence with execution of the user program.

You can read and control the operating modes using a programming device connected to the CPU: the CPU panel in Online Tools replaces the (mechanical) mode switch (see Chapter 13.3.5 “Online tools” on page 439).



**Fig. 5.1** Operating modes of a CPU 1200

### 5.1.1 STOP mode

The STOP mode is reached:

- ▷ When the CPU is switched on
- ▷ If a “serious” error occurs
- ▷ If the STP system function is executed
- ▷ If a stop request arrives from the programming device

The CPU enters the cause of the STOP operating mode into the diagnostics buffer. In this operating mode, you can also read out the CPU information using a programming device in order to find the reason for the stop.

The user program is not executed in STOP mode. The CPU takes over the device settings – either the values you have set with the hardware configuration when parameterizing the CPU, or the standard settings – and sets the connected modules to the parameterized initial state.

In the STOP mode, the CPU can passively execute one-way communication if, for example, data is requested or sent by another CPU via S7 communication. The real-time clock continues to run in the STOP mode.

You can parameterize the CPU in the STOP operating mode, for example set the IP address, transfer or modify the user program, and you can also carry out a memory reset for the CPU.

### Disabling of output modules

All output modules are disabled when in the STOP and STARTUP operating modes (OD or BASP signal, output disable or disable command output). Disabled output modules output a zero signal or – if configured accordingly – a substitute value.

Although writing to the modules influences the signal memories on them, it does not switch the signal states “to the outside” to the module terminals. The output modules are only enabled when the RUN operating mode is reached.

In the STOP operating mode, you can cancel disabling of the output modules for testing purposes (see Chapter 13.4.9 “Enable peripheral outputs and “Modify now”” on page 451).

### 5.1.2 STARTUP mode

Before the CPU changes from the STOP mode to the RUN mode, it runs through the STARTUP mode. In the STARTUP mode, the CPU initializes itself and the modules controlled by it. If the CPU detects an error in the STARTUP mode, e.g. if an invalid memory card is inserted, it returns to the STOP mode.

You can define the startup response when parameterizing the CPU with the device configuration. With the CPU module selected in the *Properties* tab of the Inspector window, click on *Startup* and select:

- ▷ No startup (stay in STOP mode)  
When the CPU is switched on, it remains in the STOP mode without executing the startup program.
- ▷ Warm restart – RUN  
After switching on, the CPU executes a warm restart. The values of the non-retentive tags are deleted. The startup program is then executed once before the CPU in the RUN mode cyclically executes the main program.
- ▷ Warm restart – mode prior to POWER OFF  
After the system has been switched on, the CPU starts the mode in which it was present prior to switching off (STOP or warm restart and RUN).

### **Warm restart**

With a warm restart, the CPU sets itself and the modules to the configured basic state. It deletes the non-retentive data in the system memory (inputs, outputs, bit memories) and sets the non-retentive data tags to the start values from the load memory. The data set as retentive (bit memories, data tags) is retained.

### **CPU activities in STARTUP mode**

In the STARTUP mode, the CPU first deletes the input process image and disables the peripheral outputs (Fig. 5.2). The signal states on the output modules are set during this process according to their parameterization: retain last value or output parameterized substitute value.

Startup organization blocks are called once if present. The input process image is updated following execution of the startup program, and disabling of the peripheral outputs is canceled.

No Open User Communication is carried out during the startup. The high-speed counters (HSC), the pulse generators, and the point-to-point connections of the CM modules are not processed. The CPU updates the real-time clock.

If a restart is aborted by a power supply failure, for example, it is then re-executed from the beginning when the CPU is switched on again.

Further information on user program processing in the STARTUP state can be found in Chapter 5.5 “Start-up routine” on page 142.

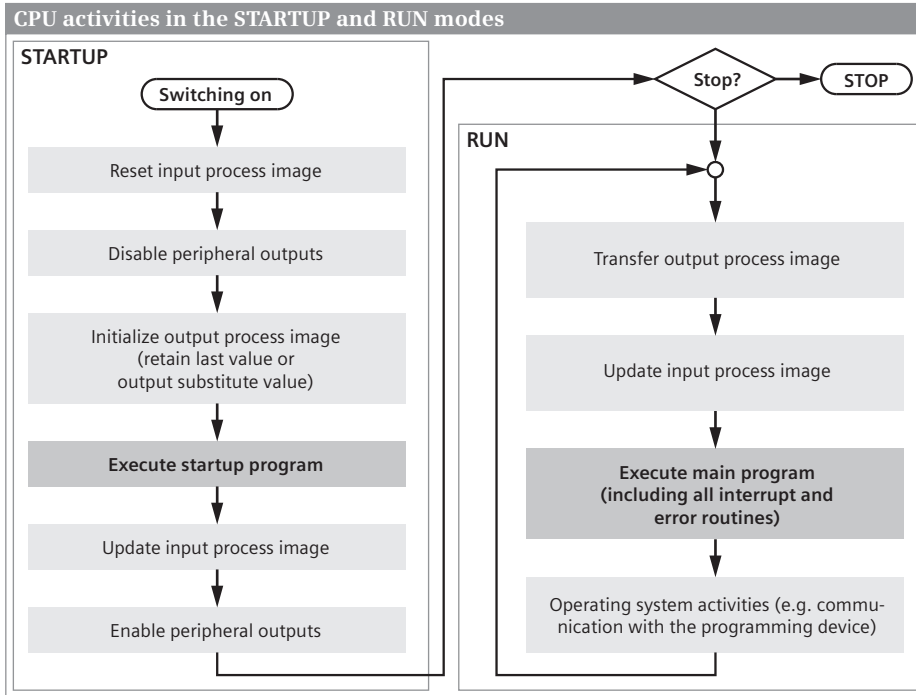
### **5.1.3 RUN mode**

The RUN mode is reached from the STARTUP mode. In the RUN mode, the PLC station controls the machine or process.

The following activities are executed cyclically by the CPU (see also Fig. 5.2):

- ▷ Transmission of output process image to the output modules
- ▷ Updating of input process image
- ▷ Execution of main program





**Fig. 5.2** CPU activities in STARTUP and RUN modes

In addition, the interrupt and error programs are implemented with event-driven execution.

The main program is present in organization block OB 1 and in further organization blocks of the *Program cycle* event class. OB 1 is the only block in the user program which must always be present. If further organization blocks are present for the main program, they are executed following the OB 1 in order of their numbers.

In the RUN operating mode, the CPU has unlimited communication capability. All functions provided by the operating system, e.g. time-of-day and runtime meter, are in operation.

Further information on execution of the user program in RUN mode can be found in Chapter 5.6 “Main program” on page 143 (including process images, cycle time, response time, time-of-day), in Chapter 5.7 “Interrupt processing” on page 153 (time-delay interrupts, cyclic interrupts, and hardware interrupts), and in Chapter 5.8 “Troubleshooting, diagnostics” on page 167 (local error handling, OB 82 “Diagnostic interrupt”, OB 80 “Time error”).

### 5.1.4 Retentive behavior of operands

A memory area is retentive if its contents are retained even when the power supply is switched off, as well as on a transition from STOP to RUN following power-up. The retentive memory of a CPU 1200 comprises 10 KB. It can accommodate bit memories and data tags.

#### Retentivity settings for bit memories

You set the retentive memory area for the bit memories in the PLC tag table or in the assignment list. Click on the *Retain* symbol in the toolbar of the working window, and enter the number of retentive bytes. The retentive area starts at memory byte 0 and ends at byte no. (number – 1). If a bit memory declaration is present within this range, it is assigned a tick or the retentivity symbol in the “Retain” column. A tag occupying more than one byte must not exceed the limit between the retentive and non-retentive ranges.

#### Retentivity settings for global data tags

If the *Optimized block access* attribute is activated in the global or type data block, individual tags can be defined as retentive. In the case of a tag with a structured data type, only the complete tag can be set to retentive. If the attribute is not activated, the retentive setting applies to the entire data block.

#### Retentivity settings for tags in function blocks

If the *Optimized block access* attribute is activated in a function block, the retentivity of individual tags can be set in the interface area. Select the settings for each tag from a drop-down list:

- ▷ Non-retentive  
The tag in the instance data block is always non-retentive.
- ▷ Retentive  
The tag in the instance data block is always retentive.
- ▷ Set in IDB  
The retentivity setting for the tag can be made in the instance data block.  
The standard setting is “Non-retentive”.

For a tag with a structured data type, the retentivity setting applies for the whole tag.

If the attribute *Optimized block access* is not activated, the setting can be made in the instance data block, but only for the complete data block. The *Optimized block access* property of the function block is “bequeathed” to the associated instance data blocks.

## 5.2 Creating a user program

### 5.2.1 Program draft

You define the structure of the user program already during the draft phase by adaptation to technological and functional conditions; this is important for program creation, testing, and commissioning. In order to achieve effective programming, it is therefore necessary to pay particular attention to the program structure.

Analysis of a complex automation task means division of it into smaller tasks or functions based on the structure of the process to be controlled. You define the individual tasks by determining the function and then defining the interface signals to the process or to other individual tasks. You can adopt this structuring of individual tasks in your program. This means that the structure of your program corresponds to the structure of the automation task.

A structured user program is easier to configure and program section by section, and means that more than one person can carry out the work in the case of very large user programs. Last but not least, program testing, servicing, and maintenance are simplified by this division.

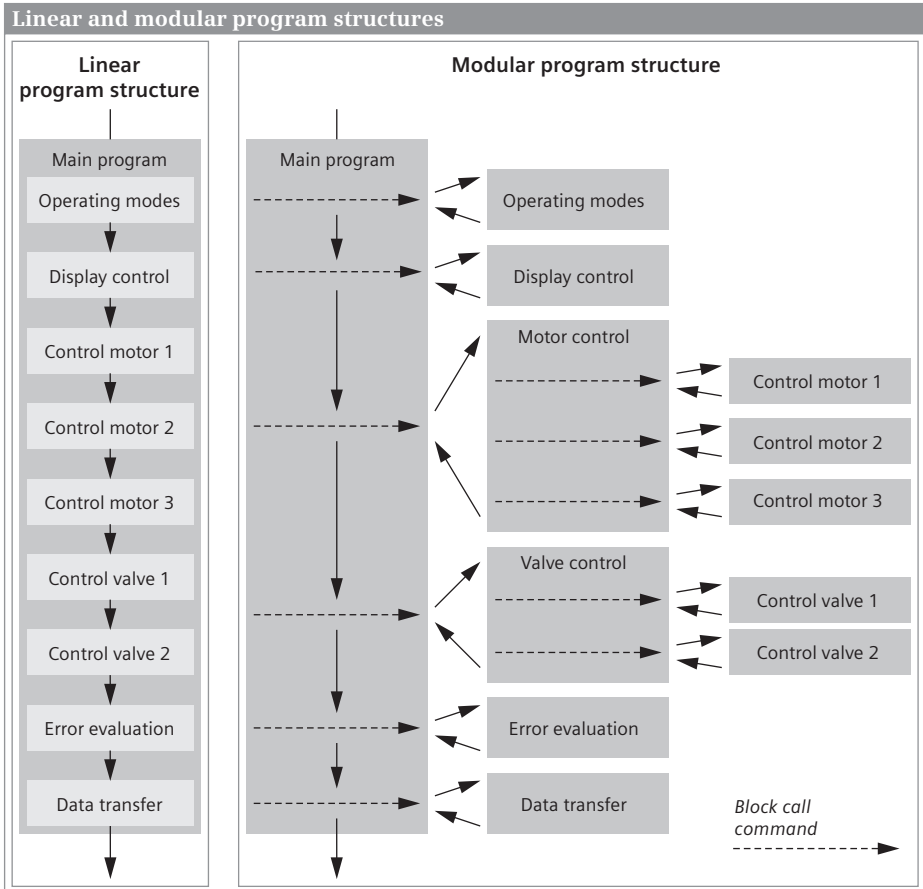
With a **linear program structure**, the entire user program is present in one single block – a good solution for small programs. The individual control functions are program parts within this block, and are executed in succession. A block with LAD or FBD program is divided into so-called networks, each of which accommodates one or more current paths or complete logic operations. STEP 7 numbers all networks in succession. During editing and testing, you can directly reference each network using its number.

The networks are executed in order of their numbering, but can also be bypassed depending on conditions. The program can be debugged in sections using jump instructions temporarily inserted during commissioning.

A **modular program structure** is used if the task is very extensive, if you wish to repeatedly use program functions, or if complex tasks exist. Structuring means dividing the program into sections (blocks) with self-contained functions or a functional correlation, and exchanging as few signals as possible with other blocks. If you assign a specific (technological) function to each program section, manageable blocks are achieved with simple interfaces to other blocks.

Fig. 5.3 shows a comparison between the principles of linear and modular program structuring using a simple example. With the linear program structure, the individual control functions are written in succession into a block. In the modular program structure, each control function is present in a block which is called by a “higher” block. Further blocks can be called in turn in the called blocks.

Blocks can also be used repeatedly. Let us assume that the control of motors 1 to 3 has the same function, only the input and output signals and the control operations are different. A *Motor* block can then be called three times with different signals (parameters) and control the motors independent of one another.



**Fig. 5.3** Comparison between linear and modular program structures

### 5.2.2 Program execution

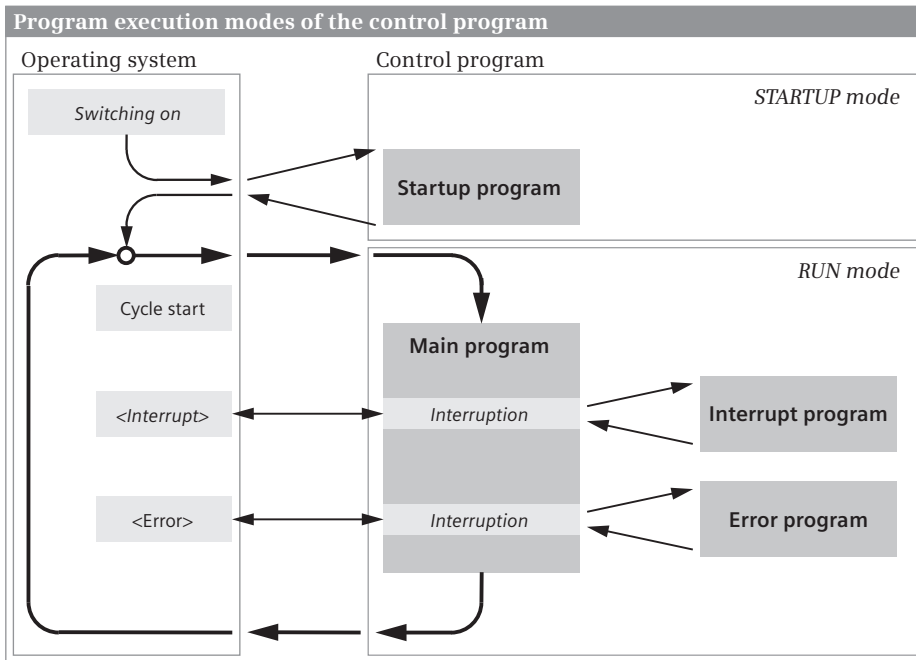
The complete program of a CPU comprises the operating system and the user program.

The *operating system* is the totality of all statements and declarations of internal operating functions (e.g. saving of data in event of power failure, activation of priority classes etc.). The operating system is a fixed part of the CPU which you cannot modify. However, you can reload the operating system from a memory card, e.g. for a program update.

The *user program* is the totality of all statements and declarations programmed by you for signal processing by means of which the plant (process) to be controlled is influenced in accordance with the control task.

## Program execution modes

The user program consists of program sections which are executed by the CPU for specific events. These events can be, for example, the starting up of the automation system, an interrupt, or detection of a program error (Fig. 5.4). The programs assigned to the events are divided into priority classes which define the sequence of program execution if several events occur simultaneously and thus the interrupt capability hierarchy.



**Fig. 5.4** Program execution modes of a SIMATIC user program

The main program, which is executed cyclically by the CPU, has the lowest execution priority. All other events can interrupt the main program following each statement; the CPU then executes the associated interrupt or error program and subsequently returns to execution of the main program.

A specific organization block (OB) is assigned to each event. The organization blocks represent the event classes in the user program. If an event occurs, the CPU calls the associated organization block. An organization block is part of the user program which you can program yourself. There are organization blocks with permanently assigned number and organization blocks with a freely assignable number.

Program execution commences in the CPU with the **startup program**. A startup can be triggered by switching on the power supply or by an operator input on a connected programming device. The startup program is optional. If you wish to create a startup program, use organization block OB 100. You can assign further user organization blocks to the startup program. These are then executed in order of their OB number following OB 100. Following execution of the startup program, the CPU commences with execution of the main program.

The **main program** is present as standard in organization block OB 1 which is always executed by the CPU. The start of the program is identical to that of the first statement in OB 1. You can assign further organization blocks to the cyclically executed main program. These user organization blocks are then executed in order of their OB number following OB 1. The main program represents the totality of all cyclically executed organization blocks.

Following execution of the main program, the CPU branches to the operating system and, following execution of various operating system functions (e.g. update process images), calls OB 1 again and the user organization blocks assigned to the main program.

The events which can interrupt the main program are **interrupts and errors**. Interrupts come from the controlled plant (hardware interrupts), from the CPU (time-delay interrupts and cyclic interrupts), or from the modules (diagnostics interrupts). The errors include time error events.

The organization block OB 80 is used for processing a time error event; for the diagnostic interrupt, there is the organization block OB 82. You can set the number of the other interrupt organization blocks yourself.

### 5.2.3 Nesting depth

The main program and the startup program have a maximum nesting depth of 16; an interrupt-controlled program has a nesting depth of 4. This means that beginning with an organization block (nesting depth 1), you can call another 15 or 3 blocks arranged “horizontally” (nested). If more blocks are called, the CPU generates a program execution error. The nesting depth is not changed if several blocks are called in succession (linear block calls).

## 5.3 Programming blocks

### 5.3.1 Block types

You can divide your program into individual sections as required. The STEP 7 programming languages support this program breakdown by providing the required statements. The individual program sections should be program parts that are self-contained, each with a technological or functional correlation. These program parts

are called “blocks”. A block is a part of the user program that is defined by its function, structure or application.

## User blocks

In the case of comprehensive and complex programs, it is recommend and sometimes necessary to “structure” (divide) the program in individual blocks. You can select different types of block depending on the application:

### ▷ Organization blocks OB

The organization blocks represent the interface between operating system and user program. The CPU's operating system calls the organization blocks when specific events occur, e.g. in the event of hardware or cyclic interrupts. The main program is in organization block OB 1 by default. There are organization blocks with a fixed number and a specific assignment to events and organization blocks with one freely selectable assignment to events and freely selectable number. When calling, some organization blocks make so-called start information available that can be evaluated in the user program.

### ▷ Function blocks FB

A function block is part of the user program whose call can be programmed using block parameters. A function block has a tag memory which is located in a data block. This data block is permanently assigned to the function block or, to be more precise, to the function block call. It is possible to assign a different data block (with the same data structure, but containing different values) to each function block call. Such a permanently assigned data block is called an instance data block, and the combination of function block call and instance data block is referred to as a call instance, or “instance” for short. If a function block is called as a single instance, a separate instance data block is assigned to the call. When called as a local instance, the data is stored in the instance data block of the calling function block.

### ▷ Functions FC

The blocks referred to as “functions” are used to program frequently recurring automation functions. The calls can be parameterized. Functions do not store information, and have no assigned data block.

### ▷ Data blocks DB

Data blocks contain data of the user program. A data block can be generated as a global data block, as an instance data block, or as a type data block. With a global data block, you program the data tags directly in the data block. With an instance data block, the programming of the assigned function block determines the data tags present in the data block, and a type data block has the structure of a PLC data type.

The number of organization blocks and the block number are partially defined by the operating system. You can freely assign the numbers of the freely assignable organization blocks and the other blocks within the permissible ranges (OB from 123 to 65 535, FC and FB from 0 to 65 535, DB from 1 to 65 535).

For our small program structure example (Fig. 5.3 on page 123) we can now roughly assign the block types to the individual control functions: the main program is present in organization block OB 1, either completely with the linear program structure or as block calls with the modular program structure. The individual control functions are either present in functions FC if they need not save their own data, or in function blocks FB when saving their own data.

The motors could be controlled by a function block *Motor* which is called with different parameter assignments for the individual motors. The same applies to control of the valves.

Finally, a global data block which is not assigned to a function block can collect the data produced during program execution. The block *Data transmission* then contains the program which transfers the data to another controller.

### **System blocks**

System blocks are components of the operating system. They can contain programs (system functions SFC or system function blocks SFB) or data (system data blocks SDB). System blocks make a number of important system functions accessible to you, such as manipulating the internal CPU clock, or communications functions. Some of the functions offered under the extended statements in the program elements catalog are system functions or system function blocks.

You can call system functions and system function blocks, but you cannot modify them or program them yourself. The blocks themselves do not require space in the user memory; only the block call and the instance data blocks of the system function blocks are in the user memory.

System data blocks contain information on such things as the configuration of the automation system or the parameterization of the modules. These blocks are generated and managed by STEP 7 itself. You determine the contents of the system data blocks, for example, when you configure a station. As a rule, system data blocks are located in the load memory. You cannot access the contents of system data blocks.

The *system blocks* also include the technological function blocks (FBT). For example, the technological object *Axis* is implemented as a technological function block. The blocks themselves do not require space in the user memory; only the block call and the instance data block of a technological function block are in the user memory.

### **Standard blocks**

In addition to the functions and function blocks you create yourself, off-the-shelf blocks are also available from Siemens. These so-called *standard blocks* can be provided on a data medium or are delivered together with STEP 7, as extended statements or in the global libraries, for example. You cannot view or edit the range of standard blocks. Standard blocks behave like user blocks: they require space in the user memory.



Standard blocks also share the number range with the user blocks. If a standard block is added to the user program by means of an extended statement, for example, the number of the standard block can no longer be occupied by a user block. If a user block is already present with the number of the standard block which you add to the user program, the number of the standard block is initially retained. The standard block is then assigned a different, unused number during the next compilation.

### 5.3.2 Editing block properties

To display and change the block properties, select the block in the project tree and then the *Properties* command in the shortcut menu. Fig. 5.5 shows examples of the block properties: the *General* section of a hardware interrupt organization block, and the *Information* section of a function block.

The **General** section contains the *Name* of the block. The block name must be unique within the program and must not already have been assigned to another block, a PLC tag, a constant, or a PLC data type. The name can contain letters, digits, and special characters (but not quotation marks). No distinction is made between upper and lower case when checking the name. The *Type* is determined when creating the block. The *number* is the block number within the block type. The *programming language* is LAD, FBD, or SLC for blocks with programs, or DB for data blocks.

The image shows two screenshots of the Siemens SIMATIC Manager block properties dialog box. The top screenshot is the 'General' tab for an OB (Organization Block) and the bottom screenshot is the 'Information' tab for an FB (Function Block).

General	
Name:	Hardware interrupt
Constant name:	OB_Hardware interrupt
Type:	OB
Number:	40
Event class:	Hardware interrupt
Language:	LAD

Information	
Title:	Conveyor belt control
Comment:	This block controls a conveyor belt with a single drive in one direction only, without overrun.
Version:	1.0
Family:	Book1200
Author:	Berger
User-defined ID:	CBV001

**Fig. 5.5** Block properties: OB tab “General” and FB tab “Information”

With organization blocks, the *Constant name* constitutes the hardware ID. Use this name if you wish to address the organization block in the program, e.g. for assignment to an event. When creating the organization block you also define the *Event class* to which the organization block belongs. The hardware IDs are listed in the *System constants* tab of the default tag table.

With data blocks, the ID *DB* together with the type of data block is present in the *Type* field: *Global DB* in the case of a global data block, *Instance DB of <FB\_name>* in the case of an instance data block of the function block *<FB\_name>*, and *Data block derived from <Type\_name>* if the structure of the data block is based on a data type *<Type\_name>*.

The **Information** section contains the *Title* and the *Comment*; these are identical to the block title and the block comment which you can enter when programming the block prior to the first network. The *Version* is entered using two two-digit numbers from 0 to 15: from 0.0 to 0.15, 1.0 to 15.15. Under *Author* you can enter the creator of the block. Under *Family* you can assign a common feature to a group of blocks, as is also the case with *User-defined ID*. The author, family, and user-defined ID can each comprise up to 8 characters (without spaces).

The time data in the **Time stamp** section indicates when the block was created and changed last, when the interface was changed last, and when the program code or data was changed last.

The **Compilation** section provides information on the processing status of the block, and – following the compilation – on the memory requirements of the block in the load and work memories.

The **Protection** section indicates the block protection. A block can be protected so that the program can no longer be read out (know-how protection). In the case of a protected block, *The block is protected* is present here. More detailed information can be found in Chapter 5.3.3 “Configuring know-how protection” on page 132. Copy protection allows block processing to be linked to the serial number of the memory card or the CPU. More detailed information can be found in Chapter 5.3.4 “Copy protection” on page 132.

## Attributes

Each block type has a different combination of attributes. Table 5.1 lists the attributes. Fig. 5.6 shows the attributes of a logic block and a data block. In the general settings (*Options > Settings > PLC programming* in the main menu), you define the default setting when creating new blocks for the *Optimized block access* and *IEC check* attributes.

You define the *Optimized block access* attribute when creating the block. If the attribute is activated, only symbolic addressing of the interface tags or the data tags in the block is possible. With instance data blocks, the *Optimized block access* attribute is “inherited” from the associated function block; in this case the data tags are addressed by the associated function block.

**Table 5.1** Block attributes

Attribute	In block	Meaning with attribute activated
Optimized block access	OB, FB, FC, DB 1)	The block parameters and local data or data tags can only be symbolically addressed; the retentivity can be set for individual tags; the data type LREAL can be used.
IEC check	OB, FB, FC	With comparison and arithmetic statements, the data types of the tags must agree.
Handle errors within block	OB, FB, FC	One of the functions <i>GetError</i> or <i>GetErrorID</i> is programmed in the block; the system response to an error in the block is suspended.
Only store in load memory	Global and type DBs	The data block is not transferred to the work memory (for future extensions).
Data block write-protected in the device	Global and type DBs	The values of the data tags cannot be overwritten by means of a program.

1) The attribute is always activated for an organization block

The *Optimized block access* attribute results in “optimized” storage of the tags: the tags are not stored in the declaration sequence but are “packed together”, i.e. bit and byte tags are combined when possible in 16-bit packages. The *Optimized block access* attribute also has effects on the retentivity setting of the data variables: with the attribute activated, individual variables can be set as retentive (in the associated function block for instance data blocks), but only the complete block can be set when the attribute is not activated.

The *IEC check* attribute indicates how strict the data type test is to be in the logic block. With the attribute not activated, it is usually sufficient if the variables used have the data width required for execution of the function or instruction; with the attribute activated, the data types of the variables must correspond to the required data types.

The figure shows two screenshots of the 'Attributes' dialog box. The top screenshot, for code blocks, has three attributes: 'Optimized block access' (checked), 'IEC check' (unchecked), and 'Handle errors within block' (unchecked). The bottom screenshot, for data blocks, has three attributes: 'Optimized block access' (checked), 'Only store in load memory' (unchecked), and 'Data block write-protected in the device' (unchecked).

**Fig. 5.6** Block properties: attributes of code and data blocks

The *Handle errors within block* attribute is activated as soon as one of the functions *GetError* or *GetErrorID* is inserted in the program. The system reaction to a programming or access error is then suppressed in favor of a self-programmed error routine.

Global and type data blocks can be assigned the *Only store in load memory* attribute. Such types of data block are only present in the load memory on the memory card, they are “not relevant to execution”. Since their data is not in the work memory, direct access is not possible. There are system functions to access data in the load memory. This attribute is switched off as standard and can be activated or deactivated at any time using the program editor.

The *Data block write-protected in the device* attribute applies to global and type data blocks. If you activate this, writing of the data tags is not possible. The write protection applies to all data relevant to execution (actual values) in the work memory. Write protection must not be confused with block protection: A data block with block protection (“know-how protection”) can be read and written by the program; however, its data can no longer be viewed, for example using a programming device.

### Block properties for interrupt organization blocks

With a process interrupt organization block, the events which start the process interrupt OB are present in a table in the *Triggers* section of the block properties. You configure the start events and the assignment to a process interrupt OB using the device configuration in the properties of the module triggering the interrupt (Fig. 5.7).

In the case of an organization block with the start event *Cyclic interrupt*, the start properties for the cyclic interrupt OB can be found in the block properties under *Cyclic interrupt*. You enter the time base and the phase shift in milliseconds here.

The image shows two sections of a configuration window. The top section is titled 'Triggers' and contains a table with the following data:

Source Module	Triggers	Event	Tag
DI14/DO10	I1.3	Rising edge	Stop signal down
DI14/DO10	I1.2	Falling edge	Stop signal above

Below the table is a section titled 'Cyclic interrupt' with two input fields:

Cyclic time (ms):

Phase shift (ms)

Fig. 5.7 Block properties for interrupt organization blocks

### 5.3.3 Configuring know-how protection

With the know-how protection for a block you can prevent a program or its data from being read out or modified. A protected block is identified in the project tree by a padlock icon. It is still possible to read the following from a block provided with know-how protection:

- ▷ Block properties
- ▷ The parameters of the block interface
- ▷ Program structure
- ▷ Global tags (listed in the cross-reference list without specification of the position of use)

The following actions are still possible:

- ▷ Modify name and number in the block properties (necessary for copying and pasting the block)
- ▷ Copy and paste block where the know-how protection is also copied
- ▷ Delete, compile, and download block
- ▷ Call block (FB or FC) in the program of another block
- ▷ Compare online and offline versions of the block (comparison only of non-protected data).

To edit the know-how protection, select the block in the project tree under *Program blocks*, and then select *Edit > Know-how protection* in the main menu. To configure the know-how protection, click the *Define* button, enter a password, confirm the password, and close the dialog with *OK*. To change the password, click the *Change* button, enter the old and new passwords, confirm the new password, and close the dialog with *OK*. To cancel the know-how protection, deactivate the *Hide code (know-how protection)* checkbox, enter the password, and close the dialog with *OK*.

You can also apply the know-how protection to several blocks simultaneously if these have the same password. If a function block is protected, the protection is “inherited” by the instance data block when calling as a single instance.

*Note:* If the password is lost, no further access to the block is possible. You can only cancel the know-how protection of a block in its offline version. If you download a compiled block to the CPU, the recovery information is lost. A protected block which you have uploaded from the CPU cannot be opened, not even with the correct password.

### 5.3.4 Copy protection

If a block has copy protection, processing of the block is dependent on a specific CPU or memory card. So that the copy protection cannot be removed, the block must then be provided with the know-how protection.

When the copy protection is being set up, the know-how protection for the block must be switched off. To set up the copy protection, select the block in the project

tree, select *Properties* from the shortcut menu and then *Protection*. In the *Copy protection* area, you can choose:

- ▷ *No binding*  
No copy protection is set or a set copy protection is canceled.
- ▷ *Bind to serial number of the memory card*  
The block can only be executed if the memory card has the specified serial number.
- ▷ *Bind to serial number of the CPU*  
The block can only be executed if the CPU has the specified serial number.

To enter the serial number, the options *Serial number is inserted when downloading to a device or a memory card* and *Enter serial number* are available with an input field for the serial number.

### 5.3.5 Block interface

#### Components of the block interface

The block interface contains the declarations of the local tags that are used solely within the block. With the organization blocks (OB), these are – if present – the *start information* and the *temporary local data*. With function blocks (FB) and functions (FC), these are the *block parameters* which, when the block is called, provide the

**Table 5.2** Declaration sections in the block interface

Section	Data type	Type, function	Included in
Input	E, Z, V, P, S, H STRING[ ] STRING	Input parameters may only be read in the program of the block  Data type STRING of any length Data type STRING with the standard length of 254 characters	FC, FB and some OBs  FB FC
Output	E, Z, P, (V) STRING[ ] STRING	Output parameters may only be written in the program of the block  Data type STRING of any length Data type STRING with the standard length of 254 characters	FC and FB  FB FC
InOut	E, Z, P, V, S STRING	In/out parameters may be read and written in the program of the block  Data type STRING with the standard length of 254 characters	FC and FB  FC and FB
Temp	E, Z, P, STRING	Temporary local data is only valid during the current block processing	FC, FB and OB
Static	E, Z, P, S	Static local data is saved in the instance data block, and remains valid following block processing	FB
Return	E, DTL, STRING, P, (V), VOID	Function value Output parameters with the return value of the function (not relevant to LAD and FBD)	FC

E = elementary data type, Z = structured data type (except STRING), P = PLC data type

V = parameter type VARIANT, (V) parameter type VARIANT only in functions FC

S = system data type, H = hardware data type;

FC = function, FB = function block, OB = organization block

interface to the calling block, and the *local data* for saving intermediate results (Table 5.2).

The block interface is shown as a table in the upper part of the working window. An example for the block interface of a function block is shown by the Fig. 5.8.

Interface					
	Name	Data type	Default value	Retain	Comment
1	▼ Input				
2	Switch_on_Auto	Bool	false	Non-retentive	
3	Switch_on_Manual	Bool	false	Non-retentive	
4	Switch_off_Auto	Bool	false	Non-retentive	
5	Switch_off_Manual	Bool	false	Non-retentive	
6	Log mode	Bool	false	Non-retentive	
7	Fault	Bool	false	Non-retentive	
8	▼ Output				
9	Switch on motor	Bool	false	Non-retentive	
10	Display motor running	Bool	false	Non-retentive	
11	▼ InOut				
12	<Add new>				
13	▼ Static				
14	Motor memory	Bool	false	Non-retentive	
15	Voltage	Real	0.0	Non-retentive	
16	Current	Real	0.0	Non-retentive	
17	Power	Real	0.0	Non-retentive	
18	Operating_time	Time	T#0ms	Non-retentive	
19	Duration	Time	T#0ms	Non-retentive	
20	Switched on	Bool	false	Non-retentive	
21	▼ Temp				
22	temp1	Int			
23	temp2	Int			
24	temp3	Real			
25	temp4	DT			
26	<Add new>				

Fig. 5.8 Example of block interface of a function block

You can assign a default value to the block parameters and static local data in the interface of a function block; exceptions: A default setting is not possible for in/out parameters with structured data type and for parameter type VARIANT. The default values are in load memory and are transferred during the CPU startup to the work memory where they overwrite the actual values.

### Input parameters

An input parameter transfers a value to the program in the block and may only be read in the called block. Input parameters are shown in the block call in the sequence of their declaration, with LAD and FBD on the left side of the call box and with SCL at the start of the parameter list.

An input parameter with data type STRING has an adjustable maximum length in a function block, and a fixed maximum length of 254 characters in a function. Some organization blocks have so-called startup information which is listed as input parameters in the block interface.

## Output parameters

An output parameter transfers a value to the calling block and may only be written in the called block. Output parameters are shown in the block call in the sequence of their declaration, with LAD and FBD on the right side of the call box and with SCL following the input parameters in the parameter list.

An output parameter with data type `STRING` has an adjustable maximum length in a function block, and a fixed maximum length of 254 characters in a function.

Caution: Output parameters which cannot be assigned a default value **must** be written in the block during **each** block processing. This applies, for example, to all output parameters for a function (FC) and thus also to the function value. Note: Set and reset statements do not execute an action if the result of the logic operation = "0", and therefore do not write to an output parameter!

## In/out parameters

An in/out parameter transfers a value to the program in the block and can return it to the calling block, usually with a changed content. An in/out parameter can be read and written in the called block. In/out parameters are shown in the block call in the sequence of their declaration, with LAD and FBD on the left side of the call box under the input parameters and with SCL at the end of the parameter list.

An in/out parameter with data type `STRING` has a fixed maximum length of 254 characters.

## Function value

The function value for functions is an output parameter which is handled in a special manner. It has the name *Ret\_Val* with the declaration `RETURN` and the data type `VOID` (= no type) as standard. The function value is used with the text-based programming language SCL (Structured Control Language). It is possible here to link self-written functions in formulae (in expressions). The function value then corresponds to the value used for calculation in the formula.

With LAD and FBD you can ignore the function value in the interface description. It is not indicated at the call box if the data type `VOID` is set. You can also assign a different name and a different data type to the function value, and this is then displayed as the first output parameter. In the program of the called block, you then treat the function value in the same way as an output parameter.

## Temporary local data

Temporary local data is stored in the system memory of the CPU. This data is addressed symbolically and is only available during block processing. It is not displayed on the call box or in the parameter list of the call statement. Further information can be found in Chapter 4.1.5 "Operand area temporary local data" on page 85.



## Static local data

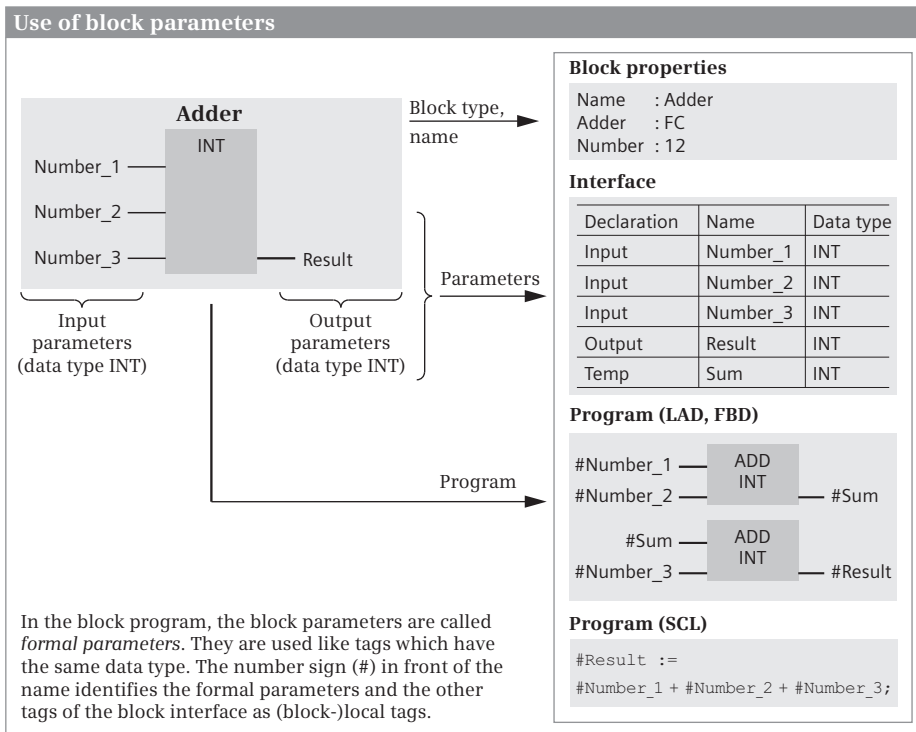
The static local data is stored in the instance data of the called function block. It can be read and written in the program of the called block. Static local data is addressed symbolically. It retains its value until written again. It is not displayed on the call box or in the parameter list of the call statement.

The static local data is usually only processed in the function block itself. However, since the static local data is saved in a data block, you can access it at any time like tags in a global data block using “*Data block name*”.*tag name*.”

The static local data is normally accessed symbolically. Absolute addressing is only possible if the *Optimized block access* attribute is not activated in the function block – and thus also in the derived instance data block – and if you address the data operands in the instance data block like the data operands of a global data block: using *%Data block number.Data operand address*, e.g. *%DB12.DBW0*.

### 5.3.6 Programming block parameters

By means of *block parameters* you enable parameterization of the processing specification (the block function) present in a block.



**Fig. 5.9** Example of programming with block parameters

The example shows an adder with three summands which can be used repeatedly in the user program with different tags. The tags are transferred as block parameters – in our example, three input parameters and one output parameter. Since the adder need not permanently save values internally, a function FC is suitable as the block type (Fig. 5.9).

The values to be transferred are declared as input parameters in the *Input* section with name and data type, the calculated value as an output parameter in the *Output* section, also with name and data type. If the program is written in LAD or FBD in the block, another tag is required as intermediate memory. This is declared in the *Temp* section, since its value is not required outside the block. A tag for intermediate storage is not required with an SCL program.

The program in the block can be written in the language with which the block function is best mapped, independent of the programming language with which the block is subsequently called. The block parameters used in the block program are called *formal parameters*. They are handled like tags which have the same data type. They are the placeholders for the current tags used later at runtime.

The “Adder” function can then be called repeatedly in the user program. Different values are transferred to the adder at the block parameters with each call. These values can be constants, operands, or tags; they are referred to as *actual parameters*. During runtime, the control processor replaces the formal parameters by the actual parameters. Section “Example of a block call” on page 138 shows how the adder block programmed here is called and supplied with current tags.

A timer or counter function can also be transferred to the program in the block with a block parameter. If the timer or counter function is transferred as an input parameter, the function status can be scanned. If the timer or counter function should be controlled, it must be transferred as an in-out parameter.

## 5.4 Calling blocks

### 5.4.1 General information on calling logic blocks

If blocks are to be processed, they must be called in the program first. The organization blocks which are started by the operating system when certain events occur are an exception.

With FBD and LAD, the call functions are boxes with an enable input EN and an enable output ENO. A block call dependent on the result of logic operation can be implemented using the enable input EN. The enable output ENO can be used to signal a malfunction determined in the block to the calling block. In SCL, the enable input EN and the enable output ENO are implicitly available parameters that you can add to the first or last position in the parameter list if needed.

The call box or call function shows all block parameters which were declared when the block was created. If you subsequently change the block interface of the called block, you must update the changes in the block call otherwise the program editor

will signal an “Interface conflict”. Finding and eliminating an interface conflict is described in Chapter 6.6.5 “Consistency check” on page 206.

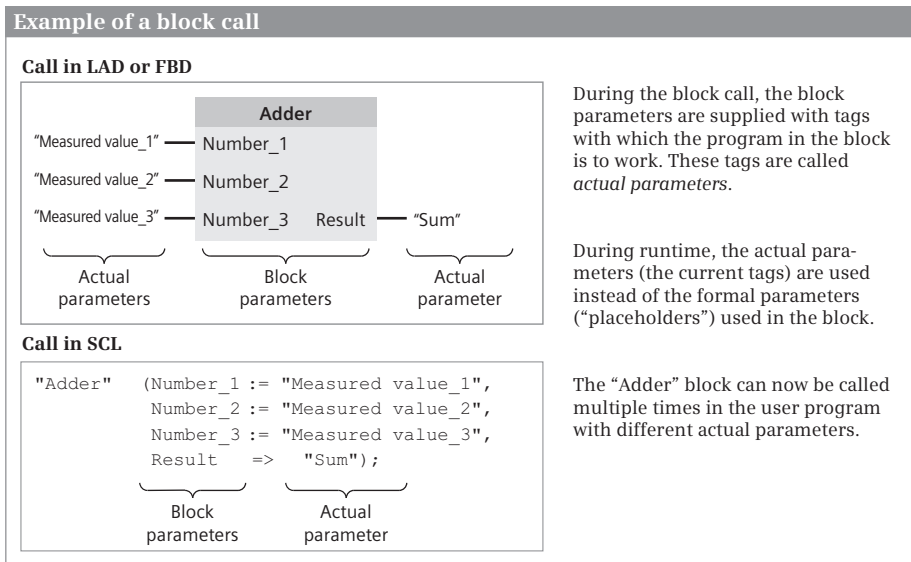
A prerequisite for calling a block is that it exists; at least its interface must be programmed. You call a block by selecting it under *Program blocks* in the project tree and dragging it into the program of an opened block using the mouse.

If you drag a block directly from a library into an opened block, it is copied into the *Program blocks* folder. If it is a system or standard block, it is saved in the *Program blocks > System blocks > Program resources* folder.

The call functions are described in detail in Chapter 12.3 “Calling of code blocks” on page 413.

### Example of a block call

Chapter 5.3.6 “Programming block parameters” on page 136 shows how a block (an FC function) is programmed with a block parameter. You can now call the “adder” function in your program and transfer the values with which the block should work to the block parameters. These values can be constants, operands or tags; they are referred to as *actual parameters* (Fig. 5.10).



**Fig. 5.10** Example of a block call with block parameters

During runtime, the control processor replaces the formal parameters by the actual parameters. When calling the “Adder” block in the example, the contents of the tags “Value\_1”, “Value\_2”, and “Value\_3” are added together and the result stored in the “Sum” tag.

Note: The tag `#Sum`, which was used in the “Adder” block, is a local tag and saved in the temporary local data. The tag “`Sum`”, which supplies the block call, is a global tag, for example a flag word. “`Sum`” as a global tag (PLC tag) only appears once in the CPU. `#Sum` as a local tag can be used with different significance in each block.

The “Adder” block can now be called multiple times in the user program, each time with a different parameter assignment. The created program is processed multiple times with various tags.

### 5.4.2 Calling a function (FC)

When calling a function, all block parameters must be supplied with actual operands, i.e. you must connect operands or tags to all block inputs and outputs. For LAD and FBD, connect the enable input EN and the enable output ENO as needed; for SCL only the use of ENO is allowed for an FC function.

#### Supplying the block parameters

You can use tags from the inputs, outputs, bit memories, and data operand ranges on all block parameters. Constants and peripheral inputs are only permissible for input parameters, peripheral outputs only for output parameters.

The data type of the actual parameter must correspond to the data type of the block parameter. The data types must agree exactly if the *IEC check* attribute is activated in the calling block, otherwise matching widths of the tag or operand are usually sufficient. The program editor uses implicit data type conversion if it is possible without data loss.

Any maximum length of an actual parameter is possible for a block parameter with data type STRING. Note that an actual parameter with data type STRING which has been declared in the temporary local data cannot be assigned a default value and therefore has any content. It must be provided with meaningful values before being used as an actual parameter.

You supply a block parameter with the data type of a timer or counter function with the instance data of a timer or counter function. This can be the instance data block of the function call or the in-out parameter with the data type of the timer or counter function.

Tags of all data types are allowed on a block parameter of VARIANT parameter type, including operand areas addressed with a pointer (Chapter 4.2.3 “Absolute addressing of an operand area” on page 86). An entire data block can only be an actual parameter if it is derived from a PLC data type or a system data type (type data block). The tags (operands or data types) which can be connected to the block parameters or which are meaningful are defined by the program within the called block.

#### Using a function value of a function (FC)

The function value of a function has no effect when declared with data type VOID. If the function value has another data type, it is handled in the block program like an

**Example for use of the function value**

In the “Adder2” function, the function value is declared in the block interface in the *Return* section, in the example with the name *Result* and data type INT.

**Block interface**

Declaration	Name	Data type
Input	Number_1	INT
	Number_2	INT
	Number_3	INT
Return	Result	INT

The block interface of the called block contains the three input parameters and the function value as result of the addition of the three input parameters. The program in the “Adder2” block can be written in any programming language. The function value is handled like an output parameter.”

**Block call in an expression**

```

"Sum" := "Adder2" (Number_1 := "Measurement_1",
                  Number_2 := "Measurement_2",
                  Number_3 := "Measurement_3") + "Correction value";

```

The “Adder2” function can now be used in an expression in the programming language SCL. The function is handled like a tag which has the data type of the function value.

**Fig. 5.11** Use of the function value with SCL

output parameter. The program editor then replaces the name of the function value in the block program with the block name.

When calling the block, the function value is represented as the first output parameter in LAD and FBD – provided it does not have data type VOID. SCL handles a function with function value like a tag with the data type of the function value. Fig. 5.11 shows an example: The function “Adder2” adds three numbers and returns the total as a function value with data type INT. The total can be directly processed further in an expression.

### 5.4.3 Calling a function block (FB)

When calling a function block, you are requested to specify the storage location of the instance data. This is the data, with which the function block works internally: the block parameters and the static local data.

Specify a data block if the call takes place in an organization block or a function. The call then takes place as a “single instance”, and the data block is the instance data block for this call. If you call the function block as a single instance for a second time, enter a different data block as the instance data block. This then contains the data for the second call. Assign a separate data block to each call of a function block as single instance.

When calling a function block in another function block, you can choose the following: You can call the function block as a “single instance” or as a “local instance” (“multi-instance”). With a single instance, the call is assigned a separate data block as instance data block. When calling a local instance, the called function block stores its instance data in the instance data block of the calling function block. You

then specify the name with which the local instance can be addressed in the static local data of the calling function block. You can repeatedly call a function block as a local instance using different names in each case.

### Supplying the block parameters

The block parameters of a function block are located in the instance data. Therefore not all block parameters have to be supplied when calling the function block. If the supply is omitted, the function block works with the “old” values from its last call or with the default settings. An exception are block parameters with data type VARIANT and in-out parameters with structured data type; these must be supplied so that a valid pointer can be entered in the instance data. You can supply the EN enable input and ENO enable output as required.

You can use tags from the inputs, outputs, and bit memories operand areas for all block parameters. Constants and peripheral inputs are only permissible for input parameters, peripheral outputs only for output parameters.

The data type of the actual parameter must correspond to the data type of the block parameter. The data types must agree exactly if the *IEC check* attribute is activated in the calling block, otherwise matching widths of the data type or operand are usually sufficient. The program editor uses implicit data type conversion if it is possible without data loss.

An input or output parameter of a function block with data type STRING can only be supplied with STRING tags whose maximum length corresponds to that of the block parameter. Any maximum length of the STRING tag is possible on an in/out parameter. Note that an actual parameter with data type STRING which has been declared in the temporary local data cannot be assigned a default value and therefore has any content. It must be provided with meaningful values before being used as an actual parameter.

You supply a block parameter with the data type of a timer or counter function with the instance data of a timer or counter function. This can be the instance data block of the function call, the name of the local instance, or an in-out parameter with the data type of the timer or counter function.

Tags of all data types are allowed on a block parameter of VARIANT parameter type, including operand areas addressed with a pointer (Chapter 4.2.3 “Absolute addressing of an operand area” on page 86). An entire data block can only be a current parameter if it is derived from a PLC data type or a system data type (type data block). The tags (operands or data types) which can be connected to the block parameters or which are meaningful are defined by the program within the called block.

### “External” access to local data

The block parameters of a function block are located in a data block. If a block parameter is saved as a value (not as a pointer), you can address it from any position in the user program like a global data tag. The address for a single instance is

“Data block”.Parameter name and for a local instance “Data block”.Instance name.  
Parameter name.

Block parameters with VARIANT data type as well as in-out parameters with structured data types are saved as pointers (as a reference).

#### 5.4.4 “Passing on” of block parameters

The “passing on” of block parameters is a special form of access and supply of block parameters. The parameters of the calling block are “passed on” to the parameters of the called block. In this case, the formal parameter of the calling block is then the actual parameter of the called block.

It always applies here that the actual and formal parameters must be of the same type, i.e. the associated block parameters must agree with regard to their data types. Note in this context that the maximum length may have to be considered with data type STRING.

It additionally applies that you can only connect an input parameter of the calling block to an input parameter of the called block, and an output parameter only to an output parameter. You can connect an in/out parameter of the calling block to all declaration types of the called block.

The “passing on” of block parameters also applies in the same manner to statements (program functions) which are represented with inputs and outputs similar to a block call. If these statements are supplied with block parameters, input (block) parameters can only be connected to function inputs, output (block) parameters only to function outputs. In/out parameters can be connected to function inputs and function outputs.

## 5.5 Start-up routine

A CPU 1200 carries out a warm restart when started up. The activities carried out during the warm restart are described in Chapter 5.1.2 “STARTUP mode” on page 118.

### Organization blocks for the startup program

When starting up a CPU 1200 – in the transition from STOP to RUN, for example when switching on the power supply – a startup program is processed once. The startup program is present in organization block OB 100 and the blocks called within it. The OB 100 is of hardware data type OB\_STARTUP and event class *Startup*. You can create additional organization blocks with this event class, which are then given a freely choosable number of 123 or higher. The further startup organization blocks are called and executed following the OB 100 in the order of their numbers.

A startup program is not essential. If no startup program is required, simply omit the organization blocks with the *Startup* event class.

**Table 5.3** Start information for a startup organization block

Name	Declaration	Data type	Description
LostRetentive	Input	BOOL	= "1" if retentive data areas have been lost
LostRTC	Input	BOOL	= "1" if the time of the real-time clock has been lost

The startup program can have any length. There is no time limit for executing the startup program; the cycle time monitoring is not active. A startup organization block has the start information shown in Table 5.3.

The process image input is reset during execution of the startup program, i.e. scanning of an input delivers the signal state "0". However, you can scan the signal states or analog values directly on the module terminals by means of the operand area "Peripheral inputs".

No interrupt events – except errors – are processed during execution of the startup program. Interrupts occurring during the startup are executed after the startup but before the main program.

## 5.6 Main program

The main program is the cyclically processed user program; this is the "normal" way in which programs are executed in PLCs. The large majority of control systems only use this form of program execution. If event-driven program execution is used, it is usually only an addition to the main program.

### 5.6.1 Organization blocks for the main program

The main program is present in organization block OB 1 and the blocks called within it. The OB 1 is of hardware data type OB\_PCYLE and event class *Program cycle*. You can create additional organization blocks with this event class, which are then given a freely choosable number of 123 or higher. The further main program organization blocks are called and executed following the OB 1 in the order of their numbers.

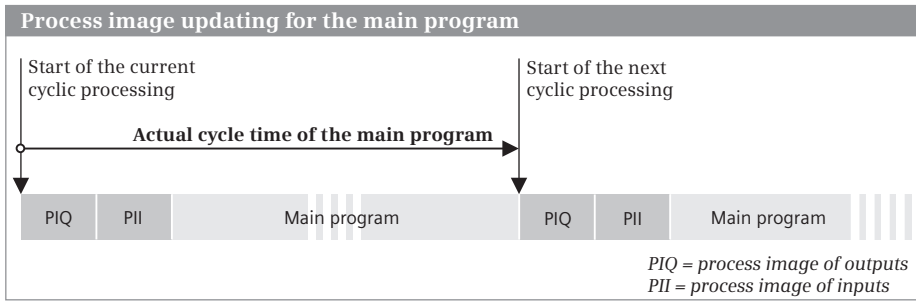
Organization blocks for cyclic program execution have no start information.

The main program runs in the lowest priority class and can be interrupted by alarm and error events. The corresponding organization blocks are then called and processed. After processing an interrupt, the main program continues from the point of interruption.

### 5.6.2 Process image update

The process image is part of the CPU's internal system memory (see Section 4.1 "Operands and tags" on page 79). The process image consists of the process image input (operand area "Inputs I") and the process image output (operand area "Outputs Q"). It has a size of 1024 bytes (addresses 0 to 1023) per area. All digital and





**Fig. 5.12** Process image update

analog input/output channels, independent of the module, are in the address range of the process image.

Following a CPU restart and prior to initial execution of the main program, the operating system transfers the signal states of the output process image to the output modules, and accepts the signal states of the input modules into the input process image.

This is followed by execution of the main program where the signal states of the inputs are combined with each other and the outputs are controlled. Following termination of the main program, a new cycle begins with updating of the process image (Fig. 5.12).

### Exceptions from process image updating

You can exclude individual modules from the automatic updating of the process image. This is carried out when configuring the modules. The signal states of these modules must then be addressed by the program using direct access (I/O area: P).

Switching off the process image updating can be applied to an interrupt program. If an input module is addressed directly in the interrupt program, the signal state currently present at the terminals is read (and not the signal state at the point in time of process image updating). It is sometimes necessary in the interrupt program to react promptly, e.g. to reset an output quasi immediately. The direct access can be used to directly influence the signal states on the output modules.

### 5.6.3 Cycle time

#### Cycle monitoring time

Processing of the main program with regard to timing is carried out by means of the so-called *Cycle monitoring time*. The default value for the monitoring time is 150 ms. You can set this value within the range from 1 ms to 6000 ms by parameterizing the CPU accordingly.

If processing of the main program takes longer than the set cycle monitoring time, the CPU calls the organization block OB 80 *Time error*. If this does not exist,

- ▷ A CPU 1200 with firmware version V1.0 ignores the error message. If the cycle monitoring is triggered for a second time during a program cycle, the CPU goes to STOP – even if an OB 80 is present.
- ▷ A CPU 1200 with firmware version V2.0 or higher switches to STOP mode.

The cycle processing time comprises:

- ▷ The total processing time of the main program (processing times of all organization blocks with the event class *Program cycle*),
- ▷ The processing times for higher priority classes which interrupt the main program (in the current cycle)
- ▷ The time required to update the process images, and
- ▷ The time for communication processes by the operating system, e.g. access operations of programming devices to the CPU (the program status in particular takes a long time!).

The current cycle processing time can be observed online on the current CPU (Chapter 13.3.5 “Online tools” on page 439).

### RE\_TRIGR Restart cycle monitoring time

RE\_TRIGR restarts the cycle monitoring time. This then starts with the value set during CPU parameterization, and ENO has the signal state “1”. RE\_TRIGR does not have any parameters (Fig. 5.13).

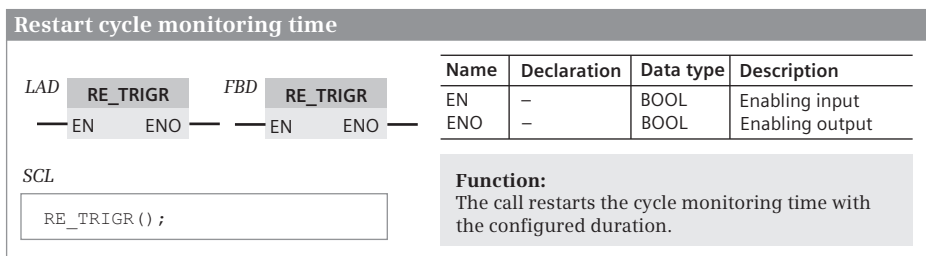


Fig. 5.13 Restart cycle monitoring time

The RE\_TRIGR function is only effective when called in the main program. The cycle monitoring time is not restarted by a call in the startup program or in an interrupt program, and ENO has the signal state “0”.

## Communication load

The CPU's operating system requires a certain time for communication with the programming device or with other stations. In the CPU properties, you can set the percentage of the cycle time which is to be available for communication tasks. If you set a high percentage, it may be necessary to adapt the cycle monitoring time and the minimum cycle time. 20% is set by default.

If  $k$  represents the communications load in percent, the processing time of the main program changes by a factor of  $100 / (100 - k)$ . This does not account for any interruptions due to alarm or error events.

Independent of this setting, the CPU carries out communication tasks every 100 ms. This should guarantee that the CPU can still be accessed and switched to STOP in the event of an endless loop with restarting of the cycle monitoring time.

## Minimum cycle time

In the CPU properties you can set a minimum cycle time in addition to the maximum cycle (monitoring) time. The highest value of the minimum cycle time cannot be greater than the maximum cycle (monitoring) time.

If the minimum cycle time is set, the CPU waits at the end of processing of the main program until the minimum cycle time has expired, and only then commences with a new program cycle. If processing of the main program takes longer than the set minimum cycle time, this has no further effects.

Application of a minimum cycle time reduces large variations in the processing time, and thus large variations in the response time. The CPU can execute communication tasks while waiting for the minimum cycle time to expire.

### 5.6.4 Reaction time

If the user program in the main program works with the signal states of the process images, this results in a response time which is dependent on the program execution time (the cycle time). The response time lies between one and two cycle times, as demonstrated in the following example.

If a limit switch is activated, for example, it changes its signal state from "0" to "1". The PLC detects this change during subsequent updating of the process image, and sets the input allocated to the limit switch to "1". The program evaluates this change by resetting an output, for example, in order to switch off the corresponding drive. The new signal state of the output that was reset is transferred at the end of program execution; only then is the corresponding bit reset on the digital output module.

In a best-case situation, the process image is updated immediately following the change in the limit switch's signal (Fig. 5.14). It then only takes one cycle for the corresponding output to respond. In a worst-case situation, updating of the process image has just been completed when the limit switch's signal changes. It is then necessary to wait approximately one cycle for the PLC to detect this change

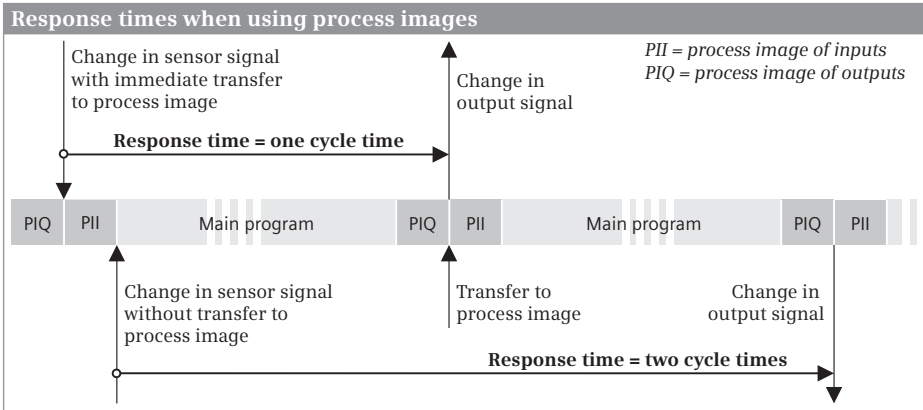


Fig. 5.14 Response times of PLCs

and to set the input in the process image. The response then takes place after one further cycle.

The response time to a change in the input signal can thus be between one and two cycles. Added to the response time are the delays at the input modules, the switching times of contactors, and so on.

In certain cases you can reduce the response times by addressing the I/O directly or by calling program sections depending on events, for example through a hardware interrupt.

Uniform response times or equal time intervals in the process control can be achieved if a program section is always executed at regular intervals, e.g. a watch-dog interrupt program.

### 5.6.5 Stop program execution

STP terminates program execution; the CPU then switches to the STOP operating mode. The STP function does not have any parameters (Fig. 5.15).

Stop program execution			
Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output

<p>LAD  FBD </p> <p>SCL</p> <pre>STP ();</pre>	<p><b>Function:</b> The call stops processing of the user program.</p>
--	--

Fig. 5.15 Stop program execution

The CPU terminates processing of the user program and updating of the process image output. In the module properties, you can set the signal states of the digital and analog outputs which the CPU is to output in the STOP mode: *Keep last value* or *Use substitute value*. As standard, the signal state “0” is output at the digital outputs and a value of zero at the analog outputs at STOP.

In the STOP operating mode, the CPU continues communication with the programming device and the diagnostics activities.

### 5.6.6 Time

Each CPU 1200 has a real-time clock which you can set and scan using a programming device or system functions. The clock is buffered by a high-performance capacitor. If the CPU was connected to the power supply for at least 24 hours, the capacitor is charged sufficiently to provide a power reserve for the clock for approx. 10 days. The buffered runtime of the real-time clock can be extended up to one year with the Battery Board BB 1297.

The time is represented in the user program in DTL format, thus comprising the date, time, day of week, and daylight saving/standard time ID.

#### System time, local time, daylight saving/standard time

The time set in the CPU's real-time clock is the system time (module time). This is decisive for all timing processes controlled by the CPU, e.g. entry of time stamp in the diagnostics buffer and in the block properties. `WR_SYS_T` sets the system time, `RD_SYS_T` reads the system time. You can also set the system time online using the programming device.

The local time (displayed time) is set by addition of a correction factor which can also be negative. Adjustment is carried out when parameterizing the CPU with the device configuration editor or during runtime with `SET_TIMEZONE`. The local time can be used to visualize time zones. It is read with `RD_LOC_T`.

#### Configuring the local time

The time zone and the switching over between daylight saving and standard time is set in the properties of the CPU: select the CPU in the device configuration, and open the *Time-of-day* section in the *Properties* tab in the inspector window. Set the time zone (local time), check the *Activate daylight saving time* box, specify the time difference between daylight saving and standard time, and also the conversion dates (Fig. 5.16).

#### `WR_SYS_T` Set system time

`WR_SYS_T` (Write System Time) sets the CPU's clock to the value specified by the IN parameter (Fig. 5.17). This value does not include the local time and the daylight saving/standard time ID. The error information is output in the `RET_VAL` parameter (0 = no error). In the event of an error, `ENO` is set to signal state “0”.

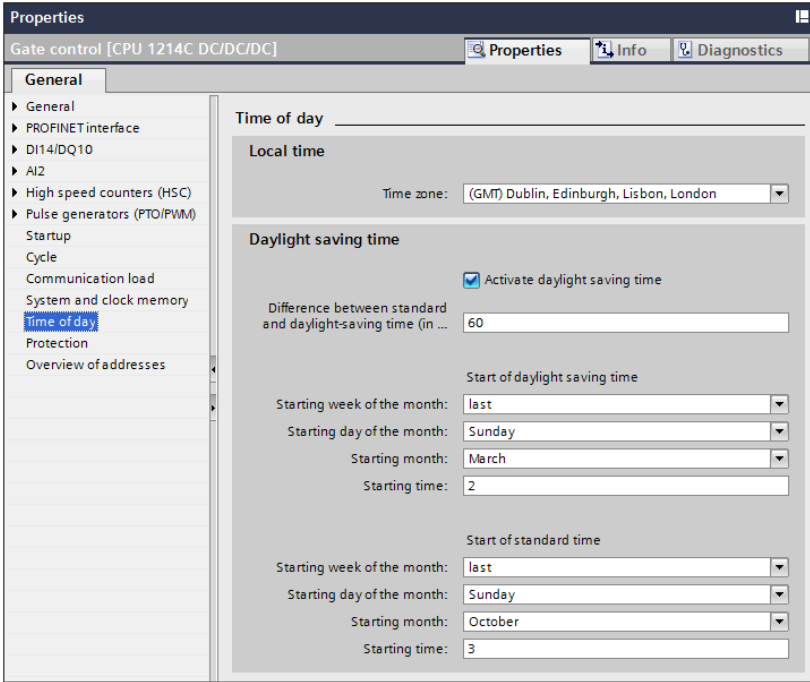


Fig. 5.16 Parameterization of local time and daylight saving/standard time switchover

### Set and read system time

#### Set system time

**LAD**

```

graph LR
    subgraph WR_SYS_T_DTL [WR_SYS_T DTL]
        EN[EN]
        ENO[ENO]
        IN[IN]
        RET_VAL[RET_VAL]
    end
    subgraph FBD [WR_SYS_T DTL]
        EN_RET_VAL[EN RET_VAL]
        IN_ENO[IN ENO]
    end
    EN --- EN_RET_VAL
    ENO --- EN_RET_VAL
    IN --- IN_ENO
    RET_VAL --- IN_ENO
        
```

**SCL**

```
#var_... := WR_SYS_T(#var_...);
```

Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
IN	INPUT	DTL	Date and time
RET_VAL	RETURN	INT	Error information

**Function:** Calling with EN = "1" sets the system time to the value present in parameter IN.

---

#### Read system time

**LAD**

```

graph LR
    subgraph RD_SYS_T_DTL [RD_SYS_T DTL]
        EN[EN]
        ENO[ENO]
        RET_VAL[RET_VAL]
        OUT[OUT]
    end
    subgraph FBD [RD_SYS_T DTL]
        EN_RET_VAL[EN RET_VAL]
        OUT_ENO[OUT ENO]
    end
    EN --- EN_RET_VAL
    ENO --- EN_RET_VAL
    RET_VAL --- OUT_ENO
    OUT --- OUT_ENO
        
```

**SCL**

```
#var_... := RD_SYS_T(#var_...);
```

Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
RET_VAL	RETURN	INT	Error information
OUT	OUTPUT	DTL	Date and time

**Function:** Calling with EN = "1" outputs the system time in parameter OUT.

Fig. 5.17 Set and read system time

### RD\_SYS\_T Read system time

RD\_SYS\_T (Read System Time) reads the CPU's actual system time and outputs it in the OUT parameter (Fig. 5.17). This value does not include the local time and the daylight saving/standard time ID. The error information is output in the RET\_VAL parameter (0 = no error). In the event of an error, ENO is set to signal state "0".

### SET\_TIMEZONE Set time zone

SET\_TIMEZONE sets the time zone and the switch-over between daylight saving and standard time (Fig. 5.18). The result is the local time calculated from the system time.

#### Set and read local time

##### Set a time zone

**LAD**

```

EN      ENO
---|---
REQ     DONE
---|---
TimeZone  BUSY
---|---
          ERROR
          ---|---
          STATUS
          ---|---
```

**FBD**

```

EN      DONE
---|---
REQ     BUSY
---|---
TimeZone  ERROR
---|---
          STATUS
          ---|---
          ENO
          ---|---
```

**SCL**

```

SET_TIMEZONE (
  REQ      := ... ,
  TIMEZONE := ... ,
  DONE     => ... ,
  BUSY     => ... ,
  ERROR    => ... ,
  STATUS   => ... );
```

Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
REQ	INPUT	BOOL	Job initiation
TimeZone	INPUT	*)	Local time rule
DONE	OUTPUT	BOOL	Job finished
BUSY	OUTPUT	BOOL	Being processed
ERROR	OUTPUT	BOOL	Error occurred
STATUS	OUTPUT	WORD	Error information

**Function:** SET\_TIMEZONE sets the local time on the CPU based on a correction factor for the system time and the daylight saving/standard time switchover.

\*) The *TimeTransformationRule* data type contains the rules for setting the local time and is explained in the text.

---

##### Read local time

**LAD**

```

EN      ENO
---|---
          RET_VAL
          ---|---
          OUT
          ---|---
```

**FBD**

```

EN      RET_VAL
---|---
          OUT
          ---|---
          ENO
          ---|---
```

Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
RET_VAL	RETURN	INT	Error information
OUT	OUTPUT	DTL	Date and time

**SCL**

```
#var_... := RD_SYS_T(#var_...);
```

**Function:** RD\_LOC\_T outputs the local time at the OUT parameter.

Fig. 5.18 Set and read local time

Table 4.13 on page 115 shows the data structure of the used system data type *TimeTransformationRule*.

### **RD\_LOC\_T Read local time**

RD\_LOC\_T (Read Local Time) reads the CPU's current local time and outputs it in the OUT parameter (Fig. 5.18). The local time is calculated from the system time, the local time zone, and the daylight saving/standard time ID. The local time zone and the daylight saving/standard time ID are set in the properties of the CPU in the device configuration editor. The error information is output in the RET\_VAL parameter (0 = no error). In the event of an error, ENO is set to signal state "0".

### **Calculating with date and time**

You can link the date and time together using further system functions, for example to generate the difference between two times of day or to add a duration to a specific point in time. The system functions available are described in the Chapter 11.4 "Arithmetic functions for time values" on page 369.

### **Setting the time on the CPU module online**

You can read and set the system time (module time) on the CPU online using the programming device. To do this, open the project and start the *Online & diagnostics* editor under the PLC station in the project tree. To establish the online mode, click in the toolbar of the Project view on the *Go online* symbol or on the *Go online* button in the *Online access* section of the diagnostics window.

Select the *Set time of day* command in the *Functions* section of the diagnostics window. The current time of the programming device is then displayed. For the module time (system time on the CPU), you can either select the programming device time, or you can set a separate system time on the CPU module.

### **Time synchronization**

The CPU module time can be synchronized over Ethernet. A time server is required which synchronizes the time of further stations in the network using the NTP procedure.

Activate the time synchronization in the properties of the PROFINET interface using the hardware configuration. To do this, select the PROFINET interface in the device configuration, and select the *Time synchronization* command in the properties in the Inspector window. Check the *Enable time-of-day synchronization using NTP mode* box, enter the IP addresses of the servers involved, and select the updating interval.

#### **5.6.7 Runtime meter**

An runtime meter counts the hours while running. You can use the runtime meter, for example, to record the CPU runtime or to determine the operating hours of connected devices.

A CPU 1200 has 10 runtime meters with a value range of 32 bits ( $2^{31}-1$  hours). An runtime meter also stops when the CPU is at STOP; if the CPU restarts, the runtime



meter must be restarted if required. The count value of an runtime meter is even retained on restart and after a general reset.

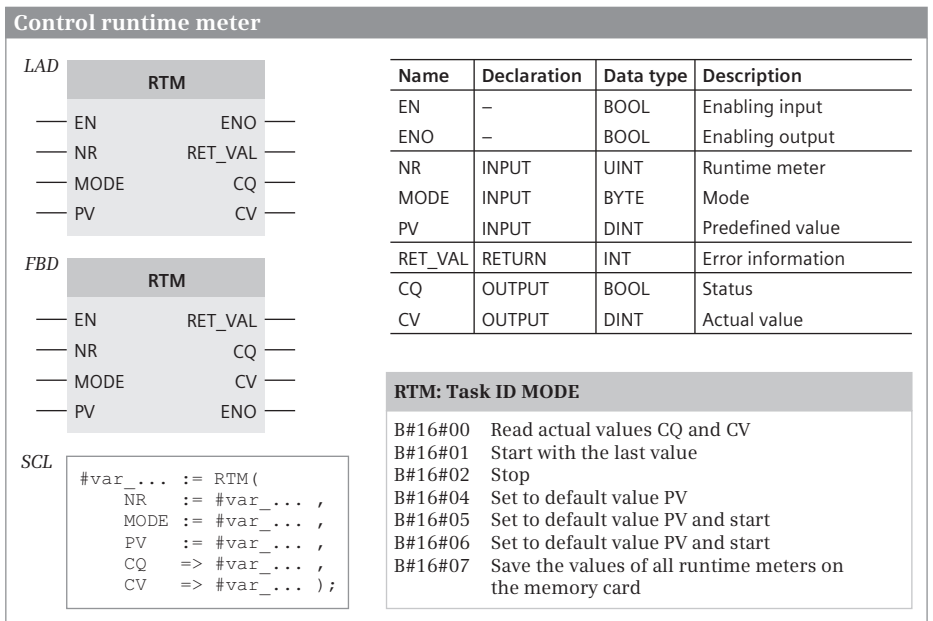
The RTM function controls an runtime meter. If the maximum duration has been reached, the runtime meter remains stationary and signals an overflow with the value W#16#8082 at the parameter RET\_VAL. An runtime meter can only be set to a new value or zero using the RTM function.

**RTM Control runtime meter**

RTM controls an runtime meter. Fig. 5.19 shows the graphic representation of the system function.

RTM controls the runtime meter whose number is specified in the NR parameter. The MODE parameter defines the function to be executed. The value to which the runtime meter is to be set (default value or start value in hours) is present in the PV parameter. The parameter CQ signals with signal state “1” that the runtime meter is running. The current value in hours is present in the CV parameter. CQ and CV are updated by the job ID MODE = B#16#00.

RTM can write the values of all runtime meters of the CPU to the memory card so that they are retained even if the backup voltage fails or a module is swapped. You should avoid frequent writing to the memory card, as there is a physical limit to the number of write operations.



**Fig. 5.19** System block for controlling the runtime meter

## 5.7 Interrupt processing

### 5.7.1 Introduction to interrupt processing

Interrupt processing is event-driven program execution. When such an event occurs, the operating system interrupts execution of the main program and calls the routine allocated to this particular event. Once this routine has been processed, the operating system resumes execution of the main program at the point of interruption. Such an interruption can take place after every operation (statement).

Applicable events may be interrupts and errors. A priority scheduler controls the execution order if interrupt events occur virtually simultaneously.

Each routine associated with an interrupt event is written in an organization block in which further blocks can be called. An event of higher priority interrupts execution of the routine in an organization block with a lower priority. You can influence the interruption of a program by events of higher priority using system functions (Chapter 5.7.6 “Delay and enable interrupts” on page 166).

### Events

The responses of the operating system are based on events. If an organization block is assigned to the event, the block is called when the event occurs. If calling is not possible at this moment, the event is placed in the queue that corresponds to its priority.

If no organization block is assigned to an event, the preset system response is carried out when the event occurs. Table 5.4 lists those events which cannot be assigned to an organization block.

**Table 5.4** Events without calling of organization block

Event class	Priority	Event	System response
Remove/insert	21	Hot swapping of a module in the central controller	STOP
		Hot swapping of a module in the distributed I/O	Ignore
Access error	22	I/O access error during updating of process image	Ignore
Programming error	23	Programming error in a block without local error handling	Ignore
I/O access error	24	I/O access error in a block without local error handling	Ignore
Maximum cycle time exceeded twice	27	Maximum cycle (monitoring) time exceeded twice in a process cycle	STOP

## Processing priorities

Table 5.5 shows the priority classes available for a CPU 1200 and the associated events which result in calling of an organization block.

**Table 5.5** Organization blocks of a CPU 1200

Priority class					
	Priority		Event class	OB number	Quantity
		Queue (size)			
1	1	1	Cycle (program cycle)	OB 1, >= OB 123	One start event for the main program, several OBs permissible
	1	1	Startup (startup)	OB 100, >= OB 123	One start event for the startup program, several OBs permissible
2	3	8	Time-delay interrupt (time-delay interrupt)	OB 20 to OB 23, >= OB 123	Total of 4 events: max. 4 start points for delays, 1 OB per event; max. 4 adjustable time intervals, 1 OB per event
	4	8	Cyclic interrupt (cyclic interrupt)	OB 30 to OB 38, >= OB 123	
	5	32	Hardware interrupt (hardware interrupt)	OB 40 to OB 47, >= OB 123	16 incoming interrupts, 16 outgoing interrupts, 1 OB per interrupt
	6	16	High-speed counter HSC (hardware interrupt)	OB 40 to OB 47, >= OB 123	6 events CV=PV, 6 changes in direction, 6 external resets, 1 OB per interrupt
	9	8	Diagnostics interrupt (diagnostic error interrupt)	OB 82	1 diagnostics event
3	26	8	Time error (time error interrupt)	OB 80	1 OB start error, 1 cycle time violation, 1 queue violation

The startup program belongs to the same priority class as the main program: the CPU's operating system prevents them from being called simultaneously. Interrupt events occurring during the startup phase are saved in a queue and processed prior to the main program following the transition to the RUN mode.

If several events of the same type occur so rapidly in succession that processing cannot keep up, they are saved in a queue and processed one after the other. Each type of event has its own queue. If the queue is full, the organization block OB 80 *Time error* is called when the next event occurs.

The assignment of an event to a priority class, the priority of the event, and the size of the queues are fixed variables.

A program can interrupt another program if the associated event belongs to a higher priority class. Programs in the same priority class – even if they have different priorities – cannot mutually interrupt.

Programs with the same priority are processed in the sequence in which the associated events have occurred. If an interrupt program cannot be processed because

a program of higher priority is currently being processed, the event is entered into the queue. The associated program is then processed when the program of higher priority has been completed, and if no other event of higher priority is present.

The main program has the lowest processing priority. Only one interrupt organization block can be processed at a time, which can then be interrupted by the organization block OB 80 *Time error*.

### Types of interrupt

The CPU 1200 provides the following interrupt events (interrupts):

- ▷ Time-delay interrupt  
An interrupt generated when a certain period of time has passed; a system function defines the time at which this period begins
- ▷ Cyclic interrupt  
An interrupt generated by the operating system at periodic intervals
- ▷ Hardware interrupt  
An interrupt from a module, either via an input derived from a process signal or generated on the module itself (e.g. by a high-speed counter HSC).

Other interrupt events are the time error (Section 5.8.3 “Time error OB 80” on page 168) and the diagnostics interrupt (Section 5.8.6 “Diagnostics interrupt OB 82” on page 176).

### Current signal states

In an interrupt routine it is sometimes necessary to work with the current signal states of the I/O modules and not with the signal states of the inputs that were updated at the start of the main program. The fetched signal states are then written directly to the I/O without waiting until the output process image has been updated at the end of the main program.

The operand area I/O permits direct access to the signal states on the module terminals (Section 4.1 “Operands and tags” on page 79). Note that the signal states on the module terminals change asynchronous to the cyclic program execution. It is therefore recommendable to maintain a strict separation between the main program and the interrupt routine.

### 5.7.2 Time-delay interrupts

A time-delay interrupt implements a delay time independent of the timer functions and asynchronous to cyclic program execution. With a CPU 1200, the organization blocks OB 20 to OB 23 and OB 123 and higher are set aside for processing a time-delay interrupt.

Such an organization block is assigned to event class *Time-delay interrupt*. It is of hardware data type *OB\_Delay*. The *System constants* tab of the default tag table lists the names and values of the constants. The name of the constant can be changed in the block properties.

## Using a time-delay interrupt

A time-delay interrupt is started by calling the system function `SRT_DINT`; this system function also passes on the delay interval and the delay organization block. When the delay interval has expired, the corresponding organization block is called.

The time between calling `SRT_DINT` and starting the organization block is a maximum of one millisecond less than the configured delay time providing that no interrupt events delay the call.

You can also use the `CAN_DINT` function to cancel execution of a time-delay interrupt that has not yet started. The associated organization block is then no longer called.

Calling of a time-delay interrupt OB can be delayed or enabled using the `DIS_AIRT` and `EN_AIRT` functions.

You must not use more than a total of four time-delay interrupt and cyclic interrupt organization blocks in your program.

Time-delay interrupt organization blocks do not have any startup information.

## Behavior during startup

During a startup, the operating system deletes all settings you have programmed for time-delay interrupts.

You can start a time-delay interrupt in the startup program by calling `SRT_DINT`. Following expiry of the delay time, the CPU must be in the `RUN` operating mode in order to process the corresponding organization block. If this is not the case, the CPU waits with the OB call until the startup has been completed and then calls the time-delay interrupt OB before the first statement in the main program.

## Configuring time-delay interrupts

Configuration of the time-delay interrupts is carried out in two steps:

- ▷ First create a user organization block for a time-delay interrupt.
- ▷ Then program the `SRT_DINT` function and possibly the `CAN_DINT` and `QRY_DINT` functions and assign the number of the time-delay interrupt OB to the `OB_NR` parameter.

To create an organization block for a time-delay interrupt, open the project in the Project view. In the project tree, double-click on *Add new block* under *Program blocks*. Click on *Organization block (OB)* in the *Add new block* window, and then on the start event *Time delay interrupt*. Assign a meaningful name to the OB, define the programming language, and select *Manual* if the OB is to have a number different

from the automatically assigned one. You can call further blocks (FB and FC) up to a nesting depth of four in the time-delay interrupt OB.

Insert the SRT\_DINT function in your program – this can be found in the program elements catalog under *Extended statements* and *Interrupts*. Then click on the selection symbol in the input box of the OB\_NR parameter and select the time-delay interrupt OB from the list. You program the CAN\_DINT and QRY\_DINT functions in the same manner.

Control time-delay interrupt																																
<b>Start time-delay interrupt</b>																																
<b>LAD</b>																																
<b>FBD</b>																																
		<b>SCL</b>	<pre>#var_... := SRT_DINT(OB_NR := #var_... ,                     DTIME := #var_... ,                     SIGN := #var_... );</pre>																													
				<table border="1"> <thead> <tr> <th>Name</th> <th>Declaration</th> <th>Data type</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>EN</td> <td>-</td> <td>BOOL</td> <td>Enabling input</td> </tr> <tr> <td>ENO</td> <td>-</td> <td>BOOL</td> <td>Enabling output</td> </tr> <tr> <td>OB_NR</td> <td>INPUT</td> <td>OB_DELAY</td> <td>Delay OB</td> </tr> <tr> <td>DTIME</td> <td>INPUT</td> <td>TIME</td> <td>Delay time</td> </tr> <tr> <td>SIGN</td> <td>INPUT</td> <td>WORD</td> <td>--- (not relevant)</td> </tr> <tr> <td>RET_VAL</td> <td>RETURN</td> <td>INT</td> <td>Error information</td> </tr> </tbody> </table>	Name	Declaration	Data type	Description	EN	-	BOOL	Enabling input	ENO	-	BOOL	Enabling output	OB_NR	INPUT	OB_DELAY	Delay OB	DTIME	INPUT	TIME	Delay time	SIGN	INPUT	WORD	--- (not relevant)	RET_VAL	RETURN	INT	Error information
Name	Declaration	Data type	Description																													
EN	-	BOOL	Enabling input																													
ENO	-	BOOL	Enabling output																													
OB_NR	INPUT	OB_DELAY	Delay OB																													
DTIME	INPUT	TIME	Delay time																													
SIGN	INPUT	WORD	--- (not relevant)																													
RET_VAL	RETURN	INT	Error information																													
<b>Cancel time-delay interrupt</b>																																
<b>LAD</b>																																
<b>FBD</b>																																
		<b>SCL</b>	<pre>#var_... := CAN_DINT(#var_... );</pre>																													
				<table border="1"> <thead> <tr> <th>Name</th> <th>Declaration</th> <th>Data type</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>EN</td> <td>-</td> <td>BOOL</td> <td>Enabling input</td> </tr> <tr> <td>ENO</td> <td>-</td> <td>BOOL</td> <td>Enabling output</td> </tr> <tr> <td>OB_NR</td> <td>INPUT</td> <td>OB_DELAY</td> <td>Delay OB</td> </tr> <tr> <td>RET_VAL</td> <td>RETURN</td> <td>INT</td> <td>Error information</td> </tr> </tbody> </table>	Name	Declaration	Data type	Description	EN	-	BOOL	Enabling input	ENO	-	BOOL	Enabling output	OB_NR	INPUT	OB_DELAY	Delay OB	RET_VAL	RETURN	INT	Error information								
Name	Declaration	Data type	Description																													
EN	-	BOOL	Enabling input																													
ENO	-	BOOL	Enabling output																													
OB_NR	INPUT	OB_DELAY	Delay OB																													
RET_VAL	RETURN	INT	Error information																													
<b>Query time-delay interrupt</b>																																
<b>LAD</b>																																
<b>FBD</b>																																
		<b>SCL</b>	<pre>#var_... := QRY_DINT(OB_NR := #var_... ,                     STATUS =&gt; #var_... );</pre>																													
				<table border="1"> <thead> <tr> <th>Name</th> <th>Declaration</th> <th>Data type</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>EN</td> <td>-</td> <td>BOOL</td> <td>Enabling input</td> </tr> <tr> <td>ENO</td> <td>-</td> <td>BOOL</td> <td>Enabling output</td> </tr> <tr> <td>OB_NR</td> <td>INPUT</td> <td>OB_DELAY</td> <td>Delay OB</td> </tr> <tr> <td>RET_VAL</td> <td>RETURN</td> <td>INT</td> <td>Error information</td> </tr> <tr> <td>STATUS</td> <td>OUTPUT</td> <td>WORD</td> <td>Status</td> </tr> </tbody> </table>	Name	Declaration	Data type	Description	EN	-	BOOL	Enabling input	ENO	-	BOOL	Enabling output	OB_NR	INPUT	OB_DELAY	Delay OB	RET_VAL	RETURN	INT	Error information	STATUS	OUTPUT	WORD	Status				
Name	Declaration	Data type	Description																													
EN	-	BOOL	Enabling input																													
ENO	-	BOOL	Enabling output																													
OB_NR	INPUT	OB_DELAY	Delay OB																													
RET_VAL	RETURN	INT	Error information																													
STATUS	OUTPUT	WORD	Status																													

**Fig. 5.20** Start, cancel, and query a time-delay interrupt

### **System functions for editing a time-delay interrupt**

You can control a time-delay interrupt using the following functions:

- ▷ SRT\_DINT Start time-delay interrupt
- ▷ CAN\_DINT Cancel time-delay interrupt
- ▷ QRY\_DINT Querying the status of a time-delay interrupt

Calling of these functions is shown in Fig. 5.20. You can also connect a constant or a tag of data type WORD or INT to the OB\_NR parameter if the *IEC check* attribute is not activated.

#### **SRT\_DINT Start time-delay interrupt**

SRT\_DINT starts a time-delay interrupt. The function call is simultaneously the start time for the parameterized period. Once the delay time has expired, the CPU calls the parameterized OB. You can set the delay time in intervals of 1 ms. The accuracy of the delay time is also 1 ms.

The SIGN parameter has no significance with a CPU 1200, but it must nevertheless be supplied. You can assign it a constant value of zero (0), for example.

Note that processing of a time-delay interrupt OB may be delayed if organization blocks of higher priority are being processed when the OB is called. The call is also delayed by use of the DIS\_AINT function.

You can overwrite a current delay time by a new value by calling SRT\_DINT again. The new delay time then commences when the function is called.

#### **CAN\_DINT Cancel time-delay interrupt**

CAN\_DINT cancels a started time-delay interrupt. The organization block selected by the OB\_NR parameter is not called in this case.

#### **QRY\_DINT Querying the status of a time-delay interrupt**

QRY\_DINT queries the status of a time-delay interrupt. The time-delay interrupt is specified with the parameter OB\_NR. The STATUS parameter contains the desired information and the individual bits have the significance shown in Table 5.6.

#### **Error response**

If the time-delay interrupt OB is not present in the user program when called, the operating system signals a program execution error which can be processed using the GET\_ERR\_ID and GET\_ERROR functions.

If the delay time has expired and the associated OB is still being processed, the operating system calls OB 80 *Time error*. The error is ignored if OB 80 is not present.

### 5.7.3 Cyclic interrupts

A cyclic interrupt is an interrupt triggered at periodic intervals and initiates execution of a cyclic interrupt organization block. A cyclic interrupt allows you to periodically execute a particular routine independent of the processing time of the cyclic program. With a CPU 1200, the organization blocks OB 30 to OB 38 and OB 123 and higher are set aside for processing the cyclic interrupts.

**Table 5.6** STATUS parameter of system function QRY\_DINT

Bit	Meaning with signal state "0"	Meaning with signal state "1"
0	The CPU is in the RUN mode.	The CPU is in the STARTUP mode.
1	The interrupt is enabled.	The interrupt has been delayed by DIS_AIRT.
2	The interrupt expired or is not active.	The interrupt is active.
3	Always "0"	
4	An OB with the number OB_NR does not exist.	An OB with the number OB_NR is loaded.
Other	Always "0"	

A cyclic interrupt organization block is assigned to event class *Cyclic interrupt*. It is of hardware data type *OB\_Cyclic*. The *System constants* tab of the default tag table lists the names and values of the constants. The name of the constant can be changed in the block properties.

#### Using cyclic interrupts

The properties of a cyclic interrupt include the time interval and the phase offset. The values can be set between 1 ms and 60 000 ms in increments of 1 ms. The time base and the phase offset are entered in the block properties of the cyclic interrupt OB.

You can use a total of four time-delay interrupt and cyclic interrupt organization blocks in your program.

Calling of a cyclic interrupt OB can be delayed or enabled using the DIS\_AIRT and EN\_AIRT functions.

Cyclic interrupt organization blocks do not have any startup information.

#### Behavior during startup

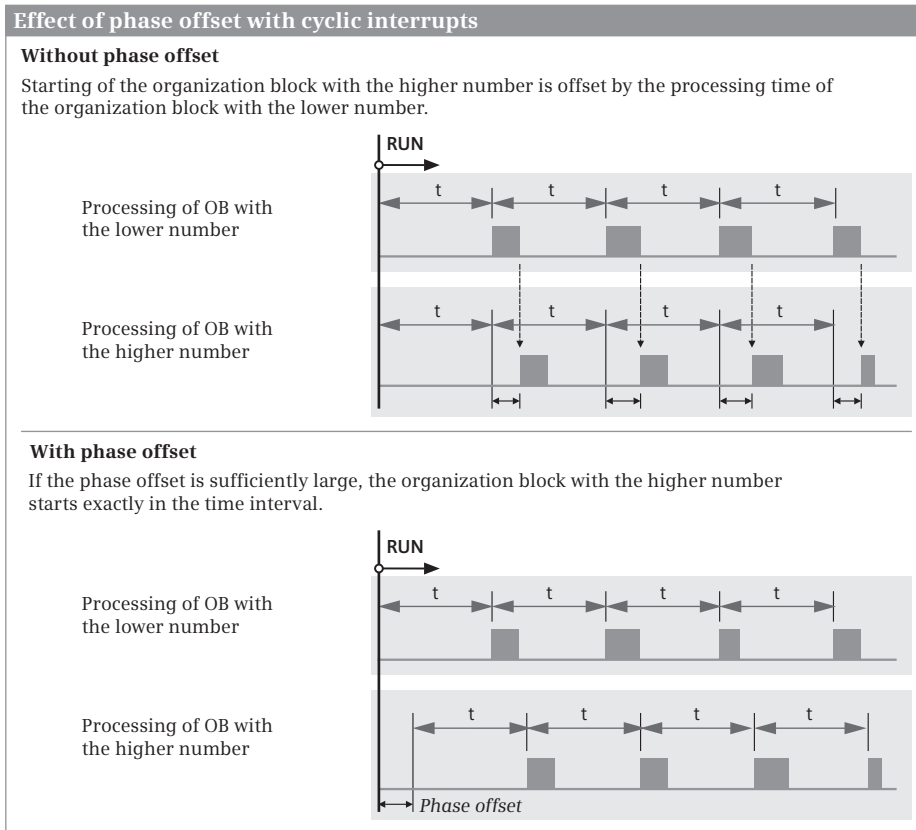
Processing of cyclic interrupts is not possible in the startup program. The time intervals only commence upon transition to the RUN mode.

#### Phase offset

The phase offset is used to process cyclic interrupt programs in a precise time frame even if they have the same time interval or a common multiple thereof. This results in higher accuracy of the processing intervals.



The start time of the time interval and the phase offset is the transition from the STARTUP state to RUN. The call instant for a cyclic interrupt OB is thus the time interval plus the phase offset. An example is shown in Fig. 5.21: No phase offset is set in the upper part, therefore the start of processing of the organization block with the higher number is shifted in each case by the respective processing time of the organization block with the lower number.



**Fig. 5.21** Effect of phase offset with cyclic interrupts

If, on the other hand, a phase offset is configured which is greater than the maximum processing time of the organization block with the lower number, the organization block with the higher number is exactly processed in the time base.

### Configuring cyclic interrupts

You configure a cyclic interrupt by creating a cyclic interrupt OB: First open your project in the Project view. In the project tree, double-click on *Add new block* under

*Program blocks.* Click on *Organization block (OB)* in the *Add new block* window, and then on *Cyclic interrupt*.

Assign a meaningful name to the OB, define the programming language, and select *Manual* if the OB is to have a number different from the automatically assigned one.

You can call further blocks (FB and FC) up to a nesting depth of four in the cyclic interrupt OB.

The time base and the phase offset are entered in the block properties. With the cyclic interrupt OB open, click *Cyclic interrupt* in the Inspector window under *Properties* and enter the time base and phase offset in milliseconds.

### **System functions for editing a cyclic interrupt**

You can set and query the parameters for processing a cyclic interrupt with the following functions:

- ▷ SET\_CINT Set cyclic interrupt parameters
- ▷ QRY\_CINT Query cyclic interrupt parameters

Calling of these functions is shown in Fig. 5.22. You can also connect a constant or a tag of data type WORD or INT to the OB\_NR parameter if the *IEC check* attribute is not activated.

#### **SET\_CINT Set cyclic interrupt parameters**

SET\_CINT sets the parameters for a cyclic interrupt. These are the interval with which the cyclic interrupt is triggered, and the phase offset. Enter the time interval in microseconds at the CYCLE parameter. If the time interval is zero, the cyclic interrupt organization block specified in parameter OB\_NR is not called. The phase offset at parameter PHASE is also specified in microseconds.

#### **QRY\_CINT Query cyclic interrupt parameters**

QRY\_CINT reads the parameters of the cyclic interrupt organization block specified at parameter OB\_NR and outputs them to the parameters CYCLE (time interval) and PHASE (phase offset). The operating mode of the selected cyclic interrupt organization block is output at parameter STATUS (Table 5.7).

### **Error response**

If the cyclic interrupt OB is not present in the user program when called, the operating system signals a program execution error which can be processed using the GET\_ERR\_ID and GET\_ERROR functions.

The processing time of a cyclic interrupt organization block must be significantly shorter than its time frame. If the associated cyclic interrupt is repeated during an ongoing cyclic interrupt OB, the operating system calls OB 80 *Time error*. The error is ignored if OB 80 is not present.

### Set and query parameters for processing a cyclic interrupt

**Set parameters**

*LAD*

```

graph TD
    subgraph SET_CINT_LAD [SET_CINT]
        EN --- ENO
        OB_NR --- RET_VAL
        CYCLE --- CYCLE
        PHASE --- PHASE
    end
    
```

*FBD*

```

graph TD
    subgraph SET_CINT_FBD [SET_CINT]
        EN --- ENO
        OB_NR --- OB_NR
        CYCLE --- RET_VAL
        PHASE --- ENO
    end
    
```

Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
OB_NR	INPUT	OB_CYCLE	Cyclic-interrupt OB
CYCLE	INPUT	UDINT	Time interval (µs)
PHASE	INPUT	UDINT	Phase offset
RET_VAL	RETURN	INT	Error information

*SCL*

```
#var_... := SET_CINT (OB_NR := #var_... ,
                    CYCLE := #var_... ,
                    PHASE := #var_... );
```

---

**Query parameters**

*LAD*

```

graph TD
    subgraph QRY_DINT_LAD [QRY_DINT]
        EN --- ENO
        OB_NR --- RET_VAL
        CYCLE --- CYCLE
        PHASE --- PHASE
        STATUS --- STATUS
    end
    
```

*FBD*

```

graph TD
    subgraph QRY_DINT_FBD [QRY_DINT]
        EN --- RET_VAL
        OB_NR --- CYCLE
        PHASE --- PHASE
        STATUS --- STATUS
        ENO --- ENO
    end
    
```

Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
OB_NR	INPUT	OB_CYCLE	Delay OB
RET_VAL	RETURN	INT	Error information
CYCLE	OUTPUT	UDINT	Time interval (µs)
PHASE	OUTPUT	UDINT	Phase offset
STATUS	OUTPUT	WORD	Status

*SCL*

```
#var_... := QRY_CINT (OB_NR := #var_... ,
                    CYCLE => #var_... ,
                    PHASE => #var_... ,
                    STATUS => #var_... );
```

**Fig. 5.22** Set and query cyclic interrupt parameters

**Table 5.7** STATUS parameter of system function QRY\_CINT

Bit	Meaning with signal state "0"	Meaning with signal state "1"
0	The CPU is in the RUN mode.	The CPU is in the STARTUP mode.
1	The interrupt is enabled.	The interrupt has been delayed by DIS_AIRT.
2	The interrupt expired or is not active.	The interrupt is active.
3	Always "0"	
4	An OB with the number OB_NR does not exist.	An OB with the number OB_NR is loaded.
Other	Always "0"	

### 5.7.4 Process interrupts

A hardware interrupt summarizes events in the controlled process or a module and responds immediately with a corresponding program. With a CPU 1200, the organization blocks OB 40 to OB 47 and OB 123 and higher are set aside for processing a cyclic interrupt.

A hardware interrupt organization block is assigned to event class *Hardware interrupt*. It is of hardware data type *OB\_HWINT*. The *System constants* tab of the default tag table lists the names and values of the constants. The name of the constant can be changed in the block properties.

You can use up to 50 independent process interrupt OBs. Only one hardware interrupt OB can be assigned to a hardware interrupt event, but several events can be assigned to one hardware interrupt OB.

#### Triggering a hardware interrupt

Possible sources of hardware interrupt events are:

- ▷ All onboard digital input channels of the CPU module
- ▷ Plus the digital input channels on the signal board and
- ▷ The signals of a high-speed counter HSC (CV = PV, change in counting direction, and external reset)

Triggering of a hardware interrupt is initially disabled by default. You can enable the processing of a process interrupt when parameterizing using the device configuration editor. With the digital input channels you can select whether the hardware interrupt is to be triggered by an incoming event, an outgoing event, or by both.

If, during processing of a hardware interrupt OB, an event occurs on the same channel of the same module which would again trigger the freshly processed hardware interrupt, this process interrupt is lost. A new hardware interrupt is only detected when processing of the old hardware interrupt has been completed. If the event to which the same process interrupt OB is assigned occurs on a different channel of the same module or on a different module, the operating system starts the organization block again following processing of the process interrupt OB.

#### Using the hardware interrupt

Hardware interrupt OBs are only called in the RUN mode.

Hardware interrupt organization blocks do not have any startup information.

At runtime, the assignment between a hardware interrupt event and an organization block can be made or removed.

Calling of a hardware interrupt OB can be delayed or enabled using the *DIS\_AIRT* and *EN\_AIRT* functions.

## Behavior during startup

The modules do not generate hardware interrupt events in STARTUP mode. Interrupt handling commences with the transition to the RUN operating mode. Hardware interrupts present during the transition are lost.

### Activate hardware interrupt event

If an onboard input of the CPU module is to trigger a hardware interrupt, start the device configuration editor, select the CPU module, and change in the Inspector window to the *Properties* tab. In the *Digital inputs* section, select the associated channel and activate the hardware interrupt.

If a digital input on the signal board is to trigger a hardware interrupt, select the signal board, and change in the Inspector window to the *Properties* tab. Under *Digital inputs* in the *Dlx/DOx* section, select the associated channel and activate the process interrupt.

If a high-speed counter is to trigger a hardware interrupt, select the CPU module, and change in the Inspector window to the *Properties* tab. Select the associated counter in the *High-speed counters (HSC)* section, activate it under *General*, and set the counter function. Under *Event configuration*, you can activate the hardware interrupt.

When activating the hardware interrupt, you can assign a name to the event and create the associated process interrupt OB: Open the *Hardware interrupt* input box and select the hardware interrupt OB in the window (if it has already been created), or create a new one using *Add object*.

### Hardware interrupt organization block

Open a project in the Project view. In the project tree, double-click on *Add new block* under *Program blocks*. Select *Organization block (OB)* and the associated event class *Hardware interrupt*. Assign a meaningful name to the organization block, and change the programming language and the number of the block if necessary.

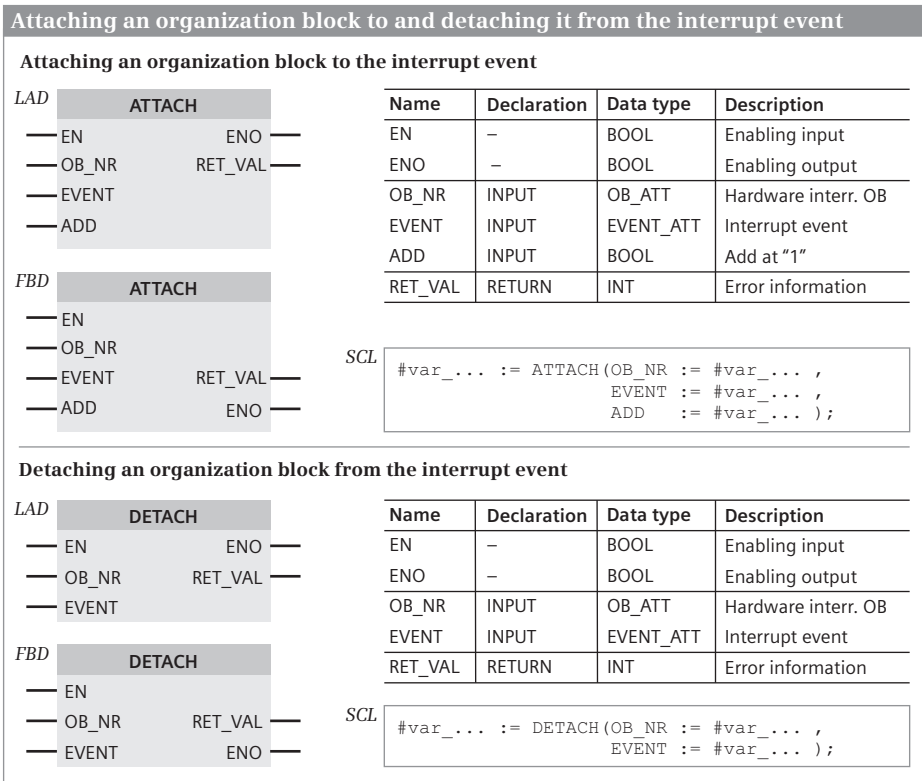
### Error response

If the hardware interrupt OB is not present in the user program when called, the operating system signals a program execution error which can be processed using the `GET_ERR_ID` and `GET_ERROR` functions.

#### 5.7.5 Assigning interrupts during runtime

With the following functions you can assign an organization block to an interrupt event during runtime and cancel the assignment again:

- ▷ `ATTACH` Assign organization block to the interrupt event
- ▷ `DETACH` Remove organization block from the interrupt event



**Fig. 5.23** Attaching an organization block to and detaching it from the interrupt event

Calling of these functions is shown in Fig. 5.23. You can also connect a constant or a tag of data type WORD or INT to the OB\_NR parameter if the *IEC check* attribute is not activated. You can also connect a constant or a tag with a 32-bit wide data type to the EVENT parameter if the *IEC check* attribute is not activated.

### **ATTACH Assign organization block to the interrupt event**

ATTACH assigns an interrupt organization block to an interrupt event. The event must be activated and defined using the device configuration editor. The interrupt organization block with the event class suitable to the event must be present in the user program.

Once the assignment has been made, the organization block is called and processed when the event occurs. The ADD parameter defines whether the previous assignments to other events are to be retained (with "1" or TRUE) or canceled (with "0" or FALSE).

The enabling output ENO has the signal state "0" with the errors: OB not present (RET\_VAL = 8090), OB is of wrong type (RET\_VAL = 8091), or event does not exist (RET\_VAL = 8093).

## DETACH Remove organization block from the interrupt event

DETACH removes the assignment of an interrupt event to an interrupt organization block.

If an event is specified in the EVENT parameter, the assignment of this event is canceled. If the EVENT parameter is "0", all assignments to the OB present on parameter OB\_NR are canceled.

If the requested assignment does not exist, the enabling output ENO has the signal state "0" and a value of 1 is output in the RET\_VAL parameter. Further errors are: OB not present (RET\_VAL = 8090), OB is of wrong type (RET\_VAL = 8091), or event does not exist (RET\_VAL = 8093).

### 5.7.6 Delay and enable interrupts

The following program functions are available for delaying and enabling interrupts:

- ▷ DIS\_AIRT Delay interrupt events
- ▷ EN\_AIRT Enable delayed interrupt events

Calling of these functions is shown in Fig. 5.24.

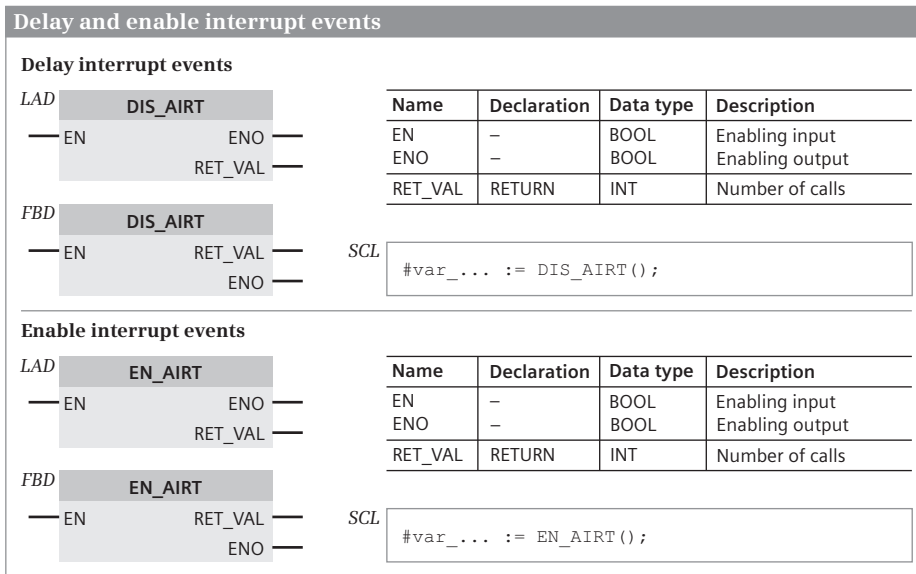


Fig. 5.24 Delay and enable interrupt events

### DIS\_AIRT Delay interrupt events

Following calling of the DIS\_AIRT function, the program in the current organization block (in the current priority class) is not interrupted by an interrupt event of higher priority should this occur. The interrupts are processed with a delay, i.e. the

operating system saves the interrupt events occurring during the delay and only processes them when the delay has been canceled. No interrupts are lost.

The delay in processing is retained until the end of processing of the current organization block or until the EN\_AIRT function is called.

You can call several DIS\_AIRT functions in succession. The RET\_VAL parameter indicates the (new) number of calls. You must then call EN\_AIRT exactly as often as DIS\_AIRT so that the processing of all interrupts is enabled again.

### **EN\_AIRT Enable delayed interrupt events**

The EN\_AIRT function enables processing of the interrupts again which have been delayed with DIS\_AIRT. You must call EN\_AIRT exactly as often as you previously called DIS\_AIRT in the current organization block or in the blocks called within this organization block.

The RET\_VAL parameter indicates the (still remaining) number of effective delays. If RET\_VAL is equal to 0, processing of all interrupts has been enabled again.

## **5.8 Troubleshooting, diagnostics**

### **5.8.1 Causes of errors and responses**

The CPU can detect and signal errors in the program execution.

These errors include:

- ▷ Errors in arithmetic operations (overflow, invalid floating-point number)
- ▷ Errors when calling blocks (block does not exist or is still being processed, program execution error)
- ▷ Errors when addressing the peripheral inputs and outputs (access errors)

In the event of “serious” errors, e.g. cycle monitoring time expired twice in a program cycle, the CPU directly enters the STOP mode.

Errors which are module-based are signaled by the diagnostics function. This can be carried out via the ERROR LED on the front of the CPU, via an entry in the diagnostics buffer, or by starting the diagnostics interrupt (see Chapter 13.3 “Hardware diagnostics” on page 436).

### **Error handling in general**

The occurrence of an error can be signaled by the following responses:

- ▷ The ENO output of a program function is set to “0” if the function has been executed incorrectly or not at all.
- ▷ The operating system starts the standard error handling, e.g. with a “fatal” error it enters the STOP mode or calls organization block OB 80 *Time error*.
- ▷ The operating system relinquishes the standard error handling and applies the (block) local error handling instead.



### 5.8.2 Error display with the ENO output

System functions for which an error can occur during processing and callable logic blocks (functions, function blocks) have an enable output ENO, with which the error in the calling program can be reported. An error is present if the ENO output has signal state “0” or is FALSE.

For LAD and FBD, the ENO output is represented as an output parameter at the call box. For SCL, the ENO output is not represented by default. If you want to use it, add scanning of the ENO output to the last position in the parameter list. You can scan the ENO output in the calling block and respond to the error (Chapter 12.4 “EN/ENO mechanism” on page 417).

In a self-written block, transfer an error message to the ENO output with a block end function (Chapter 12.2 “Block end function” on page 412). Examples of detected errors in the user program are diagnostics alarms, e.g. wire break at an analog input, or malfunction of the controlled machine, e.g. the simultaneous assignment of two conflicting limit switches.

### 5.8.3 Time error OB 80

The operating system calls the **organization block OB 80** *Time error* when one of the following events occurs:

- ▷ Cycle monitoring time exceeded
- ▷ OB request error: the requested organization block is still being processed (possible with time-delay and cyclic interrupts) or an organization block is requested too frequently within a given priority class (queue overflow)
- ▷ An interrupt is lost due to interrupt overload.

The time error organization block is assigned to event class *Time error interrupt*. It is of hardware data type *OB\_TIMEERROR*. The *System constants* tab of the default tag table lists the name and value of the constant. The name of the constant can be changed in the block properties.

Only one time error organization block can be programmed.

The error is ignored if OB 80 is not present when a time error occurs.

#### Start information

Table 5.8 shows the start information for OB 80. The tags in the start information are transferred to the *Input* section of the block interface and are present in the temporary local data.

**Table 5.8** Start information for OB 80

Variable	Data type	Description
fault_id	BYTE	Error ID B#01: Maximum cycle time exceeded B#02: Called OB is still executing B#07: Queue overflow B#09: Interrupt loss due to high interrupt load
csg_OBnr	OB_ANY	Number of OB being processed at time of error
csg_prio	UINT	Priority of OB being processed at time of error

#### 5.8.4 Local error handling

You can program local error handling in organization blocks (OB), function blocks (FB) and functions (FC). This only applies to the corresponding block. The local error handling is neither accepted by the calling block nor passed on to the called block. If the local error handling is not programmed, the system settings apply when an error occurs (ignore error or STOP).

The default responses are as follows with the local error handling active:

- ▷ With a write error: the error is ignored, and program execution is continued.
- ▷ With a read error: the substitute value “0” or zero is read, and program execution is continued.
- ▷ With an execution error: execution of the faulty statement (function) is aborted, and program execution is continued with the next statement.

Local error handling is automatically activated by inserting the `GET_ERROR` or `GET_ERROR_ID` statement in the block, and indicated in the block properties by the *Handle errors within block* attribute (cannot be edited).

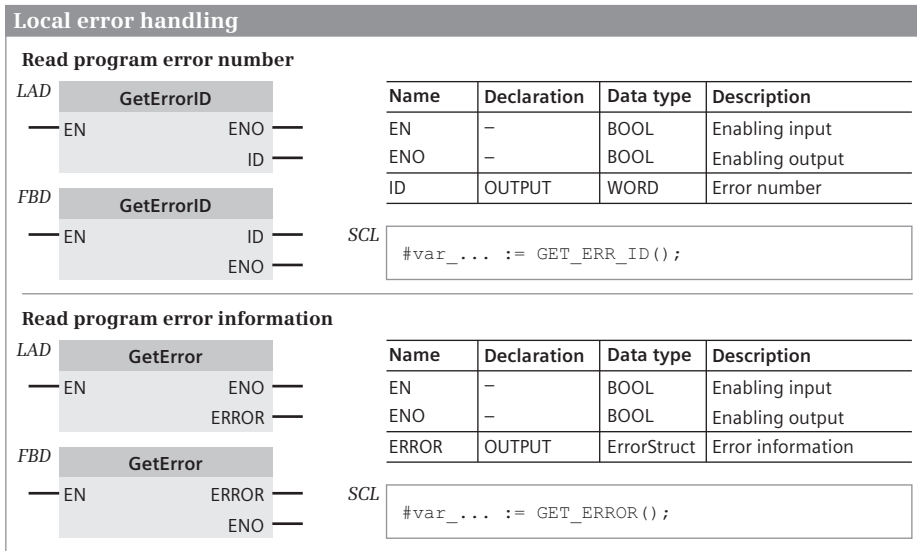
#### Evaluating program errors

Two functions are available for the evaluation of errors in the case of local error handling in the block (Fig. 5.25):

- ▷ `GET_ERR_ID` (read program error number) provides the error number (identification) when a program execution error occurs.
- ▷ `GET_ERROR` (read program error information) provides the corresponding information in a predefined data structure when a program execution error occurs.

In the event of a program execution error, the CPU enters the error into the diagnostics buffer as standard, and goes to STOP. If the `GET_ERROR` or `GET_ERR_ID` function is programmed in the block, there is neither an entry into the diagnostics buffer nor a change to STOP. Instead of this, the error is signaled by means of `GET_ERROR` or `GET_ERR_ID`.

The error may have occurred at any position between starting of the block and calling of `GET_ERROR` or `GET_ERR_ID`. Therefore, in the case of a single call of



**Fig. 5.25** Functions for local error handling

GET\_ERROR or GET\_ERR\_ID, the call is preferably positioned in the last network of the monitored block.

GET\_ERROR and GET\_ERR\_ID can also be called more than once. Calling of GET\_ERROR or GET\_ERR\_ID reinitiates error recording. The next call of GET\_ERROR or GET\_ERR\_ID outputs the first error following the previous call of GET\_ERROR or GET\_ERR\_ID. The sequence of the errors is not saved.

**GET\_ERR\_ID Read program error number**

In the event of a program execution error, the GER\_ERR\_ID function provides the error identification in the ID parameter (Table 5.9). The function is executed if EN has the signal state “1”. No error has been detected if ENO has the signal state “0” (FALSE), an error ID is present if the signal state at ENO is “1” (TRUE).

**GET\_ERROR Read program error information**

In the event of a program execution error, the GER\_ERROR function provides the error information in the ERROR parameter in data type *ErrorStruct*. The data type *ErrorStruct* has the structure shown in Section 4.8.5 “Data type ErrorStruct” on page 112. The function is executed if EN has the signal state “1”. No error has been detected if ENO has the signal state “0” (FALSE), error information is present if the signal state at ENO is “1” (TRUE).

**Error priority**

The first detected error is output when calling GET\_ERROR or GET\_ERR\_ID. If several errors occur simultaneously when processing a statement (function) they are

**Table 5.9** Error numbers with program execution errors

ERROR_ID		Fault	ERROR_ID		Fault
hex	dec		hex	dec	
16#2503	9475	Invalid pointer	16#253C	9532	Incorrect version, or function (FC) does not exist
16#2522	9506	Range violation when reading	16#253D	9533	System function (SFC) does not exist
16#2523	9507	Range violation when writing	16#253E	9534	Incorrect version, or function block (FB) does not exist
16#2524	9508	Invalid operand when reading	16#253F	9535	System function block (SFB) does not exist
16#2525	9509	Invalid operand when writing	16#2575	9589	Program nesting depth exceeded
16#2528	9512	Incorrect bit orientation when reading	16#2576	9590	Error in assignment of temporary local data
16#2529	9513	Incorrect bit orientation when writing	16#2942	10562	Read error during direct access (input channel does not exist)
16#2530	9520	Data block write error (DB read-only)	16#2943	10563	Write error during direct access (output channel does not exist)
16#253A	9530	Global DB does not exist			

output according to their priority (Table 5.10). Priority 1 is the highest priority, 12 is the lowest.

### Evaluating program error information

The data type *ErrorStruct* can be inserted into data blocks or into a block interface from a drop-down list. You can also insert the data type more than once if you assign a different name to the data structure each time. The data structure and the name of individual structure components cannot be changed.

**Table 5.10** Priorities during error output

Priority	Type of error	Priority	Type of error
1	Error in program code	7	Time or counter function does not exist
2	Reference missing	8	No write access to a DB
3	Invalid range	9	I/O error
4	DB does not exist	10	Statement does not exist
5	Operand is not compatible	11	Block does not exist
6	Width of specified range is insufficient	12	Invalid nesting depth

If the error information is saved in a data block, it can also be read by other blocks. For example, another block can be called in the event of an error which then takes over evaluation of the error information.

### 5.8.5 Diagnostic functions in the user program

The following functions are available to evaluate diagnostics alarms in the user program:

- ▷ LED                    Read status of an LED
- ▷ DeviceStates        Read status of distributed I/O devices
- ▷ ModuleStates        Read status of a central I/O modules
- ▷ GET\_DIAG            Read diagnostic information

#### LED Read status of an LED

LED reads the status of a module LED. The parameter LADDR specifies the module and the parameter LED specifies the LED. RET\_VAL indicates the current status of the specified LED. Fig. 5.26 shows the function call.

**Read status of an LED**

<p><b>LAD</b></p> <pre> EN      ENO  ----- ----- LADDR  RET_VAL  ----- ----- LED</pre> <p><b>FBD</b></p> <pre> EN  ----- LADDR  RET_VAL  ----- LED      ENO  ----- -----</pre>	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">Name</th> <th style="text-align: left;">Declaration</th> <th style="text-align: left;">Data type</th> <th style="text-align: left;">Description</th> </tr> </thead> <tbody> <tr> <td>EN</td> <td>-</td> <td>BOOL</td> <td>Enabling input</td> </tr> <tr> <td>ENO</td> <td>-</td> <td>BOOL</td> <td>Enabling output</td> </tr> <tr> <td>LADDR</td> <td>INPUT</td> <td>HW_IO</td> <td>Module ID</td> </tr> <tr> <td>LED</td> <td>INPUT</td> <td>UINT</td> <td>LED number</td> </tr> <tr> <td>RET_VAL</td> <td>RETURN</td> <td>INT</td> <td>LED status</td> </tr> </tbody> </table> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p>SCL</p> <pre>#var_... := LED (LADDR := #var_... ,                 LED := #var_... );</pre> </div>	Name	Declaration	Data type	Description	EN	-	BOOL	Enabling input	ENO	-	BOOL	Enabling output	LADDR	INPUT	HW_IO	Module ID	LED	INPUT	UINT	LED number	RET_VAL	RETURN	INT	LED status
Name	Declaration	Data type	Description																						
EN	-	BOOL	Enabling input																						
ENO	-	BOOL	Enabling output																						
LADDR	INPUT	HW_IO	Module ID																						
LED	INPUT	UINT	LED number																						
RET_VAL	RETURN	INT	LED status																						

Parameter LED		Parameter RET_VAL	
Value	LED	Value	LED status
1	STOP/RUN	0	LED does not exist
2	ERROR	1	Permanently switched off
3	MAINT	2	Color 1 permanently switched on (e.g. green for STOP/RUN LED)
4	redundant	3	Color 2 permanently switched on (e.g. orange for STOP/RUN LED)
5	Link (green)	4	Color 1 flashes at 2 Hz
6	Rx/Tx (yellow)	5	Color 2 flashes at 2 Hz
		6	Colors 1 and 2 flash alternately at 2 Hz
		7	LED is active, color 1
		8	LED is active, color 2
		9	LED exists, but no status information is available

If the RET\_VAL parameter displays the value 16#80xx, there is a parameterization error.

**Fig. 5.26** Read status of an LED

The module ID can be found in the *System constants* tab in the default tag table or the module properties in the *Properties* tab in the inspector window under *General* in the *Name* field. At the LADDR parameter, enter this name or select it from the drop-down list. You can also specify the numerical value of the module ID, which is provided as a constant or variable in the *System constants* tab.

### DeviceStates Read status of distributed I/O devices

DeviceStates reads the status of the I/O stations in a PROFINET IO system. At the LADDR parameter, you can enter the system ID of the PROFINET IO system. With the MODE parameter, you can select the type of status information that is displayed at the STATE parameter for all I/O stations. Fig. 5.27 shows the function call.

The system ID can be found either in the *System constants* tab in the default tag table or in the PROFINET IO system properties in the *Properties* tab in the inspector window under *General* in the *Name* field. At the LADDR parameter, enter this name or select it from the drop-down list. You can also specify the numerical value of the system ID, which is provided as a constant or variable in the *System constants* tab.

Via the MODE parameter you select the type of status information to be output at the STATE parameter (Fig. 5.27). With a bit set to signal state "1", the bit field at the STATE parameter shows that the selected status information applies to the affected station. Example: If you want to determine which stations are disrupted, assign the value 2 to the MODE parameter. Bit 0 at the STATE parameter has signal state "1" if

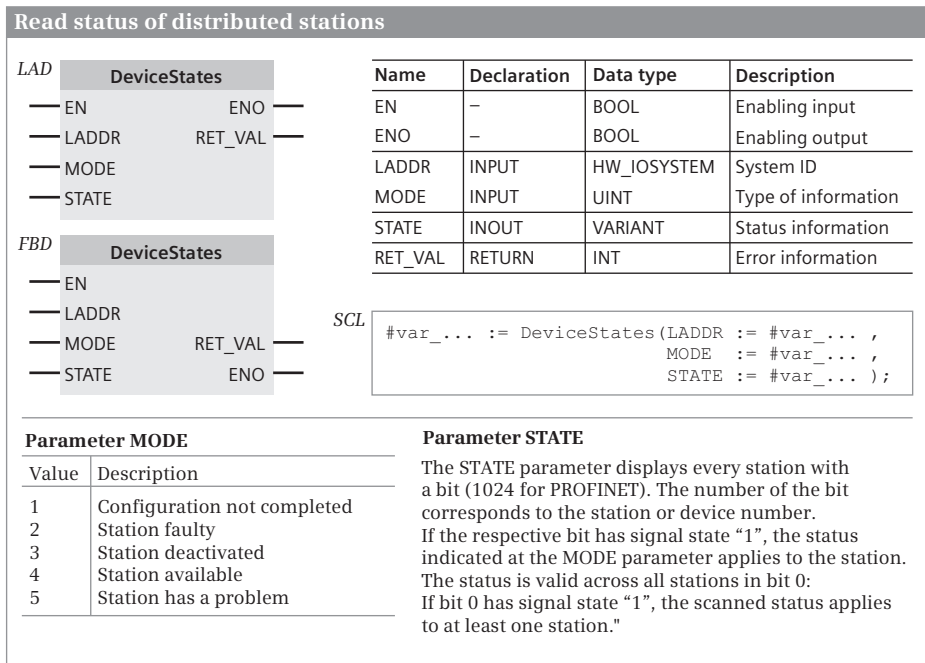


Fig. 5.27 Read status of distributed stations

at least one of the stations is disrupted. If bit 4 is set to signal state “1”, the station with device number 4 is disrupted.

The STATE parameter can be assigned to any tag or an operand area, for example, with P#DB10.DBX0.0 BYTE 128, i.e. 1024 bits in data block %DB10 from data byte %DBB0. If the tag or the area is too small, the status information is entered in the available length and error number 16#8452 is output at parameter RET\_VAL.

**ModuleStates Read status of a module**

ModuleStates reads the status of submodules in a module. At the LADDR parameter, specify the module ID and use the MODE parameter to select the type of status information that is output at the parameter STATE for all submodules of the module. Fig. 5.28 shows the function call.

The module ID can be found in either the *System constants* tab in the default tag table or in the properties of the module in the *Properties* tab in the inspector window. under *General* in the *Name* field. At the LADDR parameter, enter this name or select it from the drop-down list. You can also specify the numerical value of the module ID, which is provided as a constant or variable in the *System constants* tab.

Via the MODE parameter you select the type of status information to be output at the STATE parameter (Fig. 5.28). With a bit set to signal state “1”, the bit field at the

**Read status of a central module**

<p><b>LAD</b></p> <div style="border: 1px solid gray; padding: 5px; margin-bottom: 10px;"> <p style="text-align: center; background-color: #f0f0f0;"><b>ModuleStates</b></p> <p>— EN                      ENO —</p> <p>— LADDR                  RET_VAL —</p> <p>— MODE</p> <p>— STATE</p> </div> <p><b>FBD</b></p> <div style="border: 1px solid gray; padding: 5px;"> <p style="text-align: center; background-color: #f0f0f0;"><b>ModuleStates</b></p> <p>— EN</p> <p>— LADDR</p> <p>— MODE                  RET_VAL —</p> <p>— STATE                  ENO —</p> </div>	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">Name</th> <th style="text-align: left;">Declaration</th> <th style="text-align: left;">Data type</th> <th style="text-align: left;">Description</th> </tr> </thead> <tbody> <tr> <td>EN</td> <td>—</td> <td>BOOL</td> <td>Enabling input</td> </tr> <tr> <td>ENO</td> <td>—</td> <td>BOOL</td> <td>Enabling output</td> </tr> <tr> <td>LADDR</td> <td>INPUT</td> <td>HW_DEVICE</td> <td>Block ID</td> </tr> <tr> <td>MODE</td> <td>INPUT</td> <td>UINT</td> <td>Type of information</td> </tr> <tr> <td>STATE</td> <td>INOUT</td> <td>VARIANT</td> <td>Status information</td> </tr> <tr> <td>RET_VAL</td> <td>RETURN</td> <td>INT</td> <td>Error information</td> </tr> </tbody> </table> <div style="margin-top: 10px;"> <p><b>SCL</b></p> <pre style="border: 1px solid gray; padding: 5px;">#var_... := ModuleStates(LADDR := #var_... ,                         MODE := #var_... ,                         STATE := #var_... );</pre> </div>	Name	Declaration	Data type	Description	EN	—	BOOL	Enabling input	ENO	—	BOOL	Enabling output	LADDR	INPUT	HW_DEVICE	Block ID	MODE	INPUT	UINT	Type of information	STATE	INOUT	VARIANT	Status information	RET_VAL	RETURN	INT	Error information
Name	Declaration	Data type	Description																										
EN	—	BOOL	Enabling input																										
ENO	—	BOOL	Enabling output																										
LADDR	INPUT	HW_DEVICE	Block ID																										
MODE	INPUT	UINT	Type of information																										
STATE	INOUT	VARIANT	Status information																										
RET_VAL	RETURN	INT	Error information																										

<p><b>Parameter MODE</b></p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">Value</th> <th style="text-align: left;">Description</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Configuration not completed</td> </tr> <tr> <td>2</td> <td>Submodule faulty</td> </tr> <tr> <td>3</td> <td>Submodule deactivated</td> </tr> <tr> <td>4</td> <td>Submodule available</td> </tr> <tr> <td>5</td> <td>Submodule has a problem</td> </tr> </tbody> </table>	Value	Description	1	Configuration not completed	2	Submodule faulty	3	Submodule deactivated	4	Submodule available	5	Submodule has a problem	<p><b>Parameter STATE</b></p> <p>The STATE parameter displays every each submodule of a module with a bit (maximal 128). The number of the bit corresponds to the slot of the submodule in the module. If the respective bit has signal state “1”, the status indicated at the MODE parameter applies to the submodule. The status is valid across all submodules in bit 0: If bit 0 has signal state “1”, the scanned status applies to at least one submodule.</p>
Value	Description												
1	Configuration not completed												
2	Submodule faulty												
3	Submodule deactivated												
4	Submodule available												
5	Submodule has a problem												

**Fig. 5.28** Read status of a central module

STATE parameter shows that the selected status information applies to a submodule of the affected module. Example: If you want to determine which submodules are disrupted, assign the value 2 to the MODE parameter. Bit 0 of the STATE parameter has signal state “1” if at least one submodule is disrupted. If bit 2 is set to signal state “1”, the submodule at slot 2 is disrupted.

The STATE parameter can be assigned to any tag or an operand area, for example, with P#M512.0 BYTE 16, i.e. 128 bits from memory byte %MB512. If the tag or the area is too small, the status information is entered in the available length and error number 16#8452 is output at parameter RET\_VAL.

### GET\_DIAG Read diagnostic information

GET\_DIAG reads the diagnostic information of a hardware object. At the LADDR parameter, specify the ID of the hardware object. With the MODE parameter, select the type of diagnostic information that is output at the DIAG parameter. Fig. 5.29 shows the function call.

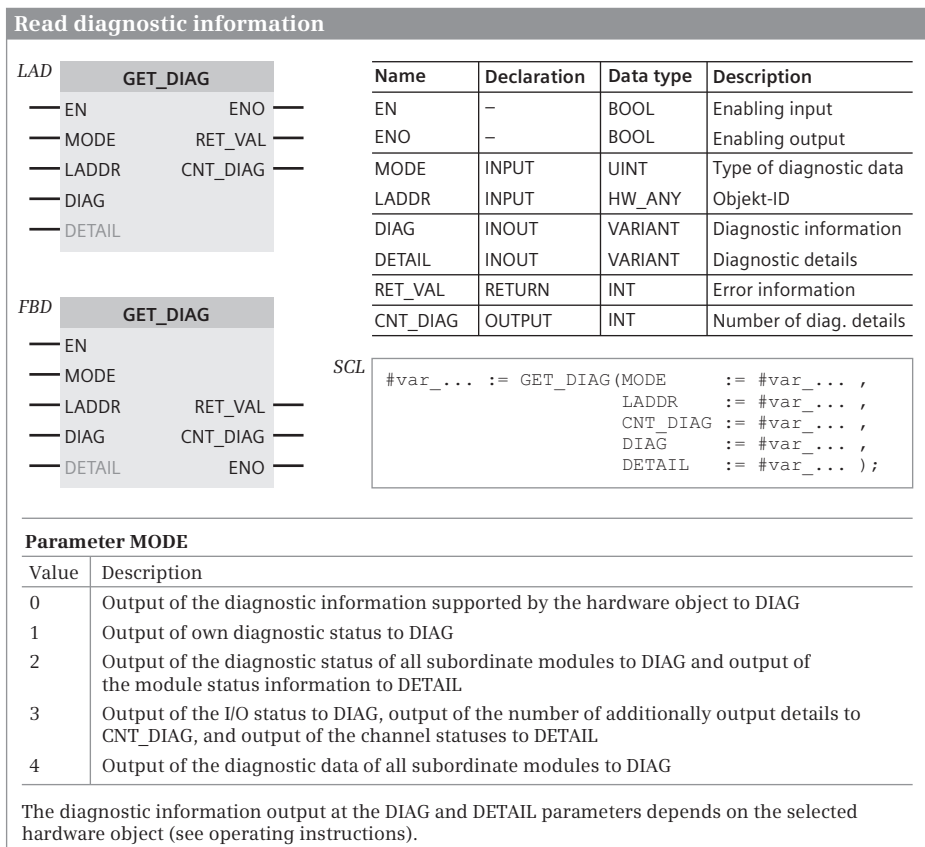


Fig. 5.29 Read diagnostic information



The object ID can be found either in the *System constants* tab in the default tag table or in the properties of the hardware object in the *Properties* tab in the inspector window under *General* in the *Name* field. At the LADDR parameter, enter this name or select it from the drop-down list. You can also specify the numerical value of the object ID, which is provided as a constant or variable in the *System constants* tab.

Via the MODE parameter you select the type of information to be output at the DIAG parameter (Fig. 5.29). The value 0 is used to query what diagnostic information the hardware object supports. Each bit set to signal state “1” at the DIAG parameter corresponds to an assignment of the MODE parameter: If the bit 1 is set, MODE = 1 is supported. If the bit 2 is set, MODE = 2 is supported, etc. CNT\_DIAG is set to value 0; DETAIL is not changed.

If MODE = 1, the diagnostic information of the selected hardware object is output at parameter DIAG. CNT\_DIAG is set to value 0, DETAIL is not changed.

If MODE = 2, the diagnostic status of all the modules in the hardware object is output at parameter DIAG. CNT\_DIAG is set to value 1, DETAIL contains module state information.

If MODE = 3, the state of the inputs and outputs of the selected hardware object is output at parameter DIAG. CNT\_DIAG is set to the number of module channels whose status data is output at the parameter DETAIL.

If MODE = 4, the state of the inputs and outputs of all modules in the hardware object is output at parameter DIAG. CNT\_DIAG is set to value 0, DETAIL is not changed.

### 5.8.6 Diagnostics interrupt OB 82

Appropriately designed modules can detect diagnostics events, for example “No load voltage present” (I/O modules), overshoot and undershoot, wire break and short-circuit (analog input modules). If the detection of a diagnostics event is activated in the device configuration editor, the **organization block OB 82 Diagnostics interrupt** is called when the event occurs.

The diagnostics interrupt organization block is assigned to event class *Diagnostic error interrupt*. It is of hardware data type *OB\_DIAG*. The *System constants* tab of the default tag table lists the name and value of the constant. The name of the constant can be changed in the block properties.

You can only use one diagnostics interrupt OB in your program.

The OB 82 is processed if no other interrupt organization block is active, otherwise the diagnostics event is entered into the queue.

When a diagnostics event occurs, it is written into the diagnostics buffer. If an OB 82 is not present when a diagnostics event occurs, the CPU ignores the event.

Calling of the diagnostics interrupt OB can be delayed or enabled using the DIS\_AIRT and EN\_AIRT functions.

## Start information

Table 5.11 contains the start information for OB 82. The tags of the start information are transferred to the *Input* section of the block interface and are present in the temporary local data.

**Table 5.11** Start information for OB 82

Variable	Data type	Description																				
IO_state	WORD	Contains the diagnostics state of the module with diagnostic capability																				
		<table border="1"> <thead> <tr> <th>Bit no.</th> <th>Meaning</th> <th>With signal state "0"</th> <th>With signal state "1"</th> </tr> </thead> <tbody> <tr> <td>Bit 0</td> <td>Configuration correct</td> <td>Configuration no longer correct</td> <td>Configuration correct</td> </tr> <tr> <td>Bit 4</td> <td>Fault</td> <td>Fault no longer present</td> <td>Fault present</td> </tr> <tr> <td>Bit 5</td> <td>Configuration not correct</td> <td>Configuration is correct again</td> <td>Configuration not correct</td> </tr> <tr> <td>Bit 6</td> <td>Access to I/O failed</td> <td>I/O is accessible again</td> <td>I/O access error present (the hardware ID is then in <i>laddr</i>)</td> </tr> </tbody> </table>	Bit no.	Meaning	With signal state "0"	With signal state "1"	Bit 0	Configuration correct	Configuration no longer correct	Configuration correct	Bit 4	Fault	Fault no longer present	Fault present	Bit 5	Configuration not correct	Configuration is correct again	Configuration not correct	Bit 6	Access to I/O failed	I/O is accessible again	I/O access error present (the hardware ID is then in <i>laddr</i> )
		Bit no.	Meaning	With signal state "0"	With signal state "1"																	
		Bit 0	Configuration correct	Configuration no longer correct	Configuration correct																	
		Bit 4	Fault	Fault no longer present	Fault present																	
Bit 5	Configuration not correct	Configuration is correct again	Configuration not correct																			
Bit 6	Access to I/O failed	I/O is accessible again	I/O access error present (the hardware ID is then in <i>laddr</i> )																			
laddr	HW_ANY	HW identification																				
channel	UINT	Channel number (starts at 0)																				
multi-error	BOOL	With signal state "1", more than one diagnostics event is present																				

The start information *laddr* provides the HW identification of the module or submodule which generated the diagnostics interrupt. Each hardware unit is provided in the configuration with an ID whose value is listed in the default tag table in the *System constant* tab. The names of the constants correspond to the names of the modules or submodules assigned during the hardware configuration.

## 6 Program editor

### 6.1 Introduction

This chapter describes how to work with the program editor, with which the user program is written in the programming languages LAD, FBD, and SCL. The special features of programming in the respective programming language are described in the Chapters 7 “Ladder logic LAD” on page 209, 8 “Function block diagram FBD” on page 246, and 9 “Structured Control Language SCL” on page 284.

The user program consists of blocks which are saved in the project tree under a PLC station in the *Program blocks* folder. Logic blocks contain the program code, and data blocks contain the control data. When programming, a block is initially created and subsequently filled with data or a program. The programming languages ladder logic (LAD), function block diagram (FBD), and structured control language (SCL) are available for programming the control function. You can define the programming language individually for each block.

The user program works with operands and tags. Block-local tags are declared during programming of the blocks, global operands and tags are present in the *PLC tags* folder. The *PLC data types* folder contains user-defined data structures for tags and data blocks.

Programming is appropriately commenced by definition of PLC tags and PLC data types. This is followed by the global data blocks with the already known data. For the logic blocks, one starts with those which are at the lowest position in the call hierarchy. The blocks in the next higher level in the hierarchy then call the blocks positioned below them. The organization blocks in the highest hierarchy level are created last.

The program editor makes various tools available to support you in program creation and testing. The cross-reference list contains the already programmed tags and blocks and the point in the program where they are used. The assignment list shows the current use of the inputs, outputs, and bit memories. The call and dependency structure shows the sequence the blocks are called.

Following completion, the user program is compiled, i.e. the program editor converts the data entered into a program which can be executed on the CPU.

### 6.2 PLC tag table

You work in the user program with operands which are e.g. inputs or outputs. These operands can be addressed in absolute mode (e.g. %I1.0) or symbolic mode (e.g. “Start signal”). Symbolic addressing uses names (identifiers) instead of the ab-

solute address. As well as the name, you define the data type of the operand. The combination of operand (absolute address, memory location), name, and data type is referred to as a “tag”.

When writing the user program, a distinction is made between *local* and *global* tags. A local tag is only known in the block in which it has been defined. You can use local tags with the same name in different blocks for different purposes. A global tag is known throughout the entire user program, and has the same meaning in all blocks. You define global tags in the PLC tag table.

Refer to Chapter 6.6.1 “Cross-reference list” on page 201 for how to create a cross-reference list of the PLC tags. Monitoring of tags using the PLC tag table is described in Chapter 13.4.4 “Monitoring with the PLC tag table” on page 445.

### 6.2.1 Creating and editing the PLC tag table

When creating a PLC station, a *PLC tags* folder with the PLC tag table is also created. You can open the PLC tag table by double-clicking on *Default tag table* in the *PLC tags* folder. The default tag table consists of the *Tags*, *User constants*, and *System constants* tabs.

You can create additional tag tables containing PLC tags and user constants with the *Add new tag table* function. These self-created tables can be renamed and organized in groups. A tag or a constant can only be defined in one of the tables. To obtain an overview of all tags and constants, double-click on *Show all tags* in the *PLC tags* folder.

You can save an incomplete or faulty PLC tag table at any time and process it again later. However, the tag table must be error-free to enable compilation of the user program.

### 6.2.2 Defining PLC tags

In the *Tags* tab, enter the name, data type, and address (operand, memory location) of the tags used. The name can contain letters, digits, and special characters (no quotation marks). It must not already have been assigned for a different PLC tag, a symbolically addressed constant, a PLC data type, or a block. No distinction is made between upper and lower case when checking the name.

You can add an explanatory comment to each defined tag. Table 6.1 contains the operands and data types permissible as PLC tags. The data type defines certain properties of the data associated with the name, basically the representation of the data content. An overview of the data types used with a CPU 1200 and the detailed description can be found in Chapter 4 “Variables and data types” on page 79.

Part of the operand area *Bit memory* can be set to retentive, i.e. this part retains the signal statuses and values following a power off and subsequent power on. To set the retain area, click in the toolbar of the PLC tags table on the *Retain* symbol and enter the number of retentive memory bytes in the dialog which is then displayed.

**Table 6.1** Approved operands and data types for PLC tags

Input operand	Output operand	Bit memory operand	Approved data types	Address range
Input bit I	Output bit Q	Memory bit M	BOOL	I, Q: 0.0...1023.7 M: 0.0...4095.7 1) M: 0.0...8191.7 2)
Input byte IB	Output byte QB	Memory byte MB	BYTE, CHAR, SINT, USINT	IB, QB: 0...1023 MB: 0...4095 1) MB: 0...8191 2)
Input word IW	Output word QW	Memory word FW	WORD, INT, UINT	IW, QW: 0...1022 MW: 0...4094 1) MW: 0...8190 2)
Input double word ID	Output double word QD	Memory double word MD	DWORD, DINT, UDINT, REAL, TIME	ID, QD: 0...1020 MD: 0...4092 1) MD: 0...8188 2)

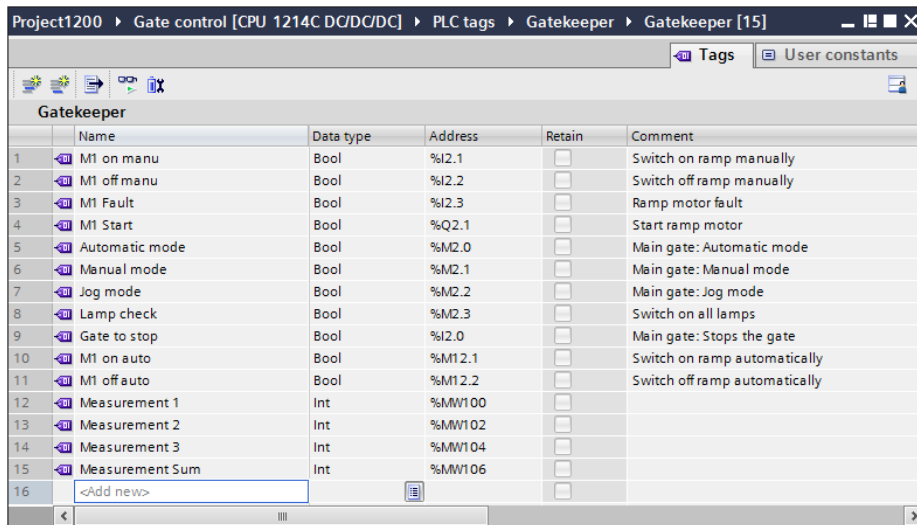
1) For CPU 1211 and CPU 1212 2) For CPU 1214 and CPU 1215

A tick in the *Retain* column then identifies which bit memory operands are retentive.

The properties of a PLC tag include the attributes

- ▷ *Accessible from HMI*  
When activated, an HMI station can access this tag during runtime.
- ▷ *Visible in HMI*  
When activated, this tag is visible by default in the selection list of an HMI station.

Fig. 6.1 shows an example of a PLC tag table.



**Fig. 6.1** Example of a PLC tag table

### 6.2.3 Editing a PLC tag table

You can use *Insert row* from the shortcut menu to insert an empty line above the selected line. *Add row* inserts a line after the selected line. The *Delete* command deletes the selected line. You can copy selected lines and add them to the end of the list. You can sort the lines according to the column contents by clicking the header of the appropriate column. Sorting is in ascending order with the first click, in descending order with the second click, and the original state is reestablished following the third click.

To fill out the table automatically, select the name of the tag to be transferred, position the cursor at the bottom right corner of the cell, and drag downward over the lines with the mouse button pressed.

If you enter the same name a second time, for example when copying lines, a consecutive number in parentheses is appended to the name. When filling out automatically, this is an underscore character with a consecutive number. Double assignment of an address is indicated by a colored background.

You can also supplement, change or delete the PLC tags when entering the user program (described in Chapter 6.3.6 “Editing tags” on page 192).

In addition to the default tag table, which is always available, you can create further PLC tag tables. In doing so, you may not give a tag multiple definitions. If you double-click on *Show all tags*, the program editor creates an overview of the tags of all of the tag tables.

You can compare a PLC tag table with one from another project if you mark the tag table and select the command *Tools > Compare > Offline/offline*.

### 6.2.4 Exporting and importing a PLC tag table

A PLC tag table can also be created or edited using an external editor. The external file is present in .xlsx format.

To export, open the PLC tag table and select the *Export* icon in the toolbar. Set the file name and path in the dialog, and select the data to be exported (tags or constants). The contents of the opened PLC tag table are exported. To export all PLC tags, open the complete table by double-clicking on *Show all tags* and then select the *Export* icon.

The external file contains the *PLC tags* worksheet for the PLC tags and the *Constants* worksheet for the symbolically addressed user constants (Table 6.2).

To import, double-click on *Show all tags* under the PLC station in the *PLC tags* folder in the project tree. Select the *Import* icon in the toolbar. Set the file name and path in the dialog, and select the data to be imported (tags or constants). The contents of the external file are imported into the tag table, which is specified in the Path column. Existing entries are identified by a consecutive number in parentheses appended to the name and/or by an address highlighted in color.

**Table 6.2** Columns in the external file for the PLC tag table

Worksheet <i>PLC tags</i>						
Name	Path	Data type	Logical address	Comment	Hmi Visible	Hmi Accessible
Name of PLC tag	Group and name of PLC tag table	Data type of tag	Absolute address (e.g. %I0.0)	Comments	TRUE or FALSE	TRUE or FALSE
Worksheet <i>Constants</i>						
Name	Path	Data type	Value	Comment		
Name of constant	Group and name of PLC tag table	Data type of constant	Default value	Comments		

### 6.2.5 Constants tables

The PLC tag table in the *User constants* tab contains symbolically addressed constant values which are valid throughout the CPU. You define a constant in the table in that you assign a name, data type, and fixed value to it and you can then use this constant in the user program with the symbolic name.

The constant name must not already have been assigned to another constant, a PLC tag, a PLC data type, or a block. The name can contain letters, digits, and special characters (but not quotation marks). No distinction is made between upper and lower case when checking the name.

In the *System constants* tab, the default tag table contains the object IDs created by the device configuration and the program editor. The data type of a constant indicates the application, the value of a constant specifies the object. The data type and the value are fixed, but you can change the name of the constant in the respective object properties.

Example: The ID for a high-speed counter in the CPU has the data type *Hw\_Hsc* and a value, for example, between 258 and 263, corresponding to one of the counters 1 to 6. The name of the constant is set in the device configuration editor in the properties of the high-speed counter.

The constants are used in the user program if a hardware or software object is to be addressed, for example a high-speed counter or an organization block: In the HSC parameter, the *CTRL\_HSC* function expects the ID for the high-speed counter to be controlled, and the *ATTACH* statement processes the organization block whose ID is specified by data type *OB\_ATT* in the *OB\_NR* parameter.

The data types of the system constants are combined under the term “Hardware data types”. Section 4.9 “Hardware data types” on page 115 includes an example of a constants table.

## 6.3 Programming a code block

### 6.3.1 Creating a new code block

A prerequisite for creating a new block is that you have created a project and a PLC station. First open the project. You can create a new block in either the Portal view or the Project view.

In the Portal view, click *PLC programming*. An overview window appears in which you can see the existing blocks. With a newly created project this is the organization block OB 1 with the name *Main* (main program). Click on *Add new block* to open the window for creating a new block.

In the Project view, the *Program blocks* folder is present in the project tree under the PLC station. This folder is created together with the PLC station. The *Program blocks* folder contains the *Add new block* editor. Double-click to open the window for creating a new block.

Then select the block type by clicking on the button with the corresponding symbol (Fig. 6.2). Assign a meaningful name to the new block. The name must not already have been assigned to a different block, a PLC tag, a symbolically addressed con-

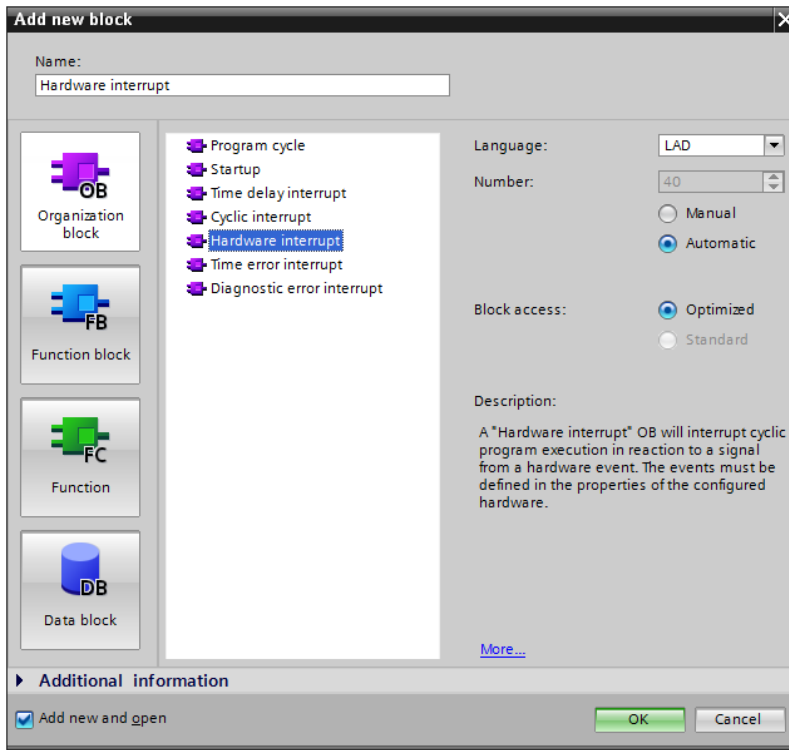


Fig. 6.2 Add new block window with organization block selected



stant, and a PLC data type. The name can contain letters, digits, and special characters (but not quotation marks). No distinction is made between upper and lower case when checking the name.

Then select the programming language for the block. With automatic assignment of the block numbers, the lowest free number for the type of block is displayed in each case. If you select the *Manual* option, you can enter a different number. The *Optimized* option activates the *Optimized block access* attribute and has effects on addressing, the retentivity, and the data types of the tags used in the block. Activation of the check box is recommended.

You must assign an event class to an organization block, i.e. you specify the type of organization block. Select the event class from the displayed list. Depending on the event class, the block number is either fixed or freely-selectable. You can create multiple organization blocks with different numbers for some event classes (see Chapter 5.7.1 “Introduction to interrupt processing” on page 153).

You set the default setting when creating a new block in the main menu in the Project view using the *Options > Settings* command in the *PLC programming* section. Under *General* and *Default setting for new blocks*, you can set the preselection for *IEC check* and *Optimized block access*.

If the *Add new and open* checkbox is activated in the *Add new block* window, the program editor is started and programming of the newly created block can begin.

### 6.3.2 Working area of program editor for code blocks

The program editor is automatically started when a block is opened. Open a block by double-clicking on its symbol: this can be found in the Portal view in the overview window of the PLC programming, and in the Project view in the *Program blocks* folder under the PLC station in the project tree.

You can adapt the properties of the program editor according to your requirements using the *Options > Settings* command in the main menu. Select the *PLC programming* section, and set the font size, layout and width of the operand field under *LAD/FBD*.

The program editor displays the opened block with interface and program in the working window (Fig. 6.3). Prior to programming, the block properties are present in the inspector window; during programming, the properties of the selected or edited object are present here. The task window contains the program elements catalog in the *Statements* task card.

The working window of the program editor shows the following details:

- ▷ The toolbar contains the symbols for the programming menu commands (e.g. *Insert network*, *Delete network*, *Go to next error*, etc.). The meaning of the symbols is displayed if you hold the mouse pointer over the symbol. Currently non-selectable icons are grayed out.

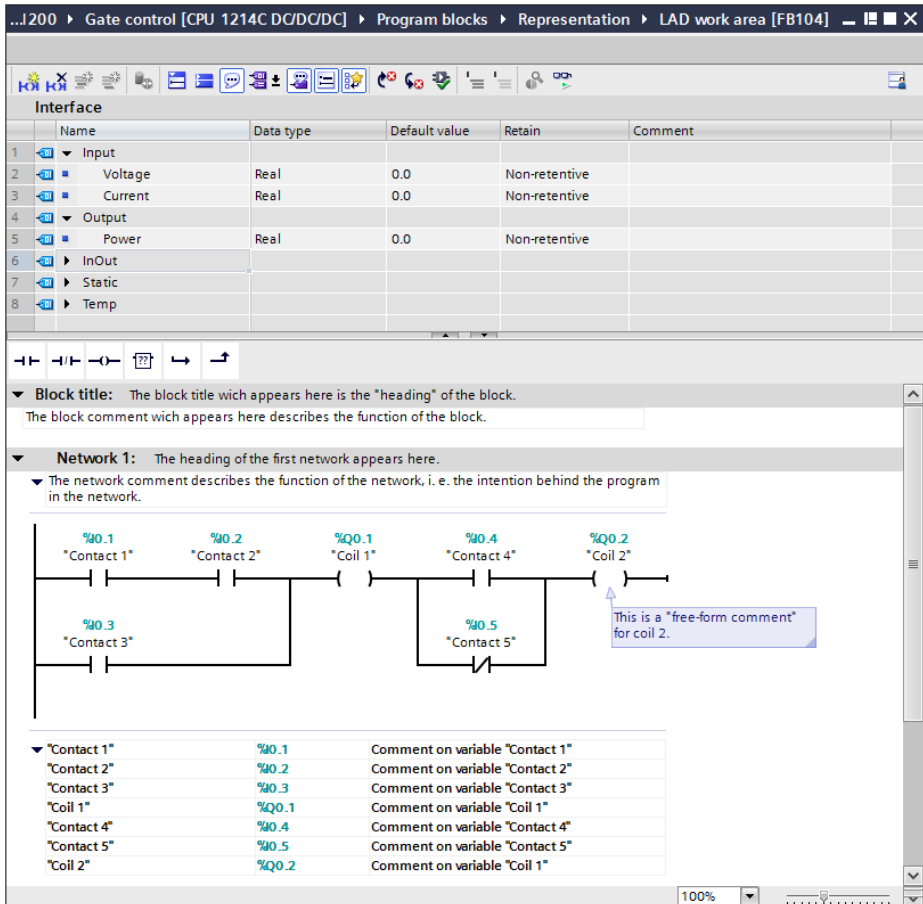


Fig. 6.3 Example of the program editor's working window in ladder logic

- ▷ The interface shows the block interface with the block parameters and the block-local tags.
- ▷ The favorites bar provides the favorite program elements (instructions), which can also be found in the *Favorites section* of the program elements catalog. You can activate and deactivate the display in the editor: Click with the right mouse button in the favorites catalog or favorites bar and activate or deactivate *Display favorites in editor*. To add a instruction to the favorites, select the instruction in the program elements catalog and drag and drop it with the mouse into the favorites catalog or favorites bar. To remove a instruction from the favorites, click with the right mouse button and then select *Remove instruction*.
- ▷ The block window contains the block program. Enter the control function of the block here.

The workspace is maximized by clicking on the *Maximize* icon in the title bar. Click on the *Embed* icon to embed it again. Display as a separate window is also possible: Click in the title bar on the icon for *Float* icon. Using the *Window > Split editor space vertically* and *Window > Split editor space horizontally* commands in the main menu, various opened objects can be displayed and edited in parallel, e.g. the PLC tag table and a block.

### 6.3.3 Specifying code block properties

To set the block properties, select the block in the *Program blocks* folder, followed by the *Edit > Properties* command in the main menu or the *Properties* command in the shortcut menu.

The block properties that can be changed are described in detail in Chapter 5.3.2 “Editing block properties” on page 128. You set the *Optimized block access* attribute when creating the block and then you can no longer change it. It is recommendable to set the *IEC check* attribute prior to block programming. A block can be protected against illegal access (“know-how protection”) and illegal use (“copy protection”).

### 6.3.4 Programming a block interface

The block interfaces of the logic blocks contain the declaration of the block-local tags. The interface structure depends on the type of block. Table 6.3 shows the individual declaration sections of the blocks. The meaning of the declaration modes is described in detail in Section 5.3.5 “Block interface” on page 133.

**Table 6.3** Declaration sections of the block interface

Declaration section	Meaning	Permissible with block type		
		OB (see text)	FC	FB
Input	Input parameters	OB (see text)	FC	FB
Output	Output parameters	–	FC	FB
InOut	In/out parameters	–	FC	FB
Static	Static local data	–	–	FB
Temp	Temporary local data	OB	FC	FB
Return	Function value	–	FC	–

You can increase or decrease the size of the block interface window by dragging on the bottom edge with the mouse. Two arrows at the bottom can be used to open and close the window. Fig. 6.4 shows an example of a function block interface.

You can click on the triangle to the left of the declaration mode to open the declaration section or to close it. If you select a line with the right mouse button, you can delete it in the shortcut menu, insert an empty line above it, or add an empty line after it.

	Name	Data type	Default value	Retain	Comment
1	Input				
2	Set	Bool	false	Non-retentive	Set tally counter to new value
3	Acknowledge	Bool	false	Non-retentive	Acknowledge counter error
4	Light barrier	Bool	false	Non-retentive	Rear light barrier
5	Number	UInt	0	Non-retentive	New value of the tally counter
6	Output				
7	Finished	Bool	false	Non-retentive	Counter value attained
8	Fault	Bool	false	Non-retentive	Counter error
9	InOut				
10	<Add new>				
11	Static				
12	Active	Bool	false	Non-retentive	Counter control is active
13	Light_barrier_ris	Bool	false	Non-retentive	Rising edge of light barrier
14	Light_barrier_fal	Bool	false	Non-retentive	Falling edge of light barrier
15	Active_ris	Bool	false	Non-retentive	Rising edge of "Counter control is active"
16	Set_ris	Bool	false	Non-retentive	Rising edge of "Set counter"
17	Acknowledge_ris	Bool	false	Non-retentive	Rising edge for "Acknowledge"
18	Tally counter	IEC_UCOUNTER		Non-retentive	
19	Monitoring	IEC_TIMER		Non-retentive	
20	Duration1	Time	T#0ms	Non-retentive	Time duration for "Light barrier covered"
21	Duration2	Time	T#0ms	Non-retentive	Time duration for "Gap"
22	Duration3	Time	T#0ms	Non-retentive	Time duration for "Monitoring active state"
23	Temp				

Fig. 6.4 Example of function block interface

The name can contain letters, digits, and special characters (but not quotation marks). No distinction is made between upper and lower case when checking the name. A drop-down list shows the currently permissible data types. You can use the comment to describe the purpose of the respective tag.

The *Default value* column is displayed for a function block (FB). You can enter a default value here which is saved in the instance data block. The *Retain* column is displayed in the interface of function blocks if the *Optimized block access* attribute is activated in the block. Here you can set the retentivity for individual tag values (*Non-retentive*, *Retentive*, *Set in IDB*). If *Optimized block access* is not activated, the retentivity can only be set in the instance data block, and only for the complete interface.

Neither a default value nor the retentivity can be set for the temporary local data.

The organization blocks OB 80 *Time error*, OB 82 *Diagnostics interrupt*, and OB 100 *Startup program* as well as all other organization blocks of event class *Startup* provide start information for the user program. Although this start information is located in the CPU's system memory like the temporary local data, it is shown in the declaration section *Input*.

In the case of a function (FC), the function value with the name *Ret\_Val* and data type VOID is displayed in the interface in the *Return* section. The function value has no significance when programming with ladder logic and function block diagram.

The data type VOID prevents the display in the call box. If you specify a different data type here, the function value is treated like the first output parameter. Using the SCL programming language, you can integrate a function in an expression instead of a tag with the data type of the function value (see Section “Using a function value of a function (FC)” on page 139).

For logic blocks with deactivated *Optimized block access* attribute, tags in the block interface can be overlaid with other data types (see Chapter 4.3.3 “Overlying tags (data type views)” on page 93).

### 6.3.5 Programming control functions

#### Working with networks

A network is part of a logic block which, in the case of the LAD and FBD programming languages, contains a complete current path or a complete logic operation. No networks are possible with SCL.

The program editor automatically numbers the networks starting from 1. You can assign a title and a comment to each network. When editing, you can directly select any network from the main menu using the *Edit > Go to > Network/line* command.

The networks can be opened or closed. To do this, select *Network* with the right mouse button and then select the *Collapse* or *Expand* command from the shortcut menu, or click in the toolbar of the working window on the *Close all networks* or *Open all networks* icon.

When programming the last network in each case, an empty network is automatically appended. To program a new network, select the *Insert > Network* command from the shortcut menu. The editor then adds an empty network after the currently selected network.

You can show or hide the network comments using the *Network comments on/off* icon in the toolbar or the *View > Display with > Network comments* command in the main menu.

Following each network, the tags used in the network can be shown with name, address, and comment. You can show or hide the tag information using the *Tag information on/off* icon in the toolbar or the *View > Display with > Tag information* command in the main menu.

The network comments and tag information view can be set for all blocks with the command *Options > Settings* in the main menu under the group *PLC Programming > View* by activating the relevant option *With comments* and/or *With tag information*.

## Program elements catalog

All program elements permissible for the respective programming language (contacts, coils, boxes, statements, etc.) can be found in the program elements catalog in the task window. The program elements catalog is divided into the following groups

- ▷ *Favorites* (frequently required program elements)
- ▷ *Basic instructions* (basic functions)
- ▷ *Extended instructions* (functions implemented by system blocks)
- ▷ *Technology* (technological functions, e.g. for PID controllers or high-speed counters)
- ▷ *Communication* (communication functions for data transmission and for communication modules).

You can combine a selection of frequently used program elements in the *Favorites* catalog and display them in the favorites bar of the program editor to allow rapid selection.

## General procedure when programming

To enter the program code, position the program elements in the desired arrangement and subsequently supply them with tags or enter the statement lines. The program editor immediately checks your inputs and indicates faulty entries.

You can interrupt block programming at any time – even if the program is still incomplete or faulty – and continue later. You can store a block by saving the complete project using the *Project > Save* command from the main menu.

You can save the structure of the windows and tables using the *Save window settings* icon in the top right corner of the working window. This structure is reestablished the next time the working window is opened.

## Programming a control function with ladder diagram (LAD)

To program the control function in LAD, select a program element in the catalog and drag it with the mouse into the open network under the network comment. The first program element is positioned automatically. With the next program element, small boxes indicate where the new program element may be positioned and – in green – where it is positioned when you “drop” it.

In the ladder logic, the binary logic operations are implemented by series and parallel connections (Fig. 6.5). With the ladder logic, the Q or ENO output is positioned in the display at the top edge of the box in order to be able to “hang” the box into the current path. The structure of an LAD current path is described in Chapter 7 “Ladder logic LAD” on page 209.

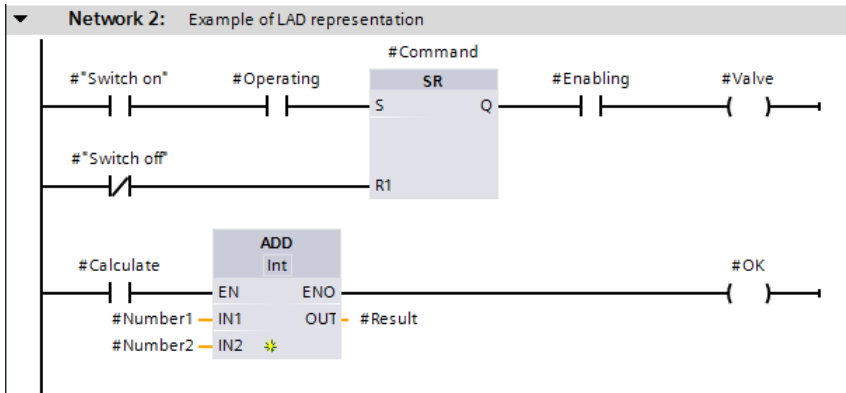


Fig. 6.5 Example of ladder logic

### Programming a control function with function block diagram (FBD)

To program the control function in FBD, select a program element in the catalog and drag it with the mouse into the open network under the network comment. The first program element is positioned automatically. With the next program element, small gray boxes indicate where the new program element may be positioned and – in green – where it is positioned when you “drop” it. You can also position program elements freely in the network and subsequently connect the corresponding inputs and outputs.

Binary logic operations are represented in the function block diagram by AND, OR, and exclusive-OR boxes (Fig. 6.6). The Q and ENO outputs are positioned at the bottom edge where they can be connected to the input of the following program element. The structure of an FBD logic operation is described in Chapter 8 “Function block diagram FBD” on page 246.

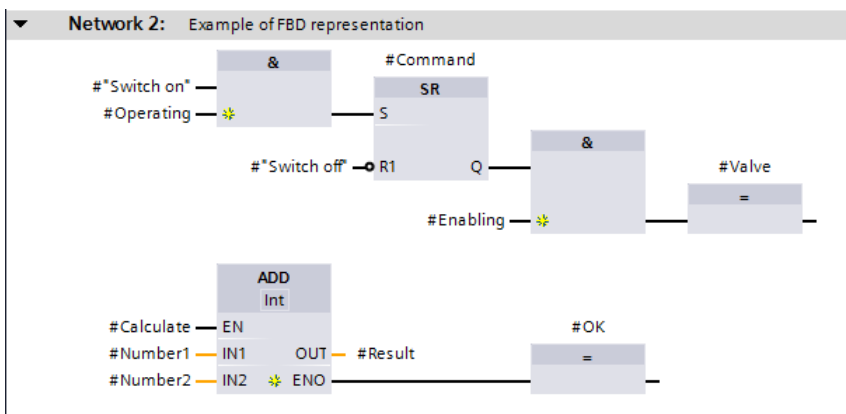


Fig. 6.6 Example of function block diagram

### Selection of function and data types with drop-down lists (LAD, FBD)

Many program elements have a variable design with regard to both function and data types. For example, if you select the ADD box from the mathematical functions, three question marks are shown underneath the function designation ADD instead of the data type. If you click on the ADD box, a small yellow triangle is displayed on the top right-hand corner as an indication that a drop-down list is present behind it. In this case, the drop-down list shows the data types permissible at this point, from which you can select the desired data type (Fig. 6.7).

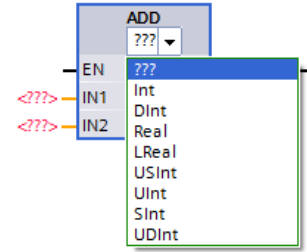


Fig. 6.7 Selection of data type using drop-down list

If a small yellow triangle is displayed in the top right corner of the program element (contact, coil, box), you can select a different function here for the program element from a drop-down list.

The empty box – which can be found in the favorites or in the program elements catalog under *General* – is particularly flexible here. Here you can select almost all program elements from the (function) drop-down list.

### Programming a control function with Structured Control Language (SCL)

The control function is entered in SCL as “structured text”. You can drag all statements from the program elements catalog into the working area. With basic instructions, for example a binary or digital operation, it is simpler to enter the statements with the keyboard.

Binary and digital logic operations are implemented in the SCL representation by expressions (Fig. 6.8). An expression is terminated by a semicolon. In the case of block calls and complex functions implemented as blocks, the block parameters are listed in parentheses following the function name. The structure of an SCL expression is described in Chapter 9 “Structured Control Language SCL” on page 284.

```

1 //Example of SCL representation
2 //-----
3 IF #"Switch off" THEN #Command := FALSE;
4 ELSIF #"Switch on" AND #Operating THEN #Command := TRUE;
5 END_IF;
6 #Valve := #Command AND #Enabling;
7
8 //Arithmetic expression with error handling
9 #OK := FALSE;
10 IF #Calculate THEN
11     ENO := TRUE;
12     #Result := #Number1 + #Number2;
13     #OK := ENO;
14 END_IF;

```

Fig. 6.8 Example of representation as Structured Control Language



### 6.3.6 Editing tags

Almost all program elements require tags in order to execute their function. Following insertion in the working area, a program element must be supplied with tags. Fig. 6.9 shows a CTUD up/down counter as a single instance with the instance data block "IEC\_Counter\_DB" in the various programming languages.

LAD representation	FBD representation	SCL representation
<pre> *IEC_Counter_DB*   CTUD   Int   --- CU      QU --- false --- CD  QD --- ... false --- R   CV --- ... false --- LD &lt;??.?&gt; --- PV           </pre>	<pre> *IEC_Counter_DB*   CTUD   Int   &lt;??.?&gt; --- CU false --- CD false --- R   QD --- ... false --- LD  CV --- ... &lt;??.?&gt; --- PV  QU ---           </pre>	<pre> 16 □ "IEC_Counter_DB".CTUD (CU:=_in_, 17   CD:=_in_, 18   R:=_in_, 19   LD:=_in_, 20   PV:=_in_, 21   QU=&gt;_out_, 22   QD=&gt;_out_, 23   CV=&gt;_out_); 24           </pre>

**Fig. 6.9** Supply of box inputs and outputs

LAD and FBD indicate with three red question marks that you must enter a tag here. If three dots are displayed, supplying a tag is optional. With SCL, the missing tags are occupied by dummy values which have to be replaced by "real" tags.

The program editor displays the global tags enclosed by quotation marks. Local tags are preceded by a number sign (#); if they possess special characters, these are additionally enclosed by quotation marks. Operands (absolute addresses) are preceded by a percentage sign (%).

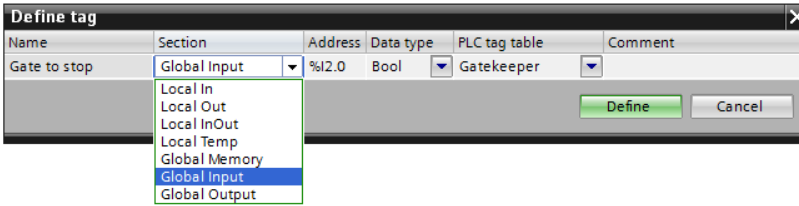
You can display the tags with absolute address, symbol address, or both. The setting is carried out using the *View > Display with > Address information* command from the main menu, or with the *Absolute/symbolic operands* symbol in the toolbar of the program editor.

The program editor provides support for the input of tags: When you enter the first letters of the name of a missing tag, the editor provides a list of (previously defined) tags which can be considered for the current data type. You can then choose the desired tag.

The data type of the tag must correspond to the data type of the supply position. If the *IEC check* attribute is activated in the block, it must be exactly the same data type. If the attribute is deactivated, it is sufficient if the tag has the appropriate data width.

If you enter an operand with the appropriate data width which is not present in the PLC tag table, the editor creates a new "Tag\_x" in the PLC tag table, with x as a consecutive number. By clicking with the right mouse button on a tag and selecting the command *Rename tag* from the shortcut menu, you can assign a different name to the tag. With *Rewire tag* you can assign a different absolute address to the tag.

When programming the control function, you can also enter the name of a tag which does not yet exist. The name of the tag is then underlined in red. By clicking



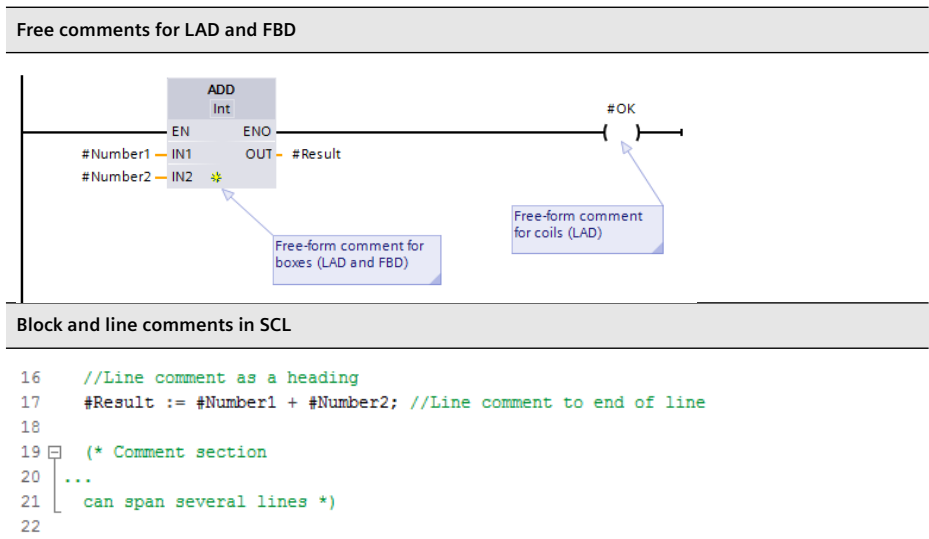
**Fig. 6.10** Defining tags during program input

with the right mouse button on the undefined tag and selecting the command *Define tag* from the shortcut menu, you are provided with a new window in which you can define the tag (Fig. 6.10). You can also select the operand area in which the tag is to be positioned: Input, output or in/output parameter, static or temporary local data, bit memories, inputs or outputs.

### 6.3.7 Working with program comments

With LAD and FBD as the programming languages, you can enter a “free-form comment” for each coil or box (LAD) and for each non-binary box (FBD). Right-click on the program element and select *Insert comment* from the shortcut menu. The program editor displays a comment box with an arrow pointing to the selected program element. You can then enter a comment in the box. You can shift the box within the network or increase its size using the triangle at the bottom right corner (Fig. 6.11).

The programming language SCL provides line and block comments. Line comments are commenced by two slashes and extend up to the end of the line. A line



**Fig. 6.11** Comments on the control function

comment can also stand alone in a line, for example as a heading. A block comment is starts with left parenthesis and asterisk and ends with a asterisk and right parenthesis. Example: (\* *This is a block comment* \*). It can extend over several lines.

In SCL, you can eliminate the comment in a code line by positioning the cursor in the code line and clicking the *Disable code* icon in the toolbar of the working window. Two slashes are then placed at the beginning of the code line. You can reverse the procedure using the *Enable code* icon.

## 6.4 Programming a data block

### 6.4.1 Creating a new data block

It is only possible to create a new data block if a project and a PLC station have first been created. First open the project. You can create a new data block in either the Portal view or the Project view.

In the Portal view, click *PLC programming* and subsequently *Add new block*. In the Project view, double-click on *Add new block* in the *Program blocks* folder. In the window for creating a new block, select the symbol for *Data block*.

Data blocks must be assigned a type:

- ▷ A *global data block* contains the tags which you specify when programming the data block. You can design the contents and structure of the data block as desired.
- ▷ An *instance data block* contains the block parameters and static local data of a function block (FB). The data structure is specified when programming the block interface. For instance data blocks for system functions, the structure is already specified and cannot be changed.
- ▷ A *data block with assigned data type* (“type data block”) contains the tags with the structure of a PLC data type or a system data type. The data structure is defined during programming of the PLC data type or is specified by the system data type.

The *Type* drop-down list shows the blocks and data types which have already been programmed and are thus currently available for use. Select the entry from the list with which you wish to structure the data block to be created. Select the *Global DB* entry for a data block whose content you wish to structure as desired.

Assign a meaningful name to the new block. The name must not already have been assigned to a different block, a PLC tag, a symbolically addressed constant, or a PLC data type. The name can contain letters, digits, and special characters (but not quotation marks). No distinction is made between upper and lower case when checking the name.

The language for data blocks is always DB. With the automatic assignment of the block numbers, the lowest free number for the type of block is displayed in each case; if you select *Manual*, you can enter a different number. You set the *Optimized* for block access option here for a global data block. In the case of an instance data

block, it is imported from the assigned function block and the attribute *Optimized block access* is fixed for a data block derived from a data type. *Optimized block access* has effects on the addressing and retentivity of the tags used in the data block. Activation of the option is recommended.

If the *Add new and open* checkbox is activated, clicking *OK* starts the program editor and the data block is opened.

#### 6.4.2 Working area of program editor for data blocks

The program editor is automatically started when a data block is opened. Open a block by double-clicking on its icon: This can be found in the Portal view in the overview window of the PLC programming, or in the Project view in the *Program blocks* folder under the PLC station in the project tree. The program editor's working window shows the following details for a data block (Fig. 6.12):

- ▷ The toolbar contains the icons for *Insert row*, *Add row*, *Reset start values*, *Updates the interface*, *Snapshot of the monitored values*, *Expanded mode*, and *Monitor all*. The meaning of the symbols is displayed if you hold the mouse pointer over the symbol. Currently non-selectable icons are grayed out.
- ▷ The tag declaration shows the contents of the data block.

The working area can be maximized by clicking on the *Maximize* symbol in the title bar, and embedded again using the symbol for *Embed*. Display as a separate window is also possible: click in the title bar on the symbol for *Float*.

	Name	Data type	Start value	Retain	Comment
1	Static				
2	DataRecord	Array [1 .. 4] of Int		<input type="checkbox"/>	Data for deliveries
3	Results	Array [1 .. 4] of Int		<input checked="" type="checkbox"/>	Total no. of units delivered
4	Results[1]	Int	0	<input checked="" type="checkbox"/>	
5	Results[2]	Int	0	<input checked="" type="checkbox"/>	
6	Results[3]	Int	0	<input checked="" type="checkbox"/>	
7	Results[4]	Int	0	<input checked="" type="checkbox"/>	
8	Quantity_min	USInt	0	<input checked="" type="checkbox"/>	Smallest no. of units delivered
9	Quantity_max	USInt	0	<input checked="" type="checkbox"/>	Largest no. of units delivered
10	Main_gate_Number	USInt	0	<input type="checkbox"/>	No. of times gate opened
11	Main_gate_Time	Time	T#5m	<input type="checkbox"/>	Monitoring time gate opened
12	Duration_time	Time	T#0ms	<input type="checkbox"/>	Duration of unloading
13	Duration_daily	Time	T#0ms	<input type="checkbox"/>	Daily average duration
14	Duration_monthly	Time	T#0ms	<input type="checkbox"/>	Monthly average duration
15	Fault_light_barrier	USInt	0	<input type="checkbox"/>	No. of faults light barrier
16	Fault_Loops	USInt	0	<input type="checkbox"/>	No. of faults induction loops
17	GateControl	UInt	0	<input type="checkbox"/>	Discrete alarms for monitoring states
18	<Add new>			<input type="checkbox"/>	

Fig. 6.12 Example of the program editor's working window for data blocks

You can save the structure of the windows and tables using the *Save windows settings* icon in the top right corner of the working window. This structure is reestablished the next time the working window is opened.

### 6.4.3 Defining properties for data blocks

To set the block properties, select the block in the *Program blocks* folder, followed by the *Edit > Properties* command in the main menu or the *Properties* command in the shortcut menu. The block properties that can be changed are described in detail in Chapter 5.3.2 “Editing block properties” on page 128. A data block can be protected against illegal access (“know-how protection”) or illegal use (“copy protection”). You can only set the *Optimized block access* attribute for a global data block when creating a block; afterwards, it can no longer be changed.

### 6.4.4 Declaring data tags

The declaration table shows the following columns depending on the block properties and the editing environment:

- ▷ Name: The name can contain letters, digits, and special characters (but not quotation marks). No distinction is made between upper and lower case when checking the name. The name is block-local, and therefore the name can also be used in other blocks for different tags. In association with the data block whose name applies throughout the CPU (“globally”), a data tag becomes a “global” tag applicable throughout the CPU.
- ▷ Data type: Select the data type of the tag from a drop-down list, or enter it directly.
- ▷ Offset: The offset indicates the relative address of the tag with respect to the start of the data block or the start of a data structure. The column is only shown if the *Optimized block access* attribute is not activated in the data block. The offset is shown after the data block has been compiled.
- ▷ Default value: The default value is the value which is automatically assigned to a new tag depending on the data type. Example: With the data type DATE, the default value is DATE#1990-01-01. If the data block is based on a data type (type data block) or a function block (instance data block), the tag value defined in the data type or in the function block is present in the *Default value* column.
- ▷ Start value: The *Start value* column lists the individual default values of the tags for this data block. The default value is used if a start value is not entered. The start value is the value with which the data block is loaded into the CPU's work memory. With an instance data block, it is then possible to commence each call (each instance) with different start values.
- ▷ Snapshot: The *Snapshot* column shows the “frozen” monitoring values from the work memory at the time of the snapshot.

- ▷ Monitor value: The monitoring value indicates the actual values of the tags in online mode. This is the value present when the work memory is scanned. This column is only displayed in *Monitoring* mode.
- ▷ Retain: A tick in this column indicates that the tag is retentive. If the *Optimized block access* attribute is activated for the global data block, individual tags can be set as retentive, otherwise only the complete data block. For an instance data block, configure the retentivity of the individual tags in the assigned function block. For a type data block, only the complete data block can be set to retentive or non-retentive.
- ▷ Visible in HMI: A tick in this column means that the tag is visible in the selection list of HMI stations by default.
- ▷ Accessible from HMI: A check in this column indicates that an HMI station can access this tag.
- ▷ Comment: The comment allows input of an explanation of the purpose of the tag.

You can determine the columns to be displayed yourself: Right-click in the line with the column headers and then select the *Show/hide columns > ...* command from the shortcut menu. You can then select or deselect the columns to be displayed.

### Expanded mode

The expanded mode is activated using the *Expanded mode* icon in the toolbar of the working window. All tags with structured data types such as DTL, ARRAY, STRUCT, PLC data types, and system data types are then “opened” (expanded) so that the individual components can be shown and – if allowed – assigned values (Fig. 6.13).

Data110						
	Name	Data type	Offset	Start value	Retain	Comment
1	Static					
2	Ramp_start	Bool	0.0	false	<input type="checkbox"/>	Switch on ramp
3	Ramp_top	Bool	0.1	false	<input type="checkbox"/>	Limit switch upper
4	Ramp_bottom	Bool	0.2	false	<input type="checkbox"/>	Limit switch lower
5	Delivery	Struct	2.0		<input type="checkbox"/>	Current delivery
6	Code	Word	0.0	0	<input type="checkbox"/>	Delivery object
7	Quantity	UInt	2.0	0	<input type="checkbox"/>	No. of objects
8	Checked	Bool	4.0	false	<input type="checkbox"/>	Check ok
9	Quantity_ramp	Array [1 .. 4] of UInt	8.0		<input type="checkbox"/>	Delivered quantity
10	Quantity_ramp[1]	UInt		0	<input type="checkbox"/>	
11	Quantity_ramp[2]	UInt		0	<input type="checkbox"/>	
12	Quantity_ramp[3]	UInt		0	<input type="checkbox"/>	
13	Quantity_ramp[4]	UInt		0	<input type="checkbox"/>	
14	Shift_start	DTL	16.0	DTL#1970-1-1-0-0	<input type="checkbox"/>	start of shift
15	YEAR	UInt	0.0	1970	<input type="checkbox"/>	
16	MONTH	USInt	2.0	1	<input type="checkbox"/>	
17	DAY	USInt	3.0	1	<input type="checkbox"/>	
18	WEEKDAY	USInt	4.0	5	<input type="checkbox"/>	
19	HOUR	USInt	5.0	0	<input type="checkbox"/>	

Fig. 6.13 Example of structured data types in expanded mode

### 6.4.5 Entering data tags in global data blocks

With a global data block, you enter the data tags directly in the block. In the *Name* column you specify the name of the tag. Following input of the name, select the data type from a drop-down list, enter a start value if applicable, and use a comment to explain the purpose of the tag.

With the `STRING` data type, enter the maximum length of the string in square brackets. If this data is missing, the standard length of 254 characters is used.

With the `ARRAY` data type, you must enter the range limits and the component data type. For example, the information in the drop-down list *Array [lo .. hi] of type* could then result in *Array [1 .. 12] of Real*. If you click on the triangle to the left of the tag name, the components are displayed, and you can assign individual start values to them as default values.

Select the `STRUCT` data type from the drop-down list and, in the line under the tag name, enter the name of the first component, its data type, possibly a default setting, and a comment. The next line contains the second component, etc.

When programming a tag with PLC data type, select the PLC data type from the drop-down list. If you click on the triangle to the left of the tag name, the components are displayed, and you can assign individual start values to them as default values.

When programming a tag with a system data type whose structure is defined by STEP 7, such as *ErrorStruct* or *IEC\_TIMER*, you cannot change the structure and can only carry out a default setting for individual components where permissible.

## 6.5 Compiling blocks

Compilation generates a program code which can execute in the CPU. A compilation process is always triggered prior to downloading the user program to the PLC station. Only blocks which have been compiled without errors can be downloaded.

It is recommendable to also trigger compilation while generating the user program to enable a quick response to any programming errors.

### 6.5.1 Starting the compilation

You start the compilation using a command from the shortcut menu.

- ▷ To compile a block opened in the program editor, click with the right mouse button on the white background of the working area and select the *Compile* command.
- ▷ To compile a block listed in the call structure or in the dependency structure, click with the right mouse button on the block and select the *Compile* command.
- ▷ To start the compilation process for the selected block, click with the right mouse button on the block in the *Program blocks* folder in the project tree followed by the *Compile > Software* command.

- ▷ You can also select several blocks in the *Program blocks* folder and compile them together using the *Compile > Software* command from the shortcut menu.
- ▷ You can compile the entire user program by selecting the *Program blocks* folder followed by *Compile > ...* from the shortcut menu. You then have the choice between ... *Software* (compile program changes since last compilation only) and ... *Software (rebuild all blocks)* (compilation of entire program).
- ▷ If you select the *PLC station* folder and then *Compile > ...* from the shortcut menu, you can select between
  - ... *All* (complete compilation of all project information relevant to execution)
  - ... *Hardware configuration* (compilation of device and network configuration)
  - ... *Software* (compilation of program changes since last compilation only) and
  - ... *Software (rebuild all blocks)* (compilation of entire user program)

The result of the compilation is displayed in the inspector window in the *Info* tab under *Compile* (Fig. 6.14). Any warnings which have been detected do not prevent continuation of the compilation. Any errors which have been detected are displayed in the result of the compilation, and end the compilation.

Path	Description	Errors	Warnings
Motor 3 (DB33)	Block was successfully compiled.	0	0
Motor 2 (DB32)	Block was successfully compiled.	0	0
TRCV_C_DB (DB52)	Block was successfully compiled.	0	0
DatenEmpfangen (FB51)	Block was successfully compiled.	0	0
DataReceive_DB (DB1)	Block was successfully compiled.	0	0
▶ Conveyor belt (FB20)		1	0
Conveyor data (DB20)	Block was successfully compiled.	0	0
Main_Comm (OB201)	Block was successfully compiled.	0	0
Main (OB1)	Block was successfully compiled.	0	0
▼ Conveyor belt (FB20)		1	0
Network 5	Tag ack not defined.	?	1
Compiling completed (errors: 2; warnings: 2)		1	0

Fig. 6.14 Example of compilation information in the inspector window

### 6.5.2 Compiling SCL blocks

You can set the attributes for compilation in the properties of SCL blocks. The setting for all newly created blocks is specified in the main menu under *Options > Settings* and *PLC programming > SCL > Compile*.

The activatable attributes are:

- ▷ Create extended status information  
permits monitoring of all tags in a block.



- ▷ Check ARRAY limits  
checks the limits of ARRAY tags during runtime and sets ENO to FALSE in the event of a limit violation.
- ▷ Set ENO automatically  
checks whether errors have occurred in program execution during runtime and sets ENO to FALSE in the event of an error.

Activation of one of the attributes increases the memory requirements and processing time of the block.

### 6.5.3 Eliminating errors following compilation

An error is indicated by a white cross on a red circle in the line of the faulty block. Click on the triangle to the left of the block name to open the list with the compilation messages.

Click on the blue question mark in an error message to display more information about the error. Double-clicking on an error message displays the program environment of the selected error in the working window so that you can correct the error directly.

#### Correcting a faulty block call

During the compilation, the program editor checks whether the supply of block parameters present in the calling block agrees with the interface of the called block. An error is not signaled if the program editor can correct the block call when compiling.

If you double-click on the error message, the program editor opens the point in the program with the faulty call. You can then correct the call, for example by entering missing actual parameters or by using actual parameters with the correct data type. If the block call is displayed with a red border, select the *Update* command from the shortcut menu. The program editor suggests a modified block call in the *Interface update* window which you can import unchanged or following modification (Fig. 6.15).

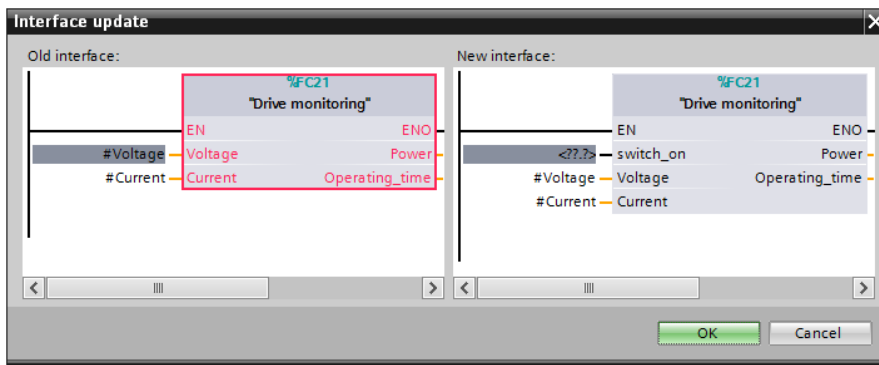


Fig. 6.15 Interface update in the case of faulty block calls

Not all block parameters have to be supplied for a function block call. If such a block parameter is added subsequently, the compilation does not signal an error – the supply simply remains open. Even if the assignment of the static local data is changed, a new instance data block is generated during the compilation, and an error message is not produced. If the instance data block for compilation is missing, a new data block with the next free number is simply generated, also without an error message.

Under *Options > Settings* and *PLC programming > General > Compilation*, you can select the option *Delete actual parameters on interface update*. The result is that an actual parameter is deleted when compiling or updating the interface if the associated block parameter has been deleted. An error message is then not output.

How you can nevertheless find the call of a block with a subsequently modified interface in the program is shown in Section 6.6.5 “Consistency check” on page 206.

## 6.6 Program information

The following program information supports you during programming and program debugging:

- ▷ Cross-references
- ▷ Assignment list (inputs, outputs and bit memories)
- ▷ Call and dependency structures
- ▷ Resources

You can start the individual tools at any time during program creation, either in the main menu using the *Tools > ...* command or in the project tree by double-clicking *Program info* under a PLC station. Following commissioning, the program information can be part of the project documentation.

### 6.6.1 Cross-reference list

The cross-reference list indicates the use of tags, hardware constants, and blocks in the user program. It provides an overview of

- ▷ Which objects have been used
- ▷ At which position in the program they have been used, and
- ▷ the context in which they were used, e.g. whether there is read or write access to a tag .

You can create cross-references from any data object of a station: select the station, a folder under the station, or one or more objects in a folder, e.g. one or more blocks or PLC tags, and then select the *Cross-reference list* command from the short-cut menu or the *Tools > Cross-references* command from the main menu. The cross-reference list is available in two views: *Used by* and *Uses*.

### Cross-reference list *Used by*

The *Used by* view is based on the referenced object. It shows the positions at which the object present in the first column is used (Fig. 6.16). For example, all the positions of where a block is called are shown, or all the program positions at which a tag is used. If the list entries are opened, a link in the *Point of use* column leads directly to the program position at which the object is used.

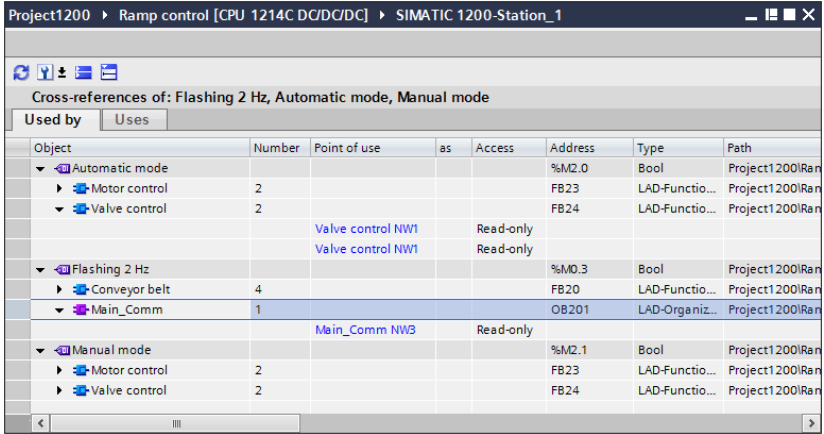


Fig. 6.16 Example of a cross-reference list in the *Used by* view

You can select the view options using the spanner icon in the toolbar of the cross-reference list: *Show used* and/or *Show unused*.

### Cross-reference list *Uses*

The *Uses* view displays the objects used by the referenced object. It shows which objects are used (Fig. 6.17). With a block, for example, it shows which blocks are

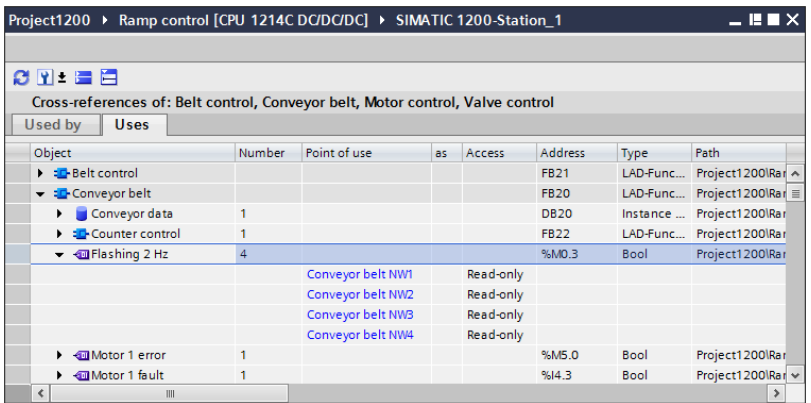


Fig. 6.17 Example of a cross-reference list in the *Uses* view

called within it and which tags are used within it. If the list entries are opened, a link in the *Location* column leads directly to the program position at which the associated object is used.

You can select the view options using the spanner icon in the toolbar of the cross-reference list: *Show defined* and/or *Show not undefined*.

### Display of cross-references in the inspector window

Select an object, e.g. a block in the project tree or a tag in the working window, and then select the *Cross-reference information* command in the shortcut menu. The inspector window – under *Cross-references* in the *Info* tab – shows the program positions at which the selected object has been used. If the cross-reference list is open in the inspector window, the use of the selected object is displayed directly.

### 6.6.2 Assignment list

The assignment list shows the assignment of the operand areas: inputs (I), outputs (Q) and bit memories (M). Peripheral inputs (I:P) which are used are shown under the input address, peripheral outputs (Q:P) which are used are shown under the output address. The use of operands as bit, byte, word or doubleword operands or tags is displayed.

You can display the assignment list for individual blocks or for the entire program: Select the blocks, the *Program blocks* folder or the folder of the PLC station, and then select *Assignment list* from the shortcut menu or *Tools > Assignment list* in the main menu (Fig. 6.18).

Input, Output											Memory													
Address	7	6	5	4	3	2	1	0	B	W	DW	Address	7	6	5	4	3	2	1	0	B	W	DW	
IB0												MB0												
IB1												MB1												
IB4				♦								MB2												
IB5				♦								MB4												
IB64				♦								MB5												
IB65				♦								MB6												
IB66				♦								MB10												
IB67				♦								MB11												
IB112				♦								MB12												
IB113				♦								MB13												
IB114				♦								MB16												
IB115				♦								MB17												
IB116				♦								MB18												
IB117				♦								MB19												
IB118				♦								MB21												
IB119				♦								MB23												

Fig. 6.18 Example of an assignment list

### Display of input/output assignment

A yellow background for inputs and outputs indicates that the address is not used by the hardware or that no hardware has been configured for this address. If you additionally address a bit in a byte, word or double word operand, the entry has a gray background. You can use the *View options* symbol in the toolbar of the assignment list to select whether the used addresses and/or the free hardware addresses are to be displayed.

### Display of bit memory assignment

The view option must be set to *Used addresses* in order to display the bit memory assignment. With the bit memories, symbols are used for the operands to indicate those which are set as clock and system memories, and up to which address the bit memories are retentive. You can switch the retentivity display on and off using the corresponding symbol in the toolbar. You can set the retentive range for the bit memories using the *Retain* symbol.

### Filter

You can filter the display of the assignment list using the *Filter* symbol in the toolbar. You specify which addresses (operands) you wish to view: to select the operand area, activate the associated check box. You can select all addresses as the filter area (with an asterisk: \*), a section of the address area using a hyphen (e.g. 0-100), an individual address (e.g. 101), or several areas separated by a semicolon (e.g. 0-100; 120-124; 160).

If you wish to repeatedly use the particular settings of a filter, assign a name to it in the drop-down list of the filter dialog. You can then use this name to recall the filter settings from the drop-down list in the toolbar of the assignment list. You can also delete filter names again.

### 6.6.3 Call structure

The call structure describes the call hierarchy of the blocks. To display the call structure, select the *PLC station* or *Program blocks* folder for the entire program or for individual blocks and then select *Call structure* from the shortcut menu or *Tools > Call structure* from the main menu.

The call structure shows the used blocks and the logic blocks called from these blocks or the data blocks used in them (Fig. 6.19). The blocks that are not called in the user program are present in the first level (color highlighted) – in the finished program, these should only be the organization blocks.

Starting with the call structure, you can display the cross-reference information or open a block for processing with the program editor. The consistency test for the block calls is described in Chapter 6.6.5 “Consistency check” on page 206.

You can set the view options using the *View options* icon in the toolbar: *Show conflicts only* then displays the call paths in which conflicts have been detected, e.g.

Call structure	Address	Call fre...	Details	Local data (in path)	Local d
1 Main	OB1			0	0
2 Belt control, Belt Data	FB21, DB21	1	Main NW1	0	0
3 Conveyor belt, Conveyor data	FB20, DB20	1	Main NW1	4	4
4 Counter control, Counter d...	FB22, DB45	1	Conveyor belt NWS	23	19
5 Motor 1	DB31	2		4	0
6 Motor 2	DB32	2		4	0
7 Motor 3	DB33	2		4	0
8 Motor 4	DB34	2		4	0
9 Motor control, Motor 1	FB23, DB31	1	Conveyor belt NW1	24	20
10 Drive monitoring	FC21	1	Motor control NW3	30	6
11 Motor control, Motor 2	FB23, DB32	1	Conveyor belt NW2	24	20
12 Motor control, Motor 3	FB23, DB33	1	Conveyor belt NW3	24	20
13 Motor control, Motor 4	FB23, DB34	1	Conveyor belt NW4	24	20
14 Valve control, Valve data	FB24, DB41	1	Conveyor belt NWS	5	1
15 Time error interrupt	OB80			0	0
16 Hardware interrupt	OB200			0	0
17 Main_Comm	OB201			4	4

Fig. 6.19 Example of call structure

interface conflicts, recursive calls, or non-existent block calls. *Group multiple calls together* displays several calls of a block or data block access operations in a single line, and specifies the number of calls in a separate column.

For compiled blocks, the memory requirements for temporary local data of a block and in the path are displayed.

#### 6.6.4 Dependency structure

The dependency structure shows the dependencies of each block. To display the dependency structure, first select the *PLC station* or *Program blocks* folder for the entire program or for individual blocks, and then select *Tools > Dependency structure* from the main menu.

For each logic block the dependency structure shows the block from which it is called, and for each data block the logic block in which it is used (Fig. 6.20).

From the dependency structure, you can display the cross-reference information or open a block for processing with the program editor. The consistency test for the block calls is described in the next Chapter 6.6.5 “Consistency check”.

You can set the view options using the *View options* icon in the toolbar: *Show conflicts only* then displays the call paths in which conflicts have been detected, e.g. interface conflicts, recursive calls, or non-existent block calls. *Group multiple calls together* displays several calls of a block or data block access operations in a single line, and specifies the number of calls in a separate column.

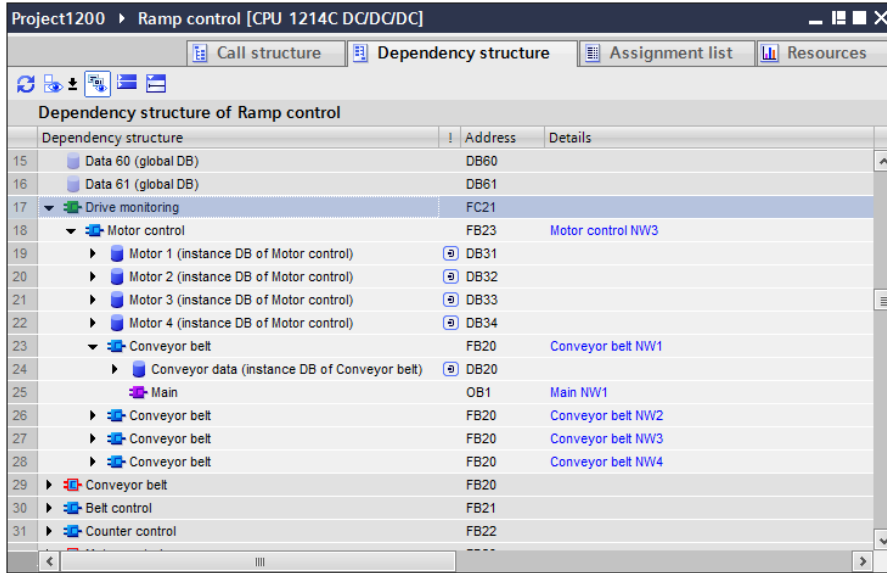


Fig. 6.20 Example of dependency structure

### 6.6.5 Consistency check

Clicking on the *Consistency check* icon in the toolbar of the call structure or dependency structure displays block calls with an “interface conflict”. These are block calls whose interface is subsequently changed, such as by assigning another data type to a block parameter or by adding a block parameter.

Blocks which have not yet been compiled following a modification are displayed with a red border. In order to compile individual blocks in the call or dependency structure, select *Compile* in the shortcut menu.

If interface conflicts cannot be eliminated by a repeated compilation, they must be handled manually. The link in the *Details* column leads to the faulty block call.

Open the calling block, select the block call identified as faulty, and select the *Update* command from the shortcut menu. When updating the block call, the program editor shows what the updated block call will look like in the *Interface update* window (for an example, see Fig. 6.15 on page 200). You can then carry out corrections and supplements in this window, for example if a new block parameter has been added.

### 6.6.6 CPU resources

Resources shows the assignment of the user memory and of the existing input/output modules. To display resources, first select the *PLC station*, *Program blocks* folder or individual blocks, and then select the *Resources* command from the shortcut menu or *Tools > Resources* from the main menu.

Objects	Load memory	Work memory	Retentive memory	I/O	DI	DO
1	7.23%	2.07%	1.52%		7.21%	12.16%
2						
3	Total:	1 MB	76800 bytes	10240 bytes	Configured:	222 74
4	Used:	75826 bytes	1593 bytes	156 bytes	Used:	16 9
5	Details					
6	▶ OB	11029 bytes	579 bytes			
7	▶ FC	2358 bytes	31 bytes			
8	▶ FB	30015 bytes	569 bytes			
9	▶ DB	32424 bytes	414 bytes	28 bytes		
10	Data types	-	-			
11	PLC tags			128 bytes		

**Fig. 6.21** Example of Resources

The resources function shows in three columns the maximum available and actual storage space being utilized by the load memory, work memory and retentive memory (Fig. 6.21). You can see the utilization for each type of block, for individual blocks and for the PLC tags. Question marks represent blocks which have not yet been compiled. The values are then displayed in red in the total lines.

You can set the maximum size of the load memory: Click on the displayed memory size in the *Total* box in the *Load memory* column, open the drop-down list, and set the corresponding value if you wish, for example, to expand the load memory with a memory card.

The existing (configured) input/output modules are divided according to DI, DO, AI and AO, together with information on how many of them are used in the program. Starting with the resources function, you can display the properties of a marked block in the inspector window or open a block for processing with the program editor.

## 6.7 Language setting

STEP 7 offers several options for working with different languages:

- ▷ The language of the operating system (character set)
- ▷ The language of the user interface of the TIA Portal
- ▷ The language of the mnemonic for the operations and operands
- ▷ The editing language and the project languages for the user text
- ▷ The language of the HMI device

Language settings can be made independent of each other.



## Language settings in the operating system

If you are working with a multilingual version of the operating system (MUI version), set the desired character set using the Windows Control Panel.

### The language of the user interface and mnemonic

STEP 7 Basic is operated with the language of the user interface. This comprises, for example, the menu names and the error messages of the TIA Portal. You can set this language in the Project view in the main menu using *Options > Settings* in the *General* section. The languages installed with STEP 7 are offered for selection under *User interface language*. You also set the programming mnemonics in this tab, i.e. the language in which the program editor uses the operands and operations. For example, with the *German* setting, “E” stands for “Eingang”, and with the *International* setting, “I” stands for “Input”.

### Editing language

The user texts are entered in the editing language. These are, for example, comments on PLC tags or the program. The editing language is independent of the language of the user interface. You select the editing language in the project tree under *Languages & Resources > Project languages* from the *Editing language* drop-down list.

### Project languages

The text entered in the editing language can be translated into various project languages and displayed. You specify the available project languages in the project tree under *Languages & Resources > Project languages*. All entered user texts can be found in the project tree under *Languages & Resources > Project texts* in the *User texts* tab. The entered texts in of the editing language and the selected project languages are shown. You can enter text directly or edit it here. The displayed texts are oriented on a reference language that you specify under *Languages & Resources > Project languages* in the *Reference languages* drop-down list. You can also export the texts for translation and reimport the translated texts.

To display the translated user texts in configuration and programming, select the desired project language as editing language.

### Language of the HMI station (HMI project language)

The HMI station can be provided with a multilingual user interface. You set the languages available at runtime in the project tree under the HMI station and *Runtime settings*. The project languages set under *Languages & Resources > Project languages* can be selected.

If during runtime you wish to switch over to another language available on the HMI station, an operator-accessible object, e.g. a button, must have been linked to the language switchover during configuration. Following selection, the new language is applied immediately. When the HMI station is switched on, the language that was active last is always set.

# 7 Ladder logic LAD

## 7.1 Introduction

This chapter describes programming with ladder logic (LAD); it uses examples to show how the program functions are represented in LAD. You can find a description of the individual functions, e.g. comparison functions, in Chapter 10 “Basic functions” on page 328.

Use of the program and symbol editor, which generally applies to all programming languages, is described in Chapter 6 “Program editor” on page 178.

LAD is used to program the contents of blocks (the user program). What blocks are, and how they are created, is described in Chapters 5.3 “Programming blocks” on page 125 and 6.3 “Programming a code block” on page 183.

### 7.1.1 Programming with LAD in general

You use LAD to program the control function of the programmable controller – the user program or control program. The user program is organized in different types of blocks. A block is divided into sections referred to as “networks”. Each network contains at least one current path that may also have an extremely complex structure. Each network is terminated by at least one coil or box.

Fig. 7.1 shows the structure of a block with the LAD program. Located at the beginning of the program is the block title, comprising the block heading and block comment. Heading and comment are optional. These are followed by the first network with its number, heading, and comment. Heading and comment are also optional for the networks. The first network shows a current path as an example with series and parallel connection of contacts, a memory function within the current path, and two coils as termination of the current path. The second network shows the processing of boxes, which can be arranged in series or parallel. A block is not terminated by a special network or function, you simply finish the program input.

The LAD editor establishes a network in accordance with the principle of the “main current path”: this is the highest branch which commences directly on the left-hand power rail and must be terminated by a coil or box. All LAD elements can be positioned within it.

An LAD element must not be “short-circuited” by an “empty” parallel branch, and “current” must not flow from right to left through a program element. A parallel branch which does not end “open” must be closed for the branch on which it was opened.

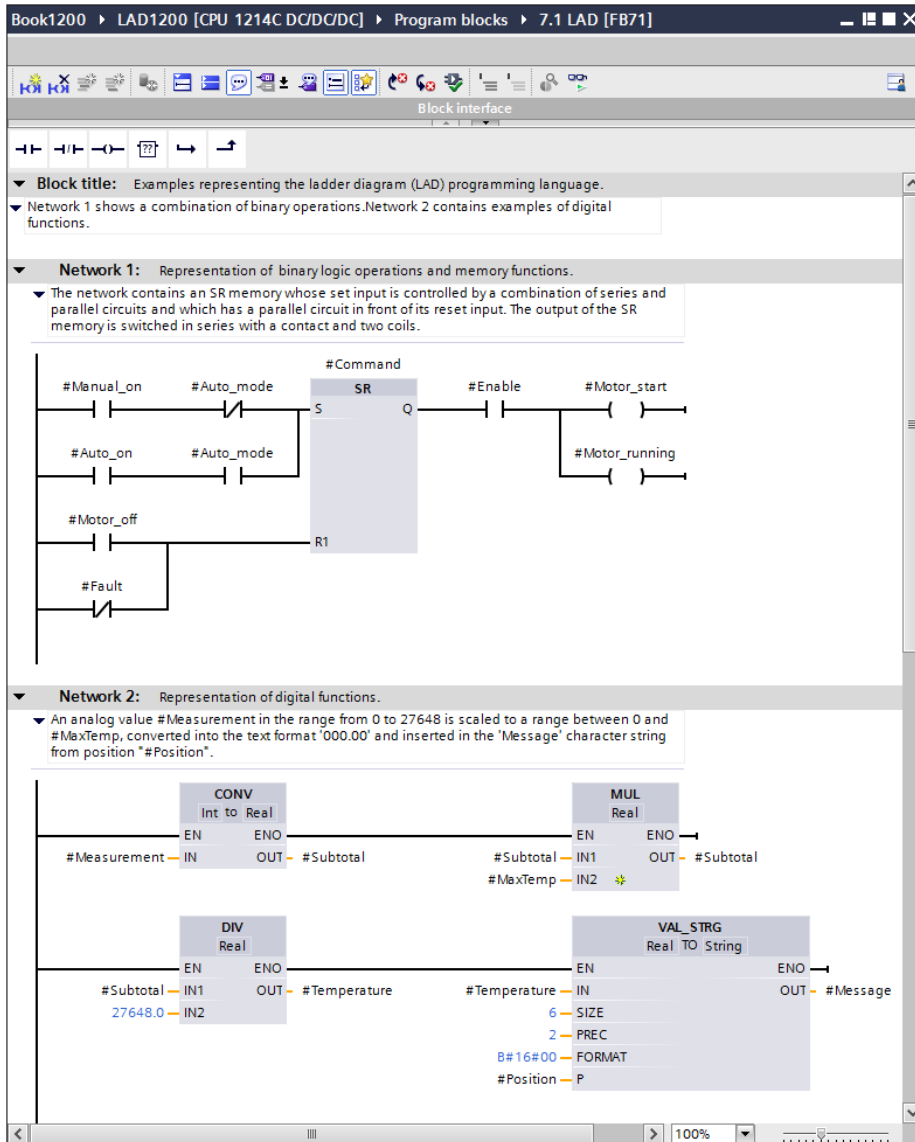


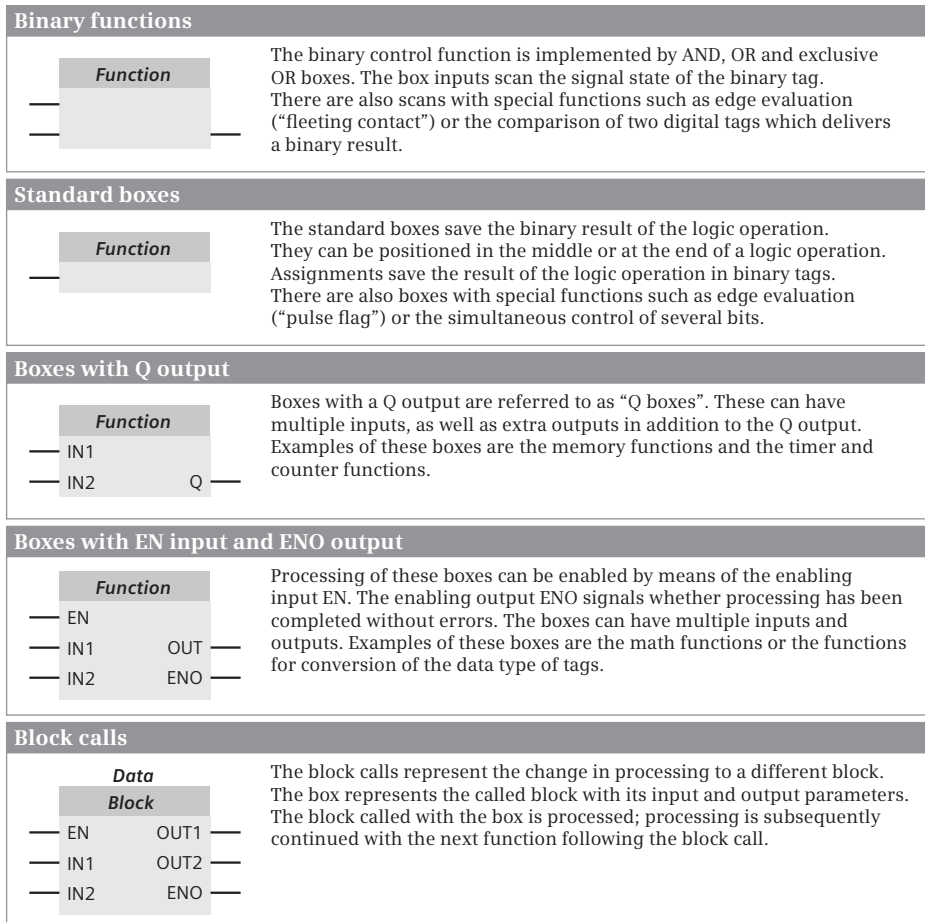
Fig. 7.1 Structure of a block with LAD program

“Open” parallel branches can lead out from the main current path and need not lead back to the main current path; these are known as “T branches”. There are certain limitations in the selection of the permissible program elements in the case of these parallel branches which do not commence on the left-hand power rail.

Where additional rules apply to the arrangement of special LAD elements, these are described in the corresponding sections.

### 7.1.2 Program elements of ladder logic

Fig. 7.2 shows which types of LAD elements exist: Contacts and coils for processing binary signals, Q boxes for implementing memory, timer, and counter functions, and EN/ENO boxes for “complex” functions which, for example, carry out calculations, manipulate strings, or convert numbers into text.






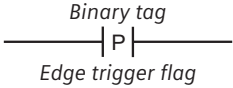
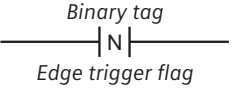
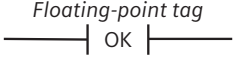
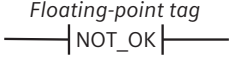
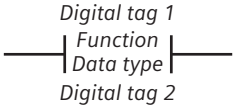
**Fig. 7.2** Overview of ladder logic program elements

Most program elements must be provided with tags or operand addresses. With contacts and coils, the tags are assigned by means of the program element. If further tags are required for the function, these are present under the element. In the case of the boxes, the tags are present at the box inputs and outputs.

It is best if you initially arrange all program elements in a current path and subsequently label them.

## 7.2 Programming with contacts

In the case of contacts you scan the binary tags, e. g. inputs, and link the scanned signal states by arranging the contacts in series or parallel. You use an NO or NC contact to define the influence of the scanned signal state on the logic operation. Further functions for contacts are negation of the signal flow, edge evaluation for a binary tag, validity checking of floating-point numbers, and the comparison function (Fig. 7.3).

Contacts			
Normally closed (NC) contact		Normally open (NO) contact	
NOT contact			
Positive edge of a binary tag		Negative edge of a binary tag	
Scan for "valid"		Scan for "invalid"	
Comparison function			

**Fig. 7.3** Overview of contacts available with LAD

### 7.2.1 NO and NC contacts

An NO or NC contact is used to scan the signal state of a binary tag. An NO contact passes on the scanned signal state directly to the logic operation, an NC contact first negates the signal state.

#### Normally open contact

If the signal state of a binary tag is scanned using an NO contact, the scanned state directly influences the signal flow in the logic operation.

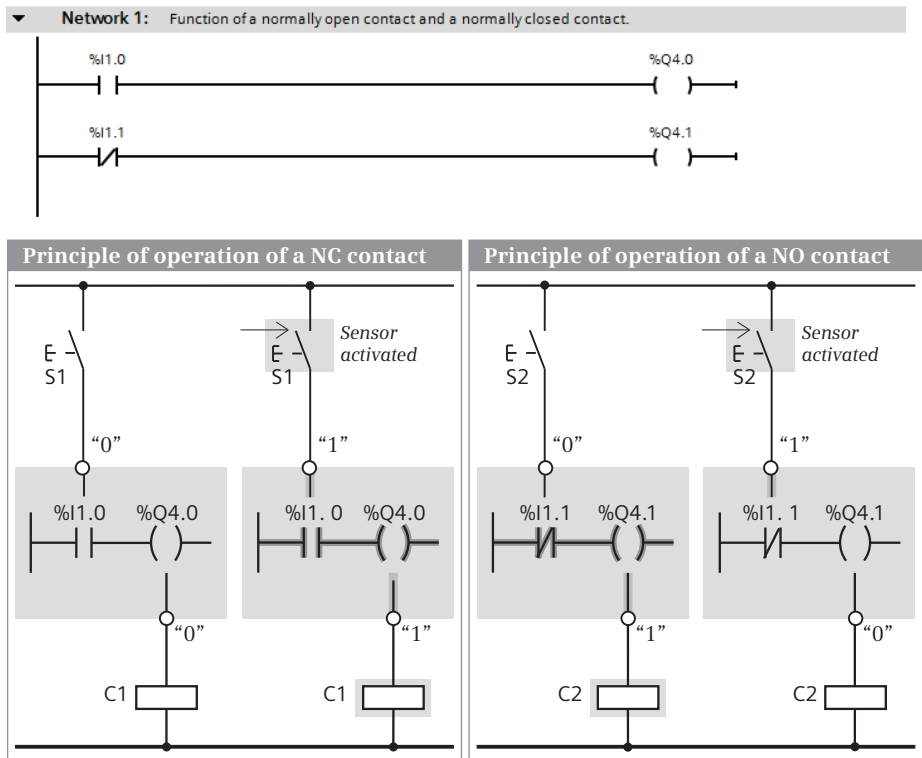
The example in Fig. 7.4 shows the sensor S1 on the left-hand side which is connected to input %I1.0 and is scanned by means of an NO contact. If the sensor S1 is open, the input %I1.0 has the signal state "0" and no current flows through the NO contact. Contactor K1 controlled by output %Q4.0 does not pull up.

If sensor S1 is then activated, input %I1.0 has the signal state "1". Current flows from the left-hand power rail through the NO contact into the coil, and contactor K1 connected to output %Q4.0 pulls up.

### Normally closed contact

If the signal state of a binary tag is scanned using an NC contact, the contact passes on the negated result of the scan to the signal flow in the logic operation.

In the example in Fig. 7.4 (right-hand side), current flows through the NC contact if the scanned sensor S2 is not closed (input %I1.1 has the signal state “0”). The current also flows into the coil and triggers the contactor K2 at output %Q4.1.



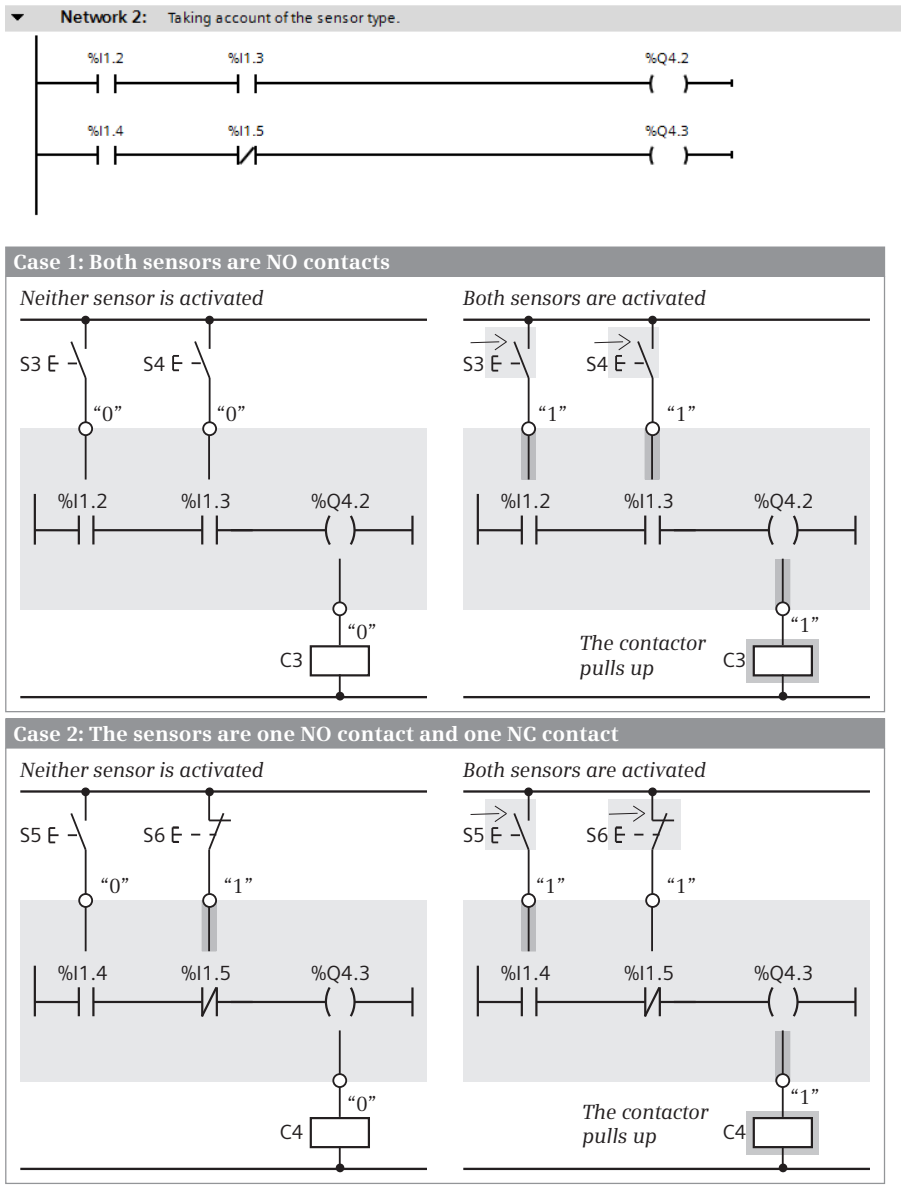
**Fig. 7.4** Principle of operation of NO and NC contacts

If sensor S2 is then activated, “1” is present at the input %I1.1 and the NC contact opens. The current flow is interrupted, and contactor K2 drops out.

#### 7.2.2 Consideration of sensor type in ladder logic

If you scan a sensor in your program, you must take into consideration whether it is an NO or NC contact. Depending on the type of sensor, different signal states are present at the corresponding input when the sensor is activated: “1” with an NO contact and “0” with an NC contact. It is not possible for the CPU to determine whether an NC or NO contact is connected to an input. It can only recognize the signal state “1” or “0”.

If you write the program to obtain “1” when a sensor is activated in order to link it further, you must scan the input in different ways depending on the type of sensor. The contact types *NO contact* and *NC contact* are available for this purpose. An NO contact delivers “1” if the scanned input is also “1”. An NC contact delivers “1” if the scanned input is “0”. In this manner you can also directly scan inputs which are to execute activities when the signal state is “0” (“zero-active”) and connect the result of the scan further.



**Fig. 7.5** Consideration of the type of sensor

The example in Fig. 7.5 on page 214 shows the programming dependent on the type of sensor. In the first case, two NO contacts are connected to the programmable controller, in the second case one NO contact and one NC contact. In both cases, a contactor connected to an output is to pull up when both sensors are activated. When an NO contactor is activated, the signal state at the input is “1” and is scanned by an NO contact so that current can flow when the sensor is activated. If both NO contacts are activated, current flows through the current path into the coil, and the contactor pulls up.

When activating an NC contact, the signal state at the input is “0”. To allow current to flow following activation, an NC contact must be used for the scan. Therefore in the second case one NO contact and one NC contact must be connected in series in order for the contactor to pull up when both sensors are activated.

### 7.2.3 Series connection of contacts

With a series connection, two or more contacts are positioned one behind the other. Current flows through a series connection when all contacts are closed.

Fig. 7.6 shows an example of a series connection: If in the upper current path all tags have signal state “1”, all NO contacts are closed and the tag “Coil 1” is set to signal state “1”. In all other cases, “Coil 1” is reset to signal state “0”.

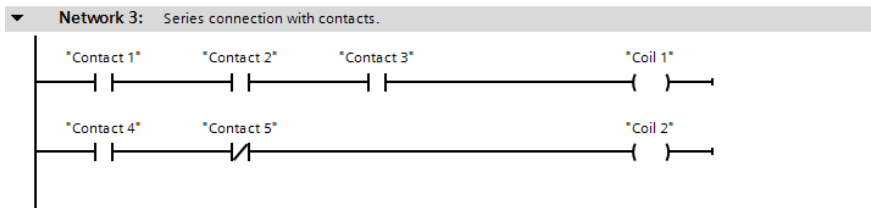


Fig. 7.6 Series connection of contacts

In the current path underneath this, the tag “Coil 2” is set to signal state “1” if “Contact 4” has the signal state “1” and “Contact 5” the signal state “0”.

### 7.2.4 Parallel connection of contacts

A parallel connection means that two or more contacts are positioned underneath each other. Current flows through a parallel connection when one of the contacts is closed.

Fig. 7.7 shows an example of a parallel connection: if none of the tags has the signal state “1” in the top current path, all NO contacts are open and the tag “Coil 3” has the signal state “0”. In all other cases, “Coil 3” is set to signal state “1”.

In the current path underneath this, the tag “Coil 4” has the signal state “0” if the tag “Contact 4” has the signal state “0” and the tag “Contact 5” the signal state “1”.



With LAD you can also program a branch in the middle of a current path, such as the top current path in . The result is a parallel branch which does not commence on the left-hand power rail. A parallel connection with an “open” termination is referred to as a “T branch” (see Chapter 7.2.6 “T branch, open parallel branch in the ladder logic” on page 217). In both cases, use of LAD program elements is limited; reference to this is made in the corresponding sections.

### 7.2.5 Mixed series and parallel connections

You can combine series and parallel connections, e.g. arrange several series connections in parallel or several parallel connections in series. You can also connect series or parallel connections together even if they have a complex structure.

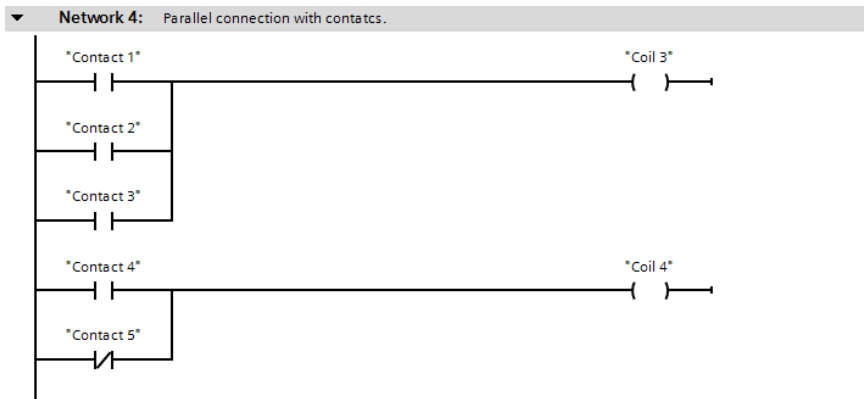


Fig. 7.7 Parallel connection of contacts

### Parallel connection of series connections

Instead of contacts you can also arrange series connections underneath each other. Fig. 7.8 shows two examples. The tag “Coil 5” is set to signal state “1” if “Contact 1”

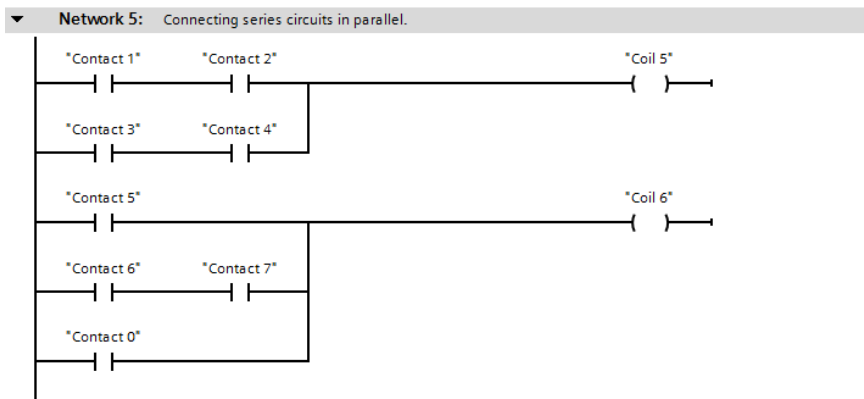


Fig. 7.8 Parallel connection of series connections

and “Contact 2” or if “Contact 3” and “Contact 4” have the signal state “1”. “Coil 6” is set to signal state “1” if “Contact 5” or “Contact 6” and “Contact 7” or “Contact 0” have the signal state “1”.

### Series connection of parallel connections

Instead of contacts you can also arrange parallel connections one behind the other. Fig. 7.9 shows two examples. The tag “Coil 7” has signal state “1” if either “Contact 1” or “Contact 3” and either “Contact 2” or “Contact 4” have the signal state “1”. So that the tag “Coil 0” can have the signal state “1”, “Contact 7” or “Contact 5” and “Contact 6” must have signal state “1” in addition to “Contact 0”.

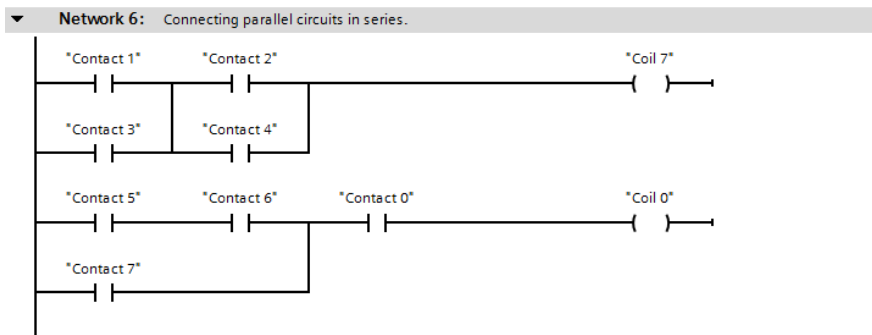


Fig. 7.9 Series connection of parallel connections

### 7.2.6 T branch, open parallel branch in the ladder logic

You can “divide” a current path so that it has two different terminations. If this is not simply a parallel connection of coils or boxes, but a case of both branches hav-

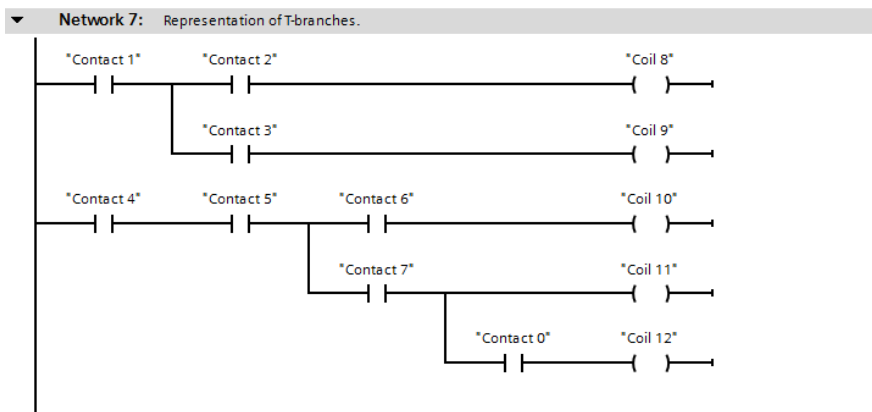


Fig. 7.10 T branch, open parallel branch

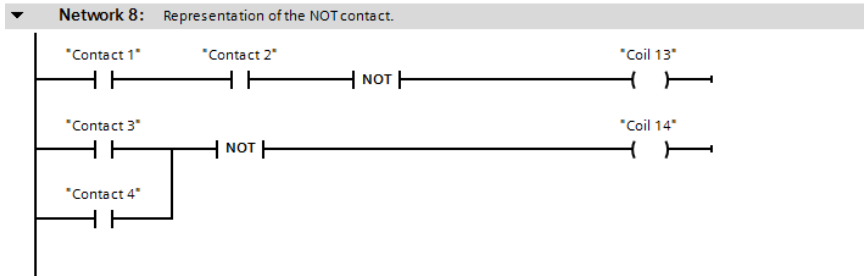
ing different logic operations, this is referred to as a “T branch” or an “open” parallel branch.

Fig. 7.10 shows a simple T branch in the top current path. “Coil 8” can be switched on by the tag “Contact 2”, and “Coil 9” by the tag “Contact 3”. A prerequisite in both cases is that the tag “Contact 1” has the signal state “1”.

Series and parallel contact connections can be programmed following a T branch. A further T branch can also be opened within a T branch. However, you cannot enter logic operations which lead from the left-hand power rail to a T branch.

### 7.2.7 Negating result of logic operation in the ladder logic

The NOT contact negates the result of the logic operation (the “current flow”). You can use this contact to negate the result of a series connection, for example. In Fig. 7.11, “Coil 13” only has a signal state “0” if both “Contact 1” and “Contact 2” have the signal state “1”.



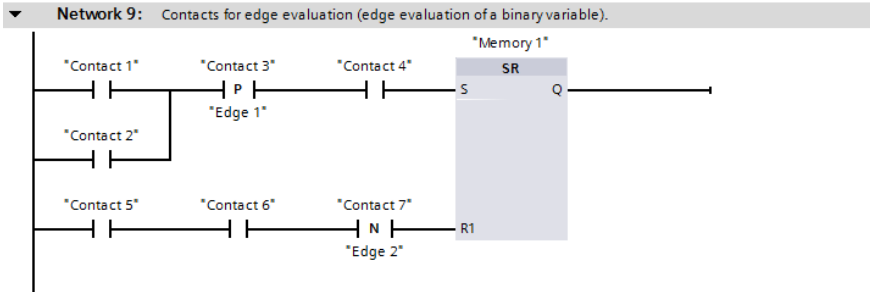
**Fig. 7.11** Example of NOT contact

In the bottom current path, a NOT contact is positioned following a parallel connection. In this case, “Coil 14” is only set to signal state “1” if both tags “Contact 3” and “Contact 4” have the signal state “0”.

You can position the NOT contact like a standard contact in a branch which commences on the left-hand power rail. Positioning following a T branch is also permissible. Positioning of the NOT contact is not permissible in a parallel branch which commences in the middle of the current path. The NOT contact can also be used to negate the result of the logic operation (the “current flow”) at box inputs and outputs.

### 7.2.8 Edge evaluation of a binary tag in ladder logic

The edge contact has the signal state “1” for one processing cycle if the signal state of the binary tags positioned above it changes from “0” to “1” (P contact, rising edge) or from “1” to “0” (N contact, falling edge). It responds like a “fleeting contact”. This “pulse” is linked to the result of the logic operation present prior to the contact.



**Fig. 7.12** Contacts for edge evaluation

The edge trigger flag is present underneath the edge contact. This is a flag or data bit which saves the signal state of the binary tag. The signal edge is recognized by comparing the signal states of binary tags and edge trigger flags (see also Chapter ).

The binary tag “Memory 1” in Fig. 7.12 is set at the moment when one of the tags “Contact 1” or “Contact 2” has the signal state “1”. A prerequisite is that the tag “Contact 4” also has the signal state “1” at that moment. “Memory 1” is reset at the moment when the series connection of “Contact 5” and “Contact 6” no longer has “current”.

The signals for setting and resetting the tag “Memory 1” are only present for one program cycle.

### 7.2.9 OK contact

The OK contact checks a floating-point tag for validity, i.e. whether the range limits for this data type have been observed. The contact is available in two versions:

- ▷ The OK contact is closed when the tag is valid.
- ▷ The NOT\_OK contact is closed when the tag is outside the permissible range.

The OK contact is programmed like a standard contact.

Fig. 7.13 shows a series connection of three contacts in the top current path. If all contacts are closed, i.e. if the tags “Contact 1” and “Contact 2” have the signal state “1” and if the floating-point tag “REAL-Var 1” is within the valid range, “Coil 1” is set to signal state “1”.

The tag “Coil 2” is set to signal state “1” if one of the tags “Contact 3” or “Contact 4” has the signal state “1” and if the floating-point tag “REAL-Var 2” is outside the permissible range.

### 7.2.10 Comparison contacts

A comparison function is shown in the ladder diagram as a “large” contact. It compares two digital tags. A comparison which is correct is equivalent to a closed contact (“current” is flowing through the comparison contact). The contact is open if the comparison is incorrect.

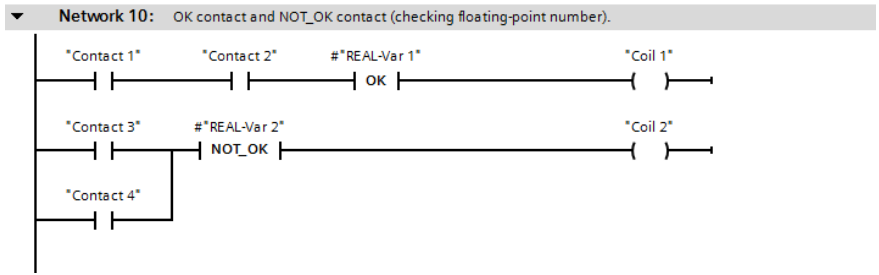


Fig. 7.13 OK and NOT\_OK contacts

Comparison functions are available for equal to, not equal to, greater than, greater than or equal to, less than, and less than or equal to. The comparison is carried out in accordance with the data type of the digital tags involved (for description, see Chapter 11.1 “Transfer functions” on page 356).

The comparison contact can be programmed wherever a standard contact can also be positioned. If comparison contacts are connected in series, both comparisons must be fulfilled so that current flows in the current path. If comparison contacts are connected in parallel, it is sufficient for one of the comparisons to be fulfilled so that current flows in the parallel connection.

The tag “Coil 3” in Fig. 7.14 is set to signal state “1” if the floating-point tags “REAL-Var 1” and “REAL-Var 2” have the same numerical value. “Coil 4” is set to “1” if the value of “REAL-Var 1” is greater than 125.75 or if the value of “REAL-Var 2” is less than  $12.5 \cdot 10^3$  and if, in both cases, the values of the tags “INT-Var 1” and “INT-Var 2” are not the same.

In addition there is the range comparison which is represented as a box. The box is positioned like a standard contact in the current path, with the input and output unnamed.

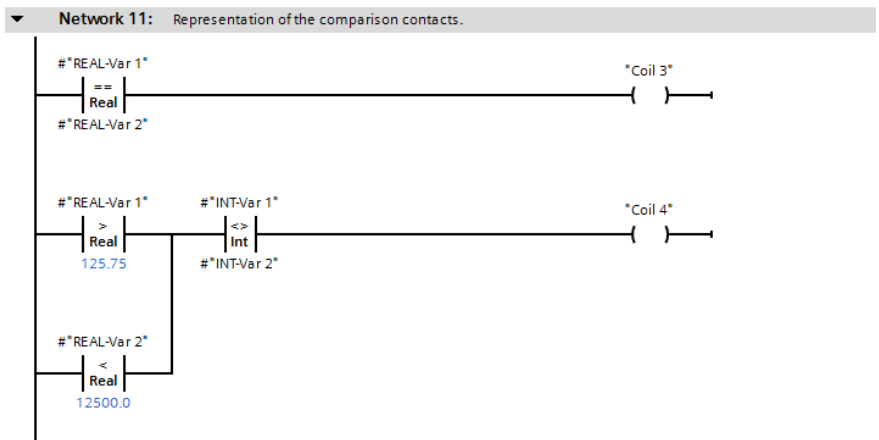


Fig. 7.14 Comparison of contacts

## 7.3 Programming with coils

Coils control binary tags such as outputs or bit memories. A simple coil sets the binary tag when current flows into the coil, and resets it when current no longer flows. The reverse is true for a negated coil.

There are coils with additional names and special functionalities, such as the set and reset coils or the coils for pulse generation during evaluation of signal edges. Coils can also be used to set and reset bit fields, start and reset timer functions, execute jumps in the program, and to terminate blocks (Fig. 7.15). The jump functions and the block end function are described in Chapter 7.6 “Functions for program flow control (LAD)” on page 241.

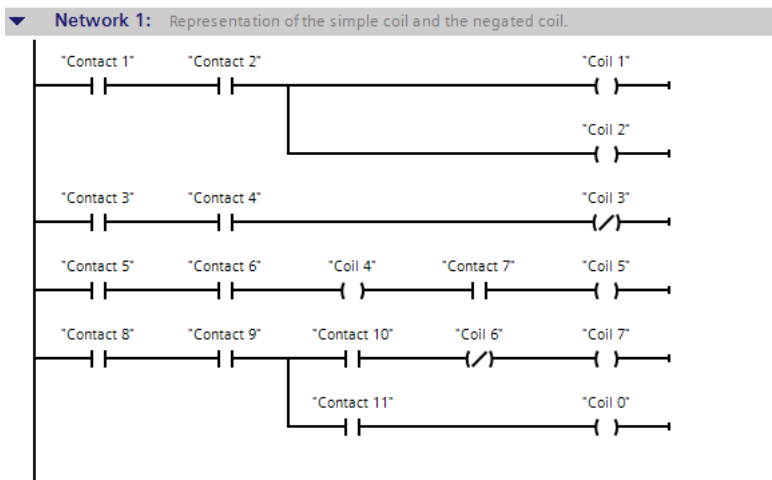
Coils			
Simple coil	$\text{---} \left( \text{ } \right) \text{---}$ <i>Binary tag</i>	Negated coil	$\text{---} \left( / \right) \text{---}$ <i>Binary tag</i>
Set coil	$\text{---} \left( S \right) \text{---}$ <i>Binary tag</i>	Reset coil	$\text{---} \left( R \right) \text{---}$ <i>Binary tag</i>
Pulse on positive edge	$\text{---} \left( P \right) \text{---}$ <i>Edge trigger flag</i>	Pulse on negative edge	$\text{---} \left( N \right) \text{---}$ <i>Edge trigger flag</i>
Multiple setting	$\text{---} \left( \text{SET\_BF} \right) \text{---}$ <i>Number</i>	Multiple resetting	$\text{---} \left( \text{RESET\_BF} \right) \text{---}$ <i>Number</i>
Start time	$\text{---} \left( \text{FKT} \right) \text{---}$ <i>Timer function</i> <i>Duration</i>	FKT: TP Pulse time TON ON delay TOF OFF delay TONR Accumulate time	
Reset time	$\text{---} \left[ \text{RT} \right] \text{---}$	Specify duration	$\text{---} \left( \text{PT} \right) \text{---}$ <i>Timer function</i> <i>Duration</i>
Jump with “1”	$\text{---} \left( \text{JMP} \right) \text{---}$ <i>Jump destination</i>	Jump with “0”	$\text{---} \left( \text{JMPN} \right) \text{---}$ <i>Jump destination</i>
Conditional block end	$\text{---} \left( \text{RET} \right) \text{---}$ <i>Binary tag</i>		

Fig. 7.15 Overview of coils available with LAD

### 7.3.1 Simple and negated coils

A simple coil directly assigns the current flow to the tag present on the coil: the tag is set when current flows into the coil, and is reset when current no longer flows. The negated coil negates the current flow in advance: the tag is set when no current flows into the coil, and is reset when current flows.

Fig. 7.16 shows the possible arrangements for simple and negated coils. The tags “Coil 1” and “Coil 2”, whose coils are connected in parallel in the first current path, react in the same manner: if “Contact 1” and “Contact 2” have the signal state “1”, the tags on the coils are set to “1”. In the second current path, the tag “Coil 3” for the negated coil has the signal state “0” if current flows through the series connection.



**Fig. 7.16** Simple and negated coils

Coils can also be positioned within a current path: “Coil 4” is set if “Contact 5” and “Contact 6” have the signal state “1”. If “Contact 7” is set to signal state “1” in addition, “Coil 5” is also set to “1”.

The current flow is not influenced by the coil, as is shown by the example in the current path underneath: if the tags “Contact 5”, “Contact 6” and “Contact 7” have the signal state “1”, the tag “Coil 6” (negated coil) has the signal state “0” and the tag “Coil 7” (simple coil) the signal state “1”.

The coils can be positioned following a T branch or directly on the left-hand power rail. Positioning in a parallel branch which does not commence on the left-hand power rail is not permissible (because of “jumping” by contacts).

### 7.3.2 Set and reset coil

If current flows into the set coil, the binary tag present above the coil is set to the signal state “1”. If current flows into the reset coil, the tag present above the coil is reset to the signal state “0”. If no current flows into the set or reset coil, the binary tag remains uninfluenced.

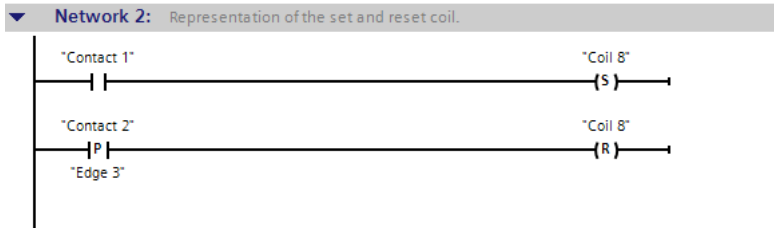


Fig. 7.17 Set and reset coil

In Fig. 7.17, “Contact 1” with a signal state “1” sets the tag “Coil 8”. “Contact 2” resets the tag “Coil 8” by means of a positive edge.

Set and reset coils can be connected in series or parallel, also in combination with simple and negated coils. They can also be positioned within a current path, on the left-hand power rail, or following a T branch. Positioning in a parallel branch which does not commence on the left-hand power rail is not permissible.

### 7.3.3 Retentive response due to latching

The memory function in a circuit diagram is usually realized through latching of the output to be triggered. This realization can also be integrated into the ladder diagram. However, compared to the memory box, it has the disadvantage that the memory function is not recognized immediately. The latching principle is simple: the binary tag triggered by the coil is scanned, and this scan (the “coil contact”) is connected in parallel to the set condition.

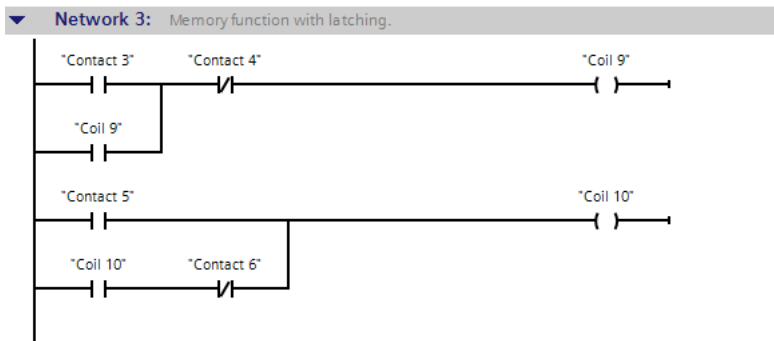


Fig. 7.18 Retentive response due to latching



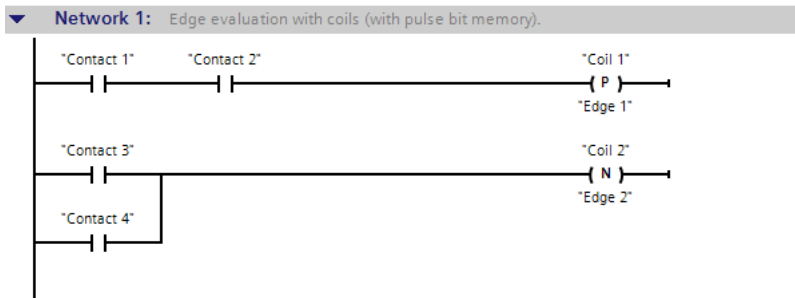
Fig. 7.18 shows both types of memory function through latching, namely set dominant and reset dominant. If “Contact 3” closes, “Coil 9” pulls up and closes the contact parallel to “Contact 3”. If “Contact 3” then opens again, “Coil 9” remains triggered. “Coil 9” drops out if “Contact 4” opens. If signal state “1” is present at both “Contact 3” and “Contact 4”, no current flows into the coil (reset dominant). This situation looks different in the bottom current path: if signal state “1” is present at both “Contact 5” and “Contact 6”, current flows into the coil (set dominant).

### 7.3.4 Edge evaluation with pulse output in the ladder logic

The P and N coils are available for edge evaluation with coils. The binary tag present above the P coil is set for the duration of one program cycle if the signal state changes from “0” to “1” prior to the P coil (rising edge). With the N coil, the binary tag present above the coil is set for the duration of one program cycle in the case of a falling edge.

The binary tag present above the coil is referred to as a “pulse bit memory”. Suitable for pulse bit memories are operand types which are not connected “outside” to modules, for example tags from the bit memory or data areas. The edge trigger flag is present under the coil, and must be a different tag for each edge evaluation (see Chapter 10.3 “Edge evaluation” on page 338).

In Fig. 7.19, the tag “Coil 1” has the signal state “1” for the duration of one program cycle if both tags “Contact 1” and “Contact 2” have the signal state “1”. “Coil 2” has the signal state “1” for the duration of one program cycle if both tags “Contact 3” and “Contact 4” have the signal state “0”.



**Fig. 7.19** Edge evaluation with coils (“pulse bit memories”)

Edge coils can be positioned within a current path or terminate a current path. Edge coils can also be programmed following a T branch. A direct connection to the left-hand power rail is not advisable.

If an edge coil is followed by further program elements, for example if the edge coil has been positioned within a current path, the signal state at the input of the edge coil is passed on directly to the coil output.

### 7.3.5 Multiple setting and resetting (filling of bit field) in the ladder logic

If the result of the logic operation is “1”, the SET\_BF coil sets the bits of a bit field to signal state “1”. The bit field is defined by the start tag present above the coil and the number of bits under the coil. If the result of the logic operation is “1”, the RESET\_BF coil resets the bits in the bit field. There is no response if the result of the logic operation is “0”.

In the example in Fig. 7.20, the bit field for the SET\_BF coil is defined by the start tag “Bitfield0” which is followed by seven bits (thus a total of eight bits). The bit field for the RESET\_BF coil is in the data block “Data180”, commences with the field (binary) component Bitfield[1], and ends after 15 subsequent bits.

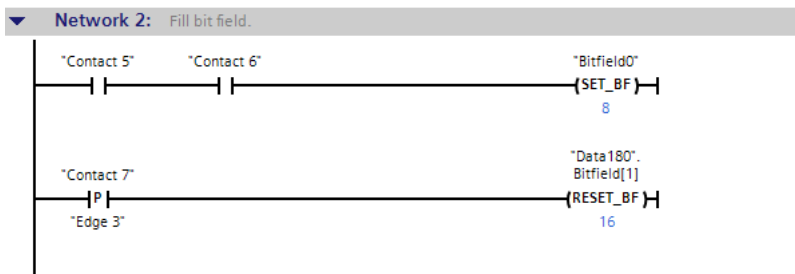


Fig. 7.20 Filling of bit field with SET\_BF and RESET\_BF

SET\_BF and RESET\_BF terminate the current path. If the coils are positioned directly on the left-hand power rail, the function is always executed.

### 7.3.6 Starting IEC timer functions in the ladder logic with coils

You can use the timer functions to implement timing processes in the program such as waiting and monitoring times, measurement of a time interval, or the generation of pulses. An IEC timer function can be started with two different program elements: with a coil or with a Q box (see Chapter 7.4.5 “Controlling IEC timer functions in the ladder logic with Q boxes” on page 230). Both variants are equally useful. A detailed description of the timer functions is provided in Chapter 10.4 “Time functions” on page 344.

A timer function can be started with one of the four behavior patterns TP, TON, TOF, and TONR. A timer function requires internal data for each application. You can specify where this data is to be saved when programming: For the *Single instance* entry in its own data block with the data type IEC\_TIMER and for the *Multi-instance* entry in the instance data block of the calling function block with a data type that depends on the behavior of the timer function (TP\_TIME, TON\_TIME, TOF\_TIME, TONR\_TIME). You address a timer function with the name of the instance data – data block or local data.

The coil to start a timer function requires a preceding logic operation. It can only be placed at the end of a current path. Under the coil is the duration with which the timer function is started.

A timer function is reset using the RT coil. The RT coil can be programmed in the middle of a current path or as its termination.

The PT coil sets the duration of a timer function. Each processing with signal state “1” overwrites the duration in the instance data with the value given under the coil.

Fig. 7.21 shows the coils used in connection with IEC timer functions. In the first current path, the timer function in the local data with the name *#Timer* is started as ON delay with the value *#Duration*. The status of the timer function can be scanned with the structure component *Q*. The example shows starting the timer function, resetting the timer function, and setting the duration with a rising edge.

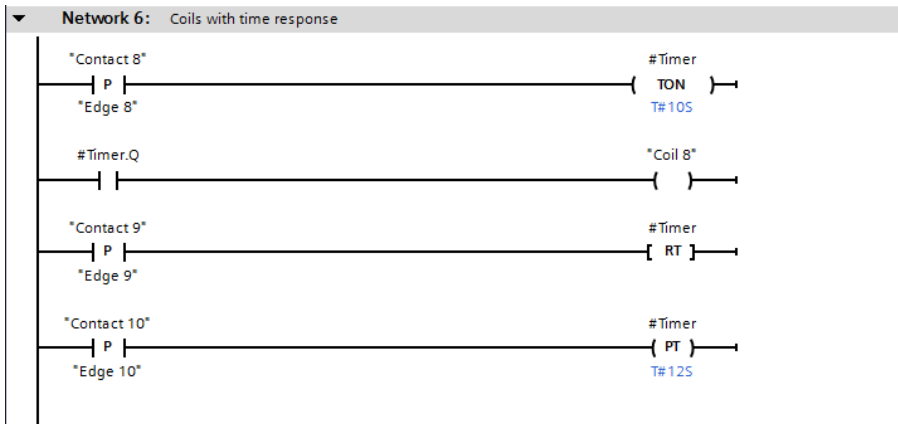


Fig. 7.21 Processing a timer function with coils

## 7.4 Programming with Q boxes in the ladder logic

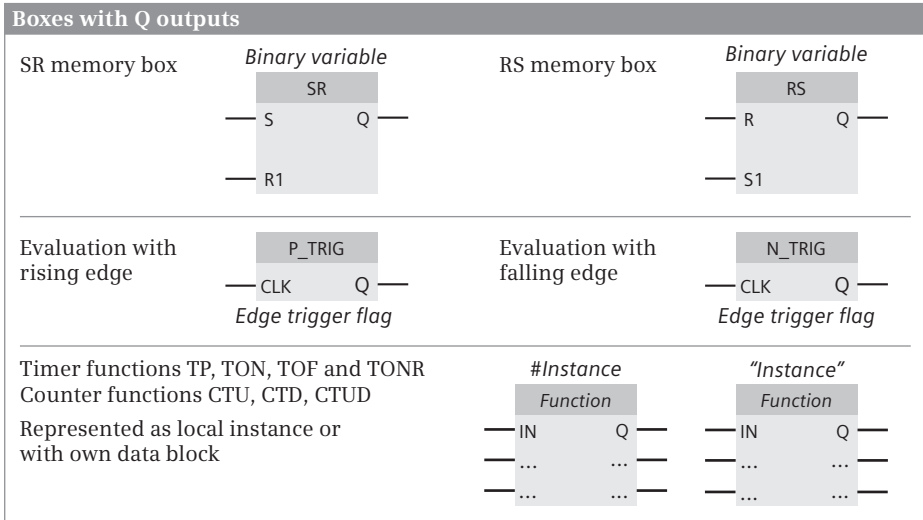
Q boxes have a binary output named “Q” which can be linked further. Q boxes are used to represent memory functions, edge evaluations, and timer and counter functions (Fig. 7.22).

With Q boxes, the first binary input (and in certain cases the associated parameter) must be connected, connection of the other inputs and outputs is optional. The binary inputs of Q boxes cannot be directly connected to the left-hand power rail.

### 7.4.1 Arrangement of Q boxes in the ladder logic

When using Q boxes as program elements, you can:

- ▷ Program one single box per network, either within the current path or as its termination



**Fig. 7.22** Overview of Q boxes available with LAD

- ▷ Arrange boxes in series by connecting the Q output of one box to a binary input of the following box, and
- ▷ Position boxes following T branches and in branches which commence on the left-hand power rail

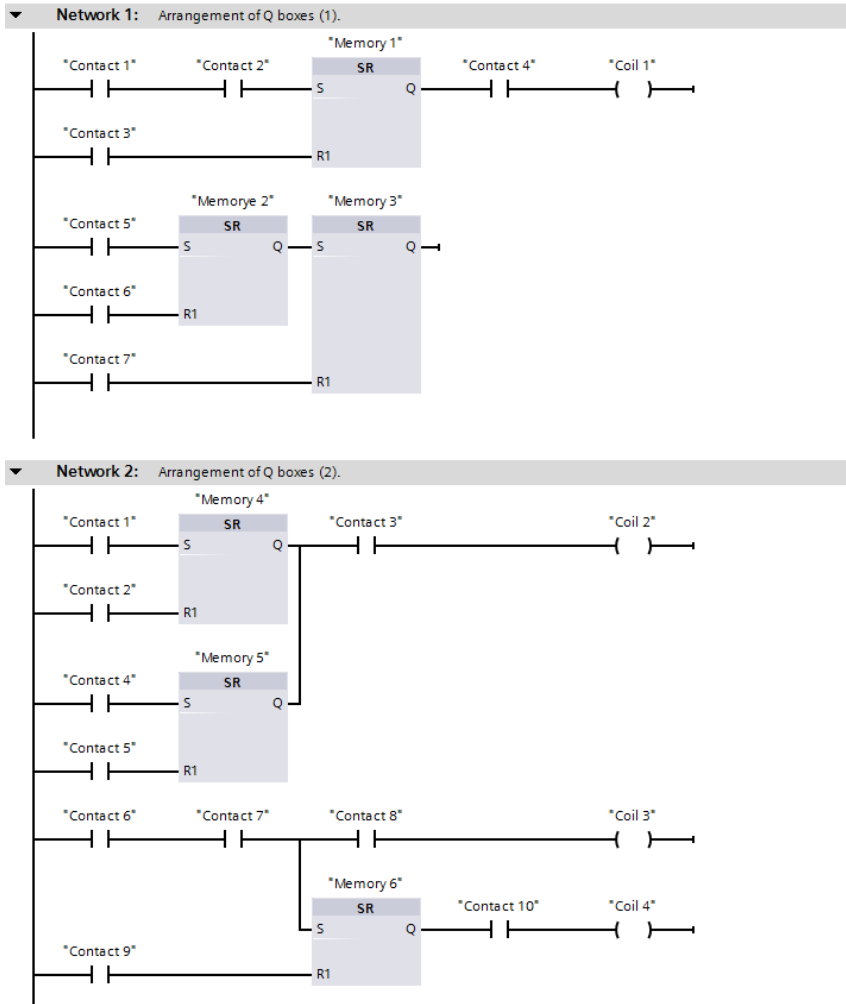
The circuits shown in Fig. 7.23 for the positioning of Q boxes use the memory box with two binary inputs as an example. This enables the possible positioning of all Q boxes to be shown.

### 7.4.2 Memory boxes in the ladder logic

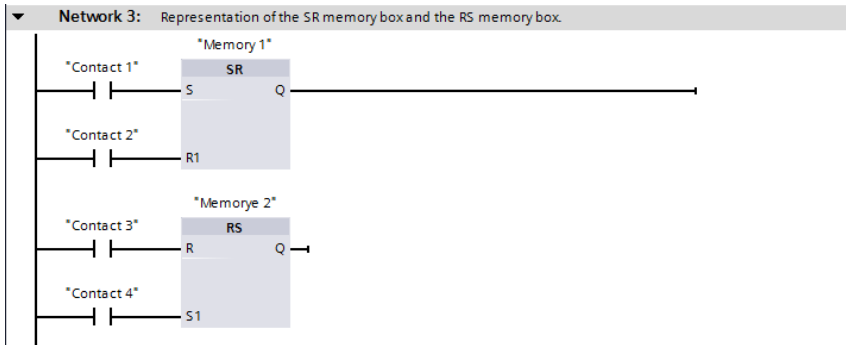
There are two versions of the memory function as box: as SR box (reset dominant) and as RS box (set dominant). In addition to the difference in the function name, the two boxes also differ in the positioning of the set and reset inputs.

The binary tag named above the memory box is set when the set input has signal state “1” and the reset input signal state “0”. The binary tag is reset when “1” is present at the reset input and “0” at the set input. Signal state “0” at both inputs has no influence on memory functions. If signal state “1” is present simultaneously at both inputs, the two memory functions respond differently: the SR memory function is reset, the RS memory function is set.

If both tags “Contact 1” and “Contact 2” in Fig. 7.24 have the same signal state “1”, “Memory 1” is reset (box input R1 is dominant). If both “Contact 3” and “Contact 4” have the same signal state “1”, “Memory 2” is set (box input S1 is dominant).



**Fig. 7.23** Positioning of Q boxes using example of SR memory function



**Fig. 7.24** Memory boxes SR and RS

### 7.4.3 Edge evaluation of current flow

The edge evaluation with Q boxes registers a change in the current flow prior to the box. If the signal state changes from “0” to “1” (rising edge) at the CLK input of the P\_TRIG box, signal state “1” is present at the Q output for the duration of one program cycle. If the result of the logic operation changes from “1” to “0” (falling edge) at the CLK input of the N\_TRIG box, the Q output is activated for the duration of one program cycle.

In Fig. 7.25, the tag “Memory 3” is set at the moment when both tags “Contact 5” and “Contact 6” have the signal state “1”. “Memory 3” is reset at the moment when both tags “Contact 7” and “Contact 0” have the signal state “1”.

The edge boxes may only be positioned within a current path.

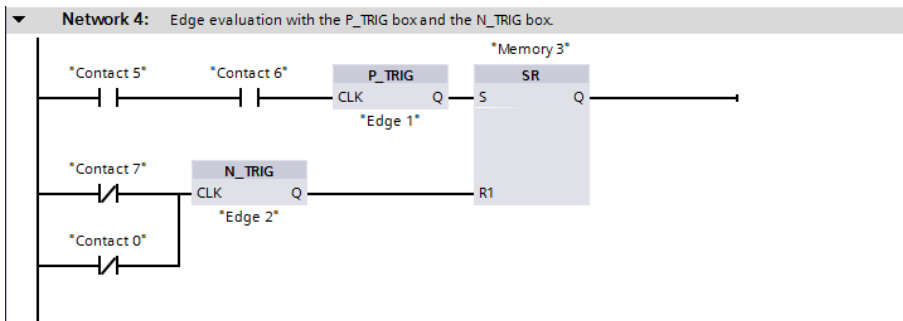
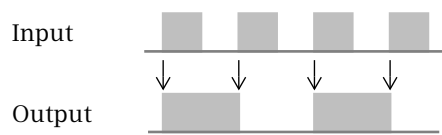


Fig. 7.25 Example of Q boxes for edge evaluation

### 7.4.4 Example of binary scaler in the ladder logic

A binary scaler has one input and one output. If the signal at the input of the binary scaler changes its state, e.g. from “0” to “1”, the output also changes its signal state. This (new) signal state is then retained until the next change, which is positive in our example. Only then does the signal state of the output change again. Half of the input frequency is then present at the binary scaler's output.



There are various methods for solving this task, two of which are shown below.

The first solution uses memory functions (Fig. 7.26). If the tag “Input 1” has the signal state “1”, the tag “Output 1” is then set (“Memory 1” is still reset). If the signal state at “Input 1” changes to “0”, “Memory 1” is also set (“Output 1” is then “1”). If “Input 1” is “1” the next time around, “Output 1” is reset again (“Memory 1” is now “1”). If “Input 1” is “0” again, “Memory 1” is reset (since “Output 1” is now also reset). The “basic state” has now been reached again following two input pulses and one output pulse.

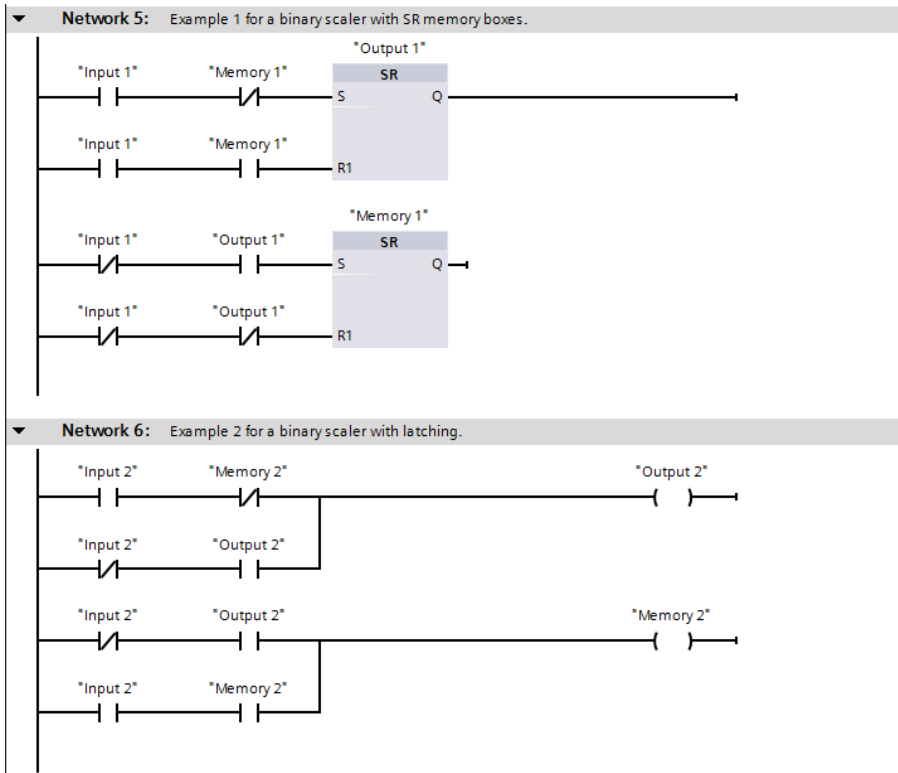


Fig. 7.26 Examples of binary scalers in ladder logic

The second solution uses the latching principle common to circuit diagrams (Network 4). The operating principle is the same as with the first solution, but the reset condition – as usual with latching – is “zero-active”.

#### 7.4.5 Controlling IEC timer functions in the ladder logic with Q boxes

You can use the timer functions to implement timing processes in the program such as waiting and monitoring times, measurement of a time interval, or the generation of pulses. An IEC timer function can be started with two different program elements: a coil or a Q box (see Chapter 7.3.6 “Starting IEC timer functions in the ladder logic with coils” on page 225). Both variants are equally useful. A detailed description of the timer functions is provided in Chapter 10.4 “Time functions” on page 344.

A timer function can be started with one of the four behavior patterns TP, TON, TOF, and TONR. A timer function requires internal data for each application. You can specify where this data is to be saved when programming: For the *Single instance* entry in its own data block with the data type IEC\_TIMER and for the *Multi-instance* entry in the instance data block of the calling function block with a data type that depends on the behavior of the timer function (TP\_TIME, TON\_TIME, TOF\_TIME,

TONR\_TIME). You address a timer function with the name of the instance data – data block or local data.

The top timer function “Timer 1” in Fig. 7.27 saves its data as a local instance with the name #“Timer 1” in the calling function block. This function is started with “Contact 1” and “Duration 1”. The tag “Coil 1” has the signal state “1” for as long as defined by the tag “Duration 1”.

The bottom timer function “Timer 2” saves its data in a separate data block “Timer 2”. This function is started with “Contact 2” and “Duration 2”. After expiry of the duration, the tag “Coil 2” has signal state “1”.

The name of the local instance (#“Timer 1”) and the name of the data block (“Timer 2”) address the respective timer functions. Component Q of the data structure supplies the status of the timer function.

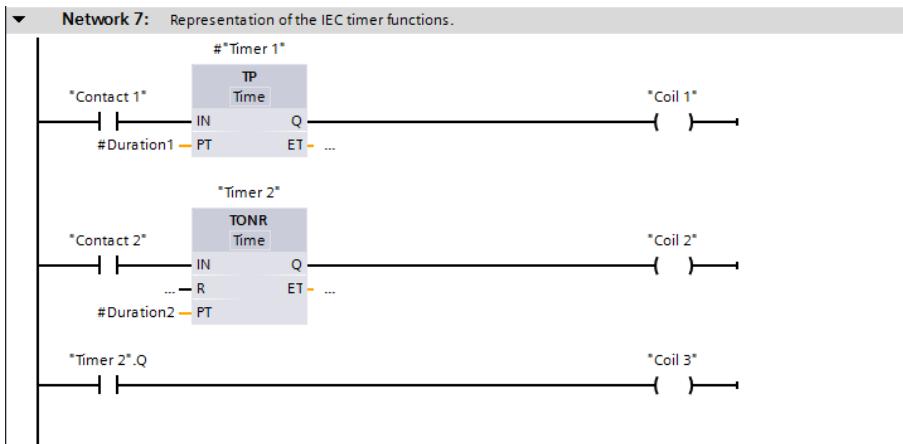


Fig. 7.27 Examples of timer functions

#### 7.4.6 Controlling IEC counter functions in the ladder logic with Q boxes

You can use the IEC counter functions to execute counting tasks directly using the control processor. The counter functions can count up and down; the numerical range depends on the data type of the preset value. The data types USINT, UINT, UDINT, SINT, INT and DINT are available.

The counting frequency of the counter functions depends on the execution time of the user program. In order to count, the CPU must recognize a change in the signal state of the input pulse, i.e. the input pulse and the pause must be present for at least one program cycle. The longer the program execution time, the lower the counting frequency. A detailed description of the counter functions is provided in Chapter 10.5 “Counter functions” on page 349.



A counter function can be controlled with one of the three behavior patterns CTU, CTD, and CTUD. A counter function requires internal data for each application. You can specify where this data is to be saved when programming: by specifying *Single instance* for storage in a separate data block, and by specifying *Multi-instance* for storage in the instance data block of the calling function block.

The data type of a counter function is based on the data type of the count value. If, for example, an up-counter (CTU) with a DINT count value is programmed as a single instance, the data type IEC\_DCOUNTER is taken as a basis for the data block (see Chapter 4.8.2 “IEC\_COUNTER system data type” on page 112); as a local instance, the counter function has the data type CTU\_DINT (see Chapter 4.6.2 “Parameter types for IEC counter functions” on page 108). You address the counter function with the name of the instance data – data block or local data.

The top counter function “Counter 1” in Fig. 7.28 saves its data as a local instance in the calling function block. The current count value “Count value 1” is set by “Contact 4” to zero. “Contact 3” increments the current count value by one unit with each positive edge. If the count value reaches the default value “Preset value 1” and then exceeds it, the tag “Coil 3” at output Q is set.

The second counter in the example is an up/down counter. “Contact 7” sets the current count value to zero, “Contact 8” loads the default value “Preset value 2” as the current count value. “Contact 5” increments the count value by one unit with each positive signal change, “Contact 6” decrements the count value by one unit with each positive signal change.

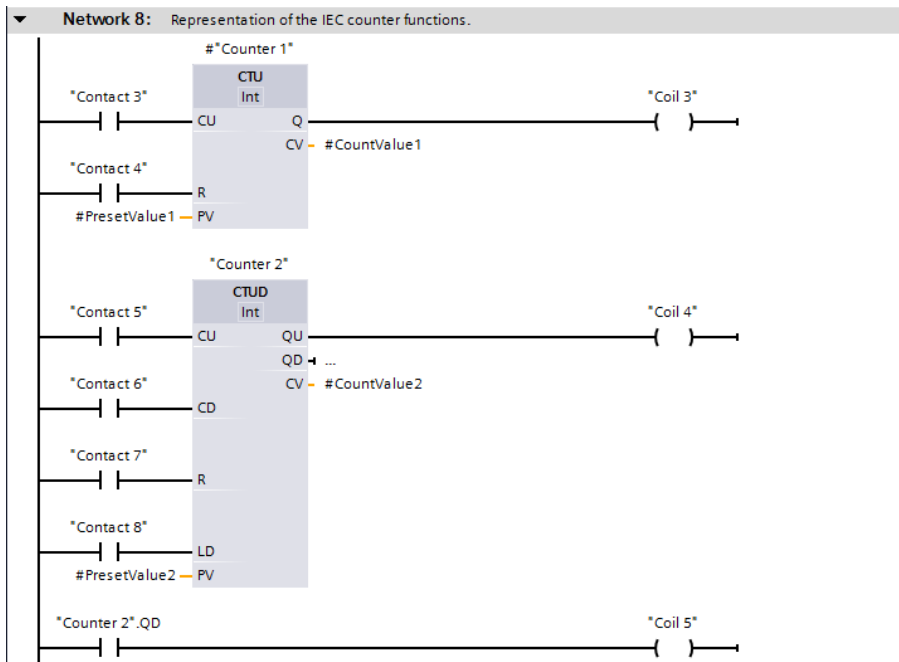


Fig. 7.28 Examples of counter functions

The QU output has the signal state “1” if the actual count value at the CV output is equal to or greater than the default value at the PV input. The QD output has the signal state “1” if the actual count value is zero or less than zero.

The upper QU output can be further connected directly. In the example, it is used to control the tag “Coil 4”. The QD output cannot be supplied, but can be scanned indirectly via the corresponding component QD of the counter structure. (For the QU output, this would be the component QU.)

The name of the local instance (“Counter 1”) and the name of the data block (“Counter 2”) address the respective counter function. In the example, “Counter 2” has its own data block, and the QD output is scanned as usual in LAD with a contact named “Counter 2”.QD. The result of the scan can be connected further, e.g. assigned directly to a coil.

## 7.5 Programming with EN/ENO boxes in the ladder logic

EN/ENO boxes have an enabling input EN and an enabling output ENO. The enabling input can be used to suppress processing of the box. If an error occurs while the box is being processed, this is displayed at the enabling output.

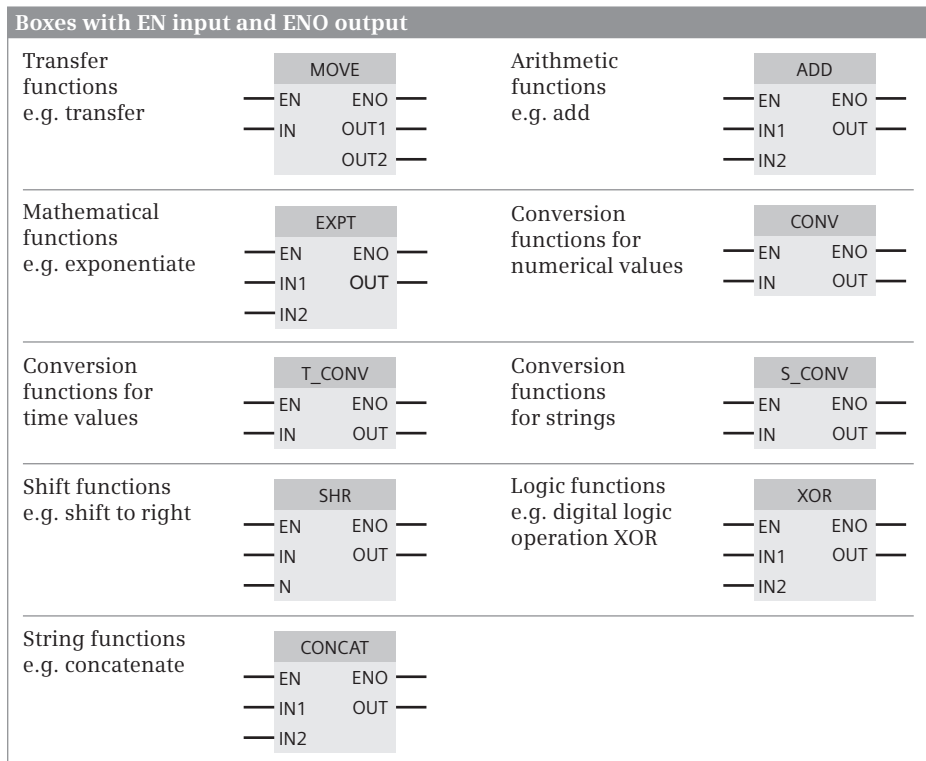


Fig. 7.29 Overview of boxes with enable input EN and enable output ENO

A detailed description of the functions with EN/ENO boxes can be found in the corresponding sections. Programming of the EN/ENO boxes in the ladder logic representation is of prime importance here. Fig. 7.29 provides an overview of the functions implemented with EN/ENO boxes.

The parameters of the EN/ENO boxes must all be connected. The enabling input EN and the enabling output ENO are not parameters of the box function. They are used for processing boxes, and are added by the program editor to the box function.

A detailed description of EN and ENO and how one can use the EN/ENO mechanism with self-created blocks can be found in Chapter 12.4.1 “EN/ENO mechanism with LAD and FBD” on page 418. The block calls in the ladder diagram, which are also shown as EN/ENO boxes, are described in Chapter 7.6.5 “Block call functions in the ladder logic” on page 245.

### 7.5.1 Positioning of EN/ENO boxes in the ladder logic

Fig. 7.30 uses the MOVE function to show the positioning of EN/ENO boxes in a current path. An EN/ENO box can be positioned on its own in a network, with or without connection of the EN input or the ENO output. The ENO output can be connected to the EN input of the following box. By means of a contact at the beginning of this

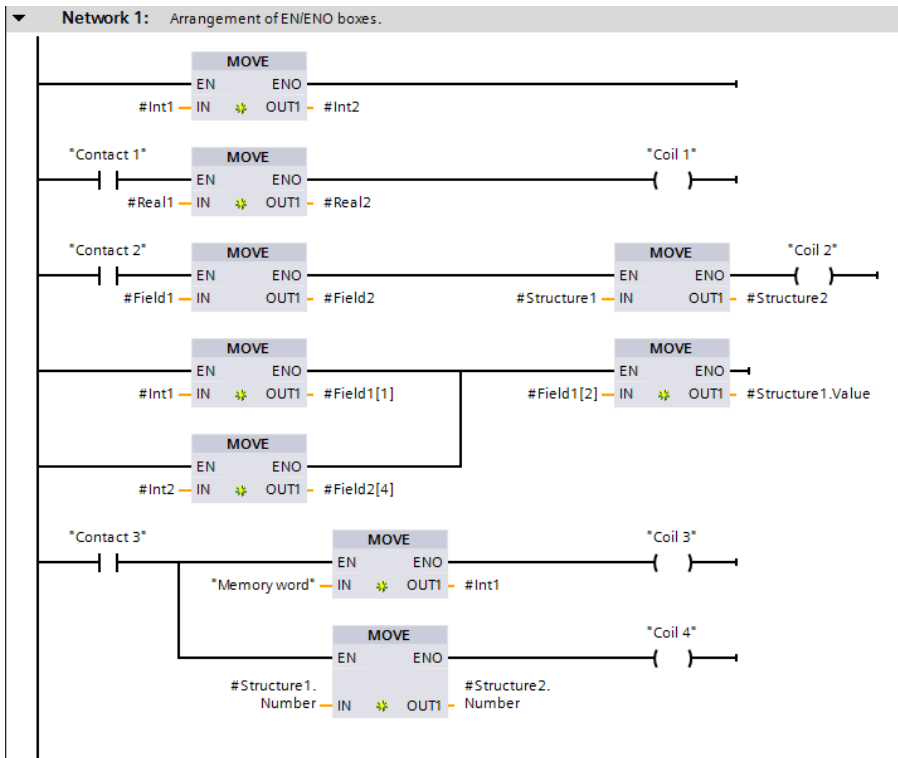


Fig. 7.30 Positioning of EN/ENO boxes using example of MOVE box

“main current path” it is possible to switch processing of the entire current path on and off. The signal state of the ENO output of the last box indicates by means of a “1” that the complete sequence has been processed without errors.

The ENO output of a box can be connected in parallel with the ENO output of a different box if the boxes are present in separate current paths which commence on the left-hand power rail. “Current” then flows in the subsequent path if one of the two boxes has completed the processing without errors.

If an EN/ENO box is positioned in a T branch, its ENO output can no longer be returned to the path at which the T branch commences.

### 7.5.2 Transfer functions in the ladder logic

Boxes with the following transfer functions are available in the programming language LAD:

- ▷ Copy an operand or tag (MOVE)
- ▷ Read (FieldRead) and write (FieldWrite) a field component with variable index
- ▷ Copy a data area (MOVE\_BLK) and copy a data area without interruption (UMOVE\_BLK)
- ▷ Fill a data area (FILL\_BLK) and fill a data area without interruption (UFILL\_BLK)
- ▷ Exchange the bytes within a tag (SWAP)

A detailed description of the transfer functions is provided in Chapter 11.1 “Transfer functions” on page 356.

Fig. 7.31 shows an example of the programming of transfer functions. If the tag “Contact 1” changes the signal state from “0” to “1”, the MOVE and FILL\_BLK boxes will be executed.

The MOVE box copies the content of tag #Int1 to tag #Field1[1]. #Field1[1] is a component of the tag #Field1 with the same data type as the tag #Int1.

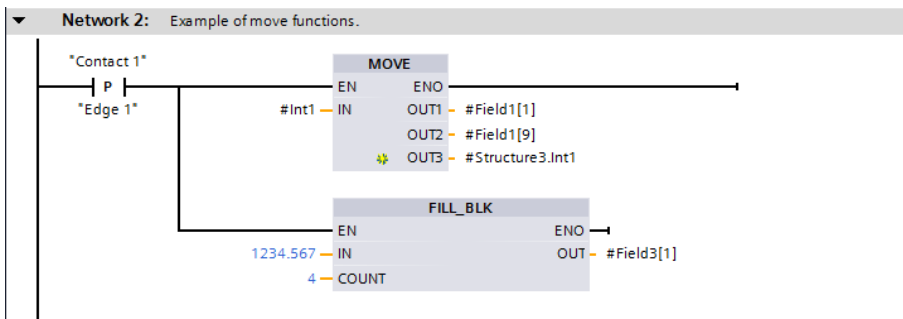


Fig. 7.31 Example of the transfer functions in the ladder logic

The MOVE box can be provided with further outputs: select the MOVE box and then the *Insert output* command from the shortcut menu. In the example, the content of #Int1 is also transferred to the tag #Field1[9] and to the component #Structure1.Int1.

The FILL\_BLK box in the example transfers the value 1234.567 to four successive components of the field tag #Field3, starting with #Field3[1]. #Field3 consists of components of data type REAL.

### 7.5.3 Arithmetic functions for numerical values in the ladder logic

Boxes with the following arithmetic functions for numerical values are available in the programming language LAD:

- ▷ Add (ADD), subtract (SUB), multiply (MUL) and divide (DIV) two numerical values
- ▷ Divide with remainder as result (MOD)
- ▷ Form absolute value (ABS), negation (NEG, multiplication by  $-1$ ), decrement (DEC, reduce numerical value by 1) and increment (INC, increase numerical value by 1)

A detailed description of these arithmetic functions is provided in Chapter 11.3 “Arithmetic functions for numerical values” on page 366.

Fig. 7.32 shows an example of the arithmetic functions with numerical values. Two tags of data type INT are added, and the intermediate result is saved in the temporary tag #t\_Int1. This intermediate result is multiplied by  $-1$  and output to tag #Int3. The absolute value of tag #Int1 is generated; this value is divided by 3 and the remainder of the division written to tag #Int4.

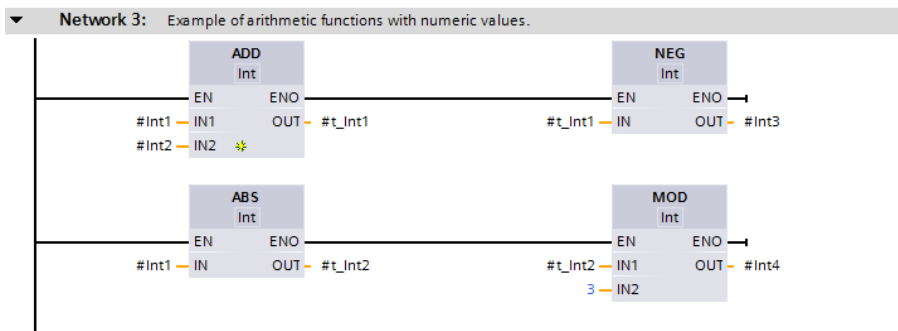


Fig. 7.32 Example of arithmetic functions for numerical values in the ladder logic

### 7.5.4 Arithmetic functions for time values in the ladder logic

In the LAD programming language, durations (time spans, data type TIME) and points in time (date and time, data type DTL) can be interlinked. Boxes with the following arithmetic functions are available for this:

- ▷ Add two durations, or add one duration to a point in time (T\_ADD)
- ▷ Subtract two durations, or subtract one duration from a point in time (T\_SUB)
- ▷ Generate the difference between two points in time (T\_DIFF)

A detailed description of these arithmetic functions is provided in Chapter 11.4 “Arithmetic functions for time values” on page 369.

Fig. 7.33 shows an example of arithmetic functions with time values. The difference between the tags #Date1 and #Date2 is generated. The result is a duration in TIME format. Eight hours are added to this duration, and the result output to tag #Duration2.

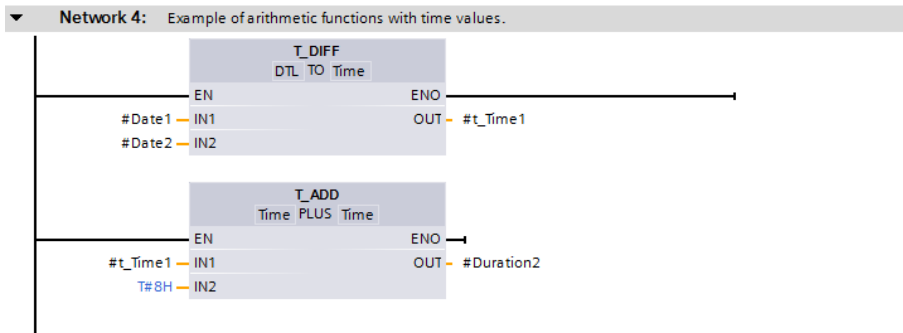


Fig. 7.33 Example of arithmetic functions for time values in the ladder logic

### 7.5.5 Math functions in the ladder logic

Boxes with the following math functions are available in the programming language LAD:

- ▷ Trigonometric functions: sine (SIN), cosine (COS), tangent (TAN)
- ▷ Arc functions: arcsine (ASIN), arccosine (ACOS), arctangent (ATAN)
- ▷ Form square (SQR) and square root (SQRT)
- ▷ Exponential function to base e (EXP) and to any base (EXPT)
- ▷ Natural logarithm (LN)
- ▷ Extract decimal places (FRAC)

A detailed description of these math functions is provided in Chapter 11.5 “Mathematical functions” on page 372.

Fig. 7.34 shows an example of the math functions.

The tag #c is calculated according to the  $c = \sqrt{a^2 + b^2}$  equation. The square of #a is formed first. When inputting tag names – which can also be keywords (in the input, “a” can also stand for “output”) or which can have the same name both locally

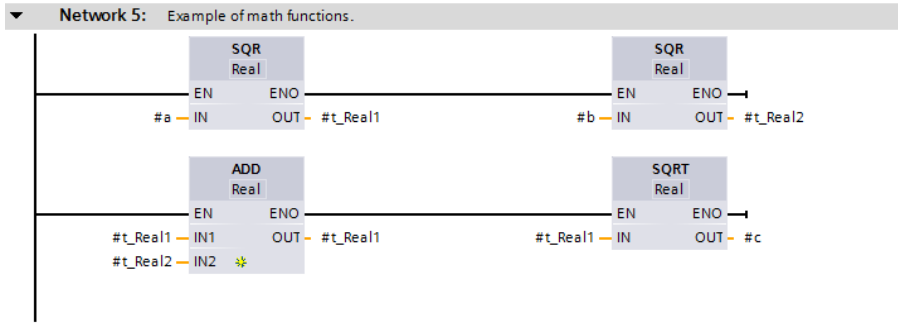


Fig. 7.34 Example of math functions in the ladder logic

and globally – the tag must be labeled accordingly: for a local tag with a preceding number sign (#), for a global tag with the name in quotation marks, and for an operand with a preceding percent sign (%).

In the example, the squares of #a and #b are stored temporarily and added. #t\_Real1 is used again for the buffer. The result of the square root extraction is saved in the tag #c.

### 7.5.6 Conversion functions in the ladder logic

Boxes with the following conversion functions are available in the programming language LAD:

- ▷ CONV (conversion of BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, REAL, LREAL, BCD16, BCD32)
- ▷ ROUND, FLOOR, CEIL, TRUNC (conversion of REAL, LREAL into SINT, INT, DINT, USINT, UINT, UDINT, REAL, LREAL)
- ▷ SCALE\_X, NORM\_X (scaling and standardization)
- ▷ T\_CONV (conversion of TIME into DINT and vice versa)
- ▷ S\_CONV, STRG\_VAL, VAL\_STRG (conversion of SINT, INT, DINT, USINT, UINT, UDINT, REAL into STRING and vice versa)

A detailed description of the conversion functions is provided in Chapter 11.6 “Conversion functions (Conversion of data type)” on page 376.

Fig. 7.35 shows an example of the conversion functions.

The conversion function CONV is used to convert a 7-digit BCD number into a DINT format number and subsequently into REAL format (tag #t\_Real2). The value of #t\_Real2 is divided by  $10^7$  and converted into a fixed-point number between the limits of -150 and +250.

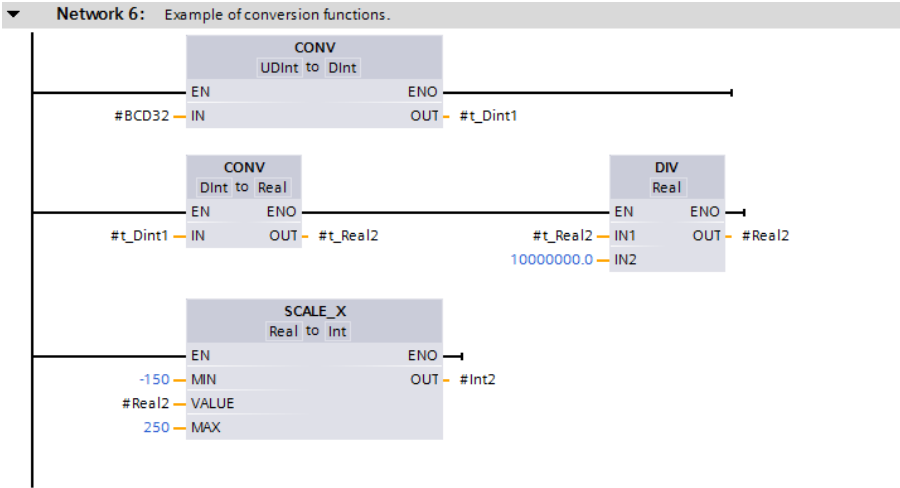


Fig. 7.35 Example of the conversion functions in the ladder logic

### 7.5.7 Shift functions in the ladder logic

Boxes with the following shift functions are available in the programming language LAD:

- ▷ Shift to right (SHR) and left (SHL)
- ▷ Rotate to right (ROR) and left (ROL)

A detailed description of the shift functions is provided in Chapter 11.7 “Shift functions” on page 389.

Fig. 7.36 shows an example of the shift functions. The content of tag #Int2 is shifted three places to the right and output to tag #Int3. Shifting of fixed-point numbers one place to the right is equivalent to a division by two. In the example, tag #Int2 is divided by eight ( $2^3$ ) and the rounded-off result output to tag #Int3.

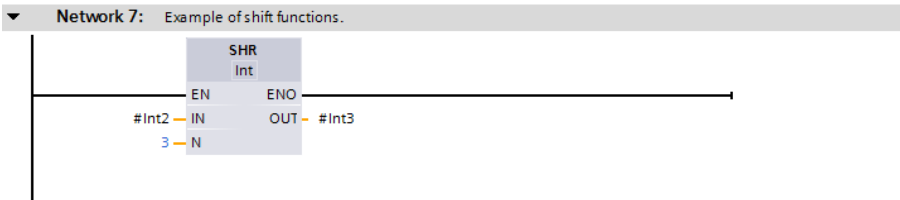


Fig. 7.36 Example of the shift functions in the ladder logic



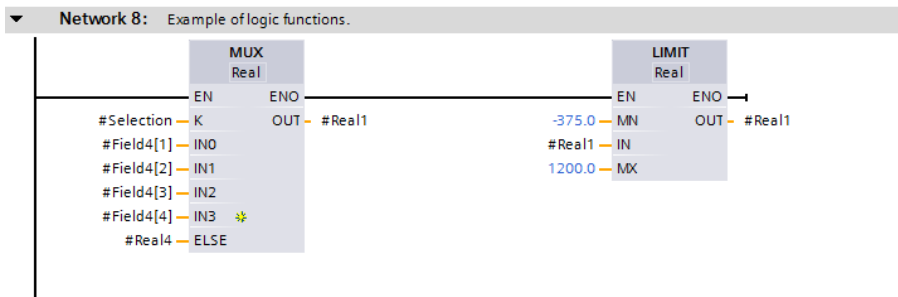
### 7.5.8 Logic functions in the ladder logic

Boxes with the following logic functions are available in the programming language LAD:

- ▷ Digital logic operations AND, OR and XOR
- ▷ Invert (INV)
- ▷ Code bit (DECO) and set bit number (ENCO)
- ▷ Selection functions (SEL, MUX), minimum and maximum selection (MIN, MAX), limiter (LIMIT)

A detailed description of the logic functions is provided in Chapter 11.8 “Logic functions” on page 392.

Fig. 7.37 shows an example of the logic functions.



**Fig. 7.37** Example of the logic functions in the ladder logic

The MUX function is used to select the components whose number is present in the tag #Selection from the first four components of field tag #Field4. For example, if the tag #Selection has a value of 3, the tag #Field4[3] will be selected.

If the value of #Selection is not between 1 and 4, the value of the tag #Real4 is used as a substitute. The result of the selection is limited to the range between -375 and +1200 and output to #Real1.

The MUX box has been extended in the example by two inputs: select the box when programming and then the *Insert input* command from the shortcut menu.

### 7.5.9 Functions for strings in the ladder logic

Boxes with the following functions for strings are available in the programming language LAD:

- ▷ LEN        Outputs the length of a string
- ▷ CONCAT    Combines two strings together
- ▷ LEFT       Outputs the left part of a string

- ▷ RIGHT Outputs the right part of a string
- ▷ MID Outputs the middle part of a string
- ▷ DELETE Deletes part of a string
- ▷ INSERT Inserts characters into a string
- ▷ REPLACE Replaces characters in a string
- ▷ FIND Outputs the position of a searched character.

A detailed description of these functions is provided in Chapter 11.9 “Processing of strings (Data type STRING)” on page 398.

Fig. 7.38 shows an example of string functions. The tag #String1 has the STRING format and is 24 characters long. By means of the LEFT box, 16 characters are removed left-justified from the tag and saved in the intermediate memory #t\_String. REPLACE replaces characters in a string. In the example, the characters '0123' are replaced in the tag #t\_String starting at the 10th position, and the complete string is written in tag #String2.

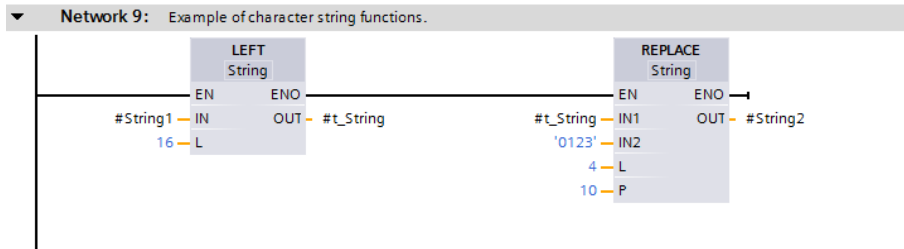


Fig. 7.38 Example of functions for strings in the ladder logic

## 7.6 Functions for program flow control (LAD)

The functions for program flow control are:

- ▷ The jump functions to continue program execution in the desired network
- ▷ The jump list to select a jump destination depending on a numerical value
- ▷ The jump distributor for selecting a jump destination depending on number ranges
- ▷ The block end function to end program execution in the block
- ▷ The block call functions for calling functions and function blocks

Fig. 7.39 shows an overview of these functions. A detailed description of these functions is provided in Chapter 12 “Program flow control” on page 406.

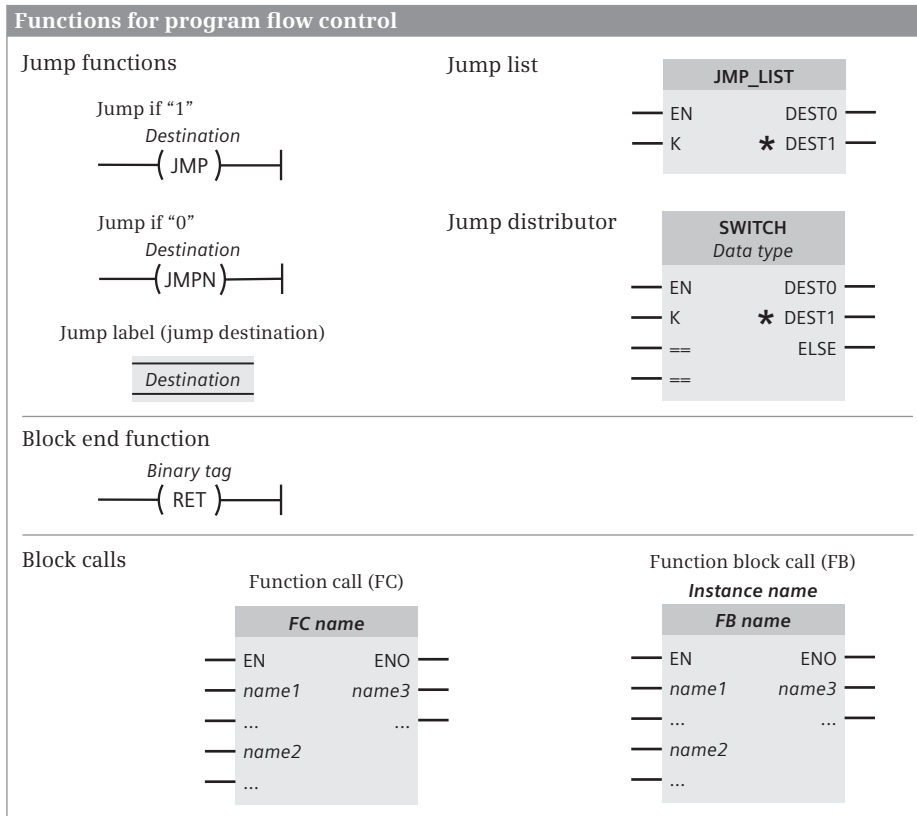


Fig. 7.39 Overview of functions for program flow control in the ladder logic

### 7.6.1 Jump functions in the ladder logic

A *JMP* or *JMPN* jump function is used to exit the linear processing in a block and – depending on the result of the preceding logic operation– continue this processing in another network in the block. If *JMP* is connected with the left power rail, the jump is always performed. To program a jump function, drag the *JMP* or *JMPN* function from the program elements catalog under *Basic instructions > Program control operations* to the working area.

The jump label above the jump function defines the jump destination, which must be at the beginning of a network. To program the jump destination, drag the *Label* function from the program elements catalog under *Basic instructions > Program control operations* to the working area.

Fig. 7.40 shows a jump function using a program loop as an example. In a *#Current* data field with 16 components from *#Current[0]* to *#Current[15]*, the maximum value is searched for. The tags *#Index* and *#MaxValue* are initialized with the value 0. A comparison function in the program loop compares the value of *#MaxValue* with the value of *#Current[#Index]*. If *#MaxValue* is less than *#Current[#Index]*, it is

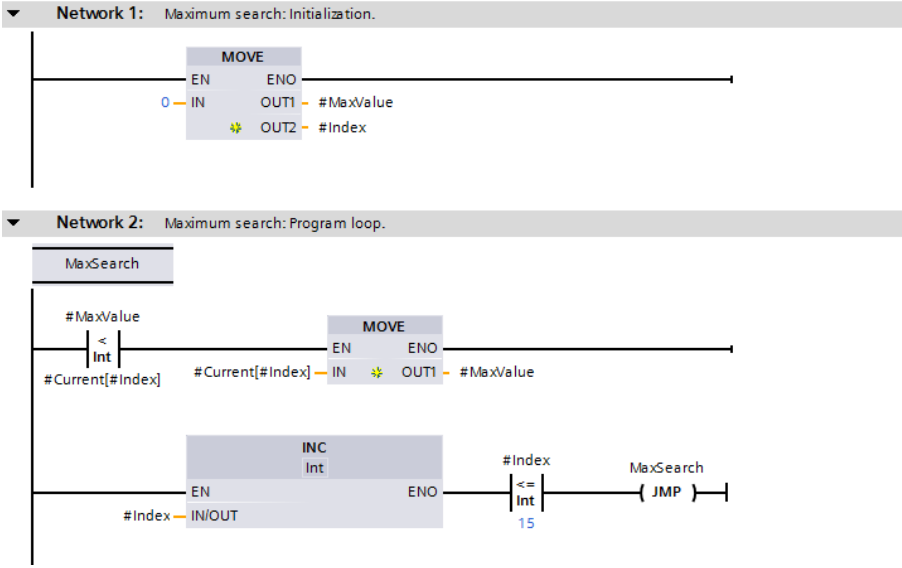


Fig. 7.40 Example of a conditional jump

overwritten with the larger value of `#Current[#Index]`. `#Index` is then increased by +1. As long as `#Index` is less than or equal to 15, it jumps to the beginning of the program loop (to the jump destination `MaxSearch`) and the program part runs again.

### 7.6.2 Jump list in the ladder logic

The jump list is represented as a box. It is only processed if the EN input signal state is “1”. The value of parameter K (0 to 99) determines the box output whose jump destination is jumped to. To program the jump list, drag the `JMP_LIST` function from the program elements catalog under *Basic instructions > Program control operations* to the working area.

If in Fig. 7.41 the `#JumpSelection` tag has the value 0, it jumps to the jump label `Adder`; if the value is 1, to jump label `FC_call`; and if the value is 3, to jump label `FB_call`.

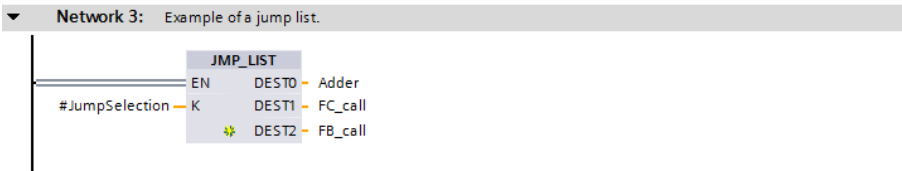


Fig. 7.41 Example of a jump list

### 7.6.3 Jump distributor in the ladder logic

The jump distributor is represented as a box. The box is only processed if the EN input signal state is “1”. The value of parameter K is compared with a value of one of the other input parameters. If the two match, program processing continues at the assigned jump destination. The comparison operations can be selected from a drop-down list. To program a jump distributor, drag the *SWITCH* function from the program elements catalog under *Basic instructions > Program control operations* to the working area.

If in Fig. 7.42 the *#JumpSelection* tag has a value less than 10, it jumps to jump label *FC\_call*; for a value greater than 120 to jump label *FB\_call*; otherwise to jump label *Adder*.

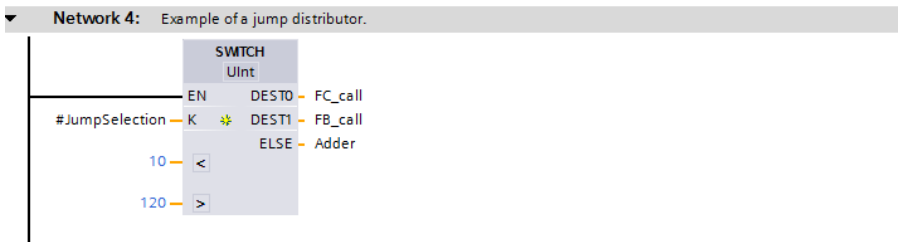


Fig. 7.42 Example of a jump distributor

### 7.6.4 Block end function in the ladder logic

The processing in a block is terminated by the RET coil. The block end function may not be present in a network together with a jump function.

To program a block end function, drag the *RET* function from the program elements catalog under *Basic instructions > Program control operations* to the working area.

In Fig. 7.43, the block is exited if an error occurs when processing the ADD box. The ENO output then has the signal state “0” which is negated, thus triggering the RET coil. The RET coil receives the result of the logic operation “0” (which is output by the terminated block at the ENO output) by means of the FALSE constant. The result can be scanned in the calling block.

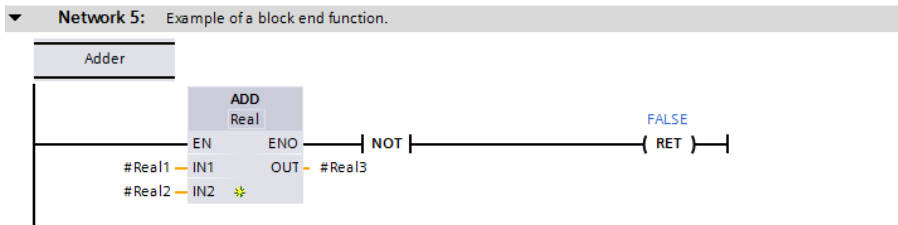


Fig. 7.43 Example of block end function

### 7.6.5 Block call functions in the ladder logic

Calling of blocks is represented by EN/ENO boxes. With functions (FC), the block name is present quasi as a function name in the box; with function blocks, the instance name (the name of the instance data block or of the local instance) is additionally present above the box.

A block call is programmed by opening the *Program blocks* folder in the project tree and dragging the desired block to the working area.

In the example in Fig. 7.44, if the signal state is “1” on the “Input 2” tag, the function “Adder\_2” is called. In the event of an error in the function “Adder\_2” (the ENO output then has signal state “0”), the tag #AddError is also set to signal state “0”. The program processing is then ended in the block and #AddError is specified as return tag. This means that if the processing in the block “Adder\_2” is faulty, the ENO output of the ended block is set to signal state “0”.

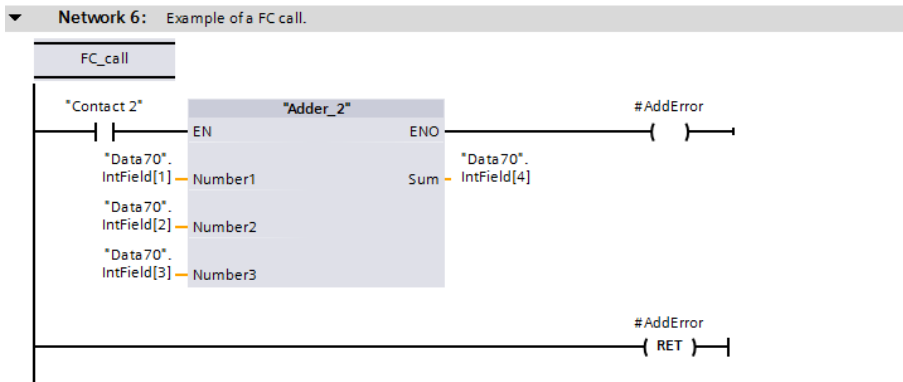


Fig. 7.44 Example of calling a function (FC)

In the example in Fig. 7.45, the “Totalizer” function block is called. Its instance data is present in the data block “Totalizer\_DB”.

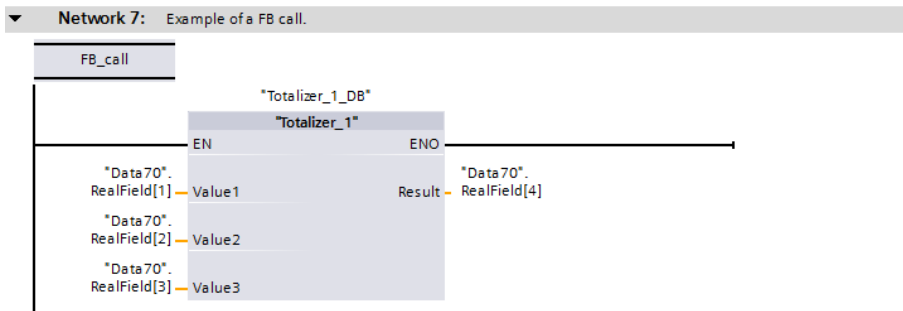


Fig. 7.45 Example of calling a function block

## 8 Function block diagram FBD

### 8.1 Introduction

This chapter describes programming with function block diagram (FBD); it uses examples to show how the program functions are represented in FBD. You can find a description of the individual functions, e.g. comparison functions, in Chapter 10 “Basic functions” on page 328.

Use of the program and symbol editor, which generally applies to all programming languages is described in Chapter 6 “Program editor” on page 178.

FBD is used to program the contents of blocks (the user program). What blocks are, and how they are created, is described in Chapters 5.3 “Programming blocks” on page 125 and 6.3 “Programming a code block” on page 183.

#### 8.1.1 Programming with function block diagram in general

You use FBD to program the control function of the programmable controller – the user program or control program. The user program is organized in different types of blocks. A block is divided into sections referred to as “networks”. Each network contains at least one logic operation, which can also have an extremely complex structure. Each network is terminated by at least one box.

Fig. 8.1 shows the structure of a block with the FBD program. Located at the beginning of the program is the block title, comprising the block heading and block comment. Heading and comment are optional. These are followed by the first network with its number, heading, and comment. Heading and comment are also optional for the networks. The first network shows a logic operation as example with AND and OR boxes, a memory function within the logic operation, and two assignments as termination of the logic operation. The second network shows the processing of EN/ENO boxes. Three of these are arranged in series, one box is on its own in the network. A block is not terminated by a special network or function, you simply finish the program input.

The program editor constructs an FBD network from left to right: position the first program element underneath the network comment, and insert further program elements at its output. The boxes with binary logic operations can be provided with further inputs. Box outputs cannot be directly connected to each other.

A logic operation must always be terminated, for example by an assignment. The assignment controls a binary tag using the result of the logic operation.

“Open” parallel branches can lead out from the top logic operation and not be “wired back” to the top logic operation; these are known as “T branches”. In these

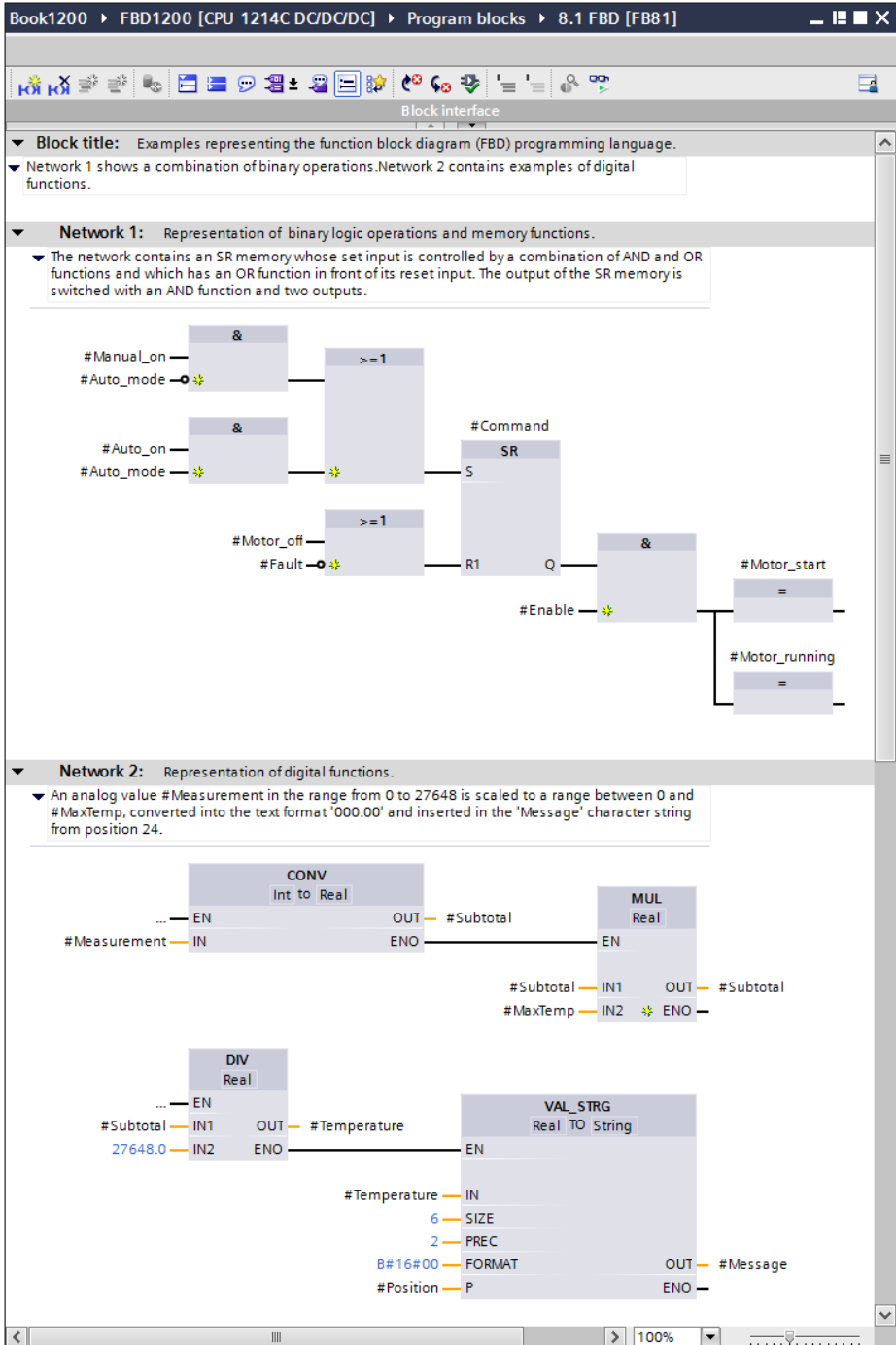


Fig. 8.1 Structure of a block with FBD program



T branches, there are certain limitations with regard to which permissible program elements can be selected.

Where additional rules apply to the arrangement of special FBD elements, these are described in the corresponding sections.

### 8.1.2 Program elements of the function block diagram

Fig. 8.2 shows which types of FBD elements exist: boxes with binary logic operations, and standard boxes for processing binary signals, Q boxes for implementing memory, timer and counter functions, and EN/ENO boxes for “complex” functions which, for example, carry out calculations, manipulate strings, or convert numbers into text.

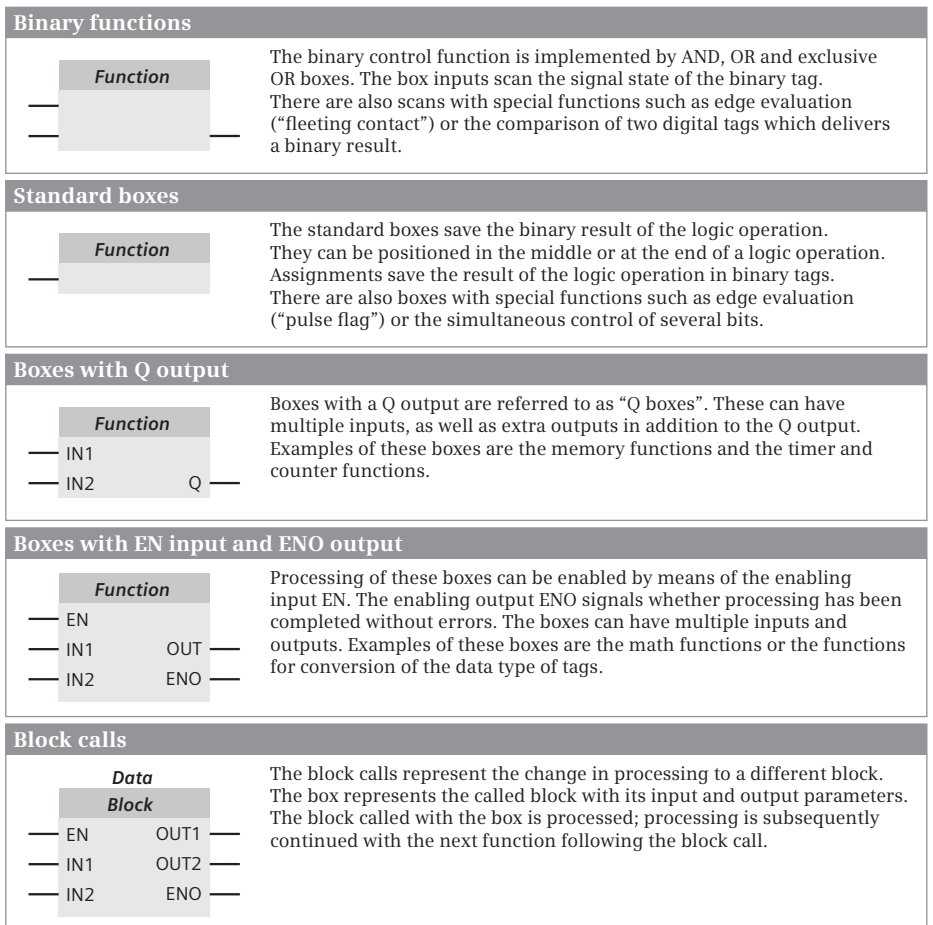
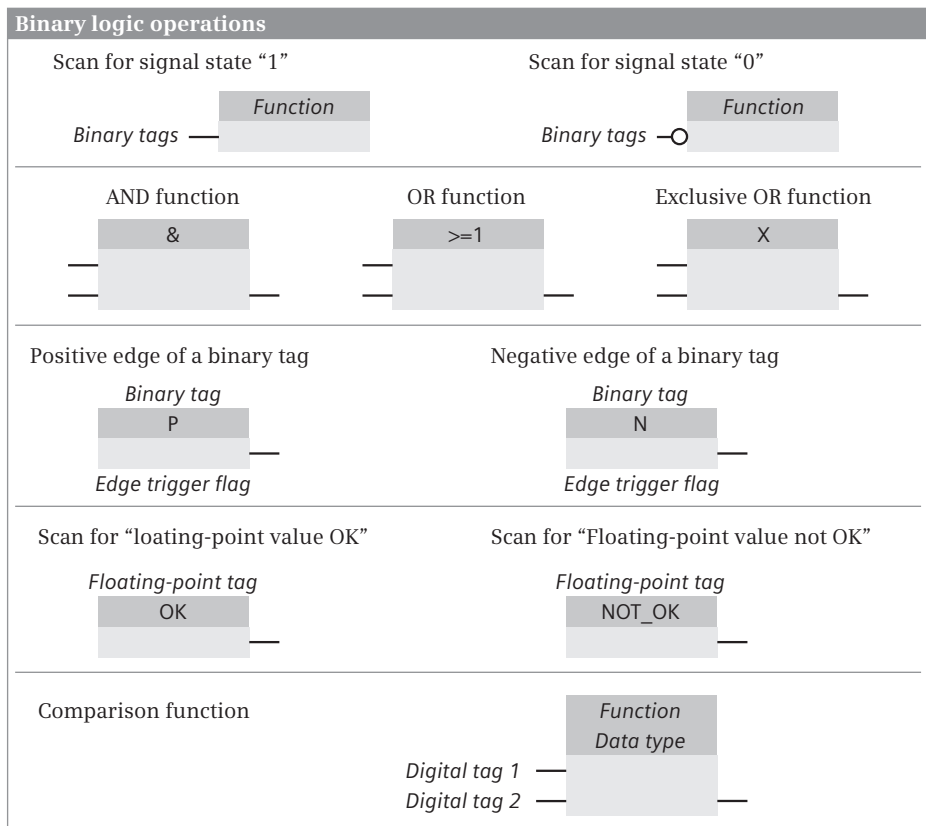


Fig. 8.2 Overview of program elements of the function block diagram

Most program elements must be provided with tags or operand addresses at the box inputs and outputs. It is best if you initially position all program elements in a logic operation and subsequently label them.

## 8.2 Programming of binary logic operations (FBD)

The binary logic operations are carried out in the function block diagram using the AND, OR, and exclusive OR boxes. The binary tags for the logic operation can be scanned for signal state “1” or “0”. The binary results of other boxes can also be included, e.g. the evaluation of a signal edge, the validity checking of floating-point numbers, or the comparison of two digital tags (Fig. 8.3).



**Fig. 8.3** Overview of binary logic operations in the function block diagram

### 8.2.1 Scanning for signal states “1” and “0”

The binary functions scan the binary tags at the function inputs before they link the signal states together. The scan can be made for signal state “1” or “0”. When scanning for signal state “1”, the function input leads directly to the box. You can recognize the scanning for signal state “0” by means of the negation circle at the input of the function.

Scanning for signal state “1” delivers a result “1” if the scanned binary tag has the status “1”, and a result “0” if the status is “0”. Scanning for signal state “0” negates the result, i.e. the result of the scan is now “1” if the status of the scanned binary tag is “0”. The binary functions link the result of the scan, in other words the result which is present “directly” on the box. Using these two possibilities for scanning binary tags, you can handle connected NO and NC contacts in the same functional manner.

Example: Fig. 8.4 shows a sensor connected to the programmable controller which is scanned for signal state “1” (left-hand side) and for signal state “0” (right-hand side). The result of the scan is connected directly to a contactor.

When scanning for signal state “1” (left-hand side), the result of the scan is equal to the signal state of the sensor: if sensor S1 is open, input %I1.0 has the signal state “0” and the logic operation is not fulfilled. Contactor K1 controlled by output %Q4.0

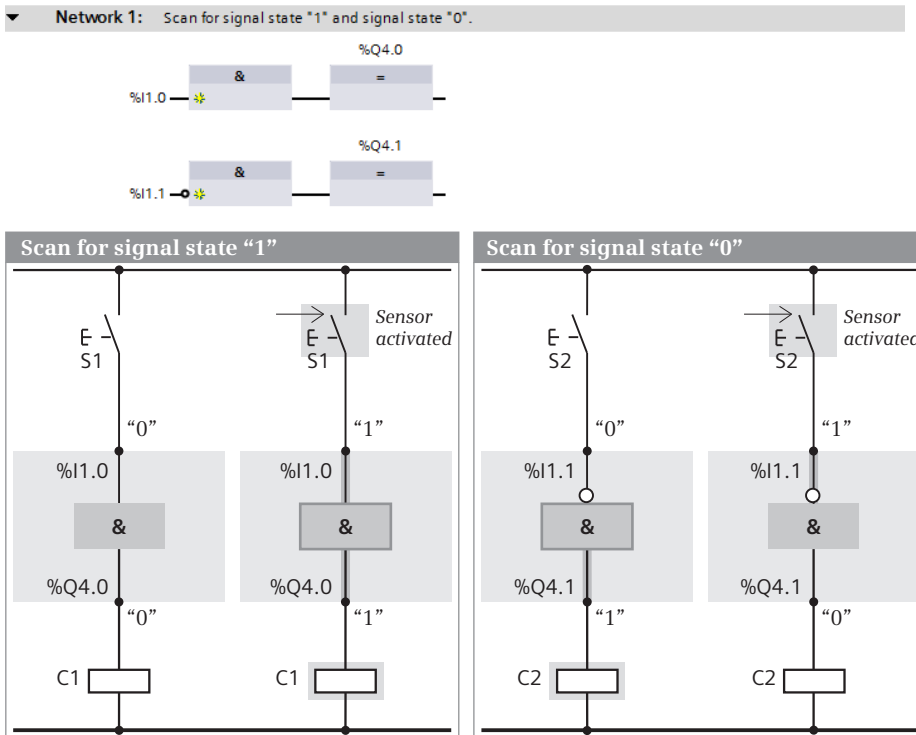


Fig. 8.4 Scanning for signal states “1” and “0”

does not pull up. If sensor S1 is then activated, input %I1.0 has the signal state “1”. The function is fulfilled, and the contactor K1 connected to output %Q4.0 pulls up.

When scanning for signal state “0” (right-hand side), the result of the scan is equal to the negated signal state of the sensor: if sensor S1 is open, input %I1.0 has the signal state “0”. The negation symbol negates this signal state, and the result of the scan is “1”. The logic operation is thus fulfilled. Contactor K1 controlled by output %Q4.0 pulls up. If sensor S1 is then activated, input %I1.0 has the signal state “1”. The negation symbol negates the signal state, and the result of the scan is “0”. The function is not fulfilled, and contactor K1 connected to output %Q4.0 does not pull up.

### 8.2.2 Taking account of the sensor type in the function block diagram

If you scan a sensor in your program, you must take into consideration whether it is an NO or NC contact. Depending on the type of sensor, different signal states are present at the corresponding input when the sensor is activated: “1” with an NO contact and “0” with an NC contact. It is not possible for the control processor to determine whether an NC or NO contact is connected to an input. It can only recognize the signal state “1” or “0”.

If you write the program to obtain “1” when a sensor is activated in order to link it further, you must scan the input in different ways depending on the type of sensor. Scanning for signal states “1” and “0” is available for this purpose. Scanning for signal state “1” delivers “1” if the scanned input is also “1”. Scanning for signal state “0” delivers “1” if the scanned input is “0”. In this manner you can also directly scan inputs which are to execute activities when the signal state is “0” (“zero-active”) and connect the result of the scan further.

The example in Fig. 8.5 shows the programming dependent on the type of sensor. The AND function used is then fulfilled, i.e. the function output has the signal state “1”, if the result of the scan “1” is present at all function inputs (for description, see Chapter 8.2.3 “AND function” on page 252).

In the first case, two NO contacts are connected to the programmable controller, in the second case one NO contact and one NC contact. In both cases, a contactor connected to an output is to pull up when both sensors are activated. When an NO contact is activated, the signal state at the input is “1” and is scanned for signal state “1” so that the AND operation can be fulfilled when the sensor is activated. If both NO contacts are activated, the AND logic operation is fulfilled and the contactor pulls up.

When activating an NC contact, the signal state at the input is “0”. In order to achieve scan result “1” in this case when activating the contact, it is necessary to scan for signal state “0”. Therefore in the second case the NO contact must be scanned for signal state “1” and the NC contact for signal state “0” in order for the contactor to pull up when both sensors are activated.

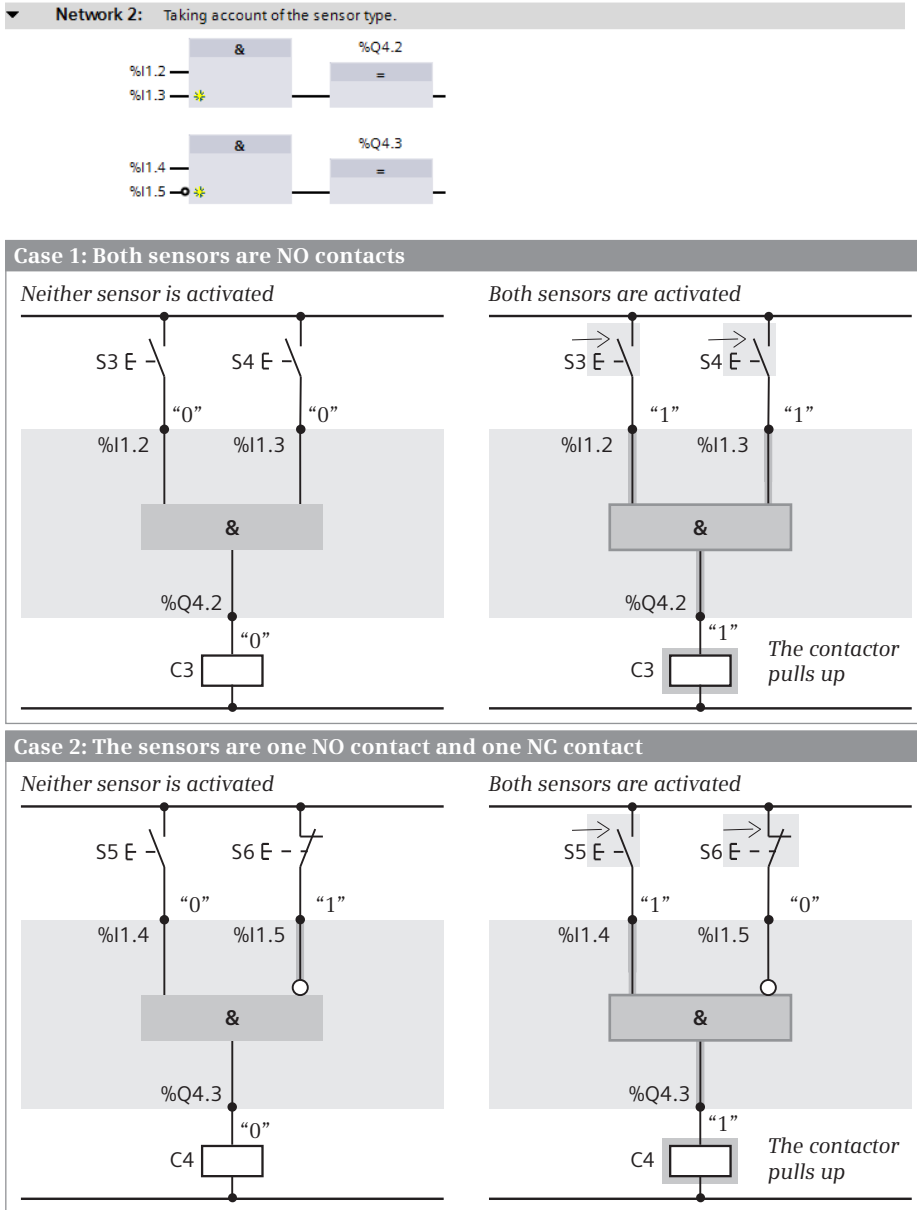


Fig. 8.5 Taking account of the sensor type in the function block diagram

### 8.2.3 AND function

The AND function links two binary signal states together and delivers a result of the logic operation “1” if both states (both results of the scans) are simultaneously “1”. If the AND function has several inputs, the results of the scans of all inputs must

be “1” so that the joint result of the logic operation is “1”. In all other cases, the AND function delivers the result of the logic operation “0” at the function output.

An AND function has two inputs as standard. If you select the AND function when programming and then the *Add input* command in the shortcut menu using the right mouse button, the program editor extends the AND function by a further input. An AND function can have any number of inputs.

Fig. 8.6 shows an example of an AND function with three inputs. The binary tags “Input 1” and “Input 2” lead directly to the function box. The binary tag “Input 3” is scanned for signal state “0”, which can be recognized by the negation circle on the box. The AND function is fulfilled, i.e. the binary tag “Output 1” has signal state “1”, if “Input 1” and “Input 2” have signal state “1” and “Input 3” has signal state “0”. In all other cases, the AND function is not fulfilled.

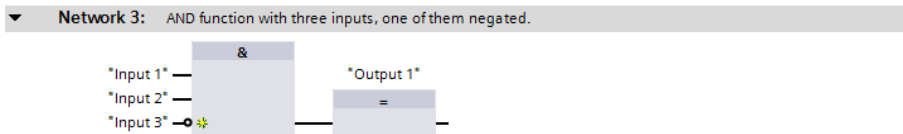


Fig. 8.6 AND function with three inputs

If you wish to directly assign the signal state of one binary tag to another binary tag, e.g. connect an input directly to an output, it is usual to use the AND function with one input.

### 8.2.4 OR function

The OR function links two binary signal states together and delivers a logic operation result “1” if one of the states (one of the results of the scans) is “1”. If the OR function has several inputs, it is sufficient if the result of one input scan is “1” so that the joint result of the logic operation is “1”. If the results of all input scans are “0”, the OR function delivers the result of the logic operation “0” at the function output.

An OR function has two inputs as standard. If you select the OR function when programming and then the *Add input* command in the shortcut menu using the right mouse button, the program editor extends the OR function by a further input. An OR function can have any number of inputs.

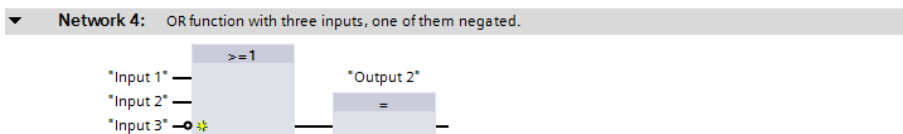


Fig. 8.7 OR function with three inputs

Fig. 8.7 shows an example of an OR function with three inputs. The binary tags “Input 1” and “Input 2” lead directly to the function box. The binary tag “Input 3” is scanned for signal state “0”, which can be recognized by the negation circle on the box. The OR function is not fulfilled, i.e. the binary tag “Output 2” has signal state “0”, if “Input 1” and “Input 2” have signal state “0” and “Input 3” has signal state “1”. In all other cases, the OR function is fulfilled.

### 8.2.5 Exclusive OR function

The exclusive OR function (non-equivalence function) links two binary signal states together and delivers a logic operation result “1” if the two states (both results of the scans) differ. It delivers a result of the logic operation “0” if the two states (both results of the scans) are the same.

You can also program an exclusive OR function with more than two inputs; the exclusive OR function is then fulfilled if an odd number of function inputs delivers scan result “1”.

An exclusive OR function has two inputs as standard. If you select the exclusive OR function when programming and then the *Add input* command in the shortcut menu using the right mouse button, the program editor extends the exclusive OR function by a further input. An exclusive OR function can have any number of inputs.

Fig. 8.8 shows an example of an exclusive OR function with two inputs. The binary tags “Input 1” and “Input 2” lead directly to the function box. The exclusive OR function is fulfilled, i.e. the binary tag “Output 3” has signal state “1”, if “Input 1” and “Input 2” have different signal states. If the signal states of “Input 1” and “Input 2” are the same, the exclusive OR function is not fulfilled.

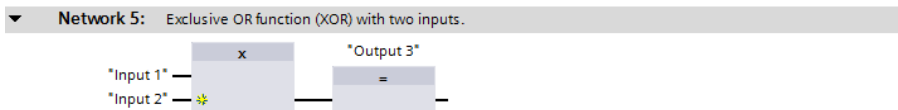


Fig. 8.8 Exclusive OR function with two inputs

### 8.2.6 Mixed binary logic operations

You can combine binary functions with each other, e.g. several AND functions lead to an OR function or two OR functions lead to an exclusive OR function. Examples are shown in Fig. 8.9.

With which signal combination in Fig. 8.9 is the binary tag “Output 4” set to signal state “1”? Firstly, “Input 5” and “Input 6” must have different signal states if the AND function at the end of the logic operation is to be fulfilled. The other condition is that either “Input 1” and “Input 2” both have signal state “1” or that “Input 3” and “Input 4” have different signal states.

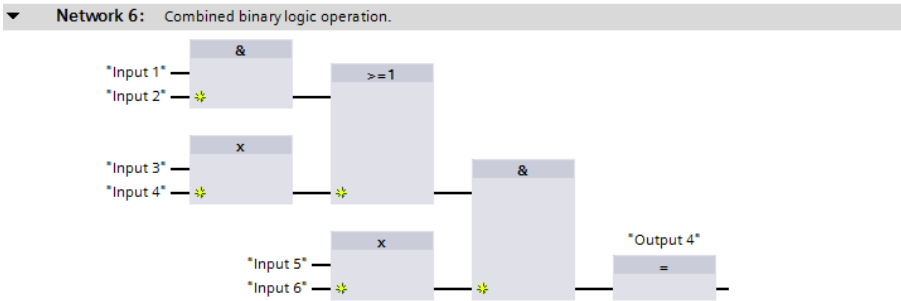


Fig. 8.9 Example of a combined binary logic operation

### 8.2.7 T branch in the function block diagram

Additional possibilities for binary logic operations are provided when a “T branch” is present in an operation. This is a parallel branch (a “parallel logic operation”) which commences in an operation and is terminated by a box.

Fig. 8.10 shows a T branch. It commences with the first AND function and is finished by the assignment to “Output 6”. In the example, “Output 5” then has signal state “1” if “Input 0”, “Input 1”, and “Input 2” have signal state “1”, which corresponds to an AND function with three inputs. “Output 6” has signal state “1” if “Input 0” and “Input 1” both have signal state “1” or “Input 3” has signal state “1”.

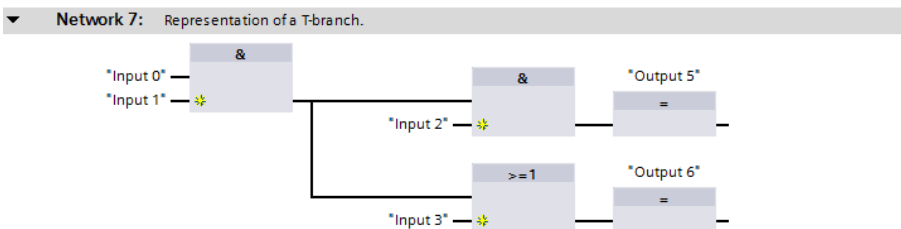


Fig. 8.10 T branch in the function block diagram

### 8.2.8 Negate result of logic operation in the function block diagram

A circle at the input or output of a function symbol negates the result of the logic operation. You can

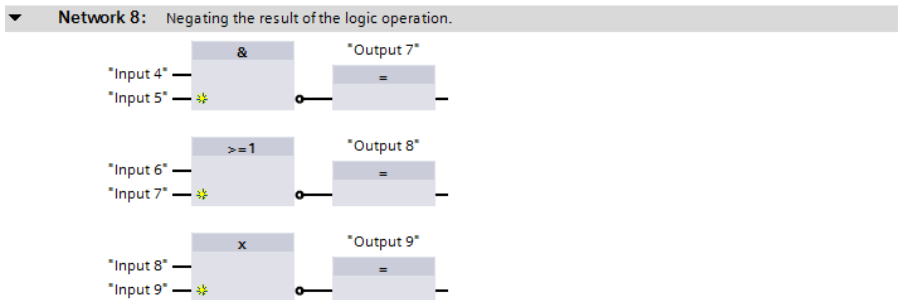
- ▷ apply the negation to the scan of a binary tag; this then corresponds to scanning for signal state “0” (see Section 8.2.1 “Scanning for signal states “1” and “0”” on page 250),
- ▷ set the negation between two binary functions (this corresponds to negation of the result of the logic operation), or



- ▷ position the negation at the output of a binary function (e.g. if you wish to set a binary tag and the logic operation is not fulfilled, i.e. the result of the logic operation = “0”).

Program the negation using the *Invert RLO* program element from the favorites of the program editor or from the program elements catalog under *General*. You can drag the *Invert RLO* element to an input or output of a box using the mouse, or select the input or output and click on *Invert RLO*. If you wish to cancel a negation, drag *Invert RLO* over again, or select the negation and click on *Invert RLO*.

Fig. 8.11 shows three examples of the negation of a function output. A NAND function is an AND function with negated output: the output only has signal state “0” if the scan result “1” is present at all inputs. A NOR function is an OR function with negated output: the output only has signal state “1” if neither of the inputs has the scan result “1”. An exclusive OR function with a negated output is also referred to as an inclusive OR function (equivalence function): The output only has signal state “1” if the scan results are the same for both inputs. If more than two inputs are present, an even number of inputs must have the scan result “1” so that the inclusive OR function is fulfilled.

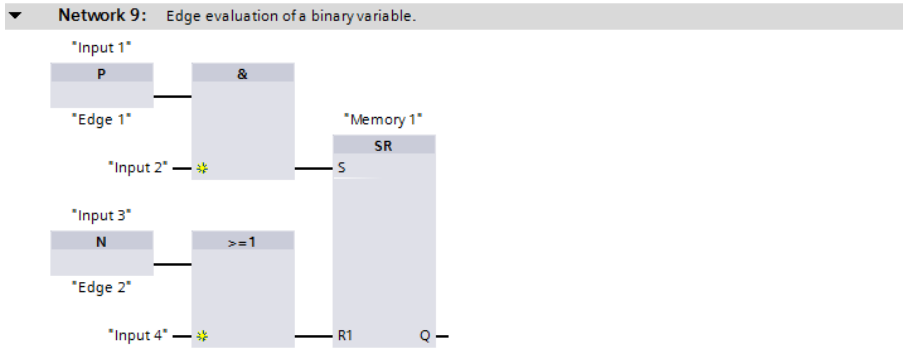


**Fig. 8.11** Negating the result of the logic operation, example of a negated function output

### 8.2.9 Edge evaluation of binary tags in the function block diagram

The edge evaluation of a binary tag has the signal state “1” for one processing cycle if the signal state of the binary tag named above it changes from “0” to “1” (P box, rising edge) or from “1” to “0” (N box, falling edge). It responds like a “fleeting contact”. This “pulse” can be linked further.

The edge trigger flag is named underneath the edge box. This is a flag or data bit which saves the signal state of the binary tag. The signal edge is recognized by comparing the signal states of binary tags and edge trigger flags (see also Chapter 10.3 “Edge evaluation” on page 338).



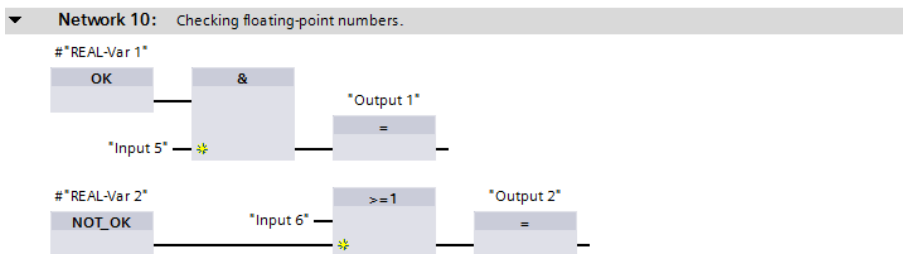
**Fig. 8.12** Edge evaluation of a binary tag

In Fig. 8.12, the binary tag “Memory 1” is set to signal state “1” if “Input 1” changes its signal state from “0” to “1” while “Input 2” has signal state “1”. “Memory 1” is reset again by a change in signal state at “Input 3” from “1” to “0” or by signal state “1” at “Input 4”.

### 8.2.10 Validity checking of floating-point numbers in the function block diagram

The OK box checks a floating-point tag for validity, i.e. whether the range limits for this data type have been observed. The NOT\_OK box is the opposite, it delivers signal state “1” if the floating-point tag is not valid. The OK box and the NOT\_OK box are positioned at the beginning of a logic operation.

The example in Fig. 8.13 shows a validity scan of “REAL-Var 1” which, when ANDed with “Input 5”, controls the binary tag “Output 1”. “Output 2” is set if “Input 6” has signal state “1” or if “REAL-Var 2” is outside the valid range of values.



**Fig. 8.13** Validity checking of floating-point numbers

### 8.2.11 Comparison functions in the function block diagram

A comparison function is represented in the function block diagram as a box with two inputs for digital tags which are compared with each other. A correct comparison delivers a logic operation result “1” at the function output. If the comparison is incorrect, the result of the logic operation is “0”.

Comparison functions are available for equal to, not equal to, greater than, greater than or equal to, less than, and less than or equal to. The comparison is carried out in accordance with the data type of the digital tags involved (for description, see Chapter 11.2 “Comparison functions” on page 364).

The example in Fig. 8.14 shows a comparison between two digital tags with the data type REAL. If the values of the two tags are the same, “Output 3” is set to signal state “1”, otherwise to “0”. “Output 4” has signal state “1” if either the tag “INT-Var 1” is greater than 125 or the tag “INT-Var 2” is smaller than 16 000.

In addition there are range comparisons which check whether the value of a digital tag is within or outside a numerical range.

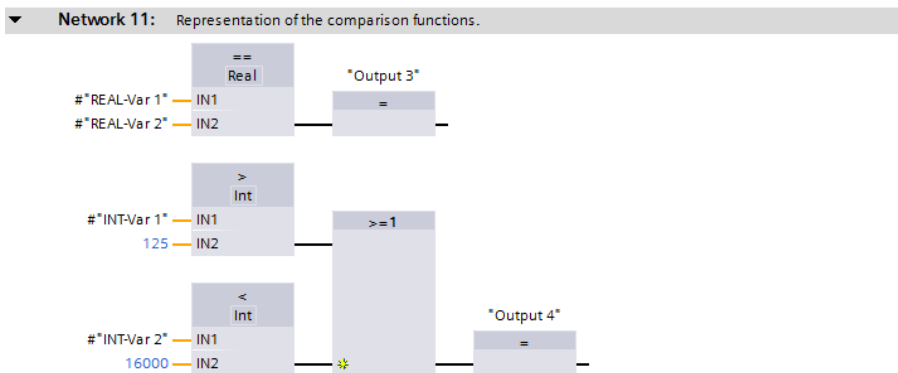


Fig. 8.14 Comparison functions in a binary logic operation

## 8.3 Programming with standard boxes (FBD)

Standard boxes control binary tags such as outputs or bit memories. An assignment box sets the binary tag if signal state “1” is present at the function input, and resets it again with signal state “0”. The reverse is true with the negated assignment box (Fig. 8.15).

Standard boxes exist for setting and resetting a binary tag or for pulse generation during evaluation of signal edges. Standard boxes can also be used to set and reset bit fields, to execute jumps in the program, and to terminate a block.

Standard boxes can be used within a logic operation, following a T branch, or as the termination of an operation. They can be positioned in series or parallel. A standard

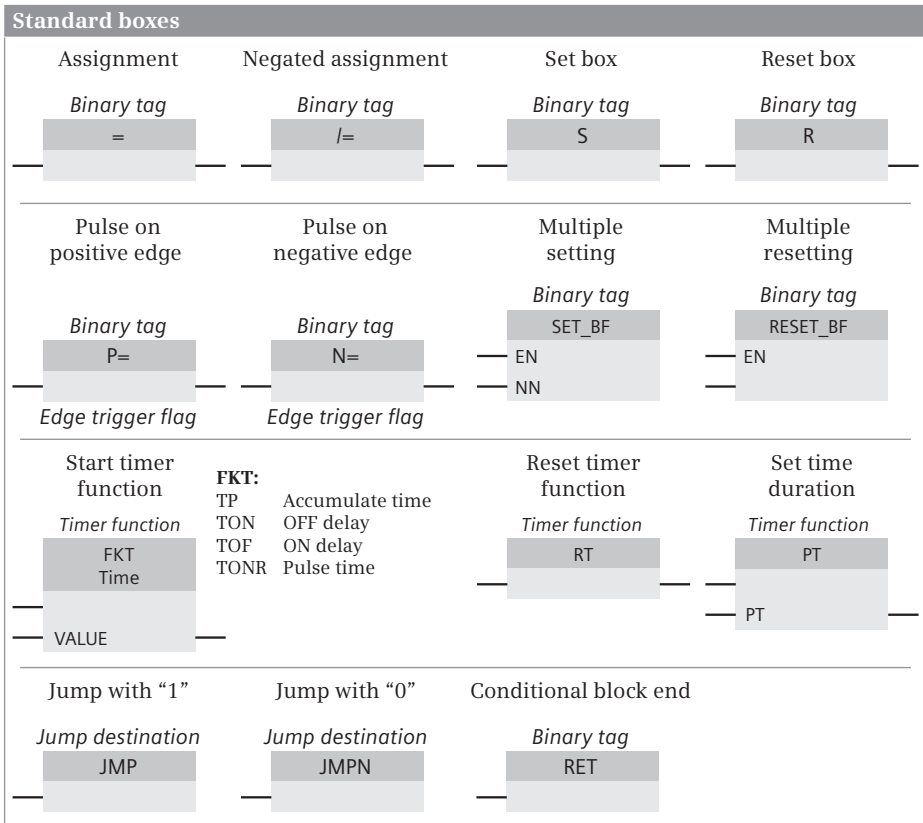


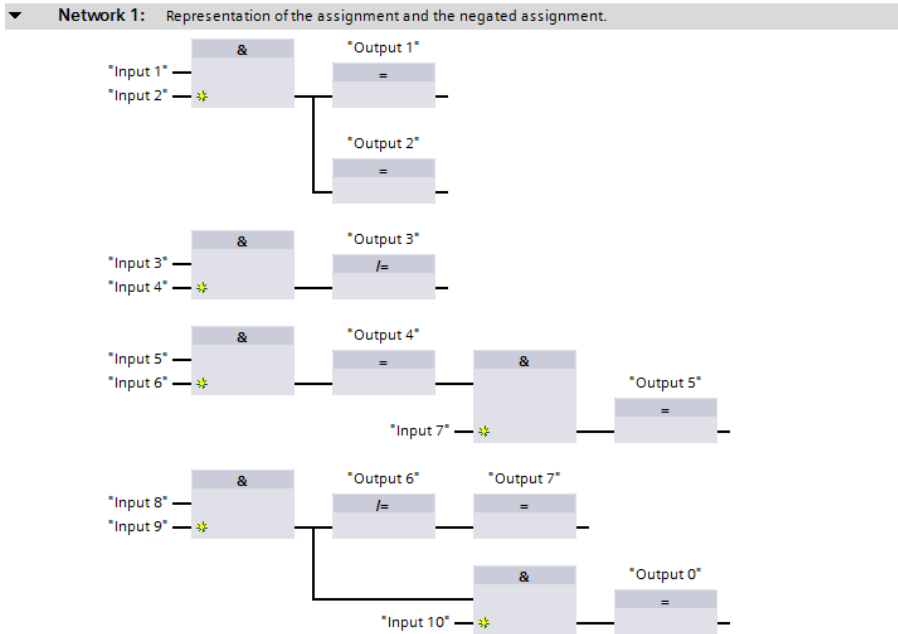
Fig. 8.15 Overview of standard boxes

box standing on its own without a previous logic operation has the signal state "1" at the function input.

### 8.3.1 Assignment and negated assignment

The result of the logic operation is directly assigned to the tag named above the assignment box: with a logic operation result "1", the tag is set, and with a logic operation result "0", it is reset. You can control several assignment boxes simultaneously using the result of the logic operation by arranging the boxes in parallel (Fig. 8.16). All tags named above the coils react in the same manner (first logic operation).

With the negated assignment, the tag named above the box is set if the result of the logic operation is "0"; it is reset if the result of the logic operation is "1" (second logic operation): if the AND function with "Input 3" and "Input 4" is fulfilled, "Output 3" is set to signal state "0".



**Fig. 8.16** Assignment and negated assignment

The third logic operation shows the position of an assignment within an operation. The tag “Output 4 “ is set if “Input 5” and “Input 6” both have signal state “1”. If “Input 7” also has signal state “1”, the tag “Output 5” is set.

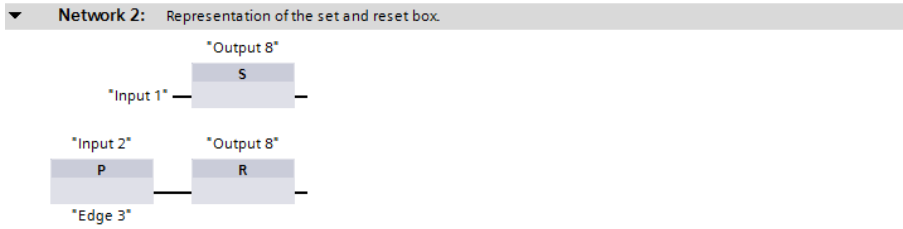
You can position further binary logic operations following a T branch or in front of the assignment (bottom operation). The result of the logic operation is not influenced by an assignment box positioned within an operation. Assignment boxes positioned in series react like parallel ones.

### 8.3.2 Set and reset boxes

With the logic operation result “1” on the set box, the binary tag named above the box is set to the signal state “1”. With the logic operation result “1” on the reset box, the tag named above the box is reset to the signal state “0”. With the logic operation result “0” on the set or reset box, the binary tag remains uninfluenced (Fig. 8.17).

You can position several set and reset boxes in any combination in the same logic operation and also together with assignment boxes.

As with the assignment box, you can position the set and reset boxes within a logic operation or following a T branch. Without a previous logic operation (with function input open), the box is always activated.

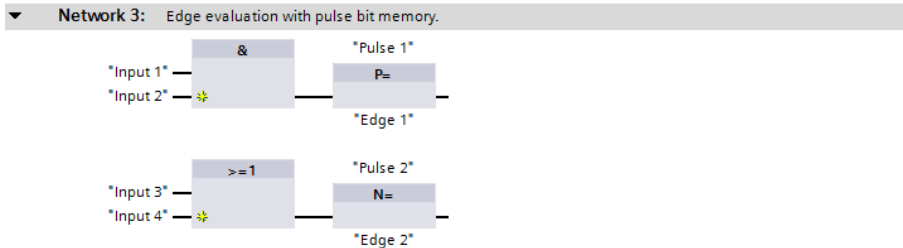


**Fig. 8.17** Set and reset boxes

### 8.3.3 Edge evaluation with pulse output in the function block diagram

The P= and N= boxes are available for edge evaluation with pulse output. The binary tag named above the P= box is set for the duration of one program cycle if the logic operation result of the previous operation changes from signal state “0” to “1” (rising edge).

With the N= box, the binary tag named above the box is set for the duration of one program cycle with a falling edge of the previous operation (change in logic operation result from “1” to “0”) (Fig. 8.18).



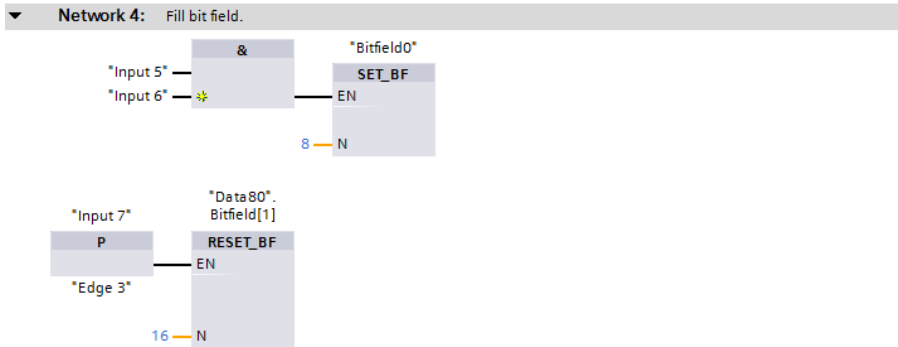
**Fig. 8.18** Edge evaluation of logic operation result (with “pulse bit memory”)

The binary tag named above the box is referred to as a “Pulse bit memory”. Suitable for pulse bit memories are operand types which are not connected “outside” to modules, for example tags from the bit memory or data areas. The edge trigger flag is named under the box, and must be a different tag for each edge evaluation (see Chapter 10.3 “Edge evaluation” on page 338).

The edge boxes can be positioned within a logic operation or terminate an operation. Edge boxes can also be programmed following a T branch.

### 8.3.4 Multiple setting and resetting (filling of bit field) in the function block diagram

With the logic operation result “1” at the EN input, the SET\_BF box sets a bit field to signal state “1”. The bit field is defined by the start tag named above the box and the number of bits at the function input N. With the logic operation result “1”, the RESET\_BF box resets the bits in the bit field.



**Fig. 8.19** Filling of bit field with SET\_BF and RESET\_BF

There is no response if the result of the logic operation at the EN input is “0”. If the SET\_BF and RESET\_BF boxes do not have a previous operation, they are always executed.

In the example in Fig. 8.19, the bit field for the SET\_BF box is defined by the start tag “Bitfield0” which is followed by seven bits (thus a total of eight bits). The bit field for the RESET\_BF box is in the data block “Data80”, commences with the field (binary) component Bitfield[1], and ends after 15 subsequent bits.

### 8.3.5 Starting IEC timer functions in the function block diagram with standard boxes

You can use the timer functions to implement timing processes in the program such as waiting and monitoring times, measurement of a time interval, or the generation of pulses. An IEC timer function can be started with two different program elements: with a standard box or with a Q box (see Chapter 8.4.5 “Controlling IEC timer functions in the function block diagram with Q boxes” on page 267). Both variants are equally useful. A detailed description of the timer functions is provided in Chapter 10.4 “Time functions” on page 344.

A timer function can be started with one of the four behavior patterns TP, TON, TOF, and TONR. A timer function requires internal data for each application. You can specify where this data is to be saved when programming: For the *Single instance* entry in its own data block with the data type IEC\_TIMER and for the *Multi-instance*

entry in the instance data block of the calling function block with a data type that depends on the behavior of the timer function (TP\_TIME, TON\_TIME, TOF\_TIME, TONR\_TIME). You address a timer function with the name of the instance data – data block or local data.

The standard box for starting a timer function requires a preceding logic operation. It can only be placed at the end of a logic operation. Under the standard box is the duration with which the timer function is started.

A timer function is reset using the RT box. The RT box can be programmed in the middle of a logic operation or as its termination.

The PT box sets the duration of a timer function. Each processing with signal state “1” overwrites the duration in the instance data with the value given under the box.

Fig. 8.20 shows the standard boxes used in connection with IEC timer functions. In the first logic operation, the timer function in the local data with the name *#Timer\_function* is started as ON delay with the value *#Duration*. The status of the timer function can be scanned with the structure component *Q*. The example shows starting the timer function, resetting the timer function, and setting the duration with a rising edge.

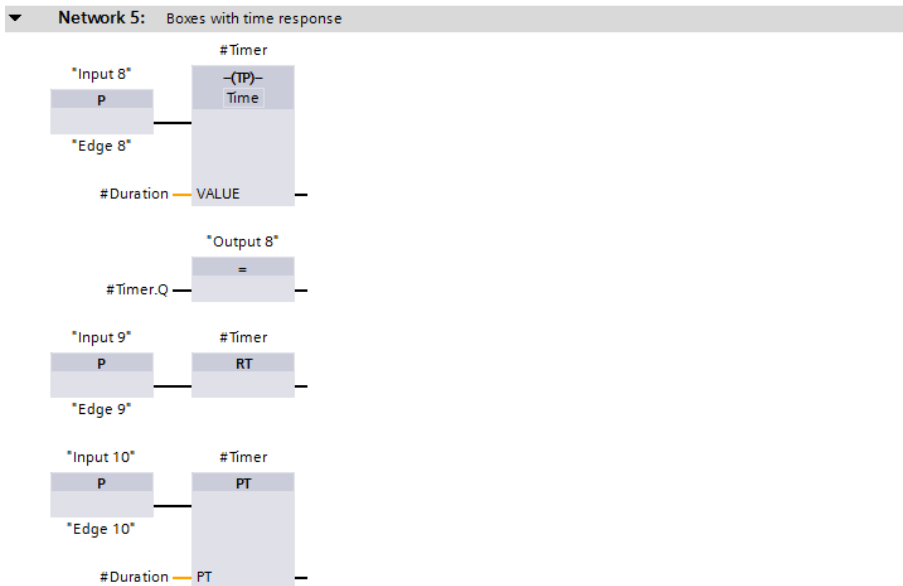


Fig. 8.20 Processing a timer function with standard boxes



## 8.4 Programming with Q boxes (FBD)

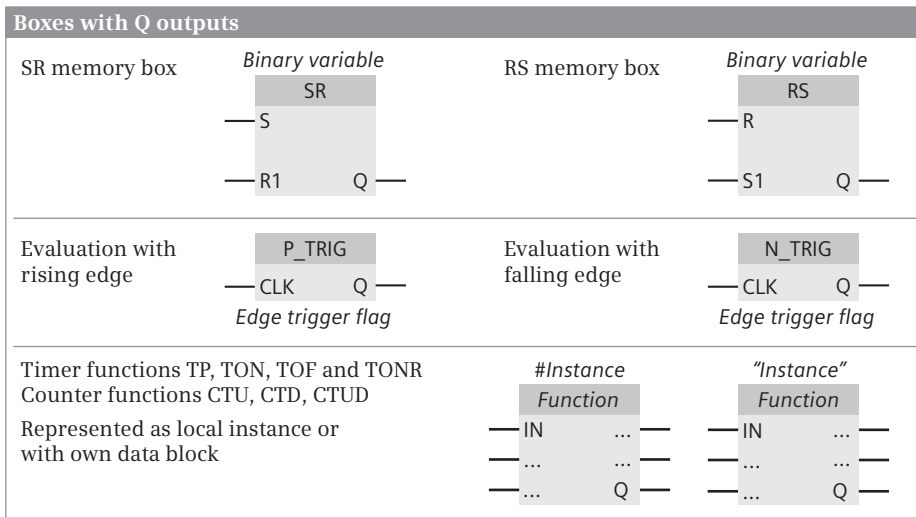
“Q boxes” is the abbreviation for boxes with an output parameter named “Q”. These are the memory boxes SR and RS, the edge evaluations P\_TRIG and N\_TRIG, and the timer and counter functions (Fig. 8.21).

With Q boxes, the first binary input (and in certain cases the associated parameter) must be connected, connection of the other inputs and outputs is optional.

### 8.4.1 Arrangement of Q boxes in the function block diagram

When using Q boxes as program elements, you can:

- ▷ Program one single box per network, either within the logic operation or as its termination
- ▷ Arrange boxes in series by connecting the Q output of one box to a binary input of the following box, and
- ▷ Position boxes following T branches.



**Fig. 8.21** Overview of Q boxes available with FBD

The circuits shown in Fig. 8.21 for the positioning of Q boxes use the memory box with two inputs as an example. This enables the possible positioning of all Q boxes to be shown.

### 8.4.2 Memory boxes in the function block diagram

There are two versions of the memory boxes: as SR box (reset dominant) and as RS box (set dominant). In addition to the difference in the function name, the two boxes also differ in the positioning of the set and reset inputs.

The binary tag named above the memory box is set when the set input has signal state "1" and the reset input has signal state "0". The binary tag is reset when "1" is present at the reset input and "0" at the set input. Signal state "0" at both inputs has no influence on the memory function. If signal state "1" is present simultaneously at both inputs, the two memory functions respond differently: the SR memory function is reset, the RS memory function is set.

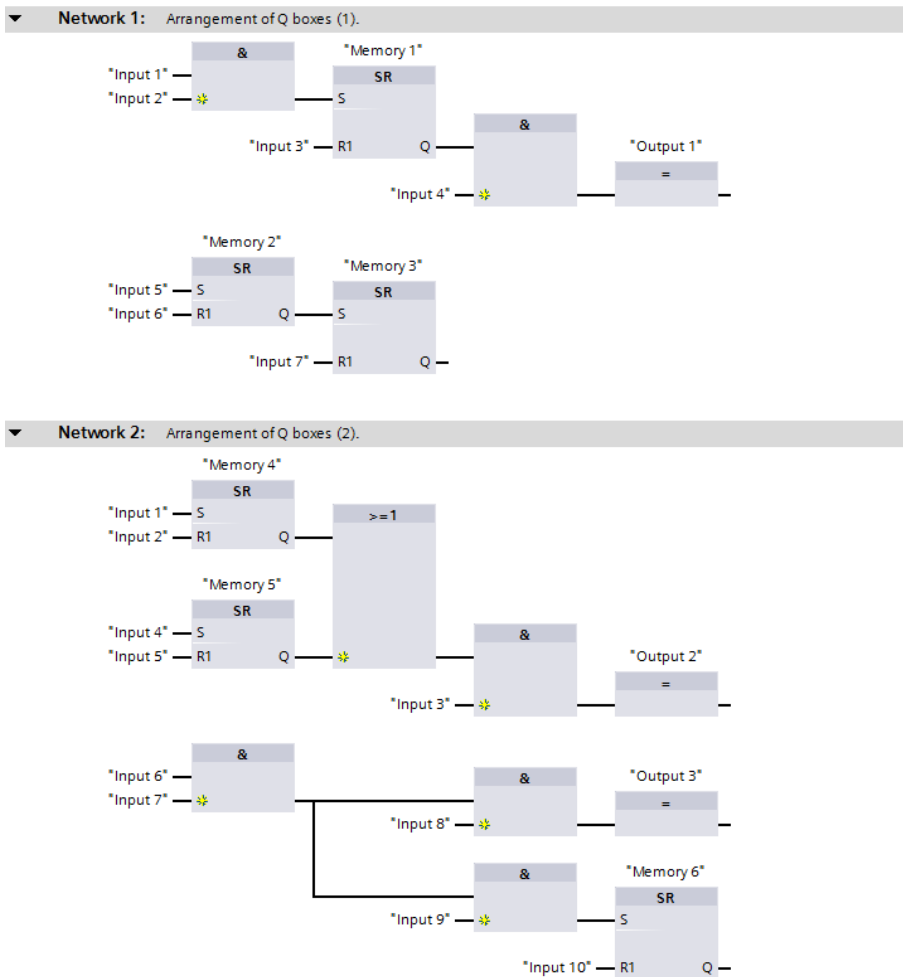


Fig. 8.22 Positioning of Q boxes using example of SR memory function

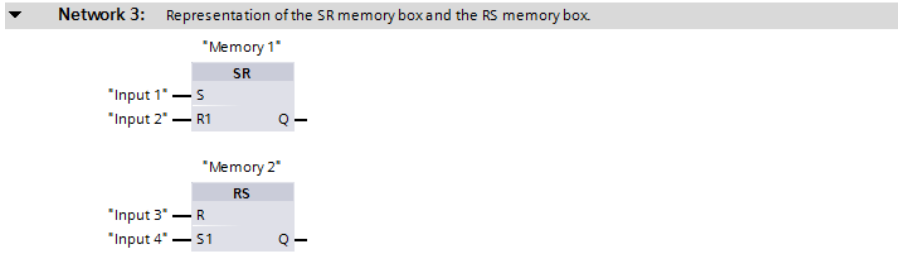


Fig. 8.23 Memory boxes SR and RS

If both tags “Input 1” and “Input 2” in Fig. 8.23 simultaneously have the same signal state “1”, “Memory 1” is reset (function input R1 is dominant). If both “Input 3” and “Input 4” simultaneously have the same signal state “1”, “Memory 2” is set (function input S1 is dominant).

### 8.4.3 Edge evaluation of logic operation result in the function block diagram

The edge evaluation with Q boxes registers a change in the result of the logic operation prior to the box. If the logic operation result changes from “0” to “1” (rising edge) at the CLK input of the P\_TRIG box, signal state “1” is present at the Q output for the duration of one program cycle.

If the result of the logic operation changes from “1” to “0” (falling edge) at the CLK input of the N\_TRIG box, the Q output is activated for the duration of one program cycle.

The P\_TRIG or N\_TRIG-Box must not terminate a logic operation.

Fig. 8.24 shows an example of Q boxes with edge evaluation. The tag “Memory 3” is set at the moment when both “Input 5” and “Input 6” have the signal state “1”. The memory function is reset at the moment when “Input 7” and “Input 0” both have signal state “1”.

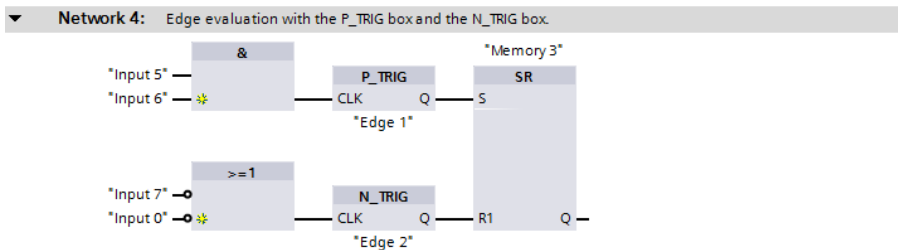
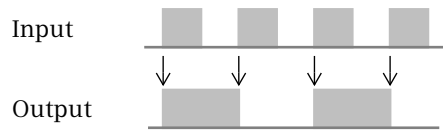


Fig. 8.24 Example of edge evaluation of the logic operation result (with Q boxes)

### 8.4.4 Example of binary scaler in the function block diagram

A binary scaler has one input and one output. If the signal at the input of the binary scaler changes its state, e.g. from “0” to “1”, the output also changes its signal state. This (new) signal state is then retained until the next change, which is positive in our example. Only then does the signal state of the output change again. Half of the input frequency is then present at the binary scaler's output.



A solution for this task is shown in Fig. 8.25. If the tag “Input 1” has signal state “1”, the tag “Output 1” is set (“Memory 1” is still reset). If the signal state at “Input 1” changes to “0”, “Memory 1” is also set (“Output 1” is then “1”). If “Input 1” is “1” the next time around, “Output 1” is reset again (“Memory 1” is now “1”). If “Input 1” is “0” again, “Memory 1” is reset (since “Output 1” is now also reset). The “basic state” has now been reached again following two input pulses and one output pulse.

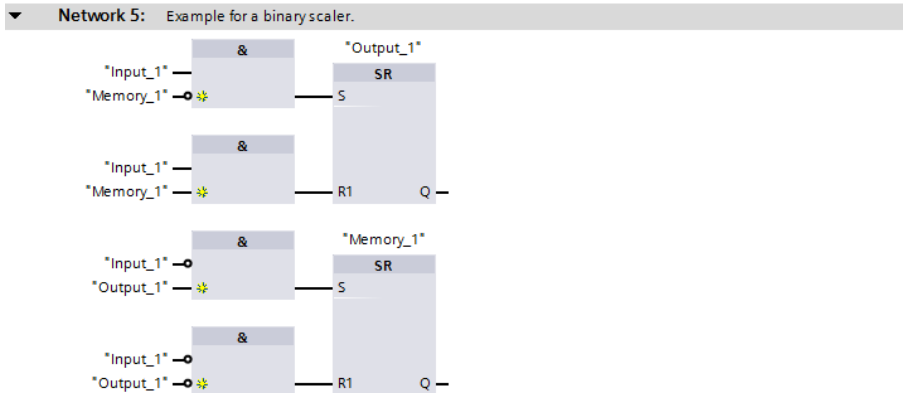


Fig. 8.25 Example of binary scaler in function block diagram

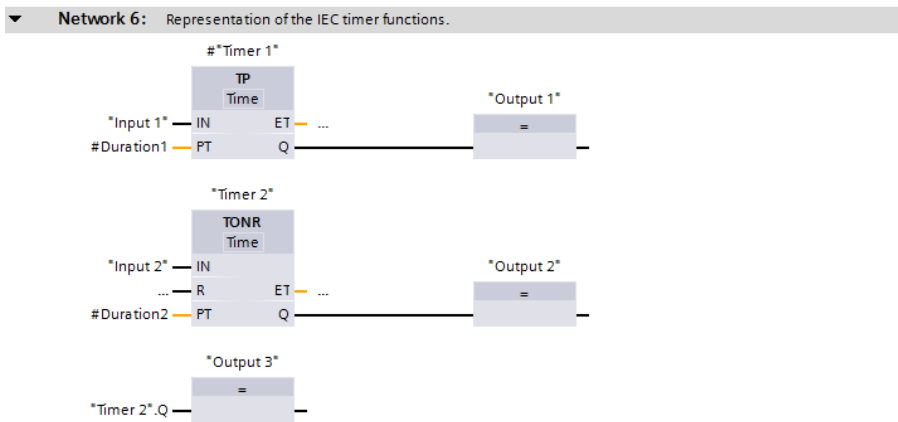
### 8.4.5 Controlling IEC timer functions in the function block diagram with Q boxes

You can use the timer functions to implement timing processes in the program such as waiting and monitoring times, measurement of a time interval, or the generation of pulses. An IEC timer function can be started with two different program elements: With a standard box (see Chapter 8.3.5 “Starting IEC timer functions in the function block diagram with standard boxes” on page 262) or with a Q box. Both variants are equally useful. A detailed description of the timer functions is provided in Chapter 10.4 “Time functions” on page 344.

A timer function can be started with one of the four behavior patterns TP, TON, TOF, and TONR. A timer function requires internal data for each application. You can

specify where this data is to be saved when programming: For the *Single instance* entry in its own data block with the data type IEC\_TIMER and for the *Multi-instance* entry in the instance data block of the calling function block with a data type that depends on the behavior of the timer function (TP\_TIME, TON\_TIME, TOF\_TIME, TONR\_TIME). You address a timer function with the name of the instance data – data block or local data.

The top timer function “Timer 1” in Fig. 8.26 saves its data as a local instance with the name #“Timer 1” in the calling function block. This is started with “Input 1” and “Duration 1”. The tag “Output 1” has signal state “1” for as long as defined by the tag “Duration 1”.



**Fig. 8.26** Examples of timer functions

The bottom timer function “Timer 2” saves its data in a separate data block “Timer 2”. This is started with “Input 2” and “Duration 2”. After the time has elapsed, the tag “Output 2” has signal state “1”.

The name of the local instance (#“Timer 1”) and the name of the data block (“Timer 2”) address the respective timer functions. Component Q of the data structure provides the status of the timer function and can also be scanned at other points in the user program.

#### 8.4.6 IEC counter functions in the function block diagram

You can use the IEC counter functions to execute counting tasks directly using the control processor. The counter functions can count up and down; the numerical range depends on the data type of the preset value. The data types USINT, UINT, UDINT, SINT, INT and DINT are available.

The counting frequency of the counter functions depends on the execution time of the user program. In order to count, the CPU must recognize a change in the signal state of the input pulse, i.e. the input pulse and the pause must be present for at

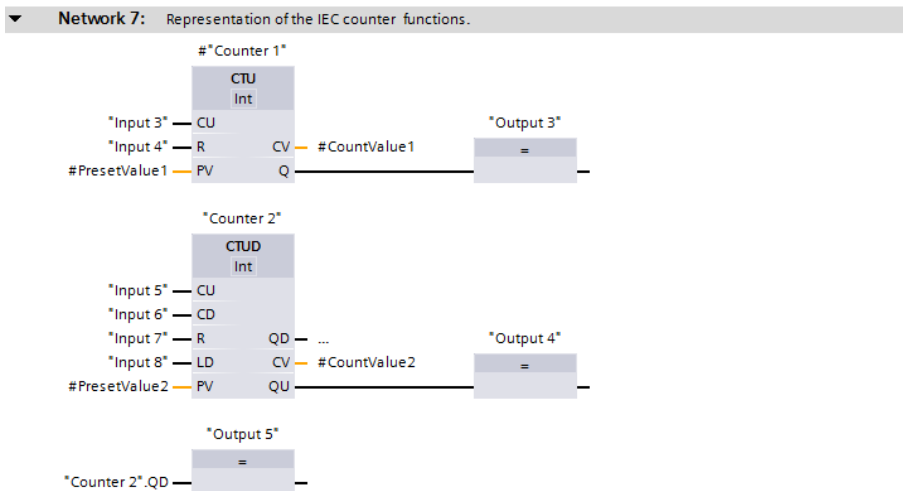
least one program cycle. The longer the program execution time, the lower the counting frequency. A detailed description of the counter functions is provided in Chapter 10.5 “Counter functions” on page 349.

A counter function can be controlled with one of the three behavior patterns CTU, CTD, and CTUD. A counter function requires internal data for each application. You can specify where this data is to be saved when programming: by specifying *Single instance* for storage in a separate data block, and by specifying *Multi-instance* for storage in the instance data block of the calling function block.

The data type of a counter function is based on the data type of the count value. If, for example, an up-counter (CTU) with a DINT count value is programmed as a single instance, the data type IEC\_DCOUNTER is taken as a basis for the data block (see Chapter 4.8.2 “IEC\_COUNTER system data type” on page 112); as a local instance, the counter function has the data type CTU\_DINT (see Chapter 4.6.2 “Parameter types for IEC counter functions” on page 108). You address the counter function with the name of the instance data – data block or local data.

The top counter function “Counter 1” in Fig. 8.27 saves its data as a local instance with the name #“Counter 1” in the calling function block. The actual count value “Count value 1” is set by “Input 4” to zero. “Input 3” increments the actual count value by one unit with each positive edge. If the count value reaches the default value “Preset value 1” and then exceeds it, the tag “Output 4” at output Q is set.

The second counter in the example is an up/down counter. “Input 7” sets the actual count value to zero, “Input 8” loads the default value “Preset value 2” as the actual count value. “Input 5” increments the count value by one unit with each positive change in signal, “Input 6” decrements the count value by one unit with each positive change in signal.



**Fig. 8.27** Examples of counter functions

The QU output has the signal state “1” if the actual count value at the CV output is equal to or greater than the default value at the PV input. The QD output has the signal state “1” if the actual count value is zero or less than zero.

The lower QU output can be further connected directly. In the example, it is used to control the tag “Output 5”. The QD output cannot be supplied, but can be scanned indirectly via the corresponding component QD of the counter structure. (For the QU output, this would be the component QU.)

The name of the local instance (“Counter 1”) and the name of the data block (“Counter 2”) address the respective counter function. In the example, “Counter 2” has its own data block, and the QD output is scanned with the name “Counter 2”.QD. The result of the query can be connected further, e.g. to control an assignment.

## 8.5 Programming with EN/ENO boxes (FBD)

EN/ENO boxes have an enabling input EN and an enabling output ENO. The enabling input can be used to suppress processing of the box. If an error occurs while the box is being processed, this is displayed at the enabling output.

A detailed description of the functions with EN/ENO boxes can be found in the corresponding sections. Programming of the EN/ENO boxes in the function block diagram is the primary focus here. Fig. 8.28 provides an overview of the functions implemented with EN/ENO boxes.

The parameters of the EN/ENO boxes must all be connected. The enabling input EN and the enabling output ENO are not parameters of the box function. They are used for processing boxes, and are added by the program editor to the box function.

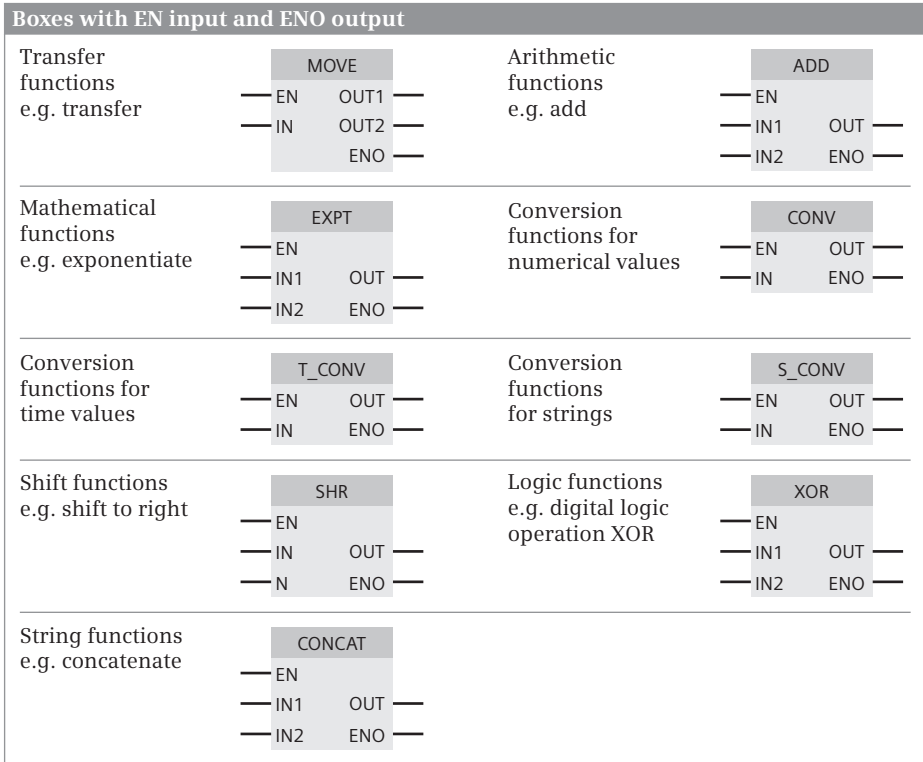
A detailed description of EN and ENO and how the EN/ENO mechanism can be used with self-created blocks can be found in Chapter 12.4.1 “EN/ENO mechanism with LAD and FBD” on page 418.

The block calls in the function block diagram which are also shown as EN/ENO boxes are described in Chapter 8.6.5 “Block call functions in the function block diagram” on page 282.

### 8.5.1 Positioning of EN/ENO boxes in the function block diagram

Fig. 8.29 uses the MOVE function to show the positioning of EN/ENO boxes in a logic operation. An EN/ENO box can be positioned on its own in a network, with or without connection of the EN input or the ENO output.

The ENO output can be connected to the EN input of the following box. A binary tag at the EN input of the first box can be used to switch processing of the complete box series on and off. The ENO output of the last box indicates by means of signal state “1” that the complete sequence has been executed without errors.



**Fig. 8.28** Overview of boxes with enable input EN and enable output ENO

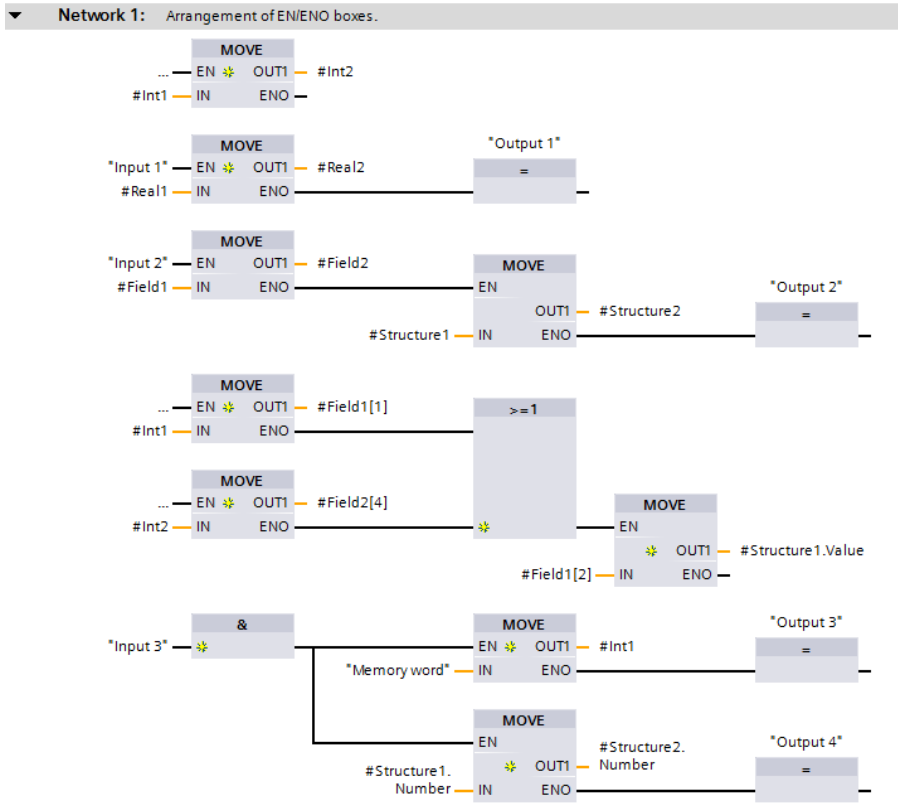
The ENO output of one box can be connected to the ENO output of another box. If an EN/ENO box is positioned in a T branch, its ENO output can no longer be returned to the logic operation at which the T tap commences.

### 8.5.2 Transfer functions in the function block diagram

Boxes with the following transfer functions are available in the programming language FBD:

- ▷ Copy an operand or tag (MOVE)
- ▷ Read a field component with variable index (FieldRead)
- ▷ Write a field component with variable index (FieldWrite)
- ▷ Copy a data area (MOVE\_BLK)
- ▷ Copy a data area without interruption (UMOVE\_BLK)
- ▷ Fill a data area (FILL\_BLK)
- ▷ Fill a data area without interruption (UFILL\_BLK)
- ▷ Exchange the bytes within a tag (SWAP).





**Fig. 8.29** Positioning of EN/ENO boxes in the function block diagram using example of MOVE box

A detailed description of the transfer functions is provided in Chapter 11.1 “Transfer functions” on page 356.

Fig. 8.30 shows an example of the programming of transfer functions. If the tag “Input 1” changes the signal state from “0” to “1”, the MOVE and FILL\_BLK boxes will be executed.

The MOVE box copies the content of tag #Int1 to tag #Field1[1]. #Field1[1] is a component of the tag #Field1 with the same data type as the tag #Int1.

The MOVE box can be provided with further outputs: select the MOVE box and then the *Insert output* command from the shortcut menu. In the example, the content of #Int1 is also transferred to the tag #Field1[9] and to the component #Structure1.Int1.

The FILL\_BLK box transfers the value 1234.567 to four successive components of the field tag #Field3, starting with #Field3[1]. The tag #Field3 consists of components with data type REAL.

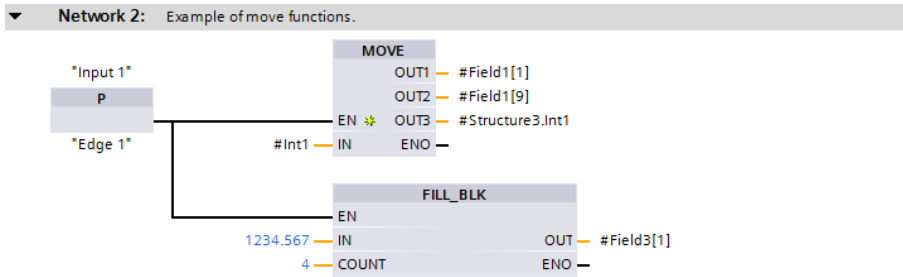


Fig. 8.30 Example of the transfer functions in the function block diagram

### 8.5.3 Arithmetic functions for numerical values in the function block diagram

Boxes with the following arithmetic functions for numerical values are available in the programming language FBD:

- ▷ Add (ADD), subtract (SUB), multiply (MUL) and divide (DIV) two numerical values
- ▷ Divide with remainder as result (MOD)
- ▷ Form absolute value (ABS), negation (NEG, multiplication by  $-1$ ), decrement (DEC, reduce numerical value by 1) and increment (INC, increase numerical value by 1).

A detailed description of these arithmetic functions is provided in Chapter 11.3 “Arithmetic functions for numerical values” on page 366.

Fig. 8.31 shows an example of the arithmetic functions with numerical values. Two tags of data type INT are added, and the intermediate result is saved in the temporary tag #t\_Int1. This intermediate result is multiplied by  $-1$  and output to tag #Int3. The absolute value of tag #Int1 is generated; this value is divided by 3 and the remainder of the division written to tag #Int4.

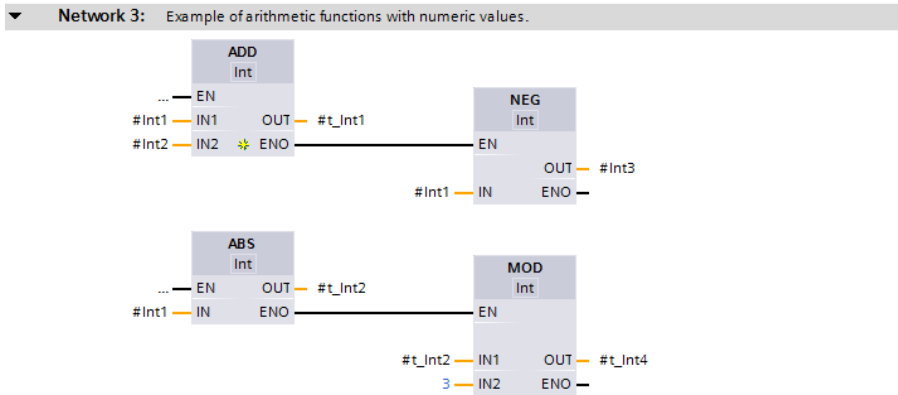
### 8.5.4 Arithmetic functions with time values in the function block diagram

In the FBD programming language, durations (time spans, data type TIME) and points in time (date and time, data type DTL) can be connected together. Boxes with the following arithmetic functions are available for this:

Boxes with the following arithmetic functions are available for this:

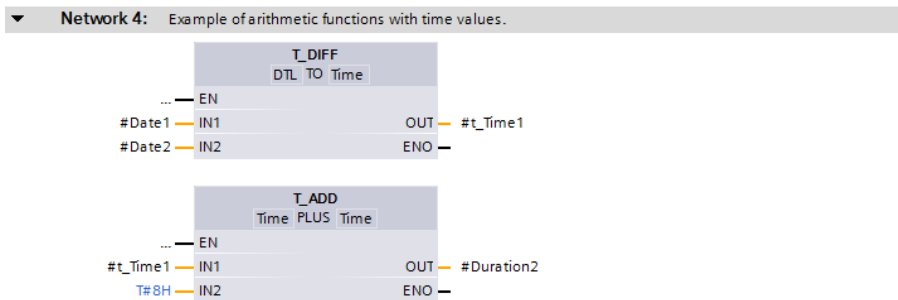
- ▷ Add two durations, or add a duration to a point in time (T\_ADD)
- ▷ Subtract two durations, or subtract one duration from a point in time (T\_SUB)
- ▷ Generate the difference between two points in time (T\_DIFF).

A detailed description of these arithmetic functions is provided in Chapter 11.4 “Arithmetic functions for time values” on page 369.



**Fig. 8.31** Example of arithmetic functions for numerical values in the function block diagram

Fig. 8.32 shows an example of arithmetic functions with time values. The difference between the tags #Date1 and #Date2 is generated. The result is a duration in TIME format. Eight hours are added to this duration, and the result output to tag #Duration2.



**Fig. 8.32** Example of arithmetic functions with time values in the function block diagram

### 8.5.5 Math functions in the function block diagram

Boxes with the following math functions are available in the programming language FBD:

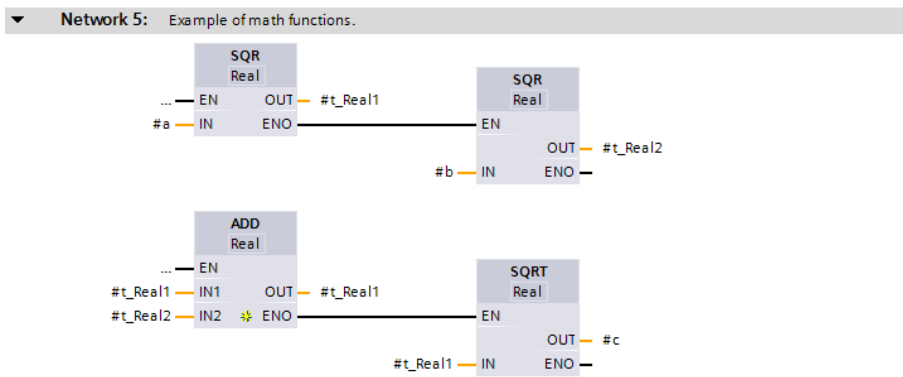
- ▷ Trigonometric functions: sine (SIN), cosine (COS), tangent (TAN)
- ▷ Arc functions: arcsine (ASIN), arccosine (ACOS), arctangent (ATAN)
- ▷ Form square (SQR) and square root (SQRT)
- ▷ Exponential function to base e (EXP)
- ▷ Exponential function to any base (EXPT)

- ▷ Natural logarithm (LN)
- ▷ Extract decimal places (FRAC)

A detailed description of these math functions is provided in Chapter 11.5 “Mathematical functions” on page 372.

Fig. 8.33 shows an example of the math functions.

The tag #c is calculated according to the  $c = \sqrt{a^2 + b^2}$  equation. The square of #a is formed first. When inputting tag names – which can also be keywords (in the input, “a” can also stand for “output”) or which can have the same name both locally and globally – the tag must be labeled accordingly: for a local tag with a preceding number sign (#), for a global tag with the name in quotation marks, and for an operand (absolute address) with a preceding percent sign (%).



**Fig. 8.33** Example of math functions in the function block diagram

In the example, the squares of #a and #b are stored temporarily and added. #t\_Real1 is used again for the buffer. The result of the square root extraction is saved in the tag #c.

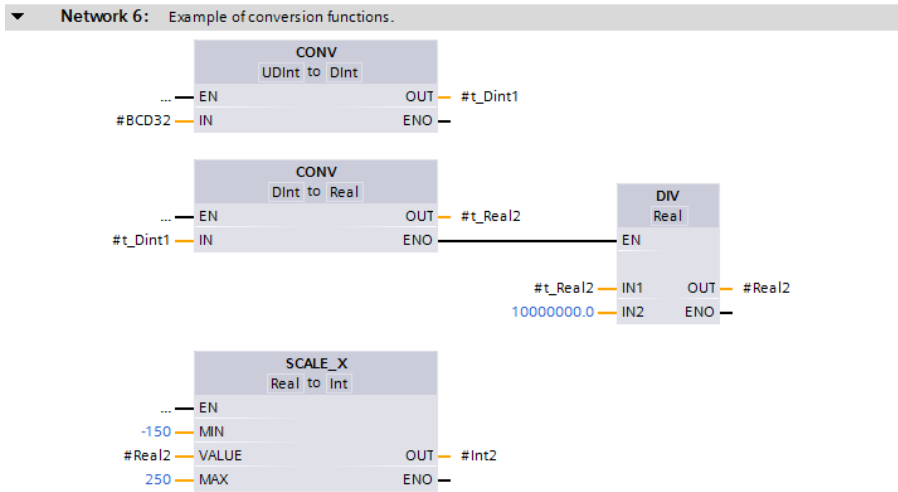
### 8.5.6 Conversion functions in the function block diagram

Boxes with the following conversion functions are available in the programming language FBD:

- ▷ CONV (conversion of BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, REAL, LREAL, BCD16, BCD32)
- ▷ ROUND, FLOOR, CEIL, TRUNC (conversion of REAL, LREAL into SINT, INT, DINT, USINT, UINT, UDINT, REAL, LREAL)
- ▷ SCALE\_X, NORM\_X (scaling and standardization)
- ▷ T\_CONV (conversion of TIME into DINT and vice versa)
- ▷ S\_CONV, STRG\_VAL, VAL\_STRG (conversion of SINT, INT, DINT, USINT, UINT, UDINT, REAL into STRING and vice versa)

A detailed description of the conversion functions is provided in Chapter 11.6 “Conversion functions (Conversion of data type)” on page 376.

Fig. 8.34 shows an example of the conversion functions.



**Fig. 8.34** Example of conversion functions in the function block diagram

The conversion function CONV is used to convert a 7-digit BCD number into a DINT format number and subsequently into REAL format (tag #Real2). The value of #Real2 is divided by  $10^7$  and converted into a fixed-point number between the limits of  $-150$  and  $+250$ .

### 8.5.7 Shift functions in the function block diagram

Boxes with the following shift functions are available in the programming language FBD:

- ▷ Shift to right (SHR) and left (SHL)
- ▷ Rotate to right (ROR) and left (ROL)

A detailed description of the shift functions is provided in Chapter 11.7 “Shift functions” on page 389.



**Fig. 8.35** Example of shift functions in the function block diagram

Fig. 8.35 shows an example of the shift functions. The content of tag #Int2 is shifted three places to the right and output to tag #Int3. Shifting of fixed-point numbers one place to the right is equivalent to a division by two. In the example, tag #Int2 is divided by eight ( $2^3$ ) and the rounded-off result output to tag #Int3.

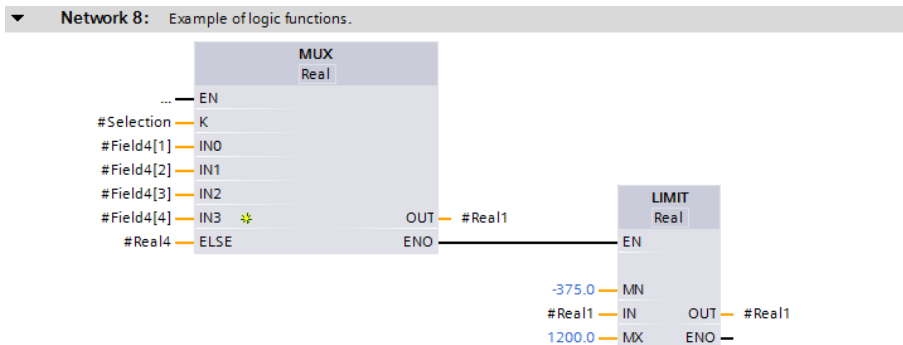
### 8.5.8 Logic functions in the function block diagram

Boxes with the following logic functions are available in the programming language FBD:

- ▷ Digital logic operations AND, OR and XOR
- ▷ Invert (INV)
- ▷ Code bit (DECO) and set bit number (ENCO)
- ▷ Selection functions (SEL, MUX), minimum and maximum selection (MIN, MAX), limiter (LIMIT)

A detailed description of the logic functions is provided in Chapter 11.7 “Shift functions” on page 389.

Fig. 8.36 shows an example of the logic functions.



**Fig. 8.36** Example of logic functions in the function block diagram

The MUX function is used to select the components whose number is present in the tag #Selection from the first four components of field tag #Field4. If, for example, the tag #Selection has a value of 3, the tag #Field4[3] will be selected.

If the value of #Selection is not between 1 and 4, the value of the tag #Real4 is used as a substitute. The result of the selection is limited to the range between  $-375$  and  $+1200$  and output to #Real1.

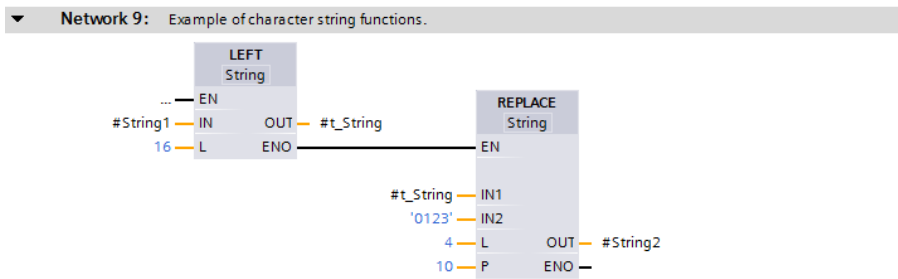
The MUX box has been extended in the example by two inputs: select the box when programming and then the *Insert input* command from the shortcut menu.

### 8.5.9 Functions for strings in the function block diagram

Boxes with the following functions for strings are available in the programming language FBD:

- ▷ LEN Outputs the length of a string
- ▷ CONCAT Combines two strings together
- ▷ LEFT Outputs the left part of a string
- ▷ RIGHT Outputs the right part of a string
- ▷ MID Outputs the middle part of a string
- ▷ DELETE Deletes part of a string
- ▷ INSERT Inserts characters into a string
- ▷ REPLACE Replaces characters in a string
- ▷ FIND Outputs the position of a searched character

A detailed description of these functions is provided in Chapter 11.9 “Processing of strings (Data type STRING)” on page 398.



**Fig. 8.37** Example of functions for strings in the function block diagram

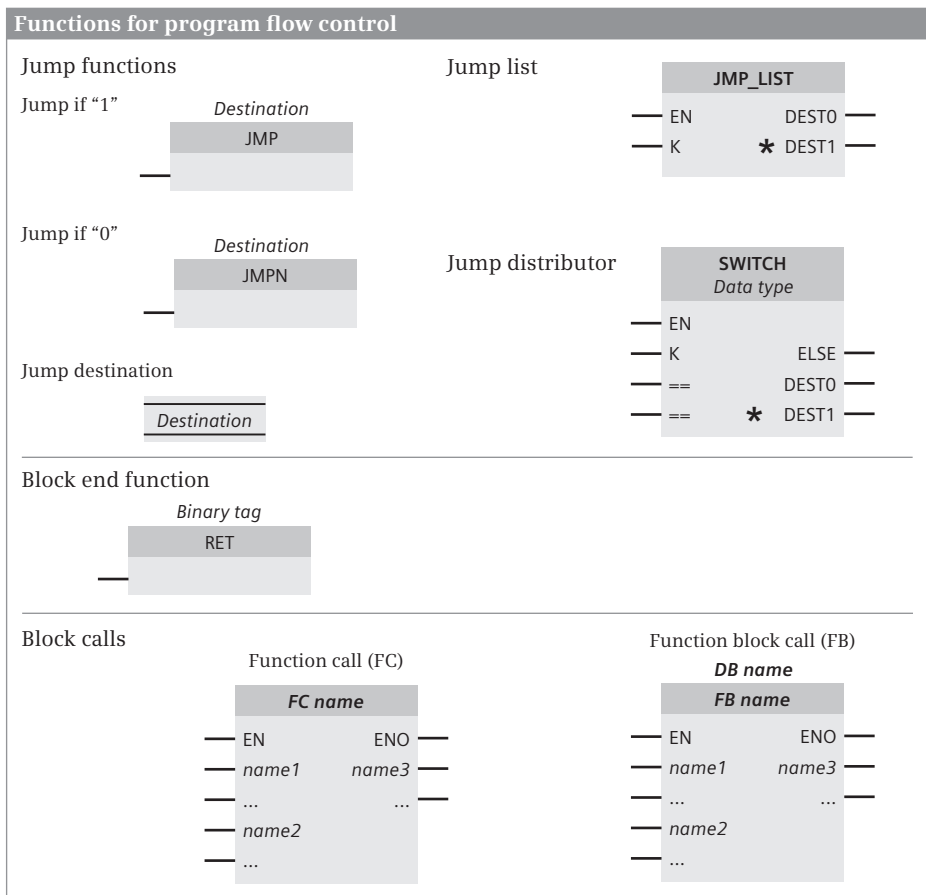
Fig. 8.37 shows an example of string functions. The tag #String1 has the STRING format and is 24 characters long. By means of the LEFT box, 16 characters are removed left-justified from the tag and saved in the intermediate memory #t\_String. REPLACE replaces characters in a string. In the example, the characters '0123' are replaced in the tag #t\_String starting at the 10th position, and the complete string is written in tag #String2.

## 8.6 Functions for program flow control (FBD)

The functions for program flow control are:

- ▷ The jump functions to continue program execution in the desired network
- ▷ The jump list to select a jump destination depending on a numerical value
- ▷ The jump distributor for selecting a jump destination depending on number ranges
- ▷ The block end function to end program execution in the block
- ▷ The block call functions for calling functions and function blocks

Fig. 8.38 shows an overview of these functions. A detailed description of these functions is provided in Chapter 12 “Program flow control” on page 406.



**Fig. 8.38** Overview of functions for program flow control in the function block diagram



### 8.6.1 Jump functions in the function block diagram

A *JMP* or *JMPN* jump function is used to exit the linear processing in a block and – depending on the result of preceding logic operation – continue this processing in another network in the block. If there is no preceding logic operation for *JMP*, the jump function is always performed. To program a jump function, drag the *JMP* or *JMPN* function from the program elements catalog under *Basic instructions > Program control operations* to the working area.

The jump label above the jump function defines the jump destination, which must be at the beginning of a network. To program the jump destination, drag the *Label* function from the program elements catalog under *Basic instructions > Program control operations* to the working area.

Fig. 8.39 shows a jump function using a program loop as an example. In a *#Current* data field with 16 components from *#Current[0]* to *#Current[15]*, the maximum value is searched for. The tags *#Index* and *#MaxValue* are initialized with the value 0. A comparison function in the program loop compares the value of *#MaxValue* with the value of *#Current[#Index]*. If *#MaxValue* is less than *#Current[#Index]*, it is overwritten with the larger value of *#Current[#Index]*. *#Index* is then increased by +1. As long as *#Index* is less than or equal to 15, it jumps to the beginning of the program loop (to the jump destination *MaxSearch*) and the program part runs again.

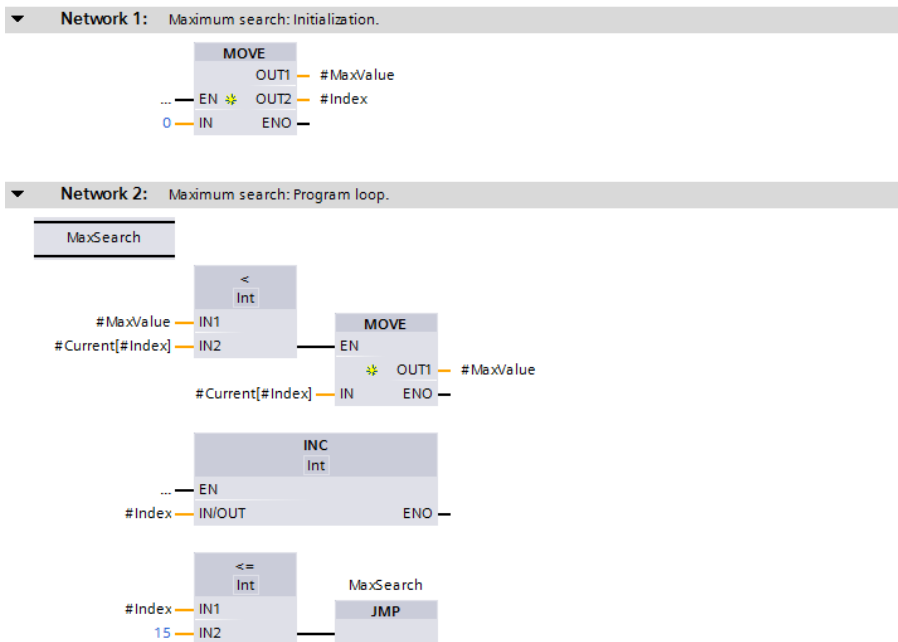


Fig. 8.39 Example of a program loop with conditional jump

### 8.6.2 Jump list in the function block diagram

The jump list is represented as a box. It is only processed if the EN input signal state is “1”. The value of parameter K (0 to 99) determines the box output whose jump destination is jumped to. To program the jump list, drag the *JMP\_LIST* function from the program elements catalog under *Basic instructions > Program control operations* to the working area.

If in Fig. 8.40 the *#JumpSelection* tag has value 0, it jumps to the *Adder* jump label; if the value is 1, to the jump label *FC\_call*, and if the value is 3, to the jump label *FB\_call*.

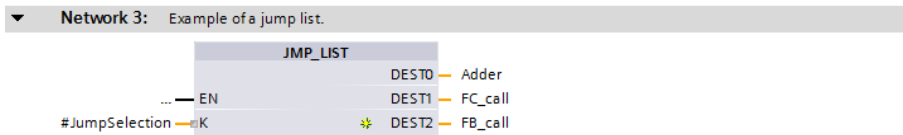


Fig. 8.40 Example of a jump list

### 8.6.3 Jump distributor in the function block diagram

The jump distributor is represented as a box. The box is only processed if the EN input signal state is “1”. The value of parameter K is compared with a value of one of the other input parameters. If the two match, program processing continues at the assigned jump destination. The comparison operations can be selected from a drop-down list. To program a jump distributor, drag the *SWITCH* function from the program elements catalog under *Basic instructions > Program control operations* to the working area.

If in Fig. 8.41 the *#JumpSelection* tag has a value less than 10, it jumps to jump label *FC\_call*; for a value greater than 120 to jump label *FB\_call*; otherwise to jump label *Adder*.

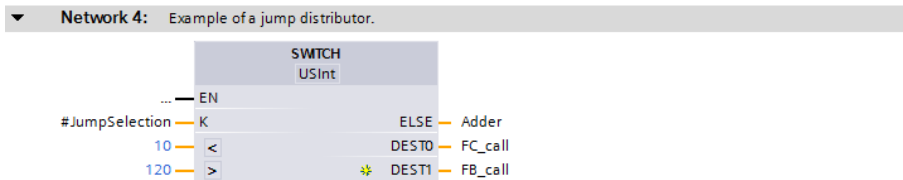


Fig. 8.41 Example of a jump distributor

### 8.6.4 Block end function in the function block diagram

The processing in a block is terminated by the RET box. The block end function may not be present in a network together with a jump function.

To program a block end function, drag the *RET* function from the program elements catalog under *Basic instructions > Program control operations* to the working area.

In Fig. 8.42, the block is exited if an error occurs when processing the ADD box. The ENO output has the signal state “0” which is then negated, thus triggering the RET box. The RET box receives the logic operation result “0” (which is output by the terminated block at the ENO output) by means of the FALSE constant. The result can be scanned in the calling block.

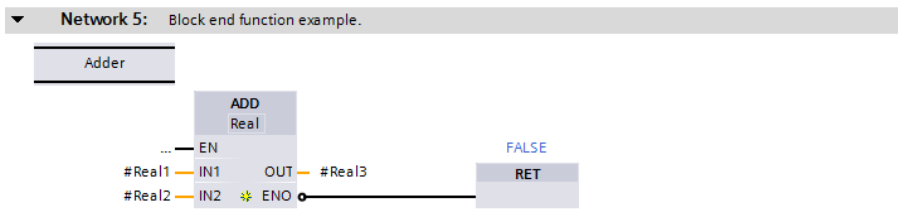


Fig. 8.42 Examples of a block end function

### 8.6.5 Block call functions in the function block diagram

Calling of blocks is represented by EN/ENO boxes. With functions (FC), the block name is present quasi as a function name in the box; with function blocks, the instance name (the name of the instance data block or of the local instance) is additionally present above the box.

A block call is programmed by opening the *Program blocks* folder in the project tree and dragging the desired block to the working area.

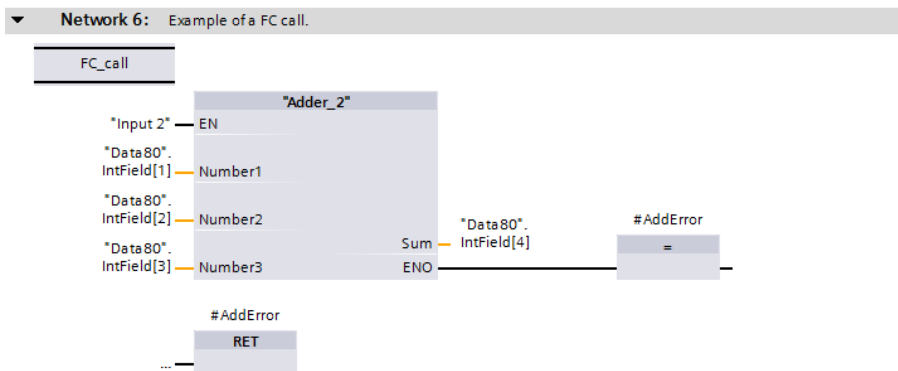
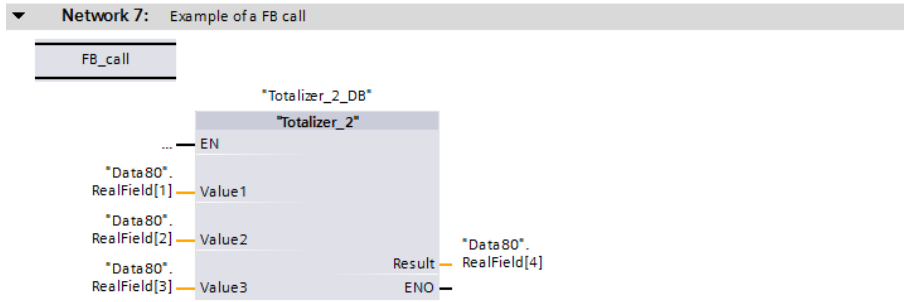


Fig. 8.43 Example of calling a function (FC)

In the example in Fig. 8.43, if the signal state is “1” at the “Input 2” tag, the function “Adder\_2” is called. Program processing in the block is then ended with the value of the return variable *#AddError*.

In the example in Fig. 8.44, the “Totalizer” function block is called. Its instance data is present in the data block “Totalizer\_DB”.



**Fig. 8.44** Example of calling a function block

## 9 Structured Control Language SCL

### 9.1 Introduction to programming with SCL

This chapter describes programming with Structured Control Language (SCL); it uses examples to show how the program functions are represented in SCL. You can find a description of the individual functions, e.g. comparison functions, in Chapters 10 “Basic functions” on page 328, 11 “Digital functions” on page 355, and 12 “Program flow control” on page 406.

Use of the program and symbol editor, which generally applies to all programming languages, is described in Chapter 6 “Program editor” on page 178.

SCL is used to program the contents of blocks. What blocks are, and how they are created, is described in Chapters 5.3.1 “Block types” on page 125 and 6.3 “Programming a code block” on page 183.

#### 9.1.1 Programming with SCL in general

You use SCL to program the control function of the programmable controller – the user program or control program. The user program is organized in different types of blocks.

Fig. 9.1 shows the SCL program for a FIFO register. With a rising edge at *#Write*, this block writes the value present at the *#Input* parameter into a FIFO register. With a rising edge at *#Read*, the value at *#Output* is output again. The values are read out in the order in which they were written into the register (FIFO, first in first out). The register can be emptied using *#Delete*. The two displays *#Full* and *#Empty* show the status of the register (*#Full* and *#Empty* are each set following writing or reading). The block works with a write pointer and a read pointer.

The program editor constructs an SCL program line by line. You commence with the first statement in the first line. Each SCL statement is concluded by a semicolon. You can write several statements in one line, or one statement can occupy several lines.

You can make the SCL program clearer and easier to read by using comments and empty lines. Comments and empty lines have no influence on the function of the SCL program.

Line comments commence with two slashes and terminate at the end of the line. Block comments commence with left parenthesis and asterisk, can extend over several lines, and terminate with asterisk and right parenthesis.

In order to program an SCL statement, use the keyboard to enter the statements in a line of the input field. Dragging the statement with the mouse from the program

elements catalog is of advantage with SCL if you import functions with a parameter list into your program. To call self-created blocks, drag the blocks from the *Program blocks* folder in the project tree into a line.

```

1 //Example FIFO register
2 (*****
3 Check control inputs
4 *****
5 IF #Write AND NOT #Write_EM
6     THEN #Write_EM := #Write; GOTO Write_register;
7     ELSE #Write_EM := #Write;
8 END_IF;
9 IF #Read AND NOT #READ_EM
10    THEN #READ_EM := #Read; GOTO Read_register;
11    ELSE #READ_EM := #Read;
12 END_IF;
13 IF #Delete AND NOT #Delete_EM
14    THEN #Delete_EM := #Delete; GOTO Delete_register;
15    ELSE #Delete_EM := #Delete;
16 END_IF;
17 RETURN;
18 (*****
19 Write_register:
20 IF #Level = #Register_length - 1
21     THEN #Full := TRUE;
22     ELSE #Register[#Write_pointer] := #Input_value;
23         #Level := #Level + 1;
24     IF #Write_pointer = #Register_length
25         THEN #Write_pointer := 0;
26         ELSE #Write_pointer := #Write_pointer + 1;
27     END_IF;
28     #Empty := FALSE;
29 END_IF; RETURN;
30 (*****
31 Read_register:
32 IF #Level = 0
33     THEN #Empty := TRUE; #Output_value := 0;
34     ELSE #Output_value := #Register[#Read_pointer];
35         #Level := #Level - 1;
36     IF #Read_pointer = #Register_length
37         THEN #Read_pointer := 0;
38         ELSE #Read_pointer := #Read_pointer + 1;
39     END_IF;
40     #Full := FALSE;
41 END_IF; RETURN;
42 (*****
43 Delete_register:
44 #Output_value := 0; #Level := 0;
45 #Write_pointer := 0; #Read_pointer := 0;
46 #Full := FALSE; #Empty := FALSE;
47 (*****
48 (** Block end **)

```

Fig. 9.1 Example of a block with SCL program

### 9.1.2 SCL statements and operators

The SCL program consists of a sequence of individual STL statements. Fig. 9.2 shows which types of SCL statements exist.

The simplest case with a *Value assignment* is that the content of a tag is transferred to another tag. *Control statements* guide program execution, for example with program loops. *Block calls* are used to continue program execution in the called block.

#### Operators

An expression represents a value. It can comprise a single address (a single tag) or several addresses (tags) which are linked by operators.

Example: “a + b” is an expression; “a” and “b” are addresses, “+” is the operator.

The sequence of logic operations is defined by the priority of the operators and can be controlled by parentheses. Mixing of expressions is permissible providing the data types generated during calculation of the expression permit this.

SCL provides the operators specified in Table 9.1. Operators of equal priority are processed from the left to the right.

**Table 9.1** Operators with SCL

Operation	Name	Operator	Priority
Parentheses	Left parenthesis, right parenthesis	(, )	1
Arithmetic	Power	**	2
	Unary plus, unary minus (sign)	+, -	3
	Multiplication, division	*, /, DIV, MOD	4
	Addition, subtraction	+, -	5
Comparison	Less than, less than-equal to, greater than, greater than-equal to	<, <=, >, >=	6
	Equal to, not equal to	=, <>	7
Binary logic operation	Negation (unary)	NOT	3
	AND logic operation	AND, &	8
	Exclusive OR	XOR	9
	OR logic operation	OR	10
Assignment	Assignment	:=	11

“Unary” means that this operator has a fixed assignment to an address

**SCL statements****SCL statement**

An SCL statement consists of a jump label with subsequent colon and the actual statement, which is terminated by a semicolon. The statement can extend over several lines. The statement can be followed by a (line) comment, which is commenced by two slashes and extends up to the end of the line. The jump label (including colon) and the line comment can be omitted.

*General SCL statement*

Label	:	SCL statement	;	//	Comment
-------	---	---------------	---	----	---------

**Value assignment**

A value assignment transfers the value of an expression to a tag. An expression can be a single tag or a formula for calculating a value. A formula links the tags by means of operators. Depending on the type of logic operation, a distinction is made between arithmetic expressions, comparison expressions, and logical expressions.

*Value assignment with assignment operator*

Label	:	Variable	:=	Expression	;	//	Comment
		#Result	:=	#Variable AND #Variable	;		Logical expression
		#Result	:=	#Variable >= #Variable	;		Comparison expression
		#Result	:=	#Variable + #Variable	;		Arithmetic expression

**Control statement**

A control statement controls the processing sequence in the program by means of branching and program loops which are processed repeatedly. A control statement begins with a keyword (xxx) and is terminated by END\_xxx.

*Control statement*

Label	:	xxx	Statement sequence	END_xxx	;	//	Comment
		IF	Statement sequence	END_IF	;		IF branch
		CASE	Statement sequence	END_CASE	;		CASE branch
		FOR	Statement sequence	END_FOR	;		FOR loop
		WHILE	Statement sequence	END_WHILE	;		WHILE loop
		REPEAT	Statement sequence	END_REPEAT	;		REPEAT loop

**Block call**

The call of a block without return value consists of the block name and the following parameter list in parentheses. If the block has a return value, the block call following an assignment operator is present in a value assignment or an expression.

Most *extended statements* in the Program Elements catalog are calls of system blocks with return value.

*Block call*

Label	:	Block name (parameter list)	;	//	Comment
		Variable := Block name (parameter list)	;		

**Fig. 9.2** Types of SCL statements



## Expressions

An expression is a formula for calculating a value and consists of addresses (tags) and operators. In the simplest case, an expression is an address, a tag, or a constant. A sign or a negation can also be included.

An expression can consist of addresses that are linked together by operators. Expressions can also be linked by operators. Expression can therefore have a very complex structure. Parentheses can be used to control the processing sequence in an expression.

The result of an expression can be assigned to a tag or a block parameter or used as a condition in a control statement.

Expressions are distinguished according to the type of logic operation into arithmetic expressions, comparison expressions, and logic expressions.

## 9.2 Programming binary logic operations with SCL

The binary logic operations are executed in SCL with logic expressions in conjunction with binary tags or expressions which deliver a binary result. The binary operations can be “nested” using parentheses and thus influence the processing sequence (Table 9.2).

**Table 9.2** Binary logic operations with SCL

Operation	Operand	Function
&	Binary operand or binary expression	Scan for signal state “1” and combination according to AND logic operation
AND	Binary operand or binary expression	Scan for signal state “1” and combination according to AND logic operation
OR	Binary operand or binary expression	Scan for signal state “1” and combination according to OR logic operation
XOR	Binary operand or binary expression	Scan for signal state “1” and combination according to exclusive OR logic operation
NOT	–	Negation of result of logic operation

### 9.2.1 Scanning for signal states “1” and “0”

The scanning of a binary operand in SCL is always the direct scanning of the status of the binary operand. This corresponds to scanning for signal state “1”. If scanning for signal state “0” is required for the program function, one uses the NOT operator in order to negate the result of scan. NOT can also be used to negate the result of binary expressions.

Fig. 9.3 shows a sensor connected to the programmable controller which is scanned for signal state “1” (left-hand side) and for signal state “0” (right-hand side). The result of the scan is connected directly to a contactor.

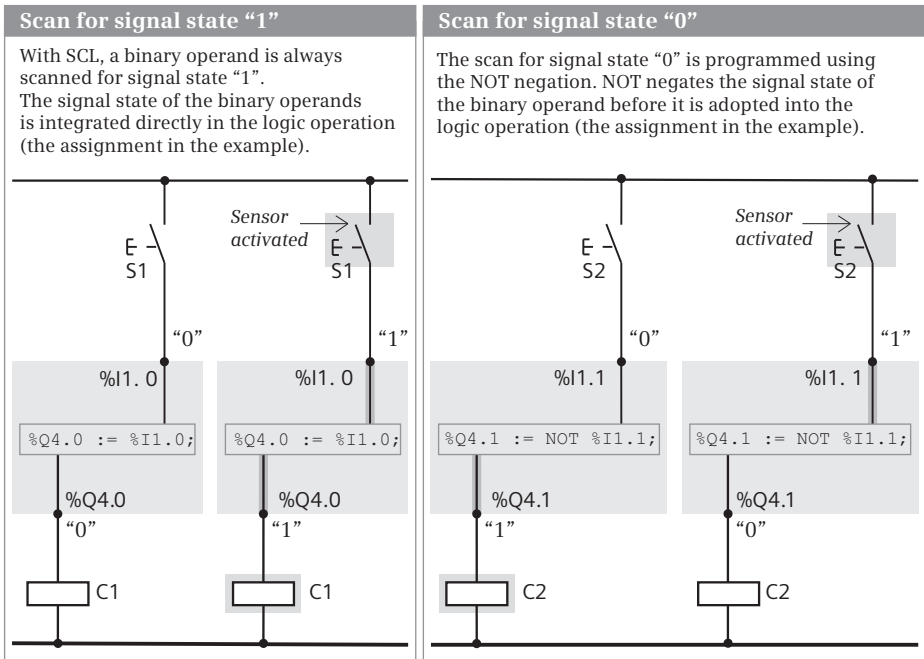


Fig. 9.3 Scanning for signal states "1" and "0"

When scanning for signal state "1" (left-hand side), the result of the scan is equal to the signal state of the sensor: if sensor S1 is open, input %I1.0 has the signal state "0" and the logic operation is not fulfilled. Contactor C1 controlled by output %Q4.0 does not pull up. If sensor S1 is then activated, input %I1.0 has the signal state "1". The function is fulfilled, and the contactor C1 connected to output %Q4.0 pulls up.

When scanning for signal state "0" (right-hand side), the result of the scan is equal to the negated signal state of the sensor: if sensor S2 is open, input %I1.1 has the signal state "0". The negation symbol negates this signal state, and the result of the scan is "1". The logic operation is thus fulfilled. Contactor C2 controlled by output %Q4.1 pulls up. If sensor S2 is then activated, input %I1.1 has the signal state "1". The negation symbol negates the signal state, and the result of the scan is "0". The function is not fulfilled, and contactor C2 connected to output %Q4.1 does not pull up.

### 9.2.2 Taking account of the sensor type for SCL

If you scan a sensor in your program, you must take into consideration whether it is an NO or NC contact. Depending on the type of sensor, different signal states are present at the corresponding input when the sensor is activated: "1" with an NO contact and "0" with an NC contact. It is not possible for the control processor to determine whether an NC or NO contact is connected to an input. It can only recognize the signal state "1" or "0".

If you write the program to obtain “1” when a sensor is activated in order to link it further, you must scan the input in different ways depending on the type of sensor. The negation NOT is available for this purpose. Direct scanning delivers “1” if the scanned input is also “1”. Together with the NOT negation, the scan then delivers “1” if the scanned input is “0”. In this manner you can also directly scan inputs which are to execute activities when the signal state is “0” (“zero-active”) and connect the result of the scan further.

The example in Fig. 9.4 shows the programming dependent on the type of sensor. The AND function used is then fulfilled, i.e. the function result has signal state “1”

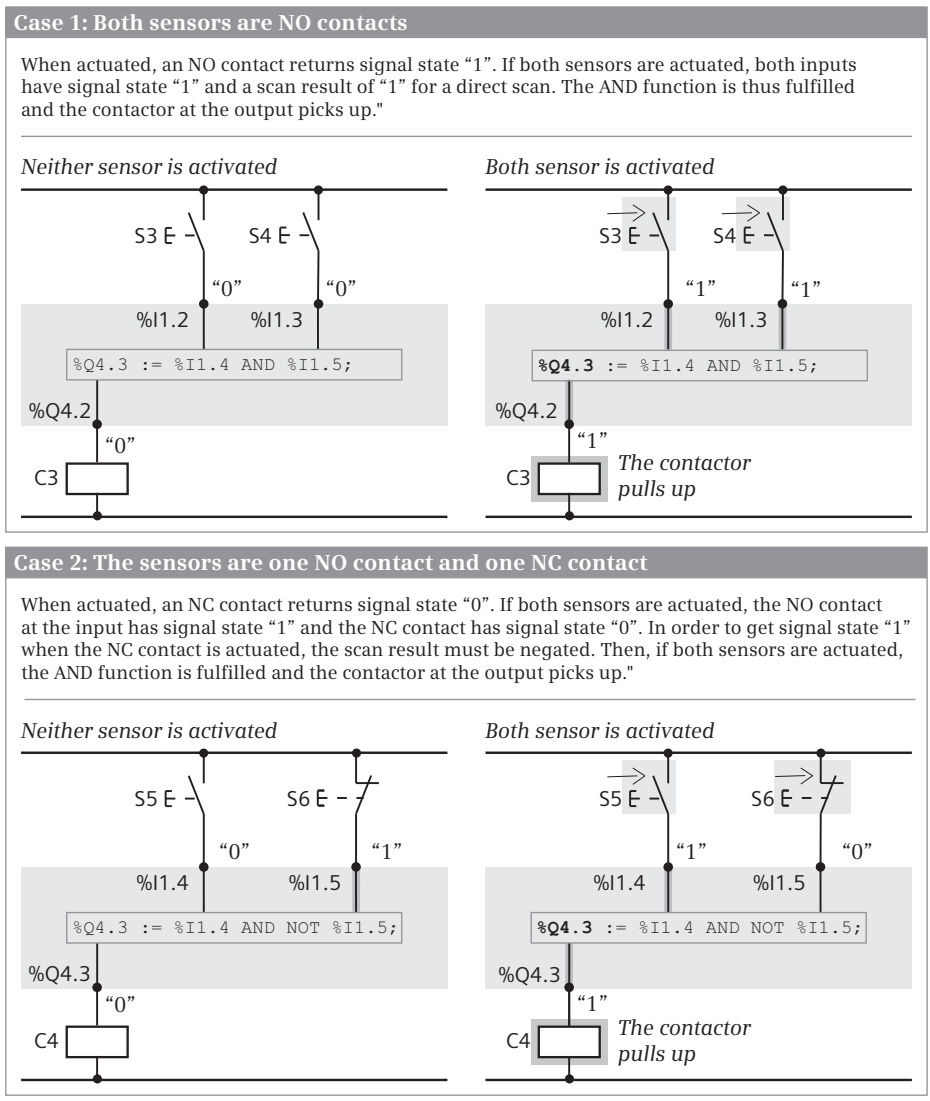


Fig. 9.4 Taking account of the sensor type for SCL

if the result of the scan “1” is present at all function inputs (for description, see Chapter 9.2.3 „AND function“).

In the first case, two NO contacts are connected to the programmable controller, in the second case one NO contact and one NC contact. In both cases, a contactor connected to an output is to pull up when both sensors are activated. If an NO contact is activated, the signal state at the input is “1” and is directly scanned so that the AND function can be fulfilled when the sensor is activated. If both NO contacts are activated, the AND logic operation is fulfilled and the contactor pulls up.

When activating an NC contact, the signal state at the input is “0”. In order to achieve scan result “1” in this case when activating the contact, it is necessary to negate the scan result. Therefore, in the second case, the NO contact must be scanned directly and the NC contact must be scanned with the negation NOT in order for the contactor to pull up when both sensors are activated.

### 9.2.3 AND function

An AND function is fulfilled and provides the result “1” (TRUE) if all function inputs have the scan result “1”. A description of the AND function is provided in Chapter 10.1.5 “AND function, series connection” on page 331.

SCL implements the AND logic operation using a logic expression with the operators & or AND, which link binary tags or binary expressions.

Fig. 9.5 shows an example of an AND logic operation with three inputs. The tags “Input 1” and “Input 2” are scanned directly. The scan result of the tag “Input 3” is negated. All three scan results are linked according to an AND logic operation. The AND function is fulfilled if “Input 1” and “Input 2” have signal state “1” and “Input 3” has signal state “0”.

### 9.2.4 OR function

An OR function is fulfilled and provides the result “1” (TRUE) if one or more function inputs have scan result “1”. A description of the OR function is provided in Chapter 10.1.6 “OR function, parallel connection” on page 332.

SCL implements the OR logic operation using a logic expression with the operator OR, which links binary tags or binary expressions.

```

1  //-----
2  // Binary logic operations with SCL
3  //-----
4  // AND function with three inputs, one of them negated
5  "Output 1" := "Input 1" AND "Input 2" AND NOT "Input 3";
6
7  // OR function with three inputs, one of them negated
8  "Output 2" := "Input 1" OR "Input 2" OR NOT "Input 3";
9
10 // Exclusive OR function (XOR) with two inputs
11 "Output 3" := "Input 1" XOR "Input 2";
12

```

**Fig. 9.5** Examples of binary logic operations with SCL

Fig. 9.5 shows an example of an OR logic operation with three inputs. The tags “*Input 1*” and “*Input 2*” are scanned directly. The scan result of the tag “*Input 3*” is negated. All three scan results are linked according to an OR logic operation. The OR function is fulfilled if “*Input 1*” or “*Input 2*” has signal state “1” or “*Input 3*” has signal state “0”.

### 9.2.5 Exclusive OR function

An exclusive OR function (antivalence function) is fulfilled and provides the result “1” (TRUE) if an odd number of function inputs have scan result “1”. A description of the exclusive OR function is provided in Chapter 10.1.7 “Exclusive OR function, non-equivalence function” on page 333.

SCL implements the exclusive OR logic operation using a logic expression with the operator XOR, which links binary tags or binary expressions.

Fig. 9.5 shows an example of an exclusive OR logic operation with two inputs. The tags “*Input 1*” and “*Input 2*” are scanned directly. The scan results are linked according to an exclusive OR logic operation. The exclusive OR function is fulfilled (it supplies the value TRUE) if the signal states at “*Input 1*” and “*Input 2*” are different.

### 9.2.6 Combined binary logic operations

The AND, OR, and exclusive OR functions can be freely combined with one another. With SCL the operators have the following priority regarding execution: AND or & are executed before XOR, followed by OR. NOT is executed before the logic operation operators.

Logic operations such as the ORing of AND functions do not require parentheses, as shown in the top example in Fig. 9.6. The first AND function is fulfilled if “*Input 1*” and “*Input 2*” have signal state “1”; the second AND function is fulfilled if “*Input 3*” and “*Input 4*” have signal state “1”. The tag “*Output 4*” is set if the first AND function is fulfilled, or if the second AND function is fulfilled, or if both are fulfilled.

```

13 //-----
14 // Combined binary logic operation
15 //-----
16 // ORing of AND functions - does not require parentheses
17 "Output 4" := "Input 1" AND "Input 2" OR "Input 3" AND "Input 4";
18
19 // ANDing of OR functions - parentheses required
20 "Output 5" := ("Input 1" OR "Input 2") AND ("Input 3" OR "Input 4");
21
22 // ORing of XOR functions - does not require parentheses
23 "Output 6" := "Input 1" XOR "Input 2" OR "Input 3" XOR "Input 4";
24

```

**Fig. 9.6** Examples of combined binary logic operations with SCL

This logic operation does not require parentheses since the AND function is processed “before” the OR function because of its higher priority. The processing priority can be influenced using parentheses. The expressions in the parentheses are processed first as it were. Parentheses can be nested.

Logic operations such as the ANDing of OR functions require parentheses, as shown in the following example in Fig. 9.6. The first OR function is fulfilled if “Input 1”, or “Input 2”, or both tags have signal state “1”; the second OR function is fulfilled if “Input 3”, or “Input 4”, or both tags have signal state “1”. Both OR functions are present in parentheses and their results of logic operation are linked according to an AND logic operation. The “Output 5” tag is set if both OR functions are fulfilled. The logic operation of exclusive OR functions according to an OR logic operation (last example in Fig. 9.6) also does not require parentheses. However, parentheses can be used for reasons of clarity.

### 9.2.7 Negating the result of logic operation

The NOT operator negates the result of logic operation at any position in an logic operation. Using the NOT operator it is possible in a simple manner to obtain:

- ▷ a NAND function (negated AND function, is fulfilled if at least one input has the result of scan “0”),
- ▷ a NOR function (negated OR function, is fulfilled if all inputs have the result of scan “0”), and
- ▷ an inclusive OR function (equivalence function, negated exclusive OR function, is fulfilled if an even number of inputs has the result of scan “1”).

Fig. 9.7 shows the negation of binary functions. The functions are present in parentheses since they have a lower processing priority than NOT. First the result of the binary function is generated in parentheses and then it is negated and assigned to the output tag.

```

25 //-----
26 // Negating the result of the logic operation
27 //-----
28 // NAND function - AND function with negated output
29 "Output 7" := NOT ("Input 1" AND "Input 2");
30
31 // NOR function - OR function with negated output
32 "Output 8" := NOT ("Input 3" OR "Input 4");
33
34 // Inclusive OR function - XOR function with negated output
35 "Output 9" := NOT ("Input 5" XOR "Input 6");
36

```

**Fig. 9.7** Examples of the negation of binary functions

## 9.3 Programming memory functions with SCL

The memory functions control binary tags such as outputs or bit memories. SCL has value assignment as a memory function, which directly imports the result of logic operation of the expression. The set and reset functions, which are only performed if the result of logic operation is “1”, can be emulated.

### 9.3.1 Value assignment of a binary tag

The value assignment assigns the result of logic operation of the expression to the right of the assignment operator to the binary tag on the left. The response of the assignment is described in Chapter 10.2.2 “Simple and negated coil, assignment” on page 334.

An example of a (binary) value assignment is shown in Fig. 9.8. Here, if the logic operation is fulfilled and both tags “*Input 1*” and “*Input 2*” have the same signal status, the tag “*Output 1*” is set to signal state “1”, and if the logical operation is not fulfilled, to signal state “0”. If an assignment should be carried out negated, use the negation NOT to negate the signal state of the expression.

```

1 //-----
2 // Memory functions
3 //-----
4 // Direct assignment, negated assignment
5 "Output 1" := "Input 1" XOR NOT "Input 2";
6 "Output 2" := NOT ("Input 1" AND "Input 2");
7
8 // Set and Reset with RLO = "1"
9 IF "Input 2" AND "Input 3" THEN "Output 3" := TRUE; END_IF;
10 IF "Input 5" OR "Input 6" THEN "Output 3" := FALSE; END_IF;
11

```

**Fig. 9.8** Assigning, setting, and resetting with SCL

### 9.3.2 Setting and resetting

With SCL, the setting and resetting on signal state “1” can be emulated, for example with a simple IF branch.

In Fig. 9.8, the tag “*Output 3*” is set to signal state “1” (value TRUE) if the AND logic operation from “*Input 3*” and “*Input 4*” is fulfilled (supplies the value TRUE). If the AND logic operation is not fulfilled (if it has the value FALSE), “*Output 3*” is not affected. Resetting of “*Output 3*” is carried out in a similar manner: If the expression “*Input 5*” OR “*Input 6*” is fulfilled (supplies the value TRUE), “*Output 3*” is set to signal state “0” (FALSE). An expression which is not fulfilled does not influence “*Output 3*”. Resetting is programmed following setting and is therefore “dominant”. If both conditions are fulfilled, “*Output 3*” is reset or remains reset.

### 9.3.3 Edge evaluation

Edge evaluation detects a change in a binary signal state.

With SCL, a change in signal state can be detected by comparing the current signal state with the previous one. The previous signal state is saved in a so-called edge trigger flag. This is, for example, a bit from the bit memories or data operand area. After the evaluation, the signal state of the edge trigger flag must be updated to the signal state of the input signal, otherwise an edge will be detected again the next time the program runs.

Either the statements dependent on a signal change are found directly at the edge evaluation or the information about a detected signal change is saved in a bit memory, the “pulse flag”. Its signal state may be scanned at any point in the user program.

Fig. 9.9 shows various types of edge evaluation. A rising edge is present if the input signal (in the first example “*Input 1*”) has signal state “1” and the edge trigger flag (in the first example “*Edge 1*”) has signal state “0”. Then the statements after THEN are processed. The signal state of the edge trigger flag is then updated.

The second example shows the evaluation of a falling edge. The last two examples use a pulse flag (this can be a bit memory or a data bit) to respond to the edge at another point in the program.

```

12 //-----
13 // Edge evaluation
14 //-----
15 // Evaluation for positive (rising) edge
16 IF "Input 1" AND NOT "Edge 1" THEN (* Statements *); END_IF;
17 "Edge 1" := "Input 1";
18
19 // Evaluation for negative (falling) edge
20 IF NOT "Input 2" AND "Edge 2" THEN (* Statements *); END_IF;
21 "Edge 2" := "Input 2";
22
23 // Evaluation with pulse output - rising edge
24 "Pulse 1" := "Input 3" AND NOT "Edge 3";
25 "Edge 3" := "Input 3";
26 // ...
27 IF "Pulse 1" THEN (* Statements *); END_IF;
28
29 // Evaluation with pulse output - falling edge
30 "Pulse 2" := NOT "Input 4" AND "Edge 2";
31 "Edge 2" := "Input 4";
32 // ...
33 IF "Pulse 2" THEN (* Statements *); END_IF;
34

```

Fig. 9.9 Examples of edge evaluation with SCL



## 9.4 Programming timer and counter functions with SCL

### 9.4.1 IEC timer functions

You can use the timer functions to implement timing processes in the user program such as waiting and monitoring times, measurement of a time interval, or the generation of pulses. A detailed description of the IEC timer functions is provided in Chapter 10.4 “Time functions” on page 344.

A timer function can be started with one of the four behavior patterns TP, TON, TOF, and TONR. A timer function requires internal data for each application. You can specify where this data is to be saved when programming: For the *Single instance* entry in its own data block with the data type IEC\_TIMER and for the *Multi-instance* entry in the instance data block of the calling function block with a data type that depends on the behavior of the timer function (TP\_TIME, TON\_TIME, TOF\_TIME, TONR\_TIME). You address a timer function with the name of the instance data – data block or local data.

For programming, drag the corresponding symbol (TP, TON, TOF, or TONR) with the mouse from the program elements catalog under *Basic instructions > Timers* into a line on the working area. When positioning, you select either as single instance or as local instance. The instance data block generated automatically when selecting as a single instance is saved in the project tree under *Program blocks > System blocks > Program resources*.

With the IEC timer functions, a binary tag must be connected to the IN input, and a duration to the PT input. You can also directly access the output parameters using the instance data, for example with “DB\_name”.Q for a single instance or #Instance\_name.Q for a local instance.

Fig. 9.10 shows the two timer functions #Timer\_on and #Timer\_off, which were programmed as local instance. Five seconds after switching on with the signal “Switch on fan”, the “Fan drive” tag is set to signal state “1” and remains set for 10 seconds, determined by the timer function #Timer\_off started as OFF delay. The time status can either be scanned on the start operation of the timer function or with the component Q at another point in the user program.

```

1  //-----
2  // Timer functions
3  //-----
4  // Fan switch on and off delayed
5  #Timer_on (IN := "Switch on fan",
6  [         PT := T#5s);
7
8  #Timer_off (IN := #Timer_on.Q,
9  [         PT := T#10s,
10 [        Q => "Fan drive");
11

```

**Fig. 9.10** Example of IEC timer functions with SCL

### 9.4.2 IEC counter functions

A counter function implements counting processes in the user program. A counter function can count up and down; the numerical range depends on the data type of the preset value. The data types USINT, UINT, UDINT, SINT, INT and DINT are available.

The counting frequency of a counter function depends on the execution time of the user program. In order to count, the CPU must recognize a change in the signal state of the input pulse, i.e. the input pulse and the pause must be present for at least one program cycle. The longer the program execution time, the lower the counting frequency. A detailed description of the counter functions is provided in Chapter 10.5 “Counter functions” on page 349.

A counter function can be controlled with one of the three behavior patterns CTU, CTD, and CTUD. A counter function requires internal data for each application. You can specify where this data is to be saved when programming: by specifying *Single instance* for storage in a separate data block, and by specifying *Multi-instance* for storage in the instance data block of the calling function block.

The data type of a counter function is based on the data type of the count value. If, for example, an up-counter (CTU) with a DINT count value is programmed as a single instance, the data type IEC\_DCOUNTER is taken as a basis for the data block (see Chapter 4.8.2 “IEC\_COUNTER system data type” on page 112); as a local instance, the counter function has the data type CTU\_DINT (see Chapter 4.6.2 “Parameter types for IEC counter functions” on page 108). You address the counter function with the name of the instance data – data block or local data.

For programming, drag the corresponding symbol (CTU, CTD, or CTUD) with the mouse from the program elements catalog under *Basic instructions > Counters* into a line on the working area. When positioning, you select either as single instance or as local instance. The instance data block generated automatically when selecting as a single instance is saved in the project tree under *Program blocks > System blocks > Program resources*.

The data type of the counter function is set with a click on the instance name (the green box). With the IEC counter functions, a binary tag must be connected to at least one counter input (CU or CD). Connection of the other function inputs and outputs is optional. You can also directly access the output parameters using the instance data, for example with “*DB\_name*”.*QD* for a single instance or #*Instance\_name.QD* for a local instance.

Fig. 9.11 shows the counter function #*Lock\_counter*, which is called as a local instance. It has saved its data in the instance data block of the calling function block. A component of the counter can be addressed globally with the name of the instance and the component name, for example #*LockCounter.CV*. The example shows the passages through a lock, either forward or backward.

```

16 //-----
17 // Counter functions
18 //-----
19 //Simple lock counter
20 #t_bool1 := "Light barrier 1" AND NOT #Lightbarrier1_edge;
21 #Lightbarrier1_edge := "Light barrier 1";
22 #t_bool2 := "Light barrier 2" AND NOT #Lightbarrier2_edge;
23 #Lightbarrier2_edge := "Light barrier 2";
24 #Lock_counter
25 (CU := "Light barrier 2" AND #t_bool1,
26  CD := "Light barrier 1" AND #t_bool2,
27  R := #Acknowledge,
28  LD := FALSE,
29  FV := 0);
30

```

**Fig. 9.11** Example of a counter function with SCL

## 9.5 Programming digital functions with SCL

The digital functions process digital values mainly with the data types for fixed-point and floating-point numbers.

The “simple” digital functions are implemented with SCL through the value assignment of an expression. When linking two values, the type of digital function depends on the operator used: comparison expression (comparison functions), arithmetic expression (arithmetic and mathematical functions), or logic expression (e.g. word logic operations). The functions for data type conversion (conversion functions) and for shifting and rotating are available for manipulating just one value.

### 9.5.1 Transfer function, value assignment of a digital tag

The “simple” transfer function corresponds with SCL to the value assignment.

Both sides of the value assignment must have a “tolerable” data type that is either automatically adapted (see Chapter 4.3.2 “Implicit data type conversion” on page

```

1 //-----
2 // Digital functions
3 //-----
4 // Transfer functions (Assignment)
5 #var_int := 123;
6 #Quantity := #var_int;
7 #var_dint := #var_int;
8 #var_real := #var_int;
9
10 // Conversion functions
11 #var_int := REAL_TO_INT(#var_real);
12 #Measurement_display :=
13  DINT_TO_BCD32(INT_TO_DINT(#Measurement_temperature));
14

```

**Fig. 9.12** Examples of transfer and conversion functions with SCL

93) or must be converted using a program with conversion functions (see Chapter 11.6 “Conversion functions (Conversion of data type)” on page 376).

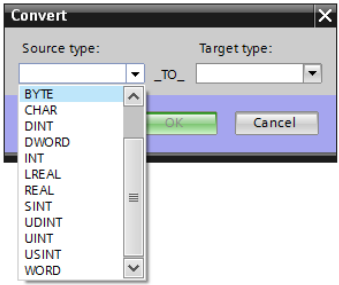
A detailed description of the transfer functions is provided in Chapter 11.1 “Transfer functions” on page 356. Fig. 9.12 shows some examples of value assignments: The constant value 123 is assigned to the `#var_int` tag. The `#Quantity` tag imports the value of the `#var_int` tag. A value in INT data format can be assigned to a tag with data type DINT or REAL, where SCL “implicitly” converts the data type. If no implicit data type conversion is possible, a conversion function must be programmed.

### 9.5.2 Conversion functions

The conversion functions convert the data formats of tags and expressions. SCL can “automatically” convert data types if no information can be lost during conversion (see also Chapter 4.3.2 “Implicit data type conversion” on page 93). In all other cases, the (explicit) conversion functions must be used.

A detailed description of the conversion functions is provided in Chapter 11.6 “Conversion functions (Conversion of data type)” on page 376. Table 9.3 shows the conversion functions available with SCL. When inserting the CONVERT function, the data types involved in the conversion are selected in a dialog box.

**Table 9.3** Conversion functions with SCL

Operation	Function	Data type selection with CONVERT
CONVERT	Data type conversion	
ROUND	Data type conversion from REAL to DINT with rounding to the next integer	
CEIL	Data type conversion from REAL to DINT with rounding to the next higher integer	
FLOOR	Data type conversion from REAL to DINT with rounding to the next lower integer	
TRUNC	Data type conversion from REAL to DINT without rounding	

The program elements catalog contains the conversion functions under *Basic instructions > Converters*.

Fig. 9.12 on page 298 shows examples of simple and “nested” conversion functions where the `#Measurement_display` tag has the data type DWORD.

Table 9.4 shows the data type conversions possible with SCL. There are also the conversions `WORD_TO_BLOCK_DB` and `BLOCK_DB_TO_WORD`. If the permissible numerical range is left during a conversion, the block-internal ENO tag is set to FALSE, and the result of the conversion is invalid.

**Table 9.4** Data type conversion with SCL

to from	BOOL	BYTE	WORD	DWORD	USINT	UINT	UDINT	SINT	INT	DINT	REAL	LREAL	TIME	TOD	DATE	DTL	CHAR	STRING	BCD16	BCD32
BOOL		x	x	x	W	W	W	W	W	W										
BYTE	x		x	x	x	x	x	x	x	x			T	T	T		x			
WORD	x	x		x	x	x	x	x	x	x			T	T	T		x			
DWORD	x	x	x		x	x	x	x	x	x	x		T	T						
USINT	x	x	x	x		x	x	x	x	x	x	x					x	S		
UINT	x	x	x	x	x		x	x	x	x	x	x	T	T	T		x	S		
UDINT	x	x	x	x	x	x		x	x	x	x	x		T			x	S		
SINT	x	x	x	x	x	x	x		x	x	x	x	T				x	S		
INT	x	x	x	x	x	x	x	x		x	x	x	T				x	S	x	
DINT	x	x	x	x	x	x	x	x	x		x	x	T	T	T		x	S		x
REAL				x	x	x	x	x	x	x		x						S		
LREAL				x	x	x	x	x	x	x	x							S		
TIME				T						T	T									
TOD				T			T				T									
DATE				T	T		T	T		T	T					T				
DTL															T	T				
CHAR		x	x	x	x	x	x	x	x	x								S		
STRING					S	S	S	S	S	S	S	S					S			
BCD16									x											
BCD32										x										

Data type conversion is possible: X With CONVERT  
 S with S\_CONV  
 T with T\_CONV  
 W By programmed conversions (example: BOOL\_TO\_INT)

### 9.5.3 Comparison functions

A comparison function is implemented with a comparison expression in SCL.

A comparison expression compares the values of two tags or expressions using a comparison operator. The comparison result has the data type BOOL and can be linked further like a Boolean tag. The comparison result has signal state TRUE if the comparison is fulfilled, otherwise FALSE. The comparison function is described in Chapter 11.2 “Comparison functions” on page 364. Table 9.5 shows the comparison operators available with SCL.

**Table 9.5** Comparison functions with SCL

Operator	Description	Operand data types
=	Compare for equal	USINT, UINT, UDINT, SINT, INT, DINT, REAL, LREAL, CHAR, STRING, TIME, DATE, TIME_OF_DAY (TOD), DTL
<>	Compare for unequal	
<	Compare for greater than	
<=	Compare for greater than-equal	
>	Compare for less than	
>=	Compare for less than-equal	
=	Compare for equal	BOOL, BYTE, WORD, DWORD
<>	Compare for unequal	

In Fig. 9.13 on page 302, a measured value *#Measurement\_temperature* is compared with an upper and a lower limit. The tags *#Measurement\_too\_high* and *#Measurement\_too\_low* are controlled depending on the comparison result. Resetting occurs at a distance from the upper or lower limit (*#Hysteresis* tag).

#### 9.5.4 Arithmetic functions

The arithmetic functions implement the basic arithmetic operations with the data formats for fixed-point and floating-point numbers as well as with time values. SCL uses arithmetic expressions with an arithmetic operator for this purpose.

**Table 9.6** Arithmetic operators with SCL

Operator	Description	Data type *)		
		1st operand	2nd operand	Result
**	Power	Fixed point, floating point	Fixed point, floating point	Fixed point, floating point
*	Multiplication	Fixed point, floating point TIME	Fixed point, floating point Fixed point	Fixed point, floating point TIME
/	Division	Fixed point, floating point	Fixed point, floating point	Fixed point, floating point
DIV	Integer division	Fixed point, floating point TIME	Fixed point, floating point Fixed point	Fixed point, floating point TIME
MOD	Division with remainder as result	Fixed point	Fixed point	Fixed point
+	Addition	Fixed point, floating point TIME TOD DTL	Fixed point, floating point TIME TIME TIME	Fixed point, floating point TIME TOD DTL
-	Subtraction	Fixed point, floating point TIME TOD DATE TOD DTL DTL	Fixed point, floating point TIME TIME DATE DATE TIME TIME DTL	Fixed point, floating point TIME TIME TIME TIME TIME DTL TIME

\*) Fixed point = USINT, UINT, UDINT, SINT, INT, DINT; floating point = REAL, LREAL

You can find a detailed description of these arithmetic functions in Chapters 11.3 “Arithmetic functions for numerical values” on page 366 and 11.4 “Arithmetic functions for time values” on page 369. Table 9.6 shows the arithmetic operators available with SCL with the allowed data types.

The tags or expressions linked to an arithmetic function can have fixed-point, floating-point or time value data types. If different data types can be matched in the context of implicit data type conversion (see Chapter 4.3.2 “Implicit data type conversion” on page 93), a programmed function for (explicit) data type conversion can be dispensed with. The result of an arithmetic function has the “most powerful” of the involved data types. If, for example, you link an INT or DINT tag with a REAL tag, the result is of data type REAL.

In the case of a division, the second operand must not be zero.

In Fig. 9.13, the upper limit of a measured value is monitored under the header “Comparison expressions”. A hysteresis is introduced to ensure that the *#Measurement\_too\_high* and *#Measurement\_too\_low* messages do not “pulsate” when the measurement changes rapidly around the upper or lower limit. The messages are only canceled when the measured value has dropped again below the upper limit or risen again above the upper limit by the magnitude of the hysteresis.

There are several calculation examples under the heading “Arithmetic expressions”: the calculation of reactive electric power, the volume calculation of a ball, one of the solutions of a quadratic equation, and the formation of an arithmetic mean value.

```

15 //-----
16 // Digital functions
17 //-----
18 // Comparison expressions
19 IF #Measurement_temperature >= #Upper_limit
20 | THEN #Measurement_too_high := TRUE; END_IF;
21 IF #Measurement_temperature <= #Upper_limit - #Hysteresis
22 | THEN #Measurement_too_high := FALSE; END_IF;
23 IF #Measurement_temperature <= #Lower_limit
24 | THEN #Measurement_too_low := TRUE; END_IF;
25 IF #Measurement_temperature >= #Lower_limit + #Hysteresis
26 | THEN #Measurement_too_low := FALSE; END_IF;
27
28 // Arithmetic expressions
29 #Reactive_power := #Voltage * #Current * SIN(#phi);
30 #Volume := 4/3 * "pi" * #Radius**3;
31 #Solution_1 := -#p/2 + SQRT(SQR(#p/2) - #q);
32 #Average := (#Motor[1].Leistung + #Motor[2].Leistung)/2;
33
34 // Logic expressions (e.g. word logic operations)
35 #Messages_changes :=
36   "Data90".Messages XOR "Data90".Messages_EM;
37 "Data90".Messages_pos :=
38   #Messages_changes AND "Data90".Messages;
39 "Data90".Messages_neg :=
40   #Messages_changes AND NOT "Data90".Messages;
41 "Data90".Messages_EM := "Data90".Messages;
42

```

**Fig. 9.13** Examples of comparison expressions, arithmetic and logical expressions

### 9.5.5 Mathematical functions

The mathematical functions comprise the trigonometric functions, exponential functions, and logarithmic functions, and deliver a result in floating-point data format. The input tag can have any data type that can be converted into the data type of the output tag using the implicit data type conversion. A detailed description of these math functions is provided in Chapter 11.5 “Mathematical functions” on page 372. Table 9.7 shows the mathematical functions available with SCL.

**Table 9.7** Math functions with SCL

Operation	Function	Operation	Function
SIN	Calculate sine	ASIN	Calculate arcsine
COS	Calculate cosine	ACOS	Calculate arccosine
TAN	Calculate tangent	ATAN	Calculate arctangent
SQR	Generate square	EXP	Generate exponential function to base e
SQRT	Extract square root	LN	Generate Napierian logarithm (to base e)

Under the heading “Arithmetic expressions”, Fig. 9.13 provides several examples of the mathematical functions SIN, SQR, and SQRT.

### 9.5.6 Word logic operations

The word logic operations apply the binary operations AND, OR, and XOR to each bit of a byte, word, or doubleword. SCL uses logic expressions for this.

A detailed description of the word logic operations is provided in Chapter 11.8.2 “Word logic operations (AND, OR, XOR)” on page 392. Table 9.8 shows the word logic operations available with SCL.

Fig. 9.13 on page 302 shows how you can program 32 edge evaluations simultaneously for rising and falling edges. The message bits are collected in a doubleword *Messages*, which is present in data block “*Data.90*”. The edge trigger flags *Messages\_EM* are also present in this data block. If the two doublewords are linked by an XOR logic operation, the result is a doubleword in which each set bit represents a different assignment of *Messages* and *Messages\_EM*, in other words: the associated message bit has changed.

**Table 9.8** Word logic operations with SCL

Operation	Operand data types	Function
AND, & OR XOR	BOOL, BYTE, WORD, DWORD BOOL, BYTE, WORD, DWORD BOOL, BYTE, WORD, DWORD	AND logic operation OR logic operation Exclusive OR logic operation
NOT	BOOL, BYTE, WORD, DWORD	Negation



In order to obtain the positive signal edges, the changes are linked to the messages by an AND logic operation. The bit is set for a rising signal edge wherever the message has a “1” and the change a “1”. This corresponds to the pulse flag of the edge evaluation. If you do the same with the negated message bits – the message bits with signal state “0” are now “1” – you obtain the pulse flags for a falling edge. At the end it is only necessary for the edge trigger flags to track the messages.

### 9.5.7 Shift functions

A shift function shifts the content of a tag bit-by-bit to the left or right. A detailed description of the shift functions is provided in Chapter 11.7 “Shift functions” on page 389. Table 9.9 shows the shift functions available with SCL.

**Table 9.9** Shift functions with SCL

Operation	Data types IN	Data type N	Function
SHR (IN, N) SHL (IN, N)	BYTE, WORD, DWORD BYTE, WORD, DWORD	INT, DINT INT, DINT	Shift to right Shift to left
ROR (IN, N) ROL (IN, N)	BYTE, WORD, DWORD BYTE, WORD, DWORD	INT, DINT INT, DINT	Rotate to right Rotate to left

The program elements catalog contains the shift functions under *Basic instructions* > *Shift and rotate*.

In Fig. 9.14 the tags *#Quantity\_high* and *#Quantity\_low* are available as positive numbers in BCD16 format. They contain three digit decades and in the highest decade the sign in each case (here the value zero). The digits of both numbers are combined into a single number in the *#Quantity\_display* tag. In addition, the *#Quantity\_high* tag is shifted 12 bits to the left (corresponds to three decades) and linked according to an OR operation with the *#Quantity\_low* tag.

```

43 //-----
44 // Digital functions
45 //-----
46 //Shift functions
47 #Quantity_display :=
48   SHL(IN := #Quantity_high, N := 12) OR #Quantity_low;
49

```

**Fig. 9.14** Example of the shift function with SCL

## 9.6 Controlling the program flow with SCL

You can influence processing of the user program by means of the program flow control functions. You can recognize errors in program execution by using the ENO tag, the control statements permit you to create program branches, and the block functions allow you to call and terminate blocks.

### 9.6.1 Working with the ENO tag

The programming language SCL offers a pre-defined tag named ENO with data type BOOL, i.e. ENO is not declared by the user but is always present. This block-local tag shows FALSE to indicate an error in process execution in an SCL block.

In order to use automatic error detection with the ENO tag, the block attribute *Set ENO automatically* must be activated. When compiling the block, additional code is generated for controlling ENO. You activate the block attribute *Set ENO automatically* in the properties of the SCL block under *Attributes*.

#### Error analysis with ENO

At the block start, the ENO tag is always TRUE. ENO is set to FALSE if a called block signals an error or following faulty execution of an arithmetic expression or conversion function. Every error in the further block program also sets ENO to FALSE: ENO is used as a group error message for program execution in a block.

You can scan the ENO tag at any time:

```
#Total := #Total + #New_value;
IF NOT ENO                                //Scan ENO
  THEN (* faulty addition *);
END_IF;
```

In this program, the THEN branch is even executed if faulty program execution took place prior to the addition which ENO also set to FALSE.

You can assign a value to the ENO tag at any time. If you only wish to check the correct execution of the addition (always assuming that a block attribute *Set ENO automatically* is activated), you can also program:

```
ENO := TRUE;                               //Set ENO
#Total := #Total + #New_value;
IF ENO                                     //Scan ENO
  THEN (* no error occurred *);
  ELSE (* faulty addition *);
END_IF;
```

You can also use the ENO tag independent of the block attribute *Set ENO automatically*, for example as a group error message:

```
IF (* error detected *)
  THEN ENO := FALSE; RETURN;              //Reset ENO and exit block
END_IF;
```

When exiting the block, the value of the ENO tag is automatically assigned to the enable output ENO of the block.

### Error evaluation following a block call

A block call can control the ENO tag via the enable output ENO. If the enable output is FALSE (this is the case if an error has occurred in the called block or if the ENO tag has been set to FALSE in the called block by the user), the “block-local” ENO tag is also set to FALSE in the current block.

```
"Block" (In1 := ..., In2 := ...);  
IF NOT ENO THEN (* an error has occurred up to here *);  
END_IF;
```

An error signaled by the called block – as well as previous errors – sets the “block-local” ENO tag to FALSE. If you wish to scan an error signal by the called block independent of a previous error, use the enable output ENO:

```
"Block" (In1 := ..., In2 := ..., ENO => #OK);  
IF NOT #OK THEN (* error in block *); END_IF;
```

The “block-local” ENO tag is not set to FALSE if the called block has not been processed via the enable input EN (with EN equal to FALSE).

### 9.6.2 EN/ENO mechanism with SCL

The EN/ENO mechanism is based on the enable input EN and enable output ENO. EN and ENO are implicitly defined parameters with a block call. EN is permissible for function blocks (FB), ENO for all block types (even system blocks) which can be called. EN and ENO are not displayed by the program editor in the offered template.

EN is the first parameter in the parameter list, ENO the last. Use of these parameters is optional. If you do not require these parameters, simply omit them.

The EN/ENO mechanism is only supported in SCL if the block attribute *Set ENO automatically* is activated.

#### Enable input EN

You can control the calling of a block using the enable input EN. If EN is TRUE or not used, the called block is processed. If EN is FALSE, the called block is not processed. You use the enable input EN in the parameter list like an input parameter:

```
"Block"(EN := #Enable, In1 := ..., In2 := ...);  
(* "Block" is only processed if #Enable = TRUE *)
```

You can use the enable input to implement a conditional block call, which depends on the value of a binary tag or binary expression.

#### Enable output ENO

You can scan the error status of the block using the enable output ENO. If ENO is TRUE, processing has been carried out correctly. If FALSE, the ENO output signals

that an error is present in the block. You can scan the state of the ENO output in the parameter list using a tag:

```
"Block" (In1 := ..., In2 := ..., ENO => #OK);
(* With error-free processing, #OK has the value TRUE *)
```

If the called block signals an error, this is transferred to the block-local ENO tag:

```
"Block" (In1 := ..., In2 := ..., ENO => #OK);
#No_error := ENO;
IF NOT #OK THEN (* error in block *); END_IF;
IF NOT #No_error THEN (* group error message *); END_IF;
```

The `#OK` tag is `FALSE` if block processing was faulty. The `#No_error` tag is `FALSE` if block processing was faulty or if an error was already present prior to the block call.

If a function block with `EN = FALSE` is not processed, this has no influence on the “block-local” ENO tag. However, the ENO output is set to `FALSE`.

```
"Block"(EN := #Enable, ... , ENO => #OK);
#No_error := ENO;
```

If the `#Enable` tag is `FALSE`, the `#OK` tag is `FALSE` and the `#No_error` tag remains uninfluenced at its “old” value.

If you wish to use the EN/ENO mechanism to switch more block calls “in series”, you can program it as follows:

```
"Block1"(EN := #Enable, ... , ENO => #OK);
"Block2"(EN := #OK, ... );
```

“Block2” is not processed if `#Enable` is `FALSE` or if an error has occurred in “Block1”.

Fig. 9.15 provides a summary of how the enable output ENO and the ENO tag are controlled with a block call.

### 9.6.3 Control statements

The control statements control program branches and loops depending on a condition. The following control statements are used with SCL:

- ▷ IF                    Program branch depending on BOOL value
- ▷ CASE                Program branch depending on INT value
- ▷ FOR                 Program loop with a loop-control tag
- ▷ WHILE              Program loop with a feasibility condition
- ▷ REPEAT            Program loop with an abort condition
- ▷ CONTINUE         Abort current loop
- ▷ EXIT                Leave the program loop

Note: Make sure when using program loops that the cycle monitoring time is not exceeded.

**Fig. 9.15** Schematic for setting of enable output ENO and the ENO tag

Is EN used?					
YES			NO		
Is EN = TRUE?			Block/function being processed		
YES		NO			
Block/function being processed		Block/function not being processed			
Has an error occurred?			Has an error occurred?		
YES		NO			
Tag at the ENO output is set to FALSE		Tag at the ENO output is set to TRUE	Tag at the ENO output is set to FALSE	Tag at the ENO output is set to FALSE	Tag at the ENO output is set to TRUE
"Block-local" ENO tag is set to FALSE		"Block-local" ENO tag remains unchanged	"Block-local" ENO tag remains unchanged	"Block-local" ENO tag is set to FALSE	"Block-local" ENO tag remains unchanged

**IF statement**

The IF statement processes a statement block depending on a Boolean value. Fig. 9.17 shows the principle of operation and the variants of the IF statement.

Example in Fig. 9.16: If the *#Actual\_value* tag is greater than the *#Setpoint* tag, the statements following THEN are processed. Otherwise the comparison for *#Actual\_value* less than *#Setpoint* is carried out and, if fulfilled, processing of the statements following ELSIF is carried out. If neither of the two comparisons is fulfilled, the statements following ELSE are processed.

```

5 //-----
6 // IF statement
7 //-----
8 #greater_than := FALSE; #less_than := FALSE; #equal := FALSE;
9 IF #Actual_value > #Setpoint
10 THEN #greater_than := TRUE;
11 ELSIF #Actual_value < #Setpoint
12 THEN #less_than := TRUE;
13 ELSE #equal := TRUE;
14 END_IF;
15

```

**Fig. 9.16** Example of an IF statement

**CASE statement**

You can use the CASE statement to process one or more sequences of statements depending on a fixed-point value. Fig. 9.18 shows the principle of operation of the CASE statement.

## Control statement IF

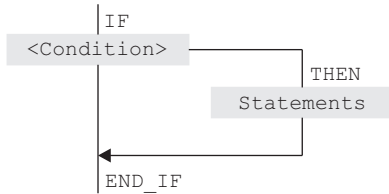
The control statement IF processes a program section <Statements> depending on a Boolean value <Condition>. <Condition> can be a binary tag or an expression with a Boolean result.

**Simple IF branch**

```
IF <Condition>
  THEN <Statements>;
END_IF;
```

If <Condition> has the value TRUE, the instruction block following THEN is processed.

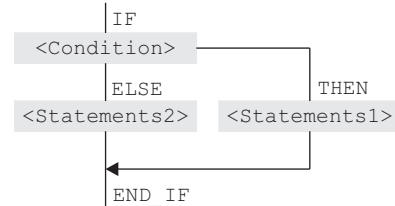
If <Condition> has the value FALSE, processing of the program is continued following END\_IF.

**IF branch with ELSE**

```
IF <Condition>
  THEN <Statements1>;
  ELSE <Statements2>;
END_IF;
```

If <Condition> has the value TRUE, the instruction block following THEN is processed.

If <Condition> has the value FALSE, the instruction block following ELSE is processed.

**Nested IF branch**

```
IF <Condition1>
  THEN <Statements1>;
  ELSIF <Condition2>
    THEN <Statements2>;
  ELSE <Statements3>;
END_IF;
```

A further condition is scanned by ELSIF ... THEN if the preceding condition is not fulfilled.

The ELSIF ... THEN instruction can be inserted cascaded: An ELSIF scan can again follow ELSIF ... THEN.

ELSE and the subsequent statements can also be omitted.

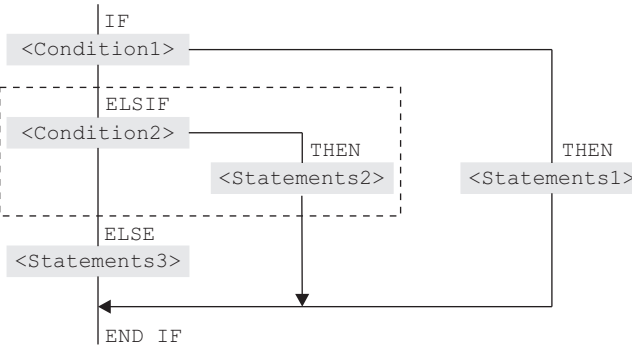
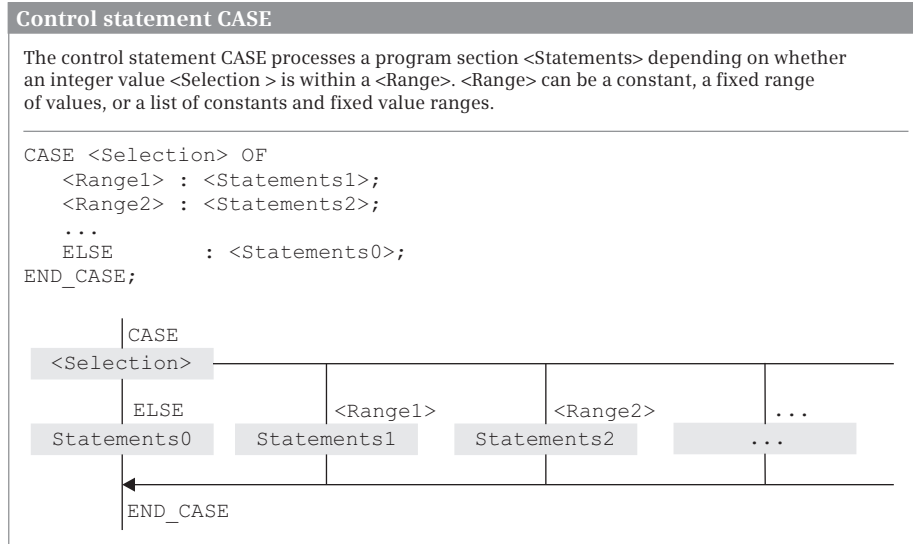


Fig. 9.17 Principle of operation of the IF branch



**Fig. 9.18** Principle of operation of the CASE branch

*Selection* is an operand or expression with a fixed-point data type. If *Selection* has the value of *Range1*, the *Statements1* are processed and then the program execution continues after END\_CASE. If *Selection* has the value of *Range2*, the *Statements2* are processed, etc.

If no value corresponding to the selection is present in the list of values, the *Statements0* following ELSE are processed. The ELSE branch can also be omitted.

The list of values with *Range1*, *Range2*, etc. consists of fixed-point constants.

Various expressions are possible for a component in the list of values:

- ▷ A single fixed-point number
- ▷ A range of fixed-point numbers (e.g. 15..20)
- ▷ A list of fixed-point numbers and fixed-point numerical ranges (e.g. 21,25,30..33).

Each value must only be present once in the list of values.

```

16 //-----
17 // CASE statement
18 //-----
19 CASE #ID OF
20 0 : #Error_number := 0;
21 1,3,5 : #Error_number := #ID + 128;
22 6..10 : #Error_number := #ID;
23 ELSE #Error_number := 16#7F;
24 END_CASE;
25

```

**Fig. 9.19** Example of CASE statement

CASE statements can be nested. A CASE statement can be present instead of a statement block in the selection table of a CASE statement.

Example in Fig. 9.19: A value is assigned to the `#Error_number` tag depending on the assignment of the `#ID` tag.

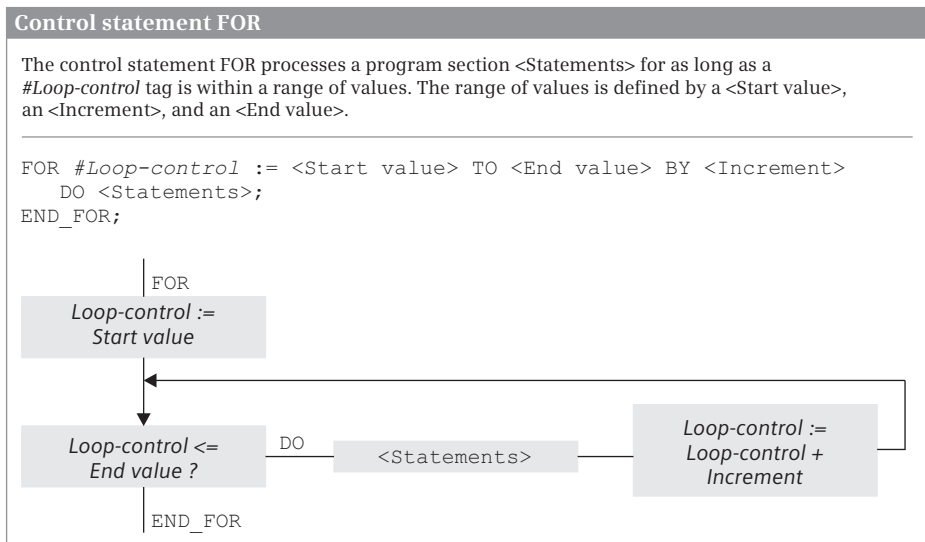
### FOR statement

Using the FOR statement, a program loop is repeatedly processed as long as a loop-control tag is within a defined range of values. Fig. 9.20 shows the principle of operation of the FOR statement.

A `<Start value>` is assigned to the `#Loop-control tag` in the start assignment. You define the loop-control tag yourself; it must be a tag with data type SINT, INT, or DINT. `<Start value>` is any expression with the data type SINT, INT, or DINT, as are `<End value>` and `<Increment>`.

`#Loop-control tag` is set to the start value at the beginning of loop processing. The end value and increment are calculated at the same time and “frozen” (a change in these values during loop processing has no effect on the processing of the loop). The abort condition is subsequently scanned and – if it is not fulfilled – the program loop is processed.

Each time the loop is executed, `#Loop-control tag` is increased by one increment (with positive increment) or decreased by one increment (with negative increment). Specification of *BY Increment* can be omitted; +1 is then used as the increment. If `#Loop-control tag` is outside the range of start value and end value, program execution is continued following `END_FOR`.



**Fig. 9.20** Principle of operation of the FOR loop



The last execution of the loop is carried out with the end value or with the value <End value> minus <Increment> if the end value is not reached exactly. Following a completely executed program loop, the loop-control tag has the value of the last loop plus <Increment>.

FOR loops can be nested: Further FOR loops with other loop-control tags can be programmed within the FOR loop. The current program execution can be aborted in the FOR loop using CONTINUE; EXIT terminates the complete FOR loop processing.

Example in Fig. 9.21: In a *#Current* data field with 16 components, the maximum value is searched for. In the FOR loop, the index tag *#Index* runs through the values 1 to 16. On each cycle, a field component *#Current[#Index]* is compared with the already saved value *#MaxValue*. If *#MaxValue* is smaller, the value of the component *#Current[#Index]* is imported.

```

26 //-----
27 // FOR statement
28 //-----
29 #MaxValue := 0;
30 FOR #Index := 1 TO 16 DO
31   IF #MaxValue < #Current[#Index] THEN #MaxValue := #Current[#Index];
32   END_IF;
33 END_FOR;
34

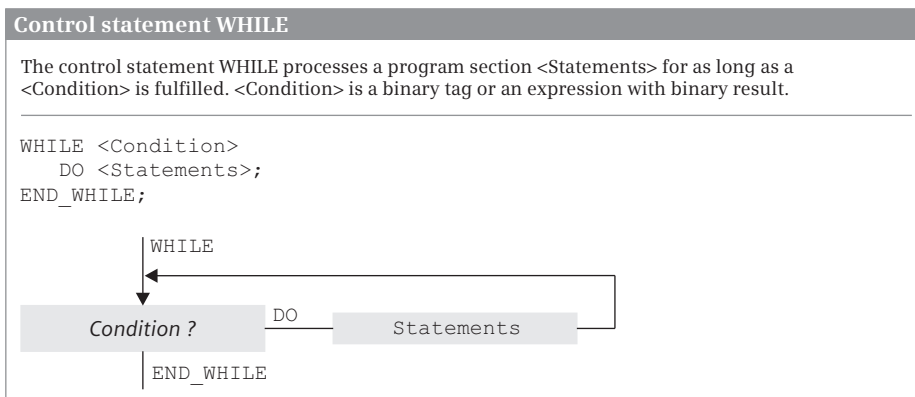
```

**Fig. 9.21** Example of the FOR statement

## WHILE statement

The WHILE statement is used to repeatedly process a program loop for as long as a feasibility condition is fulfilled. Fig. 9.22 shows the principle of operation of the WHILE statement.

<Condition> is an operand or expression with data type BOOL. The statements following DO are repeatedly processed for as long as <Condition> is TRUE.



**Fig. 9.22** Principle of operation of the WHILE loop

<Condition> is scanned prior to each loop processing. If the value is FALSE, program execution is continued following END\_WHILE. This can also already be the case prior to the first loop (the statements in the program loop are not processed in this case).

WHILE loops can be nested: Further WHILE loops can be programmed within a WHILE loop.

The current program execution can be aborted in the WHILE loop using CONTINUE; EXIT terminates the complete WHILE loop processing.

Example in Fig. 9.23: The data block %DB90 is searched word-by-word from the data word DBW16 for the bit pattern 16#FFFF. For every loop cycle, the loop-control tag #Offset is increased by 2 (bytes). Loop processing ends when the bit pattern is found. The #Quantity tag specifies the word in which the bit pattern is found.

```

35 //-----
36 // WHILE statement
37 //-----
38 #Offset := 0;
39 WHILE PEEK_WORD(area:=16#84, dbNumber:=90, byteOffset:=16 + #Offset)
40     <> 16#FFFF DO
41     #Offset := #Offset + 2;
42 END_WHILE;
43 #Quantity := #Offset/2 + 1;
44

```

**Fig. 9.23** Example of a WHILE loop

### REPEAT statement

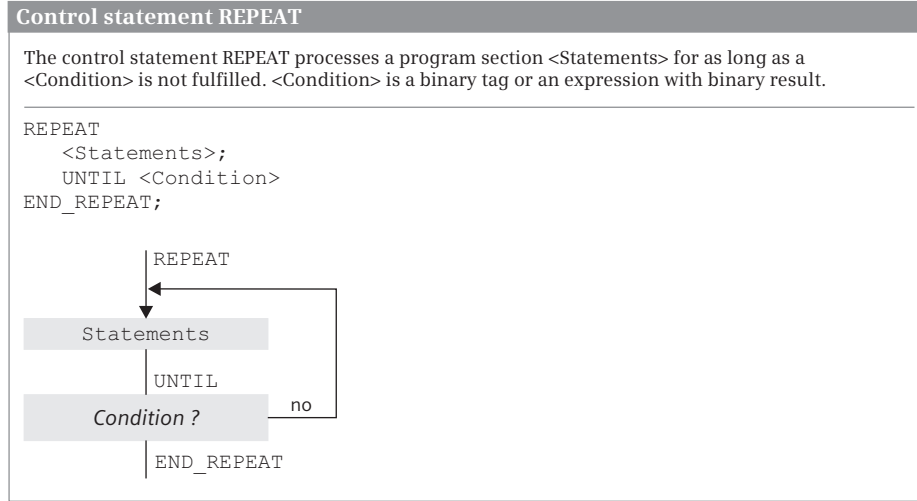
The REPEAT statement is used to repeatedly process a program loop for as long as an abort condition is not fulfilled. Fig. 9.24 shows the principle of operation of the REPEAT statement.

<Condition> is an operand or expression with data type BOOL. The statements following REPEAT are repeatedly processed for as long as <Condition> is FALSE. <Condition> is scanned after each loop processing. If the value is TRUE, program execution is continued following END\_REPEAT. The program loop is executed at least once, even if the abort condition is fulfilled right from the start.

REPEAT loops can be nested: Further REPEAT loops can be programmed within a REPEAT loop.

The current program execution can be aborted in the REPEAT loop using CONTINUE; EXIT terminates the complete REPEAT loop processing.

Example in Fig. 9.25: In the data block “Data.SCL”, the *Measurement* data field is searched. The search ends as soon as a component has the value 16#FFFF. The index of the found field component is then in the loop-control tag #k.



**Fig. 9.24** Principle of operation of the REPEAT control statement

```

45 //-----
46 // REPEAT statement
47 //-----
48 #k := 0;
49 REPEAT
50   #k := #k + 1;
51   UNTIL "Data90".Measurement[#k] = 16#FFFF
52 END_REPEAT;
53

```

**Fig. 9.25** Example of the REPEAT statement

### CONTINUE statement

CONTINUE finishes the current program execution in a FOR, WHILE, or REPEAT loop. Fig. 9.26 shows the principle of operation of the CONTINUE statement.

Following execution of CONTINUE, the conditions for continuation of the program loop are scanned (with WHILE and REPEAT) or the loop-control tag is changed by the increment and checked whether it is still in the control range. If the conditions are fulfilled, execution of the next loop starts following CONTINUE. CONTINUE results in abortion of execution of the loop which directly surrounds the CONTINUE statement.

Example in Fig. 9.27: Memory bits are reset by two nested FOR loops. The first reset memory bit has *#ByteBegin* as byte address and *#BitBegin* as bit address. The last reset memory bit has *#ByteBegin + #Quantity* as byte address and *#BitEnd* as bit address. If in the first byte the loop-control tag *#k* is less than *#BitBegin*, the program begins again with *#k* increased by +1. If in the last byte (in the last run-through of the external FOR loop) the loop-control tag *#k* is greater than *#BitEnd*, the execution of the internal FOR loop ends.

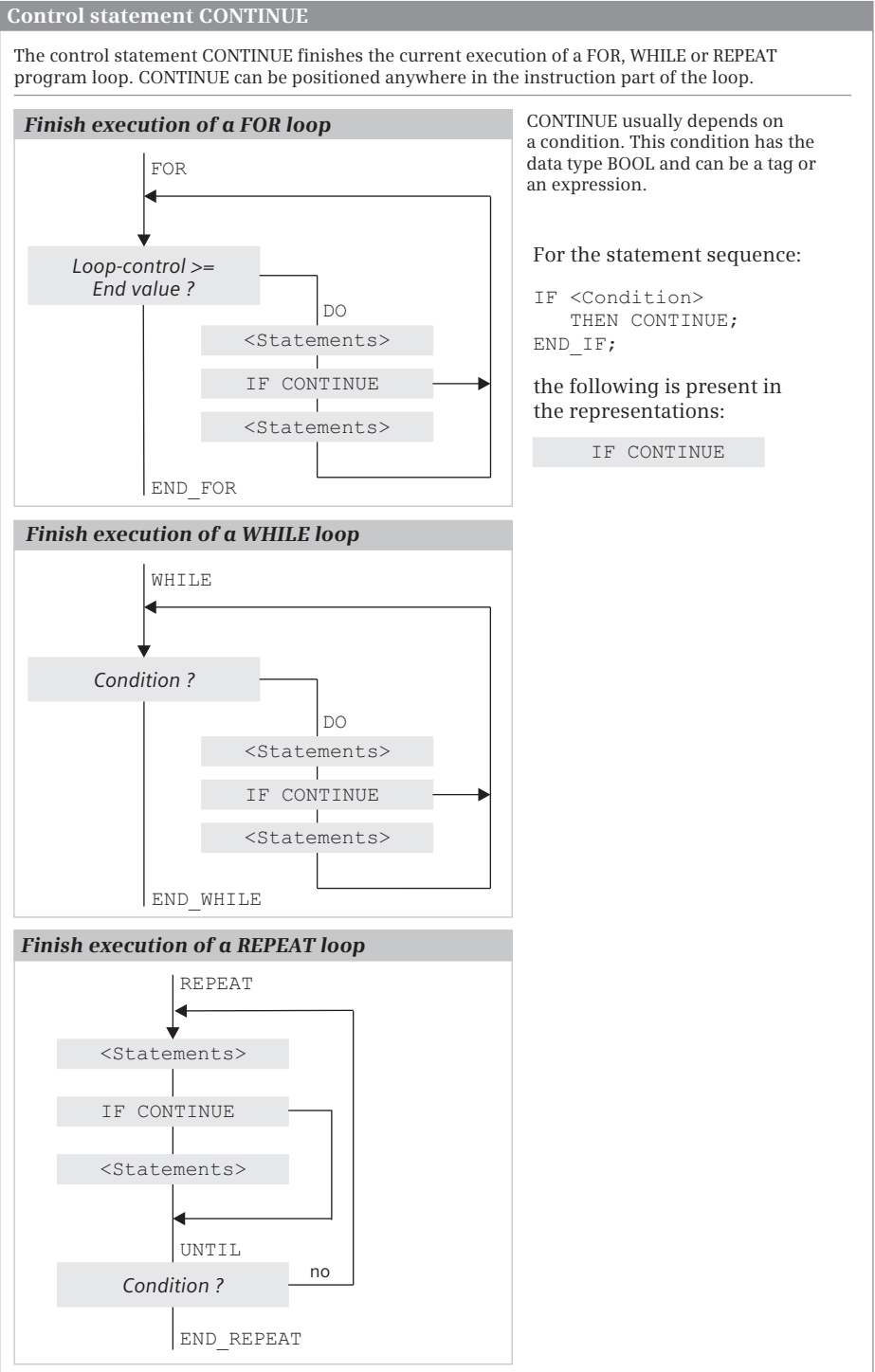


Fig. 9.26 Principle of operation of the CONTINUE control statement

```

54 //-----
55 // CONTINUE statement and EXIT statement
56 //-----
57 FOR #i := 0 TO #Quantity - 1 DO
58   FOR #k := 0 TO 7 DO
59     IF (#i = 0) AND (#k < #BitBegin) THEN CONTINUE; END_IF;
60     IF (#i = #Quantity - 1) AND (#k > #BitEnd) THEN EXIT; END_IF;
61     POKE_BOOL (area:=16#83,
62               dbNumber:=0,
63               byteOffset:=#ByteBegin + #i,
64               bitOffset:=#k,
65               value:=FALSE);
66   END_FOR;
67 END_FOR;
68

```

Fig. 9.27 Example of the CONTINUE and the EXIT statement

### EXIT statement

EXIT leaves a FOR, WHILE, or REPEAT loop at any position independent of conditions. Loop processing is aborted immediately and the program following END\_FOR, END\_WHILE, or END\_REPEAT is processed. Fig. 9.28 shows the principle of operation of the EXIT statement.

EXIT results in leaving of the loop which directly surrounds the EXIT statement. An example is shown in Fig. 9.27.

#### 9.6.4 Block functions

The block functions call and terminate blocks. A detailed description of the block functions is provided in Chapter 12.3 “Calling of code blocks” on page 413. Fig. 9.29 shows examples of the block functions with SCL.

When calling, an SFC system function is treated like an FC function and an SFB system block is treated like an FB function block. The ENO enable output can be added as the last parameter to the block parameter list. The application of the enable input EN is only allowed with function blocks (FB).

#### Terminate block with RETURN

The RETURN statement terminates processing in the current block. The program elements catalog contains RETURN under *Basic instructions > Program control operations*.

Example in Fig. 9.29: The block is left if the ENO tag signals an error (is then FALSE).

#### Call FC block without function value

When calling an FC function, the name of the function is followed by the parameter list in parentheses. All parameters must be supplied. In Fig. 9.29, the “Adder” function call is used as an example.

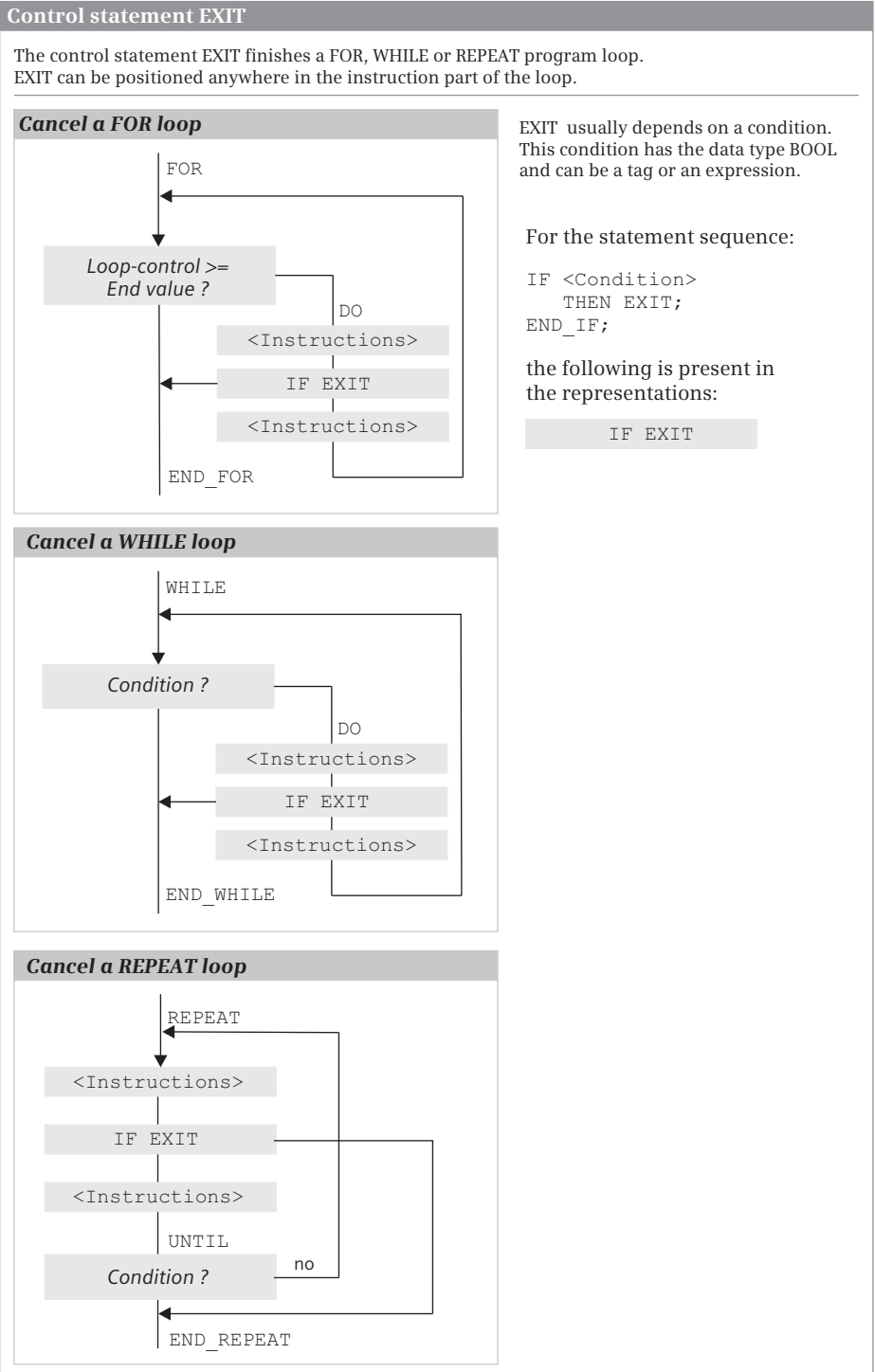


Fig. 9.28 Principle of operation of the EXIT control statement

```

69 //-----
70 // Block functions
71 //-----
72 // Block end statement
73 IF NOT ENO THEN RETURN;
74 END_IF;
75
76 // Block call FC without function value
77 #Adder(Number_1 := #Measurement[1],
78        Number_2 := #Measurement[2],
79        Number_3 := #Measurement[3],
80        Result => #Result[1]);
81
82 // Block call FC with function value
83 #Result[1] := "Adder_2"(Number_1 := #Measurement[1],
84                       Number_2 := #Measurement[2],
85                       Number_3 := #Measurement[3]);
86
87 // Block call FB as single instance
88 #Totalizer_DB(Value1 := #Distance[1],
89              Value2 := #Distance[2],
90              Value3 := #Distance[3],
91              Result => #Position[1]);
92
93 // Block call FB as local instance
94 #Totalizer(Value1 := #Distance[1],
95           Value2 := #Distance[2],
96           Value3 := #Distance[3],
97           Result => #Position[1]);
98
99 // Example of supplying parameters with values
100 #Result[4] :=
101 #Result[3] + LIMIT(MN := #Lower_limit + #Hysteresis,
102                  IN := REAL_TO_INT(#Totalizer.Result),
103                  MX := #Upper_limit);
104

```

Fig. 9.29 Examples of the block functions with SCL

### Call FC block with function value

An FC function with function value can be used like a tag with the data type of the function value, for example in an expression. The parameters of the function follow the function name in parentheses and must all be supplied with values.

In Fig. 9.29, the function value of the “Adder\_2” function is assigned to the #Result[1] tag.

### Call FB function block as single instance

When calling a function block as a single instance, the name of the instance data block is specified. This is followed by the parameter list in parentheses. Not all parameters have to be supplied with values for a function block. You simply omit the parameters which are not supplied from the list.

In Fig. 9.29, the instance data block “Totalizer\_DB” is called as an example. “Totalizer\_DB” is derived from the function block “Totalizer”.

### Call FB function block as local instance

When calling a function block as local instance, the instance name is followed by the parameter list in parentheses. Not all parameters have to be supplied with values for a function block. You simply omit the parameters which are not supplied from the list.

In Fig. 9.29, the local instance *#Totalizer* is called as an example. *#Totalizer* is derived from the function block “*Totalizer*” and is in the instance data block of the calling function block.

### Supplying the block parameters

The input parameters on blocks and functions can be constants, tags, and expressions.

In the last example in Fig. 9.29, the *#Result[4]* tag is assigned a total made up of the *#Result[3]* tag and the function value (return value) of the standard function LIMIT. In this case a function with a function value is used within an arithmetic expression.

The value to be limited by LIMIT is the output parameter of the local instance *#Totalizer* from the example above this one. It is addressed by *#Totalizer.Result* and has the data type REAL. A conversion from REAL to INT must therefore still take place at the IN parameter which expects the data type INT. The total of *#Lower\_limit* and *#Hysteresis* is output as the minimum at the MN parameter.

Additional information about supplying blocks and functions based on blocks with parameters can be found in Chapter 5.4 “Calling blocks” on page 137.

## 9.7 Working with source files

Blocks with the programming language SCL can be programmed as a text file outside the TIA Portal. Any text editor which generates ASCII-coded text can be used for this. Blocks which can be edited further with STEP 7 are generated from these text files – referred to as “source files” or “program sources” – by importing into the TIA Portal and subsequent compilation. Blocks programmed with SCL in the TIA Portal can also be saved as text files.

### 9.7.1 General procedure

A source file can be generated in two different ways: You write the source file completely using a text editor, or you take a block as template and generate a source file by exporting the block. Following editing with the text editor, you import the external source file into the TIA Portal and generate the blocks contained in the source file by compiling. You can then edit these further using the program editor of STEP 7.



### Generating a source file by exporting

In the project tree, select the block(s) from which you wish to generate a source file in the *Program blocks* folder and select the *Copy as text* command from the shortcut menu. The program editor writes the source text into the Windows clipboard.

Open the text editor of your choice – such as the *Notepad* program in Windows – and paste the content of the clipboard into it. Save the SCL source file with the extension *.scl*. Files with this extension can be imported as external source files into the TIA Portal.

### Generating a source file with a text editor

In order to program a block, you must use keywords in a specific sequence in the source file. The program of each block consists of the block header with specification of the block type and properties. With logic blocks, this is followed by the declaration of the interface and the actual program. With data blocks and PLC data types you specify the data operands or data types after the block header.

If the source file contains blocks which are called in the source file or if data operands are accessed, you should observe a specific sequence in the source file. The blocks or data operands should be located before the position of use in the source file. You can also call blocks in the source file which are present in the *Program blocks* folder or use system blocks and functions from the program elements catalog.

A source file can contain several blocks and these can be logic or data blocks as well as PLC data types. You export and import PLC tags separate from the source file (see Chapter 6.2.4 “Exporting and importing a PLC tag table” on page 181). The following chapters describe how to program blocks in a source file.

### Importing an external source file

To import an external source file, open the *External source files* folder in the project tree and double-click on *Add new external file*. In the dialog box, navigate to the storage location, select the source file and import it by clicking the *Open* button.

The source file is saved in the *External source files* folder.

### Editing an external source file in the TIA Portal

As preparation for editing an external source file in the TIA Portal, you must link the file extension *.scl* to a text editor. To do this, open the Windows Explorer, navigate to the source file, and select the *Properties* dialog from the shortcut menu of the source file. In the *General* tab, click on *Change* in the *File type* area. Under *Open with*, select the editor which you wish to link to the file extension *.scl*.

You can then edit the source file using the linked editor by double-clicking on it in the *External source files* folder.

## Generating the blocks of an external source file

To transfer the blocks from the source file to the *Program blocks* folder, select a source file in the *External source files* folder and then the *Generate blocks* command from the shortcut menu. Acknowledge the message which may appear informing that existing blocks will be overwritten. The generated blocks are imported into the *Program blocks* folder. The result of the generation is shown by STEP 7 in the inspector window in the *Info > Compile* tab. Note that these messages refer to the source file.

It is recommendable to compile the blocks imported from the source file prior to further processing in the TIA Portal.

### 9.7.2 Programming a logic block in the source file

Table 9.10 shows which keywords you require for block programming and the sequence in which the keywords are used.

#### Block header and block properties

Programming a logic block commences with the keyword for the block type and with the specification of the block name in quotation marks. On import, the block is assigned the first free number of the block type. This can be changed in the block properties after import.

An organization block has the name of the event class (e.g. “Cyclic interrupt”, see Table 5.5 on page 154). On import, the organization block is then assigned the first free number of the event class (e.g. 30 for event class “Cyclic interrupt”). Only one organization block can be imported to a source file per event class.

In the case of functions, you specify the data type of the function value following the addressing; example: FUNCTION “FC\_name” : INT. If the function does not have a function value, the data type is called VOID.

The block title and the block commentary are entered in the block properties under *General > Information*.

The data for the block properties is optional. You simply omit the surplus data together with the keywords.

You activate the block attribute *Optimized block access* by specifying {S7\_Optimized\_Access := 'TRUE'}.

The keyword KNOW\_HOW\_PROTECT protects the block from unwanted access. You can no longer cancel this protection, in contrast to block protection with password in the TIA Portal.

#### Block interface

The block interface contains the definition of the block parameters and block-local tags. You cannot program every declaration section in every block (see Table 9.10). If you do not use a declaration section, omit it including the keywords.

**Table 9.10** Keywords for logic blocks

Section	Keyword	Meaning
Block type	ORGANIZATION_BLOCK " <i>OB_name</i> " FUNCTION_BLOCK " <i>FB_name</i> " FUNCTION " <i>FC_name</i> " : <i>Data type</i>	Start of an organization block Start of a function block Start of a function
Header	TITLE = <i>block title</i> //Block comment	Block title in the block properties Block comment in the block properties
	{S7_Optimized_Access := 'TRUE'} KNOW_HOW_PROTECT	Optimized block access: selected Know-how protection (cannot be canceled)
	NAME : <i>Block name</i> FAMILY : <i>Block family</i> AUTHOR : <i>Created by</i> VERSION : <i>Version</i>	Block property: Block name Block property: Block family Block property: Created by Block property: Block version
Declaration	VAR_INPUT name : <i>Data type</i> := <i>Default setting</i> ; *) END_VAR	Input parameters (for FC, FB, and some OBs)
	VAR_OUTPUT name : <i>Data type</i> := <i>Default setting</i> ; *) END_VAR	Output parameters (for FC and FB)
	VAR_IN_OUT name : <i>Data type</i> := <i>Default setting</i> ; *) END_VAR	In-out parameters (for FC and FB)
	VAR name : <i>Data type</i> := <i>Default setting</i> ; *) END_VAR	Static local data (only with FB)
	VAR_TEMP name : <i>Data type</i> := <i>Default setting</i> ; *) END_VAR	Temporary local data
Program	BEGIN	Beginning of the block program
	Program statement;	Termination of each statement with semicolon
	//Line comment	Line comment up to end of line, also programmable following statements
	(* Block comment *)	Block comment, can extend over several lines
Block end	END_ORGANIZATION_BLOCK END_FUNCTION_BLOCK END_FUNCTION	End of an organization block End of a function block End of a function

\*) Superimposing of data types with the keyword AT is additionally possible (see text)

The declaration of a tag consists of the name, the data type, possibly a default setting, and an optional tag comment. Example:

```
Quantity : INT := +500; //Units per batch
```

If a name contains special characters such as a space, it must be given in quotation marks. Not all tags can have default values, e.g. default values are not possible for the temporary local data.

The sequence of individual declaration sections is defined as shown in Table 9.10. The sequence within a declaration section is optional. If you combine tags with data type `BOOL` and also combine byte-wide tags with data types `BYTE` and `CHAR`, you can minimize the memory requirements for blocks with non-optimized block access.

The superimposition of data types with the keyword `AT` is programmed directly after the declaration of the tags to be overlaid. The schema is as follows: `var_new AT var_old : new_data` type. Example:

```
VAR_INPUT
    Date           : DTL;
    Byte array AT date : ARRAY [1..12] OF BYTE;
END_VAR
```

You can now address the total tag in the program of the block using `#Date` or individual components such as the day using `#Byte array[3]`. Data type superimposition is possible for blocks with non-optimized block access and is described in Chapter 4.3.3 “Overlaying tags (data type views)” on page 93.

### Program section

The program section of a logic block starts with the keyword `BEGIN` and ends with the keyword for the block end.

No distinction is made between upper and lower case when compiling. The syntax of an SCL statement is described in Chapter 9.1.2 “SCL statements and operators” on page 286. You can enter one or more spaces or tabulators between operation and operand. To achieve a clearer layout of the source text, for example to increase legibility, you can enter any spaces and/or tabs between the words.

You must conclude every statement by a semicolon. You can also program several statements per line, each separated by a semicolon. Following the semicolon you can specify a statement comment, separated by two slashes; this extends up to the end of the line.

A line comment commences with two slashes at the start of the line. It extends up to the end of the line. A block comment is started by a round left parenthesis and asterisk and finished by an asterisk and round right parenthesis. A block comment can extend over several lines.

If the source file contains blocks which are called in the source file or if data operands are accessed, you should observe a specific sequence in the source file. The blocks or data operands should be located before the position of use in the source file.

When calling a block, you enter the block parameters in round parentheses, each separated by a comma. Make sure that the transferred block parameters are listed in the same order as they have been declared in the called block. The same applies to functions with parameter list from the program elements catalog.

Fig. 9.30 shows an example of an SCL source file for a function block with the associated instance data block.

```

FUNCTION_BLOCK FIFO
TITLE=Intermediate memory for 4 values

AUTHOR : Berger
FAMILY : Book1200
NAME : FIFO_SCL
VERSION : 01.00

//Example of a function block
VAR_INPUT
  Import : BOOL := FALSE; //Import with positive edge
  Input value : REAL := 0.0; //Input in data format REAL
END_VAR

VAR_OUTPUT
  Output value : REAL := 0.0; //Output in data format REAL
END_VAR

VAR
  Value1 : REAL := 0.0; //First saved REAL value
  Value2 : REAL := 0.0; //Second value
  Value3 : REAL := 0.0; //Third value
  Value4 : REAL := 0.0; //Fourth value
  Edge trigger flag : BOOL := FALSE; //Edge trigger flag for importing
END_VAR

BEGIN
//Control is implemented with positive edge on #Import
//
IF #Import = TRUE AND #Edge trigger flag = FALSE THEN
#Output value := #Value4;
#Value4 := #Value3; //Transfer starting with the last value
#Value3 := #Value2;
#Value2 := #Value1;
#Value1 := #Input value;
END_IF;

#Edge trigger flag := #Import; //Update edge trigger flag

END_FUNCTION_BLOCK

DATA_BLOCK DB_FIFO
TITLE = Instance data block for "FIFO"
//Example of an instance data block

AUTHOR : Berger
FAMILY : Book1200
NAME : FIFO_Dat
VERSION : 01.00

"FIFO" //Instance for the FB "FIFO"

BEGIN
  Value1 := 1.0; //Individual default setting
  Value2 := 1.0; //of selected values
END_DATA_BLOCK

```

**Fig. 9.30** Example of an SCL source file

### 9.7.3 Programming a data block in the source file

Table 9.11 shows which keywords you require for block programming and the sequence in which the keywords are used.

**Table 9.11** Keywords for data blocks

Section	Keyword	Meaning
Block type	DATA_BLOCK " <i>DB_name</i> "	Start of a data block
Header	TITLE = <i>block title</i> //Block comment	Block title Block comment
	{S7_Optimized_Access := 'TRUE'} KNOW_HOW_PROTECT UNLINKED READ_ONLY	Optimized block access: selected Know-how protection (cannot be canceled) Block attribute: not executable Block attribute: read-only
	NAME : <i>Block name</i> FAMILY : <i>Block family</i> AUTHOR : <i>Created by</i> VERSION : <i>Version</i>	Block property: Block name Block property: Block family Block property: Created by Block property: Block version
Declaration	STRUCT name : Data type := Default setting; END_STRUCT	for a global data block
	Data type_name	alternatively for a type data block
	FB_name	alternatively for an instance data block
Initialization	BEGIN name := Default setting;	Assignment with individual initial values
Block end	END_DATA_BLOCK	End of a data block

#### Block header and block properties

A data block commences with the keyword DATA\_BLOCK and with specification of the block name in quotation marks. On import, the first free data block number is assigned. It can be changed later in the block properties.

The block title and the block commentary are entered in the block properties under *General > Information*.

The data for the block properties is optional. You simply omit the surplus data together with the keywords.

You activate the block attribute *Optimized block access* by specifying {S7\_Optimized\_Access := 'TRUE'}.

The keyword KNOW\_HOW\_PROTECT protects the block from unwanted access. You can no longer cancel this protection, in contrast to block protection with password in the TIA Portal.

With the keyword UNLINKED, the *Only store in load memory* attribute is activated. In this way, the data block is not loaded to the work memory.

The keyword READ\_ONLY activates the *Data block write-protected in device* attribute. If this attribute is activated for a data block, a program can only read from this data block.

### Block interface

The block interface contains the declaration of the data operands. With a global data block, you program the data operand here, with a type data block, you specify the assigned PLC data type, and with an instance data block, you assign the associated function block.

The declaration of a tag in a global data block consists of the name, the data type, possibly a default setting, and an optional tag comment. Example:

```
Quantity : INT := +500; //Units per batch
```

The tag order can be random. If you combine tags with data type BOOL and also combine byte-wide tags with data types BYTE and CHAR, you can minimize the memory requirements for a block with non-optimized block access.

The declaration in a type data block consists only of the specification of the assigned PLC or system data type, e.g. IEC\_COUNTER. Data blocks derived from a system data type are stored in the project tree under *Program blocks > System blocks > Program resources*.

The declaration in an instance data block consists only of the specification of the assigned function block.

### Default setting with start values

The initialization part starts with BEGIN and ends with END\_DATA\_BLOCK. You must specify these keywords even if you do not assign default values to the tags in the initialization part.

By assigning default values to the start values, it is possible to assign individual values to each application (each instance) in the case of type and instance data blocks.

If you do not specify a start value for a data operand, the value from the block interface is used: The default value is retained for a global data block and the default value in the PLC data type or in the function block then applies to a type or instance data block.

### 9.7.4 Programming a PLC data type in the source file

Table 9.12 shows which keywords you require for data type programming and the sequence in which the keywords are used.

**Table 9.12** Keywords for PLC data types

Section	Keyword	Meaning
Block type	TYPE "Type_name"	Start of a PLC data type
Header	TITLE = Data type title //Data type comment	Data type title Data type comment
Declaration	STRUCT name : Data type := Default setting; END_STRUCT	Declaration of data type components
Block end	END_TYPE	End of the PLC data type

#### Block header

A PLC data type (UDT, user data type) starts with the keyword TYPE and with the specification of the data type name in quotation marks. This can be followed by a data type title and a data type commentary, which are entered in the properties of the PLC data type under *Information*. This information is optional.

#### Declaration of data type

The declaration part contains the definition of the data type components. The structure of a PLC data type corresponds to that of a data structure STRUCT.

The declaration of a component consists of the name, the data type, possibly a default setting, and an optional comment. Example:

```
Quantity : INT := +500;    //Units per batch
```



## 10 Basic functions

This chapter describes the basic functions largely independent of the programming language selected. Binary logic operations are an exception, since the differences between the programming languages are greatest with these functions. The Chapters 7 “Ladder logic LAD” on page 209, 8 “Function block diagram FBD” on page 246, and 9 “Structured Control Language SCL” on page 284 describe how you can program the functions using the individual programming languages and what special features exist.

### 10.1 Binary logic operations

#### 10.1.1 Introduction

Binary logic operations link the signal states of binary tags in accordance with AND, OR, and exclusive OR. In the LAD programming language, they are implemented by the series and parallel connection of contacts, in FBD by the corresponding function boxes, and in SCL by logical expressions.

Binary logic operations can be used together with all binary tags. Addressing of the binary tags can be absolute or symbolic. A binary tag has the data type BOOL and can have signal state “1” or “0”. In SCL, the designations TRUE and FALSE are common.

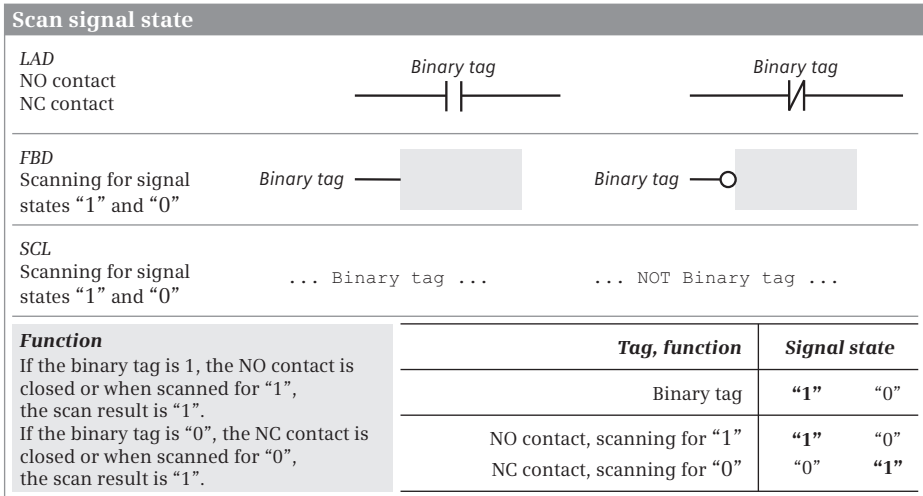
The result of a binary logic operation can be processed further as follows:

- ▷ Control of a binary tag with a binary memory function, e.g. with a simple coil (LAD) or an assignment (FBD, SCL).
- ▷ Control of program execution with a conditional jump or a conditional block call (EN input).
- ▷ Supply of a binary function input or a binary block parameter.

Binary logic operations can be combined together so that, for example, the output of one logic operation can lead to the input of the next one, or series connections can be connected in parallel. Possible combinations are described for LAD in Chapter 7.2.5 “Mixed series and parallel connections” on page 216, for FBD in Chapter 8.2.6 “Mixed binary logic operations” on page 254, and for SCL in Chapter 9.2.6 “Combined binary logic operations” on page 292.

### 10.1.2 Scanning for signal states “1” and “0”, result of the scan

Before the signal state of a binary tag is linked, the binary tag is set to either signal state “1” or scanned for signal state “0” and a scan result is formed. Only the scan result will be linked. The result of logic operation (RLO) is formed from all linked scan results. LAD uses the NC contact to scan for signal state “0”, FBD uses the negation circle at the function inputs, and SCL uses the negation NOT (Fig. 10.1).



**Fig. 10.1** Scanning for signal state “1” (NO contact) an “0” (NC contact)

There are signal transmitters, e.g. pushbuttons, which when actuated issue signal state “1” (pushbuttons with NO function) or signal state “0” (pushbuttons with NC function). The scans for signal states “1” and “0” can be used to take into account this difference in creating the program. Further information can be found for LAD in Chapter 7.2.2 “Consideration of sensor type in ladder logic” on page 213, for FBD in Chapter 8.2.2 “Taking account of the sensor type in the function block diagram” on page 251, and for SCL in Chapter 9.2.1 “Scanning for signal states “1” and “0”” on page 288.

### 10.1.3 Negating the result of the logic operation, NOT contact

#### Ladder logic

The NOT contact negates the “current flow” within a current path. If the current path has “current” prior to the NOT contact, no more “current” flows following the NOT contact and vice versa (Fig. 10.2).

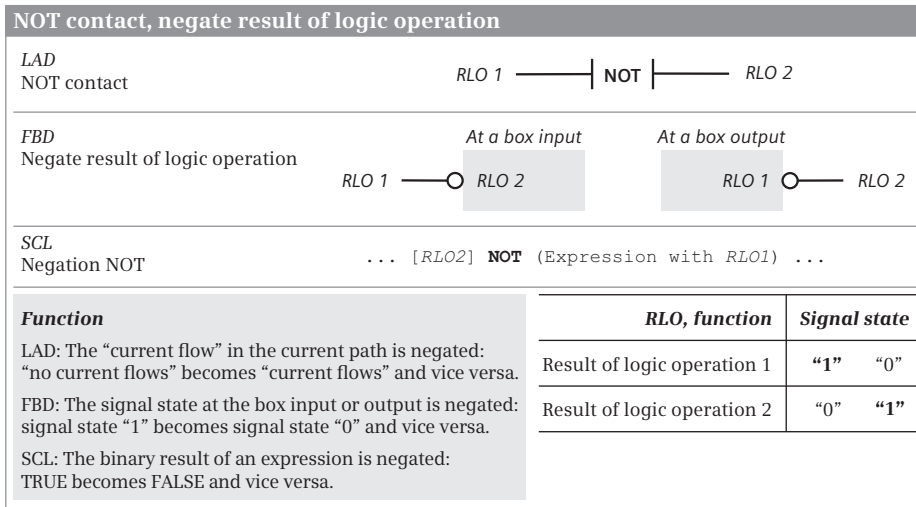


Fig. 10.2 NOT contact, negate result of logic operation

### Function block diagram

The negation circle at the input or output of a function symbol negates the result of logic operation (Fig. 10.2). You can

- ▷ apply the negation to the scan of a binary tag; this then corresponds to scanning for signal state “0” (see above),
- ▷ set the negation between two binary functions (this corresponds to negation of the result of logic operation), or
- ▷ position the negation at the output of a binary function (e.g. if you wish to set or reset a binary tag if the logic operation is not fulfilled, i.e. the result of logic operation = “0”).

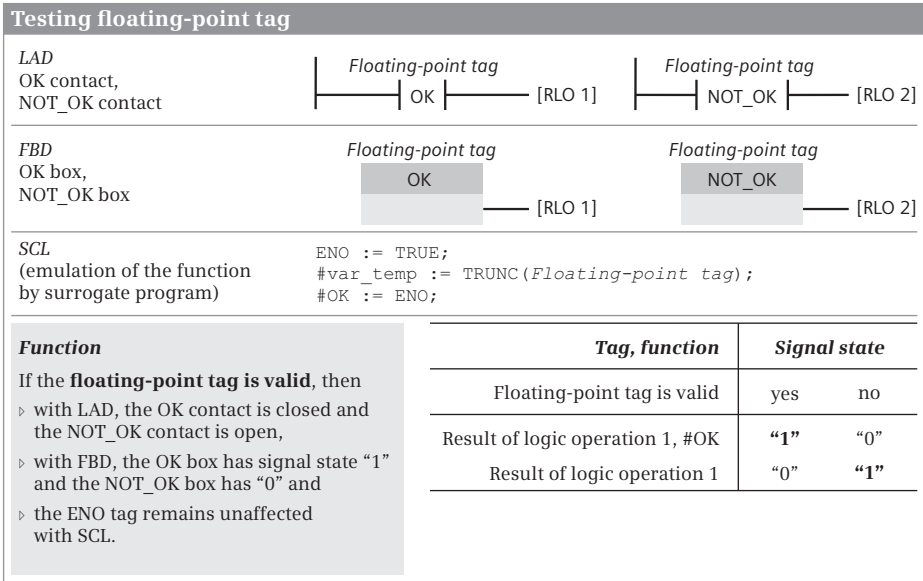
### Structured control language

The NOT negation negates the result of logic operation. You can put NOT in an expression before a tag; the tag is then scanned for signal state “0” and you can write NOT before an expression you have placed in parentheses. Then the result of logic operation of the expression is negated (Fig. 10.2).

#### 10.1.4 Testing floating-point tag, OK contact, OK box

The testing of a tag with data type REAL or LREAL is used to check the validity of the tag value (Fig. 10.3).

Testing with the OK contact or with the NOT\_OK contact is implemented in the ladder logic. The OK contact has signal state “1” if the floating-point tag is valid; the NOT\_OK contact has signal state “1” if the floating-point tag is invalid.



**Fig. 10.3** Test floating-point tag, OK contact, OK box

In the function block diagram there are the OK box and the NOT\_OK box for the scan. The OK box has signal state “1” at the function output if the floating-point tag is valid; the NOT\_OK box has signal state “1” if the floating-point tag is invalid.

In SCL, use an alternative program to test a floating-point tag. A function with a floating-point tag as input signals an error via the ENO tag if the floating-point tag is invalid (see Chapter 12.4.2 “EN/ENO mechanism with SCL” on page 418). On error, ENO is set to FALSE, otherwise the signal state of ENO is unaffected. In the example, the signal state of ENO is assigned to the self-defined tag #OK. You can also directly scan ENO, e.g. with the IF statement.

### 10.1.5 AND function, series connection

The AND function links two or more binary states together and delivers a result of logic operation “1” if all states (all results of the scans) are simultaneously “1”. In all other cases, the result of logic operation is “0”.

In the ladder logic, the AND function is implemented by series connection of contacts; in the function block diagram, this is done with the AND box. SCL uses the logic operator AND or & in connection with binary tags (Fig. 10.4).

Each AND function in the examples is shown with two inputs; the number of inputs of an AND function is theoretically unlimited. In the examples, the binary tags are scanned directly (for signal state “1”). With a scan for signal state “0” it is necessary to consider the negated signal state of the binary tags in the logic operation according to AND.

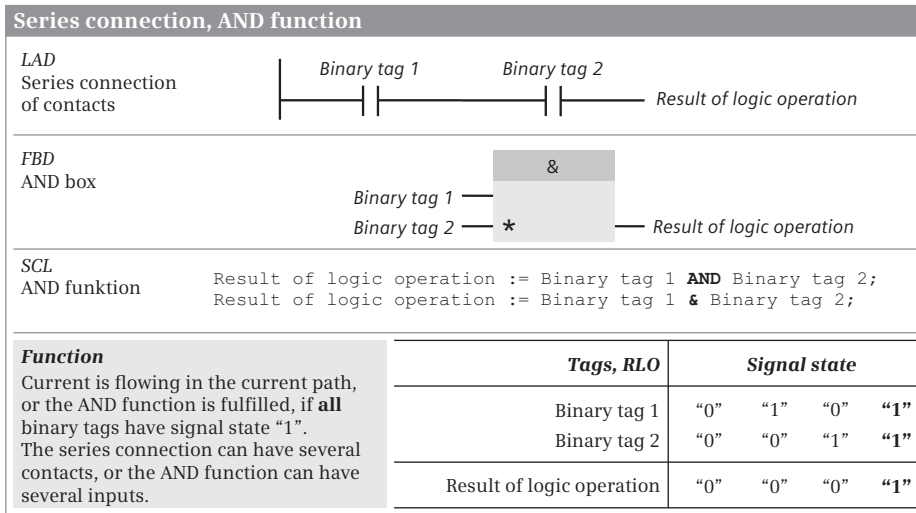


Fig. 10.4 Series connection, AND function

### 10.1.6 OR function, parallel connection

The OR function links two or more binary signal states together and delivers a result of the logic operation "0" if all states (all results of the scans) are simultaneously "0". In all other cases, the result of the logic operation is "1".

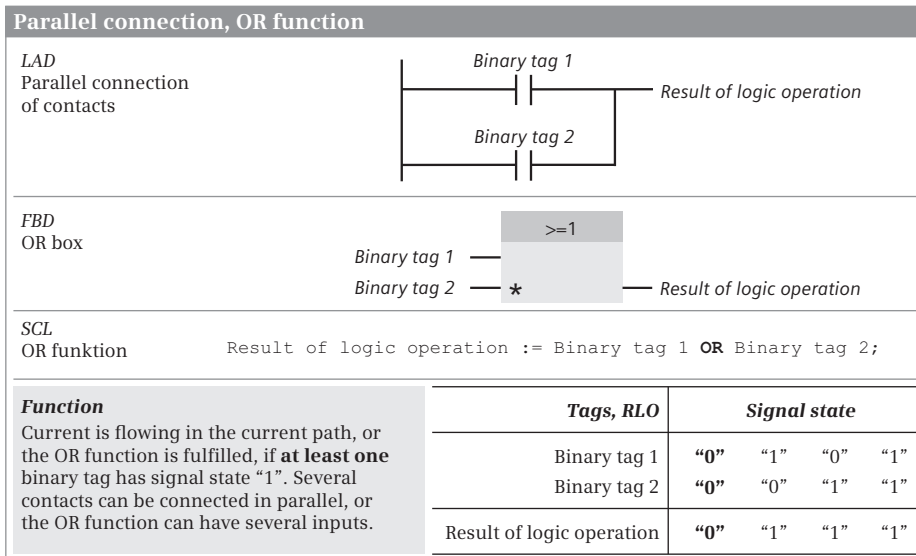


Fig. 10.5 Parallel connection, OR function

In the ladder logic, the OR function is implemented by the parallel connection of contacts; in the function block diagram, this is done with the OR box. SCL uses the logic operator OR in connection with binary tags (Fig. 10.5).

Each OR function in the examples is shown with two inputs; the number of inputs of an OR function is theoretically unlimited. In the examples, the binary tags are scanned directly (for signal state “1”). With a scan for signal state “0” it is necessary to consider the negated signal state of the binary tags in the logic operation according to OR.

### 10.1.7 Exclusive OR function, non-equivalence function

The exclusive OR function links two or more binary signal states together and delivers a logic operation result “1” if an odd number of states (the results of the scans) are simultaneously “1”. In all other cases, the result of the logic operation is “0”. In special cases where the exclusive OR function has two inputs, it delivers a logic operation result “1” if the two inputs have different signal states.

In the ladder logic, the exclusive OR function is implemented by a combination of series connection and parallel connection of contacts; in the function block diagram, this is done with the exclusive OR box. SCL uses the logic operator XOR in connection with binary tags (Fig. 10.6).

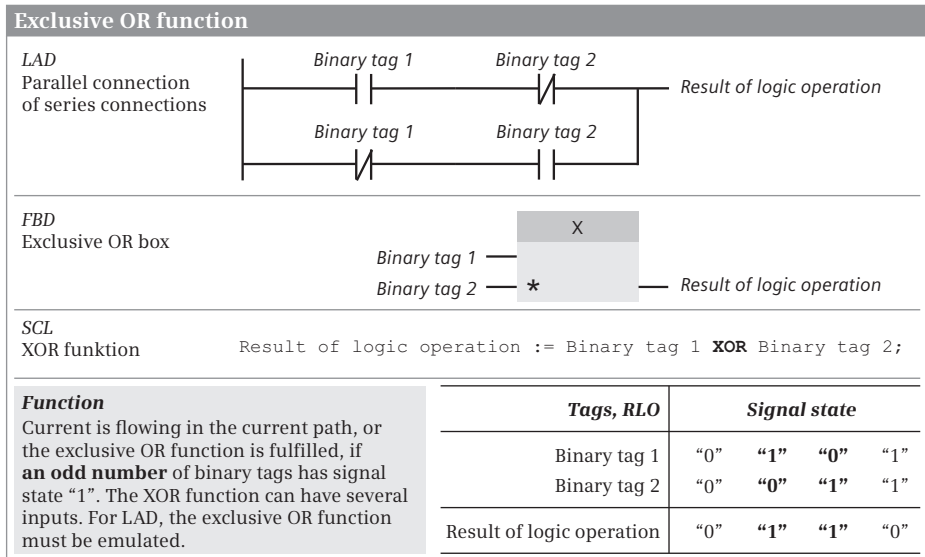


Fig. 10.6 Exclusive OR function

## 10.2 Memory functions

### 10.2.1 Introduction

The memory functions are used together with the binary logic operations in order to influence the signal states of binary tags with the assistance of the result of the logic operation generated by the control processor.

The following memory functions are available:

- ▷ Assignment of the logic operation result
- ▷ Individual set and reset
- ▷ Multiple setting and resetting
- ▷ Dominant setting and resetting with memory boxes

The memory functions can be used together with all binary tags. A result of logic operation can be used to influence several memory functions simultaneously. The result of logic operation does not change during execution of a memory function.

### 10.2.2 Simple and negated coil, assignment

The assignment is used to transfer the result of logic operation to a binary tag. If the result of logic operation is “1”, the binary tag is set to signal state “1”; if it is “0”, the binary tag is set to signal state “0”. In LAD, the assignment is represented by the single coil, in FBD by the assignment box, and in SCL by the assignment operator “:=” (Fig. 10.7).

With the negated assignment, the negated result of logic operation is transferred to a binary tag. If the result of logic operation is “1”, the binary tag is reset to signal

Simple coil, assignment		
LAD Simple coil Negated coil	Binary tag 1 RLO ——— ( ) ——— (RLO)	Binary tag 2 RLO ——— (/) ——— (RLO)
FBD Assignment Negated assignment	Binary tag 1 RLO ——— [=] ——— (RLO)	Binary tag 2 RLO ——— [/=] ——— (RLO)
SCL Assignment Negated assignment	Binary tag 1 := (... RLO ...); Binary tag 1 := NOT (... RLO ...);	
<b>Function</b>	The simple coil or the assignment transfers the result of the logic operation to the binary tag. The negated coil or the negated assignment transfers the negated result of the logic operation to the binary tag. The coil or box can also be positioned in the middle of the logic operation.	
	<b>RLO, tags</b>	<b>Signal state</b>
	Result of logic operation	“0”   “1”
	Binary tag 1	“0”   “1”
	Binary tag 2	“1”   “0”

Fig. 10.7 Simple and negated coil, assignment and negated assignment

state “0”; if it is “0”, the binary tag is set to signal state “1”. The negated assignment is represented in LAD by the negated coil, and in FBD by the negated assignment box. SCL uses the negation NOT to negate a result of logic operation.

In LAD and FBD, the assignment does not influence the result of logic operation so that a logic operation can also be continued after the assignment function.

### 10.2.3 Single set and reset

Single setting sets a binary tag to signal state “1” if the result of logic operation is “1”. The binary tag is not influenced if the result of logic operation is “0”; it remains set if it was set, and remains reset if it was reset. The individual setting is represented in LAD by the set coil, and in FBD by the set box. In SCL, the set function can be emulated with an IF statement (Fig. 10.8).

Single resetting sets a binary tag to signal state “0” if the result of logic operation is “1”. The binary tag is not influenced if the result of logic operation is “0”; it remains set if it was set, and remains reset if it was reset. The individual resetting is represented in LAD by the reset coil, and in FBD by the reset box. In SCL, the reset function can be emulated with an IF statement.

Individual setting and resetting does not affect the result of logic operation in LAD and FBD so that a logic operation can be continued even after the set and reset function.

To make the programming clearer, you should always use the single set and reset function in pairs for a specific binary tag, and only once each. You should also avoid controlling this binary tag in addition by an assignment.

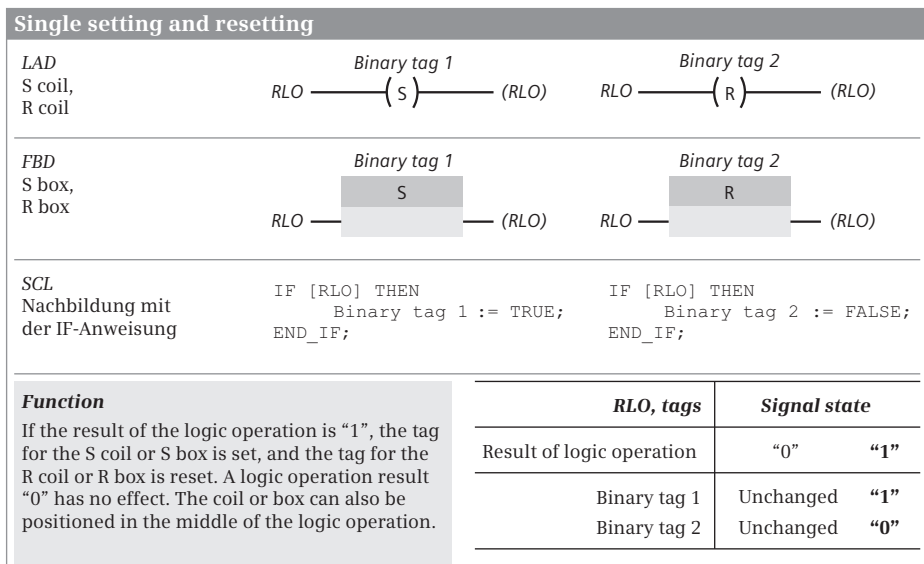


Fig. 10.8 Single setting and resetting



When applying the single memory functions to the same binary tag, the sequence of the arrangement is important since the function processed last is dominant if the set and reset functions are activated simultaneously. For example, if the reset function is processed after the set function, the reset is dominant.

Note that the binary tag used with a single memory function can be reset during the startup by the CPU's operating system. In certain cases, the signal state is retained: this depends on the operand area used (e.g. static local data) and on settings in the CPU (e.g. retentive behavior).

### 10.2.4 Multiple setting and resetting

With multiple setting and resetting, the bits are set in the specified destination area to signal status "1" (SET\_BF) or to signal state "0" (RESET\_BF).

Multiple setting and resetting is shown in the ladder logic as a coil. The binary tag present above the coil indicates the first bit in the destination area. Underneath the coil is the number of bits to be controlled as a constant in the range from 0 to 65 535. Multiple setting and resetting is performed if the coil is triggered with result of logic operation "1". If the result of logic operation is "0", there is no influence on the binary tags in the destination area; then they retain their current signal state (Fig. 10.9).

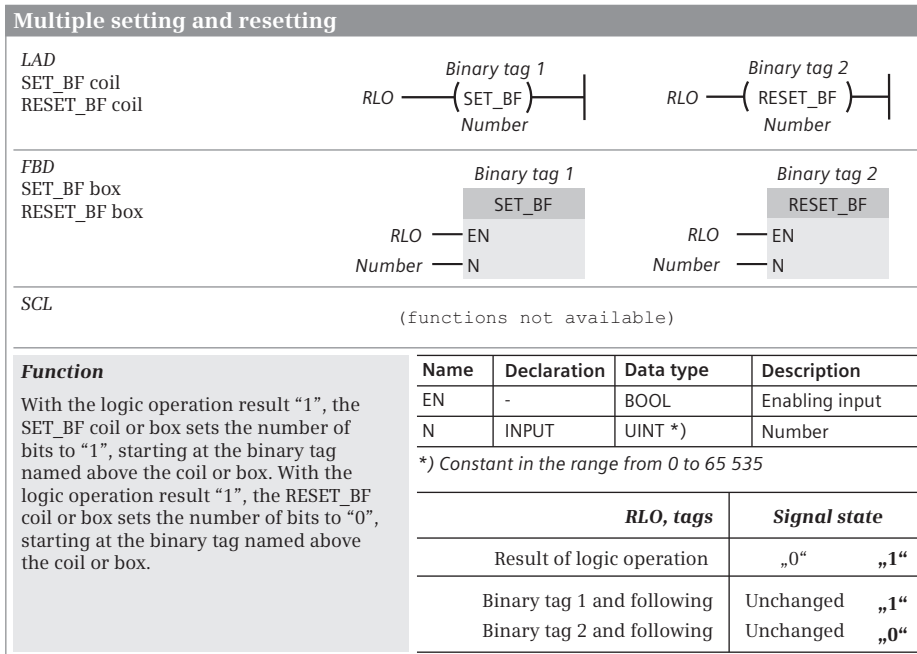


Fig. 10.9 Multiple setting and resetting

Multiple setting and resetting is shown in the function block diagram as a box. The binary tag present above the box indicates the first bit in the destination area. At parameter N is the number of bits to be controlled as a constant in the range from 0 to 65 535. Multiple setting and resetting is performed if the enabling input EN of the box is triggered with RLO “1”. If the result of logic operation is “0”, there is no influence on the binary tags in the destination area; then they retain their current signal state.

The functions SET\_BF and RESET\_BF are not present in SCL. They can be emulated, for example, using the FOR statement. An example is given in the description of the control statements in Section “CONTINUE statement” on page 314.

### 10.2.5 Dominant setting and resetting, memory boxes

The functions of the individual setting and resetting are combined in a memory box. The common binary tag is named above the box. Input S or S1 corresponds to the individual setting, input R or R1 to the individual resetting. The signal state of the binary tag named above the memory function is present at the output Q of the function.

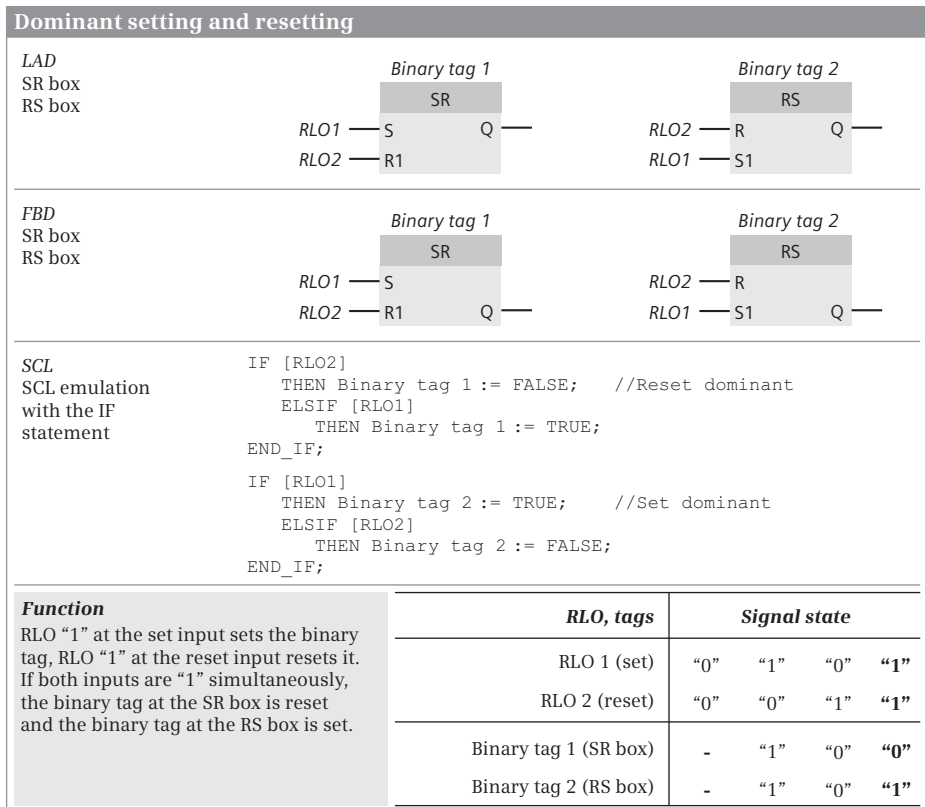


Fig. 10.10 Dominant setting and resetting, memory boxes

There are two versions of the memory function: as SR box (reset dominant) and as RS box (set dominant). In addition to the difference in labeling, the two boxes also differ in the positioning of the set and reset inputs (Fig. 10.10).

A memory function (or, to be more precise: the binary tag named above the memory box), is set when the set input has signal state “1” and the reset input signal state “0”. A memory function is reset when “1” is present at the reset input and “0” at the set input. Signal state “0” at both inputs has no influence on a memory function. If signal state “1” is present simultaneously at both inputs, the two memory functions respond differently: the SR memory function is reset, the RS memory function is set.

With SCL, the dominant memory function can be emulated, for example, by an assignment together with an IF statement.

Note that the binary tag used with a memory function can be reset during the start-up by the CPU's operating system. In certain cases, the signal state of a memory box is retained: this depends on the operand area used (e.g. static local data) and on settings in the CPU (e.g. retentive behavior).

## 10.3 Edge evaluation

### 10.3.1 Functional principle of an edge evaluation

An edge evaluation records the change in a signal state, i.e. the signal edge. A positive (rising) edge is present if the signal changes from state “0” to state “1”. The reverse case is a negative (falling) edge.

The equivalent of an edge evaluation in a circuit diagram is a fleeting contact. If the fleeting contact outputs a pulse when the relay is switched on, this corresponds to the rising edge. A pulse of the fleeting contact when switching off corresponds to a falling edge.

The detection of a signal edge – the change in a signal state – is implemented in the program. When processing an edge evaluation, the CPU compares the current result of logic operation, e.g. the result of scan of an input, with a saved result of logic operation. If the two signal states are different, a signal edge is present.

The saved result of the logic operation (RLO) is in a so-called “edge memory bit” (it need not necessarily be a bit memory). It must be a binary tag whose signal state is available again during the next processing of the edge evaluation (in the next program cycle) and which you do not use at any further point in the program. Suitable binary tags are memory bits, data bits in global data blocks, and static local data bits in function blocks.

These edge memory bits save the “old” RLO, namely the result of the logic operation with which the CPU last processed the edge evaluation. If a signal edge is then present, i.e. if the current RLO is different from the signal state of the edge memory bit, the CPU tracks the signal state of the edge memory bit by assigning it to the RLO

which is now current. During the next processing of the edge evaluation (usually in the next program cycle), the signal state of the edge memory bit is equal to the current RLO (if this has not changed again in the meantime), and the CPU no longer recognizes an edge.

A recognized edge is indicated by the RLO following the edge evaluation. If the CPU recognizes a signal edge, it sets the RLO following the edge evaluation to “1” (“current” is then flowing). If a signal edge is not present, the RLO is equal to “0”.

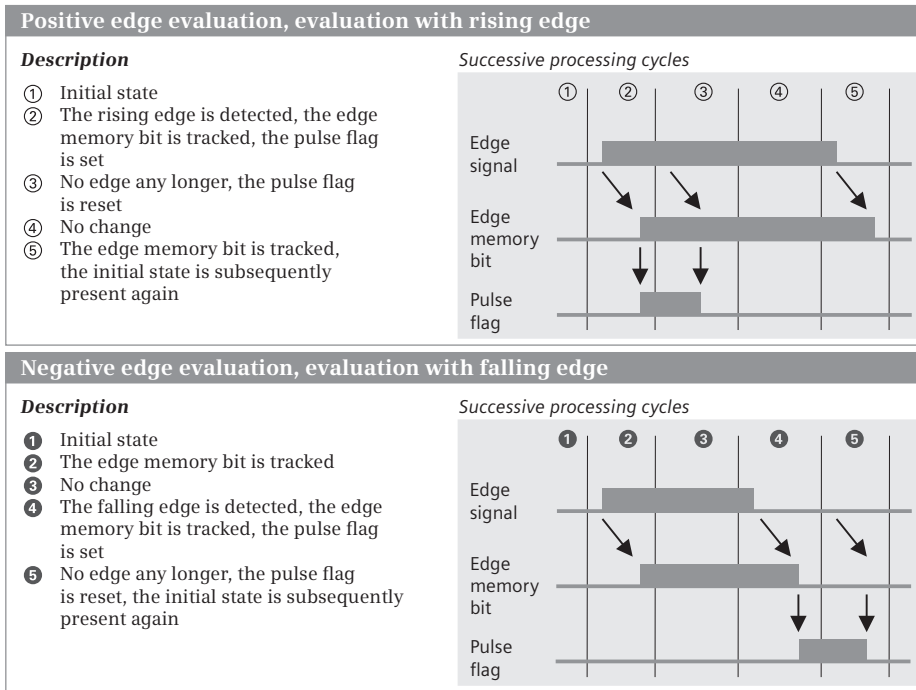
Signal state “1” following an edge evaluation therefore means “Edge detected”. Signal state “1” is only present briefly, usually for only one processing cycle. Since the CPU does not detect an edge during the next processing of the edge evaluation (if the “Input RLO” of the edge evaluation does not change), it sets the RLO to “0” again following the edge evaluation.

You can directly process the RLO following an edge evaluation, e.g. link using binary logic operations, save in a memory function, or assign to a binary tag (a so-called “pulse flag”). A pulse flag is used if the RLO of the edge evaluation is to be also processed at another position in the program; it is quasi the intermediate memory for a detected edge (the fleeting contact in the circuit diagram). Suitable binary tags for the pulse flag are memory bits, data bits in global data blocks, and temporary and static local data bits.

Also take note of the response of edge evaluation when switching on the CPU. If an edge should not be detected, the RLO prior to edge evaluation and the signal state of the edge trigger flag must be the same when switching on. It may be necessary – depending on the desired response and the operand area used – to appropriately set or reset the edge trigger flag during the startup.

Principle of operation of the positive edge (Fig. 10.11): ① In the initial state, the signal being monitored for an edge, the edge trigger flag, and the pulse flag have signal state “0”. ② The edge signal then changes its state from “0” to “1”. The signal state of the edge trigger flag is initially still “0” so that a rising edge is detected and the pulse flag is set to “1”. The edge trigger flag is updated to signal state “1”. ③ The next processing cycle does not show a change in the signal state or edge signal (comparison with signal state of edge trigger flag). The pulse flag is reset to “0”. ④ No changes take place in the next processing cycles. ⑤ If the edge signal is reset to state “0” again, the edge trigger flag is also updated and the initial state is reached. The pulse flag was therefore only set to “1” for one processing cycle.

Functional principle of the negative edge: ① In the initial state, the edge signal, the edge memory bit, and the pulse flag have the signal state “0”. ② The edge then changes its signal state from “0” to “1”. This change is saved in the edge memory bit which is also set to “1”. The pulse flag remains “0” since a falling edge is not present. ③ There is no change in the next processing cycles. ④ The edge signal then changes from “1” to “0”. The edge memory bit initially still has signal state “1” so that a falling edge is detected. The pulse flag is set to “1”, and the edge memory bit tracked to “0”. ⑤ The pulse flag is reset to “0” again. The pulse flag was therefore only set to “1” for the duration of one processing cycle.



**Fig. 10.11** Principle of operation of an edge evaluation in successive cycles

### 10.3.2 Edge evaluation of the result of the logic operation

This edge evaluation generates a pulse when the result of the logic operation changes (with ladder logic: change in the “current flow”). The P\_TRIG box is available for evaluation of a positive edge, and the N\_TRIG box for evaluation of a negative edge.

The positive edge evaluation generates the pulse upon a change in signal state from “0” to “1” (rising edge) at the CLK input, the negative edge evaluation upon a change in signal state from “1” to “0” (falling edge). The pulse is available at the Q output of the edge evaluation.

Fig. 10.12 shows the representation and signal states of edge evaluation. The principle of operation of edge evaluation is described in detail in Fig. 10.11: The edge signal corresponds to the result of logic operation present at the CLK input, the edge trigger flag corresponds to the binary tag named underneath the function, and the pulse flag corresponds to the result of logic operation following the edge evaluation.

In SCL, the edge evaluation can be emulated, for example, with the IF statement. The scan “RLO differs from the signal state of the edge trigger flag” is programmed with an AND function. The statements after THEN are processed only if the signal states differ. The edge trigger flag is then updated so that a signal edge is no longer detected during the next processing.

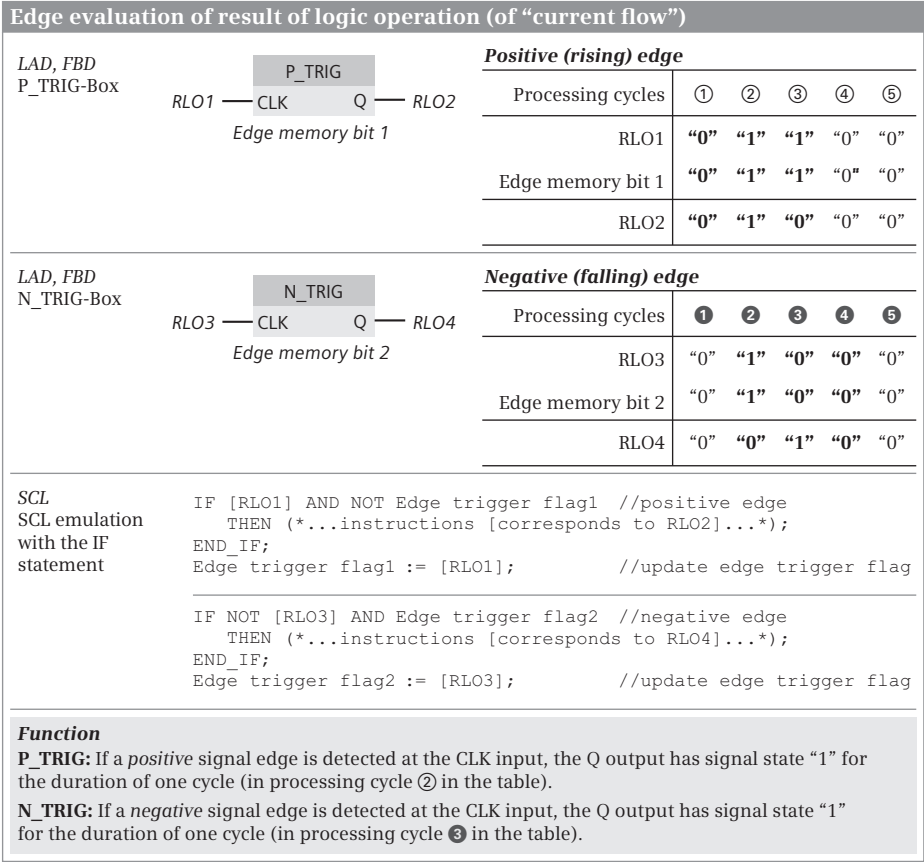


Fig. 10.12 Edge evaluation of result of logic operation (of the “current flow”)

10.3.3 Edge evaluation of a binary tag

The edge evaluation of a binary tag is represented in the ladder logic as a contact, above which the scanned binary tag and below which the edge trigger flag are named. The pulse of the edge evaluation (quasi the signal state of the pulse flag) is connected in series with the result of logic operation of the preceding logic operation. A positive, rising edge is detected by the P contact and a negative, falling edge by the N contact.

In the function block diagram, the binary tag is named above the edge evaluation box, and the edge memory bit below it. The Q output corresponds to the pulse flag. A positive, rising edge is detected by the P box, and a negative, falling edge by the N box.

In SCL, the edge evaluation can be emulated, for example, with the IF statement. The scan “The signal states of the binary tag and edge trigger flag are different” is programmed with an AND function. The statements after THEN are processed only

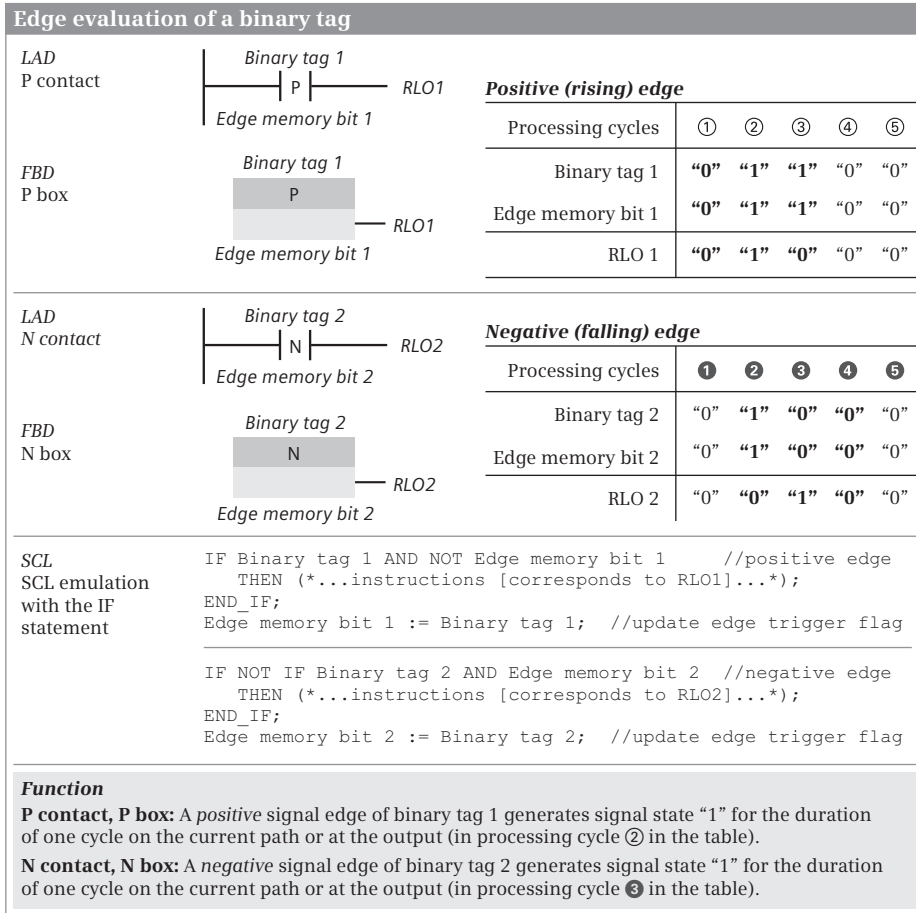


Fig. 10.13 Edge evaluation of a binary tag

if the signal states differ. The edge trigger flag is then updated so that a signal edge is no longer detected during the next processing.

Fig. 10.13 shows the representation and signal states of edge evaluation. The principle of operation of edge evaluation is described in detail in Fig. 10.11 on Seite 340.

### 10.3.4 Edge evaluation with pulse output

The edge evaluation with pulse output generates a pulse at a binary tag from the change in the result of the logic operation (the "current flow"). The positive edge evaluation (P coil or P= box) generates the pulse upon a change in signal state from "0" to "1" (rising edge), the negative edge evaluation (N coil or N= box) upon a change in signal state from "1" to "0" (falling edge). In the event of an edge, the

binary tag has signal state “1” for the duration of one processing cycle. The result of the logic operation following the contact or at the output of the box corresponds to the result of the logic operation prior to the contact or box – it is simply “passed on”.

In SCL the pulse flag is set to signal state “1” if the result of logic operation and the signal state of the edge trigger flag are different. The edge trigger flag is then updated so that a signal edge is no longer detected during the next processing and the pulse flag is reset to signal state “0”.

The function of the edge evaluation is shown in Fig. 10.14; the description corresponds to that in Fig. 10.12 on Seite 341. Here, the edge signal corresponds to the result of preceding logic operation (the “current flow”).

Edge evaluation with pulse output																										
<p><b>LAD</b> P coil</p>	<p><b>Positive (rising) edge</b></p> <table border="1"> <thead> <tr> <th>Processing cycles</th> <th>①</th> <th>②</th> <th>③</th> <th>④</th> <th>⑤</th> </tr> </thead> <tbody> <tr> <td>RLO 1</td> <td>“0”</td> <td>“1”</td> <td>“1”</td> <td>“0”</td> <td>“0”</td> </tr> <tr> <td>Edge memory bit 1</td> <td>“0”</td> <td>“1”</td> <td>“1”</td> <td>“0”</td> <td>“0”</td> </tr> <tr> <td>Pulse flag 1</td> <td>“0”</td> <td>“1”</td> <td>“0”</td> <td>“0”</td> <td>“0”</td> </tr> </tbody> </table>	Processing cycles	①	②	③	④	⑤	RLO 1	“0”	“1”	“1”	“0”	“0”	Edge memory bit 1	“0”	“1”	“1”	“0”	“0”	Pulse flag 1	“0”	“1”	“0”	“0”	“0”	
Processing cycles	①	②	③	④	⑤																					
RLO 1	“0”	“1”	“1”	“0”	“0”																					
Edge memory bit 1	“0”	“1”	“1”	“0”	“0”																					
Pulse flag 1	“0”	“1”	“0”	“0”	“0”																					
<p><b>FBD</b> P=-Box</p>																										
<p><b>LAD</b> N coil</p>	<p><b>Negative (falling) edge</b></p> <table border="1"> <thead> <tr> <th>Processing cycles</th> <th>①</th> <th>②</th> <th>③</th> <th>④</th> <th>⑤</th> </tr> </thead> <tbody> <tr> <td>RLO 2</td> <td>“0”</td> <td>“1”</td> <td>“0”</td> <td>“0”</td> <td>“0”</td> </tr> <tr> <td>Edge memory bit 2</td> <td>“0”</td> <td>“1”</td> <td>“0”</td> <td>“0”</td> <td>“0”</td> </tr> <tr> <td>Pulse flag 2</td> <td>“0”</td> <td>“0”</td> <td>“1”</td> <td>“0”</td> <td>“0”</td> </tr> </tbody> </table>	Processing cycles	①	②	③	④	⑤	RLO 2	“0”	“1”	“0”	“0”	“0”	Edge memory bit 2	“0”	“1”	“0”	“0”	“0”	Pulse flag 2	“0”	“0”	“1”	“0”	“0”	
Processing cycles	①	②	③	④	⑤																					
RLO 2	“0”	“1”	“0”	“0”	“0”																					
Edge memory bit 2	“0”	“1”	“0”	“0”	“0”																					
Pulse flag 2	“0”	“0”	“1”	“0”	“0”																					
<p><b>FBD</b> N=-box</p>																										
<p><b>SCL</b> Emulation with the IF statement</p>	<pre>//The pulse flag is set with positive edge. Pulse flag 1 := [RLO1] AND NOT Edge memory bit 1; Edge memory bit 1 := [RLO1]; //update edge trigger flag //During the next cycle the pulse flag will be reset.  //The pulse flag is set with negative edge. Pulse flag 2 := NOT [RLO2] AND Edge memory bit 2; Edge memory bit 2 := [RLO2]; //update edge trigger flag //During the next cycle the pulse flag will be reset.</pre>																									
<p><b>Function</b></p> <p><b>P coil, P=-box:</b> If a positive signal edge occurs (RLO 1) in front of the P coil or at the input of the P=-box, pulse flag 1 has signal state “1” for the duration of one cycle (in processing cycle ② in the table). After the P-coil or P=-box, there is the same RLO as before the coil or box (RLO 1).</p> <p><b>N coil, N=-box:</b> If a negative signal edge occurs (RLO 2) in front of the N coil or at the input of the N=-box, pulse flag 1 has signal state “1” for the duration of one cycle (in processing cycle ③ in the table). After the N-coil or N=-box, there is the same RLO as before the coil or box (RLO 2).</p>																										

Fig. 10.14 Edge evaluation with pulse output



## 10.4 Time functions

### 10.4.1 Introduction

The timer functions implement timing processes in the user program such as waiting and monitoring times, measurement of a time interval, or the generation of pulses. The following timer functions are available:

- ▷ TP           Pulse generation
- ▷ TON          ON delay
- ▷ TOF          OFF delay
- ▷ TONR        Accumulating ON delay

Fig. 10.15 shows the statements in connection with the timer functions.

A time function is a statement with its own data. When programming a time function, you specify the data block in which the data is to be saved. If you select the *Single instance* button, it must be a different data block each time. If you program a time function in a function block, you can also select *Multi-instance*. The data of the time function is then saved as a local instance in the instance data block of the function block.

The data structure of a timer function is mapped in the system data type (SDT) IEC\_TIMER. The individual components of the data structure is shown in Section 4.8.1 “IEC\_TIMER system data type” on page 110.

The data from several timer functions can be stored in an instance data block under different names. This reduces the number of required data blocks. You can also take advantage of this when calling timer functions in FC and OB blocks: In a global data block, you create a tag with the data type IEC\_TIMER for each call of a timer function and cancel the *Call options* dialog when programming the timer function. Instead, you specify the tag name for the storage location of the instance data. Example: If in the data block “*Timer data*” you create a *Pulse* tag with data type IEC\_TIMER, you can specify the instance name “*Timer\_Data.Pulse*” when calling the timer function.

When calling a timer function TP, TON, TOF, or TONR, you must supply the start input IN and the defined duration PT (preset time) with tags. Supplying of the timer status Q and the elapsed time value ET is optional. You can scan the time status like a binary tag at any point in the user program with *Instance\_Name.Q*.

LAD and FBD know statements that only start a timer function, reset a timer function (RT, Reset Timer), and set a new duration (PT, Preset Timer).

The time functions run in the STARTUP and RUN modes.

Note that the instance data of a timer function is only updated if you call one of the statements TP, TON, TOF, and TONR or on direct access to the structure components Q (time status) and ET (current time value). Thus it may happen that the scans of the time status or the current time value deliver different values at two different points in the program. You can avoid different values in a program cycle if

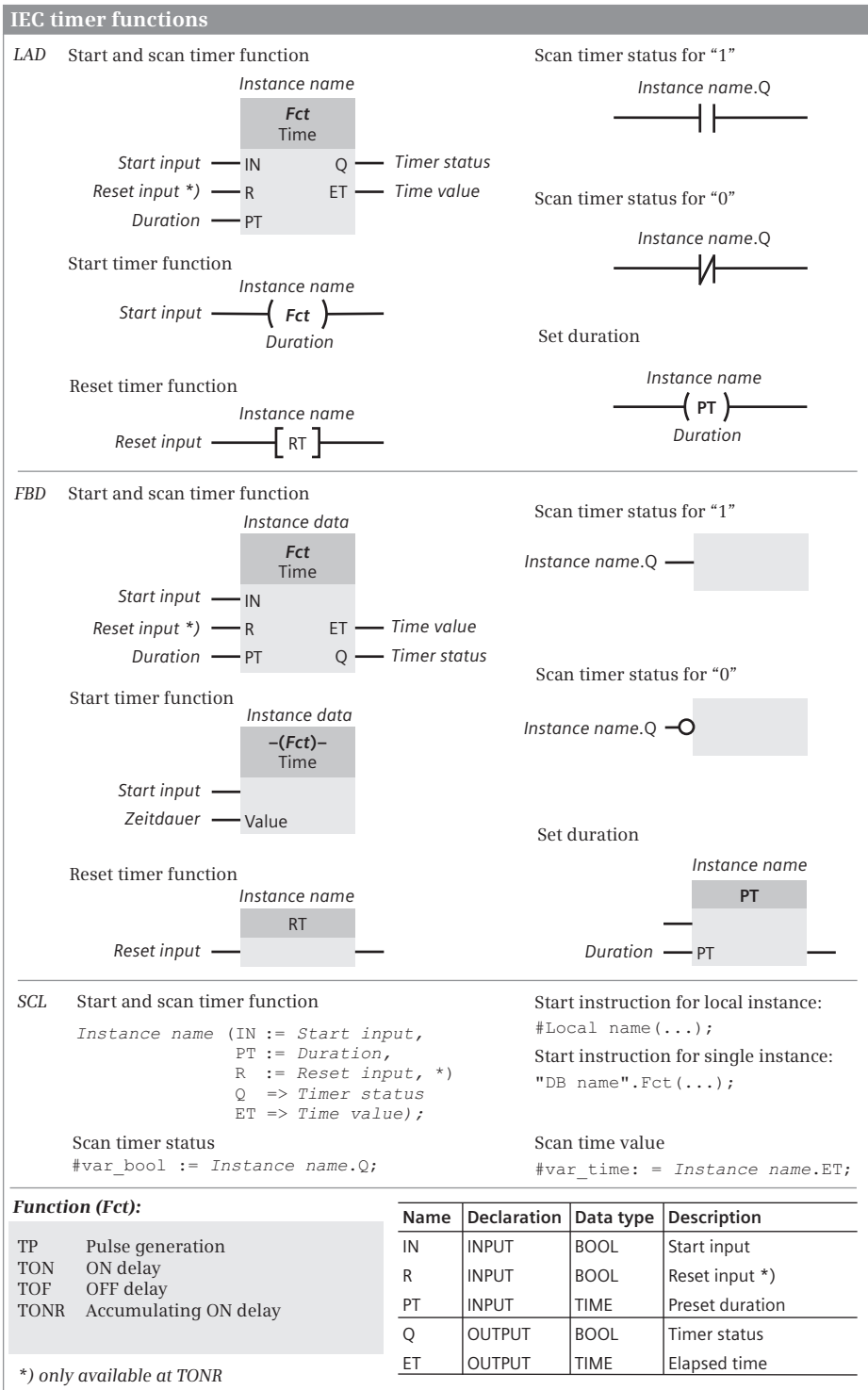


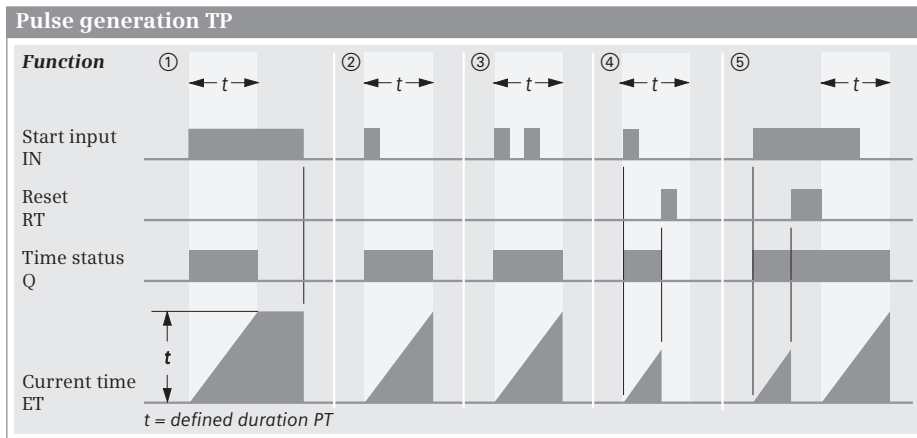
Fig. 10.15 Statements for the timer functions

you assign the time status and/or the current time value to a tag and then scan only the tag.

The time functions are also referred to as “IEC time functions” to indicate that they are different from the SIMATIC S7-300/400” SIMATIC time functions.

### 10.4.2 Pulse generation TP

The pulse generation shortens or extends an input signal to the programmed duration. Fig. 10.16 describes the time response based on the statement TP. The statement RT is available to reset the timer function.



**Fig. 10.16** Time response when starting as pulse TP

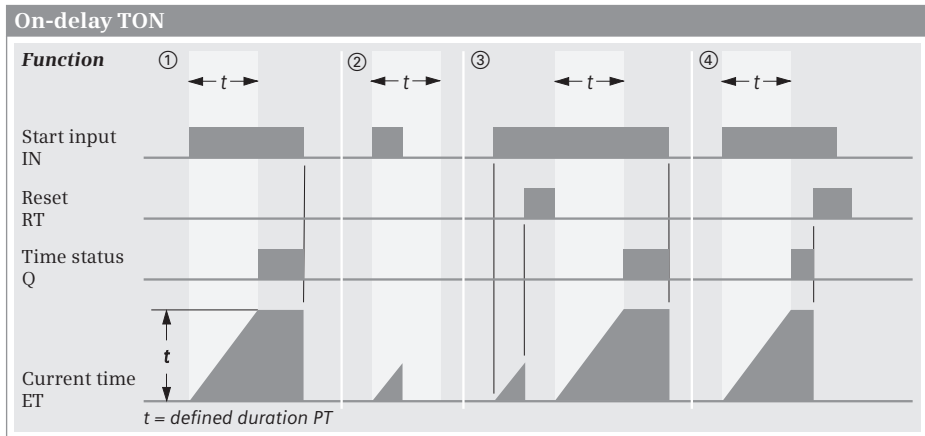
① ② ③ The time function is started if the signal state at the IN start input of the time function changes from “0” to “1”. It runs for the duration programmed at the PT input, independent of the further sequence of the signal state at the start input. The Q output delivers signal state “1” for as long as the time is running.

The ET output delivers the duration which has already expired. This duration starts at T#0s and ends with the set duration PT. If the time has expired, ET remains at the expired value until the signal state at the IN input changes to “0” again. If the IN input has the signal state “0” before PT has expired, the ET output immediately changes to T#0s following expiry of PT.

Resetting the timer function stops the current time when processing with signal state “1”, and resets the current time value to zero. ④ If during the resetting the signal state is “0” at the start input, output Q is also reset. ⑤ If during the resetting the start input IN is assigned signal state “1”, output Q remains set to “1”. After – at signal state “1” at the start input – the reset has been canceled, the timer starts again.

### 10.4.3 On-delay TON

The ON delay delays an input signal by the programmed duration. Fig. 10.17 describes the time response based on the statement TON. The statement RT is available to reset the timer function.



**Fig. 10.17** Time response when starting as ON delay TON

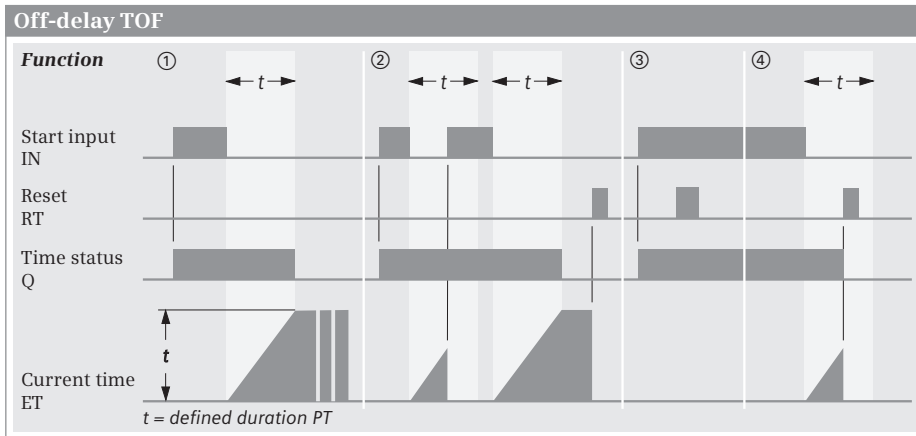
- ① The timer function starts if the signal state at its start input IN changes from "0" to "1". It expires with the duration programmed at the PT input. Output Q delivers signal state "1" if the time has expired and for as long as the start input is still "1".
- ② The elapsed time is reset if the signal state at start input IN changes from "1" to "0" before the time has expired. It starts again with the next positive edge at the IN input.

The ET output delivers the expired time. This duration commences at T#0s and ends at the preset time PT. If PT has expired, ET remains at the expired value until the IN input changes again to "0". If the IN input has signal state "0" prior to expiry of PT, the ET output immediately changes to T#0s.

When processing with signal state "1", resetting the timer function stops the current time and resets the current time value back to zero. ③ The timer function remains stopped as long as the reset signal state is "1". If the start input IN has signal state "1" after the reset has signal state "0" again, the timer function starts again. ④ If the time period has expired and the reset has "1", output Q is reset to signal state "0".

### 10.4.4 OFF delay TOF

The OFF delay delays the switching off of an input signal by the programmed duration. Fig. 10.18 describes the time response based on the statement TOF. The statement RT is available to reset the timer function.



**Fig. 10.18** Time response when starting as OFF delay TOF

① Output Q has signal state “1” if the signal state at start input IN of the timer function changes from “0” to “1”. If the signal state at the start input returns to “0”, the time starts with the duration programmed at the PT input. Output Q remains at signal state “1” for as long as the time is running. Output Q is reset if the time has expired. ② The duration is reset and output Q remains “1” if the signal state at the start input changes to “1” again before the time has expired.

The ET output delivers the expired time. This duration commences at T#0s and ends at the preset time PT. If PT has expired, ET remains on the elapsed value until input IN or reset RT has signal state “1”. If input IN has signal state “1” prior to expiry of PT, output ET immediately changes to T#0s.

The reset of the timer function stops the current time when processing signal state “1” and resets the current time value back to zero. ③ If the signal state at start input IN is “1”, signal state “1” has no effect when resetting. ④ If start input IN changes to “0” and the timer function is still running, the reset with signal state “1” resets the duration and output Q is also reset to signal state “0”.

#### 10.4.5 Accumulating ON delay TONR

The accumulating ON delay delays an input signal by the programmed duration, where an interruption of the input signal prolongs the expiry of the duration (Fig. 10.19).

① The timer function starts if the signal state at its start input IN changes from “0” to “1”. It expires with the duration programmed at the PT input. Output Q delivers signal state “1” if the time has expired, regardless of the further course of the signal state at the start input. ② If the signal state at start input IN changes from “1” to “0” while the time is running, the timer function is stopped, but not reset. ③ If the signal state at the start input switches again to “1”, the timer function continues to run from the interrupted time.

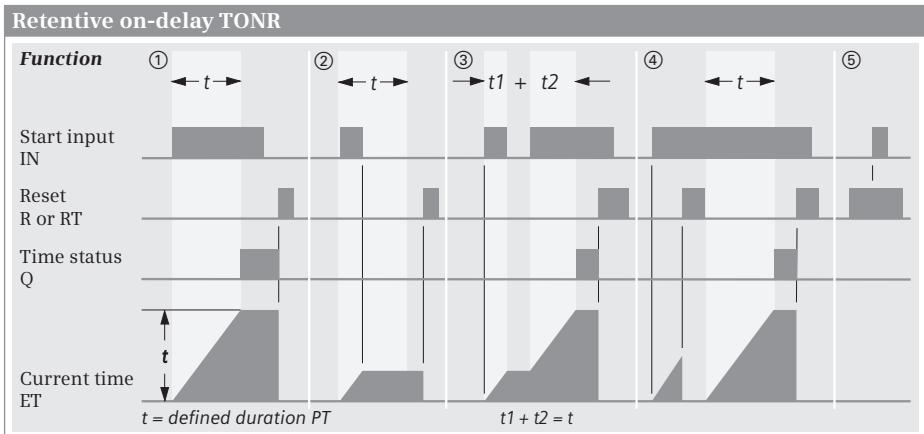


Fig. 10.19 Time response when starting the accumulating ON delay

① ② ③ With signal state “1”, the reset input R resets output Q to signal state “0” and clears the time duration ET. The resetting of Q and deletion of ET take place regardless of the signal state at the start input. ④ If the reset input R is again “0” while the start input IN is still “1”, the time starts again. ⑤ If the signal state at the start input changes from “0” to “1” while the reset input R has signal state “1”, the timer function is not started.

The RT function has the same effect as the reset input R. Resetting the timer function when processing with signal state “1” stops the current time running, sets the time status to signal state “0”, and deletes the current time value.

## 10.5 Counter functions

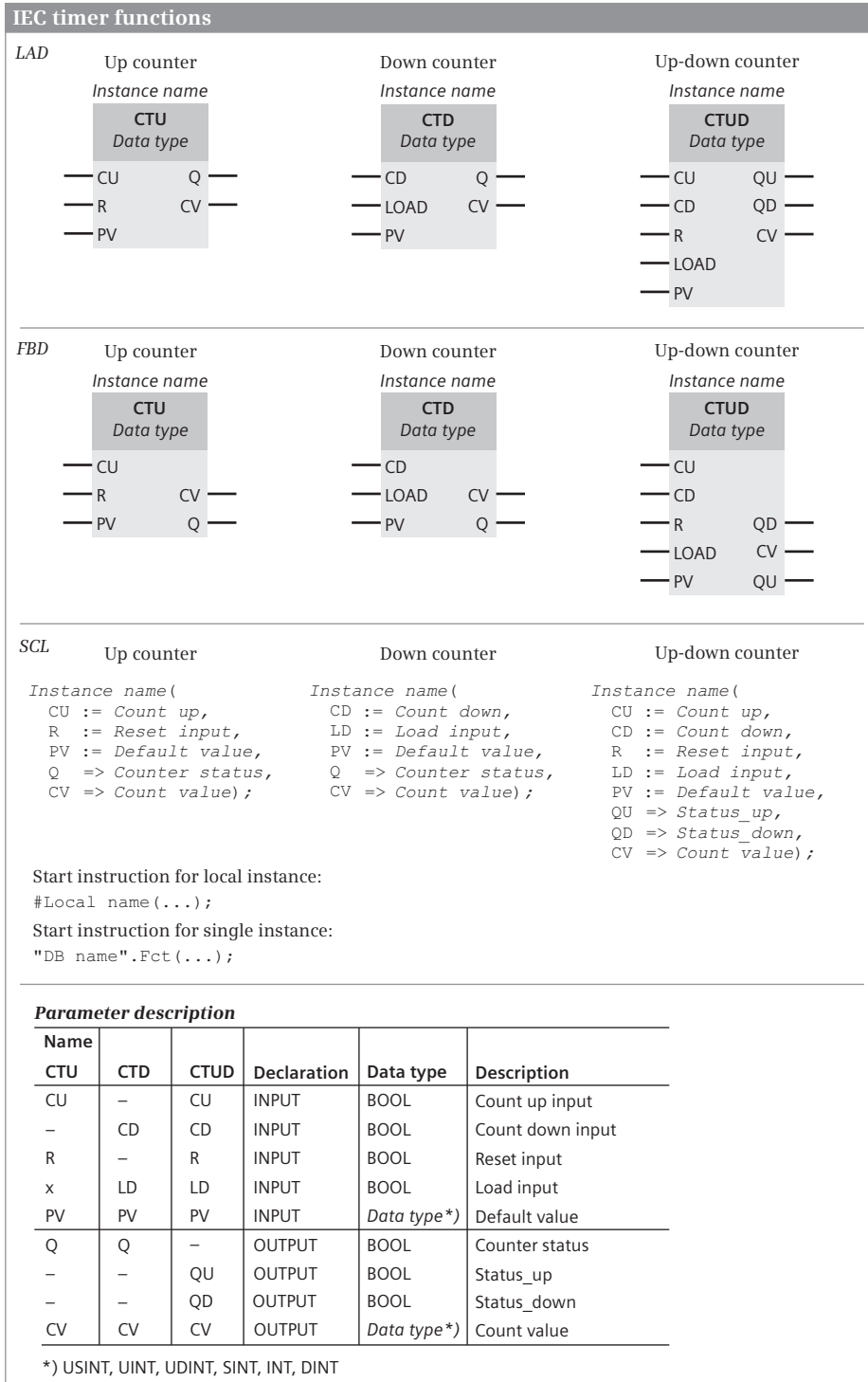
### 10.5.1 Introduction

The counter functions perform counting tasks in the user program directly through the CPU. The counter functions can count up and down; the numerical range corresponds to the set data type. The counting frequency of the counter functions depends on the execution time of the program. In order to count, the CPU must recognize a change in the signal state of the input pulse, i.e. the input pulse and the pause must be present for at least one program cycle. The longer the program execution time, the lower the counting frequency.

The following counter functions are available:

- ▷ CTU Up counter
- ▷ CTD Down counter
- ▷ CTUD Up-down counter

The Fig. 10.20 shows the statements in connection with the counter functions.



**Fig. 10.20** Statements for the counter functions

A counter function is a statement with its own data. When programming a counter function, you specify the data block in which the data is to be saved. If you select the *Single instance* button, it must be a different data block each time. If you program a counter function in a function block, you can also select *Multi-instance*. The data of the counter function is then saved as a local instance in the instance data block of the function block.

The count value of a counter function can be set when programming for the data types SINT, INT, DINT, USINT, UINT, and UDINT. The data structure of a counter function is dependent on this setting. The setup of the data structure is shown in Section 4.6.2 “Parameter types for IEC counter functions” on page 108.

The data from several counter functions can be stored in an instance data block under different names. This reduces the number of required data blocks. You can also take advantage of this when calling counter functions in FC and OB blocks: In a global data block, you create a tag with the data type IEC\_xCOUNTER for each call of a counter function and cancel the *Call options* dialog when programming the counter function. Instead, when calling the counter function you then specify the tag name for the storage location of the instance data. Example: If in the data block “Counter data” you create a *Number* tag with the data type IEC\_COUNTER, you can specify the instance name “Counter\_Data”.*Number* when programming the counter function.

When calling a counter function CTU, CTD, or CTUD, you must supply a start input and the defined count value PV (preset value) with tags. Supplying of the counter status Q (QU, QD) and the current count value CV is optional. You can scan the counter status like a binary tag at any point in the user program with *Instance\_Name.Q*.

The counter functions execute in the STARTUP and RUN modes.

The counter functions are also referred to as “IEC counter functions” to indicate that they are different from the SIMATIC S7-300/400 “SIMATIC counter functions”.

In addition to the counter functions, high-speed counters are integrated in the CPU. These counters are independent of the program execution time (see Chapter 17.1.1 “High-speed counter (HSC)” on page 548).

### 10.5.2 Up counter CTU

If the signal state at the count up input CU changes from “0” to “1” (positive edge), the current count value is incremented by 1 and is indicated at the CV output. If the current count value reaches the upper limit of the set data type, it is no longer incremented. A positive edge at CU then has no effect. Fig. 10.21 describes the count behavior of the up-counter.

The counted value is reset to zero if the reset input R has signal state “1”. A positive edge at CU has no effect as long as the R input has signal state “1”.

The Q output has the signal state “1” if the actual counted value is greater than or equal to the defined counted value ( $CV \geq PV$ ).



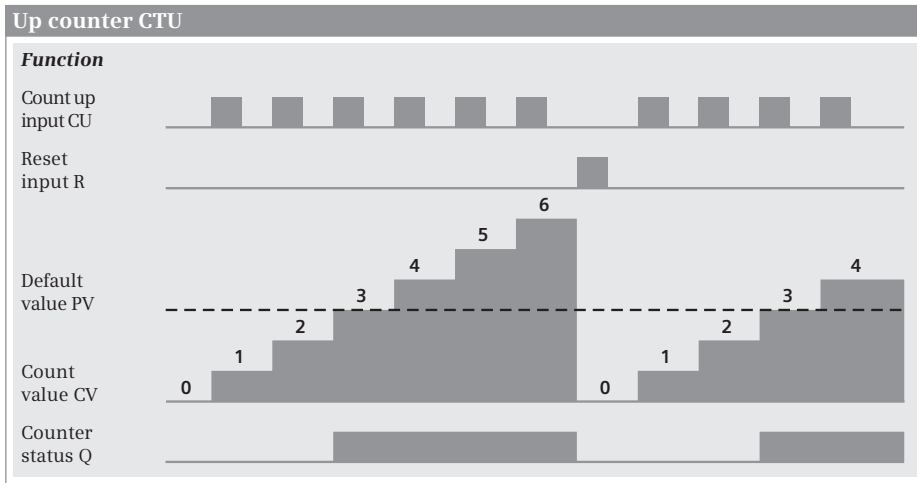


Fig. 10.21 Count behavior of the CTU up-counter

### 10.5.3 Down counter CTD

If the signal state at the down-counter input CD changes from “0” to “1” (positive edge), the current count value is decremented by 1 and is present at the CV output. If the current count value reaches the lower limit of the selected data type, it is no longer decremented. A positive edge at CD then has no effect. Fig. 10.22 describes the count behavior of the down-counter.

The count value CV is set to the specified count value PV if the LOAD input has signal state “1”. A positive edge at the CD input has no effect for as long as the LOAD input has signal state “1”.

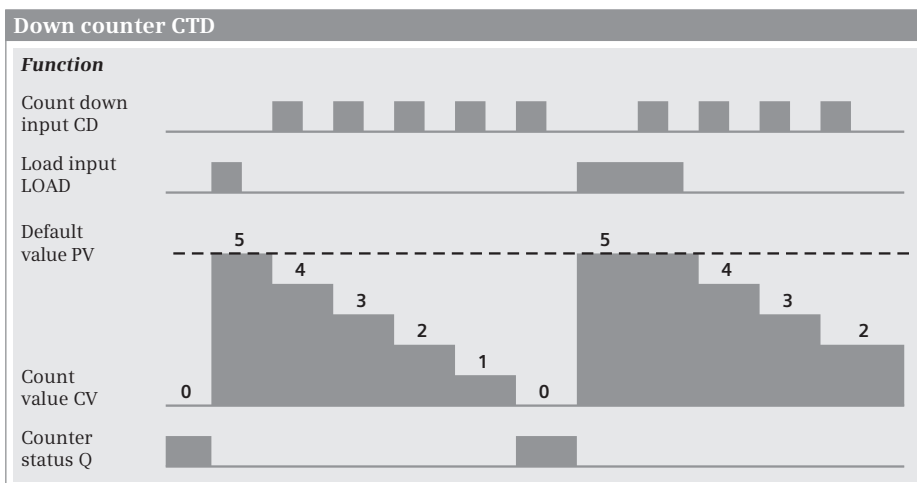


Fig. 10.22 Counter behavior of the CTD down-counter

The Q output has the signal state “1” if the actual counted value is less than or equal to zero (CV ≤ 0).

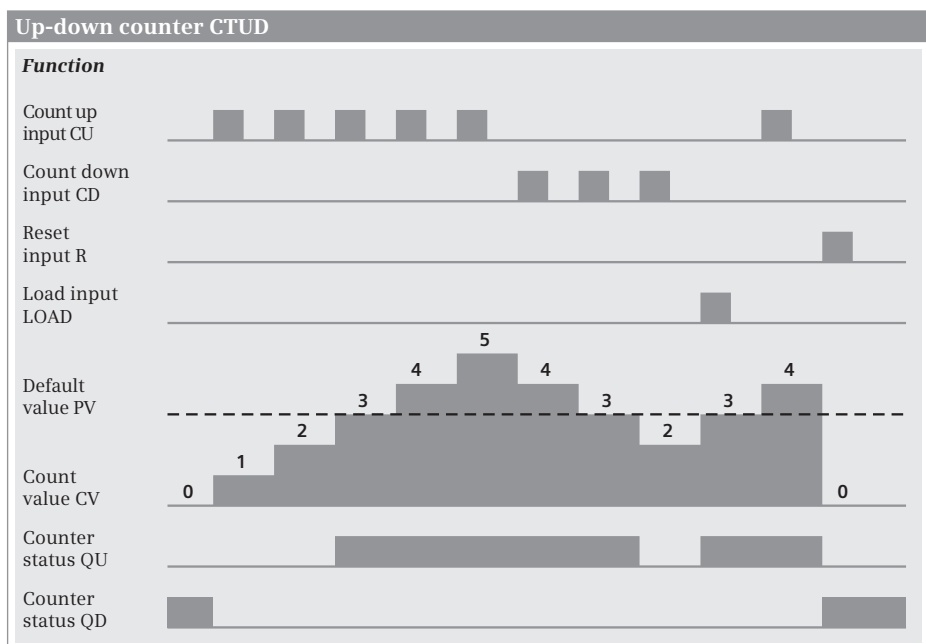
#### 10.5.4 Up-down counter CTUD

If the signal state at the count up input CU changes from “0” to “1” (positive edge), the count value is incremented by 1 and is indicated at the CV output. If the signal state at the count down input CD changes from “0” to “1” (positive edge), the count value is decremented by 1 and is present at the CV output. If both count inputs have a positive edge, the current count value is not changed. The Fig. 10.23 describes the count behavior of the up/down-counter.

If the actual counted value reaches the upper limit of the selected data type, it is no longer incremented. A positive edge at the count up input CU then has no effect. If the actual counted value reaches the lower limit of the selected data type, it is no longer decremented. A positive edge at the count down input CD then has no effect.

The actual counted value CV is set to the pre-defined counted value PV if the LOAD input has signal state “1”. Positive signal edges at the counting inputs CU and CD have no effect as long as the LOAD input has signal state “1”.

The counted value is reset to zero if the reset input R has signal state “1”. Positive signal edges at the counting inputs CU and CD and signal state “1” at the LOAD input have no effect as long as the R input has signal state “1”.



**Fig. 10.23** Up/down-counter CTUD, representation and function

The QU output has the signal state “1” if the actual counted value is greater than or equal to the defined counted value ( $CV \geq PV$ ).

The QD output has the signal state “1” if the actual counted value is less than or equal to zero ( $CV \leq 0$ ).

# 11 Digital functions

This chapter describes the digital functions which mainly link digital tags together, for example the basic arithmetic operations for the arithmetic functions. As far as possible, the description is independent of the programming language.

The Chapters 7 “Ladder logic LAD” on page 209, 8 “Function block diagram FBD” on page 246, and 9 “Structured Control Language SCL” on page 284 describe how you can program the functions using the individual programming languages and what special features exist.

The digital functions are implemented internally – not visible to you as the user – either by means of basic instruction sequences or by calling a system or standard block. Therefore you can find the digital functions in the program elements catalog under *Basic instructions* and *Extended instructions*. The following digital functions are available with a CPU 1200:

- ▷ The transfer functions transfer the value of a (digital) tag or memory area.
- ▷ The comparison functions generate a binary result by comparing two tags.
- ▷ The arithmetic functions for numerical values link two tags with data types fixed-point and floating-point data types in accordance with the basic arithmetic operations.
- ▷ The arithmetic functions for time values link two tags with data types DTL, TOD, DATE, and TIME.
- ▷ The math functions convert the value of a tag with data type REAL or LREAL in accordance with the specified function, for example calculation with a trigonometric function.
- ▷ The conversion functions convert the data type of a tag.
- ▷ The shift functions shift the content of a tag bit by bit to the right or left.
- ▷ The logic functions comprise, for example, the word logic operations, which link two tags bit by bit, and the selection and limiting functions.
- ▷ The functions for strings process tags with data type STRING. Two strings can be combined, for example.
- ▷ In LAD and FBD, the CALCULATE box allows a complex, user-defined logical operation with logical, arithmetic, and mathematical functions.

The “simple” digital functions are boxes in the case of LAD and FBD (with LAD, the comparison is a contact) and arithmetic, logic and comparison expressions in the case of SCL.

## 11.1 Transfer functions

### 11.1.1 Introduction

The transfer functions allow values to be exchanged between tags. The tags can also be overwritten with fixed values (constants). The tags – except data type BOOL – can be of all data types; any limitations are referred to in the respective functional descriptions. The transfer function is executed when the EN input has signal state “1” or is unused. The ENO output has signal state “0” if the box has not been processed (EN = “0”) or if an error has occurred during box processing.

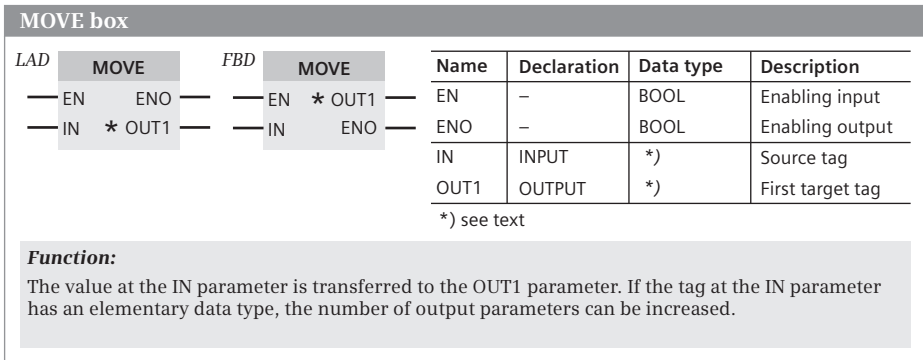
The following transfer functions are available:

- ▷ MOVE-Box, S\_MOVE-Box  
Copy tag content (LAD, FBD)
- ▷ Value assignment  
Copy the value of a tag or of an expression (SCL)
- ▷ MOVE\_BLK box  
Copy data area
- ▷ UMOVE\_BLK box  
Copy data area without interruption
- ▷ FILL\_BLK box  
Fill the data area
- ▷ UFILL\_BLK box  
Fill the data area without interruptions
- ▷ READ\_DBL, WRIT\_DBL  
Transfer data area from and to load memory
- ▷ SWAP box  
Swap bytes

The FieldRead and FieldWrite functions, which in STEP 7 V10.5 allowed field components to be addressed with a variable index, is still available in STEP 7 V11, but is no longer required. With indirect addressing of field components, there is a more elegant way.

### 11.1.2 Copy tag, MOVE box for LAD and FBD

The MOVE box transfers the content of the tag at the IN parameter to the tag at the OUT1 parameter (Fig. 11.1). If there is a tag with elementary data type at parameter IN, the MOVE box can be expanded with additional outputs OUT2, OUT3, etc. using the command *Insert output* from the shortcut menu. The content of the input tag is then transferred to all box inputs. A tag with elementary data type can also be a component of a structured data type.



**Fig. 11.1** MOVE box, representation and function

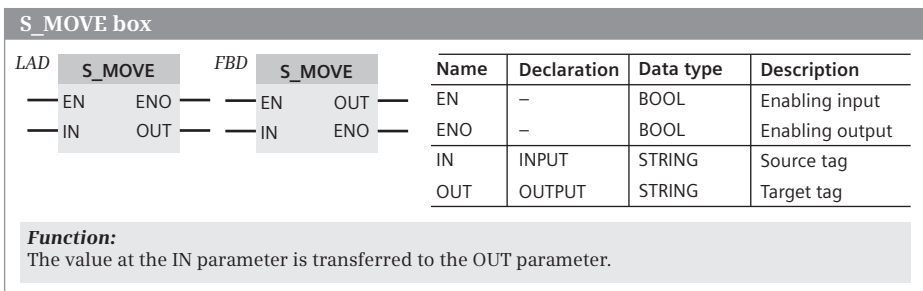
The tags at the parameters IN and OUT1 or OUT $n$  can have different types of data. Which data types are allowed depends on the block attribute *IEC check* (Table 11.1). If the data width of the target tags is smaller than the data width of the tags at parameter IN, the higher-level bits are “cut off” and lost. If the data width of the target tag is larger, the higher-level bits are filled with zeros.

The MOVE box can also transfer tags with structured data type, tags with hardware data type, PLC data type, system data type, and entire data blocks that are derived from a data type (type data blocks). In these cases, the data types at IN and OUT1 must always coincide. An extension of the box outputs (with OUT2, OUT3, etc.) is then not possible.

### 11.1.3 Copy string, S\_MOVE box for LAD and FBD

The S\_MOVE box transfers the content of the tag at the IN parameter to the tag at the OUT parameter (Fig. 11.2). The tags are of data type STRING.

If the target tag is greater than the source tag, the source tag is transferred completely to the target tag and the current length is updated.



**Fig. 11.2** S\_MOVE box, representation and function

**Table 11.1** Data types for the MOVE box

Source (IN)	Target (OUT1)	
	with IEC check	without IEC check
BYTE	BYTE, WORD, DWORD	BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, TIME, DATE, TOD, CHAR
WORD	WORD, DWORD	BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, TIME, DATE, TOD, CHAR
DWORD	DWORD	BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, REAL, TIME, DATE, TOD, CHAR
SINT	SINT	BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, TIME, DATE, TOD
INT	INT	BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, TIME, DATE, TOD
DINT	DINT	BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, TIME, DATE, TOD
USINT	USINT, UINT, UDINT	BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, TIME, DATE, TOD
UINT	UINT, UDINT	BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, TIME, DATE, TOD
USINT	UDINT	BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, TIME, DATE, TOD
REAL	REAL	DWORD, REAL
LREAL	LREAL	LREAL
TIME	TIME	BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, TIME
DATE	DATE	BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, DATE
TOD	TOD	BYTE, WORD, DWORD, SINT, INT, DINT, USINT, UINT, UDINT, TOD
DTL	DTL	DTL
CHAR	CHAR	BYTE, WORD, DWORD, character *)
Character *)	Character *)	CHAR, character *)
ARRAY	ARRAY	ARRAY
STRUCT	STRUCT	STRUCT

\*) Individual character of a string (STRING data type)

If the target tag is smaller than the source tag, only as many characters are transferred as will fit in the target tag. The current length is given the value of the maximum length and the ENO output is set to signal state “0”.

#### 11.1.4 Value assignments with SCL

A value assignment transfers the value of an expression to a tag. On the left of the assignment operator is the output tag, which accepts the value of the expression positioned on the right. The expression can be a constant, a single tag, a combination of tag values, or a function whose function value is assigned to the output parameter.

```
#Output_tag := #Input_tag; //Assignment of tag value
```

The data type of the value assignment is determined by the output tag. The data types on both sides of the assignment operator must be the same. Exception: With the *IEC check* block attribute deactivated, the “implicit data type conversion” is applicable, see Chapter 4.2.3 “Absolute addressing of an operand area” on page 86.

### Assignment for elementary data types

A constant value, a different tag, or an expression can be assigned to a tag or operand.

Absolutely addressed operands (e.g. %MW10) have one of the data types BOOL, BYTE, WORD, or DWORD. If you wish to assign a value with a different data type to an absolutely addressed operand, you can use the data type conversion or assign a name and the desired data type to the operand in the PLC tag table.

### Assignment of DTL tags

Every DTL tag can be assigned another DTL tag or a DTL constant. A single component can be used like a tag with the data type of the component. Example: In the *#Delivery\_date* tag with the DTL data type, the hour should be set:

```
#Delivery_date.HOUR := #Hour; //Data_type USINT
```

### Assignment of STRING tags

Every STRING tag can be assigned another STRING tag or a STRING constant. If the source tag is smaller than the target tag to the left of the assignment operator, all characters are transferred and the current length is updated. If the source tag is longer, only as many characters are transmitted as will fit in the target tag and the current length is set to the maximum length.

A STRING tag can be assigned a tag with data type CHAR. Example:

```
#String := #Single_character;
```

### Assignment of STRUCT tags or PLC data types

A STRUCT tag or PLC data type can only be assigned to another STRUCT tag or PLC data type if

- ▷ the data structures agree,
- ▷ the data types of the structure components agree, and
- ▷ the names of the structure components agree.

Individual structure components can be handled like tags of the corresponding data type, for example a structure component *#Motor1.Setpoint* with data type INT can be assigned to another INT tag, or an INT value can be assigned to this structure component.



### Assignment of ARRAY tags

An ARRAY tag can only be assigned to another ARRAY tag if the data types of the array components as well as the array limits with smallest and largest array index agree with each other.

Individual array components can be handled like tags of the corresponding data type. Example: If *#Measured\_values* is a field of REAL components, and *#Index* is a tag with the data type INT, a field component can be assigned the value of the *#Width* tag with data type REAL:

```
#Measured_values[#Index] := #Width;
```

#### 11.1.5 Copy data area (MOVE\_BLK, UMOVE\_BLK)

MOVE\_BLK and UMOVE\_BLK transfer the contents of sequential components of an ARRAY tag to components of another ARRAY tag. The source area is defined by the start tag at parameter IN, and the target area by the start tag at parameter OUT. As many values are copied as the number specified at the COUNT parameter (Fig. 11.3).

**MOVE\_BLK box, UMOVE\_BLK box**

LAD	Function	FBD	Function	Name	Declaration	Data type	Description
— EN	ENO	— EN	EN	EN	—	BOOL	Enabling input
— IN	OUT	— IN	OUT	ENO	—	BOOL	Enabling output
— COUNT		— COUNT	ENO	IN	INPUT	ARRAY[n] *)	Source tag
				COUNT	INPUT	*)	Number
				OUT	OUTPUT	ARRAY[n] *)	Target tag

\*) see text

SCL

**Function** (IN := ... , COUNT := ... , OUT => ... );

**Data type:**

IN, OUT: ARRAY[n] means a component (an element) of a tag with data type ARRAY. This component has an elementary data type.

COUNT: for LAD and FBD: UINT (with activated attribute *IEC check*)  
for SCL: USINT, UINT, UDINT (with activated attribute *IEC check*)

**Funktion:**

**MOVE\_BLK** Move data area  
**UMOVE\_BLK** Move data area without interruption

The contents of the source tags are transferred to the ARRAY tags at the OUT parameter into the number of components that is specified at the COUNT parameter. The component specified at the OUT parameter is the start component for the target.

**Fig. 11.3** MOVE\_BLK and UMOVE\_BLK box, representation and function

With the IEC check activated, the tags at the IN and OUT parameters must have the same data type; if the attribute *IEC check* is not activated in the block, it is only necessary for the data widths to agree.

MOVE\_BLK copies the values so that the process can be interrupted by a higher-priority program (advantage: quick response time to alarms). UMOVE\_BLK copies without interruption (advantage: transfer of consistent data areas). During transfer by UMOVE\_BLK, alarm events that occur are stored and processed after the transfer ends.

MOVE\_BLK and UMOVE\_BLK report an error (ENO = "0") if a range limit is exceeded during runtime. No values are copied if an error occurs.

### 11.1.6 Filling the data area (FILL\_BLK, UFILL\_BLK)

FILL\_BLK and UFILL\_BLK transfer the content of a tag to sequential components of an ARRAY tag. The source tag is defined by parameter IN and the target area by the start tag at parameter OUT. As many values are copied as the number specified at the COUNT parameter (Fig. 11.4).

**FILL\_BLK box, UFILL\_BLK box**

LAD	Function	FBD	Function	Name	Declaration	Data type	Description
— EN	ENO	— EN		EN	—	BOOL	Enabling input
— IN	OUT	— IN	OUT	ENO	—	BOOL	Enabling output
— COUNT		— COUNT	ENO	IN	INPUT	Data type *)	Source tag
				COUNT	INPUT	UINT	Number
				OUT	OUTPUT	ARRAY[n] *)	Target tag

\*) Same data width at IN and OUT

SCL **Function** (IN := ... , COUNT := ... , OUT => ... );

**Data type:**

IN, OUT: ARRAY[n] means a component (an element) of a tag with data type ARRAY. This component has an elementary data type.

COUNT: for LAD and FBD: UINT (with activated attribute *IEC check*)  
for SCL: USINT, UINT, UDINT (with activated attribute *IEC check*)

**Funktion:**

**FILL\_BLK** Fill data area  
**UFILL\_BLK** Fill data area without interruption

The contents of the source tags are transferred to the ARRAY tags at the OUT parameter into the number of components that is specified at the COUNT parameter.  
The component specified at the OUT parameter is the start component for the target.

The diagram illustrates the data transfer process. On the left, a box labeled 'Source tag' contains 'Parameter IN'. An arrow points from this box to a vertical stack of five rectangular boxes representing the 'ARRAY tag (target)'. The top box in the stack is labeled 'Parameter OUT'. A downward-pointing arrow on the right side of the stack is labeled 'COUNT', indicating the number of elements to be filled.

Fig. 11.4 FILL\_BLK and UFILL\_BLK box, representation and function

With the IEC check activated, the tags at the IN and OUT parameters must have the same data type; if the *IEC check* attribute is not activated in the block, it is only necessary for the data widths to agree.

FILL\_BLK copies the values so that the process can be interrupted by a higher-priority program (advantage: quick response time to alarms). UFILL\_BLK copies without interruption (advantage: transfer of consistent data). During transfer by UFILL\_BLK, alarm events that occur are stored and processed after the transfer ends.

FILL\_BLK and UFILL\_BLK report an error (ENO = "0") if a range limit is exceeded during runtime. No values are copied if an error occurs.

### 11.1.7 Read and write the load memory (READ\_DBL, WRIT\_DBL)

A data block is normally present twice in the user memory: The data block with declaration of the data tags and start values is present in the load memory and with the actual values with which the user program is working in the work memory. READ\_DBL transmits a data block or a data area – for example, recipe data – from the load memory to the work memory. WRIT\_DBL transmits a data block or a data area – for example, archive data – from the work memory to the load memory. Both the source and the target data blocks must have the same access type: The block attribute *Optimized block access* must be either enabled or disabled in both data blocks. Fig. 11.5 shows the graphic representation of these functions.

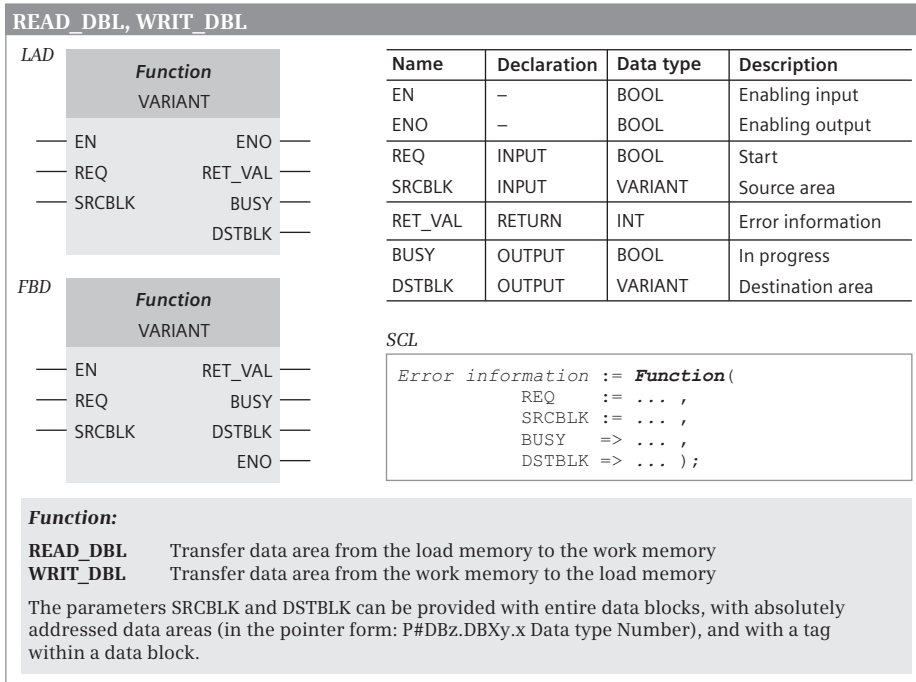


Fig. 11.5 Transfer data areas from and to the load memory

Complete data blocks or parts of data blocks are permissible as actual parameters at the SRCBLK and DSTBLK block parameters. They can be supplied with:

- ▷ entire data blocks that are derived from a PLC data type or system data type,
- ▷ Tags from the data blocks, and
- ▷ – for data blocks with a disabled *Optimized block access* attribute – with an absolutely addressed data area, e.g. P#DB100.DBX16.0 BYTE 64 (see Chapter 4.2.3 “Absolute addressing of an operand area” on page 86 for description).

The READ\_DBL and WRIT\_DBL functions work asynchronously: They trigger the transfer process by signal state “1” at the REQ parameter. You may only access the read and written data areas again when the parameter BUSY has signal state “0” again. Also observe the CPU’s system resources when using asynchronous system functions.

If the source area is smaller than the destination area, the source area is written completely into the destination area. The remaining bytes of the destination area are not changed. If the source area is larger than the destination area, the destination area is written completely; the remaining bytes of the source area are ignored.

Note that the load memory only permits a limited number of write operations as a result of the physical design. Too frequent writing, e.g. writing in every program cycle, reduces the service life of the load memory.

### 11.1.8 Swap bytes (SWAP)

SWAP reads the tag at the IN parameter, exchanges its bytes, and makes the result available at the OUT parameter. If the block attribute *IEC check* is enabled, the data types WORD and DWORD are permitted at the IN and OUT parameters. If the block

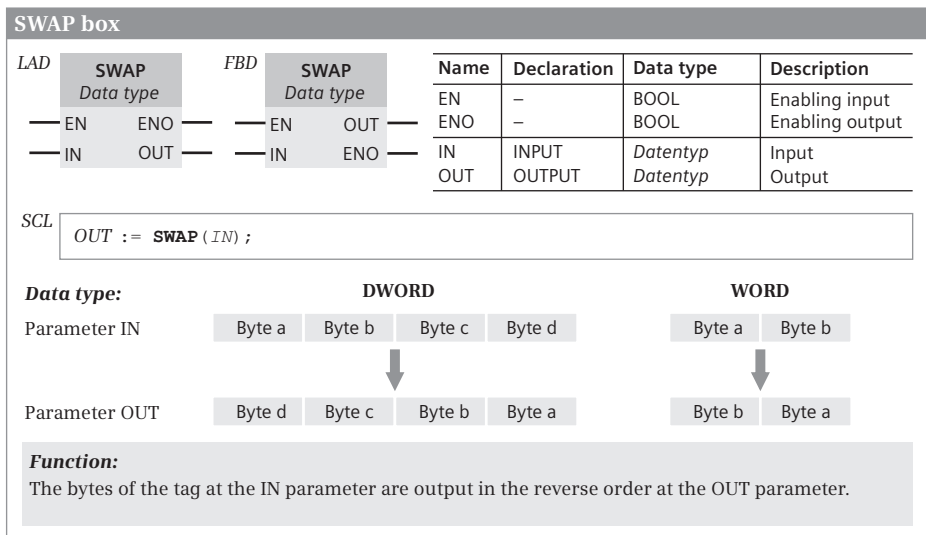


Fig. 11.6 Swap bytes with SWAP

attribute is disabled, you can create word-wide or doubleword-wide tags. The data types may differ, but not the data widths (Fig. 11.6).

## 11.2 Comparison functions

### 11.2.1 Overview

A comparison function compares the values of two tags with each other, or checks whether a tag value is within or outside a certain range. A comparison function delivers a binary comparison result.

The following comparison functions are available:

- ▷ The comparison of two tag values for equal, not equal, greater than, greater than or equal to, less than, and less than or equal to
- ▷ The range comparison

### 11.2.2 Comparison of two tag values

The comparison function for two tag values compares the contents of both input tags and forms the comparison result according to the comparison function. The

**Comparison function, compare two values**

LAD

FBD

Name	Declaration	Data type	Description
IN1	INPUT	Data type	Input tag 1
IN2	INPUT	Data type	Input tag 2
-	OUTPUT	BOOL	Result of comparison

SCL

```
Result of comparison := IN1 Function IN2;
```

Function: IN1 <Comparison> IN2	Data types:						
	USINT, UINT, UDINT	SINT, INT, DINT	REAL, LREAL	CHAR, STRING	TIME, DTL	BYTE, WORD, DWORD	
=	equal to	x	x	x	x	x	x
<>	not equal to	x	x	x	x	x	x
>	greater than	x	x	x	x	x	-
>=	greater than or equal to	x	x	x	x	x	-
<	less than	x	x	x	x	x	-
<=	less than or equal to	x	x	x	x	x	-

Tags with the following data types can be compared to each other:

- > USINT, UINT, UDINT, SINT, INT, DINT, REAL and LREAL
- > BYTE, WORD, DWORD

Tags with the other data types can only be compared with the same data type.

**Fig. 11.7** Comparison function, comparison of two values

comparison result is “1” if the comparison is fulfilled, otherwise “0”. The data types allowed for the comparison function types are listed in Fig. 11.7.

The comparison of numerical values to the data types USINT, UINT, UDINT, SINT, INT, DINT, REAL, and LREAL is made within the framework of the specified data type. When comparing, different data types can be used if they can be converted to the respective “most powerful” data type using the implicit data type conversion and thus become comparable (see Chapter 4.3.2 “Implicit data type conversion” on page 93). The “most powerful” data type must then be set at the comparison function.

When comparing tags with the data types BYTE, WORD, and DWORD, the data types can be different. In the comparison function, the data type must then be set with the larger data width.

A prerequisite for a fulfilled comparison of floating-point numbers is that they are valid. If an invalid floating-point number is compared, the comparison is always invalid.

The comparison of character values CHAR and STRING is carried out in the context of ASCII coding. Two strings are identical if the relevant (occupied) characters are the same and the actual length is the same. If the first characters are identical, a string is considered to be greater if it is longer. The maximum lengths of the strings are not included in the comparison.

The comparison of time values TIME and DTL is carried out in the context of the specified data type. A point in time (date, time) is considered as smaller if the numerical value is smaller, i.e. if the point in time is older.

### 11.2.3 Range comparison

The range comparison checks whether the value of a digital tag is within or outside a range of values defined by limits (Fig. 11.8).

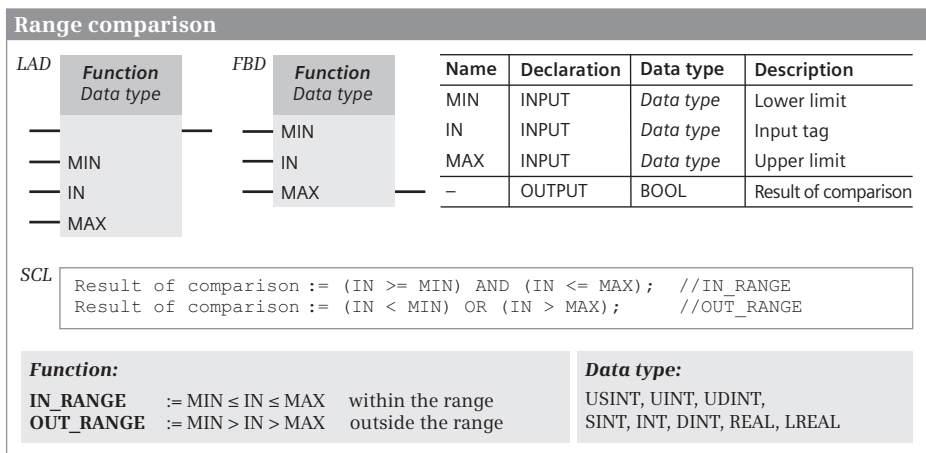


Fig. 11.8 Range comparison

The comparison is fulfilled (comparison result = “1”) if the digital value for the function IN\_RANGE is within the range or if the digital value for OUT\_RANGE is outside the range. The limits (MIN, MAX) and the digital tag to be compared (IN) are at the inputs of the box. The binary comparison result is available at the unlabeled output of the box.

All fixed-point and floating-point numbers are permitted as data types of the input parameters. The data types at the input parameters can be different if they can be changed to the “most powerful” data type with the implicit conversion and thus compared. For different data types, the “most powerful” data type is specified in the box.

The lower limit and upper limit may also be assigned constants. If invalid floating-point numbers are specified, the comparison is not fulfilled.

### 11.3 Arithmetic functions for numerical values

#### 11.3.1 Introduction

The arithmetic functions for numerical values link two values according to the basic arithmetical operations addition, subtraction, multiplication, and division (Fig. 11.9). The arithmetic functions include absolute value generation, changing the sign (negation), and changing the value by 1 (decrementing, incrementing).

LAD and FBD: An arithmetic function is executed if the enable input EN is not connected, or if “1” is present at the enable input, or if “current” flows into the enable

Arithmetic functions for numerical values																										
LAD	Function		FBD	Function	Name	Declaration	Data type	Description																		
	Data type			Data type																						
— EN	ENO	—	— EN		EN	—	BOOL	Enabling input																		
— IN1	OUT	—	— IN1	OUT	ENO	—	BOOL	Enabling output																		
— IN2		—	— IN2	ENO	IN1	INPUT	Data type	Input tag 1																		
					IN2	INPUT	Data type	Input tag 2																		
					OUT	OUTPUT	Data type	Result																		
<p>SCL</p> <pre>OUT := IN1 Function IN2;</pre>																										
<p><b>Function:</b></p> <table border="1"> <thead> <tr> <th>LAD/FBD</th> <th>SCL</th> <th></th> </tr> </thead> <tbody> <tr> <td>ADD</td> <td>+</td> <td>Addition</td> </tr> <tr> <td>SUB</td> <td>-</td> <td>Subtraction</td> </tr> <tr> <td>MUL</td> <td>*</td> <td>Multiplication</td> </tr> <tr> <td>DIV</td> <td>/</td> <td>Division</td> </tr> <tr> <td>MOD</td> <td>MOD</td> <td>Division with remainder as result</td> </tr> </tbody> </table>					LAD/FBD	SCL		ADD	+	Addition	SUB	-	Subtraction	MUL	*	Multiplication	DIV	/	Division	MOD	MOD	Division with remainder as result	<p><b>Data type:</b> USINT, UINT, UDINT, SINT, INT, DINT REAL, LREAL (<i>not with MOD</i>)</p> <p>If different data types are specified, the result adopts the “most powerful” data type.</p>			
LAD/FBD	SCL																									
ADD	+	Addition																								
SUB	-	Subtraction																								
MUL	*	Multiplication																								
DIV	/	Division																								
MOD	MOD	Division with remainder as result																								

Fig. 11.9 Arithmetic functions for numerical values, representation and function

input. If an error occurs during calculation, the enable output ENO is set to “0”, otherwise to “1”. If the execution of the function is not enabled (EN = “0”), the calculation does not take place and ENO is also “0”.

SCL: The basic arithmetical operations are implemented with an arithmetic expression in which two or more tags are linked. The result of an arithmetic expression can in turn be used in another arithmetic expression. If, during the execution of an arithmetic function, an error such as exceeding a number range occurs, the ENO tag is set to FALSE (signal state “0”).

### 11.3.2 Addition ADD

The ADD function interprets the values present at the IN1 and IN2 inputs as numbers with the specified data type. It adds the two numbers, and saves the total at the OUT output. Leaving the permissible range is reported by ENO = “0”.

Floating-point addition: with an impermissible calculation (one of the input values is an invalid floating-point number, or you attempt to add  $+\infty$  and  $-\infty$ ), ADD delivers an invalid value at the OUT output and sets ENO to “0”.

### 11.3.3 Subtraction SUB

The SUB function interprets the values present at the IN1 and IN2 inputs as numbers with the specified data type. It subtracts the value at IN2 from the value at IN1, and saves the difference at the OUT output. Leaving the permissible range is reported by ENO = “0”.

Floating-point subtraction: with an impermissible calculation (one of the input values is an invalid floating-point number, or you attempt to subtract  $+\infty$  from  $+\infty$  or  $-\infty$  from  $-\infty$ ), SUB delivers an invalid value at the OUT output and sets ENO to “0”.

### 11.3.4 Multiplication MUL

The MUL function interprets the values present at the IN1 and IN2 inputs as numbers with the specified data type. It multiplies the two numbers, and saves the product at the OUT output. Leaving the permissible range is reported by ENO = “0”.

Floating-point multiplication: In an unauthorized calculation (one of the input values is an invalid floating point number, or you try to multiply  $\infty$  with 0), MUL supplies an invalid value at OUT and sets ENO to “0”.

### 11.3.5 Division DIV

Fixed-point division: the DIV function interprets the values present at the IN1 and IN2 inputs as numbers with the specified data type. It divides the value at input IN1 (dividend) by the value at input IN2 (divisor) and delivers the quotient at the OUT output. This is the integer result of the division. The quotient is zero if the dividend is equal to zero and the divisor is not equal to zero, or if the value of the div-



idend is smaller than the value of the divisor. The quotient is negative if the divisor is negative. A division by zero delivers a value of zero as the quotient, and sets ENO to “0”.

Floating-point division: the value at the IN1 parameter is divided by the value at the IN2 parameter, and the result output at the OUT parameter. With an impermissible calculation (one of the input values is an invalid floating-point number, or you attempt to divide  $\infty$  by  $\infty$  or 0 by 0), DIV delivers an invalid value at the OUT output and sets ENO to “0”.

### 11.3.6 Division with remainder as result MOD

The MOD function interprets the values present at the IN1 and IN2 inputs as numbers with the specified data type. It divides the value at input IN1 (dividend) by the value at input IN2 (divisor) and saves the remainder of the division at the OUT output. The remainder refers to the remaining part of the division, and does not correspond to the decimal positions. With a negative dividend, the remainder is also negative.

A division by zero delivers a value of zero as the remainder, and sets ENO to “0”. The MOD function does not allow tags of data type REAL or LREAL.

### 11.3.7 Generation of absolute value ABS

The ABS function generates the absolute value from the number at the IN parameter and outputs the result at the OUT parameter. With a floating-point number, the sign of the mantissa is set to “0”, even if the number is invalid.

If the allowed number range is exceeded, for example ABS(-128) for data type SINT, or for a valid floating-point number, the ENO output has signal state “0” (Fig. 11.10).

Generation of absolute value, negation									
LAD	<b>Function</b> Data type		FBD	<b>Function</b> Data type		<b>Name</b>	<b>Declaration</b>	<b>Data type</b>	<b>Description</b>
	EN	ENO		EN	OUT	EN	-	BOOL	Enabling input
	IN	OUT		IN	ENO	ENO	-	BOOL	Enabling output
						IN	INPUT	Data type	Input tag
						OUT	OUTPUT	Data type	Result
SCL	<pre> OUT := ABS(IN1); //Generation of absolute value OUT := -IN; //Negation (two's complement)                 </pre>								
<b>Function:</b>				<b>Data type:</b>					
ABS Generation of absolute value				SINT, INT, DINT, REAL, LREAL					
NEG Negation									

Fig. 11.10 Absolute value generation and negation, representation and function

### 11.3.8 Negation NEG

The NEG function reverses the sign of the number at the IN parameter and outputs the result at the OUT parameter. The negation is equivalent to a multiplication by  $-1$ . With a floating-point number, the sign of the mantissa is changed, even if the number is invalid (Fig. 11.10).

If the result is out of the valid number range, e.g.  $\text{NEG}(-128)$  for the data type SINT, the enable output ENO is set to signal state “0”.

### 11.3.9 Decrement DEC, increment INC

The function DEC (decrement) reduces the value at the IN/OUT parameter by 1 as in a subtraction. The function INC (increment) increases the value at the IN/OUT parameter by 1 as in an addition (Fig. 11.11).

Decrementing, incrementing									
LAD	Function Data type		FBD	Function Data type		Name	Declaration	Data type	Description
—	EN	ENO	—	EN	ENO	EN	—	BOOL	Enabling input
—	IN/OUT		—	IN/OUT ENO		ENO	—	BOOL	Enabling output
						IN/OUT	INOUT	Datentyp	Digital tag
SCL	<pre>Variable := Variable + 1; //Incrementing Variable := Variable - 1; //Decrementing</pre>								
<b>Function:</b>					<b>Data type:</b>				
DEC     Decrement					SINT, INT, DINT, USINT, UINT, UDINT				
INC     Increment									

Fig. 11.11 Decrementing and incrementing, representation and function

LAD, FBD: When reaching the lowest and highest numerical value for the data type, the enable output ENO is set to signal state “0”.

SCL: If the allowed number range is exceeded, the ENO tag is set to FALSE (signal state “0”).

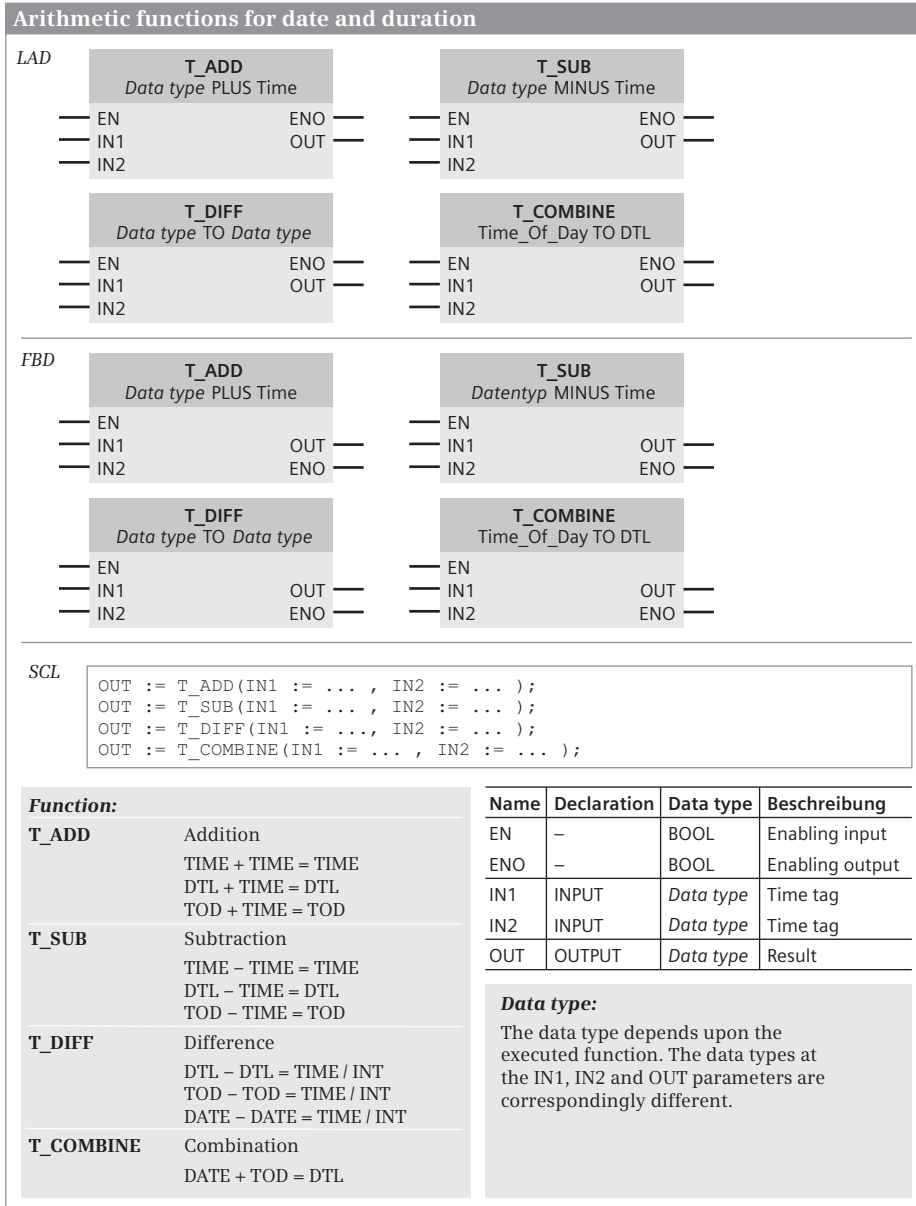
## 11.4 Arithmetic functions for time values

### 11.4.1 Introduction

The arithmetic functions for time values link durations (data type TIME) and points in time (data type DTL) together.

An arithmetic function is executed if the enable input EN is not connected, or if “1” is present at the enable input, or if “current” flows into the enable input. If an error

occurs during calculation, the enable output ENO is set to “0”, otherwise to “1”. If the execution of the function is not enabled (EN = “0”), the calculation does not take place and ENO is also “0”. Zero is then output at the OUT parameter (Fig. 11.12).



**Fig. 11.12** Arithmetic functions for time values, representation and function

The following errors can occur with an arithmetic function for time values:

- ▷ Invalid bit assignment for DTL
- ▷ Violation of permissible range of values for TIME or DTL

#### **11.4.2 Addition T\_ADD**

The T\_ADD function adds the values at the IN1 and IN2 parameters and applies the result to the OUT parameter. The parameters have different data types depending on the calculation:

- ▷ Addition of two durations  $\text{TIME} + \text{TIME} = \text{TIME}$
- ▷ Addition of a duration to a point in time  $\text{DTL} + \text{TIME} = \text{DTL}$
- ▷ Addition of a duration to the time of day  $\text{TOD} + \text{TIME} = \text{TOD}$

T\_ADD sets ENO to “0” when the permissible range of values is violated. The result is limited to the permissible range.

#### **11.4.3 Subtraction T\_SUB**

The T\_SUB function subtracts the value at the IN2 parameter from the value at the IN1 parameter and applies the result to the OUT parameter. The parameters have different data types depending on the calculation:

- ▷ Subtraction of two durations  $\text{TIME} - \text{TIME} = \text{TIME}$
- ▷ Subtraction of a duration from a point in time  $\text{DTL} - \text{TIME} = \text{DTL}$
- ▷ Subtraction of a duration from a time of day  $\text{TOD} - \text{TIME} = \text{TOD}$

T\_SUB sets ENO to “0” when the permissible range of values is violated. The result is limited to the permissible range.

#### **11.4.4 Difference T\_DIFF**

The function T\_DIFF subtracts the value at the IN2 parameter from the value at IN1 parameter and stores the result at the OUT parameter. The difference between two points in time (DTL, TOD, or DATE) is formed. The result is available in the data type TIME or INT.

T\_DIFF sets ENO to “0” when the permissible range of values is violated. The result is limited to the permissible range.

#### **11.4.5 Combine T\_COMBINE**

The T\_COMBINE function summarizes tags together with data types DATE and TIME\_OF\_DAY (TOD), converts the data type to DATE\_AND\_TIME (DTL), and returns the value at the OUT parameter.

The statement does not report any errors.

## 11.5 Mathematical functions

### 11.5.1 Introduction

The following mathematical functions are available:

- ▷ Trigonometric functions: sine (SIN), cosine (COS), tangent (TAN)
- ▷ Arc functions: arc sine (ASIN), arc cosine (ACOS), arc tangent (ATAN)
- ▷ Generation of square (SQR) and square root (SQRT)
- ▷ Exponential function to base e (EXP) and to any base (EXPT)
- ▷ Napierian logarithm (LN)
- ▷ Extract decimal places (FRAC)

All mathematical functions process floating-point numbers.

#### Mathematical functions

LAD	Function <i>Data type</i>		FBD	Function <i>Data type</i>	Name	Declaration	Data type	Description
—	EN	ENO	—	EN	ENO	—	BOOL	Enabling input
—	IN	OUT	—	IN	ENO	—	BOOL	Enabling output
					IN	INPUT	REAL, LREAL	Digital tag
					OUT	OUTPUT	REAL, LREAL	Result

SCL `OUT := Function(IN);`

<b>Function:</b>	SIN Sine	ASIN Arc sine
	COS Cosine	ACOS Arc cosine
	TAN Tangent	ATAN Arc tangent
	SQR Formation of square	EXP Exponential function to base e
	SQRT Extraction of square root	LN Natural logarithm
		FRAC Decimal places

#### Exponentiate to any base EXPT

LAD	EXPT <i>DT1 ** DT2</i>		FBD	EXPT <i>DT1 ** DT2</i>	Name	Declaration	Data type	Description
—	EN	ENO	—	EN	ENO	—	BOOL	Enabling input
—	IN1	OUT	—	IN1	OUT	—	BOOL	Enabling output
—	IN2		—	IN2	ENO	—	<i>Data type 1</i>	Base
					IN2	INPUT	<i>Data type 2</i>	Exponent
					OUT	OUTPUT	<i>Data type 1</i>	Result

SCL `OUT := IN1 ** IN2;`

<b>Function:</b>	EXPT Exponential function to any base	<b>Data type 1:</b> REAL, LREAL
		<b>Data type 2:</b> USINT, UINT, UDINT, SINT, INT, DINT, REAL, LREAL

Fig. 11.13 Mathematical functions, representation and function

A mathematical function is executed if the enabling input EN is unused or if “1” is present at the enabling input (“current” flows into the enabling input). If an error occurs during the calculation, the enabling output ENO is set to “0”, otherwise to “1”. The calculation is not carried out if it has not been enabled (EN = “0”), and ENO is also “0”.

### 11.5.2 Trigonometric functions SIN, COS, TAN

The trigonometric functions generate the sine (SIN), cosine (COS) or tangent (TAN) of the tag at the IN parameter and output it at the OUT parameter. An angle specified in radians as a floating-point number is expected at the IN parameter.

Two units are common for the magnitude of an angle: degrees from 0° to 360° and radians from 0 to  $2\pi$  (where  $\pi = +3.141593e+00$ ). Either can be converted proportionally. For example, the radian measure for a 90° angle is  $\pi/2$ , i.e.  $+1.570796e+00$ . With values greater than  $2\pi$  ( $+6.283185e+00$ ),  $2\pi$  or a multiple thereof is subtracted from this until the input value for the trigonometric function is less than  $2\pi$ .

The enabling output ENO is set to signal state “0” when an invalid floating-point number,  $+\infty$  or  $-\infty$  is present at the IN parameter. The value of IN is then output at the OUT parameter.

### 11.5.3 Arc functions ASIN, ACOS, ATAN

The arc functions generate the arc sine (ASIN), arc cosine (ACOS) or arc tangent (ATAN) of the tags the IN parameter and output the result at the OUT parameter. They are the inverted functions of the respective trigonometric function. They expect a floating-point number within a certain range at the IN input, and return an angle in radians (Table 11.2).

**Table 11.2** Range of arc functions

Function	Permissible range	Returned value
Arc sine ASIN	-1 to +1	$-\pi/2$ to $+\pi/2$
Arc cosine ACOS	-1 to +1	0 to $\pi$
Arc tangent ATAN	Complete range	$-\pi/2$ to $+\pi/2$

The signal state “0” is set at the enabling output ENO if the value at the IN parameter is not in the range  $\pm 1$  (with ASIN or ACOS) or if an invalid floating-point number is present at the IN parameter. An invalid floating-point number is then output at the OUT parameter.

#### 11.5.4 Formation of square SQR

The SQR function forms the square of the value present at the IN input and applies the result to the OUT output.

$$OUT = IN^2$$

The enabling output ENO is set to signal state “0” if the value at the IN parameter or the result of the calculation is an invalid floating-point number. In the first case, an invalid floating-point number is output at the OUT parameter, and in the second case  $+\infty$ .

#### 11.5.5 Extraction of square root SQRT

The SQRT function extracts the square root of the value present at the IN input and applies the result to the OUT output.

$$OUT = \sqrt{IN}$$

If a value less than zero is present at the IN input, SQRT returns an invalid floating-point number. If the value at the IN input is an invalid floating-point number or  $\pm\infty$ , the invalid floating-point number or  $\pm\infty$  is output at OUT. The enabling output ENO is set to signal state “0” in both cases.

#### 11.5.6 Exponentiate to base e EXP

The EXP function calculates the power from base e (= 2.718282e+00) and the value present at the IN input, and applies the result to the OUT output.

$$OUT = e^{IN}$$

The enabling output ENO is set to signal state “0” if the value at the IN parameter or the result of the calculation is an invalid floating-point number. In the first case, an invalid floating-point number is output at the OUT parameter, and in the second case  $+\infty$ .

#### 11.5.7 Calculation of Napierian logarithm LN

The LN function calculates the Napierian logarithm to base e (= 2.718282e+00) of the number present at the IN input and applies the result to the OUT output.

$$OUT = \ln(IN)$$

The enabling output ENO is set to signal state “0” if:

- ▷ The value at the IN1 parameter is zero, negative,  $-\infty$  or a negative, invalid floating-point number.  $-\infty$  is then output at the OUT parameter.
- ▷ The value at the IN1 parameter is  $+\infty$  or a positive, invalid floating-point number. The value of IN1 is then output at the OUT parameter.

The Napierian logarithm is the inverted function of the exponential function:

If  $y = e^x$ , then  $x = \ln y$ .

If you wish to calculate a logarithm, use the equation

$$\log_b a = \frac{\log_n a}{\log_n b}$$

where  $b$  or  $n$  is any base. If  $n = e$  is set, it is possible to use the Napierian logarithm to calculate a logarithm to any base:

$$\log_b a = \frac{\ln a}{\ln b}$$

In the special case for base 10, the equation is:

$$\lg a = \frac{\ln a}{\ln 10} = 0.4342945 \cdot \ln a$$

### 11.5.8 Extracting decimal places FRAC

The FRAC function extracts the decimal places from the number at the IN parameter and outputs the result at the OUT parameter.

$$OUT = FRAC(IN)$$

The enabling output ENO is set to signal state “0” when the value at the IN parameter is an invalid floating-point number or  $\pm\infty$ . A positive, invalid floating-point number is then output at the OUT parameter.

### 11.5.9 Exponentiation to any base EXPT

The EXPT function calculates the power from the base at parameter IN1 and the exponent at parameter IN2 and stores the result at parameter OUT.

$$OUT = IN1^{IN2}$$

The enable output ENO or the tag ENO is set to signal state “0”

- ▷ if the value at parameter IN1 is  $+\infty$  and at parameter IN2 is not  $-\infty$ . Then  $+\infty$  is output at the OUT parameter.
- ▷ if the value at parameter IN1 is  $-\infty$  or negative. Then an invalid floating-point number is output at the OUT parameter (if IN2 is a floating-point number), otherwise  $-\infty$ .
- ▷ if the value at parameter IN1 or IN2 is an invalid floating-point number. Then an invalid floating-point number is output at OUT.
- ▷ if the value at parameter IN1 is 0 (zero) and there is a floating-point number at parameter IN2. Then an invalid floating-point number is output at OUT.



## 11.6 Conversion functions (Conversion of data type)

### 11.6.1 Introduction

If you link tags together, they must have the same data type. This also applies if you assign values or supply function or block parameters. If a tag is not available in the required data type, the data type must be converted. The “simple” conversion functions are available for this.

With the “extended” conversion functions, for example, a number available in STRING format can be converted into a number format or a character sequence into a number sequence. The “extended” conversion functions are usually based on a system or standard block.

The following conversion functions are available:

- ▷ CONV  
Conversion of bit sequences (BYTE, WORD, DWORD) and numerical types (SINT, INT, DINT, USINT, UINT, UDINT, REAL, LREAL) among themselves, conversion of INT to BCD16 and vice versa, conversion of DINT to BCD32 and vice versa
- ▷ ROUND, FLOOR, CEIL, TRUNC  
Conversion of floating-point numbers (REAL, LREAL) into floating-point and fixed-point numbers (SINT, INT, DINT, USINT, UINT, UDINT, REAL, LREAL)
- ▷ SCALE\_X, NORM\_X  
Conversion (scaling) of floating-point numbers (REAL) to floating-point and fixed-point numbers (SINT, INT, DINT, USINT, UINT, UDINT, REAL) and conversion (standardization) of floating-point and fixed-point numbers (SINT, INT, DINT, USINT, UINT, UDINT, REAL) to floating-point numbers (REAL)
- ▷ T\_CONV  
Conversion of a duration (TIME) into a fixed-point number (DINT) and vice versa
- ▷ S\_CONV  
Conversion of floating-point and fixed-point numbers (SINT, INT, DINT, USINT, UINT, UDINT, REAL) into a string (STRING) and vice versa and copying of a string (STRING)
- ▷ STRG\_VAL, VAL\_STRG  
Conversion of floating-point and fixed-point numbers (SINT, INT, DINT, USINT, UINT, UDINT, REAL) into a string (STRING) and vice versa with specification of notation
- ▷ Strg\_TO\_Chars, Chars\_TO\_Strg  
Conversion of a string (STRING) to a character array (ARRAY OF CHAR) and vice versa
- ▷ ATH, HTA  
Conversion of a character sequence in ASCII format into a number sequence in hexadecimal format, and vice versa

### Execution of a conversion function

LAD and FBD: The conversion function is executed if the signal state “1” is present at enable input EN (if “current” is flowing into the enable input) or if the enable input remains unconnected. If an error occurs during conversion, the enable output ENO is set to “0”, otherwise to “1”. If the execution of the function is not enabled (EN = “0”), the conversion does not take place and ENO is also “0”.

SCL: If an error occurs when processing a conversion function, the tag ENO is reset to signal state “0”.

These conversion functions are “explicit” conversion functions, where the bit assignments of the tags change or where conversion errors can occur, for example a conversion from DINT to REAL. These conversions must be programmed. “Implicit” conversion functions also exist which convert a data type without changing the bit assignments and do not signal an error, for example the conversion from BYTE to WORD. These conversions are carried out “automatically”. Further details can be found in Chapter 4.3.2 “Implicit data type conversion” on page 93.

#### 11.6.2 Conversion function CONV

##### Conversion function CONV for fixed-point numbers and bit sequences

The conversion function CONV converts the data type of the tags at parameter IN to another and outputs the result at the OUT parameter.

When setting BYTE, WORD or DWORD, a constant (at the IN parameter) or a tag with elementary data type can be specified which has the set data width (8, 16 or 32 bits).

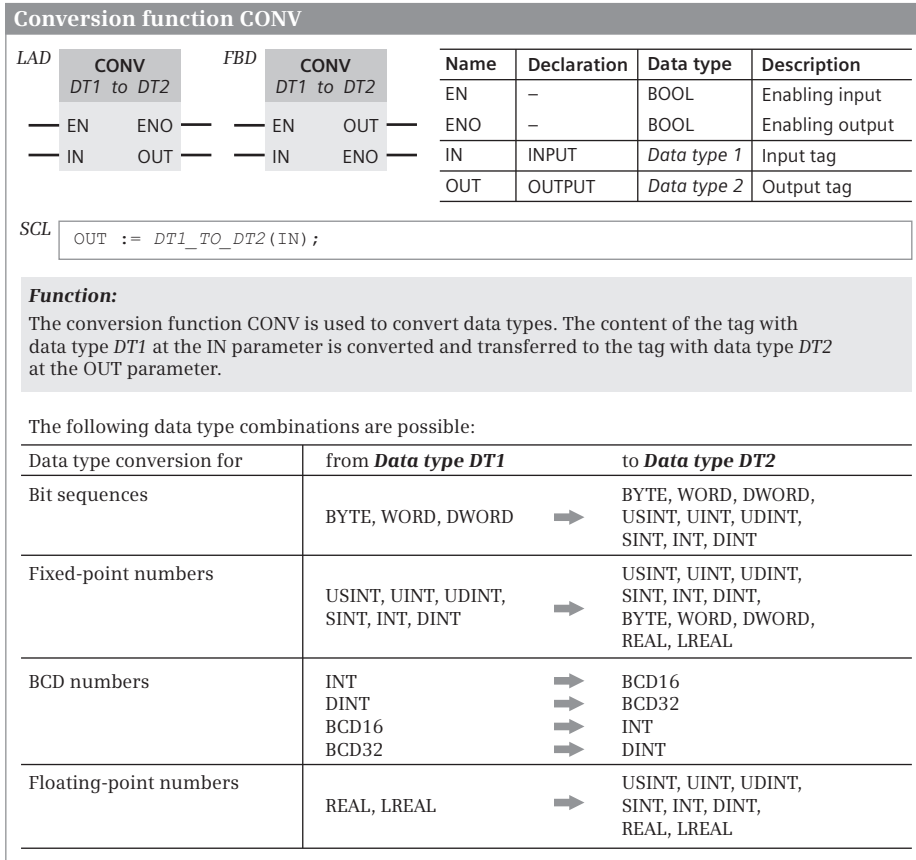
The numerical value of the tag at the IN parameter is transferred right-justified to the tag at the OUT parameter. If “vacant” digits result because the data types have different lengths, they are padded with the sign. If the permissible numerical range at the OUT parameter is exceeded during the conversion, ENO is set to signal state “0”.

##### Conversion function CONV for BCD numbers

The conversion function CONV converts a fixed-point number at the IN parameter into a BCD number and vice versa. The result is available at the OUT parameter.

##### Conversion of INT to BCD16 or of DINT to BCD32

The value at parameter IN is converted to a BCD-coded number with three (BCD16) or seven (BCD32) decades at parameter OUT. The right-justified decades represent the absolute value of the decimal number. The sign is located in the bits 12 to 15 (BCD16) or 28 to 31 (BCD32). If bit 15 or 31 is set to signal state “0”, the sign is positive; bit 15 or bit 31 set to signal state “1” indicates a negative sign. If the number is too large for a conversion into a BCD number, the conversion function sets ENO to “0”. The conversion is not carried out in this case.



**Fig. 11.14** Conversion function CONV, representation and function

**Conversion of BCD16 to INT or of BCD32 to DINT**

The value at the IN parameter is converted into a fixed-point number at the OUT parameter. If the number is too large for conversion to a fixed-point number with the corresponding data type, the conversion function sets ENO to “0”. The conversion is not carried out in this case.

**11.6.3 Conversion functions for floating-point numbers**

The conversion function CONV converts fixed-point numbers into floating-point numbers and vice versa. For conversion from floating-point numbers into fixed-point numbers, there are additional features that differ in rounding behavior (Table 11.3). If an error occurs during conversion, no conversion takes place.

**Table 11.3** Conversion of floating-point numbers

Function	Conversion
CONV	Conversion from USINT, UINT, UDINT, SINT, INT or DINT to REAL or LREAL
	Conversion from REAL or LREAL to USINT, UINT, UDINT, SINT, INT or DINT
CONV	With rounding to the next integer (as ROUND)
CEIL	With rounding to the next higher integer
FLOOR	With rounding to the next smaller integer
ROUND	With rounding to the next integer
TRUNC	Without rounding

### Conversion of fixed-point numbers into floating-point numbers with the CONV function

The conversion function CONV converts a fixed-point number (SINT, INT, DINT, US-INT, UINT, UDINT) at the IN parameter into a floating-point number (REAL, LREAL) and outputs the result at the OUT parameter. If the accuracy suffers during the conversion of DINT or UDINT into REAL, ENO is set to “0”.

### Conversion of floating-point numbers into fixed-point numbers with the CONV function

The conversion function CONV converts a floating-point number at the IN parameter (REAL, LREAL) into a fixed-point number (SINT, INT, DINT, USINT, UDINT) and outputs the result at the OUT parameter. During conversion figures are rounded to the next integer. ENO is set to “0” if during conversion the allowed number range is exceeded or an invalid floating-point number is specified.

### Conversion of REAL into LREAL and vice versa with the CONV function

The conversion function CONV converts a floating-point number at the IN parameter into a floating-point number with a different data type (REAL to LREAL or LREAL to REAL) and outputs the result at the OUT parameter. ENO is set to “0” if the permissible numerical range is violated during the conversion or if an invalid floating-point number is specified.

### Conversion of REAL into fixed-point number with the CEIL function

The conversion function CEIL converts the data type REAL or LREAL of the tag at the IN parameter into a fixed-point data type, and outputs the result at the OUT parameter. During the conversion, CEIL rounds-off to the next largest integer. ENO is set to “0” if the permissible numerical range is violated during the conversion or if an invalid floating-point number is specified.

Conversion functions with rounding-off for floating-point numbers									
LAD	<b>Function</b> DT1 to DT2		FBD	<b>Function</b> DT1 to DT2		<b>Name</b>	<b>Declaration</b>	<b>Data type</b>	<b>Description</b>
	EN	ENO		EN	OUT	EN	–	BOOL	Enabling input
	IN	OUT		IN	ENO	ENO	–	BOOL	Enabling output
						IN	INPUT	Data type 1	Input tag
						OUT	OUTPUT	Data type 2	Output tag
SCL	OUT := <b>Function_DT2</b> (IN);								
<b>Function:</b>									
The ROUND, CEIL, FLOOR and TRUNC functions convert a floating-point number at the IN parameter into a number at the OUT parameter, taking into account three types of rounding-off:									
<b>ROUND</b>	Conversion with rounding-off to the next integer								
<b>CEIL</b>	Conversion with rounding-off to the next largest integer								
<b>FLOOR</b>	Conversion with rounding-off to the next smallest integer								
<b>TRUNC</b>	Conversion without rounding-off								
<b>Data type 1 (DT1):</b> REAL, LREAL									
<b>Data type 2 (DT2):</b> USINT, UINT, UDINT, SINT, INT, DINT, REAL, LREAL									

Fig. 11.15 Conversion functions for floating-point numbers, representation and function

### Conversion of REAL into fixed-point number with the FLOOR function

The conversion function FLOOR converts the data type REAL or LREAL of the tag at the IN parameter into a fixed-point data type, and outputs the result at the OUT parameter. During the conversion, FLOOR rounds-off to the next smallest integer. ENO is set to “0” if the permissible numerical range is violated during the conversion or if an invalid floating-point number is specified.

### Conversion of REAL into fixed-point number with the ROUND function

The conversion function ROUND converts the data types REAL and LREAL of the tags at parameter IN to a fixed-point data type and outputs the result at parameter OUT. During conversion, ROUND rounds to the nearest integer. If the result is exactly between even and odd numbers, the even number is selected: ROUND(0.5) = 0, ROUND(1.5) = 2. ENO is set to “0” if during conversion the allowed number range is exceeded or an invalid floating-point number is specified.

### Conversion of REAL into fixed-point number with the TRUNC function

The conversion function TRUNC converts the data type REAL or LREAL of the tag at the IN parameter into a fixed-point data type, and outputs the result at the OUT parameter. TRUNC returns the whole number part of the number to be converted; the fractional part (the decimal places) is “truncated”. ENO is set to “0” if the permissible numerical range is violated during the conversion or if an invalid floating-point number is specified.

### Overview of rounding-off when converting floating-point into fixed-point

Table 11.4 shows the different effects of rounding when converting floating point to fixed point using the example of the REAL data type in the range between -1 and +1.

**Table 11.4** Types of rounding for converting REAL numbers

Input value		Result as fixed-point number			
REAL	16#	ROUND	CEIL	FLOOR	TRUNC
1.0000001	3F80 0001	1	2	1	1
1.0	3F80 0000	1	1	1	1
0.99999995	3F7F FFFF	1	1	0	0
0.50000005	3F00 0001	1	1	0	0
0.5	3F00 0000	0	1	0	0
0.49999996	3EFF FFFF	0	1	0	0
5.877476E-39	0080 0000	0	1	0	0
0.0	0000 0000	0	0	0	0
-5.877476E-39	8080 0000	0	0	-1	0
-0.49999996	BEFF FFFF	0	0	-1	0
-0.5	BF00 0000	0	0	-1	0
-0.50000005	BF00 0001	-1	0	-1	0
-0.99999995	BF7F FFFF	-1	0	-1	0
-1.0	BF80 0000	-1	-1	-1	-1
-1.0000001	BF80 0001	-1	-1	-2	-1

#### 11.6.4 Conversion functions SCALE\_X and NORM\_X

##### Scaling SCALE\_X

The function **SCALE\_X** maps the floating-point number at the VALUE parameter in the value range of 0.0 to 1.0 to a range of values defined by the range limits at the parameters MIN and MAX. The result is output at the OUT parameter (Fig. 11.16).

*Please note:* The value applied at the VALUE parameter must be within the limits of 0 and 1 (inclusive)! If this is not the case, the value at the OUT parameter may be smaller than MIN or greater than MAX. If it is still within the range permissible for this data type, SCALE\_X does not signal an error (ENO = "1").

The function SCALE\_X reports an error (ENO = "0") if there is an invalid floating-point number at the VALUE parameter (VALUE is then written to the OUT parameter), if the result is outside the range of validity of the data type at the OUT param-

eter and the value at the MAX parameter is less than or equal to that at the MIN parameter (in both cases, OUT is assigned without definition).

### Normalizing NORM\_X

The **NORM\_X** function normalizes the number at the VALUE parameter to the range 0 to 1, based on a value range specified with the MIN and MAX parameters, and returns it as a REAL number at the OUT parameter (Fig. 11.16).

*Please note:* The value applied at the VALUE parameter must be within the limits of MIN and MAX (inclusive)! If this is not the case, the value at the OUT parameter may

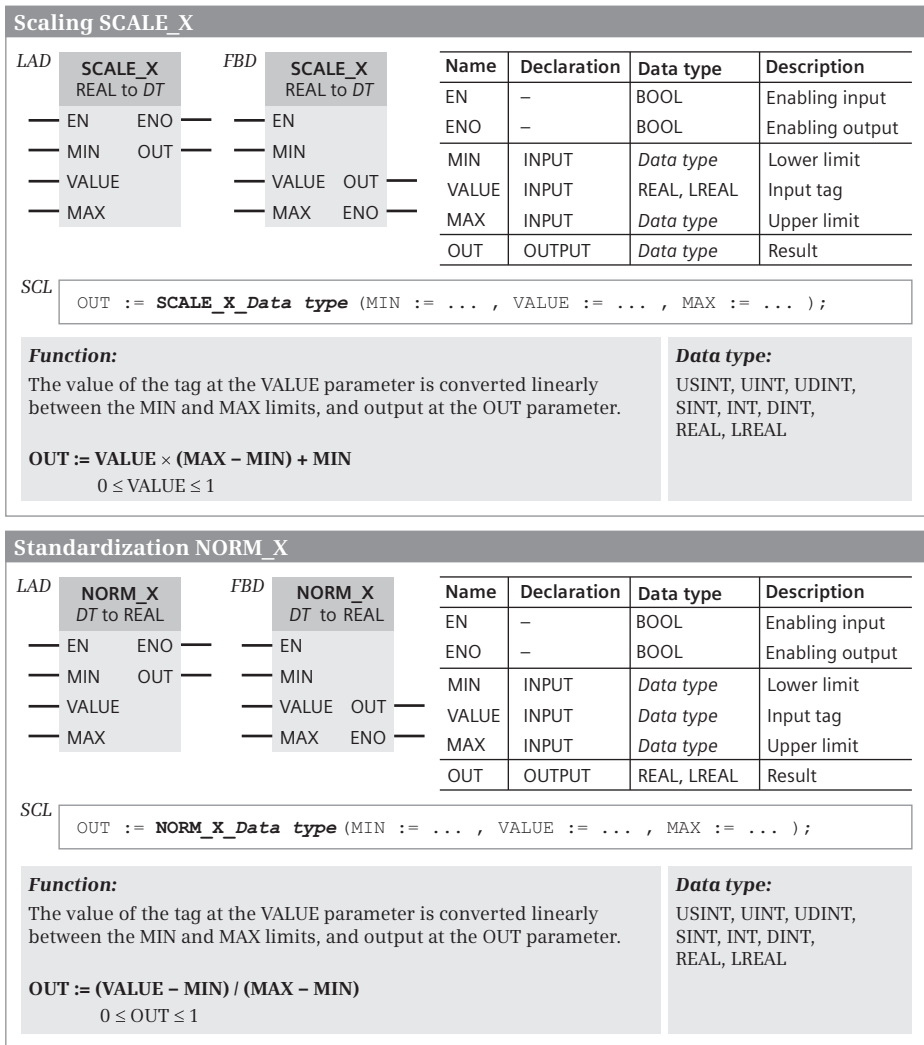


Fig. 11.16 SCALE\_X and NORM\_X functions, representation and function

be smaller than 0 or greater than 1. NORM\_X does not signal an error in this case (ENO = “1”).

The NORM\_X function signals an error (ENO = “0”) if an invalid floating-point number is present at the VALUE parameter (VALUE is then written to the OUT parameter), if the result is outside the valid range of the OUT data type, and if the value at the MAX parameter is less than or equal to that at the MIN parameter (in both cases, OUT is undefined).

### 11.6.5 Conversion function T\_CONV

The conversion function T\_CONV converts the data type of the tag at parameter IN (from TIME to DINT or DINT to TIME) and returns the result at the OUT parameter (Fig. 11.17). T\_CONV does not signal errors.

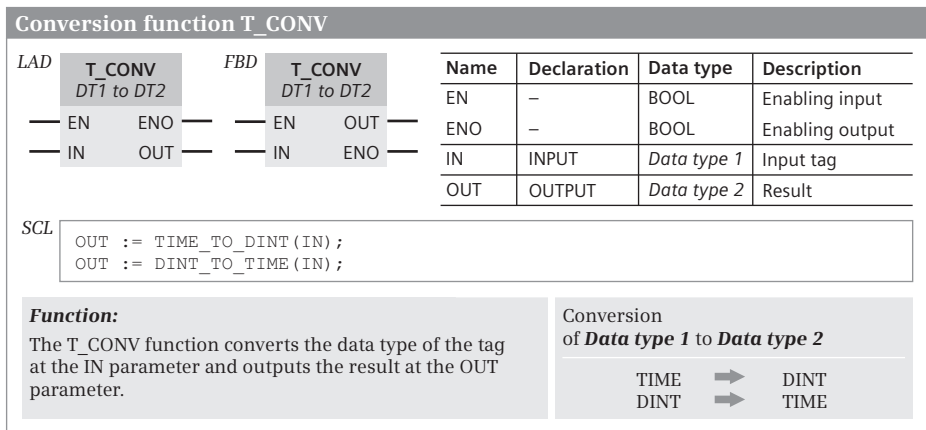


Fig. 11.17 Conversion function T\_CONV, representation and function

### 11.6.6 Conversion function S\_CONV

The conversion function S\_CONV converts a string (STRING) at parameter IN to a number, or a number at parameter IN to a string. The result is output at the OUT parameter. S\_CONV can also be used to copy a string (Fig. 11.18).

#### Conversion of a string into a number

The conversion starts with the first character of the string and ends at the end of the string or at the first character which is not a digit, sign or point.

An error occurs if the structure of the string is invalid with regard to the data type conversion, or if the numerical range of the data type specified at the OUT parameter is exceeded. ENO is then set to “0”, and the OUT parameter is zero.



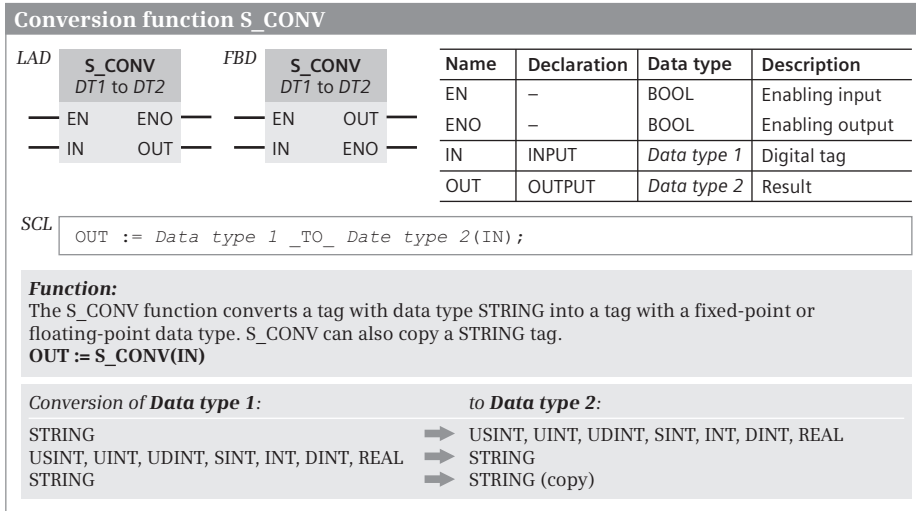


Fig. 11.18 Conversion function S\_CONV

The first character of the string may be a sign (+, -), digit (0 to 9) or space. Leading spaces are ignored. Floating-point numbers with exponential notation are not allowed (“e” or “E” is not recognized as exponent identifiers). A decimal point is used as the separator for specifying fractional numbers, and the comma as the separator for thousands is permissible left of the decimal point and is ignored.

**Conversion of a number into a string**

The output parameter must be supplied with a valid string (plausible length data and characters). The conversion function writes the string starting at the first character (third byte) and tracks the actual length in the second byte. The maximum length entered in the first byte is not changed.

The maximum length must be sufficiently large such that the converted number fits into the string. The minus sign must be considered in the case of signed numbers, and also the decimal point in the case of data type REAL (maximum length >= number of digits + sign + decimal point). In the event of an error, ENO is set to “0”, and the OUT parameter is zero.

If a REAL number is present at the IN parameter which represents -∞ or +∞, the characters 'Inf' (Infinity) are output, or the characters 'NaN' (Not a Number) in the case of an invalid floating-point number.

Following the conversion, the first character in the string is a digit or, with negative numbers, the sign (-). A decimal point is used as the separator with a REAL number as the input (no exponential representation in the string).

### Copying a string

A string at the IN parameter is copied to the OUT parameter if a tag with the data type STRING is connected to it. If the actual length of the string at the IN parameter is greater than the maximum length of the string at the OUT parameter, copying is up to the maximum length of the tag at the OUT parameter, and ENO is set to “0”.

### 11.6.7 Conversion functions STRG\_VAL and VAL\_STRG

#### Conversion of a string into a number (STRG\_VAL)

The string to be converted is at the IN parameter. The first character to be converted is specified at the parameter P, the format to be converted at the parameter FORMAT. The conversion stops when the end of the string is reached or at the first character that is not a digit (0 to 9), a sign (+, -), a point, a comma, or an “e” or “E”. After successful conversion, the position of the last converted character is in P and the result is in OUT. OUT must be filled in with a valid string prior to conversion (Fig. 11.19).

The first character to be converted must be a digit, a sign or a space. Leading spaces are ignored. If a decimal point is used as the separator (bit 0 in FORMAT is “0”), a comma as the separator for thousands is permissible left of the decimal point and is ignored. If a decimal comma is used as the separator (German notation) a point as the separator for thousands is permissible and is ignored.

In the event of an error, zero is output at the OUT parameter, and ENO is set to “0”.

**Conversion function STRG\_VAL**

LAD	STRG_VAL String to DT	FBD	STRG_VAL String to DT	Name	Declaration	Data type	Description
— EN	ENO	— EN		EN	—	BOOL	Enabling input
— IN	OUT	— IN		ENO	—	BOOL	Enabling output
— FORMAT		— FORMAT	OUT	IN	INPUT	STRING	String
— P		— P	ENO	FORMAT	INPUT	BYTE, WORD	Format specificat.
				P	IN_OUT	USINT, UINT	Character position
				OUT	OUTPUT	Data type	Result

SCL `STRG_VAL(IN := ... , FORMAT := ... , P := ... , OUT => ... );`

**Function:**  
The STRING tag at the IN parameter is converted into a numerical value and output at the OUT parameter. The conversion format is defined by the FORMAT parameter.

**Data type:**  
USINT, UINT, UDINT, SINT, INT, DINT, REAL, LREAL

<b>Structure of the FORMAT parameter</b>																D = decimal separator	“0” = point “1” = comma
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	N	D	“0” = decimal fraction “1” = exponential

Fig. 11.19 Conversion function STRG\_VAL

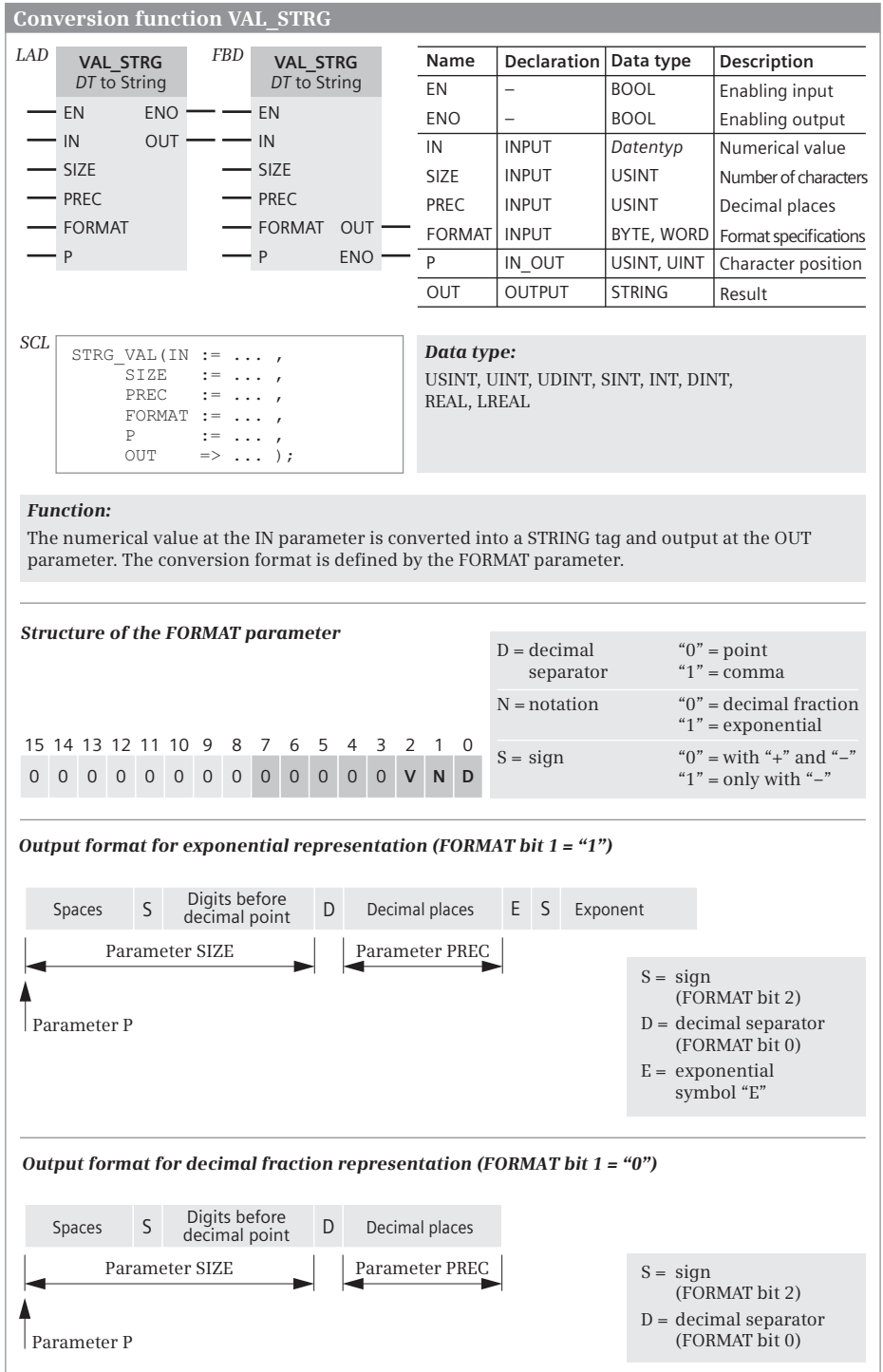


Fig. 11.20 Conversion function VAL\_STRG

### Conversion of a number into a string (VAL\_STRG)

The conversion function VAL\_STRG converts a numerical value at the IN parameter into a string and outputs it at the OUT parameter. OUT must be occupied by a valid STRING tag which is long enough to accommodate the converted value (Fig. 11.20).

The first converted character is written into the STRING tag at the position specified at the P parameter. If P is greater than the current length of the string, spaces are appended up to the P position. Following the conversion, the position of the next character in the string which has not yet been replaced is present in P.

The SIZE parameter specifies the number of digits in front of the decimal point. Leading spaces are inserted if the converted value occupies fewer digits.

The PREC parameter specifies the decimal places, even with a whole number. Example: The number 123 is converted on PREC = 1 into the string "12.3". The maximum value for PREC is 7. If PREC = 0, the decimal separator and the decimal places can be omitted.

The floating point notation places an "E" before the exponent, followed by the sign of the exponent and the exponent without leading zeros. The digits before the "E" are assigned as in fixed-point notation.

If an error occurs during processing of the conversion function, ENO is set to "0". The OUT parameter then contains a value with leading spaces and with a "C" as the last character.

#### 11.6.8 Conversion functions STRG\_TO\_CHARS and CHARS\_TO\_STRG

The function STRG\_TO\_CHARS converts a string with data type STRING into an array with data type ARRAY OF CHAR or ARRAY OF BYTE. The function CHARS\_TO\_STRG converts an array with data type ARRAY OF CHAR or ARRAY OF BYTE into a string with data type STRING. In LAD and FBD, the conversion functions are displayed as EN/ENO boxes (Fig. 11.21).

STRG\_TO\_CHARS converts the string at the parameter STRG with data type STRING into a character sequence. The character sequence is inserted into the array at the parameter CHARS. The array comprises components with data type CHAR or BYTE. The position where the character sequence is inserted is specified by the parameter PCHARS. The parameter CNT indicates the number of inserted characters.

CHARS\_TO\_STRG converts a character sequence into a string with data type STRING and outputs it at the parameter STRG. The character sequence is taken from the array at the parameter CHARS. The parameter PCHARS specifies the position of the first character, while parameter CNT specifies the number of removed characters. If the value is zero at parameter CNT, all characters are copied. The array has the data type ARRAY OF CHAR or ARRAY OF BYTE. Only characters with ASCII coding are accepted.

If an error occurs during conversion (e.g. if the target area is too small to accept the copied characters), ENO is reset to signal state "0".

### Conversion function STRG\_TO\_CHAR

**LAD** **Strg\_TO\_Chars**

**FBD** **Strg\_TO\_Chars**

Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
Strg	INPUT	STRING	String
pChars	INPUT	DINT	Position
Chars	INOUT	VARIANT	Character array
Cnt	OUTPUT	UINT	Number

**SCL**

```
Strg_TO_Chars(Strg := ... , pChars := ... , Cnt => ... , Chars := ... );
```

**Function:**

The string at the STRG parameter is inserted from the position PCHARS in the character array at the CHARS parameter with the data type ARRAY OF CHAR or ARRAY OF BYTE. The CNT parameter indicates the number of inserted characters.

### Conversion function CHARS\_TO\_STRG

**LAD** **Chars\_TO\_Strg**

**FBD** **Chars\_TO\_Strg**

Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
Chars	INPUT	VARIANT	Character array
pChars	INPUT	DINT	Position
Cnt	INPUT	UINT	Number
Strg	OUTPUT	STRING	String

**SCL**

```
Chars_TO_Strg(Chars := ... , pChars := ... , Cnt := ... , Strg => ... );
```

**Function:**

From the character array at the parameter CHARS with the data type ARRAY OF CHAR or ARRAY OF BYTE, CNT characters after the position PCHARS are taken, converted to a string, and output at the STRG parameter.

*Valid characters:*  
all characters with ASCII coding

**Fig. 11.21** Conversion functions STRG\_TO\_CHARS and CHARS\_TO\_STRG

### 11.6.9 Conversion functions ATH and HTA

The ATH function converts a sequence of ASCII-coded characters into a sequence of hexadecimal numbers. The HTA function converts a sequence of hexadecimal numbers into a sequence of ASCII-coded characters. The conversion functions are represented as EN/ENO boxes in the case of LAD and FBD and as function calls with the conversion result as the function value in the case of SCL (Fig. 11.22).

Area pointers in the form of *P#Data\_block.Data\_operand* or *P#Operand* that point to the first byte of the input or output data area are expected at the IN and OUT parameters. Example: P#DB10.DBX12.0. The N parameter with data type INT specifies the number of characters to be converted.

ATH converts a character sequence present in ASCII code into a sequence in hexadecimal code. Only the digits 0 to 9 and the uppercase letters A to F are permissible. An illegal character is converted to zeros. In the event of error at parameter RET\_VAL, an error message is output and ENO is reset to signal state “0”.

HTA converts a character sequence present in hexadecimal code into an ASCII-coded character sequence. HTA does not report any errors.

## 11.7 Shift functions

### 11.7.1 Introduction

A shift function moves the contents of the tags at parameter IN by as many digits as parameter N specifies. The result is output at the OUT parameter (Fig. 11.23).

The shift function is executed if “1” is present at the enabling input (“current” flows into the EN input) or if EN is unused. ENO is then “1”. The shift is not carried out if it has not been enabled (EN = “0”), and ENO is also “0”. A shift function does not report any errors.

### 11.7.2 Shift to right (SHR)

The SHR shift function shifts the contents of the tag present at the IN input bit by bit to the right by the number of positions specified by the shift number at the N input. The bit positions that become free through shifting are filled with the data types SINT, INT, and DINT with the signal state of the most significant bit. The most significant bit contains for the data types SINT, INT and DINT the sign of the fixed-point number.

With a fixed-point number, shifting to the right corresponds to a division by a power to base 2. The exponent is then the shift number. The result of such a division corresponds to the rounded-off integer.

If the shift number = 0, the function is not executed, and the value at the IN parameter is output at the OUT parameter; if it is larger than the data width of the tag, it is shifted by the available places.

**Conversion function ATH**

**LAD**

```

graph LR
    EN1[EN] --- ENO1[ENO]
    IN1[IN] --- RET_VAL1[RET_VAL]
    N1[N] --- OUT1[OUT]
    
```

**FBD**

```

graph LR
    EN2[EN] --- RET_VAL2[RET_VAL]
    IN2[IN] --- OUT2[OUT]
    N2[N] --- ENO2[ENO]
    
```

**SCL**

```


RET_VAL := ATH (
    IN := ... ,
    N := ... ,
    OUT => ... );
    
```

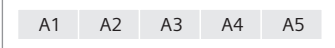
**Data type at the IN parameter:**  
 STRING, ARRAY OF CHAR,  
 ARRAY OF BYTE


**Data type at the OUT parameter:**  
 BYTE, WORD, DWORD, USINT, UINT,  
 UDINT, SINT, INT, DINT, STRING,  
 ARRAY OF CHAR, ARRAY OF BYTE

Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
IN	INPUT	VARIANT	Character sequence
N	INPUT	USINT, UINT, INT	Number of characters (signs)
RET_VAL	RETURN	WORD, DWORD	Error information
OUT	OUTPUT	VARIANT	Number sequence

**Function:**  
 ATH converts the character sequence at the IN parameter to a hexadecimal number and outputs it at the OUT parameter. The parameter N specifies the number of characters to be converted. The digits 0 to 9, the uppercase letters A to F, and the lowercase letters a to f are permissible.

**Parameter N** 

**Parameter IN** 

**Parameter OUT** 

In the case of an odd quantity, a zero is written into the last half-byte.

**Conversion function HTA**

**LAD**

```

graph LR
    EN1[EN] --- ENO1[ENO]
    IN1[IN] --- RET_VAL1[RET_VAL]
    N1[N] --- OUT1[OUT]
    
```

**FBD**

```

graph LR
    EN2[EN] --- RET_VAL2[RET_VAL]
    IN2[IN] --- OUT2[OUT]
    N2[N] --- ENO2[ENO]
    
```

**SCL**

```


RET_VAL := HTA (
    IN := ... ,
    N := ... ,
    OUT => ... );
    
```


**Data type at the IN parameter:**  
 BYTE, WORD, DWORD, USINT, UINT,  
 UDINT, SINT, INT, DINT, STRING,  
 ARRAY OF CHAR, ARRAY OF BYTE

**Data type at the OUT parameter:**  
 STRING, ARRAY OF CHAR,  
 ARRAY OF BYTE

Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
IN	INPUT	VARIANT	Character sequence
N	INPUT	USINT, UINT, INT	Number of characters (signs)
RET_VAL	RETURN	WORD, DWORD	Error information
OUT	OUTPUT	VARIANT	Number sequence

**Function:**  
 HTA converts the hexadecimal number at the IN parameter to a character sequence and outputs it at the OUT parameter. The parameter N specifies the number of characters to be converted. The digits 0 to 9 and the uppercase letters A to F are permissible.

**Parameter N** 

**Parameter IN** 

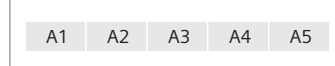
**Parameter OUT** 

Fig. 11.22 Conversion functions ATH and HTA

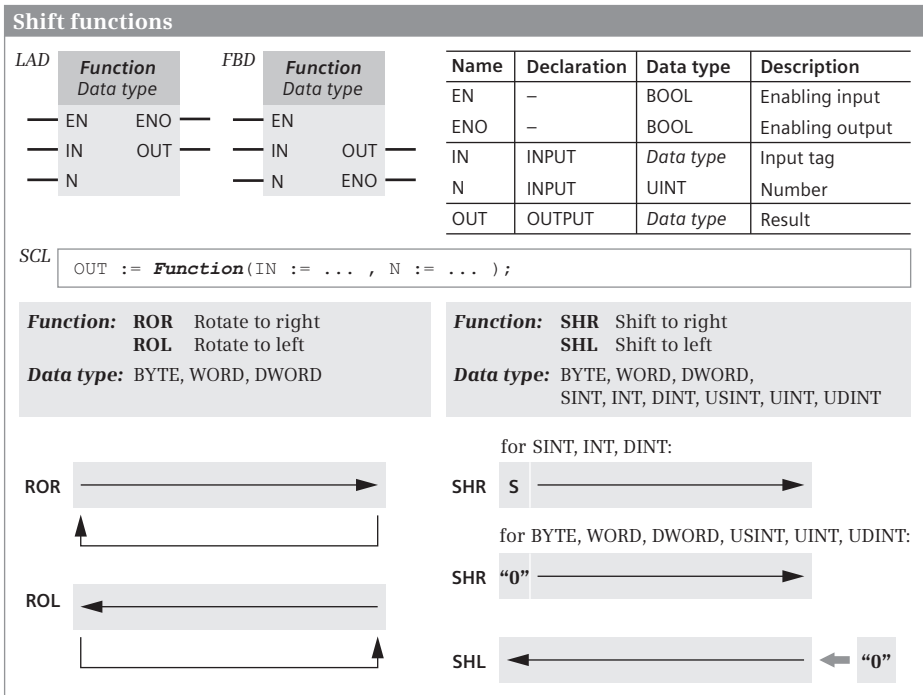


Fig. 11.23 Shift functions, representation and function

### 11.7.3 Shift to left (SHL)

The shift function SHL shifts the contents of the tag at the IN input to the left, bit-by-bit, by the number of positions specified by the shift number at the N input. The bit positions which become vacant during shifting are padded with zeros.

With a fixed-point number, shifting to the left corresponds to a multiplication by a power to base 2. The exponent is then the shift number.

If the shift number = 0, the function is not executed, and the value at the IN parameter is output at the OUT parameter; if it is larger than the data width of the tag, it is shifted by the available places.

### 11.7.4 Rotate to right (ROR)

The shift function ROR shifts the contents of the tag at the IN input to the right, bit-by-bit, by the number of places specified by the shift number at the N input. The bit positions which become vacant during shifting are padded with the bit positions which have been shifted out.

If the shift number = 0, the function is not executed, and the value at the IN parameter is output at the OUT parameter; if it is larger than the data width of the tag, it is rotated by the available places.



### 11.7.5 Rotate to left (ROL)

The shift function ROL shifts the contents of the tag at the IN input to the left, bit-by-bit, by the number of places specified by the shift number at the N input. The bit positions which become vacant during shifting are padded with the bit positions which have been shifted out.

If the shift number = 0, the function is not executed, and the value at the IN parameter is output at the OUT parameter; if it is larger than the data width of the tag, it is rotated by the available places.

## 11.8 Logic functions

### 11.8.1 Introduction

The (digital) logic functions comprise the following functions:

- ▷ AND, OR, XOR  
Word logic operations according to AND, OR, and XOR
- ▷ INV  
Invert
- ▷ DECO, ENCO  
Code bit and set bit number
- ▷ SEL, MUX, DEMUX, MIN, MAX, LIMIT  
Selection functions, minimum and maximum selection, limiter

LAD and FBD: A logic function is performed if the enable input is “1” or if “current” is flowing into the EN input or if EN is not connected. ENO is then “1”. In the event of error, ENO is “0”. If the execution of the function is not enabled (EN = “0”), processing does not take place and ENO is also “0”.

SCL: If the function execution is faulty, the tag ENO is set to FALSE.

### 11.8.2 Word logic operations (AND, OR, XOR)

The word logic operations link the values of two tags at the parameters IN1 and IN2 according to AND, OR, or XOR. Depending on the data type, the logic operation is performed byte by byte, word by word, or doubleword by doubleword. The result is output at the OUT parameter (Fig. 11.24).

The word logic operation generates the result bit by bit. Bit 0 of the IN1 input is linked to bit 0 of the IN2 input; the result is saved in bit 0 of the OUT output. The same logic operation takes place with bit 1, bit 2, etc.

#### AND operation

The AND logic operation links the individual bits of the value present at the IN1 input to the corresponding bits of the value at the IN2 input according to AND.

Digital logic operations AND, OR and XOR																																		
LAD	Function	Data type	FBD	Function	Data type	Name	Declaration	Data type	Description																									
— EN	ENO	—	— EN	—	—	EN	—	BOOL	Enabling input																									
— IN1	OUT	—	— IN1	OUT	—	ENO	—	BOOL	Enabling output																									
— IN2			— IN2	ENO	—	IN1	INPUT	Data type	Input tag 1																									
						IN2	INPUT	Data type	Input tag 2																									
						OUT	OUTPUT	Data type	Result																									
<p>SCL</p> <pre>OUT := IN1 <b>Function</b> IN2;</pre>																																		
<p><b>Function:</b>  <b>AND</b> AND Bit-by-bit AND operation  <b>OR</b> Bit-by-bit OR operation  <b>XOR</b> Bit-by-bit exclusive OR operation</p>					<p><b>Linking of individual bits:</b></p> <table border="1"> <tr> <td>Bit of parameter IN1</td> <td>"0"</td> <td>"0"</td> <td>"1"</td> <td>"1"</td> </tr> <tr> <td>Bit of parameter IN2</td> <td>"0"</td> <td>"1"</td> <td>"0"</td> <td>"1"</td> </tr> <tr> <td>Result with AND</td> <td>"0"</td> <td>"0"</td> <td>"0"</td> <td>"1"</td> </tr> <tr> <td>Result with OR</td> <td>"0"</td> <td>"1"</td> <td>"1"</td> <td>"1"</td> </tr> <tr> <td>Result with XOR</td> <td>"0"</td> <td>"1"</td> <td>"1"</td> <td>"0"</td> </tr> </table>					Bit of parameter IN1	"0"	"0"	"1"	"1"	Bit of parameter IN2	"0"	"1"	"0"	"1"	Result with AND	"0"	"0"	"0"	"1"	Result with OR	"0"	"1"	"1"	"1"	Result with XOR	"0"	"1"	"1"	"0"
Bit of parameter IN1	"0"	"0"	"1"	"1"																														
Bit of parameter IN2	"0"	"1"	"0"	"1"																														
Result with AND	"0"	"0"	"0"	"1"																														
Result with OR	"0"	"1"	"1"	"1"																														
Result with XOR	"0"	"1"	"1"	"0"																														
<p><b>Data type:</b>          BYTE, WORD, DWORD</p>																																		

Fig. 11.24 Word logic operations, representation and function

The individual bits in the result word OUT only have signal state "1" if the corresponding bits of both values to be linked have signal state "1".

Since the bits with signal state "0" at the IN2 input also set these bits in the result to "0", independent of the assignments of these bits at the IN1 input, these bits are referred to as being "masked". This "masking" is the main field of application of the (digital) AND logic operation.

### OR logic operation

The OR logic operation links the individual bits of the value present at the IN1 input to the corresponding bits of the value at the IN2 input according to OR. The individual bits in the result word OUT only have signal state "0" if the corresponding bits of both values to be linked have signal state "0".

Since the bits with signal state "1" at the IN2 input also set these bits in the result to "1", independent of the assignments of these bits at the IN1 input, these bits are referred to as being "displayed". This unmasking is the main field of application of the (digital) OR logic operation.

### Exclusive OR operation

The exclusive OR logic operation links the individual bits of the value present at the IN1 input to the corresponding bits of the value at the IN2 input according to Exclusive OR. The individual bits in the result word OUT only have signal state "1" if just one of the corresponding bits of the two values to be linked has signal state "1". If a bit at the IN2 input has signal state "1", this position in the result has the inverted signal state of the bit at the IN1 input. In the result, only those bits have signal state "1" which have different signal states in the two tags prior to the digital

Exclusive OR operation. The finding of the bits occupied by different signal states together with the negation of the signal states of individual bits is the main field of application of the (digital) Exclusive OR logic operation.

### 11.8.3 Invert (INV)

The inversion negates the value at parameter IN bit by bit and writes it to the tag at parameter OUT. The bit-by-bit negation replaces the zeros with ones and vice versa. The function INV does not report any errors (Fig. 11.25).

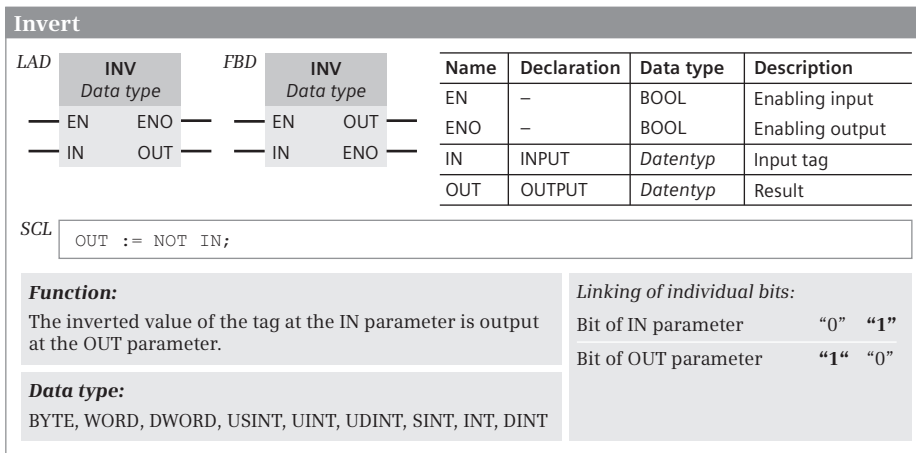


Fig. 11.25 Inverting, representation and function

### 11.8.4 Coding functions DECO and ENCO

The coding functions comprise the DECO functions (converting a binary number into a bit pattern) and ENCO (converting a bit pattern into a binary number). Fig. 11.26 shows these functions.

#### Code bit DECO

DECO sets the bit whose number is at the IN parameter in the bit sequence tag at the OUT parameter. All other bits are set to signal state "0". Depending on the data type at the OUT parameter, only some of the bits are selected in the IN parameter: 3 bits (range 0 to 7) for BYTE, 4 bits (range 0 to 15) for WORD, and 5 bits (range 0 to 31) for DWORD. The function DECO does not report any errors.

SCL: DECO returns the output parameter as function value. Its data type is DWORD by default. If the function value has a different data type, the data type is "attached" to the function name with an underscore. Example:

```
#var_byte := DECO_BYTE(#var_usint);
```

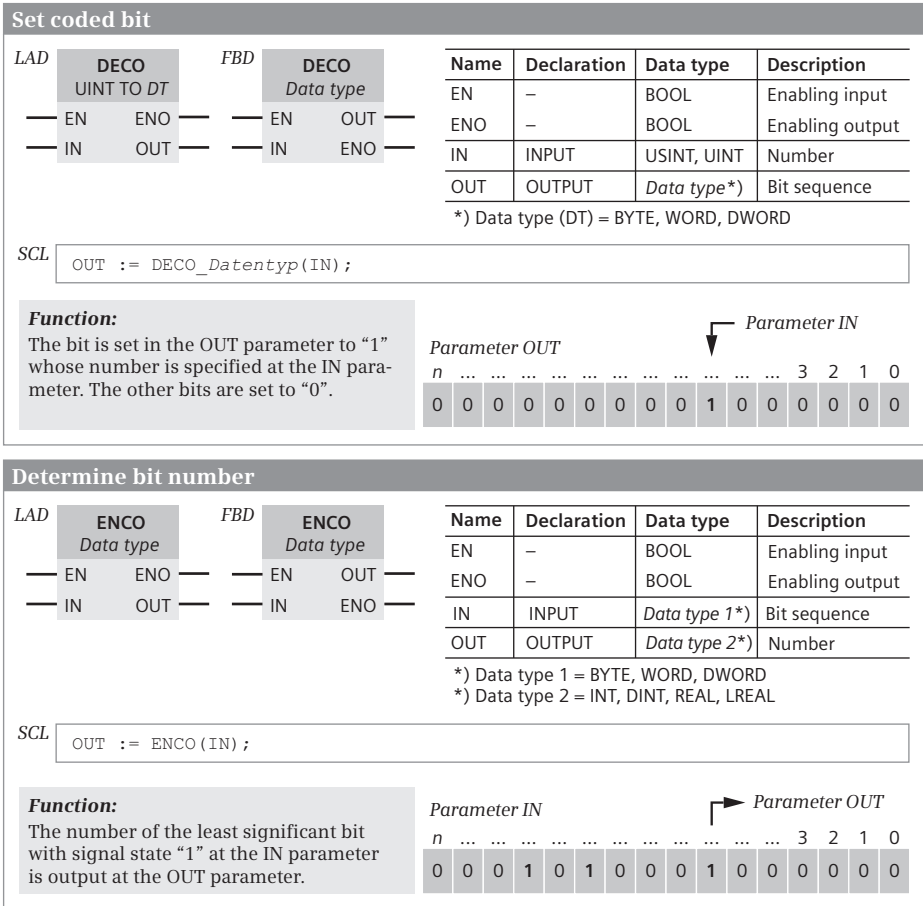


Fig. 11.26 Code bit, representation and function

**Set bit number ENCO**

ENCO searches for the first bit set to signal state “1” in the bit sequence tag at the IN parameter starting from the right (starting with bit number 0) and outputs its number at the OUT parameter. If no bit is set, the number 0 is output at the OUT parameter and signal state “0” at the ENO output.

SCL: ENCO returns the output parameter as function value.

**11.8.5 Selection functions SEL, MUX, and DEMUX**

Depending on a switch (parameter G), SEL selects one of two tag values (parameters IN0 and IN1) and outputs it at parameter OUT. If the signal state is “0” at parameter G, the tag at parameter IN0 is selected; if it is “1”, the tag at parameter IN1 is selected. The SEL function does not report any errors (Fig. 11.27).

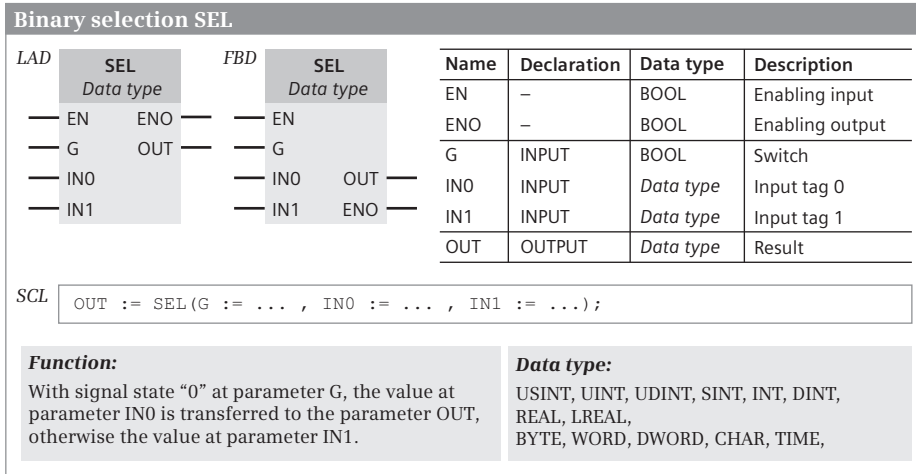


Fig. 11.27 Binary selection SEL, representation and function

Dependent on the value of the K parameter, MUX outputs a tags at the box inputs (parameters IN0 to INn and ELSE) at the OUT parameter. The MUX box is initially offered by the program editor with a choice of two input values (IN0, IN1) and can then be expanded to multiple values. MUX selects from these tag values (IN0 to INn) the one whose number is specified at parameter K. If K = 0, the tag at IN0 is selected; if K = 1, the tag at IN1, etc.

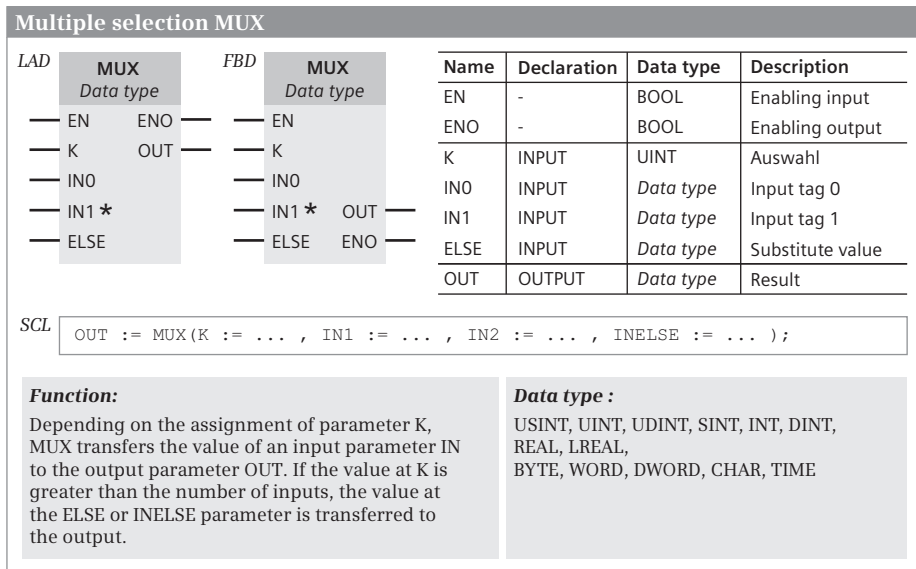


Fig. 11.28 Multiplexing MUX, representation and function

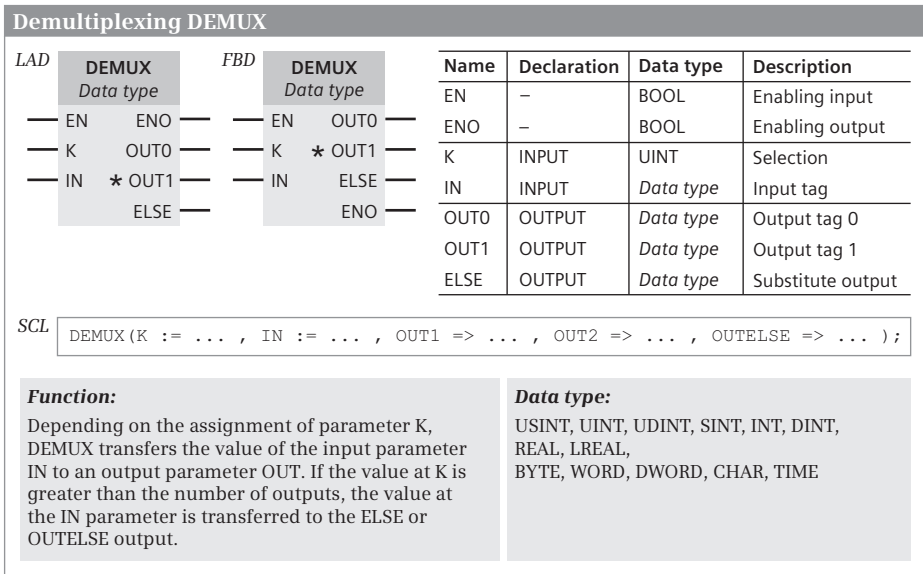


Fig. 11.29 Demultiplexing DEMUX, representation and function

If the value of K is outside the range of input parameters IN0 to INn, the alternative value is output by parameter ELSE; if ELSE is not supplied, OUT remains unchanged. ENO is set to signal state “0” in both cases (Fig. 11.28).

Dependent on the value of the parameter K, DEMUX issues the tag at the input (parameter IN) to a parameter OUT0 to OUTn, or ELSE or OUTELSE. DEMUX is initially offered by the program editor with a choice of two output values (OUT0, OUT1) and can then be extended to multiple values. DEMUX selects from these tag values (OUT0 to OUTn) the one whose number is specified at parameter K. If K = 0, the tag at OUT0 is selected; if K = 1, the tag at OUT1, etc.

If the value of K is outside the range output parameters OUT0 to IOUTn, the value is output alternatively at parameter ELSE or OUTELSE ; if ELSE or OUTELSE is not supplied, ENO is set to signal state “0” (Fig. 11.29).

### 11.8.6 Minimum selection MIN, Maximum selection MAX

The minimum selection **MIN** transfers the lowest of the values at input parameters to parameter OUT. With LAD and FBD up to 100 inputs and with SCL up to 32 inputs can be configured. If there is an invalid REAL number at the input parameters, the function is not executed and the enable output ENO is set to signal state “0” (Fig. 11.30).

The maximum selection **MAX** transfers the highest of the values present at the input parameters to the parameter OUT. With LAD and FBD up to 100 inputs and with SCL up to 32 inputs can be configured. If an invalid REAL number is present at

Minimum selection MIN, Maximum selection MAX									
LAD	Function	Data type	FBD	Function	Data type	Name	Declaration	Data type	Description
— EN	ENO	—	— EN	—	—	EN	—	BOOL	Enabling input
— IN1	OUT	—	— IN1	OUT	—	ENO	—	BOOL	Enabling output
— IN2 *			— IN2 *	ENO	—	IN1	INPUT	Data type	Input tag 1
					—	IN2	INPUT	Data type	Input tag 2
					—	OUT	OUTPUT	Data type	Result
<p>SCL</p> <pre>OUT := <b>Function</b>(IN1 := ... , IN2 := ... );</pre>									
<p><b>Function:</b></p> <p><b>MIN</b> Minimum selection The lowest input value is copied to the OUT output.</p> <p><b>MAX</b> Maximum selection The highest input value is copied to the OUT output</p>						<p><b>Data type:</b></p> <p>USINT, UINT, UDINT, SINT, INT, DINT, REAL, LREAL</p>			

Fig. 11.30 Minimum and maximum selection, representation and function

the input parameters, the function is not executed, and the enabling output ENO is set to signal state “0”.

### 11.8.7 Limiter LIMIT

The limiter LIMIT compares the value at parameter IN with the values of the parameters MIN and MAX. If the value at IN is between the limits, it is output at parameter OUT; it is less than MIN, the value is output at OUT; if it is above MAX, the value goes to MAX. The upper and lower limits can also be assigned constant values (Fig. 11.31).

If there is an invalid REAL number at the parameters MIN, IN, or MAX, an invalid REAL number is output and the enable output ENO is set to signal state “0”. The enable output is also set to “0” if the value at parameter MIN is greater than the value at parameter MAX; the value is then output at parameter IN.

## 11.9 Processing of strings (Data type STRING)

A string can be processed using the following functions:

- ▷ LEN Outputs the length of a string
- ▷ CONCAT Combines two strings together
- ▷ LEFT Outputs the left part of a string
- ▷ RIGHT Outputs the right part of a string
- ▷ MID Outputs the middle part of a string
- ▷ DELETE Deletes part of a string
- ▷ INSERT Inserts characters into a string
- ▷ REPLACE Replaces characters in a string
- ▷ FIND Outputs the position of a searched character

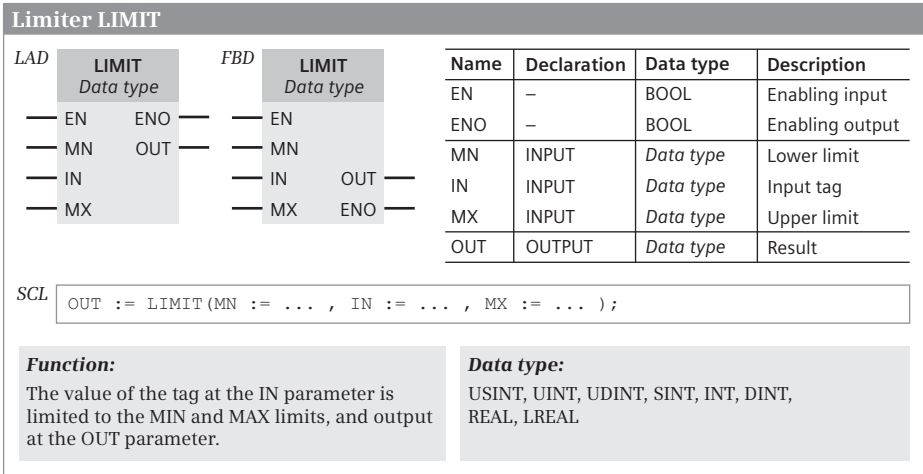


Fig. 11.31 Limiter LIMIT, representation and function

All functions for string processing expect a valid string with plausible values in the length bytes at the parameters with data type STRING (maximum length ≤ 254, actual length ≤ maximum length). Strings which you do not assign with default values during their declaration are automatically assigned as an empty string (actual length = 0) of maximum length (= 254).

Note that you cannot assign default values to strings which you declare in the temporary local data. In this case you must assign a defined value (this can also be an empty string) to a STRING tag in the program before you use the STRING tag together with a function or block.

### 11.9.1 Output length of a string LEN

The LEN function outputs the current length of a string present at the IN parameter at the OUT parameter. For an “empty” string, the current length is zero. The maximum length of a string is 254 characters. LEN returns an error on incorrect parameter assignment (Fig. 11.32).

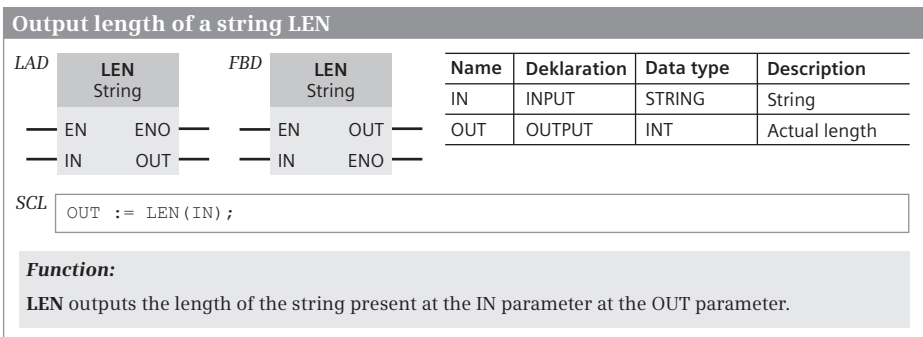


Fig. 11.32 Output length of a string LEN, function and representation



### 11.9.2 Combine strings CONCAT

The CONCAT function combines the STRING tags at parameters IN1 and IN2 into a single tag and outputs it at parameter OUT. The string of IN2 is appended to the string of IN1. If the length of both source strings exceeds the maximum length of the target string, they are truncated to the maximum length and ENO is set to “0” (Fig. 11.33).

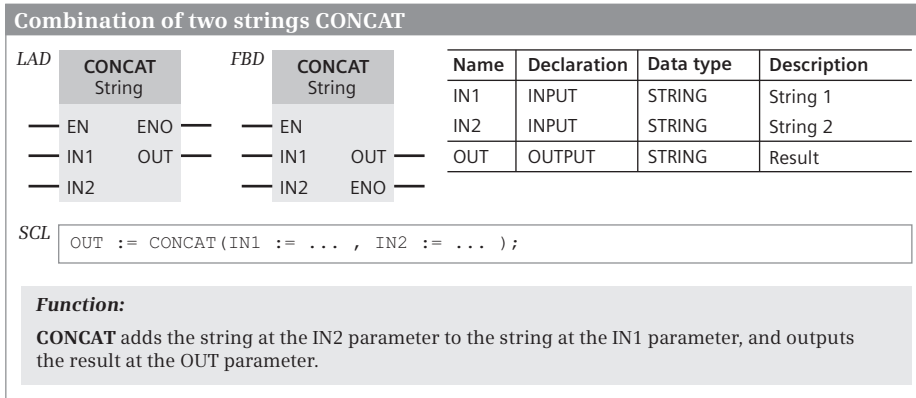


Fig. 11.33 Combining two strings CONCAT, function and representation

### 11.9.3 Output left part of string LEFT

The function LEFT returns the first characters of the string, whose number is specified at parameter L, at the IN parameter and writes it as a STRING tag to the OUT parameter. If L is greater than the current length of the input tags, the input value is output. With an empty string as the input value, an empty string is output. If L is equal to zero or negative, an empty string is output and ENO is set to “0” (Fig. 11.34).

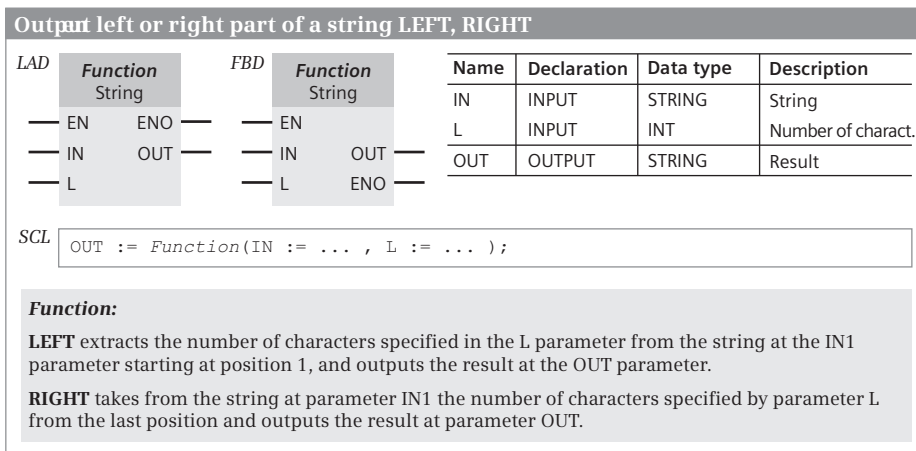


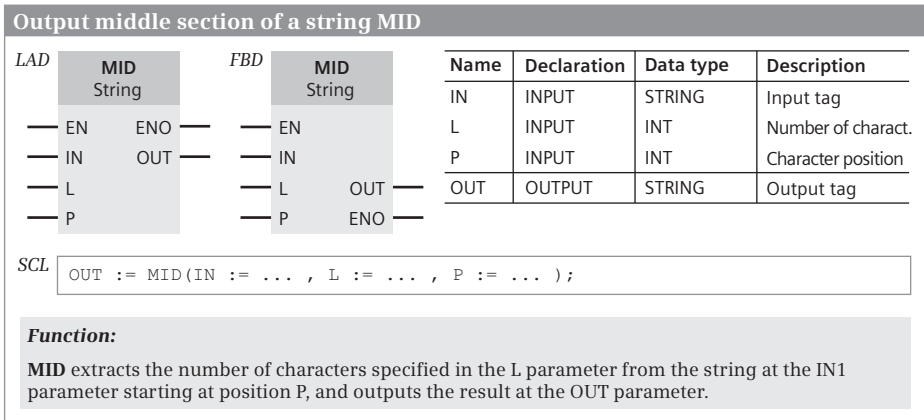
Fig. 11.34 Output left or right part of a string LEFT and RIGHT

### 11.9.4 Output right part of string RIGHT

The **RIGHT** function delivers the last characters (whose number is specified at the L parameter) from the string applied to the IN parameter, and writes them as a STRING tag to the OUT parameter. If L is greater than the current length of the input tags, the input value is output. An empty string is output if the input value is an empty string. If L is equal to zero or negative, an empty string is output and ENO is set to “0” (Fig. 11.34).

### 11.9.5 Output middle part of string MID

The **MID** function extracts a middle section of the string present at the IN parameter, and outputs it at the OUT parameter. The middle section starts at the position specified at the P parameter, and has as many characters as defined by the L parameter (Fig. 11.35).



**Fig. 11.35** Output the middle part of a string MID, function and representation

If the sum of P and L exceeds the current length of the input tags, a string beginning at position P and reaching to the end is output. If P is outside the current length of IN, a blank string is output and ENO is set to “0”. If P or L is zero or negative, a blank string is output and ENO is set to “0”.

### 11.9.6 Delete part of a string DELETE

The **DELETE** function removes part of the string at the IN parameter and outputs the “collapsed” remainder at the OUT parameter. The removed part begins at the character position specified by parameter P and has as many characters as specified in parameter L (Fig. 11.36).

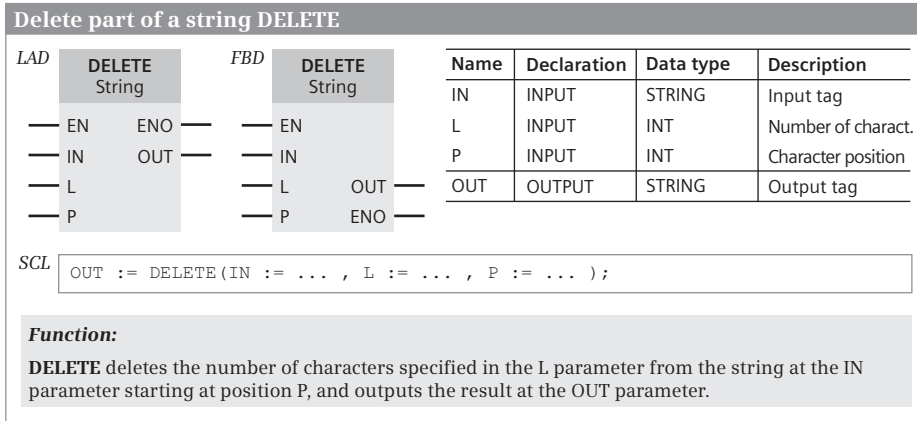


Fig. 11.36 Delete part of string DELETE, function and representation

If L is equal to zero, the input string is output. If P is greater than the current length of the input tag, this tag is output and ENO is set to “0”. If the sum of P and L is greater than the current length of the input tag, the string is deleted up to the end. If L is negative or if P is zero or negative, an empty string is output and ENO is set to “0”.

### 11.9.7 Insert string INSERT

The INSERT function inserts the string at the IN2 parameter into the string at the IN1 parameter and outputs the result at the OUT parameter. Parameter P specifies the position from which the insertion is to take place (Fig. 11.37).

If P is equal to zero or negative, an empty string is output and ENO is set to “0”. If P is greater than the current length of IN1, IN2 is appended to IN1, and ENO is set

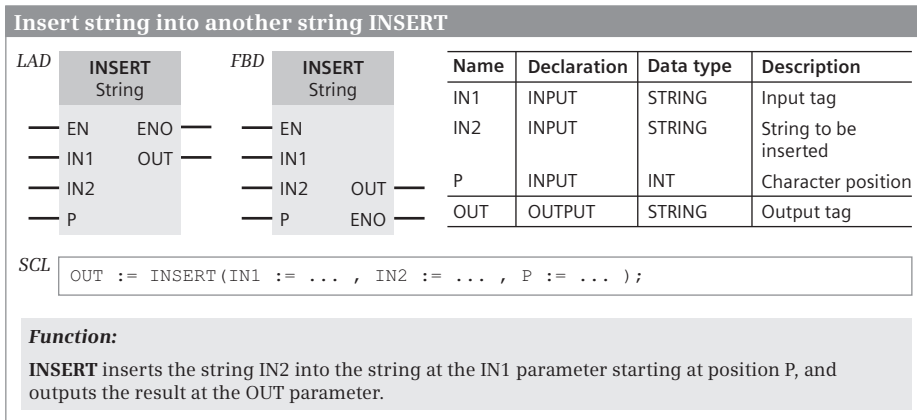


Fig. 11.37 Insert string INSERT, function and representation

to “0”. If the new string is longer than the maximum length permitted by the output string, the characters are entered up to the permissible length, and ENO is set to “0”.

### 11.9.8 Replace part of string REPLACE

REPLACE replaces characters present in the string at the IN1 parameter by the string at the IN2 parameter and outputs the result at the OUT parameter. Beginning with the position specified by parameter P, the characters are replaced for a length given at parameter L (Fig. 11.38).

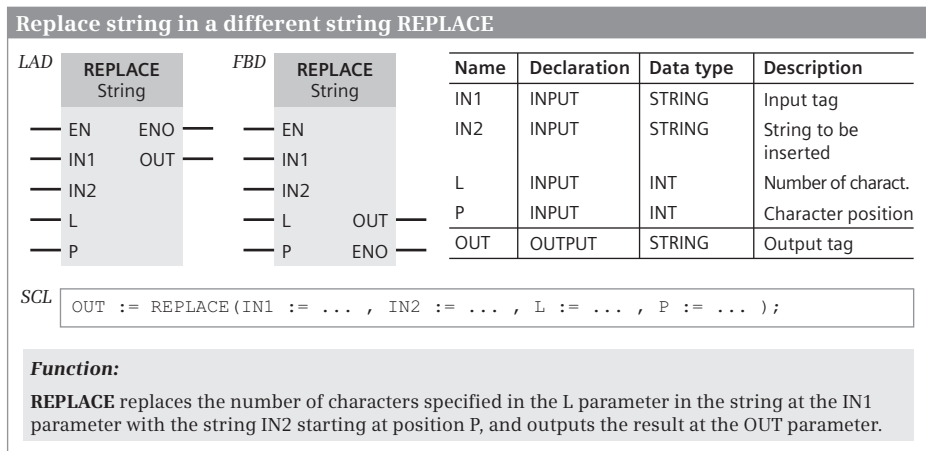


Fig. 11.38 Replace part of string REPLACE, function and representation

If L is equal to zero, the IN2 string is inserted into the IN1 string from position P without deleting characters in IN1. If P is equal to 1, the first L characters of IN1 are replaced by IN2. If P is greater than the current length of IN1, IN2 is appended to IN1, and ENO is set to “0”. If L is negative or if P is zero or negative, an empty string is output and ENO is set to “0”. If the new string is longer than the maximum length of the output string, only the maximum length is output, and ENO is set to “0”.

### 11.9.9 Find part of string FIND

the FIND function determines the position of the string at parameter IN2 in the string at parameter IN1 and outputs it at the OUT parameter. The position of the first character is output if a match has been found. If IN2 is not contained in IN1, zero is returned (Fig. 11.39).

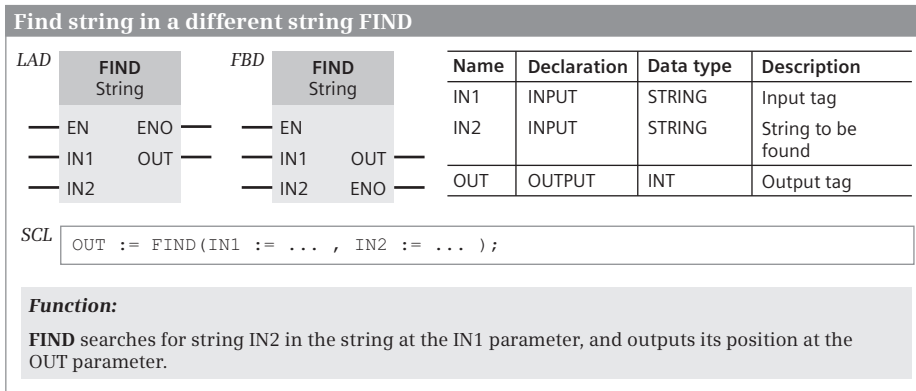


Fig. 11.39 Find part of string FIND, function and representation

## 11.10 Calculating with the CALCULATE box in LAD and FBD

The CALCULATE box can link digital tags with arithmetic, mathematical, and logical functions in a complex expression with each other. You define the tags to be linked as input parameters of the box and specify the data type of the expression (the output parameter). The logic operation function is specified in a dialog (Fig. 11.40).

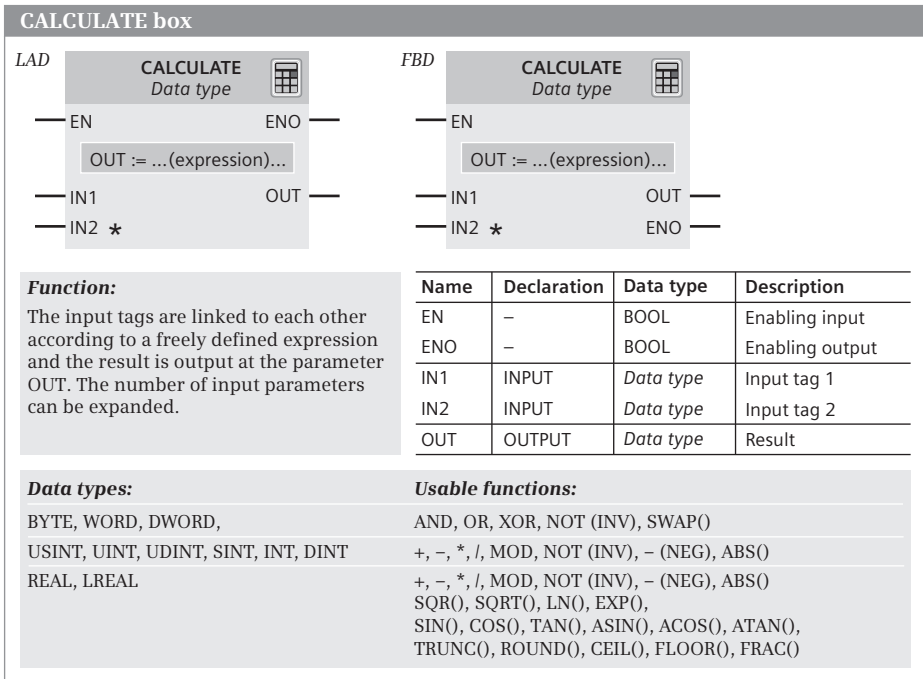
In the basic state, the box contains two inputs. The number of inputs can be increased. The inputs are numbered without gaps. Not all inputs must be used in the expression. If, when defining the expression, a (new) input with the next available number is used, the input is automatically added. In the expression, only the tags defined as input parameters may be used.

After inserting the CALCULATE box, select the data type of the expression (the output parameter OUT) from a drop-down list. The input parameters will automatically be given the same data type. The actual operands must be of the same data type or a data type that can be converted using implicit conversion to the data type of the input parameter. Example: If you select data type LREAL for the expression, an actual operand with the data type REAL or LREAL can be created at an input.

In the expression, the input tags can be linked with each other according to their data type. The order of the linking can be controlled using brackets.

### Linking of bit strings

Input parameters with the data types BYTE, WORD, and DWORD can be used in connection with the word logic operations AND (digital AND logic operation), OR (digital OR logic operation), and XOR (digital exclusive OR logic operation). A bit string can be inverted with the NOT operator (one's complement formation INV). With SWAP(), the bytes of a bit string can be replaced.



**Fig. 11.40** CALCULATE box, representation and function

### Linking of fixed-point numbers

Input parameters with data types USINT, UDINT, SINT, INT, and DINT can be used as a result (MOD) in conjunction with the arithmetic functions add (+), subtract (-), multiply (\*), division (/), and division with the rest. From fixed-point numbers, the one's complement INV (operator: NOT), the two's complement (operator: - , multiplication by - 1) and the absolute value (ABS() ) are formed.

### Linking of floating-point numbers

Input parameters with the data types REAL and LREAL can, in addition to the arithmetic functions add (+), subtract (-), multiply (\*) and divide (/), also be used in connection with the mathematical functions SQR (generate square), SQRT (generate square root), LN (generate natural logarithm) and EXP (generate exponential value), with the trigonometric functions SIN (sine), COS (cosine), TAN (tangent), ASIN (arcsine), ACOS (arccosine) and ATAN (arctangent), with the conversion functions ROUND (round), TRUNC ("truncate" decimal places), FRAC (determine decimal places), CEIL (generate next highest fixed-point number) and FLOOR (generate next lowest fixed-point number) as well as in connection with the two's complement (-, multiplication by -1) and absolute-value generation ABS.

## 12 Program flow control

This chapter describes the functions for controlling program execution, independent of the programming language as far as possible. The Chapters 7 “Ladder logic LAD” on page 209, 8 “Function block diagram FBD” on page 246, and 9 “Structured Control Language SCL” on page 284 describe how you can program the functions using the individual programming languages and what special features exist.

The jump functions allow program branches dependent upon or independently of the logic operation, a numerical value, or a comparison with a numerical value. The control statements which are only present with SCL for controlling program execution are described in Chapter 9.6.3 “Control statements” on page 307.

The block functions are used to structure the user program. A block end function prematurely terminates the processing in a block, the block call functions call another block for processing. A block call is parameterizable and can be used multiple times with different parameters. A block call can be controlled using the enable input EN and the enable output ENO.

### 12.1 Jump functions

#### 12.1.1 Overview

Jump functions interrupt linear program execution in the block and continue at a different point in the program in the block. This point is identified by means of a jump label which you specify in the jump statement as the jump destination. The jump function and destination must be in the same block. The jump destination or label must be unique within a block. It is permissible to jump to a destination from more than one position. Both forward and backward jumps are possible with regard to the direction of program execution.

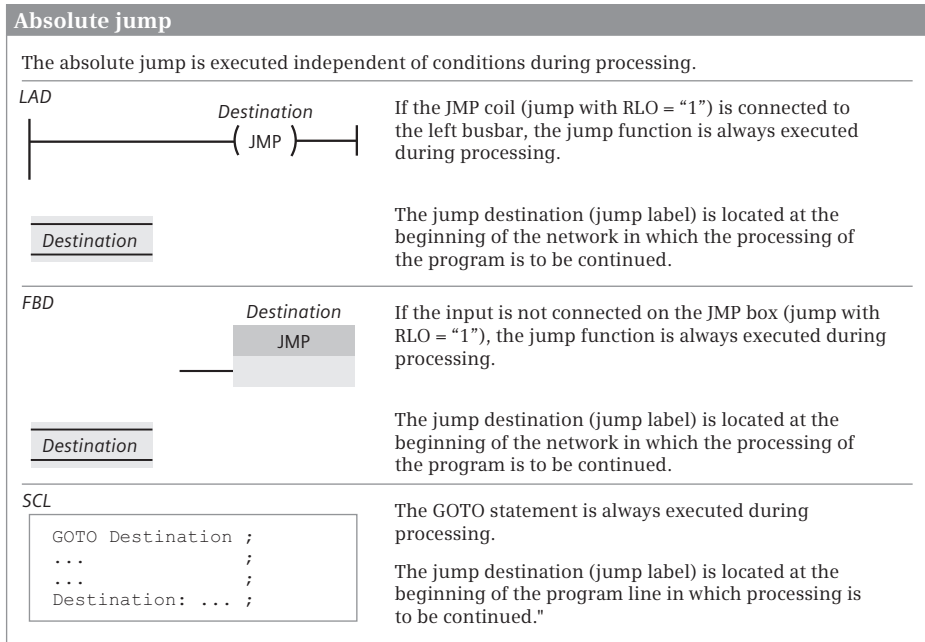
The following jump functions are available:

- ▷ Absolute jump, is always executed during processing.
- ▷ Conditional jump, is executed depending on the result of logic operation.
- ▷ Jump list, branches depending on a value
- ▷ Jump distributor, branches depending on a comparison result

Jump list and jump distributor are available in the described form for LAD and FBD. SCL uses control statements for program branches.

### 12.1.2 Absolute jump

An absolute jump is carried out independent of conditions. The absolute jump interrupts linear program execution and continues it at a the specified jump label. Fig. 12.1 shows the implementation of the jump function in the various programming languages.



**Fig. 12.1** Absolute jump function JMP or GOTO

#### Absolute jump function JMP (LAD and FBD)

The jump functions consist of the jump statement (coil or box) and a jump label. The jump label (jump destination) identifies the entry point in the block at which program execution is continued when the jump function has been processed.

The jump function JMP is connected to the left-hand power rail or does not have a preceding logic operation. The entry point can only be positioned at the start of a network. Only one jump statement ore one block end function is permissible per network.

#### Absolute jump GOTO (SCL)

The jump function GOTO interrupts the linear program execution and continues it at a different position in the block. If statements form a defined block, e.g. a program body within a program loop, the jump destination must be within this state-



ment block if the GOTO statement is also within the statement block, and it cannot jump to this statement block “from the outside”.

A jump label must always be followed by a statement. A “dummy statement” is also permissible:

```
Label: ; //Entry with “dummy statement”
```

### 12.1.3 Conditional jump

A conditional jump is executed depending on the result of logic operation (RLO). Program execution is continued at the specified jump label with RLO = “1” or with RLO = “0” depending on the jump function. Fig. 12.2 shows the implementation of the conditional jump function in the various programming languages.

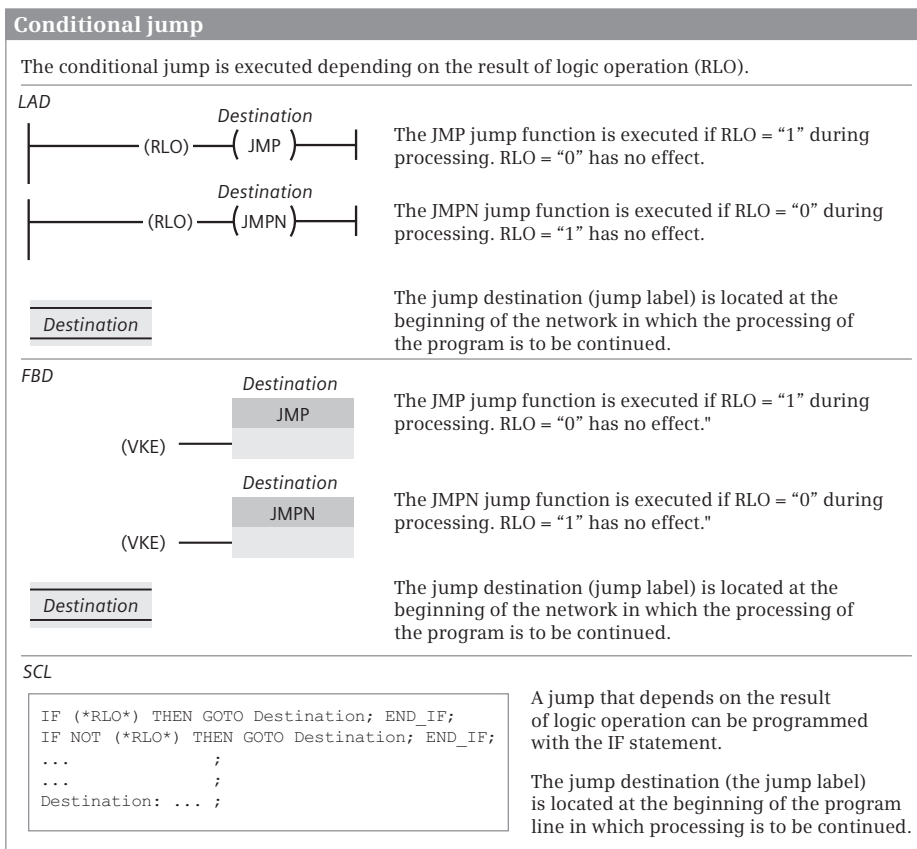


Fig. 12.2 Conditional jump functions JMP and JPN

### Conditional jump functions JMP and JMPN (LAD and FBD)

The jump functions consist of the jump statement (coil or box) and a jump label. The jump label (jump destination) identifies the entry point in the block at which program execution is continued when the jump function has been processed.

JMP branches to the entry point if the preceding logic operation is fulfilled; JMPN branches to the entry point if the preceding logic operation is not fulfilled. The jump functions terminate a current path or a logic operation. The entry point can only be positioned at the start of a network.

### Conditional jump functions with SCL

With SCL, the dependency of the jump statement GOTO on the result of logic operation can be emulated, for example, by an IF statement:

```
IF (* Condition *) THEN GOTO Destination; END_IF;
```

Further information on the IF statement can be found in Chapter 9.6.3 “Control statements” on page 307.

#### 12.1.4 Jump list JMP\_LIST

In LAD and FBD, the jump list JMP\_LIST allows jumping to a program part in the block depending on a numerical value. In SCL, the CASE statement can be used for this functionality (Fig. 12.3).

#### Jump list with LAD and FBD

With the JMP\_LIST box you define a list of jump labels. The two output parameters DEST0 and DEST1, where you specify one jump label each, are displayed when the box is inserted. The list can be expanded up to 99 jump labels. The jump destinations are in the same block at the beginning of a network.

JMP\_LIST executes a jump dependent upon the value at parameter K. If K has a value of zero, processing of the program continues at the point defined by the jump label at parameter DEST0. If K has a value of one, the jump label at parameter DEST1 is selected, etc. If the value of K is greater than the number of defined jump labels, processing of the program continues in the next network.

The enable input EN can be used to control processing of the JMP\_LIST box. The box is present alone in a network.

#### Jump list with SCL

With SCL, the dependency of the jump statement GOTO on a numerical value can be emulated, for example, by a CASE statement. For more information on the CASE statement, refer to Chapter 9.6.3 “Control statements” on page 307.

**Jump depending on a numerical value**

The jump list is used to define jump functions which are executed depending on a numerical value.

**LAD**

**FBD**

Name	Declaration	Data type	Description
EN	–	BOOL	Enabling input
K	INPUT	UINT	Selection
DEST0	–	–	Jump label 0
DEST1	–	–	Jump label 1

*Destination*

The jump destinations (jump labels) are located at the beginning of the network in which the processing of the program is to be continued.

---

**SCL**

```

CASE K OF
0   : GOTO Destination1;
1   : GOTO Destination2;
...
ELSE : GOTO ... ;
END_CASE;
...
Destination1: ... ;
...
Destination2: ... ;
        
```

A jump that depends on a numerical value can be programmed with the CASE statement.

The jump destinations (the jump labels) are located at the beginning of the program line in which processing is to be continued.

**Fig. 12.3** Jump list JMP\_LIST

### 12.1.5 Jump distributor SWITCH

In LAD and FBD, the jump distributor SWITCH allows jumping to a program part in the block depending on a comparison with a numerical value. In SCL, the IF statement can be used for this functionality (Fig. 12.4).

#### Jump distributor with LAD and FBD

With the SWITCH box you define a list of jump labels. The two output parameters DEST0 and DEST1, where you specify one jump label each, are displayed when the box is inserted. The list can be expanded up to 99 jump labels. The jump destinations are in the same block at the beginning of a network.

SWITCH executes a jump dependent upon a comparison with the parameter K. The value to which K is to be compared is specified by you at a (comparison) input parameter. You can select the comparison function from a drop-down box at this input parameter. An additional (comparison) input parameter is provided for each newly inserted jump label.

You can set the data type of parameter K and of the (comparison-) inputs at the SWITCH box. Tags with the data types BYTE, WORD, and DWORD can only be compared to determine “equal to” or “not equal to”.

### A jump that depends on a comparison with a numerical value

The jump distributor is used to define jump functions which are executed depending on a comparison with a numerical value.

LAD		FBD		Name	Declaration	Data type	Description
				EN	-	BOOL	Enabling input
				K	INPUT	Data type	Selection
				== *	INPUT	Data type	Comparison value 0
				== *	INPUT	Data type	Comparison value 1
				DEST0	-	-	Jump label 0
				DEST1	-	-	Jump label 1
				ELSE	-	-	Jump label x

\*) Selection of the type of comparison from a drop-down menu

#### Data type:

BYTE, WORD, DWORD, USINT, UINT, UDINT, SINT, INT, DINT, REAL, LREAL, TIME, DATE, TOD

The jump destinations (jump labels) are located at the beginning of the network in which the processing of the program is to be continued.

Destination

#### SCL

```
IF K <Comparison> Value0 THEN GOTO Destination1; END_IF;
IF K <Comparison> Value1 THEN GOTO Destination2; END_IF;
...
;
GOTO ... ; //ELSE branch
...
;
...
;
Destination1: ... ;
...
;
Destination2: ... ;
```

A jump that depends on a comparison with a numerical value can be programmed with IF statements.

The jump destinations (jump labels) are located at the beginning of the program line in which processing is to be continued.

**Fig. 12.4** Jump distributor SWITCH

If the first comparison is fulfilled, processing of the program continues at the point defined by the jump label at parameter DEST0. If the second comparison is fulfilled, the jump label at parameter DEST1 is selected, etc. If none of the comparisons is fulfilled, processing of the program continues at the jump label specified at parameter ELSE. If ELSE is not assigned, the next network is processed in this case.

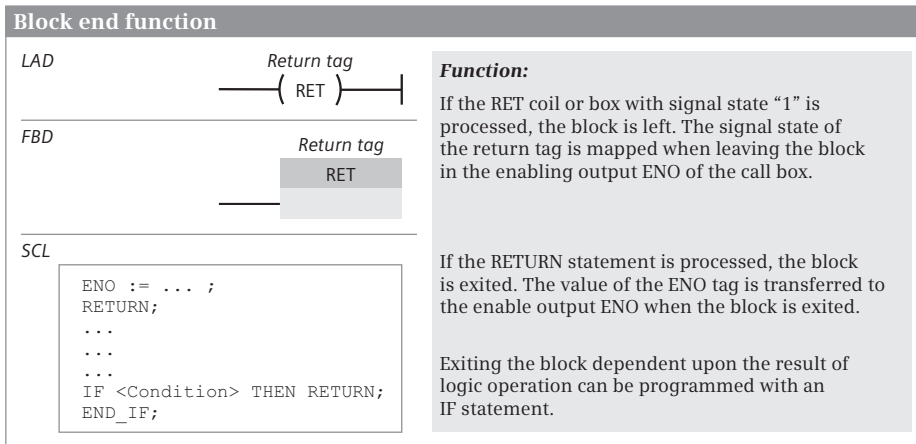
The enable input EN can be used to control processing of the SWITCH box.

### Jump distributor with SCL

With SCL, the dependency of the jump statement GOTO on a comparison with a numerical value can be emulated, for example, by an IF statement. For more information on the IF statement, refer to Chapter 9.6.3 “Control statements” on page 307.

## 12.2 Block end function

You can use the block end function RET or RETURN to prematurely terminate processing in a block (Fig. 12.5).



**Fig. 12.5** Block end function RET or RETURN

### Block end with LAD and FBD

The block end function is programmed as a RET coil or RET box at the end of a network. If the preceding logic operation has been completed, the block is left (conditional block end). A return is made to the previously processed block in which the block call was present. If an organization block is terminated, a branch is made to the operating system. If the preceding logic operation has not been completed, the next network in the block is processed.

If the RET coil is connected to the left-hand power rail or if the RLO is fixed at “1” at the box input or is unused, the block is always left (absolute block end). A subsequent network can then only be processed if it has a jump label and is accessed by a jump function.

In the return tags, a signal state can be saved, which is mapped to the ENO output at the call box (see Chapter 12.4 “EN/ENO mechanism” on page 417). To set the return tags, double-click on the RET function and select *Ret* from the drop-down box (RLO, corresponds to the result of logic operation), *Ret True* or *Ret False* for a constant value, or *Ret Value* for a tag. If an organization block ends with the block end function, the signal state of the return tags has no meaning.

In a network with a jump function JMP or JMPN, there can be no Block end function. The block end function can be programmed multiple times in a block. The block end function is not needed to end a block. It does not need to be programmed at the end of a block.

### Block end with SCL

With RETURN the currently processed block is exited without conditions. A conditional block end can be programmed using the IF statement. If an organization block is terminated, processing is continued in the CPU's operating system.

RETURN transfers the signal state of the ENO tag to the enable output of the exited block. If an organization block is exited via RETURN, the signal state of ENO has no meaning. Programming of RETURN at the end of the block is optional.

## 12.3 Calling of code blocks

### 12.3.1 Introduction

If a logic block is to be edited, it must be “called up”. With LAD and FBD, the block call consists of the call box which contains the name (symbolic address) of the called block, the enable input EN, the enable output ENO, and the parameter list. With SCL, the block name is specified during the block call, followed by the parameter list. The enable input EN and the enable output ENO can be inserted into the parameter list of a function block call.

Following processing of the call function, the CPU continues program execution in the called block. The block is processed up to a block end function or up to its end. The CPU then returns to the calling block and continues processing of this block after the call function.

An organization block cannot be called; it is started by the operating system depending on events. If an organization block is terminated, the CPU continues to work in the operating system.

You can pass data on to the called block for processing, or accept data from the called block. The data is transferred by the block parameters.

You can use the enable input EN to structure the block call depending on the result of logic operation. The called block can report faulty processing to the calling block via the enable output ENO. Further details can be found in Chapter 12.4 “EN/ENO mechanism” on page 417. The block properties, the design of the block interface, and the transfer of block parameters are described in Chapters 5.3 “Programming blocks” on page 125 and 5.4 “Calling blocks” on page 137.

### 12.3.2 Calling a function FC

Calling a function (FC) is depicted in LAD and FBD with the call box. In SCL, it is depicted with the block name and the parameter list (Fig. 12.6).

The prerequisite for calling a function (FC) is that the function must already be in the user program. You program the call by dragging the block from the project tree under *Program blocks* to the opened block. To call a function located in a

Calling a function (FC)		
<p>A <b>function (FC)</b> has a function value (return value) with the preset name <code>RET_VAL</code>. This name can be changed, however. This function value can be “deactivated” for the interface declaration if it is occupied with the data type <code>VOID</code>. Any other data type “activates” the function value.</p> <p>All block parameters must be supplied with actual parameters when called.</p>		
<p><b>LAD</b></p>	<p>An FC block is called in LAD by an EN/ENO box. The input and in/out parameters are present on the left-hand side of the call box in the order of their declaration, and the output parameters on the right-hand side. If the function value is “activated”, it is represented as the first output parameter”</p>	
<p><b>FBD</b></p>	<p>An FC block is called in FBD by an EN/ENO box. The input and in/out parameters are present on the left-hand side of the call box in the order of their declaration, and the output parameters on the right-hand side. If the function value is “activated”, it is represented as the first output parameter”</p>	
<p><b>SCL</b></p> <pre>"FC Name" (   In1 := ... ,   In2 := ... ,   Out1 =&gt; ... ,   Out2 =&gt; ... );  #Variable := "FC Name" (   In1 := ... ,   In2 := ... ,   Out1 =&gt; ... ,   Out2 =&gt; ... );</pre>	<p>An FC block is called in SCL by its name. This is followed by the parameter list in parentheses. The parameters are specified in the order of their declaration, each separated by a comma.</p> <p>If the function value is “activated”, the block call responds like a tag with the value and the data type of the function value. It can then be assigned to a tag, for example, or used in an expression.</p>	

**Fig. 12.6** Calling a function (FC)

library, open the library in the task card and drag the block to the working area. When calling functions, you must provide all of the available parameters (see Chapter 5.4.2 “Calling a function (FC)” on page 139).

### Calling a function (FC) with LAD and FBD

With the EN input you can structure the block call depending on conditions. If the EN input leads directly to the left-hand power rail or if it is not connected, the call is an absolute call and is always executed. If EN has a preceding logic operation, the block call is only executed if the preceding logic operation is fulfilled.

You label the parameters of the called block with the current tags for the call, with absolute or symbolic addressing. If an input parameter is of data type `BOOL`, the parameter should be preceded by

- ▷ A contact or a current path (LAD) or
- ▷ A binary tag or binary logic operation (FBD).

A Boolean output parameter cannot be connected further.

The calling of functions with a function value is identical to the calling of functions without a function value. The function value (with the declaration RETURN and a data type not equal to VOID) is depicted as the first output parameter.

### Calling a function (FC) with SCL

You call a function (FC) without a function value with your name. This is followed by the parameter list in round brackets. You must assign values to all block parameters; the sequence is defined by the declaration.

The call of a function (FC) with function value must be handled in the SCL program like a tag which has the data type of the function value. The call function is then present in the assignment or expression instead of the function value. The call function is comprised of the block name, followed by the parameter list in parentheses.

With a function (FC) you cannot use the implicitly defined enable input EN. You can program an FC call depending on a condition without a function value using the IF statement. If you wish to use the implicitly defined enable output ENO, add it to the parameter list as the last parameter.

#### 12.3.3 Calling a function block (FB)

Calling a function block (FB) is depicted in LAD and FBD with the call box. In SCL, it is depicted with the instance name and the parameter list (Fig. 12.7).

The prerequisite for calling a function block is that the function block must already be in the user program. You program the call by dragging the block in the project tree from the *Program blocks* folder to the opened block. To call a function block located in a library, open the library in the task card and drag the block to the working area. Select the type of call in the following dialog.

In the *Call options* dialog, click on the *Single instance* button if the data of the function block is to be saved in a separate data block (in the instance data block). If you call an additional function block in a function block, you can save the data of the called function block in the instance data block of the calling function block. In this case, select the *Multi-instance* button in the *Call options* dialog.

When calling a function block you only have to supply the block parameters that are saved as pointers. The block parameters that are not supplied retain their current value (see Chapter 5.4.3 “Calling a function block (FB)” on page 140).

### Calling a function block with LAD and FBD

With the EN input you can structure the block call depending on conditions. If the EN input leads directly to the left-hand power rail or if it is not connected, the call is

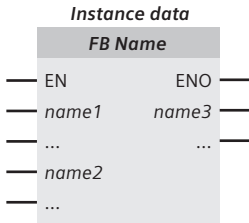


### Calling a function block (FB)

A **function block FB** stands for a program section (a block) with its own data which is present in an instance data block. If the instance data is in a separate data block, one refers to a "single instance". If the instance data is in the instance data block of the calling function block (if this is a "multi-instance"), one refers to a "local instance".

In/out parameters with structured data type and block parameter with parameter type have to be supplied. Supplying of the other block parameters is optional.

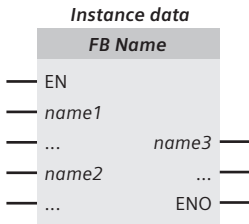
#### LAD



An FB block is called in LAD by an EN/ENO box. The input and in/out parameters are present on the left-hand side of the call box and the output parameters on the right side, in the order of their declaration in each case.

The name of the call instance is shown above the call box. In the case of a single instance, this is the instance data block. In the case of a local instance, this is the instance name in the static local data of the calling function block.

#### FBD



An FB block is called in FBD by an EN/ENO box. The input and in/out parameters are present on the left-hand side of the call box and the output parameters on the right side, in the order of their declaration in each case.

The name of the call instance is shown above the call box. In the case of a single instance, this is the instance data block. In the case of a local instance, this is the instance name in the static local data of the calling function block.

#### SCL

```
"DB Name" (
  In1 := ... ,
  In2 := ... ,
  Out1 => ... ,
  Out2 => ... );

#Instance Name (
  In1 := ... ,
  In2 := ... ,
  Out1 => ... ,
  Out2 => ... );
```

When called as a single instance, the name of the instance data block is specified. When called as a local instance, the instance name is specified.

The list of block parameters follows in parentheses, in the order of their declaration and separated by a comma. Only the block parameters which are supplied need be listed.

**Fig. 12.7** Calling a function block

an absolute call and is always executed. If EN has a preceding logic operation, the block call is only executed if the preceding logic operation is fulfilled.

You can label the parameters of the called block with the current tags for the call, absolutely or symbolically addressed. If an input parameter has data type BOOL, place the following in front of this parameter:

- ▷ a contact or a current path (LAD) or
- ▷ a binary tag or a binary logic operation (FBD).

A Boolean output parameter cannot be further linked.

## Calling a function block with SCL

When calling a function block, you can use the implicitly defined enable input EN to make the call dependent on a condition. If you want to use the enable input EN, add it to the parameter list as the first parameter.

If you wish to use the implicitly defined enable output ENO, add it to the parameter list as the last parameter.

## 12.4 EN/ENO mechanism

Block calls and functions (instructions) in which a runtime error can occur have an enable input EN and an enable output ENO. If the enable input has signal state “1” (TRUE), the block or function is not processed.

**Table 12.1** Schematic diagram for setting of enable output ENO

Is EN connected?				
YES			NO	
Is EN = “1”?			Block/function being processed	
YES		NO		
Block/function being processed		Block/function not being processed		
Has an error occurred?			Has an error occurred?	
YES	NO		YES	NO
ENO is set to “0”	ENO is set to “1”	LAD and FBD: ENO is set to “0” SCL: ENO is set to “1”	ENO is set to “0”	ENO is set to “1”

The signal state of the enable output ENO can be controlled for a user block by the user program. For a function (statement), the ENO output has signal state “1” (TRUE) if processing is problem-free. If an error occurs during processing, e.g. number range overflow during an arithmetic function, the ENO output is set to signal state “0” (FALSE).

The the EN input has signal state “0” (FALSE), with LAD and FBD the ENO output is also set to “0” (FALSE); with SCL the ENO output is set to “1” (TRUE). Table 12.1 shows how the ENO output is controlled.

The enable input EN and the enable output ENO are not block parameters, but statement sequences which the program editor generates automatically before and after all calls depending on EN and ENO. With an SCL block, it is necessary to activate the attribute *Automatically set ENO* to ensure that the necessary statement sequences are generated during compilation.

Functions with an EN/ENO mechanism in the *Statements* folder in the program elements catalog are the mathematical functions, moves, converters, digital logic operations, move and rotate, all functions in the *Expanded statements* folder and all block calls.

#### 12.4.1 EN/ENO mechanism with LAD and FBD

A contact or a binary tag or a preceding logic operation upstream of the enable input EN controls the calling of the EN/ENO box. If the enable input EN is connected to the left busbar or remains unconnected, the EN/ENO box is always processed.

You can use the properties of EN and ENO to connect several boxes into a sequence, where the enable output ENO leads to the enable input EN of the next box (Fig. 12.8). Thus, for example, the entire sequence can then be “deactivated” (no box is processed, if the binary tag “Enable” in the example has signal state “0”) or the rest of the sequence is no longer processed if a box reports an error.

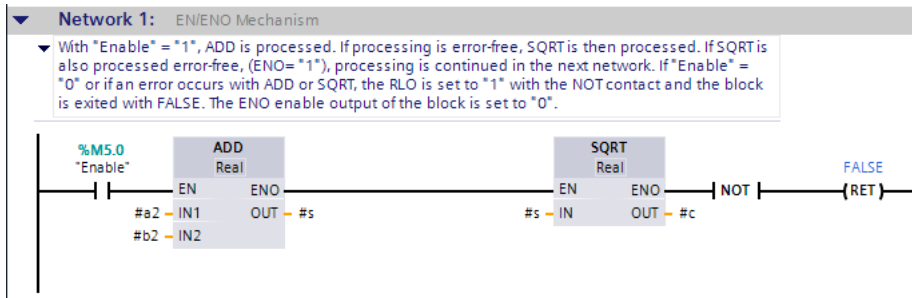


Fig. 12.8 Series connection of ENO and EN

#### 12.4.2 EN/ENO mechanism with SCL

In SCL, the enable input EN and the enable output ENO are not included in the template of the function call or block call during programming. If you want to use the enable input EN when calling a function block, add it to the parameter list as the first parameter. If you would like to use the enable output ENO during a block call, insert it as the last parameter in the parameter list. Example of calling a function block as local instance:

```
#Instance_data (EN := #Enable,
                IN1 := ... ,
                IN2 := ... ,
                OUT => ... ,
                ENO => #Error);
```

If an error has occurred during block processing in the example, the tag #Error is set to FALSE; otherwise, it is set to TRUE. If the tag #Enable has the value FALSE in the example, the block call is not carried out. The tag #Error is set to TRUE in this case.

### 12.4.3 EN/ENO for user blocks

#### Controlling the ENO output in LAD and FBD

With the block end function RET, you can influence the signal state of the enable output ENO: The release output ENO takes over the signal state of the return tag. If, for example, you want to set the enable output ENO to signal state “0” due to an error found in the program, exit the block with RET and FALSE as the return value.

An example for this can be found in the Chapters 7.6 “Functions for program flow control (LAD)” on page 241 and 8.6 “Functions for program flow control (FBD)” on page 279.

#### Controlling the ENO output in SCL

When the block is exited, the enable output ENO assumes the value of the block-local tags ENO. A prerequisite is the activated attribute *Automatically set ENO*. ENO has the value TRUE at the start of the block. An erroneously carried out statement, a numerical range overflow in an arithmetic function, for example, sets ENO to FALSE. If the block is now exited, the enable output ENO assumes the value FALSE.

The use of the block-local tags ENO is described in detail in Chapter 9.6.1 “Working with the ENO tag” on page 305.

## 13 Online operation, diagnostics and debugging

One refers to online operation or online mode if a programming device is connected to a PLC or HMI station and an online connection has been established. An online connection is required in order to upload the user program to the CPU, to test it in the CPU during runtime, or to find hardware faults using diagnostic functions.

The connection between a programming device and a PLC station is made over Industrial Ethernet. The mechanical connection (networking) and the logical connection (definition of the transmission protocols) are not configured. Only the network addresses – the addresses of the PROFINET interfaces of the two devices – must be matched to each other.

In online mode, STEP 7 Basic changes the display of the user interface: the title bars of the windows are displayed in orange. In the project tree, the objects of the stations which are connected online are assigned symbols which indicate their operating or diagnostics state.

You can use the online and diagnostics tools, for example, to control the operating mode of the CPU, to set the time on the CPU, and fetch the diagnostic information, e.g. read the diagnostics buffer. The online and diagnostics tools support you in troubleshooting during commissioning.

The user program which you have created offline can be transferred to the CPU in online mode. When carried out for the first time, all configuration data and the complete user program are transferred, subsequently only the modified configuration data and program blocks. The transfer is always possible in the STOP operating mode of the CPU. If certain prerequisites are met, it can also take place in the RUN operating mode.

You can compare the online and offline versions of a block. Modifications to a block are always carried out in the offline version which you then transfer to the CPU in the STOP mode.

Two functions are available for testing the user program: the program status and the watch tables. You use the program status to monitor the program execution directly on the control functions. The watch tables contain tags whose values you can read and modify (control) during runtime or also set permanently (force).

The program editor also allows you to display the user program in a CPU without a corresponding offline project being present. If you then wish to edit the blocks, you must first upload the online project into the offline data management.

## 13.1 Connecting a programming device to the PLC station

The programming device can only exchange data with a PLC station if it is addressed in the same subnet and has a node address which is different from that of the PLC station. The IP addresses of the programming device and PLC station must therefore be identical in the part whose bits are occupied by “1” in the subnet mask, and different in the remaining part. You can find information on the structure of the IP address and the subnet mask in Section “IP address and subnet mask” on page 73.

If the programming device already has an address different from that of the PLC station, STEP 7 Basic sets a “temporary” IP address on the programming device. This temporary IP address is deleted again when Windows is shut down.

### 13.1.1 IP addresses of the programming device

#### Determining and setting network addresses using Windows tools

You can edit the network addresses of the programming device using the *Network connections* tool (Windows XP) or *Network and enable center* (Windows 7) in the Windows Control Panel. Open the Control Panel – for example from the Windows desktop via *Start > Control Panel* – and start the tool. Then double-click to select the LAN or WLAN connection that is used.

In the displayed status window *Status of ...*, click on the *Details* button located in the *General* or *Network support* tab. The currently active IP address and the subnet mask are displayed, for example. SIMATIC S7 supports the Internet protocol Version 4 with the 4-byte long IPv4 address.

The connection status is displayed in the *General* tab. Click here on the *Properties* button. In the *Properties of ...* window, select the *Internet Protocol (TCP/IP)* entry in the *This connection uses the following items* field, and then click on the *Properties* button (Fig. 13.1).

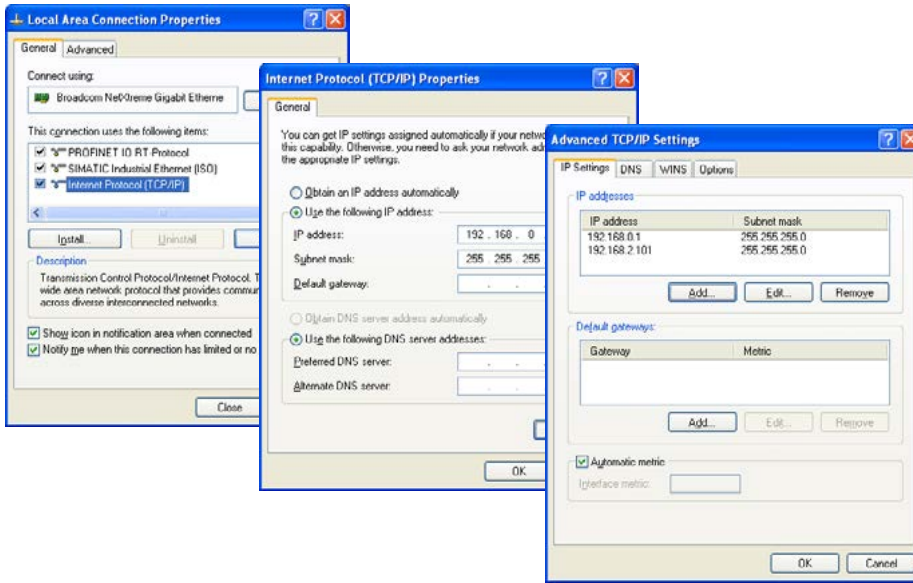
In the following dialog *Internet Protocol (TCP/IP) Properties* you can set the IP address and subnet mask by selecting the *Use the following IP address:* option.

If you want to enter an additional IP address, for example for the SIMATIC project, click on the *Advanced* button. In the advanced settings, enter an additional IP address and the subnet mask in the *IP Settings* tab after clicking on the *Add* button.

#### Setting of access point

When installing STEP 7 Basic, the *Set PG/PC interface* tool is created in the Windows Control Panel. This allows the user to check the access point to the Ethernet network and to reset it if necessary.

Open the *Set PG/PC interface tool*, for example from the Windows desktop using *Start > Control Panel*. The *Access path* tab should show *S7ONLINE (STEP 7) in the Access*



**Fig. 13.1** Setting IP addresses with the Windows Control Panel

*point of application* box. Select the LAN or WLAN interface module used under *Interface Parameter Assignment Used* and close the tool.

### Determining IP addresses with STEP 7 Basic

If you wish to find out the IP address of the programming device (to be more precise: the IP address of the interface module used) or, for example, wish to delete the temporary IP address, proceed as follows:

Connect the interface module of the programming device to Ethernet, for example to a CPU. Switch on the CPU to activate the interface. The CPU can be in any operating mode.

Start STEP 7 and change to the Project view. The virtual and physical online interfaces of the programming device are listed in the project tree under *Online access*. Select the interface used and then the *Properties* command from the shortcut menu. The MAC address, the fixed IP address, and the subnet mask are displayed in the properties dialog in the *Configuration* section under *Industrial Ethernet*. All project-specific IP addresses are listed under *IE-PG access*. You can delete all of these addresses using the *Delete project-specific IP addresses* button.

#### 13.1.2 Connecting the programming device to the PLC station

Connect the LAN connection of the programming device to the PROFINET interface of the CPU module. You can use a standard or crossover cable since the CPU module is suitable for both types. Make sure that no memory card is inserted in the CPU module, and then switch on the power supply to the CPU module.

Following the restart, the CPU module – if it has been obtained directly from the factory or has been reset to the factory settings – is at STOP, and the RUN/STOP LED lights up yellow.

### Searching for accessible devices

Start STEP 7 Basic, select the *Online & Diagnostics* portal in the portal view, and then *Accessible devices*. If the programming device has several interfaces, select the interface (module) to which the CPU module is connected in the *Accessible devices* window.

A station which has been found is listed in the table with its IP address or – if it does not have an IP address – with its MAC address. At the same time, the graphic is provided with an orange background (Fig. 13.2).

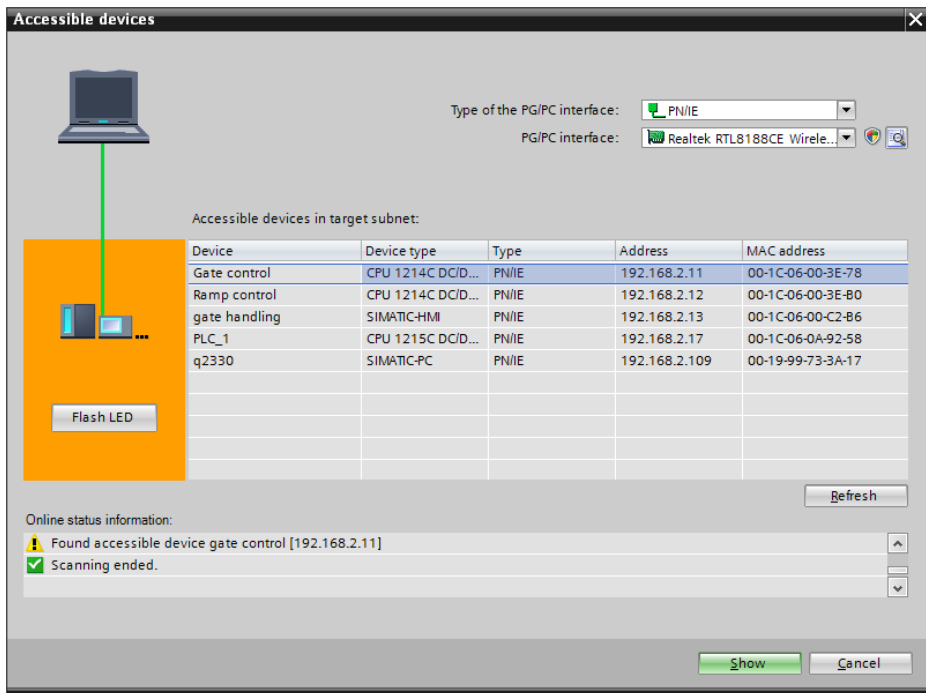


Fig. 13.2 Dialog window *Accessible devices*

Select the line with the station. You can then click the *Flash LED* button in order to briefly flash the status LED on the front panel of the CPU. To process the selected station further in the project view, click on the *Show* button.



## Setting a temporary IP address on the programming device

If the network settings of the programming device do not agree with those of the CPU module, STEP 7 Basic allows you to set an appropriate, project-specific IP address on the programming device. This IP address is only present temporarily until you switch off the programming device or delete the address. Answer the corresponding dialogs with *Yes* or *OK* in order to assign an IP address. The assigned IP address is displayed in the response dialog.

STEP 7 Basic now shows the found CPU in the project view, and this is positioned with its IP or MAC address in the *Online access* group under the used interface module as a new group in the project tree.

### 13.1.3 Assigning an IP address to the CPU module

If a CPU module is displayed with its MAC address, you can assign an IP address to it: select the PLC station and then the *Online & Diagnostics* command from the shortcut menu. Select the *Assign IP address* entry in the diagnostics window in the *Functions* section. Enter the desired IP address and subnet mask, and click on the *Assign IP address* button. The result of the action is signaled in the Inspector window in the *Info* tab.

### 13.1.4 Switching on the online mode

If you select the PLC station present under *Online access* in the project tree and then *Online & Diagnostics* from the shortcut menu, the online tools are displayed with the CPU operator panel and the diagnostics window. Further details can be found in Section 13.3 “Hardware diagnostics” on page 436.

If the project matching the online PLC station is present, open it and select the PLC station in the project tree. Select *Go online* from the shortcut menu, or activate the *Go online* symbol in the main menu. The status of the PLC station is displayed in the Inspector window in the *Diagnostics* tab. Diagnostics symbols in the project tree signal agreement between the online and offline versions of the blocks and PLC tags (see Section 13.3.6 “Further diagnostics information via the programming device” on page 440).

Alternatively you can switch to online mode by selecting the *Online & Diagnostics* command in the shortcut menu with the PLC station selected. The diagnostics window is then opened. Now click on the *Go online* button in the *Online access* section.

### Further procedure

- ▷ How you can use the diagnostics and online tools, for example in order to start and stop the CPU or to reset to the default settings, can be found in Section 13.3 “Hardware diagnostics” on page 436.
- ▷ The following Chapter 13.2 “Transferring project data” describes how you can upload a user program to the PLC station and edit the user program online.

- ▷ Chapter 13.4 “Testing the user program” on page 441 describes how you can test a user program.
- ▷ Chapter 13.2.7 “Editing online project without offline project” on page 433 describes how you can access the online project data of the CPU without the user program.

## 13.2 Transferring project data

You have configured the hardware and completed and compiled the user program. Downloading to the PLC station can be carried out in the following ways:

- ▷ Transfer via an online connection
- ▷ Transfer using a memory card as a transfer card
- ▷ Transfer using a memory card as a program card

The project data can only be downloaded when the CPU is in the STOP mode.

### 13.2.1 Loading project data for the first time

A CPU module without project data – for example if it has been obtained directly from the factory – is in the STOP mode following connection of the power supply. In order to download the project data, connect the programming device to the CPU module, switch it on, and open the project.

Select the PLC station in the project tree and then the *Download to the device > All* command from the shortcut menu. When loading for the first time, the dialog window *Advanced loading* shows the IP address of the PLC station in the *Configured access nodes of ...* table. If the programming device has several interface modules, select the interface at which the PLC station is connected from the *PG/PC interface* drop-down list.

#### **The PLC station does not have an IP address, or a different one**

If the configured IP address does not agree with the IP address set in the CPU, STEP 7 cannot find the device matching the configuration. This is indicated by an error message in the *Extended download to device* dialog window. In this window, activate the *Show all accessible devices* check box. Browsing is then started again.

The found devices with their IP address are displayed in the *Accessible devices in target subnet* table. The MAC address is displayed if a device does not have an IP address. Select the desired PLC station in this table, and click on the *Load* button.

If the network settings of the programming device do not match the configured IP address, the dialog window *Assign IP address* is displayed. STEP 7 then inquires whether the programming device is to be assigned a further temporary, project-specific IP address. Acknowledge the inquiry with *Yes* or with *OK*.

## The project data is compiled prior to downloading

The project data is compiled again prior to downloading. Only consistent project data which has been compiled without errors can be downloaded. The compilation process can be observed in the *Load preview* dialog window.

Following error-free compilation, the *Check before loading* message is displayed in the *Load preview* dialog window. Via the dropdown menu, select the desired action from the *Action* column and/or activate the proposed actions using the checkbox. You can continue with loading by clicking on the *Load* button (Fig. 13.3).

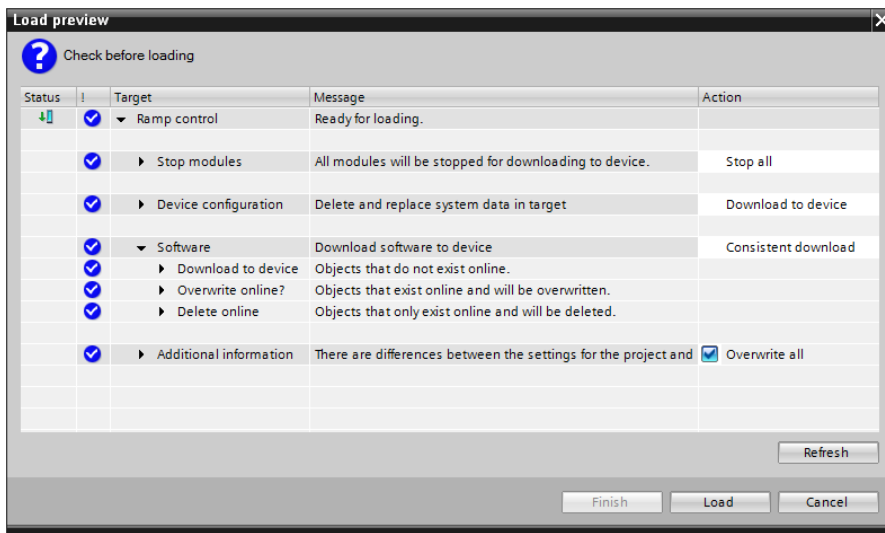


Fig. 13.3 *Load preview* dialog window

## Starting the CPU following downloading

The results of loading are displayed in the dialog window *Load results*. Following loading without errors, you can start the CPU with the new user program.

*Caution: Make sure when starting the CPU on the controlled machine – with a program which could possibly be faulty – cannot cause any damage to persons or property and that no dangerous states can result!*

If the CPU was in RUN mode prior to the downloading process, the *Start all* check box in the *Action* column is activated. If it was in STOP mode, activate the check box in order to start the CPU. Click on the *Finish* button.

With the *Start all* check box activated, the CPU is started when the downloading process has been completed. If no errors occur, the CPU is then in the RUN mode. The RUN/STOP LED lights up green.

## Switching on and testing online operation

Following loading of the project data, you can activate online mode by means of the *Go online* symbol in the main menu. The title bar of the active window has an orange background. You can open the diagnostics window using the *Online & Diagnostics* command from the shortcut menu of the selected PLC station. You can then use the online tools, for example the CPU operator panel.

If you open a block in the project tree, you can use the *Monitoring on/off* symbol in the toolbar of the working window to switch on the program status and to debug the program. The possibilities offered by STEP 7 for program debugging are described in Section 13.4 “Testing the user program” on page 441.

### 13.2.2 Delta downloading of project data

When reloading project data, only the changes compared to the online project data are loaded. It is possible to specify for the software (in the user program) whether only the changes are loaded or everything.

You determine with the download command which project data is to be downloaded. Select the object to be downloaded in the project tree and then the *Download to device > ...* command from the shortcut menu. You can:

- ▷ Select the *All, Hardware configuration, Software* or *Software (all blocks)* commands for the selected PLC station
- ▷ Select the *Software* or *Software (all blocks)* commands for the selected *Program blocks* folder
- ▷ Select the *Software* command for one or more selected blocks

The result of loading is shown in the inspector window under *Info > General*.

### The CPU is in RUN mode

The configuration data can only be download in the STOP mode. If you select *Download to device > ...* from the shortcut menu, you will be asked whether the selected PLC station is to be set to STOP. Loading is canceled using *Cancel*, the CPU is set to STOP using *OK*, and the loading process is continued with the compilation of the project data.

### The project data is partially up-to-date

If only the configuration data has been modified, for example, the *Load preview* dialog window shows following compilation of the project data that the unmodified project data is not loaded, e.g. *The software is not downloaded because the online status is up-to-date*. Select the desired actions and continue loading by clicking on the *Load* button.

## Starting the CPU

If the CPU was in RUN mode prior to the downloading process, the *Load results* dialog window then inquires whether the CPU is to be started (*Start modules after downloading to device* message with *Start all* check box activated).

*Caution: Make sure when starting the CPU on the controlled machine – with a program which could possibly be faulty – cannot cause any damage to persons or property and that no dangerous states can result!*

Continue by clicking on the *Finish* button. The downloading process has been completed when the RUN/STOP LED lights up permanently following short flashing.

## Downloading a non-consistent program following faulty compilation

If an error is detected when compiling prior to downloading, this is signaled in the *Load preview* dialog window. The component at which the error has occurred is indicated in the *Target* column under *Compile* (click the triangle on the left of this). It is only possible to continue the downloading process when the error has been eliminated.

### 13.2.3 Error message following downloading

If the CPU does not start following loading – the RUN/STOP LED remains yellow – or if the ERROR LED flashes, the diagnostics buffer can provide information on the cause. Remaining in the STOP mode or returning to it could be the result of, for example, a faulty I/O access in the user program. If the CPU is in RUN and the ERROR LED is flashing, there may be a difference between configured and actual hardware, for example.

Section 13.3.3 “Diagnostics buffer” on page 437 describes how the diagnostics buffer can support you during troubleshooting.

### 13.2.4 Working with the memory card

A SIMATIC Memory Card for a CPU 1200 is an SD memory card (secure digital memory card) preformatted by Siemens. If you wish to delete the contents of the memory card, you must only delete files or folders. Formatting the memory card makes it unusable in a CPU 1200. Please make sure that the write protection – the small slide switch on the side of the card – is switched off if it is used in the CPU.

If you insert or remove a memory card when the CPU is in the RUN mode, the mode immediately changes to STOP. *Caution: Make sure that stopping of the CPU cannot cause any damage to persons or property and that no dangerous states can result!*

## Setting the type of card

You can use the memory card as a transfer card or as a program card. You can use the transfer card as a replacement for the online connection to download a project to the CPU; the program card replaces the internal load memory.

To set the type of card, insert the memory card into the programming device's card reader. In the project tree, open the *SIMATIC Card Reader* folder and the subordinate folders down to the SD card (to the drive letter). Select the SD card and click on the *Properties* command in the shortcut menu. In the dialog window that is then displayed, select the *Program* or *Transfer* entry from the drop-down list in the *Card type* field.

### **Transferring project data to the memory card**

Once the memory card has been set as a transfer or program card, copy the project data of the PLC station onto the memory card, e.g. using *Copy* with the PLC station selected and *Insert* from the shortcut menu with the SD subsequently selected, or by dragging the PLC station to the memory card with the mouse button pressed. The project will be compiled. Following error-free compilation, the *Load preview* window is displayed; continue the downloading process with the *Continue* check box activated by clicking on the *Load* button. Click the *Complete* button to terminate the downloading process.

### **Using the memory card as a transfer card**

Insert the memory card into the CPU module. If the CPU was in the RUN mode, it changes to STOP. The MAINT LED flashes as an indication that the memory card has to be evaluated. The memory card is evaluated following a CPU restart, e.g. when switching the power supply off and on again.

Following the restart, the project data is transferred from the memory card to the internal load memory. The flashing MAINT LED signals the end of the transfer. You must remove the memory card before starting the CPU again.

You can start the CPU by switching the power supply off and on again, for example. The CPU starts with the set operating mode.

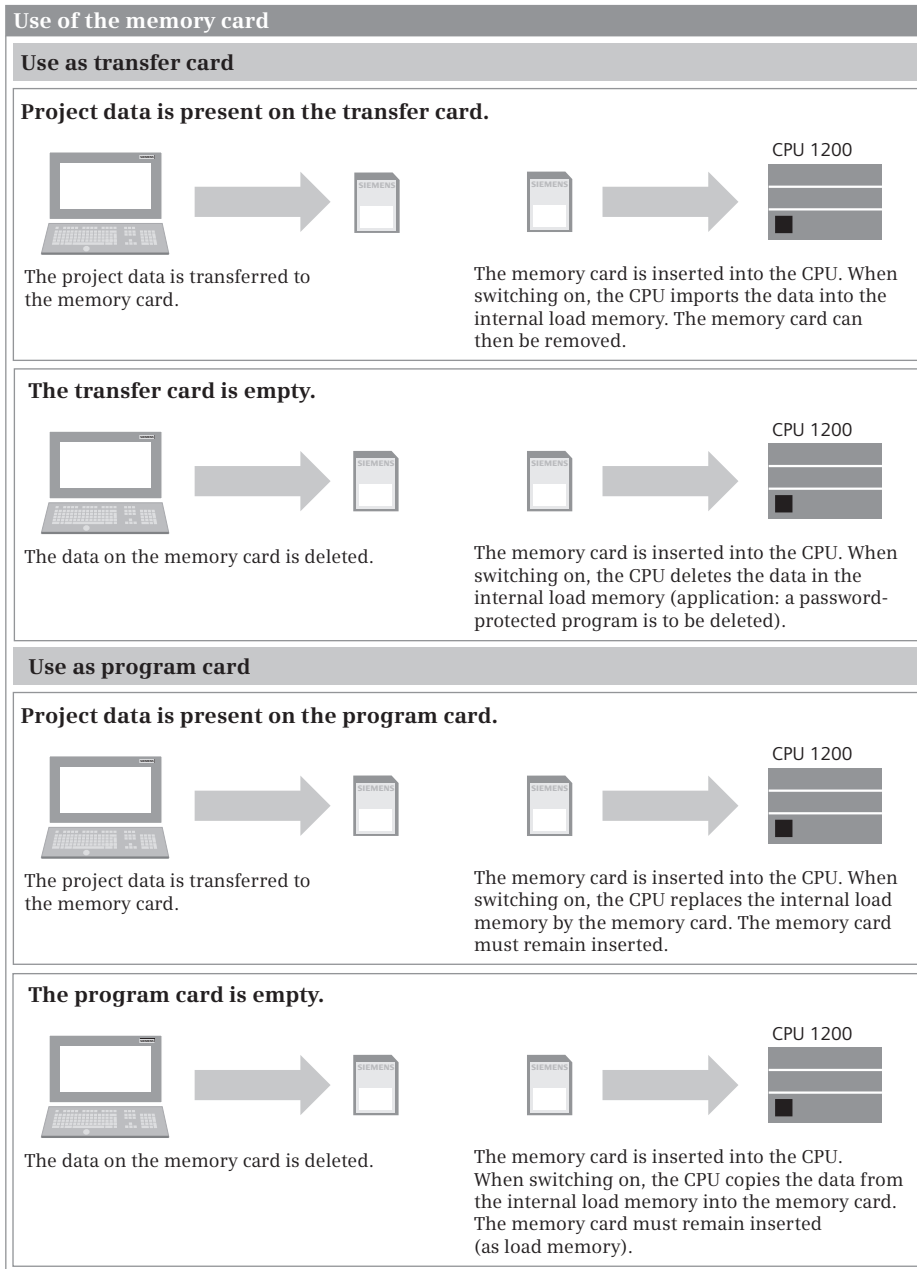
If the memory card is an empty transfer card, the internal load memory of the CPU is deleted during the transfer. A password-protected user program can be deleted in this manner if the password is unknown.

### **Using the memory card as a program card**

Insert the memory card into the CPU module. If the CPU was in the RUN mode, it changes to STOP. The MAINT LED flashes as an indication that the memory card has to be evaluated. The memory card is evaluated following a CPU restart, e.g. when switching the power supply off and on again.

The contents of the internal load memory are deleted following the CPU restart, and the executable data transferred from the memory card to the work memory. The CPU has the set operating mode following the restart.

The memory card must remain in the CPU module since it now contains the load memory. If the memory card is removed during runtime, the CPU goes to STOP and the ERROR LED flashes.



**Fig. 13.4** Use of the memory card as transfer and program card

If the memory card is an empty program card, the contents of the internal load memory of the CPU are transferred to the memory card following a CPU restart. Following a further CPU restart, the memory card is then used as an external load memory.

### 13.2.5 Processing blocks offline/online

Prerequisites: you have transferred the project data from the programming device to the CPU, the project is open, and the CPU connected online.

The program editor displays the user interface either in offline or online mode. In offline mode, the project data in the programming device is displayed; in online mode, the data in the CPU. You can swap between the two modes using the *Go online* and *Go offline* symbols in the toolbar of the project view. How to switch to online mode for the first time is described in Section 13.1 “Connecting a programming device to the PLC station” on page 421.

#### Changing the CPU's configuration data

It is only possible to change the configuration data in offline mode. Switch to offline mode, change the configuration data offline, and transfer it to the CPU. If you wish to transfer only the configuration data, select the PLC station in the project tree and then the *Download to device > Hardware configuration* command from the shortcut menu. The CPU is switched to STOP during downloading.

#### Changing a block in the CPU

It is only possible to change program blocks in offline mode. If you wish to change the program of a block in the CPU during program debugging, you must switch to offline mode, change the block in the offline data management, and then transfer it to the CPU. The switch to online mode again and continue with debugging.

Online mode is switched on; the title bar of the active window has an orange background, the title bars of the inactive windows are underlined in orange. Blocks which are the same offline and online are identified by a green filled circle in the project tree. A block is open, and you wish to change this block's program.

If you wish to carry out the changes on the online version of the block, the color of the title bar of the working window is changed, i.e. the program editor has automatically switched to the offline block which you can then change. A circle divided in two colors in the project tree indicates that there is a difference between the offline and online versions of the block.

Following completion of the change, transfer the modified block to the CPU: select the block and then the *Download to device > Software* command from the shortcut menu. Follow the displayed dialog windows. The CPU must be set to STOP during the downloading process, and subsequently restarted.

#### Adding a block online

You can generate a new block in the offline data management using the *Add new block* tool in the project tree, even if online mode is switched on. Program the block offline, and program the block call as well – if the new block is not an organization block – and then transfer both blocks to the CPU.



## Deleting a block online

You delete a block by selecting it and then the *Delete* command from the shortcut menu. If the online mode is switched on and you wish to delete a block which is present both online and offline, the program editor inquires which of the blocks is to be deleted: the online block (*Delete from device*), the offline block (*Delete from project*), or both.

Before deleting a block in the CPU, you should delete its call, i.e. remove the call of the block to be deleted from the calling online block, otherwise the error *The called block does not exist* is signaled during runtime.

### 13.2.6 Comparing blocks offline/online

The compare editor enables you to compare the offline and online versions of blocks. The comparison is possible in the STOP and RUN modes.

To start the compare editor, select the object to be compared in the project tree, e.g. the PLC station, the *Program blocks* folder, or individual blocks. Select the *Compare > Offline/online* command from the shortcut menu, or the command *Tools > Compare > Offline/online* in the main menu.

The comparison editor displays the compared objects and the comparison status in the working window. As standard, objects which have different offline and online versions are displayed (Fig. 13.5). The filter icon in the toolbar can be used to switch to a display of all objects. In addition, the compared objects are also identified by a comparison symbol in the project tree.

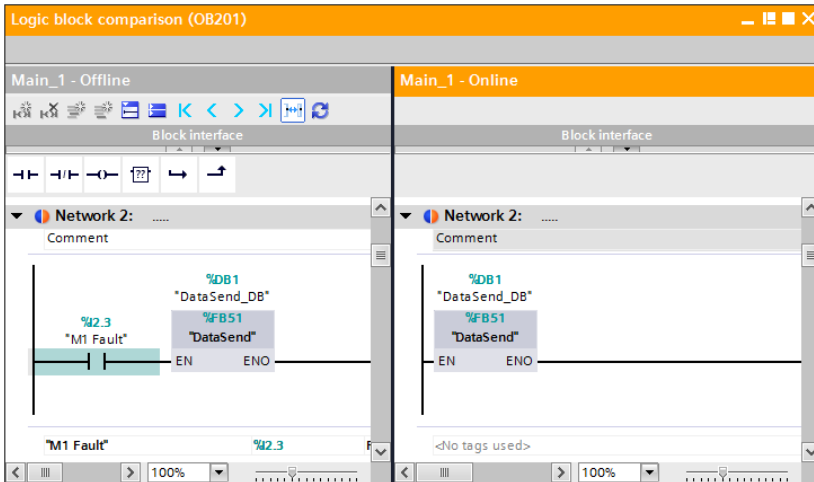
A green filled circle indicates that the offline and online versions are identical. A blue/orange circle indicates that the object is different in the offline and online versions. If one half circle is not filled, the corresponding version is missing (left side or blue represents offline, right side or orange represents online). An exclamation mark in an orange circle indicates an object with differences in the named folder.

Reference program	Status	Action	Compare to	Details
Gate control	!	Different actions	Gate control	
Program blocks	!	Different actions		
Main [OB1]	!	No action	Main [OB1]	Code is different.
Main_1 [OB201]	!	Download to device	Main_1 [OB201]	Code is different.
RecipeData [DB71]	!	Download to device	RecipeData [DB71]	Data is different.
Communication	!	No action		
Representation	!	No action		
Test counter	!	No action		
System blocks	!	No action		
Technology objects	●			
PLC tags	●			
PLC data types	●			

**Fig. 13.5** Example of an offline/online comparison with different objects

In the *Action* column, you can select the desired action from a dropdown menu, such as *Download to device*, if the offline version differs from the online version. Clicking on the *Execute actions* icon in the toolbar starts the set actions.

You can start a detailed comparison - providing the object allows it - if you select an object with different versions and select *Start detailed comparison*, either in the shortcut menu or via the icon. In a successful detailed comparison, both versions are opened at the point with the first difference and the detected difference is visible in the inspection window in the *Comparison result* tab (Fig. 13.6).



**Fig. 13.6** Example of a detailed comparison

Following elimination of the differences and execution of the appropriate downloading action, you can start the comparison editor again using the *Refresh the view* symbol.

Note that only one offline/offline comparison or one offline/online comparison can be carried out at a time.

### 13.2.7 Editing online project without offline project

You can access the project data in a CPU using the programming device even without offline project data.

Connect the programming device to the CPU, switch on the module and select the portal *Online & Diagnostics* and then *Accessible nodes* in the portal view. Set the PG/PC interface module if applicable. In the *Accessible devices in target subnet list*, select the PLC station and click on the *Show* button. If the programming device does not possess the matching network parameters, a dialog window is displayed to allow you to set these temporarily. Acknowledge with *YES* or *OK*.

In the project view, the PLC station is displayed in the project tree of the interface module used in the *Online accesses* folder. Select the PLC station and then the *Online & diagnostics* editor from the shortcut menu. In online mode, you can select the mode using the CPU operator panel, for example, or read out the diagnostics buffer in the diagnostic functions.

The online blocks are present in the *Program blocks* folder. When you open the folder, STEP 7 downloads the blocks into it. A block is opened by double-clicking, and the program in the block is displayed.

If you wish to edit or debug an online block, you must create an offline project and transfer the online blocks to the project (see next section). Only blocks which exist offline can be newly created, modified or debugged.

### 13.2.8 Uploading project data from the CPU

In order to upload online project data, an offline project must be present in the programming device. If the offline project matching the online project is not available, an “empty” offline project must be created into which the online data can be loaded.

To create an “empty” project: select the *Project > New* command in the main menu, and assign a name to the project. Double-click in the project tree on *Add new device* and select the *non-specified CPU 1200 6ES7 2XX-XXXX-XXXX* from the *SIMATIC PLC* catalog in the *Add new device* dialog window. Enter a name for the “empty” PLC station.

If you have not already done so, connect the programming device to the CPU module whose project data you wish to upload, and switch the CPU module on.

#### Hardware detection

When the unspecified CPU is inserted, the hardware configuration provides a link for determining the configuration of the connected device. If this is not already done, connect the programming device to the CPU that has project data you want to upload and switch on the CPU.

Click on the link for determining the configuration or select the “empty” PLC station in the project tree and select the command *Online > Hardware detection* from the main menu. The devices that are found are listed in the *Hardware detection* dialog. Select the PLC station in the table and click on the *Detect* button. If the programming device needs a project-specific IP address, STEP 7 allows you to set up a corresponding address. Acknowledge the inquiry with *Yes* or *OK*. The “empty” PLC station is now filled with configuration data, which correspond to data of the connected CPU.

Now copy the online blocks into the offline project. To do this, open the desired PLC station in the project tree under *Online accesses* and the relevant interface module, select the folder *Program blocks*, and select *Copy* from the shortcut menu. In the project tree, go to the offline station, select the *Program blocks* folder, and then select the *Paste* command from the shortcut menu.

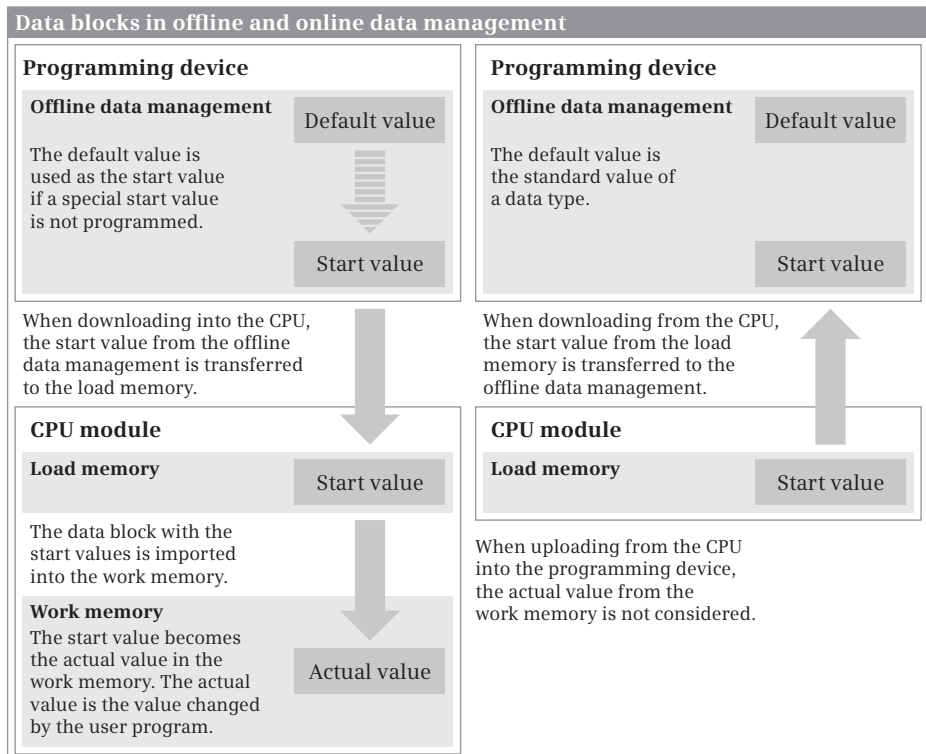
In the window *Preview for loading from device* you are notified which offline objects are replaced by the online objects, e.g. including the PLC tag table. Check the **Continue** checkbox and click on the *Load from device* button.

### What is uploaded?

During the hardware detection, there can be differences in the parameterization of the modules between the offline and online configuration, for example in the parameterization of the system memory and cycle memory. If you want the same configuration online and offline, you must adapt the uploaded configuration data and then upload it to the CPU.

In addition to the configuration data, the load memory of the CPU also contains the online PLC tag table and the compiled user program. The PLC tag table is newly created during the upload; it also contains the names of the blocks. The names of the block-local tags are present in the block's program.

The values of the data tags are the start values which are fetched from the load memory when uploading. The actual values from the work memory are not considered (Fig. 13.7).



**Fig. 13.7** Data tags when loading to and from the PLC station

You have access to the actual values in the work memory

- ▷ by directly monitoring the actual values (Chapter 13.4.5 “Monitoring of data tags” on page 446),
- ▷ with a watch table (13.4.6 “Testing with watch tables” on page 447) and
- ▷ via a program with WRIT\_DBL (copying a data block with the actual values into the load memory, where the actual values then become start values).

## 13.3 Hardware diagnostics

The hardware diagnostics detects and signals module faults, e.g. failure of load voltage or open-circuit with signal modules.

The modules with diagnostics capability distinguish between parameterizable and non-parameterizable diagnostic events. With the parameterizable diagnostics events, a message is only output if you enable the diagnostics in the parameter settings. The non-parameterizable diagnostic events are always signaled irrespective of the diagnostics enable.

With a diagnostics event to be signaled,

- ▷ the ERROR LED lights up on the CPU module
- ▷ the diagnostics event is sent to the CPU's operating system, and
- ▷ a diagnostics interrupt is triggered if you have enabled it in the parameter settings (in the default setting, the diagnostics interrupts are disabled).

All diagnostic events signaled to the CPU's operating system are entered in a diagnostics buffer in the order in which they occurred with date and time (see Chapter 13.3.3 “Diagnostics buffer” on page 437). In addition to the diagnostics buffer, which saves the events in chronological order, the programming device offers comprehensive information functions which display the current module states.

### 13.3.1 Status displays on the modules

The status displays on the modules signal malfunctions and help to locate the faults.

The CPU's operating system indicates malfunctions as follows:

- ▷ The ERROR LED lights up permanently.  
The CPU hardware is faulty.
- ▷ The ERROR LED flashes.  
Possible causes are an internal fault in the CPU or a configuration error (the configured hardware configuration does not agree with the actual hardware configuration).
- ▷ The RUN/STOP LED lights up yellow.  
The CPU does not enter RUN mode when switched on or goes to the STOP mode during RUN mode. Possible causes: Manual change in mode through the programming device, set startup type (“Startup – STOP”), STP function

in user program, system response to an execution error in the program. The events triggering the STOP mode are entered into the diagnostics buffer (see Chapter 13.3.3 “Diagnostics buffer” on page 437).

Each input/output channel of a digital module shows by means of a green status LED whether voltage is present at the input or output channel. This can be used, for example, to check the wiring from the sensor to the digital input channel or from the digital output channel to the actuator.

An appropriately designed input/output channel of an analog module indicates by means of a green status LED that the channel has been configured and is active. A red LED indicates a faulty analog channel or one which is not ready. For example, a wire-break or short-circuit could be present. If the LED flashes red, the load voltage is missing or an I/O error is present if the diagnostics is activated.

In addition, the modules have a DIAG LED. When permanently green, this indicates that the module is ready. If the DIAG LED flashes green, the module is not configured. If the LED flashes red, a module fault exists, e.g. the load voltage is missing.

### 13.3.2 Diagnostics information

The diagnostics information is displayed in the work window if the programming device is switched to online mode using the *Online & Diagnostics* command. The following diagnostics information is then available:

- ▷ General: module names, module and vendor information.
- ▷ Diagnostic status: status information of the selected module, e.g. *Module exists and OK*, differences between configured and existing modules.
- ▷ Cycle time: Display of preset or configured cycle (monitoring) time and minimum cycle time and – in RUN mode – the cycle time diagram and the shortest, current, and longest cycle (processing) times.
- ▷ Memory: display of utilization for the load, work and retentive memories.
- ▷ Diagnostics buffer: display of diagnostics buffer content.

If the CPU is still without a user program, it is in STOP mode. The cycle time and memory utilization then indicate zero values.

### 13.3.3 Diagnostics buffer

The diagnostics buffer contains the faults detected by the CPU and the modules with diagnostic capability, the triggered hardware and diagnostic interrupts, and the changes in CPU modes in the sequence of occurrence. Up to 50 entries can be stored. If more entries are present, the oldest entries are overwritten (ring buffer principle). If the operating voltage is switched off and on again, the last 10 entries (the most recent ones) are retained. The entries can only be erased by resetting the CPU to its factory settings (Fig. 13.8).

Diagnostics buffer

**Events**

Display CPU Time Stamps in PG/PC local time

No.	Date and time	Event		
1	9/26/2012 1:08:08.850 ...	Follow-on operating mode change - CPU changes from STARTUP to RUN mode	✓	i
2	9/26/2012 1:08:08.840 ...	Communication initiated request: WARM RESTART - CPU changes from STOP to ...	✓	i
3	9/26/2012 1:08:08.840 ...	New startup information - Current CPU operating mode: STOP	✓	i
4	9/26/2012 1:08:06.240 ...	New startup information - Current CPU operating mode: STOP	✓	i
5	9/26/2012 1:07:57.335 ...	New startup information - Current CPU operating mode: STOP	✓	i
6	9/26/2012 1:07:57.135 ...	New startup information - Current CPU operating mode: STOP	✓	i
7	9/26/2012 1:07:57.033 ...	Communication initiated request: STOP - CPU changes from RUN to STOP mode	✓	i
8	9/26/2012 1:06:58.167 ...	Follow-on operating mode change - CPU changes from STARTUP to RUN mode	✓	i
9	9/26/2012 1:06:58.158 ...	Communication initiated request: WARM RESTART - CPU changes from STOP to ...	✓	i

Freeze display

**Details on event:**

Details on event:  of  Event ID:

Description: CPU info: Communication initiated request: WARM RESTART  
Pending startup inhibit(s):  
- No startup inhibit set

CPU changes from STOP to STARTUP mode

Time stamp:

Module:

Rack/slot:

Plant designation:

Location identifier:

Priority:

Incoming/outgoing:

Help on event Open in editor Save as...

**Fig. 13.8** Diagnostics buffer

The most recent event is positioned in the first line in the diagnostics buffer. A diagnostics entry consists of the time stamp (date and time at which the event was detected) and the event text. The time stamp is only meaningful if the time of the CPU module is correct. Each event is exactly specified by an event ID. When you select a line, the event ID is displayed on the right below the table.

Using the *Help on event* button, you can obtain additional information on the selected event. If the entry refers to a block, e.g. with an access error to the I/O, it is possible to switch to the position of the fault in the user program by using the *Open in editor* button.

The *Freeze display* button stops display of the entries; you can then call information for a specific event, or study the sequence of displayed events without haste. Clicking on the button again (label now: *Cancel freeze*) changes to the updated display. Using the *Save as ...* button you can save the contents of the diagnostics buffer as a text file.

### 13.3.4 Diagnostics functions

You can start the diagnostics functions in online mode using the *Online & Diagnostics* command from the project tree.

- ▷ Assign IP address: Setting of the IP address, subnet mask, router address.
- ▷ Set time of day: Display of programming device and module time, setting of real-time clock on CPU.
- ▷ Reset to factory settings: The user memory, operand areas, and diagnostics buffer are deleted, all parameters including the time are reset to the default settings, and the IP address is also deleted or retained depending on the pre-selection.
- ▷ Assign name: Setting a device name for a PROFINET IO device.

### 13.3.5 Online tools

You can start the task card with the online tools using the *Online & Diagnostics* command from the project tree:

#### **CPU operator panel:**

The CPU operator panel shows the current status of the LEDs on the front panel of the CPU. The RUN and STOP buttons can be used to set the CPU – following confirmation – to the corresponding state. A pressed (darker) button symbolizes the currently set state. A CPU without a user program, e.g. after being reset to the factory settings, does not start up.

#### **Memory reset:**

The *MRES* button is used to trigger a memory reset. A memory reset can only be carried out in the STOP mode. During the memory reset, the contents of the work memory, retentive memory, and all operand areas are deleted. The contents of the load memory are retained, even if present on a memory card. The contents of the load memory relevant to execution are copied into the work memory, just like when transferring the user program to the CPU. The diagnostics buffer, time, force jobs, and IP address remain uninfluenced.

The existing (logical) connections to the CPU module are canceled. Following a CPU memory reset, the programming device must be switched to online mode again using the *Online & Diagnostics* or *Go online* command.

#### **Cycle time:**

*Cycle time* displays the shortest, current and longest cycle (processing) time in milliseconds, also graphically.

#### **Memory:**

*Memory* displays the utilization of the load, work and retentive memories as bar charts.



### 13.3.6 Further diagnostics information via the programming device

#### Diagnostics symbols in the device and network views

In online mode, the device configuration editor shows, in the device or network view, the device status with diagnostics symbols on each PLC station connected online. For example, a green tick indicates that the station has not signaled any faults. The operating mode is indicated by a colored square: green for RUN and yellow for STOP.

#### Diagnostic symbols in the project tree

In online mode, diagnostics symbols are also displayed in the project tree. A green tick is displayed after the name of the PLC station if everything is OK. A spanner signals maintenance required (green), maintenance request (yellow), or fault (red).

The result of a comparison between offline and online project data is also displayed in the project tree. If an orange circle with an exclamation mark is displayed, the folder contains objects which differ between the online and offline versions. Identifications for individual objects have the following meaning:

- ▷ Green filled circle: everything OK
- ▷ Blue/orange filled circle: indicates that the online and offline versions of the object are different
- ▷ Blue/orange filled circle, right half (orange) filled: only the online object is present
- ▷ Blue/orange filled circle, left half (blue) filled: only the offline object is present

#### Device information in inspector window

The status of the devices signaled as faulty is displayed in the inspector window in the *Diagnostics > Device information* tab. A device is considered to be faulty if it is inaccessible when establishing the online connection, if it signals a fault or if it is not in RUN mode. (Fig. 13.9). Via the link in the *Details* column you can access the *Go online* dialog or the online and diagnostics view of the faulty device.

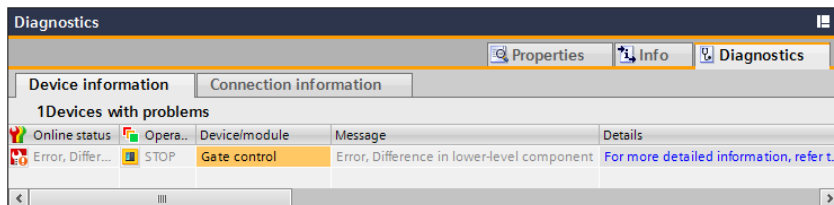


Fig. 13.9 Diagnostics tab in the inspector window

## 13.4 Testing the user program

Following the establishment of a connection to a CPU and loading of the user program, you can test the entire program or parts of it, such as individual blocks. You supply the tags with signals and values and evaluate the information returned by the program. If the CPU switches to STOP as the result of a fault, the diagnostics buffer provides support toward locating the cause.

Comprehensive programs are tested in sections. If you only wish to test one block, for example, load the block into the CPU and then call it in OB 1. If OB 1 is structured such that the program can be tested in sections “from front to rear”, you can select the blocks or program sections to be tested in that you bypass the calls or program sections which are not to be processed, e.g. using a jump function.

The following testing functions are available:

- ▷ Test in program status  
Monitor program execution directly in the program of the block and control tags
- ▷ Monitor PLC tags  
Monitor the values in a PLC tag table
- ▷ Monitor data tags  
Monitor the tag values in a data block
- ▷ Test with watch tables  
Monitor and control the tag values in watch tables
- ▷ “Enable peripheral outputs” and “Modify now”  
Control peripheral outputs with CPU at STOP
- ▷ Test with force table  
Monitor the tags in the force table and set to a fixed value (force).

A general prerequisite for testing the user program is an existing online connection. When testing with program status, the offline and online versions of the block must be identical. The CPU is in RUN mode.

### 13.4.1 Introduction to testing with program status

The program status shows the program execution during runtime. You can monitor the current signal status of the binary tags and the current values of digital tags.

*Caution! Functional disturbances may occur as a result of program modifications when testing the user program during ongoing operation on the process. Make sure with each testing step that no serious damage to property or injury to persons can occur!*

Note that the program status requires significant resources, and that under certain circumstances the test function may therefore be executed with limitations.

Changing a block with the program status switched on, i.e. with an online connection present, is described in Section 13.2.5 “Processing blocks offline/online” on page 431.

### Switching the program status on and off

In order to switch on the program status, open the block to be monitored and click on the *Monitoring on/off* symbol in the toolbar of the work window.

If an online connection to the CPU has not yet been established, STEP 7 searches for accessible devices. If necessary, set the interface used in the programming device in the dialog window *Go online*, select the PLC station found, and click on the *Go online* button.

To switch off the program status, click again on the *Monitoring on/off* symbol in the toolbar. You will be asked whether the online connection created when switching on the program status is to be disconnected. If you click on the *No* button, the program status is terminated but the online connection is retained.

#### 13.4.2 Program status with LAD and FBD

The program status shows the binary signal flow and the values of digital tags directly in the LAD or FBD program.

Open the block, e.g. by double-clicking in the project tree, set the network whose program you wish to debug, and click on the *Monitoring on/off* symbol in the toolbar of the work window.

#### LAD program status

In the LAD program status, green continuous lines are used to identify contacts, coils, and the connections between the program elements which have signal state “1”. Program elements with signal state “0” are identified by blue dashed lines (Fig. 13.10).

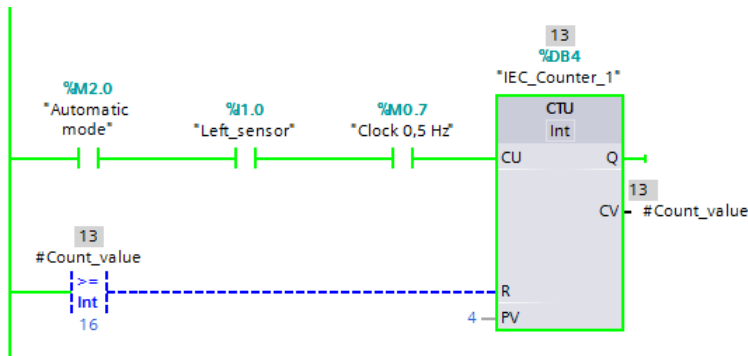


Fig. 13.10 LAD program status

### FBD program status

In the FBD program status, the boxes of the binary program elements and the connections are displayed by continuous green lines if they have signal state “1”, and by dashed blue lines if they have signal state “0” (Fig. 13.11). In addition to the colored identification, the signal state (“0” or “1”) is shown for the binary inputs.

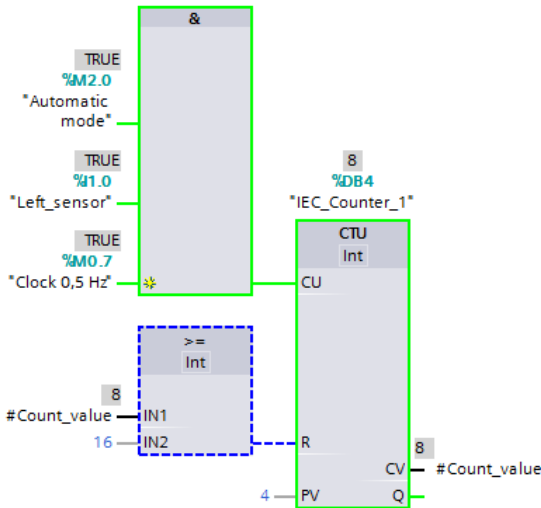


Fig. 13.11 FBD program status

### Display format with digital tags

You can select the display format of digital tags, which is set as standard to *Automatic*: You select the digital tag and select *Modify > Display format > ...* from the shortcut menu. ... > *Decimal*, ... > *Hexadecimal*, and ... > *Floating-point* are available, depending on the data type.

You set the display format for the complete network by clicking with the right mouse button on a free space in the network and selecting *Modify > Display format for network > ...* from the shortcut menu.

### Controlling operands in the program status

In the program status you can use the programming device to define the signal states of binary tags and the values of digital tags. This is usually only meaningful if these tags cannot be controlled from another position, for example inputs which receive their signal state from the peripheral input channel during the automatic updating of the process image.

Select the binary tag and then the *Modify > Modify to 0* command from the shortcut menu if the binary tag is to be set to signal state “0” or *Modify > Modify to 1* if

the binary tag is to be set to signal state “1”. In the case of digital tags, select the *Modify > Modify operand...* command from the shortcut menu and specify the desired value.

### Selective monitoring and up-to-dateness of the tag values

You can determine at which position the program status is to be executed: You select the program element or tag and then the *Modify > Monitor from here* command from the shortcut menu. The *Modify > Monitor selection* command means that only the selected program element is monitored.

Program elements with unknown status or those which are not processed are identified by continuous gray lines. Black tag values come from the current monitoring cycle, grey tag values come from a cycle that was processed earlier.

### Block calls in the monitored block

If the tested network contains a block call, the call box is represented by green continuous lines if the EN input is “1”. The box has blue dashed lines if the EN input is “0”.

You can continue the program status in the called block: You select the block call and then the *Open and monitor* command from the shortcut menu. The program status then changes to the called block.

#### 13.4.3 Program status in SCL

The program status is shown in tabular form to the right of the statements. The line in the table contains the name and value of the (first) tag in the statement line (Fig. 13.12).

1			
2	□	"IEC_Counter_1".GTCU(CU := "Automatic mode"	"Automatic mode" TRUE
3		AND "Left_sensor"	"Left_sensor" TRUE
4		AND "Clock 0,5 Hz",	"Clock 0,5 Hz" FALSE
5		R := #CountValue >= 16,	#CountValue 3
6		PV := 0,	
7		CV => #CountValue);	#CountValue 3
8			

Fig. 13.12 SCL program status

If the statement line contains several tags, a table with all tags is displayed when you position the cursor in the statement line. The table line can also be opened. It will then show all of the tags of the statement line with the monitored values.

If the line contains one of the IF, WHILE, or REPEAT statements, the result of the condition (TRUE, FALSE) is shown in the line. Once the cursor is positioned in the statement line, a table with all of the tags of the statement line appears. The opened table line continuously shows the tag values of the statement line.

## Addressing and display format

By clicking on the *Absolute/symbolic operands* in the toolbar of the working window, you can supplement the symbolic addressing or the absolute addressing or hide the absolute address again.

You can select the display format, which is set as standard to *Automatic*: You select the tag value and select *Display format > ...* from the shortcut menu. ... *> Decimal, ... > Hexadecimal*, and ... *> Floating-point* are available, depending on the data type.

## Modifying operands in the program status

In the program status, you can define the signal states of binary tags and the values of digital tags using the programming device. This is usually only meaningful if these tags are not controlled from another point, for example inputs which receive their signal state from the I/O input channel during automatic updating of the process image.

Select the binary tag and then the *Modify > Modify to 0* command from the shortcut menu if the binary tag is to be set to signal state “0” or *Modify > Modify to 1* if the binary tag is to be set to signal state “1”. In the case of digital tags, select the *Modify > Modify operand...* command from the shortcut menu and specify the desired value.

## Up-to-dateness of the tag values

Black tag values come from the current monitoring cycle. Grey ones come from a cycle that was processed earlier. If a tag value is not displayed, the corresponding tag is not processed or has an unknown status.

If no value can be shown for a tag or event due to missing compilation options, the cell in the Value column contains three question marks on a yellow background. In this case, activate the *Create extended status information* attribute in the block properties and load the block again into the CPU.

## Block calls in the monitored block

If the tested statement line contains a block call, you can continue the program status in the called block : You select the block call and then the *Monitor* command from the shortcut menu. The program status then changes to the called block.

### 13.4.4 Monitoring with the PLC tag table

To monitor the tag table, double-click on the corresponding PLC tag table. Click the *Monitor all* icon in the toolbar. The PLC tag table changes to online mode and the *Monitor value* column is displayed. You can now monitor the tag values (Fig. 13.13).

	Name	Data type	Address	Retain	Monitor value	Comment
1	M1 on manu	Bool	%I2.1	<input type="checkbox"/>	FALSE	Switch on ramp manually
2	M1 off manu	Bool	%I2.2	<input type="checkbox"/>	FALSE	Switch off ramp manually
3	M1 Fault	Bool	%I2.3	<input type="checkbox"/>	FALSE	Ramp motor fault
4	M1 Start	Bool	%Q2.1	<input type="checkbox"/>	FALSE	Start ramp motor
5	Automatic mode	Bool	%M2.0	<input checked="" type="checkbox"/>	TRUE	Main gate: Automatic mode
6	Manual mode	Bool	%M2.1	<input type="checkbox"/>	FALSE	Main gate: Manual mode
7	Jog mode	Bool	%M2.2	<input type="checkbox"/>	FALSE	Main gate: Jog mode
8	Lamp check	Bool	%M2.3	<input type="checkbox"/>	FALSE	Switch on all lamps
9	Gate to stop	Bool	%I2.0	<input type="checkbox"/>	FALSE	Main gate: Stops the gate
10	M1 on auto	Bool	%M12.1	<input type="checkbox"/>	FALSE	Switch on ramp automatically
11	M1 off auto	Bool	%M12.2	<input type="checkbox"/>	FALSE	Switch off ramp automatically

Fig. 13.13 Monitoring with the PLC tag table

### 13.4.5 Monitoring of data tags

The tag values saved in a data block can be monitored during runtime. This works for all types of data blocks: global, instance and type data blocks. The program status shows the value of the data tags in the *Monitor value* column.

To monitor the data tags, open the data block, for example with a double-click in the project tree, and click on the *Monitor all* icon in the toolbar of the working window. The *Monitor value* column with the current values of the data tags is displayed. The monitored value is the value that exists in the work memory at the time of reading. A further click on the *Monitor all* icon exits monitoring mode.

Please note that tag values displayed in monitoring mode can originate from different program cycles.

You can “freeze” the monitor values. With monitoring mode switched on, click on the *Snapshot of the monitored values* icon in the toolbar of the working window. A new column *Snapshot* with the currently present monitor values is displayed. If you

	Name	Data type	Start value	Monitor value	Retain	Comment
1	Static					
2	CU	Bool	false	TRUE	<input type="checkbox"/>	
3	CD	Bool	false	FALSE	<input type="checkbox"/>	
4	R	Bool	false	FALSE	<input type="checkbox"/>	
5	LD	Bool	false	FALSE	<input type="checkbox"/>	
6	QU	Bool	false	TRUE	<input type="checkbox"/>	
7	QD	Bool	false	FALSE	<input type="checkbox"/>	
8	PV	Int	0	0	<input type="checkbox"/>	
9	CV	Int	0	15	<input type="checkbox"/>	

Fig. 13.14 Program status for data tags, the data for a counter in the example

would like to use the monitored actual values as start values, copy the desired values from the *Monitor value* column into the *Start value* column.

Fig. 13.14 shows the monitoring function for an instance data block for a counter function.

### 13.4.6 Testing with watch tables

Watch tables contain tags whose values can be monitored and modified (controlled) during runtime. Any combination of tags is possible, and a specially tailored watch table can therefore be created for each debugging case.

The following functions can be performed with a watch table:

- ▷ Monitor tags
- ▷ Modify tags
- ▷ Force tags
- ▷ “Enable peripheral outputs” and “Modify now”

Tags from the peripherals, inputs, outputs, and bit memory areas as well as tags from data blocks (global, instance, and type data blocks) can be used in watch tables.

### Creating watch tables

Underneath a PLC station in the project tree there is the *Watch and force tables* folder with the watch tables and the force table. Further subfolders can be created within this folder in order to structure the watch tables: Select the *Watch and force tables* folder and then the *Add group* command from the shortcut menu. You can assign separate names to the new subfolders and the watch tables by using the *Rename* command from the shortcut menu.

In order to create a new watch table, double-click on the *Add new watch table* command. In the empty table, enter the names of the tags line by line and the display

	Name	Address	Display format	Monitor value	Monitor with trig...	Modify with trigge
1	"M1 Fault"	%I2.3	Bool	FALSE	Permanently, at s...	Permanent
2	"Clock 2 Hz"	%M0.3	Bool	FALSE	Permanent	Permanent
3	"Main_gate".Switch_on_1	%DB2.DBX6.0	Bool	FALSE	Permanent	Permanent
4	"Main_gate".Switch_on_2	%DB2.DBX6.1	Bool	FALSE	Permanently, at s...	Permanent
5		%DB2.DBW6	DEC_signed	0	Permanent	Permanent
6		%DB2.DBW8	Hex		Permanent	Permanent
7	"Left_sensor":P	%I1.0:P	Bool		Permanent	Permanent
8	"Left_sensor"	%I1.0	Bool	TRUE	Permanent	Permanent
9	"Right_sensor"	%I1.7	Bool	FALSE	Permanent	Permanent
10	"Beacon"	%Q0.3	Bool	FALSE	Permanently, at e...	Permanent
11	"Beacon":P	%Q0.3:P	Bool		Permanent	Permanent
12	"Automatic mode"	%M2.0	Bool	TRUE	Permanent	Permanent

Fig. 13.15 Example of monitoring of tags in expanded mode



format from a drop-down list. The display format may differ from the data type of the tag. You can enter a short explanatory text for each tag in the comment column. In the watch table, you create a blank line between filled-in lines by moving a blank line or by copying and pasting. The tags entered with names must previously have been defined in the PLC tag table or in a data block.

The globally applicable tags (peripherals, inputs, outputs, and bit memories) and data tags from data blocks for which the *Optimized block access* attribute is not activated can also be entered with their memory address (absolute address) in the *Address* column (Fig. 13.15).

### Monitoring and modifying with triggers

The watch tables permit specification of the monitoring and control time. The following are available:

- ▷ Permanent  
In each program cycle, the inputs are monitored and modified at the start of the cycle prior to execution of the main program, and the outputs at the end of the cycle following execution.
- ▷ Permanently, at start of scan cycle  
In each program cycle, the tags are monitored and modified prior to execution of the main program (useful for inputs or tags which control functions).
- ▷ Once only, at start of scan cycle  
The tags are monitored and modified once only prior to execution of the main program (useful for inputs or tags which control functions).
- ▷ Permanently, at end of scan cycle  
In each program cycle, the tags are monitored and modified following execution of the main program (useful for outputs or tags which are controlled by functions).
- ▷ Once only, at end of scan cycle  
The tags are monitored and modified once only following execution of the main program (useful for outputs or tags which are controlled by functions).
- ▷ Permanently, at transition to STOP  
The tags are continuously monitored and modified during transition to the STOP mode.
- ▷ Once only, at transition to STOP  
The tags are monitored and modified once only during transition to the STOP mode.

It is also possible to control tags using the *Online > Modify > Modify now* command in the menu bar of the project view. The selected tags are then updated as quickly as possible.

## Using watch tables

In order to use the watch tables, the programming device must be connected online to the PLC station. Open the watch table and start the desired function (monitor, modify, force) using the corresponding symbol.

If an online connection to the CPU has not yet been established, STEP 7 searches for accessible devices. If necessary, set the interface used in the programming device in the dialog window *Go online*, select the PLC station found, and click on the *Go online* button.

You can call the test functions in the shortcut menu or using the icons in the toolbar of the working window shown in Fig. 13.16.

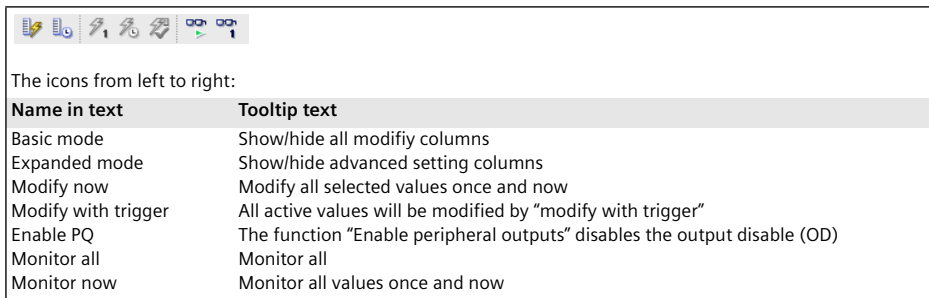


Fig. 13.16 Icons in the toolbar of the watch table

### 13.4.7 Monitoring tags using watch tables

Double-click to open the watch table and select one of the symbols *Monitor all* or *Monitor now*. An online connection to the CPU will be established.

In standard mode, the *Name*, *Address*, *Display format*, *Monitor value*, and *Comment* columns are displayed. The *Monitor value* column shows the tag value in the display format which has been set in the *Display format* column. If *Monitor now* was selected, *Monitor value* shows a snapshot; if *Monitor all* was selected, the values in the *Monitor value* column are updated continuously.

The time of monitoring corresponds to the trigger mode *Permanent* (see "Monitoring and modifying with triggers" on page 448). You can stop the current monitoring by clicking again on the *Monitor all* icon.

Please note that peripheral outputs generally cannot be monitored, just like non-existing data tags (in Fig. 13.15 the output %Q0.3 and the data word %DB2.DBW8). Undefined data tags can be specified as an address (%DB2.DBW6 in the example) if the *Optimized block access* attribute is not activated in the data block.

#### Monitor with trigger

In expanded mode, you can select the trigger time at which the tag values are read out of the CPU. If you click on the *Expanded mode* button in the toolbar, the columns

*Monitor with trigger* and *Modify with trigger* are displayed. You can then define the read time for each tag from a drop-down list.

Tag values which are read out once only or which are not read out (yet) are shown in the *Monitor value* column with a gray background; permanently read values have an orange background

Fig. 13.5 on page 432 shows an example of monitoring with watch table in extended mode.

### 13.4.8 Modifying tags using watch tables

Double-click to open the watch table and select the symbol *Basic mode*. In addition to the *Name*, *Address*, *Display format*, *Monitor value*, and *Comment* columns, the *Modify value* and *Tag selection* (represented by a lightning icon) columns are now displayed (Fig. 13.17).

	Name	Address	Display format	Monitor value	Modify value	Tag selection	Comment
1	*M1 Fault*	%I2.3	Bool	FALSE		<input type="checkbox"/>	
2	*Clock 2 Hz*	%M0.3	Bool	FALSE		<input type="checkbox"/>	
3	*Main_gate*.Switch_on_1	%DB2.DBX6.0	Bool	FALSE	TRUE	<input checked="" type="checkbox"/>	⚠
4	*Main_gate*.Switch_on_2	%DB2.DBX6.1	Bool	FALSE		<input type="checkbox"/>	
5	*Main_gate*.Switch_on_2	%DB2.DBW6	DEC_signed	0	32	<input checked="" type="checkbox"/>	⚠
6		%DB2.DBW8	Hex			<input type="checkbox"/>	
7	*Left_sensor*.P	%I1.0:P	Bool	FALSE		<input type="checkbox"/>	⚠
8	*Left_sensor*	%I1.0	Bool	TRUE		<input type="checkbox"/>	
9	*Right_sensor*	%I1.7	Bool	FALSE		<input type="checkbox"/>	
10	*Beacon*	%Q0.3	Bool	FALSE		<input type="checkbox"/>	
11	*Beacon*.P	%Q0.3:P	Bool	TRUE	TRUE	<input checked="" type="checkbox"/>	⚠
12	*Automatic mode*	%M2.0	Bool	TRUE		<input type="checkbox"/>	

**Fig. 13.17** Example of controlling tags with an error message

Enter the value to which the tag is to be set in the *Modify value* column, and activate the check box in the *Tag selection* column if the associated tag is to be modified. A yellow triangle with exclamation mark indicates that the selected tag has not yet been modified.

It is recommendable to switch on monitoring mode prior to modification. An online connection to the CPU is then already made, and the effects of modification can be monitored.

*Caution! Make sure when modifying tags that no dangerous states can result!*

To modify the activated tags, click on the symbol *Modify now*. The tags activated in the *Tag selection* column are immediately set (as fast as possible) to the control value. If a tag is immediately overwritten after the modification by a value from the program – for example if a switched-on input has been controlled to “0” and the

process image updating overwrites the control value again – the yellow triangle appears again in the *Tag selection* column.

Alternatively, modification can be triggered by means of the *Online > Modify > Modify now* command from the main menu or the *Modify > Modify now* command from the shortcut menu. The *Modify > Modify to 0* and *Modify > Modify to 1* commands from the shortcut menu immediately control the binary tag selected in the watch table.

Please note that peripheral inputs can never be modified (in Fig. 13.17 the peripheral inputs %I1.0:P and %I1.7:P). Only the tags visible in the table are modified. Multiple controlling of a tag or parts of a tag in the monitor table is not allowed (in the figure: the data bit %DB2.DBX6.0 and the data word %DB2.DBW6).

### Modifying with triggers

In expanded mode, you can select the trigger time at which the tag values are modified in the CPU. If you click on the *Expanded mode* button in the toolbar, the columns *Monitor with trigger* and *Modify with trigger* are displayed. You can then define the control time for each tag from a drop-down list.

If you click the symbol *Modify with trigger*, all activated tags are updated (following confirmation) with the control value in accordance with the trigger conditions. Clicking on the icon again exits permanent control.

Alternatively, modifying can be triggered or terminated using the *Online > Modify > Modify with trigger* command from the main menu or the *Modify > Modify with trigger* command from the command menu.

#### 13.4.9 Enable peripheral outputs and “Modify now”

In STOP mode, the output modules are normally disabled by the OD signal (command output disable); the *Enable peripheral outputs* (PQ) function can be used to switch off the OD signal so that you can also control the output modules with the CPU at STOP. Controlling is carried out using a watch table. An application for this would be checking the wiring of the outputs in STOP mode and without a user program.

*Caution! Make sure that no dangerous states can occur with “Enable peripheral outputs”!*

Prerequisites for *Enable peripheral outputs*: An online connection exists to the CPU which is in the STOP mode. A watch table with the peripheral outputs to be controlled has been created and expanded mode is switched on. *Exit all force jobs* (see next Chapter)!

Open the watch table, click on the *Basic mode* icon, enter the control value in the *Modify value* column, and activate the selection checkbox. You switch off the command output disable (OD) for the output modules via the *Enable PQ* icon in the toolbar. You can now control the tags using the *Modify now* icon. Alternatively you can

	Name	Address	Display format	Monitor value	Modify value	Comm
1	"M1 Fault"	%I2.3	Bool	FALSE		
2	"Clock 2 Hz"	%M0.3	Bool	TRUE		
3	"Main_gate".Switch_on_1	%DB2.DBX6.0	Bool	FALSE		
4	"Main_gate".Switch_on_2	%DB2.DBX6.1	Bool	FALSE		
5		%DB2.DBW6	DEC_signed	0		
6		%DB2.DBW8	Hex			
7	"Left_sensor":P	%I1.0:P	Bool			
8	"Left_sensor"	%I1.0	Bool	TRUE		
9	"Right_sensor"	%I1.7	Bool	FALSE		
10	"Beacon"	%Q0.3	Bool	FALSE		
11	"Beacon":P	%Q0.3:P	Bool		TRUE	
12	"Automatic mode"	%M2.0	Bool	TRUE		

**Fig. 13.18** Example for *Enable PQ* and *Modify now*

use the *Online > Modify > Enable peripheral outputs* and *Online > Modify > Modify now* commands from the main menu (Fig. 13.18).

You can then control the peripheral outputs for as long as *Enable peripheral outputs* is switched on.

You deactivate the *Enable peripheral outputs* function – the OD signal is then switched on again – by selecting the *Enable PQ* function again, by changing the CPU mode, or by canceling the online connection.

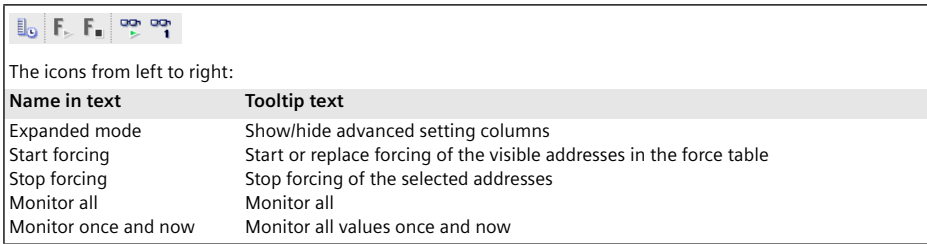
### 13.4.10 Forcing tags

Tags can be preassigned fixed values. This action is referred to as forcing. A CPU 1200 can force tags out of the I/O area, i.e. assign a peripheral input with signal state “0” or “1”, regardless of the voltage at the input terminal, or a peripheral output (the output terminal) with a signal state of “0” or “1” regardless of the program function. The tags to be forced are entered in the force table. The force table is present once for a CPU and cannot be copied or renamed. Forcing is then only possible if the *Enable peripheral outputs* function is deactivated.

Please note the following special features when using the force function: forcing is sent to the CPU by means of a force job. *The force job remains active even if online mode has been terminated and the online connection to the programming device canceled!*

*The force job also remains active when the CPU is switched off and on again!* Forcing can only be terminated by the *Online > Force > Stop forcing* command; this command deletes the force job in the CPU.

You can call the test functions when forcing from the shortcut menu or using the icons in the toolbar of the working window shown in Fig. 13.19.



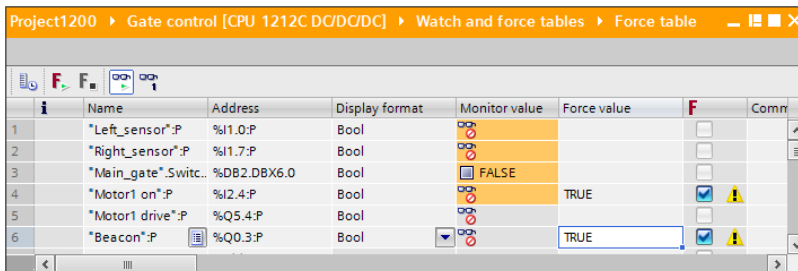
**Fig. 13.19** Icons in the toolbar of the force table

### Filling a force table

Open the force table by double-clicking in the project tree in the *Watch and force tables* folder.

In the empty table, enter the names of the tags line by line and the display format from a drop-down list. The display format may differ from the data type of the tag. You can enter a short explanatory text for each tag in the *comment* column.

The tags entered with names must previously have been defined in a PLC tag table or in a data block. The globally applicable tags (peripherals, inputs, outputs and bit memories) and data tags from data blocks for which the *Optimized block access* attribute is not activated, can also be entered with their memory address (absolute address) in the *Addresses* column (Fig. 13.20).



**Fig. 13.20** Example of forcing of peripheral inputs and outputs

### Monitoring tags in the force table

The entered tags can be monitored. The *Expanded mode* icon in the toolbar of the working window opens the *Monitor with trigger* column. You can set the monitoring conditions here. You start monitoring by clicking on the *Monitor all* symbol (refer to Chapter 13.4.6 “Testing with watch tables” on page 447 for details).

### Forcing with the force table

You can call the test functions when forcing from the shortcut menu or using the icons in the toolbar of the working window shown in Fig. 13.19.

To carry out forcing, enter a value in the *Force value* column and activate the checkbox in the *Force* column (tag selection depicted by a red “F”). A yellow triangle with exclamation mark indicates that the selected tag has not yet been forced.

It is recommendable to switch on monitoring mode prior to forcing. An online connection to the CPU is then already made, and the effects of forcing can be monitored.

*Caution: Make sure when forcing tags that no dangerous states can result!*

The *Start forcing* icon sends a force job to the CPU which contains the tags selected for forcing. Forcing is effective immediately. A forced tag is marked with a red “F” in the first column of the watch table. It is not possible to force parts of a tag, for example of individual peripheral bits, if the peripheral byte is already being forced.

To exit forcing for individual tags, deactivate the checkbox in the tag selection and click on the *Start forcing* icon again. A new force job is sent to the CPU which terminates forcing for the tags which are no longer selected.

You exit forcing for all tags using the *Stop forcing* icon. A new force job is sent to the CPU which terminates forcing for all forced tags.

*Note that termination of forcing leave the tags in their last state!* Only the force job is deleted. For example, an output of a digital module remains in signal state “1” after termination of forcing if it is not controlled otherwise by the program.

As an alternative to forcing using the icons, you can select one or more tags in the force table and then the *Force > Force to 0*, *Force > Force to 1*, *Force > Force all*, and *Force > Stop forcing* commands from the shortcut menu or the commands from the main menu under menu item *Online > Force > ...* .

### **Forcing in association with memory card**

The offline project data that is copied to a memory card (program or transfer card) does not contain any force jobs. If the user program is transferred from a memory card to the CPU, the force jobs are deleted in the internal load memory.

If an empty program card is plugged in and the user program is copied from the internal load memory into the memory card, which is then used as an external load memory, the force jobs are also copied.

If a program card is plugged in during runtime and force jobs are set up, the force jobs are saved on the memory card (in the external load memory).

Usage of the memory card is described in Chapter 13.2.4 “Working with the memory card” on page 428.

---

# 14 Distributed I/O

## 14.1 Introduction, overview

Distributed I/O is the term used for input/output modules connected to the central PLC station over a bus system. SIMATIC S7 uses the PROFINET IO, PROFIBUS DP, and AS-Interface (AS-i) bus systems.

The distributed I/O is handled like the central I/O. The distributed inputs/outputs are in the same address volume as the central inputs/outputs, and therefore the addresses of the distributed I/O must not overlap with those of the central I/O. The distributed I/Os can be addressed via the following operand areas: peripheral inputs (I:P) and peripheral outputs (Q:P) and – if they are present in the process image – also via the inputs (I) and outputs (Q).

Transfer between the distributed modules and the central CPU is carried out “automatically” and you need not take this into account when addressing.

Data transfer to and from the distributed I/O is controlled from a central point: With PROFINET IO it is the IO controller, with PROFIBUS DP it is the DP master, and with AS-Interface it is the AS-i master. The distributed stations – these are the IO devices with PROFINET IO, the DP slaves with PROFIBUS DP, and the AS-i slaves with AS-Interface – are the passive partners in the data transfer.

S7 stations and ET200 stations with a CPU can also be used as distributed I/O stations and these are then “intelligent” DP slaves or IO devices. While these stations are controlling their own modules (considered from their viewpoint as central modules), they also satisfy – when working at the same time as IO devices or DP slaves – the data requirements of the respective IO controller or DP master.

The distributed I/O is configured using the hardware configuration. PROFINET IO, PROFIBUS DP (with the CM 1243-5 module as the DP master) and AS-Interface (with the CM 1243-2 module as the AS-i master) are handled like subnets, in which the connections needed for the data transfer are “automatically” available.

Network transitions between the subnets can be produced using link and coupler modules which allow data exchange between the stations connected to the various networks. The programming device is able to handle programming and servicing functions over PROFINET IO and PROFIBUS DP. It can reach all (“intelligent”) stations connected to the subnets if the subnet gateways are present in stations with routing capability.



## 14.2 PROFINET IO

### 14.2.1 PROFINET IO components

PROFINET IO offers a standardized interface in accordance with IEC 61158 for industrial automation over Industrial Ethernet. An IO controller in the central programmable controller controls the data exchange with the distributed field devices which are referred to as IO devices (Fig. 14.1).

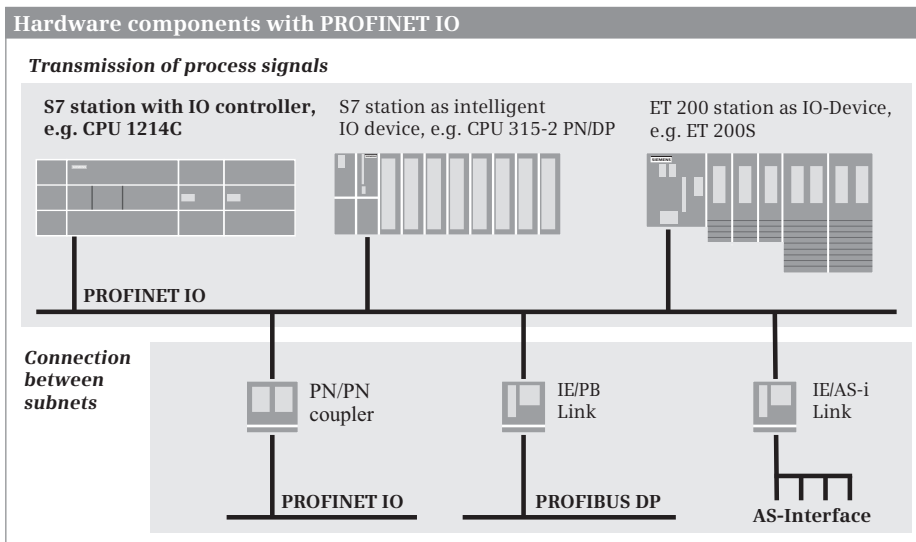
Industrial Ethernet can be designed physically as an electrical, optical, or wireless network. FastConnect Twisted Pair (FC TP) cables with RJ45 connections or Industrial Twisted Pairs (ITP) cables with sub-D connections are available for implementing the electrical cabling. Fiber-optic (FO) cabling can consist of glass fiber, PCF, or POF. It offers galvanic isolation, is impervious to electromagnetic influences, and is suitable for long distances. Wireless transmission uses the frequencies 2.4 GHz and 5 GHz with data transfer rates up to 54 Mbit/s (depending on the national approvals).

#### IO controller

The IO controller is the active participant on the PROFINET. It exchanges data cyclically with “its” IO devices. In an S7-1200 automation system, each CPU is also an IO controller.

#### IO devices

IO devices are the passive stations on the PROFINET IO. These can be stations with process inputs and outputs, routers, or link modules. Examples of IO devices



**Fig. 14.1** Components of a PROFINET IO system

from the ET 200 distributed I/O system are the ET 200eco, ET 200M, ET 200S, and ET 200pro.

IO devices with user data are distinguished as follows:

- ▷ Compact IO devices which are addressed like a single module
- ▷ Modular IO devices which can contain several modules or submodules which are addressed individually
- ▷ Intelligent IO devices with a configured transfer area as user data interface to the IO controller.

The compact and modular IO devices can be found in the hardware catalog under *Distributed I/O* and the corresponding ET200 system. The hardware catalog of STEP 7 V11 does not contain intelligent IO devices.

### **Coupling modules**

Bus couplers and link modules connect subnets and permit data exchange between stations connected on different subnets. The following are available for the Ethernet subnet:

- ▷ PN/PN coupler for connecting two Ethernet subnets
- ▷ IE/PB Link PN IO for connecting an Ethernet subnet to a PROFIBUS subnet
- ▷ IE/AS-i Link for connecting an Ethernet subnet to an AS-i subnet.

You can find the PN/PN coupler and the IE/AS-i link in the hardware catalog under *Other field devices > PROFINET IO > Gateway > Siemens AG > ...* and the IE/PB link under *Network components > Gateways > ...*

### **PROFINET IO system**

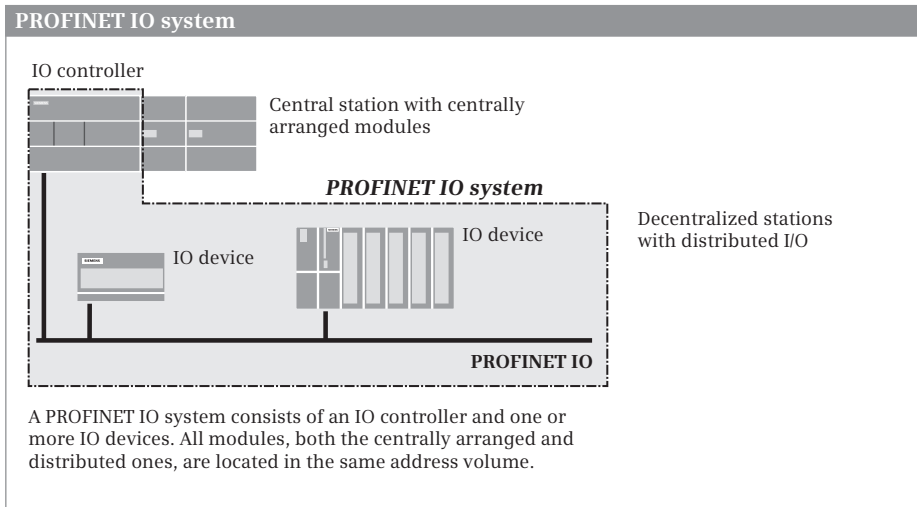
The IO controller and all IO devices controlled by it constitute a PROFINET IO system (Fig. 14.2). An IO device is supplied with data by its IO controller within an update time which is calculated by STEP 7 in specific intervals and in turn sends its data to the IO controller.

Several PROFINET IO systems can be operated in a PN/IE subnet.

#### **14.2.2 Addresses with PROFINET IO**

##### **Station addresses on the Ethernet subnet**

The stations on an Ethernet subnet which use the TCP/IP protocol are addressed via the *IP address*. This consists of four decimal numbers, each in the range from 0 to 255, and is represented by four bytes separated by dots, for example 192.168.1.3. This address consists of the subnet number and the actual station address, which one can extract with the subnet mask from the IP address. Example: If the subnet mask has the value 255.255.255.0, the subnet number for the above-mentioned IP address is 192.168.1 and the station address 3.



**Fig. 14.2** Schematic representation of a PROFINET IO system

Each station on the PROFINET is additionally assigned a device name and number. Further information on the station addresses in an Ethernet subnet can be found in Chapter 3.4.5 “Configuring a PROFINET subnet” on page 73.

### Geographic addresses with PROFINET IO

The geographic address identifies the slot of a module. With an IO device, the geographic address comprises the ID of the PROFINET IO system, the device number, the number of the slot, and possibly also a submodule number.

The PROFINET IO system ID is assigned by STEP 7 and is in the range from 100 to 115. Within the station, the “virtual” slot 0 (not physically present) represents the IO device. The modules with the user data are arranged in an IO device starting at slot 1.

### Logical addresses with PROFINET IO

The user data of the IO devices shares the range of logical addresses with the user data of the central modules in the S7 station with the IO controller. The logical addresses of all modules are within the range of peripheral inputs or outputs. This means that the addresses of the central modules must not overlap with those of the IO devices.

You use the logical address to address the user data, in other words the signal states of the digital input/output channels or the values at the analog input/output channels. Each byte of user data is unequivocally defined by the logical address. The logical address corresponds to the absolute address. A symbol (name) can be assigned to it so that it is easier to read (symbolic addressing). Further details can be found in Chapter 4.2 “Addressing” on page 85.

## Consistent user data transfer to and from IO devices

Data consistency means that a block of user data is handled together. With PROFINET IO a CPU 1200 transfers a data block with up to 64 bytes consistently. If longer data blocks are to be consistently transferred, use the system function DPRD\_DAT for reading and DPWR\_DAT for writing. These system functions are described in Chapter 14.3.4 “System functions for PROFINET IO and PROFIBUS DP” on page 470.

The handling of consistent user data areas in the user memory is described in Chapter 4.1.2 „Operand areas: inputs and outputs“ in Section “Consistent user data areas” on page 81.

### 14.2.3 Configuring PROFINET IO

#### General procedure

A prerequisite for configuration of the distributed I/O with PROFINET IO is a created project with a PLC station. To select the stations involved, start the hardware configuration in the Network view.

- ▷ The starting point for the configuration is the IO controller in a CPU 1200. The *IO controller* mode is preset.
- ▷ Assign a PROFINET IO system to the PN interface of the IO controller. The Ethernet subnet required is created automatically in the process.
- ▷ Select an IO device from the hardware catalog and drag it with the mouse into the working window.
- ▷ Link the IO device to the PROFINET IO system by dragging the PN interface of the IO device with the mouse to the PN interface of the IO controller.
- ▷ Repeat the three steps for every further IO device.
- ▷ To parameterize a PN interface, select it in the working window and set the desired properties in the inspector window.

The result is networking of the IO controller with the assigned IO devices to a PROFINET IO system (Fig. 14.3).

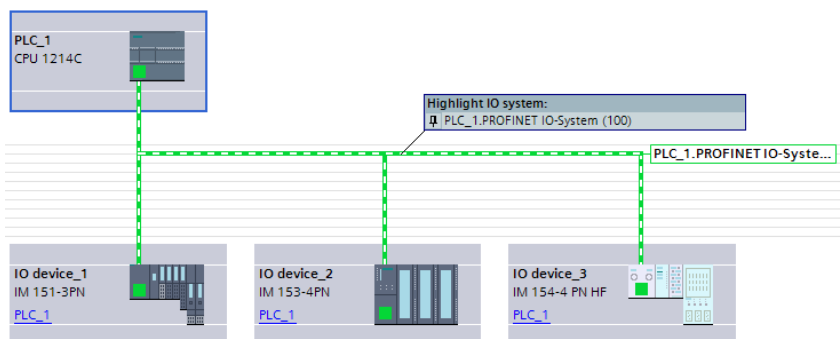


Fig. 14.3 Example of representation of a PROFINET IO system

You then make the parameter settings for the stations and the fitting with input/output modules in the Device view.

### **Configuring the IO controller in the Network view**

Prerequisite: You have created a project and a PLC station, for example a CPU 1200. Start the device configuration and select the *Network view* tab in the working window.

Select the PN interface shown in green in the graphic of the CPU and then the *Ethernet addresses* group in the *Properties* tab in the inspector window. Activate the *Set IP address in project* option and change the preset IP address and subnet mask if necessary. Information on the IP address can be found in Chapter 3.4.5 “Configuring a PROFINET subnet” on page 73. Activate the *Set PROFINET device name using a different method* option if you wish, for example, to set the IP address per user program.

Connect the PN interface to a PROFINET subnet. You can do this in the properties of the PN interface: Select an existing subnet under *Ethernet address* in the *Subnet* drop-down list or create a new subnet using the *Add new subnet* button. You can also click on the PN interface with the right mouse button and select the *Add subnet* command from the shortcut menu. A green subnet is shown with the name PN/IE\_x. You can change the name in the subnet properties.

Configure a PROFINET IO system. To do this, click with the right mouse button on the PN interface and select the *Assign IO system* command from the shortcut menu. A green/white marking is shown with the name <Station name>.PROFINET IO system (xxx). xxx is the number of the IO system. You can change the name and number in the properties of the PROFINET IO system.

### **Adding an IO device to the IO system**

With the left mouse button pressed, drag the desired IO device from the hardware catalog to the IO system on the working area. Fig. 14.3 shows three stations of the distributed I/O: an ET 200M station from the object tree *Distributed I/O > ET 200M > Interface modules > PROFINET > IM 153-4 PN > ...* , an ET200eco station from the object tree *Distributed I/O > ET 200eco PN > Compact modules PROFINET > DI/DO > 8DIO x DC24V / 1.3A 8xM12 > ...* and an ET 200S station from the object tree *Distributed I/O > ET 200S > Interface modules > PROFINET > IM 151-3 PN > ...* .

The interfaces of the IO devices are connected in the graphic with the green/white marking and are thus part of the PROFINET IO system.

The automatically assigned station name is applied as the PROFINET device name. You can change the name in the station properties and also the device number and IP address.

### **Configuring an IO device**

With the IO device selected, you can set its properties in the inspector window in the Device view. You fit a modular IO device with the desired modules or submodules from the hardware catalog and then set their parameters.

In the properties of the PROFINET interface, set the Ethernet addresses. Set the desired application in the *Advanced options* group.

#### 14.2.4 Real-time communication with PROFINET IO

PROFINET IO offers several types of data transfer:

- ▷ Non-time-critical data such as configuration and diagnostic information is transferred acyclically with the TCP/IP communication standard.
- ▷ User data (input/output information) is exchanged cyclically between the IO controller and the IO device (real-time RT) within a defined time period – the update time.
- ▷ Time-critical user data, e.g. for motion control applications, is transferred isochronously with hardware support (isochronous real-time IRT).

A permanent communication channel is reserved on the Ethernet subnet for IRT communication. RT communication – cyclic data exchange between the IO Controller and IO Devices – and non-real-time TCP/IP communication take place parallel to the update time. In this way, all three communication types can exist in parallel on the same subnet.

The configuration of IRT communication is not possible with STEP 7 V11.

#### Send clock in the PROFINET IO system

Cyclic data exchange is handled within a specific time frame, the send clock. STEP 7 calculates the send clock from the configuration information on the PROFINET IO system. The send clock is the shortest possible update time.

You configure the send clock in the interface properties of the IO controller. With the PN interface selected, select a value in the properties tab under *Advanced > Real time settings > IO communication* from the drop-down list *Shortest possible update interval*.

#### Update time and watchdog timer for IO devices

The update time is the period within which each IO device in the IO system has exchanged its user data with the IO controller. The update time corresponds to the send clock or a multiple thereof. You can increase the update time manually, for example to reduce the bus load. Under certain circumstances, you can reduce the update time for individual IO devices if you in return increase the update time for other devices whose user data can be exchanged non-time-critically.

If the IO device is not supplied by the IO controller with input or output data within the watchdog timer, it switches to a safe state. The watchdog timer is calculated as the product of the update time and “Accepted update cycles without IO data”.

You configure the times in the interface properties of the IO device. To do this, select the IO device and then the *PROFINET interface > Advanced options > Real time settings > IO cycle* group in the properties tab. Under *Update time*, select the *Can be set* option and then the update time from the drop-down list. To achieve automatic

adaptation to the send clock, activate the *Adapt update time when send clock changes* checkbox. You select the watchdog timer in the *Accepted update cycles without IO data* drop-down list.

## Real time

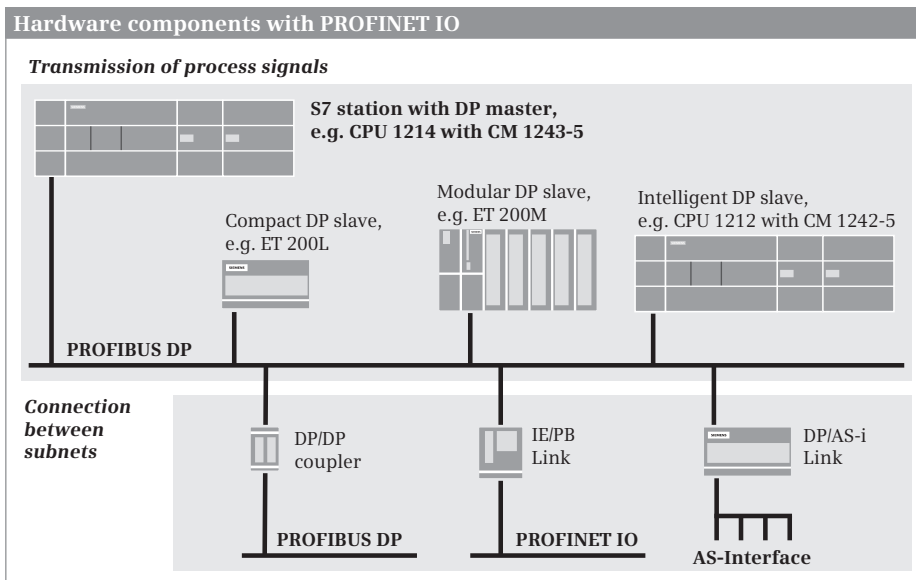
Real time (RT) means that a system processes external events within a defined time. If it responds predictably, it is called deterministic. In RT communication, transfer takes place at a specific point in time (send clock) within a defined interval (update time). PROFINET IO allows the use of standard network components for RT communication.

If not all data to be exchanged is transferred within the planned time frame, for example due to the addition of new network components, some data is distributed to other send clocks. This can result in an increase in the update time for individual IO devices.

## 14.3 PROFIBUS DP

### 14.3.1 PROFIBUS DP components

PROFIBUS DP offers an interface in accordance with the international standard IEC 61158/61784 for transmission of process data between an “interface module” in the central programmable controller and the field devices. This “interface module” is referred to as DP master and the field devices as DP slaves (Fig. 14.4).



**Fig. 14.4** Hardware components with PROFIBUS DP

The PROFIBUS network can be designed physically as an electrical network, optical network, or wireless coupling with different data transfer rates. The length of a segment depends on the transfer rate and is adjustable in steps for an electrical or optical network from 9.6 Kbit/s to 12 Mbit/s. The electrical network can be configured as a bus or tree structure. It uses a shielded, twisted two-wire cable (RS485 interface).

The optical network uses either plastic, PCF or glass fiber-optic cables. It is suitable for long distances, offers galvanic isolation, and is impervious to electromagnetic influences. Using optical link modules (OLMs) it is possible to construct a linear, ring, or star topology. An OLM also provides the connection between electrical and optical networks with a mixed design. A cost-optimized version is the design as a linear topology with integral interface and optical bus terminal (OBT).

Using the PROFIBUS Infrared Link Module (ILM), a wireless connection can be provided for one or more PROFIBUS slaves or segments with PROFIBUS slaves. The maximum data transfer rate of 1.5 Mbit/s and the maximum range of 15 m mean that communication is possible with moving system components.

### **DP master**

The DP master is the active station on the PROFIBUS. It exchanges data cyclically with “its” DP slaves. A DP master can be:

- ▷ A CPU with integral PROFIBUS interface (with the letters “DP” in the short designation, e.g. CPU 315-2 PN/DP)
- ▷ A communication module in the PLC station (e.g. CM 1243-5)
- ▷ The IE/PB Link PN IO

The CM 1243-5 communication module is not included in the hardware catalog of STEP 7 V11 on delivery. It must be integrated with a hardware support package (HSP) (see Section “Expanding the hardware catalog” on page 60). After the installation, the CM 1243-5 communication module is in the hardware catalog under ...  
> *Communication modules* > *PROFIBUS* > *CM 1243-5* > ...

### **DP slaves**

The DP slaves are the passive stations on the PROFIBUS DP. These can be stations with process inputs and outputs, routers, or link modules. Examples of DP slaves from the ET200 distributed I/O system are the ET 200eco, ET 200M, ET 200S, and ET 200pro.

DP slaves with user data are distinguished as follows:

- ▷ Compact DP slaves which are addressed like a single module
- ▷ Modular DP slaves which can contain several modules or submodules which are addressed individually
- ▷ Intelligent DP slaves with a configured transfer area as user data interface to the DP master

Intelligent DP slaves contain a user program which controls the subordinate (own) modules. The user data interface to the DP master is a transfer area which can be



divided into different address areas. Examples of intelligent DP slaves are S7 stations with CPUs having an integral DP slave functionality, as well as the ET 200S distributed I/O station with the IM 151-8 PN/DP CPU interface and the ET 200pro distributed I/O station with the IM 154-8 PN/DP CPU interface.

STEP 7 Basic V11 only supports an S7-1200 station (as an intelligent DP slave) with the CM 1242-5 communication module. The CM 1243-5 communication module is not included in the hardware catalog of STEP 7 V11 on delivery. It must be integrated with a hardware support package (HSP) (see Section “Expanding the hardware catalog” on page 60). After the installation, the CM 1242-5 communication module is in the hardware catalog under ... > *Communication modules* > *PROFIBUS* > *CM 1242-5* > ...

### **Coupling modules**

Bus couplers and link modules connect subnets and permit data exchange between stations connected on different subnets. The following are available for the PROFIBUS subnet:

- ▷ DP/DP coupler for connecting two PROFIBUS subnets
- ▷ DP/AS-i link for connecting a PROFIBUS subnet to an AS-i subnet
- ▷ IE/PB Link PN IO for connecting an Ethernet subnet to a PROFIBUS subnet

You can find the DP/DP coupler and the DP/AS-i link in the hardware catalog under *Other field devices* > *PROFIBUS DP* > *Gateways* > *Siemens AG* > ... and the IE/PB link under *Network components* > *Gateways* > *IE/PB Link PN IO* > ... .

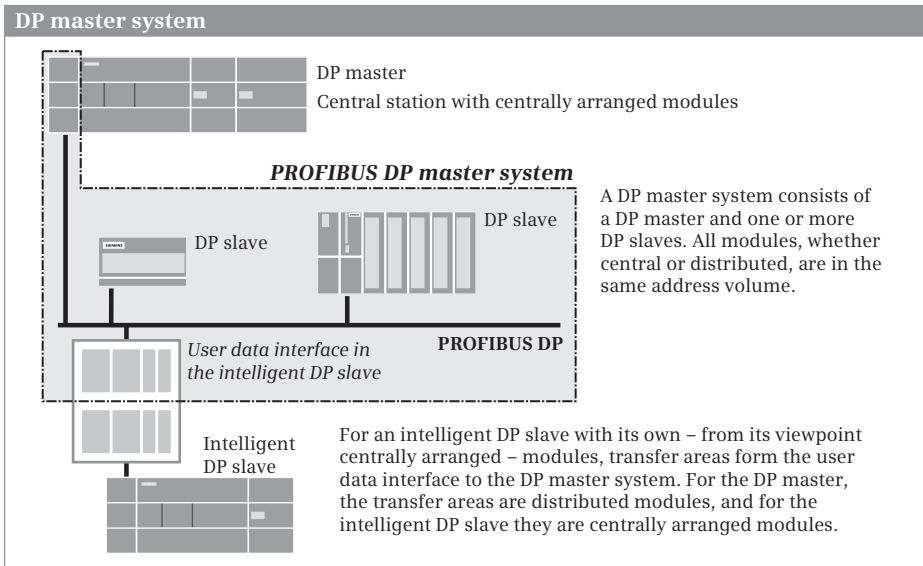
### **PROFIBUS DP master system**

The DP master and all DP slaves controlled by it form a PROFIBUS DP master system (Fig. 14.5). The update time within which a DP slave receives data from its DP master and in turn sends data to the DP master depends on the number of DP slaves in the master system.

PROFIBUS DP is usually operated as a “mono-master system”, i.e. a single DP master in a bus segment controls several DP slaves. Except for a temporary programming device for diagnostics and servicing, the DP master is the only master on the bus. You can also install several DP master systems in a PROFIBUS subnet (“multi-master system”). However, this increases the response time in individual cases since, once a DP master has supplied “its” DP slaves, the access privileges are assigned to the next DP master which in turn supplies “its” DP slaves, etc.

### **DPV0, DPV1, and S7-compatible operating modes**

DP slaves and DP masters are available with different scopes of PROFIBUS functions. DP slaves with a range of functions in accordance with EN 50170 (abbreviated to: “DPV0 slaves”) can handle the cyclic exchange of process data. DP slaves with a range of functions in accordance with IEC 61158/EN 50170 Volume 2 (abbreviated to: “DPV1 slaves”) have an extended functionality in addition to the cyclic data exchange, e.g. an increased diagnostics and parameterization capability through



**Fig. 14.5** Schematic representation of a PROFIBUS DP master system

the use of data records transferred acyclically or the use of new types of interrupt. PROFIBUS devices from Siemens (“DP S7 slaves”), which can handle further functions in addition to the cyclic data exchange, e.g. diagnostic interrupts, have the operating mode “S7-compatible”.

The operating modes of DP master and DP slaves must be matched to each other. DP masters in operating mode “DPV0” control DPV0 slaves, those in operating mode “S7-compatible” control DPV0 and DP S7 slaves. DPV1 masters from Siemens can control DP slaves with all operating modes.

The CM 1243-5 module is a DPV1-Master, the CM 1242-5 is a DPV1-Slave.

### 14.3.2 Addresses with PROFIBUS DP

#### Station addresses on PROFIBUS DP

Each station on the PROFIBUS subnet has a unique address within the subnet – the station address (station number) – which distinguishes it from all other stations on the subnet. The station (the DP master or a DP slave) is addressed on the PROFIBUS by means of this station address.

STEP 7 assigns the station addresses automatically and you can change the addresses within the specified range. You set the highest station address in the properties of the subnet or DP master system under *Network settings*.

### **Geographic address with PROFIBUS DP**

The geographic address identifies the slot of a module. With a DP slave, the geographic address comprises the ID of the DP master system, the station number, and the slot number.

The DP master system ID is assigned by STEP 7 and is in the range from 1 to 32 for a DP master integrated in the CPU.

Slot numbering of a DP slave depends on its type. If it is integrated using a GSD file, the entries in the GSD file determine the slot at which the I/O modules start. With DP standard slaves, the slots for I/O modules start at 1. The slot numbering of a DP S7 slave depends on the slots of an S7-300 station. Slots 1 (power supply) and 3 (expansion unit interface module) remain vacant. Slot 2 (CPU) corresponds to the interface module (header module) of the modular DP slave. The signal modules (SM) are positioned starting at slot 4. There is also the “virtual” slot 0 (not physically present); this represents the complete station.

### **Logical addresses with PROFIBUS DP**

The user data of the DP slaves share the range of logical addresses with the user data of the central modules in the DP master station. The logical addresses of all modules are within the range of peripheral inputs or outputs. This means that the addresses of the central modules must not overlap with those of the DP slaves.

You use the logical address to address the user data, in other words the signal states of the digital input/output channels or the values at the analog input/output channels. Each byte of user data is unequivocally defined by the logical address. The logical address corresponds to the absolute address; a symbol (name) can be assigned to it so that it is easier to read (symbolic addressing). Further details can be found in Chapter 4.2 “Addressing” on page 85.

### **Consistent user data transfer to and from DP slaves**

Data consistency means that a block of user data is handled together. With PROFIBUS DP a CPU 1200 transfers a data block with up to 64 bytes consistently. If longer data blocks are to be consistently transferred, use the system function DPRD\_DAT for reading and DPWR\_DAT for writing. These system functions are described in Chapter 14.3.4 “System functions for PROFINET IO and PROFIBUS DP” on page 470.

The handling of consistent user data areas in the user memory is described in Chapter 4.1.2 „Operand areas: inputs and outputs“ in Section “Consistent user data areas” on page 81.

### **User data interface with intelligent DP slaves**

With the compact and modular DP slaves, the addresses of the inputs and outputs are together with the addresses of the central modules in the address volume of the DP master. With intelligent DP slaves (abbreviated to: I-slaves), the input/output modules of the DP slaves are assigned to the slave CPU. Every intelligent DP slave

therefore has a user data interface as common memory area with the DP master whose size depends on the slave CPU used.

The user data interface can be divided into several areas of different length and data consistency. The individual areas then respond like modules whose lowest address is the module start address. From the viewpoint of the DP master, the I-slave then appears like a compact or modular DP slave depending on the division.

A transfer area which is represented as an input module from the viewpoint of the DP master is an output module from the viewpoint of the DP slave and vice versa. The logical addresses on the master side are in the address volume of the DP master and the logical addresses on the slave side in the address volume of the DP slave. The addresses on the master side can be different from those on the slave side.

You address a transfer area like a peripheral input (I:P) or peripheral output (Q:P). You can address transfer areas with addresses in the area of the process image like inputs (I) or outputs (Q).

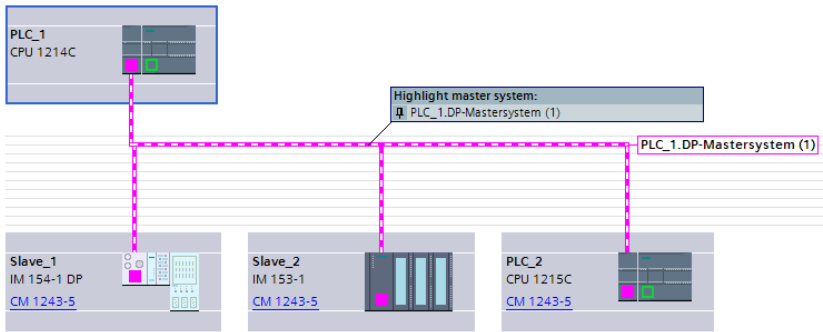
### 14.3.3 Configuring PROFIBUS DP

#### General procedure

A prerequisite for configuration of the distributed I/O with PROFIBUS DP is a created project with a PLC station. To select the stations involved, start the hardware configuration in the Network view.

- ▷ The starting point of the configuration is the DP master. For a CPU 1200, it is the CM 1243-5 communication module. The *DP master* mode is automatically activated.
- ▷ Assign a PROFIBUS DP master system to the DP interface of the DP master. The PROFIBUS subnet required is created automatically in the process.
- ▷ Set the bus parameters if necessary (highest PROFIBUS address, data transfer rate, profile).
- ▷ Select a DP slave from the hardware catalog and drag with the mouse into the working window.
- ▷ Link the DP slave to the DP master system by dragging the DP interface of the DP slave with the mouse to the DP interface of the DP master.
- ▷ Repeat the last two steps for every further DP slave.
- ▷ To parameterize the DP interface, select it in the working window and set the desired properties in the inspector window.
- ▷ Move an intelligent DP slave (an S7-1200 station with CM 1242-5) into the working window as an independent PLC station, add a CM 1242-5 communication module (the *DP slave* mode is automatically activated), assign the DP master and configure the transfer area of the user data interface.

The result is networking of the DP master with the assigned DP slaves to a PROFIBUS DP master system (Fig. 14.6).



**Fig. 14.6** Example of representation of a PROFIBUS DP master system

You then make the parameter settings for the stations and the fitting with input/output modules in the Device view.

### Configuring the DP master in the Network view

Prerequisite: You have created a project and a PLC station, for example a CPU 1200 with CM 1243-5. Start the device configuration and select the *Network view* tab in the working window.

In order to assign a DP master system to the interface, click with the right mouse button on the DP interface in the working window and select the *Assign master system* command from the shortcut menu. A PROFIBUS subnet and a magenta/white DP master system is created with the name *<Station name>.DP master system (<Master system ID>)*. You can change the master system ID in the properties of the DP master system under *General*.

You can change the highest PROFIBUS address, the data transfer rate, and the bus profile in the properties of the DP master system or in the properties of the PROFIBUS subnet under *Network settings*.

### Adding a DP slave to the DP master system

With the left mouse button kept pressed, drag the desired DP slave from the hardware catalog to the DP master system in the working window. Fig. 14.6 shows two stations of the distributed I/O: An ET 200eco station from the object tree *Distributed I/O > ET 200eco > Compact modules PROFIBUS > DI/DO > 8DI/8DO > ...* and an ET 200M station from the object tree *Distributed I/O > ET 200M > Interface modules > PROFIBUS > IM 153-2 OD > ...*. Furthermore, an S7-1200 station with a CPU 1215 and a CM 1242-5 communication module as intelligent DP slave were added.

The interfaces of the DP slaves are connected in the graphic with the magenta/white marking and are thus part of the PROFIBUS DP master system.

### Configuring a compact or modular DP slave

With the DP slave selected, you can set its properties in the Device view. You fit a modular DP slave with the desired modules or submodules from the hardware catalog and then set their parameters.

You set the PROFIBUS address in the properties of the PROFIBUS interface. Furthermore, it is possible in the *Module parameters* group and depending on the DP slave and application to set, for example, the startup property *Start up if preset configuration does not match actual configuration*, the DP interrupt mode, or the handling of options.

### Coupling an intelligent DP slave to the PROFIBUS DP master system

You initially create an intelligent DP slave (“I-slave”) as a stand-alone PLC station and then connect the DP interface of the I-slave to the DP master system. You can find the I-slaves in the hardware catalog in the *PLC* folder. For STEP 7 Basic V11 SP2, this is a CPU 1200 with CM 1242-5. Press and hold the left mouse button to drag the CPU from the object tree *PLC > SIMATIC S7-1200 > CPU > ...* and into the working window. In the same way, add the CM 1242-5 module from the object tree *PLC > SIMATIC S7-1200 > Communication module > PROFIBUS > CM 1242-5 > ...* to the station.

You establish a connection to the existing subnet if you drag the DP interface of the DP slave to the DP interface of another device on the subnet with the left mouse button pressed, for example to the DP interface of the DP master.

In the properties of the DP interface of the I-slave, the *DP slave* option under *Operating mode* is already activated. Select the assigned DP master from the drop-down list. The station is then added as DP slave to the PROFIBUS DP master system.

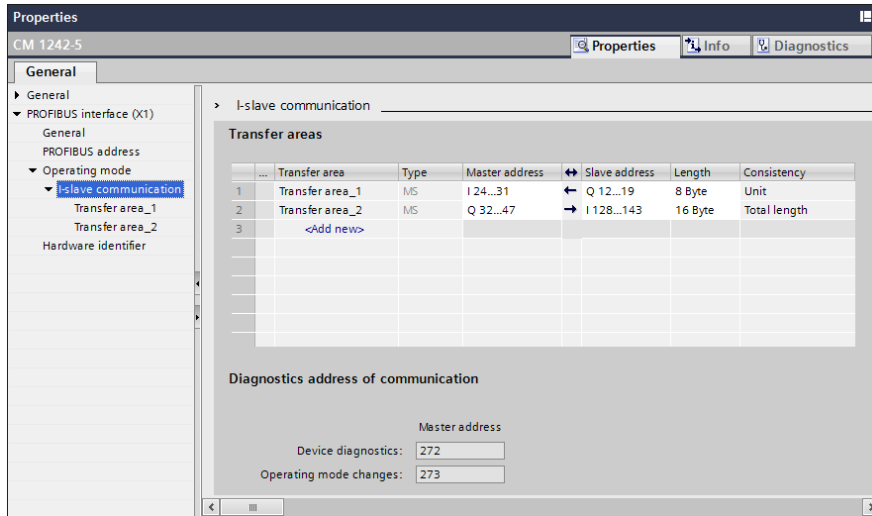
### Configuring the user data interface

You configure the user data interface to the DP master in the module properties of the I-slave. Select the CM module in the working window, and in the inspector window tab *Properties* in the group *PROFIBUS interface*, select the entry *Operating mode > I-slave communication*.

Double-click on *<Add new>* in the *Transfer areas* table. A new transfer area is created. You can change the name in the *Transfer area* column. In the *Transfer direction* column ( $\leftrightarrow$ ), click on the arrow to set the type of transfer area (arrow to the right  $\rightarrow$  means input area, arrow to the left  $\leftarrow$  means output area from the viewpoint of the I-slave).

Now set the start address in the *Slave address* column and the length of the transfer area in the *Length* column. The transfer area has a maximum length of 64 bytes. In the *Master address* column, set the start address which the transfer area has from the viewpoint of the DP master. In the *Consistency* column you can select between *Unit* and *Total length* (Fig. 14.7).

In this manner you can configure further transfer areas. The configured transfer areas are displayed in the *I-slave communication* properties group. If you click a



**Fig. 14.7** Example of configuration of the transfer areas of an I-slave

transfer area here, you obtain its details. You can also select here whether a cyclic update in the process image should take place in the DP master and/or in the DP slave.

#### 14.3.4 System functions for PROFINET IO and PROFIBUS DP

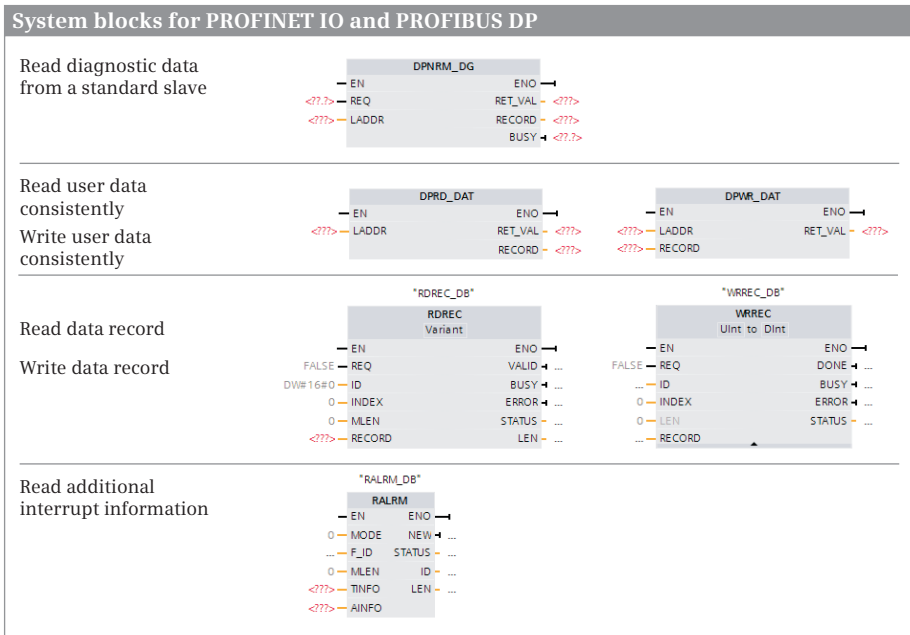
In connection with PROFINET IO and PROFIBUS DP, you can use the following system functions in the user program:

- ▷ DPNRM\_DG Read diagnostic data from a DP standard slave (only PROFIBUS DP)
- ▷ DPRD\_DAT Read user data consistently
- ▷ DPWR\_DAT Write user data consistently
- ▷ RDREC Read data record
- ▷ WRREC Write data record
- ▷ RALRM Read additional interrupt information

Fig. 14.8 shows the graphic representation of the function calls.

#### Common parameters

The parameter LADDR (for DPNRM\_DG, DPRD\_DAT and DPWR\_DAT), ID (for RDREC and WRREC) and F\_ID (for RALRM) define the addressed hardware object, for example a compact DP slave, a module in an IO device, or a transfer area of an intelligent DP slave. The parameters are supplied with the hardware ID. The hardware ID is assigned automatically during configuration and is displayed in the object properties. When programming, you can also select the hardware ID from a drop-down



**Fig. 14.8** Graphic representation of system blocks for PROFINET IO and PROFIBUS DP

menu: Double-click on the supply position in front of the parameter and select the desired object from the drop-down menu, which contains all of the configured relevant objects.

Depending on the function, the RECORD parameter defines the source area from which the data to be transferred is read, or the destination area into which the transferred data are written. This can be a symbolically addressed tag or an absolutely addressed data area with the format *P#[data block.]Operand Data type Number* (also see Chapter 4.2.3 “Absolute addressing of an operand area” on page 86). The source or destination area must be equal in length, as configured for the selected hardware object.

### **DPNRM\_DG Read diagnostic data**

DPNRM\_DG reads the diagnostic data of a DP standard slave in the format specified in EN 50 170 Volume 2, PROFIBUS. The read procedure is triggered by REQ = “1” and is finished when BUSY signals “0”. The number of read bytes is then present in the function value RET\_VAL. Depending on the slave, the diagnostic data is at least 6 bytes and a maximum of 240 bytes long. The first 240 bytes are transferred if the diagnostic data is longer and then the corresponding overflow bit is set in the data.

The LADDR parameter is supplied with the hardware ID of the addressed object. The RECORD parameter defines the area in which the read data is saved.

Note that DPNRM\_DG is a system function which operates asynchronously. It must be processed until the BUSY parameter has signal state “0”. RALRM is a system



block which makes the data available synchronously, i.e. immediately following the call.

### **DPRD\_DAT Read user data consistently**

DPRD\_DAT reads consistent user data from a DP standard slave or IO device. A CPU 1200 supports 64 bytes of consistent data. Larger areas must be read with DPRD\_DAT. This is optional for smaller areas. Areas read with DPRD\_DAT must be removed from the cyclic process image update.

The LADDR parameter is supplied with the hardware ID of the addressed object. The RECORD parameter defines the area in which the read data is saved.

### **DPWR\_DAT Write user data consistently**

DPWR\_DAT writes consistent user data to a DP standard slave or IO device. A CPU 1200 supports 64 bytes of consistent data. Larger areas must be written with DPWR\_DAT. This is optional for smaller areas. Areas written with DPRW\_DAT must be removed from the cyclic process image update.

The LADDR parameter is supplied with the hardware ID of the addressed object. The RECORD parameter defines the area from which the transferred data is read.

### **RDREC Read data record**

RDREC reads a data record from a module. The ID parameter is supplied with the hardware ID of the addressed object. The RECORD parameter defines the area to which the transferred data is written.

With signal state “1” at the REQ parameter, RDREC reads the data record INDEX from the module and saves it in the destination area RECORD. The MLEN parameter specifies how many bytes are to be read. The assignment of INDEX and MLEN is described in the manual of the respective module.

The transmission can be divided between several program cycles; the BUSY parameter has signal state “1” during the transmission.

Signal state “1” in the VALID parameter signals that the data record has been read without errors. The LEN parameter then indicates the number of transferred bytes. In the event of an error, ERROR is set to “1”. Error information is then written to the STATUS parameter.

### **WRREC Write data record**

WRREC writes a data record to a module. The ID parameter is supplied with the hardware ID of the addressed object. The RECORD parameter defines the area from which the data to be transferred is read.

With signal state “1” at the REQ parameter, WRREC writes the data record INDEX from the source area RECORD to the module. The LEN parameter specifies how

many bytes are to be written. The assignment of INDEX and LEN is described in the manual of the respective module.

The transmission can be divided between several program cycles; the BUSY parameter has signal state “1” during the transmission.

Signal state “1” in the DONE parameter signals that the data record has been written without errors. In the event of an error, ERROR is set to “1”. Error information is then written to the STATUS parameter.

### **RALRM Read additional interrupt information**

RALRM reads additional alarm information from an alarm-triggering hardware object on PROFINET IO or PROFIBUS DP, for example, from an IO device or a DP slave. It is called in the diagnostic interrupt organization block OB 82 or in a block called within that block. Processing of RALRM is synchronous, i.e. the requested data is available at the output parameters immediately following the call.

The parameter F\_ID is supplied with the hardware ID of the addressed object. The assignment of the MODE parameter determines the mode of the system block RALRM. With Mode = 0, RALRM shows you the interrupt-triggering component in the ID parameter; signal state “1” is assigned to NEW. With Mode = 1, all output parameters are written. With Mode = 2, RALRM checks whether the component specified by the F\_ID parameter was the interrupt-triggering one. If this applies, the NEW parameter has signal state “1” and all other output parameters are written.

In bytes 0 to 19, the destination area TINFO (task information) contains the complete start information of the organization block in which RALRM was called, independent of the nesting depth in which it was called. Bytes 20 and 21 are filled with the address of the hardware object. Bytes 22 to 31 contain management information, e.g. the ID number of the PROFINET IO device or the PROFIBUS DP slave.

The destination area AINFO (alarm information) contains the header information and additional interrupt information. The header information for PROFIBUS DP occupies bytes 0 to 3 and, for PROFINET IO, bytes 0 to 25 and contains, for example, the length of the received additional alarm information and the alarm type. The additional alarm information occupies bytes 4 to 223 (for PROFIBUS DP) and bytes 26 to 1431 (for PROFINET IO) and is dependent upon the alarm type.

## **14.4 Actuator/sensor interface**

### **14.4.1 Components of actuator/sensor interface**

The actuator/sensor interface (AS-i) is an industrial fieldbus system for the lowest process level in automation plants in accordance with the open international standard EN 50295. An AS-i master controls up to 62 AS-i slaves via a 2-wire AS-i cable that transfers both the control signals and the supply voltage. Examples of modules with AS-i master are the S7-1200 communication module CM 1243-2, the

DP/AS-i Link 20E and DP/AS-i Link Advanced modules for linking to PROFIBUS DP, and the IE/AS-i Link PN IO module for linking to PROFINET IO.

Configuration of an AS-i bus system with STEP 7 V11 Basic is possible with the CM 1243-2 communication module and the AS-i slaves included in the hardware catalog.

Fig. 14.9 shows the components of an AS-Interface bus system. The AS-i master is positioned to the left, next to the CPU. If you use a DCM 1271 data decoupling module, a standard power supply unit can be used for the power supply of the AS-i cable. The data decoupling unit and power supply are not configured in the hardware configuration.

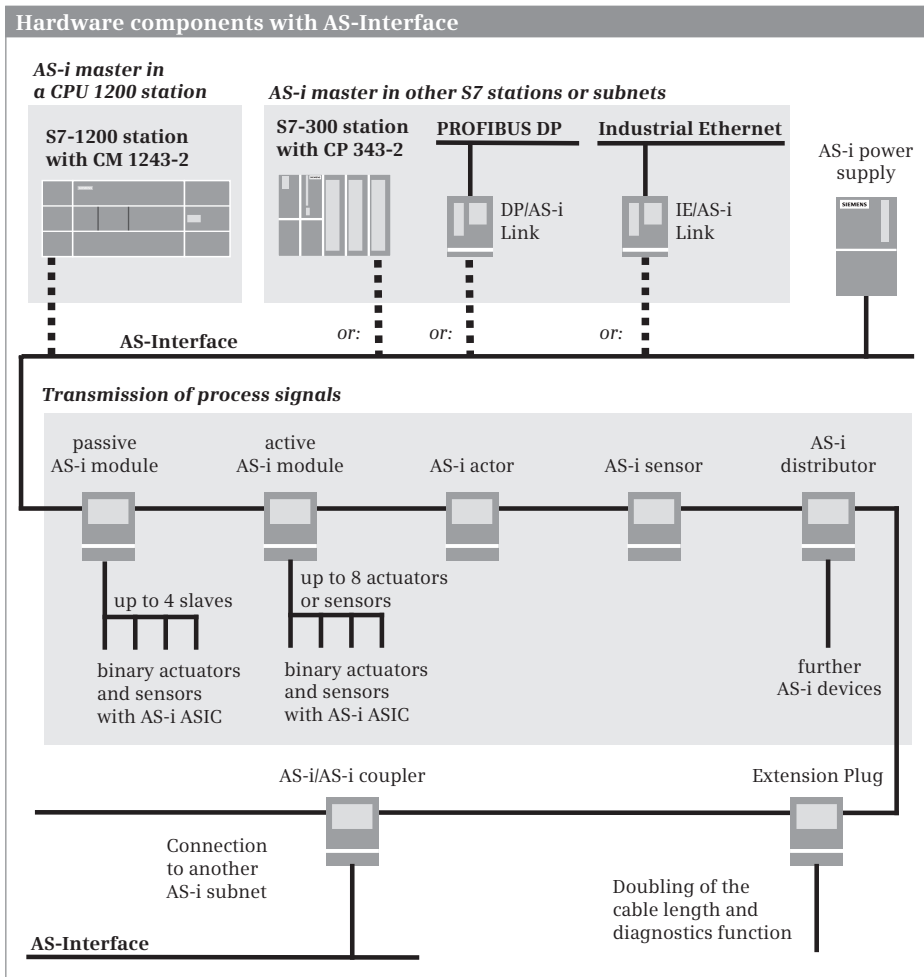


Fig. 14.9 Components for AS-Interface

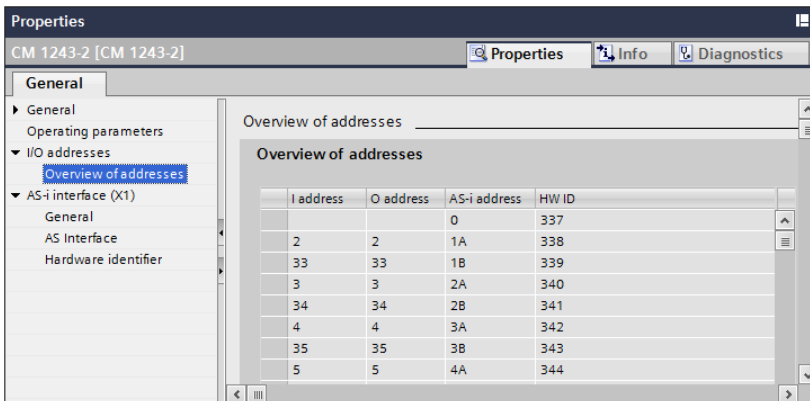
### 14.4.2 Configuring an AS-i master CM 1243-2

The CM 1243-2 communication module is not included in the hardware catalog of STEP 7 V11 on delivery. It must be integrated with a hardware support package (HSP) (see Section “Expanding the hardware catalog” on page 60).

Open the project, select the station, and start the hardware configuration in the device view. You can find the AS-i master – after installing the corresponding hardware support package – in the hardware catalog under ... > *Communication module* > *AS Interface* > *CM 1243-2* > ... . Double-click on the module and drag it with the mouse to the station in the working window. The module is positioned to the left, next to the CPU.

#### Basic configuration

No AS-i slaves are configured for the “Basic configuration” of the AS-i master. The hardware configuration reserves one byte of inputs and one byte of outputs as the “system default” for each possible AS-i slave. The start address of these 62-byte long blocks in the address area can be set in the properties of the module under *I/O addresses*. The *Overview of addresses* shows the assignment of the AS-i slave addresses to the logical addresses of the station (Fig. 14.10).



The screenshot shows the 'Properties' window for the 'CM 1243-2 [CM 1243-2]' module. The 'General' tab is selected, and the 'I/O addresses' section is expanded to show the 'Overview of addresses' sub-tab. The table below displays the mapping of I/O addresses to AS-i addresses and hardware IDs.

I address	O address	AS-i address	HW ID
		0	337
2	2	1A	338
33	33	1B	339
3	3	2A	340
34	34	2B	341
4	4	3A	342
35	35	3B	343
5	5	4A	344

**Fig. 14.10** Assigning the AS-i addresses to the I/O addresses with system default

You can configure the assignment to an AS-i subnet in the module properties under *AS-i interface* > *AS Interface*.

For the basic configuration, the configuration data of the AS-i slaves is imported from an already configured AS-i system. Load the configuration data into the S7-1200 station and switch the AS-i master into configuration mode. In the *Online* > *Diagnostics* window, the *ACTUAL* -> *PLANNED* button imports the slave configuration into the AS-i master.

### 14.4.3 Configuring an AS-Interface

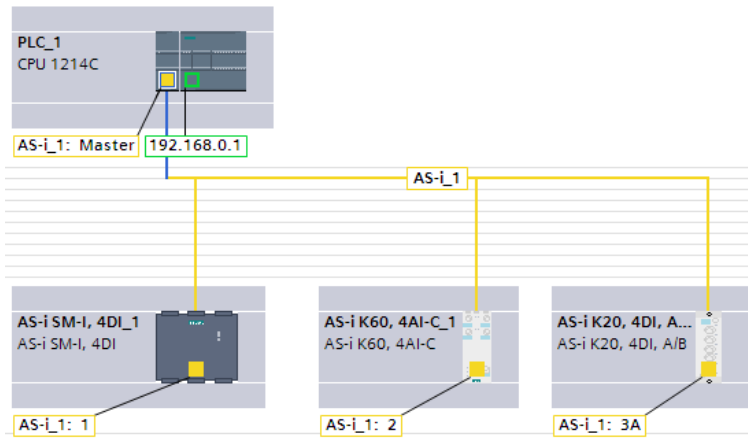
As an alternative to the “Basic configuration”, in which the AS-i master loads the slave configuration from an already configured AS-i system, you can also configure the AS-i system with the slaves in the hardware configuration.

After you have placed a CM 1243-2 communication module in the device view, switch to the network view. If you have not already done so, insert an AS-i subnet: Select the AS-i Interface of the master shown in yellow and then select the *Add subnet* command from the shortcut menu.

#### Configuring an AS-i slave

Drag the desired AS-i slave with the mouse from the hardware catalog under *Field devices > AS-Interface slaves > ...* to the AS-i subnet. STEP 7 assigns the AS-i address and the I/O address in the order of the configuration. AS-i slaves from Siemens can be selected directly from the hardware catalog. If you use AS-i slaves from other manufacturers, use the universal AS-i slave and configure it with the corresponding data.

The result is networking of the AS-i master with the AS-i slaves (Fig. 14.11).



**Fig. 14.11** Example of representation of an AS-i bus system

You set the properties of an AS-i slave in the device view. You can change the AS-i address under *AS Interface* in the module properties of an AS-i slave. Under *I/O addresses*, set the start address in the area of the inputs or outputs and define whether the user data is to be automatically updated in the process image.

The properties of a universal slave are displayed after an initial compilation of the hardware. Now you can also set the profile and the parameters under *Options*.

#### 14.4.4 Interface to user program

##### Access to AS-i slaves with binary values

You address the bits of an AS-i slave with binary scan or save functions. If you have activated the process image update in the configuration, the data is located in the area of the inputs and outputs. If the process image update is deactivated, you must access the slave data via the operand area Peripherals.

##### Access to AS-i slaves with analog values

If you have configured the AS-i slaves in the hardware configuration, you can access the analog values via the set I/O addresses. If the process image update has been activated in the configuration, the data is located in the area of the inputs and outputs. If the process image update is deactivated, you must access the slave data via the operand area Peripherals. As 16-bit value in a two's complement, the analog value occupies two byte addresses.

If the slave configuration has been imported from an already configured AS-i system, access the analog values with data records.

##### Access over the data record interface

To do this, use the system function RDREC for reading and WRREC for writing. RDREC and WRREC are described in the Chapter 14.3.4 “System functions for PROFINET IO and PROFIBUS DP” on page 470.

Calling up this function makes all of the functions of the M4 master profile of the AS-i master specification available. In the device manual of the CM 1243-2 communication module, you can see which functions these are and how the parameters of RDREC and WRREC must be assigned.

## 14.5 Communication via Modbus

Modbus is a standard protocol for data exchange between a “central” station (master/client) and several “distributed” stations (slaves/server). With Modbus RTU, the data is transferred in binary-coded form, with Modbus TCP as TCP/IP packets. In a Modbus network, a CPU 1200 can be both a master/client station and a slave/server station.

### 14.5.1 Modbus RTU

Modbus RTU (Remote Terminal Unit) utilizes serial data transmission with CM 1241 RS232, CM 1241 RS485, or CB 1241 RS485. In the Modbus network (RTU), a master station controls the data transfer and the slave stations receive or provide the data. In both stations, the communication function MB\_COMM\_LOAD configures the connector (port) on the module or on the board. In the master station, the MB\_MASTER function controls the data transfer. In the slave station, the MB\_SLAVE function controls the data transfer (Fig. 14.12).

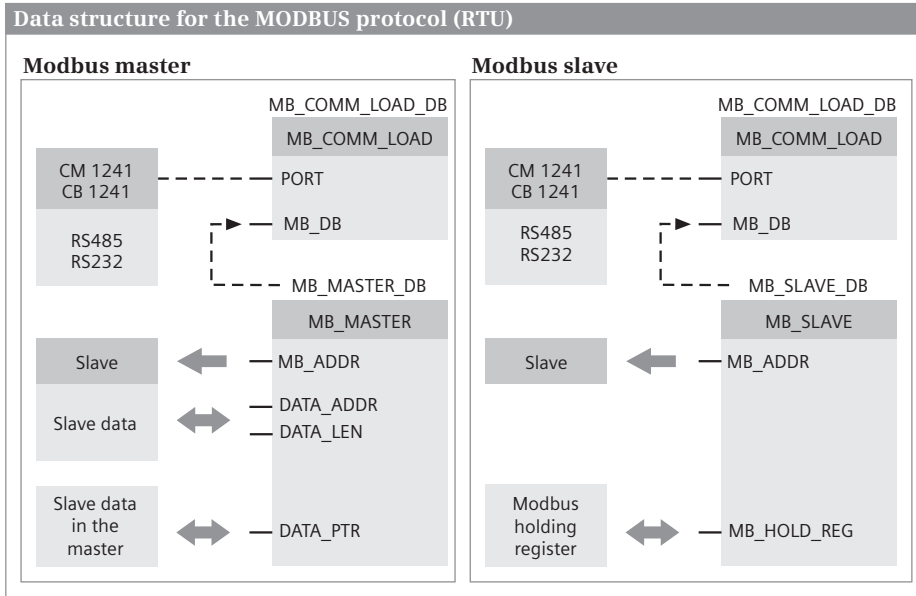


Fig. 14.12 Data structure for the Modbus protocol (RTU)

The following functions are available for Modbus RTU communication:

- ▷ MB\_COMM\_LOAD Configuration of a port for the Modbus RTU protocol
- ▷ MB\_MASTER Control for the Modbus master
- ▷ MB\_SLAVE Control for a Modbus slave

Fig. 14.13 shows the calls of the communication functions in LAD representation. STEP 7 Basic V11 provides Version 2 of the Modbus RTU functions. The functions of this version may not be used together with functions of Version 1, as provided by STEP 7 Basic V10.5.

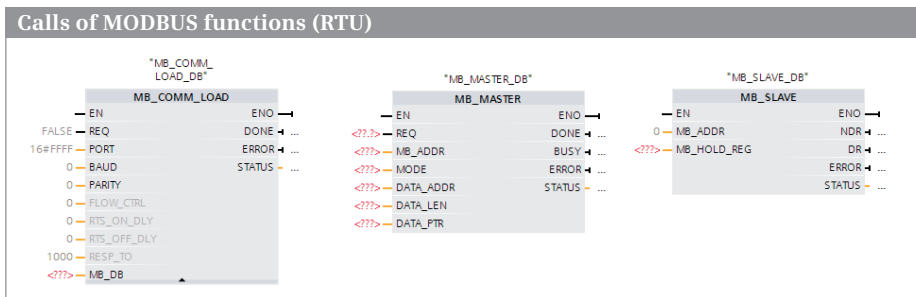


Fig. 14.13 Calling the Modbus RTU functions in LAD representation

### Parameterization of the port with MB\_COMM\_LOAD

MB\_COMM\_LOAD configures the port (the connection) at the CM module or at the communication board for the Modbus RTU protocol. Executing MB\_COMM\_LOAD is a prerequisite for using MB\_MASTER or MB\_SLAVE. MB\_COMM\_LOAD can be called up in the start-up program, for example, in order to set the port properties.

A rising edge at the REQ parameter starts a new job. A successfully executed job is indicated with the signal state “1” at the DONE parameter. If an error occurs during job processing, the ERROR parameter is set to signal state “1” and an error number is output at the STATUS parameter.

The instance data block of MB\_MASTER is created at the parameter MB\_DB or, in the slave station, the instance data block of MB\_SLAVE is created.

### Controlling data traffic with MB\_MASTER

MB\_MASTER is called as single instance in the main program. A rising edge at the REQ parameter starts a new job. While the job is running, the BUSY parameter has signal state “1”. A successfully executed job is indicated with the signal state “1” at the DONE parameter. If an error occurs during job processing, the ERROR parameter is set to signal state “1” and an error number is output at the STATUS parameter. The assigning of these control parameters is only valid for one cycle until the next processing of MB\_MASTER.

The address of the slave station is at parameter MB\_ADDR. The type of job at the slave station is defined at the MODE parameter, e.g. read inputs or write outputs. The DATA\_ADDR and DATA\_LEN parameters define the data area in the slave to be read or written.

MB\_MASTER uses the data buffer defined at the DATA\_PTR parameter as a clipboard for the data which is read from the Modbus slave or written to the Modbus slave. The data buffer can be in the bit memory address area or in a data block. The *Optimized block access* attribute must be deactivated for a data block.

### Responding master requests with MB\_SLAVE

MB\_SLAVE is called as single instance in the main program. The address of the slave station is at parameter MB\_ADDR. If the Modbus master has written data, the NDR parameter has signal state “1”. If the Modbus master has read data, the parameter DR has signal state “1”. If an error occurs during job processing, the ERROR parameter is set to signal state “1” and an error number is output at the STATUS parameter. The assigning of these control parameters is only valid for one cycle until the next processing of MB\_SLAVE.

The parameter MB\_HOLD\_REG points to the Modbus holding register, which is used by the MB\_SLAVE as a clipboard for the data that is read from the Modbus master or is written to the Modbus master. The holding register can be in the bit memory address area or in a data block. The *Optimized block access* attribute must be deactivated for a data block.



### 14.5.2 Modbus TCP

Modbus TCP (Transmission Control Protocol) uses the PROFINET interface of the CPU. In the Modbus network (TCP), a client station controls the data transfer and the server stations receive or provide the data. In the client station, the MB\_CLIENT function controls the data transfer. In the server station, the MB\_SERVER function controls the data transfer (Fig. 14.14).

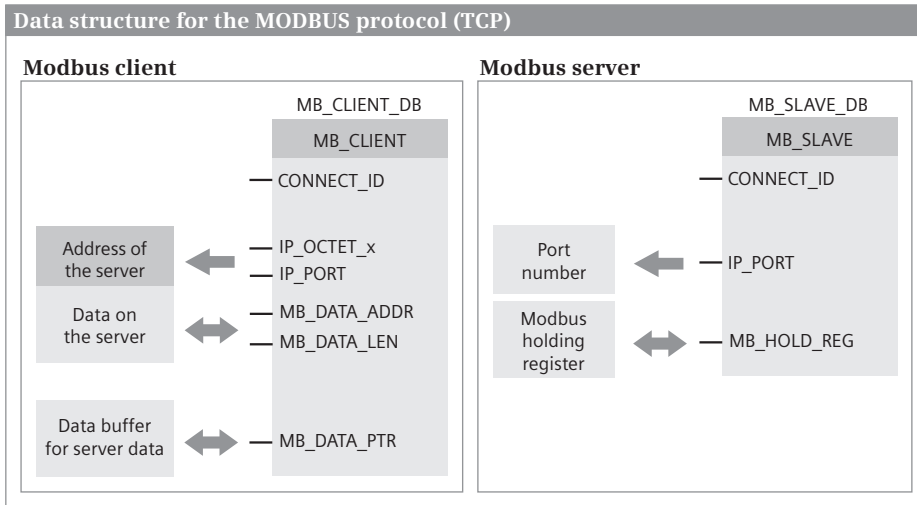


Fig. 14.14 Data structure for the Modbus protocol (TCP)

The following functions are available for Modbus-TCP communication:

- ▷ MB\_CLIENT Control for the Modbus client
- ▷ MB\_SERVER Control for a Modbus server

Fig. 14.15 shows the calls of the communication functions in LAD representation.

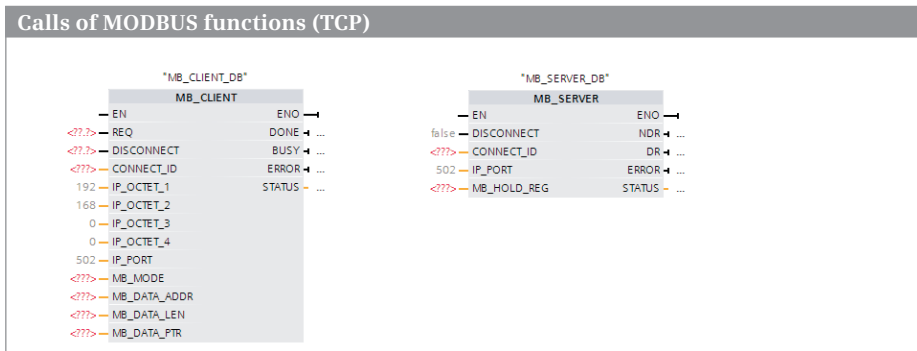


Fig. 14.15 Calling the Modbus TCP functions in LAD representation

### Controlling data traffic with MB\_CLIENT

MB\_CLIENT is called in the main program. A rising edge at the REQ parameter starts a new job. While the job is running, the BUSY parameter has signal state “1”. A successfully executed job is indicated with the signal state “1” at the DONE parameter. If an error occurs during job processing, the ERROR parameter is set to signal state “1” and an error number is output at the STATUS parameter. The assigning of these control parameters is only valid for one cycle until the next processing of MB\_CLIENT.

Modbus TCP communication requires a communication connection with the specification as per Open User Communication. The connection is established when the job is initiated if the DISCONNECT parameter has signal state “0”. Signal state “1” at the DISCONNECT parameter leads to the connection being canceled. Assigning the parameter CONNECT\_ID specifies the connection. Each connection requires its own function call (own instance data).

You can address the Modbus server for data exchange via its IP address and the port number. The four bytes of the IP address are at the parameters IP\_OCTET\_1 to IP\_OCTET\_4. The port number is at the IP\_PORT parameter.

You can define the type of job at parameter MB\_MODE, e.g. read inputs or write outputs. The start address is at parameter MB\_DATA\_ADDR. The quantity of data to be transferred is at parameter MB\_DATA\_LEN. MB\_CLIENT uses the data buffer defined at the MB\_DATA\_PTR parameter as a clipboard for the data which is read from the Modbus server or written to the Modbus server. The data buffer can be in the bit memory address area or in a data block. The *Optimized block access* attribute must be deactivated for a data block.

### Responding to client requests with MB\_SERVER

MB\_SERVER is called in the main program. It only responds to a connection request from MB\_CLIENT if the parameter DISCONNECT has the signal state “0”. The ready-to-receive state of a job can be controlled via this parameter.

Assigning the parameter CONNECT\_ID specifies the connection. Each connection needs its own function call (own instance data) if, for example, the server station communicates with several Modbus clients. Parameter IP\_PORT shows the number of the port which is monitored by MB\_SERVER for connection requests from MB\_CLIENT.

If the Modbus client has written data, the NDR parameter has signal state “1”. If the Modbus client has read data, the parameter DR has signal state “1”. If an error occurs during job processing, the ERROR parameter is set to signal state “1” and an error number is output at the STATUS parameter. The assigning of these control parameters is only valid for one cycle until the next processing of MB\_SERVER.

The parameter MB\_HOLD\_REG points to the Modbus holding register, which is used by the MB\_SERVER as a clipboard for the data that is read from the Modbus client or is written to the Modbus client. The holding register can be in the bit memory address area or in a data block. The *Optimized block access* attribute must be deactivated for a data block.

# 15 Communication

## 15.1 Overview

### Introduction to data communication with S7-1200

Communication is understood to be the data exchange between networked stations. A station is a device containing a module with communication capability, for example a programmable controller, an HMI device, or any other suitable third-party device. Communication is usually considered from the point of view of the “local” station connected with a partner station – the “remote” station. In the case of a bus system, all stations are connected together over one single line; in the case of a point-to-point connection, the connection is limited to two stations.

The physical connection on its own – the *networking* – is not sufficient for communication. A specifically defined sequence, referred to as the *protocol*, is required to exchange the data. The communication partners and the protocol are defined when establishing a *connection* (“communication connection”).

A PLC station with a CPU 1200 can exchange data with other stations using various methods. The connection is made

- ▷ for Industrial Ethernet via the PROFINET interface on the CPU
- ▷ for PROFIBUS via the CM modules CM 1243-5 as DP master and CM 1242-5 as DP slave
- ▷ for AS-Interface via the CM module CM 1243-2 as AS-i master
- ▷ for point-to-point communication via a CM module CM 1241 or the communication board CB 1241

The communication with distributed I/O, for which the relevant “master module” that controls data traffic is inserted in the PLC station, is described in Chapter 14 “Distributed I/O” on page 455.

An HMI station (of an HMI device) can be connected via Ethernet or PROFIBUS, depending on the device version. Data transmission requires a configured connection (HMI connection). Special communication functions in the user program are not required. The data exchange is defined during the configuration of the HMI station. Details can be found in Chapter 16 “Visualization” on page 507.

The communication between a PLC station and a programming device does not require any connection configuration or communication functions in the user program.

## Data transmission via Industrial Ethernet

When configuring the networking and connections, Industrial Ethernet is handled like a bus system. Strictly speaking, however, the Ethernet network consists only of single point-to-point connections. If only two stations exchange data with each other, they can be directly connected with a cable. If there are more than two stations, a connection multiplier – such as the Compact Switch Module CSM 1277 – is needed, which provides, for example, an interface with four ports. If a station has two ports connected with a switch, the bus cable can be guided to the next station via the second port. The configuration of each individual point-to-point connection is necessary to calculate runtimes and response times.

## Data transmission with open user communication

Open user communication transfers data between two PLC stations that are connected together via an Ethernet network. Data transmission can be secured with the TCP and ISO-on-TCP protocols or unsecured with the UDP protocol. TCP and ISO-on-TCP require a configured connection. The communication functions TSEND\_C for sending data and TRCV\_C for receiving data form the interface to the user program. The transmission with UDP is connectionless. The communication access point is established in both stations with the TCON communication function; it can then be used to send data (TUSEND) and receive data (TURCV).

## Data transmission with S7 communication

S7 communication allows data to be transferred between two PLC stations connected via an Ethernet network or PROFIBUS. Data transmission requires a configured connection (“S7 connection”). The communication functions in the user program are GET to read data and PUT to write data. In the remote station, the CPU operating system controls the data traffic without a communication function in the user program.

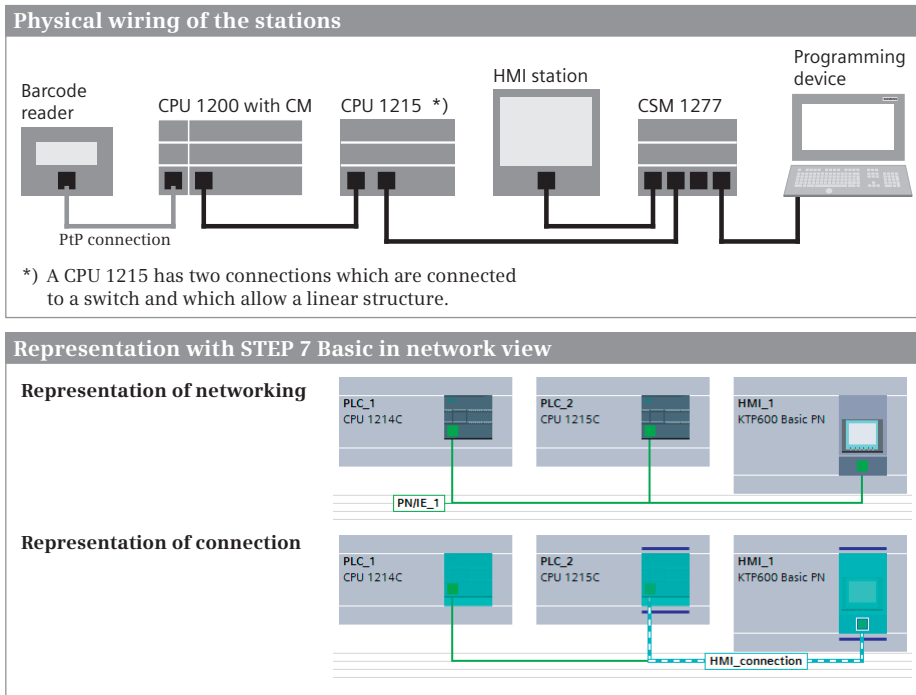
## Data transmission with a point-to-point connection

A PLC station with a CPU 1200 also supports a serial, character-based point-to-point connection. This type of communication allows a very generous definition of the protocol. For example, a printer, barcode reader or modem can be connected to a CPU 1200 via a CM 1241 communication module using the RS232 or RS485 transmission standard.

The ASCII protocol is available for point-to-point communication, and – in global libraries – the communication functions for MODBUS and USS drives.

## Configuring communication with STEP 7 Basic

The networking and the connections are configured with the hardware configuration in the network view. The connection of a programming device is not configured except for the parameterization of the PROFINET interface (IP address). Networking with a programming device and a connection multiplier as well as a point-to-point connection are not displayed in the network configuration.



**Fig. 15.1** Comparison of the wiring and configuration

In Fig. 15.1 you can see the networking (wiring) between the stations necessary for data exchange. The lower part shows the configuration with the hardware configuration: In the networking view, the networking is represented by the Ethernet subnet *PNIE\_1* and in the connection view the configured connections between stations are highlighted; in the example an *HMI\_Connection*.

Before connecting to Ethernet, the PROFINET interface of the CPU must be parameterized (see Section “IP address and subnet mask” on page 73). The connection of a programming device is described in Chapter 13.1 “Connecting a programming device to the PLC station” on page 421.

The PLC and HMI stations to be networked must be located together in one project.

## 15.2 Open user communication

### 15.2.1 Basics

Open user communication is a procedure for transmitting data between two stations connected to the Ethernet subnet. Data exchange can be implemented using the protocols TCP in accordance with RFC 793, ISO-on-TCP in accordance with RFC 1006, and UDP in accordance with RFC 768.

Data transmission with TCP and ISO-on-TCP is secure (with acknowledgment) over configured connections. Data transmission with UDP is unsecured (without acknowledgment) and connectionless.

Prerequisite for open user communication is the networking of the participating stations via an Ethernet network. The (communication) connection is not created when configuring the network with the connection table, but occurs via a data area. The communication functions at runtime use the address data in this data area to establish the connection (for the TCP and ISO-on-TCP protocols) or to set up the communication access point (for UDP).

For open user communication with TCP or ISO-on-TCP, use the communication functions TSEND\_C and TRCV\_C in the user program. These functions establish a connection, control data traffic, and – depending on the programming – end the connection. For the UDP protocol, the TCON communication function sets up the access point for the participating partner stations. The communication functions TUSEND and TURCV control the data traffic, where dynamic addressing of the remote station at runtime is possible.

### 15.2.2 Open user communication with TCP and ISO-on-TCP

#### Data structure

Fig. 15.2 shows the data structure of open user communication with the use of the TCP and ISO-on-TCP protocols.

In the transmitting station, the communication function TSEND\_C is called in the user program. At the parameter CONNECT there is a pointer to a data block that has the structure of the system data type TCON\_Param and contains the connection data. The parameter DATA points to the data to be sent. In the receiving station, the communication function TRCV\_C is called in the user program. The parameter CONNECT is supplied with a data block that contains the connection data. The parameter DATA contains a pointer to the receive mailbox in which the received data is stored.

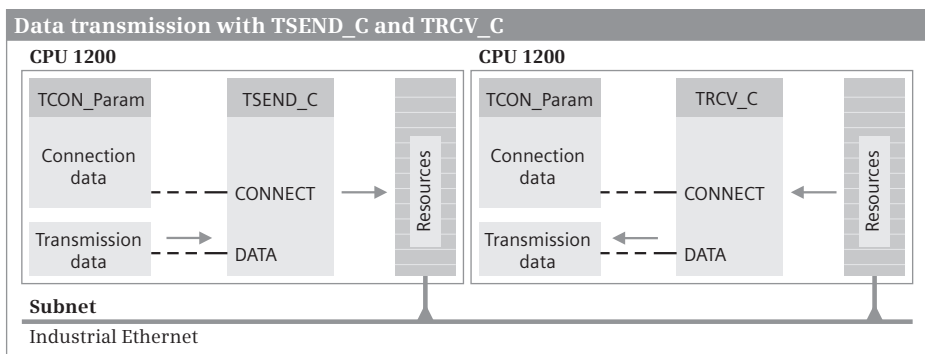
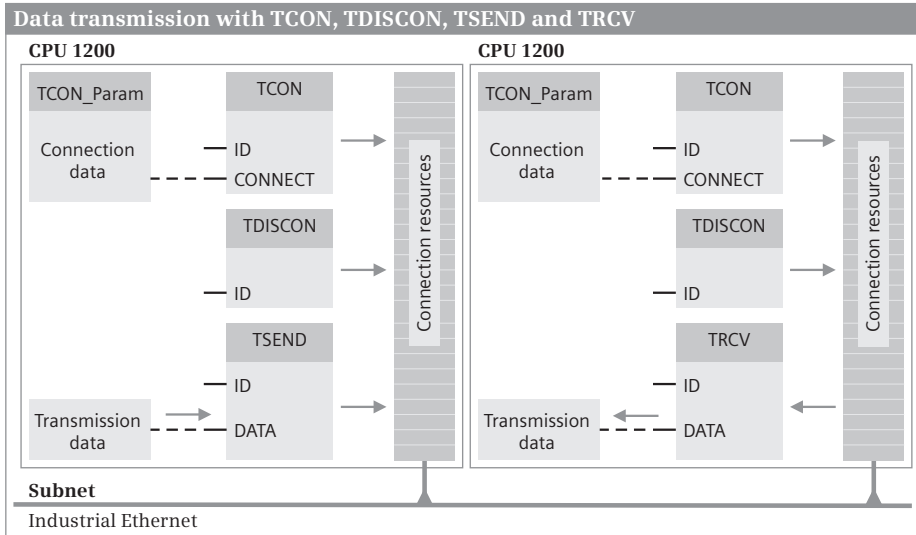


Fig. 15.2 Communication functions TSEND\_C and TRCV\_C



**Fig. 15.3** Communication functions TCON, TDISCON, TSEND, and TRCV

At runtime, the communication functions TSEND\_C and TRCV\_C establish a connection in both stations, transfer the data, and then – depending on the programming – end the connection.

As an alternative to TSEND\_C and TRCV\_C, the communications functions TCON, TDISCON, TSEND, and TRCV can be used. Fig. 15.3 shows the data structure used here.

Before a data transmission, a connection must be established in both stations with the TCON communication function. At the parameter CONNECT there is a pointer to a data block that has the structure of the system data type TCON\_Param and contains the connection data. The ID parameter identifies the communication connection and must match the *id* tag in the connection data and the ID parameters of all communication functions involved in the connection.

In the transmitting station, the communication function TSEND is called in the user program. The parameter DATA points to the data to be sent. In the receiving station, the communication function TRCV is called in the user program. The parameter DATA contains a pointer to the receive mailbox in which the received data is stored.

With the communication function TDISCON, the connection can be terminated and the connection resources released.

### Transmission Control Protocol (TCP)

The transmission control protocol (TCP) is described in RFC 793. TCP is suitable for medium to large amounts of data (up to 8192 bytes) with static (fixed) data lengths. It is capable of routing. Performance features include, for example, recovery in case of failure, flow control, and reliability.

TCP is connection-oriented. Applications are addressed by means of port numbers. TCP is used if the communication partner does not support a connection using ISO-on-TCP. For such communication partners, enter “unspecified” as the partner end point in the connection parameters.

If the number of sent bytes is smaller than the number of received bytes (controlled by the LEN parameters), the job is only signaled as being complete when the receive mailbox is full, i.e. the first bytes of the next job are also present in the receive mailbox. If the number of sent bytes is greater than the number of received bytes, the number of bytes specified in the LEN parameter are written into the receive mailbox, and the value of LEN is output in the RCVD\_LEN parameter. The next block of sent data is received with each further call.

### ISO Transport over TCP (RFC 1006)

The RFC 1006 protocol (ISO-on-TCP) can be used to import ISO applications into the TCP/IP network. It is suitable for medium to large quantities of data (up to 8192 bytes) with dynamic lengths, is capable of routing, and use in a wireless network is possible.

Multiple connections can be established to a single IP address. The unambiguous assignment of a connection (communication endpoint) to an IP address is provided by a Transport Service Access Point (TSAP). You can define the TSAP for the connection in the inspector window in the *Configuration* tab under *Connection parameters*.

If the number of sent bytes is smaller than the number of received bytes (controlled by the LEN parameters of both functions), the sent bytes are written into the receive mailbox, the job is signaled as being complete, and the RCVD\_LEN parameter contains the number of received bytes. If the number of sent bytes is greater than the number of received bytes, no data is written into the receive mailbox, and the error *Destination buffer too small* (status = 16#8088) is signaled.

### 15.2.3 Open user communication with the UDP protocol

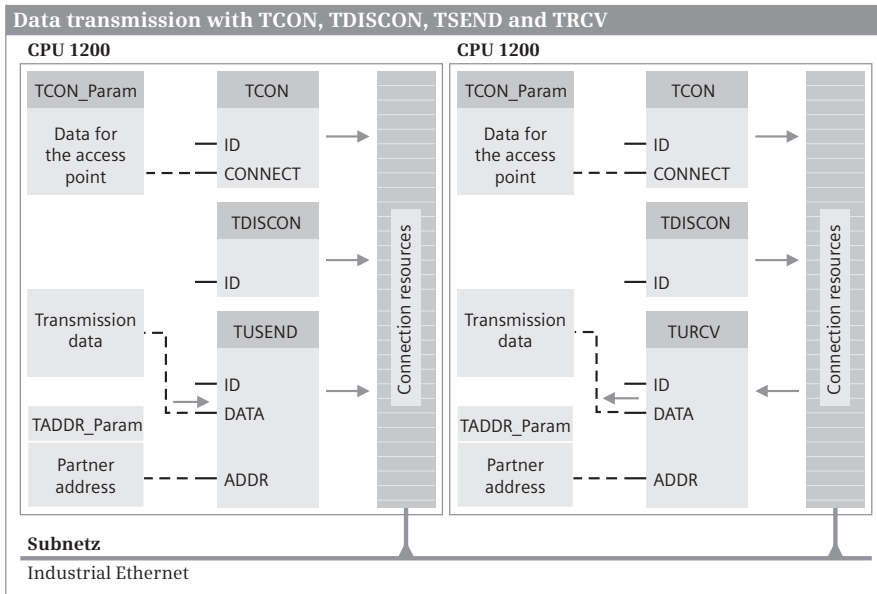
#### Data structure

Fig. 15.4 shows the data structure of the open user communication with the use of the User Datagram Protocol (UDP).

In the transmitting station, the TCON communication function is called in the user program; it sets up the communication access point in the CPU operating system. The ID parameter identifies the connection between the user program and the CPU operating system. The specification must match the ID parameter of the communication function TUSEND and the *id* tag in the connection data at the parameter CONNECT. At the parameter CONNECT there is a pointer to a data block that has the structure of the system data type TCON\_Param and contains the connection data.

The communication function TUSEND sends the data specified at the DATA parameter to the partner station. At the ADDR parameter there is a pointer to a data area with the structure of the system data type TADDR\_Param, which contains the IP address and the port number of the partner station.





**Fig. 15.4** Communication functions TUSEND and TURCV

In the receiving station, the communication function TCON is also called in the user program. The ID parameter identifies the connection between the user program and the CPU operating system. The specification must match the ID parameter of the communication function TURCV and the *id* tag in the connection data at the parameter CONNECT. At the parameter CONNECT there is a pointer to a data block that has the structure of the system data type TCON\_Param and contains the connection data.

TURCV receives data from the transmitting station and writes it to the receive mailbox specified at the DATA parameter. At the ADDR parameter there is a pointer to a data area with the structure of the system data type TADDR\_Param, which contains the IP address and the port number of the partner station.

After data transmission, the communication function TDISCON disconnects the communication connection specified at the ID parameter or dissolves the communication access point and frees the resources.

### User datagram protocol (UDP)

Data transmission with UDP is described in RFC 768. The protocol is suitable for transmitting small to medium amounts of data (up to 2048 bytes) quickly, because it is close to the hardware. It is capable of routing. The delivery of the data is unsecured and there is no feedback about its receipt. The communication partner is addressed as connectionless via the IP address and a port.

The communication functions TUSEND and TURCV handle the interface to the user program. The communication access point must first be set up with TCON in both the local and the remote station.

### 15.2.4 Communication functions for open user communication

The following communication functions are available for open user communication with the protocols TCP, ISO-on-TCP, and UDP:

- ▷ TCON Establish a connection to the partner station or to the operating system
- ▷ TDISCON Disconnect
- ▷ TSEND Send data with TCP or ISO-on-TCP
- ▷ TRCV Receive data with TCP or ISO-on-TCP
- ▷ TSEND\_C Establish connection and send data with TCP or ISO-on-TCP (replaces TCON, TDISCON, and TSEND)
- ▷ TRCV\_C Establish connection and receive data with TCP or ISO-on-TCP (replaces TCON, TDISCON, and TRCV)
- ▷ TUSEND Send data with UDP
- ▷ TURCV Receive data with UDP

Fig. 15.5 shows the calls of the functions in ladder logic representation.

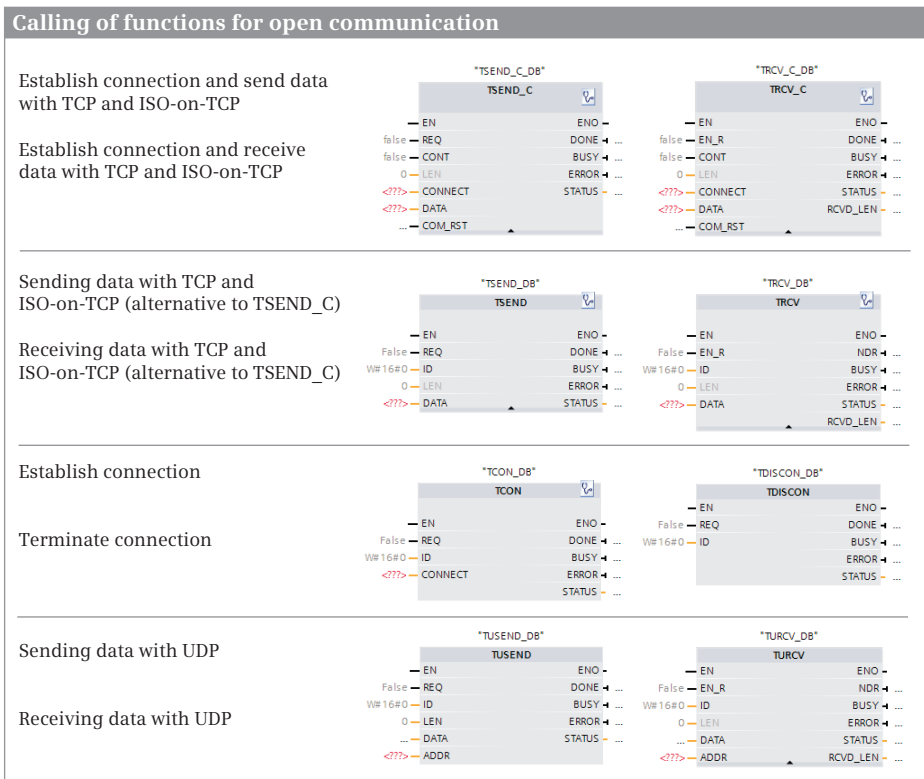


Fig. 15.5 Calls of the communication functions for open user communication (LAD)

## Description of common parameters

The communication functions for open user communication work asynchronously, i.e. job processing may require several program cycles under certain circumstances. Use the REQ and EN\_R parameters to control the data transmission. The status of data transmission is shown by the BUSY, NDR, DONE, ERROR and STATUS parameters. You must evaluate these parameters immediately following each processing of the communication function since they are only valid up to the next call.

The *CONT* parameter with signal state “1” establishes a connection and retains it. Data transmission – the sending or receiving of data – only functions with the connection established. Signal state “0” on this parameter cancels the connection again.

If a rising signal edge occurs at parameter REQ, the task to send data starts. As long as the signal state is “1”, no other task for the specified connection is accepted. Only when the communication function recognizes signal state “0” at REQ, can a new task be started with the change to “1”.

The *EN\_R* parameter activates data reception with signal state “1”. Data receipt is blocked if EN\_R has the signal state “0”.

The *DONE* parameter shows with signal state “1” that the started job has been completed without errors. It is only set for the duration of one program cycle.

The *NDR* parameter (New Data Ready) shows with signal state “1” that the started job has been completed without errors and that new data has been received. It is only set for the duration of one program cycle.

The *BUSY* parameter shows with signal state “1” that job processing has not yet been completed and that a new job cannot be started.

The *ERROR* parameter shows with signal state “1” that the started job has been completed with an error. It is only set for the duration of one program cycle.

The *STATUS* parameter contains intermediate states or error information.

- ▷ If DONE or NDR = “1”, STATUS is occupied by 16#0000.
- ▷ If ERROR = “1”, STATUS is occupied by error information.
- ▷ If none of these bits is set, STATUS may contain intermediate states which indicate the progress of the started job.

The following parameters specify the data transmission:

The ID parameter identifies the communication connection between the transmitting and receiving station, or for UDP the connection between the user program and the communication access point in the CPU operating system.

The CONNECT parameter points to a data block with the data structure of the system data type TCON\_Param, which contains the description of the connection parameters.

The DATA parameter contains the send or receive mailbox for the transferred data. The send and receive mailbox is a tag, e.g. an ARRAY tag in a data block, or an abso-

lutely addressed range, e.g. a range of 32 bytes in the data block DB 10 starting at byte 64: P#DB10.DBX64.0 BYTE 32.

The ADDR parameter contains the IP address and the port number of the partner station with the data structure of the system data type ADDR\_Param.

### **TSEND\_C Establish connection and send data**

The communication function TSEND\_C establishes a connection, and sends data with the TCP or ISO-on-TCP protocol. TSEND\_C can also cancel the connection again.

Establishment of the connection is started if TSEND\_C is processed with CONT = "1". Following successful establishment, the DONE parameter is set to "1" for the duration of one program cycle. Data is sent via an established connection when a rising edge occurs at the REQ parameter. Following successful transmission, DONE is set to "1" for the duration of one program cycle.

With CONT = "0" the connection is canceled immediately in both the send station and the partner station.

If the COM\_RST parameter has signal state "1", TSEND\_C is started again: an existing connection is canceled, and a new one established.

The LEN parameter specifies the maximum number of bytes which are sent.

### **TRCV\_C Establishing a connection and receiving data**

The communication function TRCV\_C establishes a connection, and receives data with the TCP or ISO-on-TCP protocol. TRCV\_C can also cancel the connection again.

Establishment of the connection is started if TRCV\_C is processed with CONT = "1". Following successful establishment, the DONE parameter is set to "1" for the duration of one program cycle. TRCV\_C only receives data when the EN\_R parameter has signal state "1". Following successful receipt of data, the DONE parameter is set to "1" for the duration of one program cycle. Data is received continuously if CONT = "1" and EN\_R = "1".

With CONT = "0" the connection is canceled immediately in both the send station and the partner station.

If the COM\_RST parameter has signal state "1", TRCV\_C is started again: an existing connection is canceled, and a new one established.

The parameter LEN specifies the number of bytes received. If LEN is filled with zero, as many bytes are received as contained in the tag at the DATA parameter. After successful data transmission, the parameter RCVD\_LEN contains the number of bytes actually transmitted.

**Data transmission with TCON, TDISCON, TSEND and TRCV**

TCON establishes a communication connection to the partner device. The ID parameter identifies the connection. The specification must correspond to the *id* tag in the connection data at the parameter CONNECT.

The *TSEND function* sends data with the TCP or ISO-on-TCP protocol over an existing communication connection specified by the ID parameter. The DATA parameter contains a pointer to the send mailbox. You can use the LEN parameter to specify the maximum number of bytes to be sent.

The *TRCV function* receives data with the TCP or ISO-on-TCP protocol over an existing communication connection specified by the ID parameter. The DATA parameter contains a pointer to the receive mailbox. You can use the LEN parameter to specify the number of bytes to be received. If LEN is "0", the number of bytes received corresponds to the tag at the DATA parameter. Following successful data transmission, the RCVD\_LEN parameter contains the number of actually transmitted bytes.

TDISCON terminates the communication connection specified at the ID parameter and releases the connection resources.

**TUSEND Send data with the UDP protocol**

The TUSEND communication function sends data with the UDP protocol. A prerequisite is the previous establishment of a communication access point by TCON.

The assignment of the ID parameter designates the connection between the user program and the communication access point of the operating system. The value must agree with the *id* tag in the connection data. Specify the send mailbox with a pointer at the DATA parameter.

The information on the communication partner is located in a data area to which the pointer at the ADDR parameter points. With each new send job, the address and thus the partner can be changed without having to redefine the communication access point with TCON.

In the initial state, the REQ, BUSY, DONE, and ERROR parameters have signal state "0". Start data transfer with a rising edge at the REQ parameter. On the initial call with "1", the data is fetched from the area specified by the DATA parameter. The number of bytes specified at the LEN parameter is sent (1 to max. 1460).

While the job is running, BUSY = "1". The job has been successfully completed if BUSY = "0", DONE = "1", and ERROR = "0". If the job contains errors, then BUSY = "0", DONE = "0", and ERROR = "1". The error is then specified at the STATUS parameter. BUSY, DONE, and ERROR are reset to "0" if REQ is returned to "0".

The data in the send area can then be modified again when either DONE or ERROR has signal state "1".

### TURCV Receive data with the UDP protocol

The communication function TURCV receives data with the UDP protocol. A prerequisite is the previous establishment of a communication access point by TCON.

The assignment of the ID parameter designates the connection between the user program and the communication access point of the operating system. The value must agree with the *id* tag in the connection data. Specify the receive mailbox with a pointer at the DATA parameter.

The information on the communication partner is located in a data area to which the pointer at the ADDR parameter points. The number of bytes to be received is set at the LEN parameter (1 to max. 1460). After a data block has been received, the number of bytes received is made available at the RCVD\_LEN parameter and NDR is set to signal state “1”. Data is only received if the EN\_R parameter has signal state “1”.

While the job is running, BUSY = “1”. The job has been successfully completed if BUSY = “0”, NDR = “1”, and ERROR = “0”. If the job contains errors, BUSY = “0”, NDR = “0”, and ERROR = “1”. The error is then specified at the STATUS parameter. BUSY, NDR, and ERROR are reset to “0” if EN\_R is returned to “0”.

The data in the receive area is consistent if NDR has signal state “1”.

#### 15.2.5 Configuring open user communication

Prerequisite for open user communication is the networking of the PLC stations participating in the data exchange via Industrial Ethernet. The required procedure is described in Chapter 3.4 “Configuring the network” on page 67. No connection is configured in the connection table for open user communication.

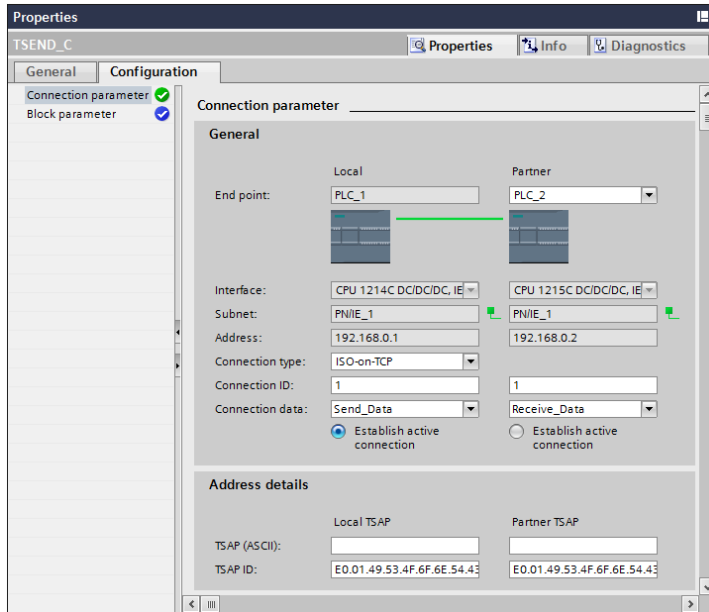
The communication functions for open user communication are called in the main program, for example in a function block that is called in the organization block OB 1 or another organization block with the event class *Program cycle*.

The communication functions can be found in the program elements catalog in the folder *Communication > Open user communication*. Drag the desired communication function into the opened block. When you release the mouse button, you will be prompted to specify the call option. Select the *Single instance* option; a separate data block is then assigned to the call.

In the inspector window under *Properties* in the *Configuration* tab, the program editor shows the block parameters and also the connection parameters for the communication functions TCON, TSEND\_C, and TRCV\_C (Fig. 15.6).

Fill out the fields highlighted in red that are still empty. If entries remain open, for example because the data areas still have to be created in the partner station, you will later be shown the connection dialog again if the communication function is selected.

In the connection dialog, you specify the partner device, the connection type (*TCP* or *ISO-on-TCP*, for TCON also *UDP*), and the connection ID. The connection ID iden-



**Fig. 15.6** Configuring the connection parameters for open user communication

ties the connection and must be the same in both stations. (Multiple connections between the partners can be created.)

In the *Connection data* field, specify the name of a data block, which is then created by the program editor and parameterized at the `CONNECT` parameter. If you program the communication function appropriate to the connection in parallel in the partner device, a data block with the connection data is also created here. Enter this data block in the *Connection data* field under *Partner*. The data block with the connection data is a type data block with the structure of the system data type `TCON_Param`.

Use the *Establish active connection* option to specify which of the stations is to initiate the connection.

For the *TCP* and *UDP* connection types, enter the port number of the partner under *Address details*. The number 2000 (dec) is specified by default. If you create multiple connections, assign each connection its own port number (in the range of 2000 to 5000 decimal).

For the connection type *ISO-on-TCP*, enter the access points (TSAP). The `TSAP_ID` must be unique in a station. If you create multiple connections, assign each connection its own TSAP ID. A TSAP has a length of 2 to 16 bytes.

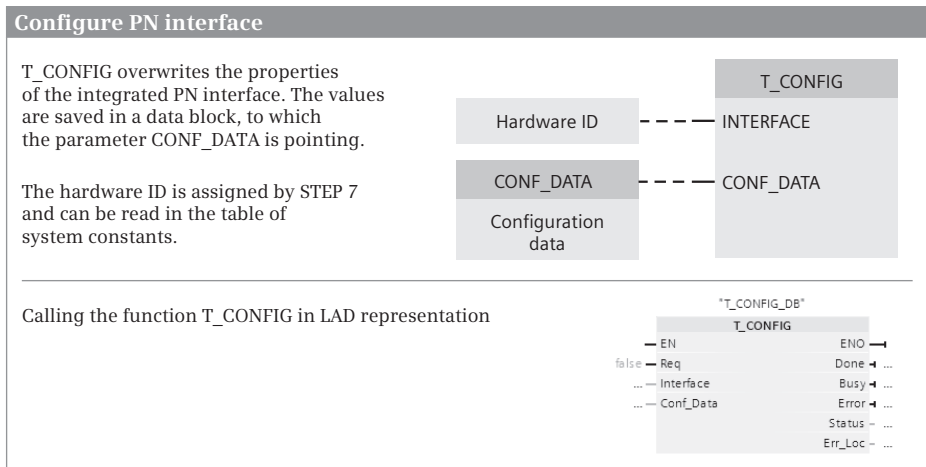
By default, when configuring the connection editor, the TSAP “E0.01.49.53.4F.6F.6E.54.43.50.2D.31” is assigned. The first byte “E0” stands for open user communication. “01” specifies the module (rack = 0, slot = 1). The next bytes are the ASCII characters “ISOonTCP-1”.

If you enter a new TSAP, first enter the character sequence in the field *TSAP (ASCII)*. You then add the characters “E0.01.” before it in the *TSAP ID* field. In the *TSAP (ASCII)* field, the TSAP is no longer displayed because the first character (E0) is not an ASCII character.

You then complete the fields for the block parameters in the inspector window in the *Configuration* tab in the *Block parameters* group.

### 15.2.6 Configuring a PN interface with T\_CONFIG

T\_CONFIG configures the integral PROFINET interface of the CPU. A prerequisite is that the *Set PROFINET device name using a different method* option was set during parameterization of the PROFINET interface with the hardware configuration when assigning the IP parameters. Fig. 15.7 shows the data structure used and the function call in LAD representation.



**Fig. 15.7** Setting a PN interface with T\_CONFIG

The adjustable parameters are the IP address, subnet mask, and router address. If the station is an I/O device, the PROFINET device name can also be changed.

IP\_CONF works asynchronously, i.e. processing of a job can extend over several program cycles. The job is started with signal state “1” at the REQ parameter. The REQ parameter must remain “1” for as long as the BUSY parameter has signal state “1”. The job has been completed if BUSY = “0”. The DONE parameter indicates with signal state “1” that the job has been completed without errors. In the event of an error, ERROR has signal state “1”. The STATUS parameter provides information on errors which have occurred and the ERR\_LOC parameter identifies the source.

You specify the hardware ID of the PROFINET interface at the INTERFACE parameter. STEP 7 specifies the hardware ID during configuration and lists it in the *System con-*



starts table of the default tag table with the data type HW\_INTERFACE. You can reach the default tag table in the project tree under the station in the *PLC tags* folder. The CONF\_DB parameter is a pointer to the configuration data.

## 15.3 S7 communication

### 15.3.1 Basics

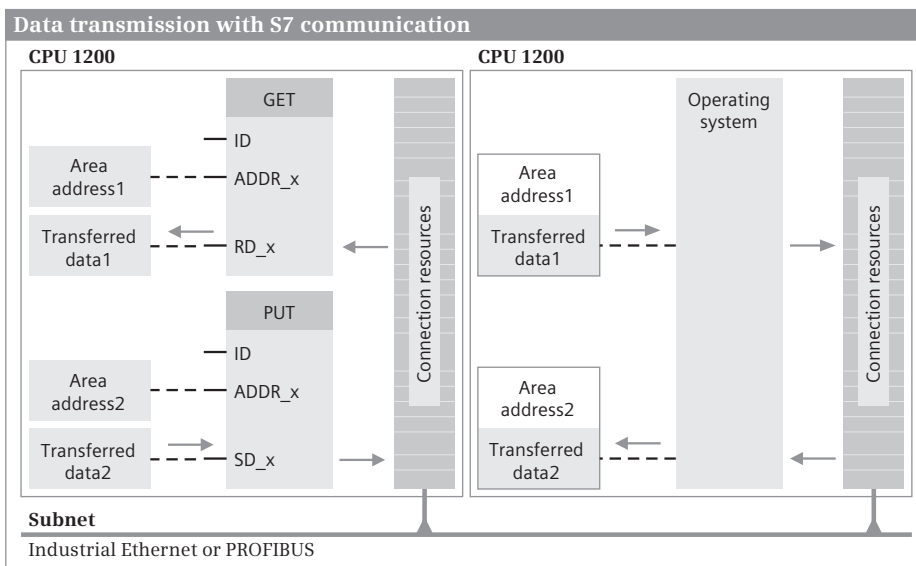
Data can be exchanged with other S7 stations using S7 communication via PROFINET or PROFIBUS. The communication connections are configured in the connection table.

In the local station, communication functions control the data traffic: GET requests data from the remote station, PUT writes data to the remote station. No communication functions are required in the user program in the remote station (partner station). The operating system of the CPU performs the necessary work (“one-way data exchange”).

### 15.3.2 Data structure for one-way data exchange

Fig. 15.8 shows the data structure of S7 communication with one-way data exchange.

In the user program in the local station, the communication function GET is called if data should be read from the remote station, and the communication function PUT is called if data should be written to the remote station. In the remote station, the operating system controls the data traffic.



**Fig. 15.8** Data structure for the communication functions GET and PUT

At the parameters ADDR\_1 to ADDR\_4, you define the data areas in the remote station with absolute addressing. With the communication function GET you define the receive mailboxes for the read data at the parameters RD\_1 to RD\_4. With the communication function PUT, you define the send mailboxes for the data to be written at the parameters SD\_1 to SD\_4.

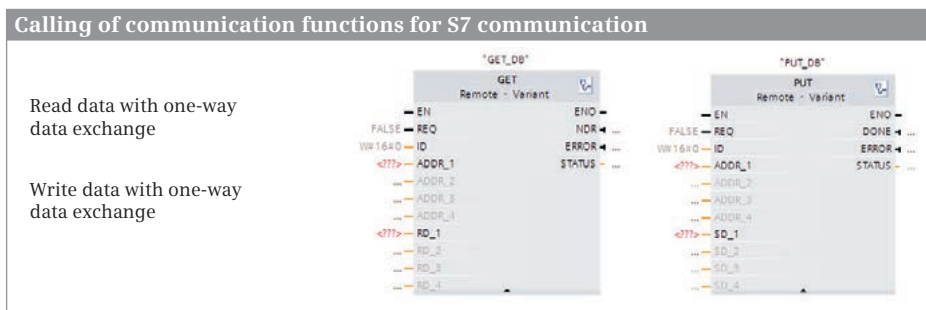
No user program for sending or receiving is required in the remote station. The remote station can transmit the data in the operating modes STOP and RUN.

### 15.3.3 Communication functions for one-way data exchange

The following blocks are available for one-way data exchange:

- ▷ GET Read data from a partner CPU
- ▷ PUT Write data to a partner CPU

GET and PUT have four send and receive areas that are shown when the representation is opened. Fig. 15.9 shows the calls of the functions in ladder logic representation.



**Fig. 15.9** Calls of the communication functions for S7 communication (LAD)

#### Description of parameters

At the ID parameter, you specify the connection over which the data traffic is to be implemented. Here you specify the connection ID from the connection table. The connection ID can be changed during runtime so that an instance data block can be used for several connections.

At the parameters ADDR\_1 to ADDR\_4 with the REMOTE data type, you define the data areas in the remote station with absolute addressing. Example: The pointer P#DB24.DBX0.0 BYTE 32 defines an area of 32 bytes in data block DB 24 beginning with data byte DB 0 (see also Chapter 4.2.3 “Absolute addressing of an operand area” on page 86).

With the communication function GET you define the receive mailboxes for the read data at the parameters RD\_1 to RD\_4: RD\_1 for the data from the field ADDR\_1, etc. With the communication function PUT, you define the send mailboxes for the data to be written at parameters SD\_1 to SD\_4, with SD\_1 for the area ADDR\_1, etc.

Not all ADDR\_x, RD\_x, or SD\_x parameters must be assigned. This assignment begins with the first parameter and must be continuous. The data length at parameter ADDR\_x must agree with the data length of the associated tags at RD\_x or SD\_x. The data to be transferred can be absolutely or symbolically addressed tags or absolutely addressed operand areas. The tags in a data block with activated *Optimized block access* attribute can only be accessed with symbolic addressing.

The REQ parameter starts data transmission. Note: On the first call, the REQ parameter must have signal state “0” (FALSE). To start a transmission job, a rising edge is required.

The function signals with signal state “1” at the DONE or NDR parameters that the job has been completed without errors. Any errors are signaled by “1” at the ERROR parameter. The STATUS parameter shows with an assignment which is not zero either a warning (ERROR = “0”) or an error (ERROR = “1”). You must evaluate the DONE, NDR, ERROR, and STATUS parameters after every function call.

### 15.3.4 Configuring S7 communication

Prerequisite for S7 communication is the networking of the PLC stations participating in the data exchange via Industrial Ethernet or PROFIBUS. The required procedure is described in Chapter 3.4 “Configuring the network” on page 67. For S7 communication, an S7 connection must be configured in the connection table.

The communication functions for S7 communication are called in the main program, for example in a function block that is called in the organization block OB 1 or another organization block with the event class *Program cycle*.

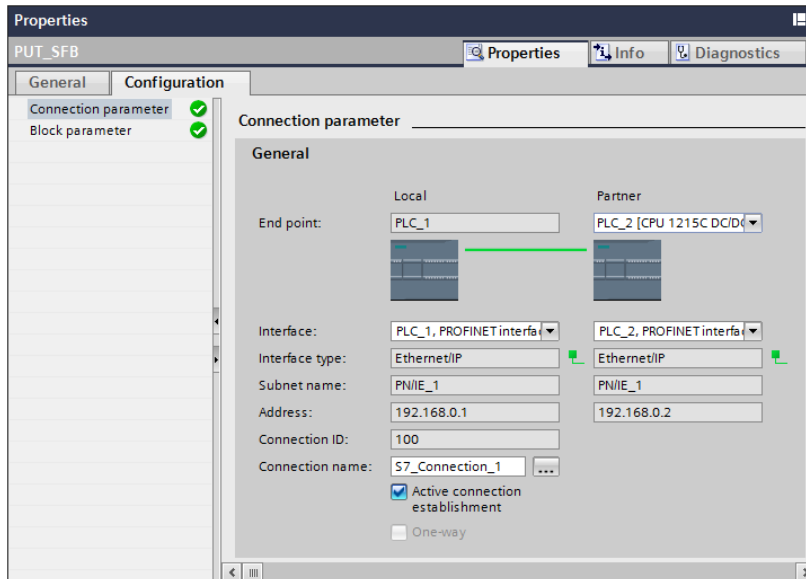


Fig. 15.10 Configuring the connection parameters for S7 communication

The communication functions can be found in the program elements catalog in the folder *Communication > S7 communication*. Drag the desired communication function into the opened block. When you release the mouse button, you will be prompted to specify the call option. Select the *Single instance* option; a separate data block is then assigned to the call.

Under *Properties* in the *Configuration* tab in the inspector window, the program editor shows the connection parameters and the block parameters (Fig. 15.10).

Fill out the fields highlighted in red that are still empty. If entries remain open, for example because the partner station still has to be created, you will later be shown the connection dialog again if the communication function is selected.

In the connection dialog, select the S7 connection for data transmission or create a new connection by clicking *Connection overview* (the button with three dots). The connection ID identifies the connection and must be the same in both stations. (Multiple connections between the partners can be created.)

Use the *Active connection establishment* option to specify which of the stations is to initiate the connection.

You then complete the fields for the block parameters in the inspector window in the *Configuration* tab in the *Block parameters* group or supply the block parameters directly at the function call in the working area.

## 15.4 Point-to-point communication

### 15.4.1 Introduction to point-to-point communication

Point-to-point (PtP) communication can be used to exchange data with external devices such as printers or barcode readers over a serial interface.

A CPU 1200 supports the Freeport protocol for character-based, serial communication so that the data transmission protocol can be completely configured via the user program. The scope of delivery of STEP 7 also includes functions for data transmission with the USS Drive protocol and the protocols Modbus RTU Master and Modbus RTU Slave.

The point-to-point communication is implemented with the CM 1241 communication module (transmission media RS232 or RS485). On the CM 1241 module are the indicators

- ▷ **Diagnostics LED**  
Flashes red after switching-on until the module is addressed (recognized) by the CPU. It then flashes green until the module has been parameterized. When ready for operation, the LED lights up green permanently.
- ▷ **Send LED**  
Lights up when data is sent to the connected device.
- ▷ **Receive LED**  
Lights up when data is received from the connected device.

Fig. 15.11 shows the communication functions available for point-to-point communication.

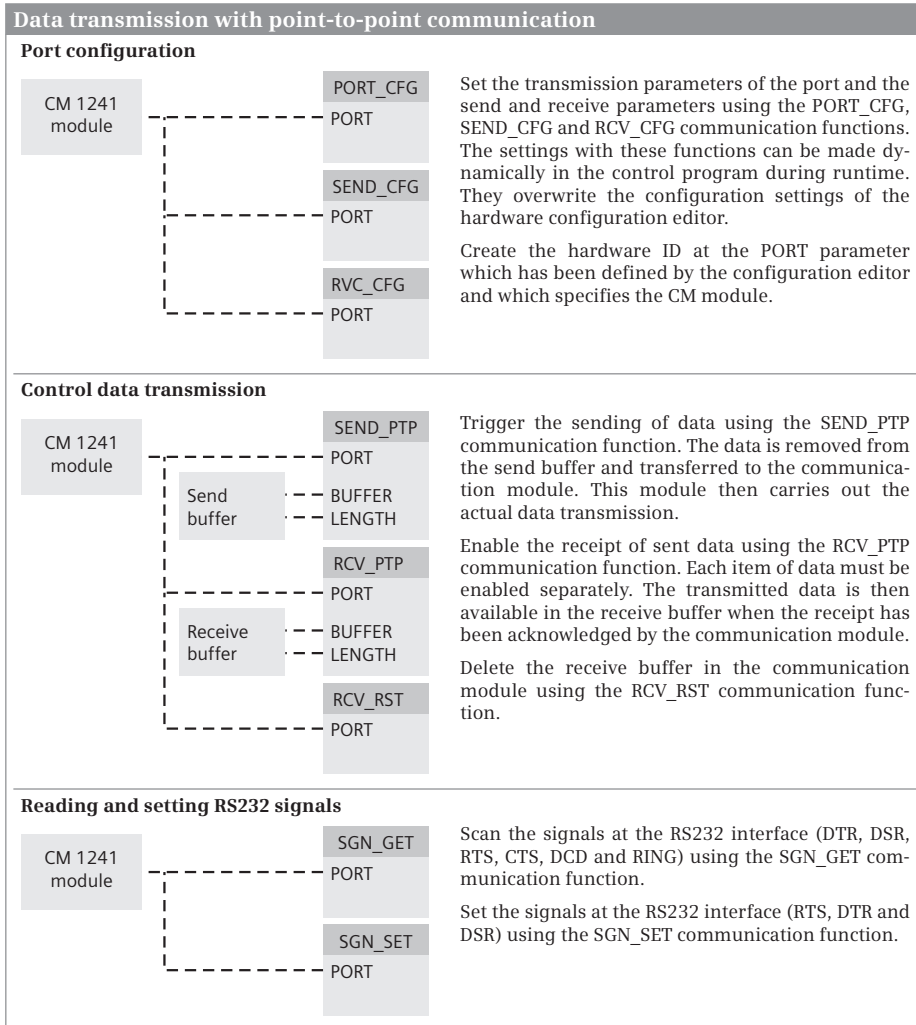


Fig. 15.11 Communication functions for PtP communication

#### 15.4.2 Configuring the CM 1241 communication module

A CM 1241 communication module is arranged on the left side of the CPU. You can operate up to three CM modules on a CPU 1200. Each module has a port (a connection of the interface) for the external device.

You can configure the parameters of the PtP interface with the hardware configuration or set them with the PORT\_CFG communication function in the user program during runtime.

A prerequisite for configuring a communication module is a project with a PLC station. Start the *device configuration* editor in the project tree under the PLC station. In the device view, select the module on the *Hardware Catalog* task card – with active filter function – under *Communication modules > Point-to-point >* and the desired interface media (RS232, RS485, or RS422/485) and drag it to the slot on the left of the CPU.

You can then set the configuration data in the inspection window.

### Port configuration

With PtP communication, data is transmitted character-by-character. A character can consist of 7 or 8 bits. A parity bit can be transmitted in addition to the character bits and is used for error detection: the signal state of the parity bit is selected for even parity such that the total of the bits with signal state “1” is even, or odd with an odd parity. The transmitted character is terminated by 1 or 2 stop bits.

When you configure the ports, set the transmission rate (baud rate), the parity type, number of data bits per character, and the number of stop bits.

Under *Wait time* you can select which time period to wait between transmissions.

You set the operating mode at the RS422/485 interface: Full duplex (simultaneous transmission in both directions) or half duplex (transmission in only one direction at a time). In the multipoint-capable connection in full duplex operation, you can operate the communication module as master or slave.

With the RS232 interface and the RS422/485 interface in full-duplex mode, you specify under *Flow control* the method used to control the exchange of data between sender and receiver. If you choose *XON/XOFF*, specify the encoding for the XON character and the XOFF character.

### Configuration of data transfer

It is possible to control the data traffic over the serial interface using a self-defined communication protocol. Set the transmission parameters depending on the RS232 or RS485 standard under *Configuration of transmitted message* and then define the beginning and end of the message under *Configuration of received message*.

#### 15.4.3 Point-to-point communication functions

The following communication functions are available for PtP communication:

- ▷ PORT\_CFG Set port configuration
- ▷ SEND\_CFG Set send parameters
- ▷ RCV\_CFG Set receive parameters
- ▷ SEND\_PTP Trigger sending of data
- ▷ RCV\_PTP Enable data receipt
- ▷ RCV\_RST Empty receive buffer
- ▷ SGN\_GET Read RS232 signals
- ▷ SGN\_SET Write RS232 signals

The calling of functions for PtP communication are shown in Fig. 15.12.

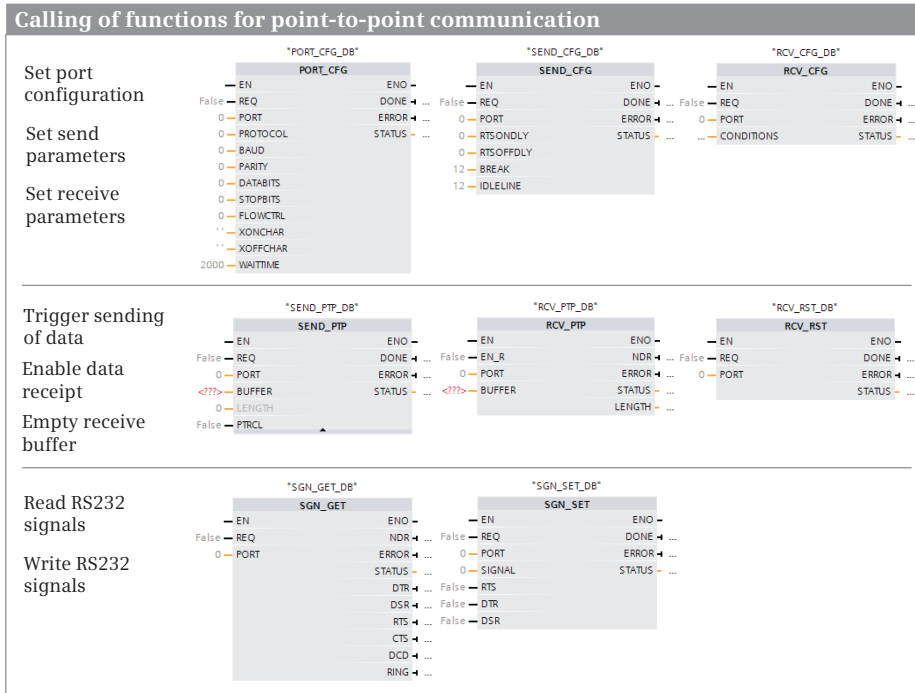


Fig. 15.12 Calls of the functions for PtP communication in LAD representation

### Programming communication functions

The communication functions for PtP communication are called in the main program; for example, in a function module which is called in the organization block OB1 or in another organization block with the event class *Program cycle*.

Open the block and the program elements catalog to program a communication function. Under *Communication* in the folder *Communications processor > Point-to-Point*, select the function and drag it to the open block. When you release the mouse button, you will be prompted to specify the call option: Call as a single instance with its own data block or as multi-instance with storage of instance data in the instance data block of the calling function block.

### Description of common parameters

If a rising signal edge occurs on the *REQ* parameter, the job is started. As long as the signal state remains “1”, no other job is accepted. Only when the communication function recognizes a signal state “0” at REQ can a new job be started with a change to “1”.

The *EN\_R* parameter activates data reception with signal state “1”.

The *DONE parameter* shows with signal state “1” that the started job has been completed without errors. The parameter is only set for the duration of one program cycle.

The *NDR parameter* (New Data Ready) shows with signal state “1” that the started job has been completed without errors and that new data has been received. It is only set for the duration of one program cycle.

The *ERROR parameter* shows with signal state “1” that the started job has been completed with an error. It is only set for the duration of one program cycle.

The *STATUS parameter* contains intermediate states or error information:

- ▷ If *DONE* or *NDR* = “1”, *STATUS* is occupied by 16#0000.
- ▷ If *ERROR* = “1”, *STATUS* is occupied by error information.
- ▷ If none of these bits is set, *STATUS* may contain intermediate states which indicate the progress of the started job.

The following parameters specify the data transmission:

The *PORT parameter* specifies the CM module with the hardware ID which is defined when configuring the CM module.

The *BUFFER* and *LENGTH* parameters define the send or receive mailbox for the transmitted data.

### **Changing the configuration settings during runtime**

The properties of a port are set when configuring the CM module. These properties are “static”, which means they are transmitted from the load memory to the CM module when switching on the PLC station and apply to further operation. Using communication functions, these properties can be changed “dynamically” during runtime. The “dynamically” changed properties are not permanent; they are replaced by the “static” properties during the next call.

The *PORT\_CFG function* changes the *PORT* properties such as baud rate, parity, number of data bits and stop bits.

The *SEND\_CFG function* controls the intervals between activation of the RTS signal and start of data transmission, between the end of data transmission and deactivation of the RTS signal, and the pauses at the beginnings and ends of messages.

The *RCV\_CFG function* influences the conditions for the start and end of data to be transmitted. Data which satisfies these conditions can be received using the *RCV\_PTP* communication function. The receive conditions are combined in the data structure *CONDITIONS*.

### **Sending and receiving data**

The *SEND\_PTP function* transmits the data from the send mailbox in the user memory to the CM module, and triggers the sending of data. The actual transmission to the external device is handled by the CM module.



The *RCV\_PTP* function enables receipt of sent data, whereby each item of data must be enabled separately. The data is transmitted from the CM module to the receive mailbox in the user memory.

The *RCV\_RST* function empties the receive buffer of the CM module.

### Reading and writing RS232 signals

The *SGN\_GET* function reads the signal states of the DTR, DSR, RTS, CTS, DCD and RING signals at the RS232 port.

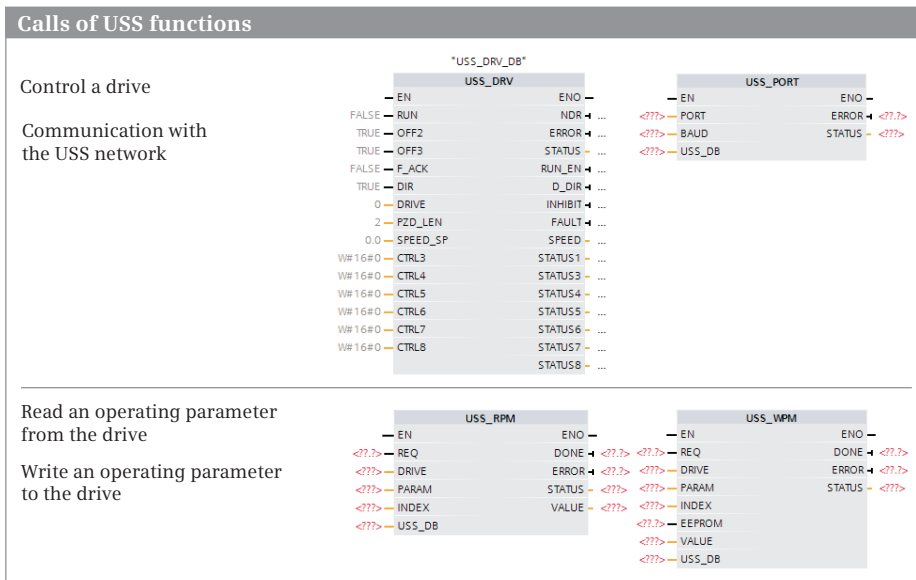
The *SGN\_SET* function writes the signal states of the RTS, DTR and DSR signals at the RS232 port.

### 15.4.4 USS protocol for drives

With the communication module CM 1241 RS485 or the communication board CB 1241 RS485, up to 16 Siemens drives that support the universal serial interface (USS) can be controlled. The following functions are provided by STEP 7:

- ▷ **USS\_DRV** Control a drive
- ▷ **USS\_PORT** Communication with the USS network
- ▷ **USS\_RPM** Read an operating parameter from the drive
- ▷ **USS\_WPM** Write an operating parameter to the drive

Fig. 15.13 shows the calls of the functions of the USS protocol in ladder logic representation.



**Fig. 15.13** Calls of the USS functions in LAD representation

A single data block is available for all controlled drives per CM 1241 communication module or CB 1241 communication board. The transmission of data is taken over by the block USS\_PORT. The block USS\_DRV controls a drive whose number is specified at the DRIVE parameter. The block USS\_RPM reads an operating parameter from the drive; the block USS\_WPM writes an operating parameter to the drive (Fig. 15.14).

The *USS\_DRV function block* controls a drive. A separate call of the function block is required for each drive. You specify the drive number at the DRIVE parameter. When calling for the first drive, assign an instance data block to the function block. For all future calls, select the same data block as instance data block, which you choose from a drop-down list.

The *USS\_RPM function* reads an operating parameter from the drive whose number you specify at the DRIVE parameter. At the parameter USS\_DB, the data block is specified that contains the data for all drives of a CM module.

The *USS\_WPM function* writes an operating parameter to the drive whose number you specify at the DRIVE parameter. At the parameter USS\_DB, the data block is specified that contains the data for all drives of a CM module. If you want to write the parameter to the EEPROM of the drive control, take note of the limited number of write accesses for an EEPROM.

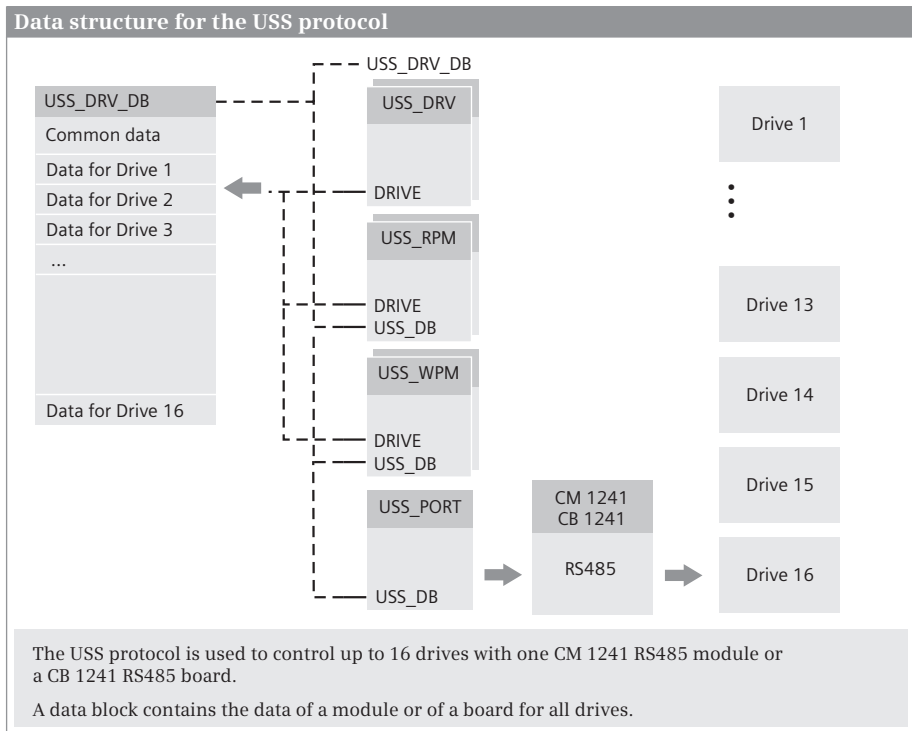


Fig. 15.14 Data structure for the USS protocol

The *USS\_PORT function* transfers the drive data between the data block and the CM module. It is called only once per CM module.

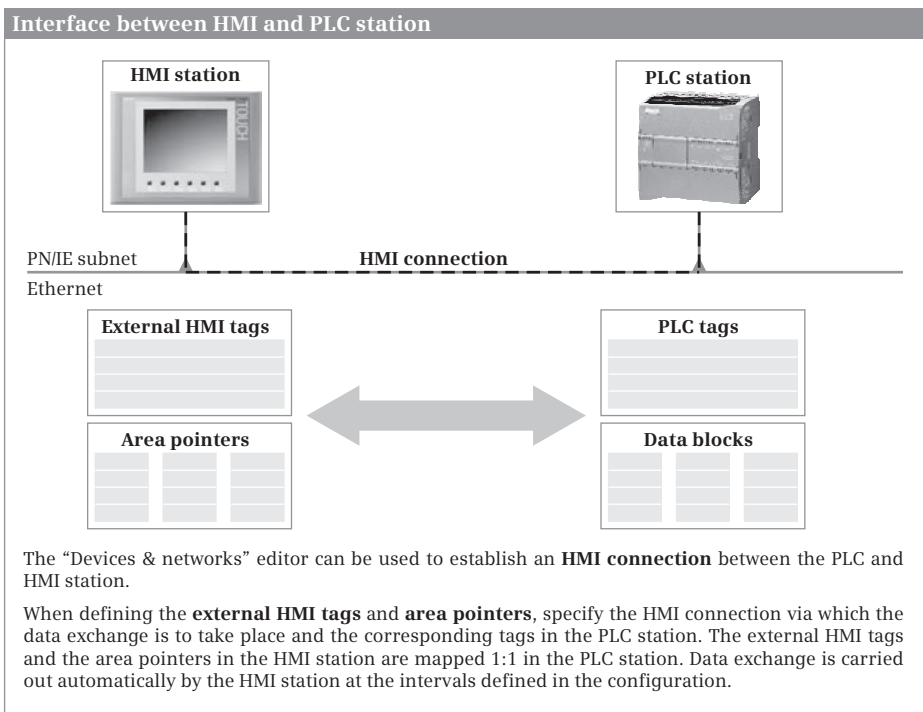
The blocks `USS_DRV`, `USS_RPM`, and `USS_WPM` must be called in the main program; any organization block is possible for the block `USS_PORT`. The processing of `USS_PORT` must not be interrupted. The block must be called in a time interval that depends on the baud rate of the serial connection and the time response of the drive.

# 16 Visualization

## 16.1 Introduction to visualization

An HMI station (HMI = Human Machine Interface) is an operator control and monitoring device for manually controlling a process, for recording process data, and for displaying process alarms.

The configuration data of the HMI station is saved in a “project”. A project contains all of the data for an automation solution. The project also includes the configuration data for a PLC station, since an HMI station on its own cannot control a process (machine or plant). The HMI station receives the data to be displayed from the process via the PLC station and controls the process via the PLC station. It is recommendable to generate the user program for the PLC station prior to configuring the HMI station; at least, however, the interface between the stations so that data exchange can be configured at the HMI station end. The interface between the stations comprises a logical connection which defines the type of data exchange (the



**Fig. 16.1** Interface between HMI and PLC station

“protocol”) and the tags and area pointers whose data is exchanged between the stations (Fig. 16.1).

The HMI program is created offline – without a connection to the HMI station. STEP 7 Basic includes WinCC Basic as the configuration software for a Basic Panel. You insert an HMI station into a project, create the process screens, and configure the contents of the screens using predefined screen objects which you can also adapt according to requirements. These objects can be texts and graphics, input and output fields for process values, alarm displays, etc.

Following completion, you can simulate the configuration on the programming device, e.g. test the configuration without an HMI station. The process values are defined either using a table of values or via the previously programmed PLC station. Following simulation, the configuration data is loaded into the HMI station and put into operation together with the PLC station.

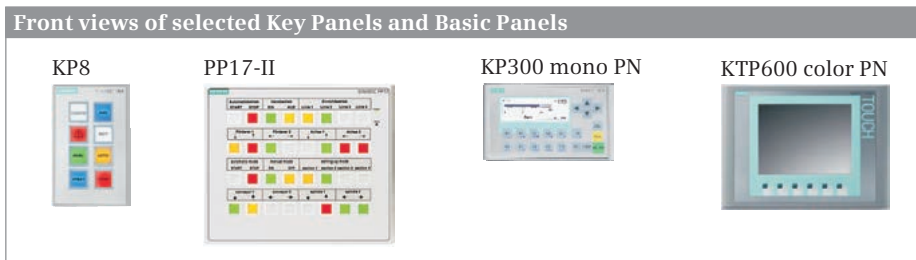
Chapter 16 “Visualization” describes the configuration of Basic Panels. The following Chapter 16.1.1 provides a brief overview of the panels that can be configured with STEP 7 Basic V11.

### 16.1.1 Overview of HMI Panels in STEP 7 Basic

Together with the new SIMATIC S7-1200 programmable controller, a range of Key Panels and Basic Panels has been provided as HMI devices, which work excellently together with the S7-1200 controllers. These devices are configured using the *WinCC Basic* application. WinCC Basic is integrated in the STEP 7 Basic engineering software and can be accessed via the *Visualization* portal. Fig. 16.2 shows the front views of the different types.

#### SIMATIC Key Panels

Key Panels are pre-assembled key panels for simple machine operation. They feature large illuminated keys with excellent tactile feedback, which can even be operated with gloves and are thus suitable for harsh industrial environments. The buttons have LED backlighting that can be adjusted in terms of brightness and color (red, yellow, green, blue, white). All keys can be individually labeled using slide-in labels. The connection to the control is implemented via PROFINET. The connection con-



**Fig. 16.2** Front views of different panels

**Table 16.1** Selected technical specifications of the Key Panels

	KP 8	KP 8F	KP 32F	PP 7	PP 17-I	PP 17-II
Number of keys	8	8	32	8	16	32
Number of LEDs	8	8	32	8	16	32
Digital inputs/outputs (user-configurable)	8	8	16	–	–	–
Additional digital inputs	–	–	16	4	16	16
Additional digital outputs	–	–	–	–	16	16
Failsafe digital inputs	–	2	4	–	–	–

sists of two RJ45 sockets that are interconnected by an integrated switch and which allow the construction of a linear structure without additional module (Table 16.1).

### SIMATIC Push Button Panels

Push Button Panels are pre-assembled operator panels for simple machine operation. They feature short-stroke keys in different numbers depending on the version. These keys can be labeled and have built-in, two-color surface LEDs. The connection to the controller is implemented through a serial interface, either as a DP standard slave for a PROFIBUS DP connection or as an I/O device for connection to PROFINET IO. The PROFINET connection consists of two RJ45 sockets that are interconnected by an integrated switch and which allow the construction of a linear structure without additional module (Table 16.1).

### SIMATIC Basic Panels

Basic Panels are available in size 3" as Key Panel (KP), from 4" to 12" with touch screen and additional keys (KTP), and as a pure touch screen (TP) in the size 15". Variants can be selected for connection to PROFINET and PROFIBUS. The degree of protection achieved when installed is IP 65 for the front and IP 20 for the rear (Table 16.1).

**Table 16.2** Selected technical data of Basic Panels

	KP 300 Basic mono PN	KTP 400 Basic mono PN	KTP 600 Basic mono PN	KTP 600 Basic color PN or DP	KTP 1000 Basic color PN or DP	TP 1500 Basic color PN
Display size	3.6"	3.8"	5.7"	5.7"	10.4"	15"
Resolution, pixels	240 × 80	320 × 240	320 × 240	320 × 240	640 × 480	1024 × 768
Colors	black/white	4 gray levels	4 gray levels	256 colors	256 colors	256 colors
Touch screen	no	yes	yes	yes	yes	yes
Function keys	10	4	6	6	8	no
User memory	512 KB	512 KB	512 KB	512 KB	1024 KB	1024 KB

### 16.1.2 Creating a project with an HMI station

The basis for configuring an HMI station is a project. The project contains the data structure for the automation solution within which the configuration data of an HMI station is also to be accommodated. At the top of the hierarchy, a project contains the configured PLC and HMI stations, the *Online access* folder with the LAN adapters of the programming device, and the *SIMATIC card readers* folder with the SD card readers.

In order to configure an HMI station, it is sufficient to merely create the HMI station in the project. However, it is recommendable to first create the PLC station to which the HMI station is to be connected. The communication connections can then be defined directly when configuring the corresponding HMI elements.

How to create a project is described in Section 1.3.3 “Creating and editing a project” on page 41.

#### Adding an HMI station

Prerequisite: You have created a project.

An HMI station can be added in both the portal view and in the project view. In the *portal view*, select the *Open existing project* command in the start portal, select the project in the table, and click on the *Open* button. In the next window, select *Configure a device*. In the *Devices & networks* portal, which then opens, choose *Add new device*. The device selection window is opened (Fig. 16.3).

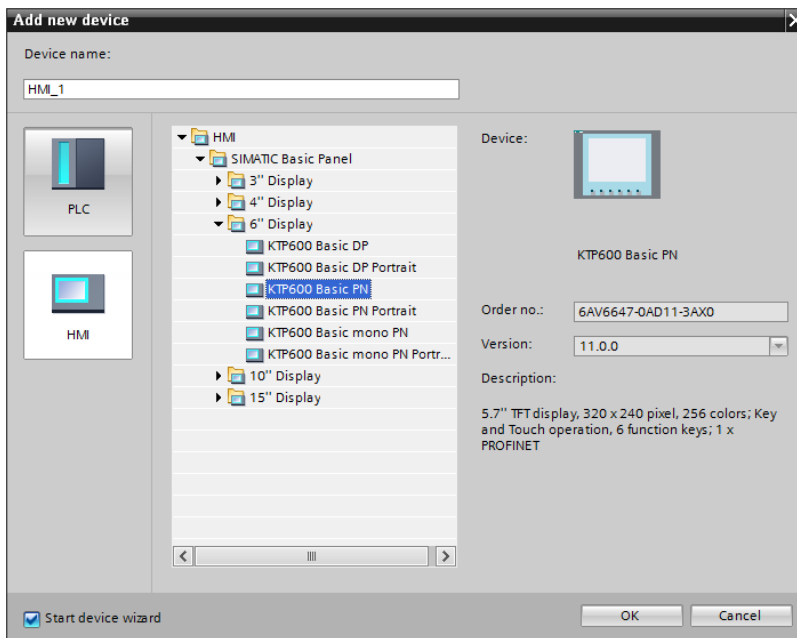


Fig. 16.3 Selection window *Add new device*

To add an HMI station in the *project view*, open the project and double-click in the project tree on the *Add new device* command. The device selection window is then opened.

In the selection window, click on the “HMI” button, select the desired device from the hardware catalog, and assign a meaningful name. If you wish to be guided by the HMI device wizard through the further basic configuration, activate the *Start device wizard* check box. Click on the OK button. The HMI device wizard is now started if it has been activated, otherwise the HMI editor starts.

### Using the HMI device wizard

The HMI device wizard guides you through the basic configuration in the following steps:

- ▷ In the *PLC connections* window you can select – if present – the PLC station to which the HMI station is to be connected.
- ▷ In the *Screen layout* window you can select the screen resolution and background color and define whether a header is to be displayed with the date and company logo. The settings are stored in *Template\_1*.
- ▷ In the *Alarms* window, you can select the alarms to be displayed: unacknowledged alarms, active alarms, and active system events. The settings are stored in *Template\_1*.

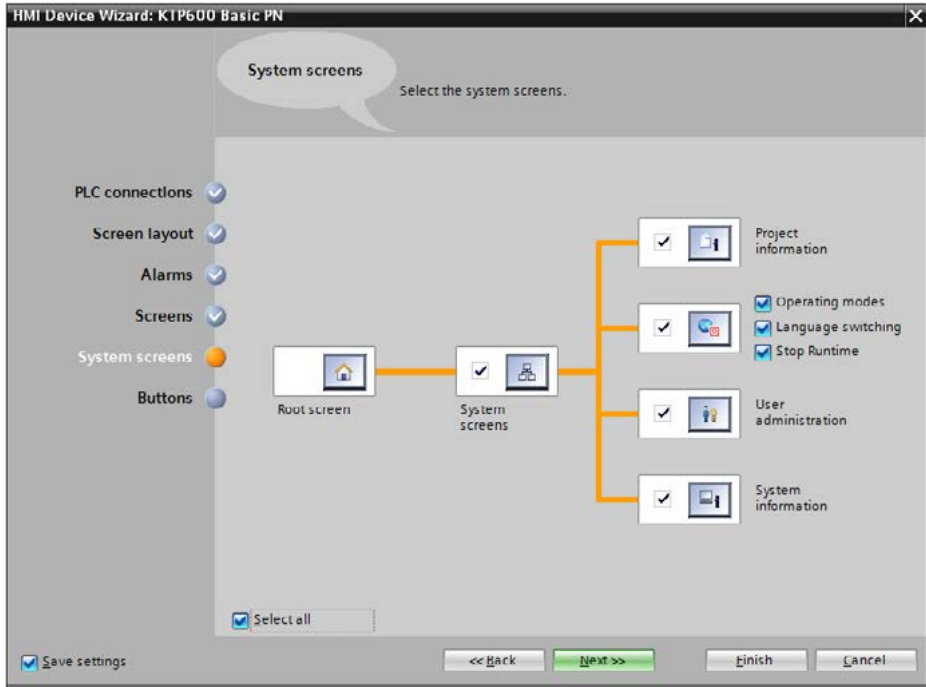


Fig. 16.4 Configuring system screens using the HMI device wizard



- ▷ In the *Screens* window, you can use a root screen to generate further screens in your desired selection hierarchy, i.e. which screens can be called from a specific screen. You can assign names to the screens.
- ▷ In the *System screens* window you can specify which predefined screens are to be used to display system functions such as operating modes, user administration, and project and system information (Fig. 16.4).
- ▷ In the *Buttons* window you can position the button areas (left, bottom, or right) and arrange the system buttons (symbols for root screen, login, language, and exit) in the button area. The buttons for the root screen and unacknowledged alarms are already present. Drag the icons – even those that have already been added – to the buttons you want.
- ▷ Click *Finish* to exit the HMI device wizard.

You can change or supplement these settings during the further configuration.

### 16.1.3 Cross-references for HMI objects

The cross-reference list provides an overview of the locations at which HMI objects are used. If you wish to change an object, for example, the cross-reference list shows you the positions in the configuration at which this change has an effect.

The objects displayed in the cross-reference list are the HMI station, all folders in the project tree under the HMI station, and the HMI editors with the configured objects. To display the cross-reference list, select an object and then the *Cross-references* command from the shortcut menu.

The cross-reference list is available in two views: The *Used by* tab shows the positions at which the selected object is used. The *Uses* tab shows the objects used by

Object	Num...	Point of use	Property	Connected to	Type	Path
Root screen					Screen	Project1200\Gate handling\Screens
Binary display	1				Screen	Project1200\Gate handling\Screens
Conveyor belt	1				Screen	Project1200\Gate handling\Screens
Counter display	1	Button_2	Release		Screen	Project1200\Gate handling\Screens
Delivery data	1				Screen	Project1200\Gate handling\Screens
Gate handling	1				Device	Project1200
System screens	1	Start screen			Screen	Project1200\Gate handling\Screens
Template_1	1	Template_2	Release		Template	Project1200\Gate handling\Screens
User administration	1				Screen	Project1200\Gate handling\Screens

**Fig. 16.5** Example of a cross-reference list for HMI configuration

the selected object. In the *Used by* view you can choose the options *Show used* and/or *Show unused*, in the *Uses* view the options *Show defined* and/or *Show undefined* (Fig. 16.5).

You can use the icons in the toolbar to update the cross-reference list and to define its settings. For example, the selection *Show undefined* also shows the references to previously deleted objects.

The entries in the list can be opened or closed in order to display or hide the subordinate objects. Clicking on the link in the *Point of use* column opens the corresponding editor in order to process the referenced object.

You can also display the cross-reference list in the inspector window: Select an object in the working window or in the project tree. Select *Cross-reference information* from the shortcut menu or open the *Info* tab in the inspector window and then the *Cross-reference* tab. You are provided with the cross-references for the selected object and the subordinate objects.

## 16.2 Creating HMI tags and area pointers

### 16.2.1 Introduction to HMI tags

A tag identifies a memory location for a value (data content) with a data type (data format) by means of a name (symbol). For example, a tag can be defined with the name “Level”, which represents a fixed-point number (data type UINT) with a (start) value of 0.

Two types of HMI tags are encountered in the engineering software for an HMI station: internal tags and external tags.

An *internal HMI tag* does not have a connection to the PLC station. It is saved in the HMI station, and can only be used by the program of this station. Table 16.3 shows the data types which can be used for internal HMI tags.

**Table 16.3** Data types for HMI tags

Designation	Description	Designation	Description
BOOL	Binary tag	REAL	32-bit floating-point number
SINT	8-bit value with sign	LREAL	64-bit floating-point number
USINT	8-bit value without sign	WSTRING	String with 16-bit character set
INT	16-bit value with sign	DATETIME	Date and time in the format DD.MM.YYYY hh:mm:ss
UINT	16-bit value without sign		
DINT	32-bit value with sign	ARRAY	One-dimensional field with an index range from 0 to maximum 99
UDINT	32-bit value without sign		

An *external HMI tag* (process tag) is the image of a memory location in the PLC station. This memory location can be accessed by both the HMI station – with the HMI tag name – and by the PLC station – with the PLC tag name. If an HMI connection between the PLC station and the HMI station is configured in network view, this is called an *integrated connection*. In an integrated connection, the PLC tag can be addressed absolutely or symbolically and can be in a data block with optimized access or standard access. A connection is not integrated if it is configured in the connection table, for example because both stations are not located together in the same project. A PLC tag can be accessed via a non-integrated connection only with absolute addressing.

With the HMI configuration, the expression “PLC tag” stands for a tag in the PLC station. This can be a tag from the PLC tag table or a data tag from a data block.

The external HMI tags accept the data types assigned to them in the PLC station (PLC data types, not with data type STRUCT). With a PLC tag with data type ARRAY, the HMI tag accepts the number (“array elements”) and the data type of the elements.

In addition to the external tags, it is also possible to exchange data between the HMI and PLC station using *Area pointers*.

### 16.2.2 Creating an HMI tag

All HMI tags are located in the project tree under the HMI station in the *HMI tags* folder. When creating an HMI station, the *default tag table*, into which you can enter the HMI tags, is automatically created. Double-click on *Add new tag table* to add additional tag tables. For the selected *HMI tags* folder, you can create subfolders for a better overview with the *Add group* command from the shortcut menu. A tag can only be defined once (in an HMI station there is only a single tag table, which can be divided into subtables). Double-click on *Show all tags* to obtain an overview of all configured HMI tags.

To create an HMI tag, open an HMI tag table by double-clicking on it. If you want to create an internal tag, select *<Internal tag>* as connection. In the *Data type* column, select the HMI data type from a drop-down list. For the ARRAY data type, enter the range in square brackets, followed by the keyword OF and the data type, e.g. ARRAY [0..24] OF INT.

With an external tag (process tag), set the connection to the PLC station and select the PLC tag. You can select tags from a PLC tag table or from data blocks (in the *Program blocks*) folder. The data type of the PLC tag is automatically applied. Under *Acquisition cycle*, set the time interval with which the updating is to be carried out. Make sure with these settings that the communication load on the connection between the HMI and PLC stations remains within acceptable limits.

### Configuring an HMI tag

In the *Properties* tab of the inspector window, you can set further properties of the selected tag. For example, you can set the type of recording in the *Properties* section under *Settings*:

- ▷ *Cyclic in operation* = the tag is updated regularly as long as it is displayed in a screen
- ▷ *Cyclic continuous* = the tag is updated regularly even if it is not displayed in a screen
- ▷ *On demand* = the tag is updated on request, e.g. by a system function

Under *Range* you can set an upper and a lower limit value for the selected tag. Under *Linear scaling* the range of values of the PLC tag is converted linearly into a range of values in the HMI station. Scaling is carried out for the data exchange in both directions. The start value when switching on the HMI station or up to the first update is defined under *Values. Multiplexing* (indirect addressing) allows you to determine the used tag during runtime, and not before. A multiplex tag consisting of a list of tags is used for configuration. An index tag then selects the used tag from this list.

In the *Events* tab you can assign a function list with system functions to be executed if certain events such as a change in value or a limit violation occur.

### 16.2.3 Creating an area pointer

An area pointer defines a memory area in the PLC station. The programs of the HMI and PLC stations exchange data via this memory area. Example: when changing the screen, the number of the current process screen is transferred to the PLC station by means of the area pointer *Screen number*. The program of the PLC station can respond to this.

The area pointers must be configured prior to use. The length of an area pointer is specified in 16-bit words. The memory area to which an area pointer refers is present in a data block in the user program. An area pointer with a length = 1 can also be a PLC tag from the PLC tag table. With a length of > 1, the PLC tag in the data block is created with data type ARRAY and a 16-bit data type (e.g. WORD) with the length of the area pointer, e.g. ARRAY [1..4] OF WORD.

#### Overview of area pointers

Table 16.4 lists the available area pointers. The global area pointers *Screen number*, *Date/time PLC* and *Project ID* can only be used once per station and only in one connection.

- ▷ *Screen number*: When changing the screen, the HMI station transfers the screen number to the PLC station (global area pointer).
- ▷ *Job mailbox*: A control job consists of a job number and up to three parameters. The PLC station writes a control job into the data area; if the HMI station has accepted the job, it overwrites the job number with zero.

- ▷ *Date/time*: The PLC station writes the control job No. 41. The HMI station then transfers the date and time in data type DTL to the PLC station.
- ▷ *Date/time PLC*: The PLC station writes the date and time in data type DTL to the data area which is then read cyclically by the HMI station. The HMI station imports the date and time (global area pointer).
- ▷ *Coordination*: The HMI station transfers its actual operating mode in this data area: Startup (bit 0 = “0” during startup), offline/online mode (bit 1 = “0” with online mode) and communication readiness (by means of a “sign-of-life bit”, bit 2 changes its signal state at approx. 1 Hz).
- ▷ *Project ID*: The HMI station can recognize whether it is connected to the “correct” PLC station. The project ID is created when configuring the HMI station: double-click on the *Runtime settings* editor in the project tree under the HMI station, and enter a value between 1 and 255 in the *Screens* section in the *Project ID* field. Then write the same value in the PLC station in the data area of the *Project ID* area pointer. During startup, the HMI station compares the two values, and does not start unless they agree.
- ▷ *Data record*: The *Data record* area pointer is required for synchronized transfer of a recipe data record. Handling of recipes is described in Section 16.4.3 “Working with recipes” on page 535.

**Table 16.4** Area pointers for Basic Panels

Area pointer	Required for	Length	HMI	PLC
Screen number	Transfer of number of current screen	5	Writes	Reads
Job mailbox	Triggering of functions on the HMI station with the job numbers: 14: Set time BCD-coded 15: Set date BCD-coded 23: Log on user 24: Log off user 40: Transfer date/time to PLC station 41: Transfer date/time to PLC station (DTL) 46: Update tag 49: Clear event buffer 50: Clear error alarm buffer 51: Screen selection 69: Read data record from PLC station 70: Write data record to PLC station	4	Reads and writes	Reads and writes
Date/time	Transfer of date and time to PLC station	6	Writes	Reads
Date/time PLC	Transfer of date and time to HMI station	6	Reads	Writes
Coordination	Transfer of HMI station status to PLC station	1	Writes	Reads
Project ID	Checking project ID in HMI station	1	Reads	Writes
Data record	Transfer of recipe data records with synchronization	5	Reads and writes	Reads and writes

## Configuring an area pointer

Prerequisite: A PLC station, an HMI station, and an HMI connection have been created in the project. The user program of the PLC station contains a data block in which the data area for the area pointer is declared (see Chapter 6.4 “Programming a data block” on page 194).

Double-click on the *Connections* editor in the project tree under the HMI station. The top part of the connection window contains a table with the configured connections.

There are two tables in the bottom part of the connection window in the *Area pointers* tab: the table with the area pointers created for the connection selected in the connection table is shown at the top, and at the bottom the table with the global area pointers which can only be created once in the HMI station and only in one connection.

You can activate an area pointer in the top table using the *Active* check box and in the bottom table by specifying the connection. Assign the PLC tag to the area pointer. The PLC tag must have been created previously in the PLC station with the same length as the area pointer. You can set the acquisition cycle using the *Job mailbox* and *Date/time PLC* area pointers.

## 16.3 Configuring process screens

### 16.3.1 Introduction to configuring process screens

You use process screens to operate and monitor the process. A process screen can map a plant, display process sequences, output process values, or facilitate operator actions. Predefined objects are available for creating a screen, and can be inserted and adapted according to your requirements.

The properties of a screen, for example the resolution and colors, depend on the HMI station used.

A screen may consist of static and dynamic objects. Static objects are texts or graphics which do not change during process operation. Dynamic objects are e.g. texts, numerical values, trends and bars which change depending on process values.

You can also use a screen to access the process by means of control elements or to call another screen. An input field allows definition of a setpoint for the process, and you can use function keys – if the HMI station is appropriately equipped – to trigger actions in the process, for example. Global function keys always trigger the same action, regardless of the screen in which they are positioned. With local function keys the triggered action depends on the displayed screen.

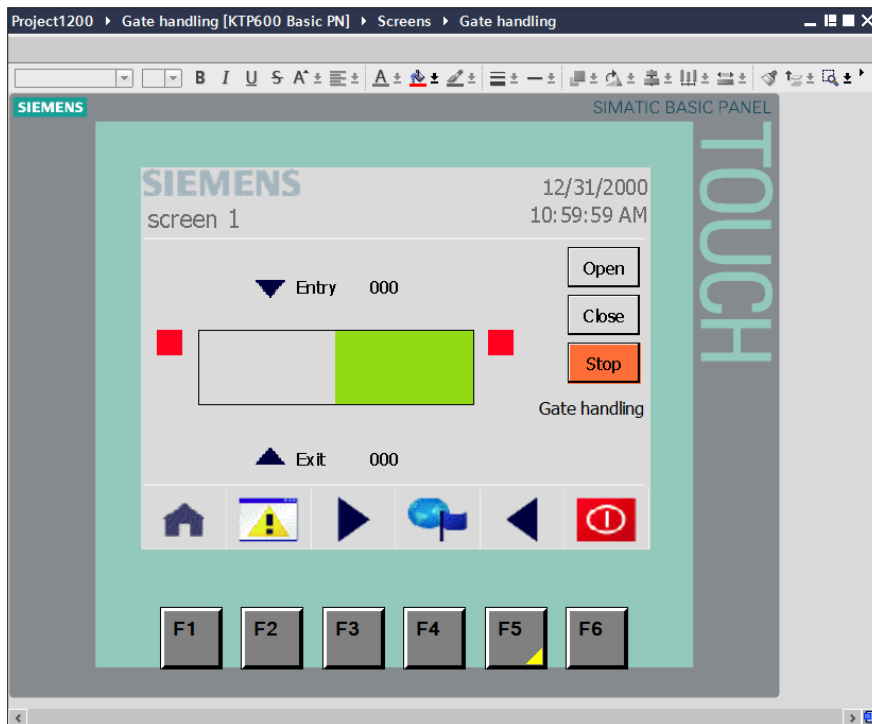
Configuration of the process screens is carried out in the following steps:

- ▷ Beginning with a root screen, the number of screens and their call hierarchy are defined.
- ▷ The navigation strategy within a screen and between screens is defined.

- ▷ The existing template or the global screen is adapted, and/or new templates are created.
- ▷ Screens are created using the objects saved in the libraries.

### 16.3.2 Working window for process screens

The toolbar of the working window for process screens contains the most important setting options for the properties of a selected screen object or for displaying the screen contents (Fig. 16.6).



**Fig. 16.6** Working window for process screens

Object properties such as the font family and color can also be set in the inspector window. You can overlap objects, rotate them vertically and horizontally, and align them on the grid. You can set the grid width, grid display, and alignment on the grid in the *Grid* section on the *Layout* task card.

You can use the zoom function to set the size – either using the icons in the toolbar or in the *Zoom* section on the *Layout* task card.

### 16.3.3 Working with screen layers

Each process screen has 32 layers which are “overlaid”. Layer 0 is the lowest layer, layer 31 is in the foreground. Layers allow objects to be given nesting depth. The objects of a layer are also “stacked”. The first object inserted is located at the “rear” layer, the next one above it, etc. Objects can be shifted within a layer from the front to the rear, or can be shifted to a different layer.

One of the 32 layers is always active; this is the layer at which the process screen is currently being processed. All 32 layers are displayed when a screen is opened. You can hide all layers during configuration except for the active layer.

You can set the visibility of a layer either with the selected screen background in the inspector window in the screen properties under *Layers* or on the *Layout* task card under *Layers*.

To change the (stack) sequence of an object, select the object and then one of the *Bring to front*, *Move forward*, *Move backward* or *Send to back* commands from the *Order* shortcut menu. The stack order of the objects present in the layer is also shown in the *Layout* task card: the highest object is the one at the rear. You can use the mouse to drag the screen objects into a different order. In the same manner you can drag a screen object to a different layer. You can also define the layer for a screen object in the inspector window, in the screen properties under *Miscellaneous*.

### 16.3.4 Working with templates

You can use a (screen) template as the basis for a process screen. The screen then imports the template's objects. You can modify these objects in the current screen, and add new objects. If you modify an object in the template, the object is modified in all screens based on this template.

The templates are saved in the project tree under the HMI station in the *Screen management* folder under *Templates*. You can copy, rename or delete templates. You can structure the *Templates* folder by selecting it and then the *Add group* command in the shortcut menu.

A standard template is also created when you create an HMI station. You can adapt this to your requirements: open the template in the project tree under the HMI project in the *Screen management* and *Templates* folder by double-clicking on *Template\_1*. You can then add new objects or modify existing ones (see Section 16.3.8 “Working with objects in process screens” on page 522).

You can also create other templates in addition to the standard template: double-click on *Add new template* in the *Templates* folder, and set the template's properties in the inspector window. You can then configure the contents of the template.

### Global screen

In the global screen you can configure the screen objects for the entire HMI station which then apply to all screens irrespective of the template on which they are



based. These objects are the function keys, the alarm window, and the alarm indicator.

You define the basic assignments of the function keys in the global screen. These assignments are imported into all other screens, and can be replaced by local assignments in the templates and in the individual screens.

To configure the global screen, double-click on *Global screen* in the *Screen management* folder.

### 16.3.5 Working with function keys

A function key is a key on the HMI station with a configurable assignment for *Press key* and *Release key*. Function keys with a *global assignment* always trigger the same action, regardless of the currently displayed process screen. Function keys with a *local assignment* can trigger a different function in each screen. The local assignment of a function key in a template applies to all screens based on this template. The local assignment of a function key in a screen overwrites the assignment from the template on which the screen is based, and this in turn overwrites the global assignment of the global screen.

Note: If a screen with local function keys is overlapped by an alarm view or an alarm window, the function keys are nevertheless active. This may occur in particular on HMI stations with small displays.

#### Configuring a function key

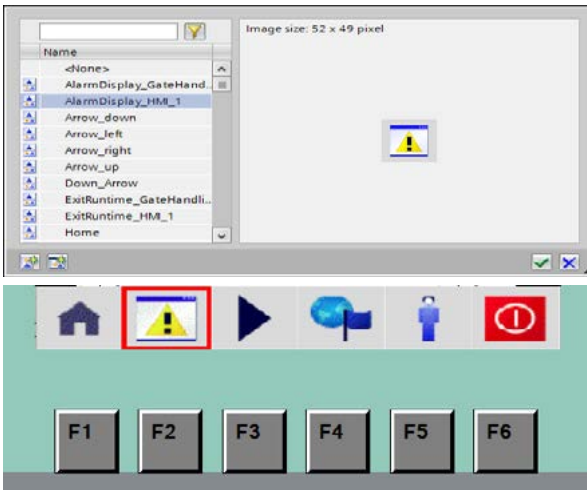
The configuration procedure is the same in a global screen, template, or process screen. Open the global screen, template or process screen, and select the function key. Make the settings in the inspector window under *Properties* and *General*.

When configuring a template, the assignment of a function key is imported from the global screen if the *Use global assignment* check box is activated. When configuring a screen, the assignment is imported from the template or global screen if the *Use local template* check box is activated. In order to make an assignment which is only applicable to the template or screen, deactivate the check box, and select a screen and, if applicable, an access privilege.

Assign a graphic to the function key which is then displayed on the function key in the screen area. Fig. 16.7 shows some examples of standard graphics.

The graphics from the *Project graphics* folder under *Languages & resources* in the project tree are offered as standard. You can also integrate your own graphics into this folder.

You can limit the circle of users with access to a function key. A requirement is the definition of user groups and privileges. Set the desired privilege in the properties of the function key under *General* in the *Runtime authorization* input field. Creation of user administration is described in Section 16.4.4 “Working with the user administration” on page 539.



In the lower figure, several standard graphics are shown as examples from left to right:

- ▷ “NavigateHome\_HMI”
- ▷ “AlarmDisplay\_HMI”
- ▷ “Right\_Arrow”
- ▷ “ToggleLanguage\_HMI”
- ▷ “Login\_HMI”
- ▷ “ExitRuntime\_HMI”

**Fig. 16.7** Selection window for the graphic and examples of standard graphics

You can configure the action to be triggered in the properties of the function key under *Events* and *Press key* or *Release key*. Click in the function list on *<Add function>*, and select the desired function from the drop-down list (see Section 16.4.1 “Input and display of process values” under “Working with function lists” on page 528). Add further functions as required.

### 16.3.6 Creating a new screen

When creating a project you can use the HMI device wizard to create the required screens and their call hierarchy. You can also add new screens at any time.

Double-click in the project tree under the HMI station and the *Screens* folder on the *Add new screen* editor. In the inspector window in the *Properties* tab under *General*, enter the name and number of the screen, define the colors for the background and grid, and also specify whether the screen display is based on a template.

In a screen without template – or, to be more precise: with the general template containing the properties of the HMI station – the configuration only applies to the current screen. If the screen is based on a template, modify the template centrally for all screens derived from it. For example, you can create a template with a specific assignment of the function keys, and derive all screens with function keys from this.

The screen is stored in the *Screens* folder. You can copy and rename a screen. You can structure the *Screens* folder by inserting further folders: select the *Screens* folder and then *Add group* in the shortcut menu.

### 16.3.7 Configuring a screen change

Starting with a root screen which is displayed when the device is switched on, it is possible to call further screens with a button or function key during runtime. The root screen, the further screens, and the links between the screens can be defined when creating the HMI project using the HMI device wizard. You can modify or extend these definitions at any time.

To define the root screen, double-click in the project tree under the HMI station on *Runtime settings* and set the start screen in the *General* section.

Prerequisite for configuring a screen change: the screen to be selected is present in the *Screens* folder, and the current screen is open in the working window.

In order to assign a screen change to a button, drag the screen to be opened into the working window with the mouse button pressed. A button is created labeled with the name of the screen. In order to assign a screen change to a function key, drag the screen to be opened to the function key with the mouse button pressed. The function key then shows a yellow triangle.

You can then set the button's properties in the inspector window. Under *Events* and *Click* (with a button) or *Events* and *Press key* (with a function key) you can use the *ActivateScreen* system function to set the new screen and the number of the object which is to be focused following the screen change.

### 16.3.8 Working with objects in process screens

The graphic objects for designing process screens are present on the *Tools* task card (Table 16.5). When configuring screens and templates, *Tools* has the following categories:

- ▷ *Basic objects* (line, ellipse, circle, rectangle, text field and graphic view)
- ▷ *Elements* (I/O field, button, symbolic I/O field, graphic I/O field, date/time field, bars and switches)
- ▷ *Controls* (alarm view, trend view, user view and recipe view)
- ▷ *Graphics* (symbols from a variety of fields)

Objects for process screens can also be included in libraries. *Global libraries* are supplied together with WinCC Basic, for example the *Buttons-and-Switches* global library on the *Libraries* task card. You can save your own frequently-used objects in the *Project library*. You can copy the HMI objects directly into the project library or create a separate folder: select the project library and then *Add folder* in the shortcut menu. You can also insert external graphics with standard file formats into the graphics libraries.

You can copy objects from the tools folder or from the library into the open screen, or drag them into the screen using the mouse. You can then process the object, for example change the size or color, rotate or mirror it, or shift it in front of or behind other objects (for processing of stack order and layers, see Section 16.3.3 “Working with screen layers” on page 519).

If you wish to select several objects, use the mouse to draw a selection frame around the objects, or click on the objects with the Shift key pressed. Copy an object by dragging it to the new position with the Strg or Ctrl key pressed.

Object groups consist of several objects which have been “grouped together”. It is also possible to group groups together (hierarchical structure of grouping). All objects of a group are located in the same layer.

An HMI object in a process screen is configured, and usually also assigned parameter settings. The structure, text format and object representation can all be defined during configuration if so desired. During parameterization, tags are assigned to the object, for example texts for alarms to be displayed or numerical values for bar heights. The parameters which are essential or optional for the HMI objects are shown in Table 16.5.

**Table 16.5** Overview of screen objects and the parameters used

Category	Object	Necessary parameters	Parameters as required
Basic object	Line, ellipse, circle, rectangle, text field, graphic view	–	–
Elements	Bar	Tag for bar height	–
	I/O field	Tag for input or output	Function list (with configuration as input field)
	Symbolic I/O field	Tag for text selection or input, text list	Function list (with configuration as input field)
	Graphic I/O field	Tag for graphic selection or input, graphics list	Function list (with configuration as input field)
	Button	Function list	Tag for selection of inscription, text or graphics list
	Switch	Tag for switch position	Function list
	Date/time field	None (with display of system time)	Tag for date/time, function list (with input)
Controls	Alarm view	(configured alarms)	Function list
	Trend view	Data source for displayed trends	Function list
	Recipe view	Recipes	Tag for recipe data record
	User view	(configured users and privileges)	Function list

It is advantageous to create the tags for these parameters in advance. If these are external tags, it is recommendable to create the tags in the PLC station prior to definition of the HMI tags.

### 16.3.9 Changing screen objects during runtime

The dynamic updating of screen objects is used to display changes or process sequences in process screens. For example, the level in a tank can be displayed graphically depending on a process value. Predefined animations are available for the dynamic updating of the following object properties:

- ▷ Structure: the object changes its appearance, e.g. its color
- ▷ Position: the object moves in the screen
- ▷ Visibility: the object is displayed or hidden
- ▷ Operability: operator control of an object is enabled or locked

You can only configure one animation of the same type (movement, structure, visibility) for each object.

To configure an animation, select the object in the process screen and click in the inspector window in the *Properties* tab on the *Animations* tab. Select a new animation by double-clicking on *Add new animation* in the tabbed browsing or selecting it from the overview. When adding, specify the tag whose value is to dynamize the object property. The new animation is displayed in the *Animations* tab in the inspector window.

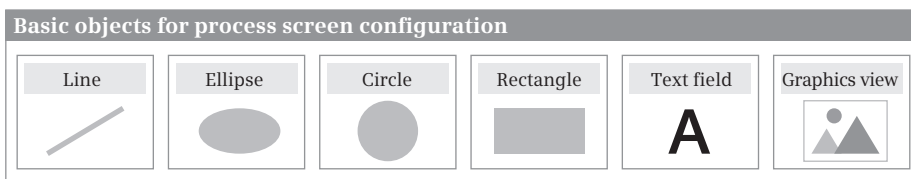
When multiple objects have been selected, set the animations of the reference object in the inspector window. The settings apply to all objects which support these animations.

If you configure an animation for an object group, this animation applies to all individual objects which support this animation.

### 16.3.10 Basic objects for screen configuration

The *Basic objects* category contains the following geometric objects: line, ellipse, circle and rectangle. The size, color, border type and filling pattern can be set depending on the object.

The group also contains a text field in which you can change, for example, the text style or the text, background and frame color, as well as a graphic view with a selectable graphic (Fig. 16.8).



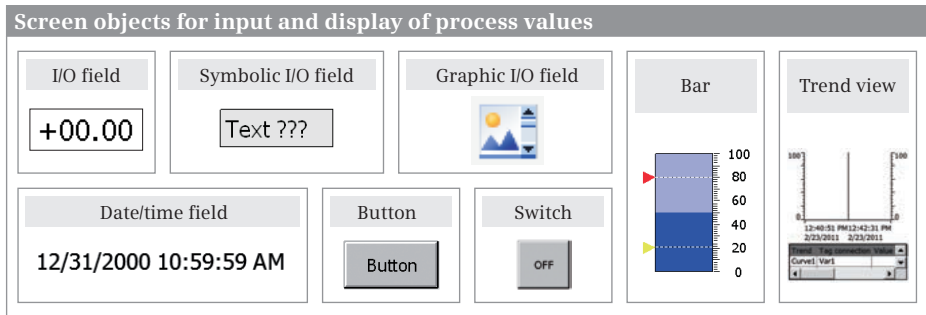
**Fig. 16.8** Basic objects in the *Tools* task card

## 16.4 HMI functions

### 16.4.1 Input and display of process values

The following screen objects are available for the input and display of process values (Fig. 16.9):

- ▷ I/O field: Input and display of numerical values
- ▷ Symbolic I/O field: Input and display of numerical values with text support
- ▷ Graphic I/O field: Input and display of numerical values with graphic support
- ▷ Date/time field: Display of date and time
- ▷ Bar: Display of a numerical process value in the form of a bar graph
- ▷ Trend view: Display of several associated, numerical process values as a trend



**Fig. 16.9** Screen objects for input and display of process values

The input and display of process values in association with alarms and recipes as well as user administration are described in the following sections. How to configure the objects can be found in Sections 16.3.8 “Working with objects in process screens” on page 522 and 16.3.9 “Changing screen objects during runtime” on page 524.

### HMI objects

The **I/O field** object is used for the input and display of process values. During configuration, you can define the mode in which the I/O field is to work during runtime: *Input*, *Input/output* or *Output*. You can set that the input remains hidden (only asterisks are then displayed) and you can assign an input privilege to the I/O field. Possible display formats include the decimal value with sign and decimal place, date/time, and string. You can set a change in color depending on upper and lower limits.

The **Symbolic I/O field** object is used for the input or output of a process value with display of a text. If the I/O field is configured as an input field, a value is assigned to the configured tag depending on the displayed text which is selected from a text list during runtime. With an output field, the text stored in a text list is displayed, depending on the value of a selection tag. Working with text lists is described in the next section.

The **Graphic I/O field** object is used for the input or output of a process value with display of a graphic. If the I/O field is configured as an input field, a value is assigned to the configured tag depending on the displayed graphic which is selected from a graphics list during runtime. With an output field, the graphic stored in a graphics list is displayed depending on the value of a selection tag. Working with graphics lists is described in the next section.

The **Date/time field** object can be used to display and enter the date and/or time. The date/time tag can be the system time of the HMI station or an HMI tag.

The **Button** object is used to trigger a configurable action. A function list with system functions can be processed depending on the event (e.g. press, release). The inscription can be text or a graphic, either fixed or selected from a text or graphics list. The processing of text, graphics and function lists is described in the next sections.

The **Switch** object enables the selection of two configurable states. The current state can be shown in the switch by a text or graphic. A function list with system functions can be processed depending on events (e.g. switch ON, switch OFF). The processing of function lists is described in the section after next.

The **Bar** object displays a process value in graphic form. The display properties which can be set include: the color response (change in color on violation of limits), the marking of limits, the division into bar segments, and the scale inscriptions and division.

The **Trend view** object displays tag values in the form of trends. You can set trend display features with regard to position, geometry, style, color and font in the inspector window. You can also specifically define whether a value table, a ruler or a grid is to be displayed in addition to the coordinate system.

### **Working with text and graphics lists**

Text lists consist of individual texts which are assigned to the values of a tag. For example, a text list can be assigned as a selection list to a symbolic I/O field: if the I/O field is an input field, the tag assumes a specific value when a text is selected; if the I/O field is an output field, the corresponding text is displayed if the tag has a specific value.

Graphics lists consist of individual graphics which are assigned to the values of a tag. A graphic can be present in a graphics library or be an existing file in the standard file format. Graphics lists are handled like text lists.

A multilingual configuration is possible for a text or graphics list, which is displayed in the current operating during operation (runtime). This is meaningful e.g. for the graphic text of a graphics list.

To create a text or graphics list, double-click in the project tree under the HMI project on *Text and graphics lists*. Create the text lists in the *Text lists* tab, and the graphics lists in the *Graphics lists* tab.

Double-click on *<Add>* in the table, and assign a meaningful name to the text or graphics list. Select the list type under *Selection* which defines how the tag linked to the list is to be interpreted:

- ▷ As *Value/Range*, then the text or graphic is displayed if the tag value is within the range or ranges
- ▷ As *Bit (0, 1)*; a different text or graphic is then displayed for the signal states “0” and “1”
- ▷ As *Bit number (0 to 31)*; texts or graphics are then assigned to the individual bits.

Then define the list contents. Select the list and enter the values of the tags and the text or graphics in the *Entries in text/graphics list* table: Double-click on *<Add>* to open a drop-down dialog box which depends on the range selection for the text/graphics list:

- ▷ If *Value/Range* is selected, you can select *Single value* and enter the value or you select *Range* and specify the range; you can repeat this in the next lines with additional ranges or values
- ▷ If *Bit* is selected you can enter the texts/graphics for the signal states “0” and “1”
- ▷ If *Bit number* is selected you can enter the texts/graphics for the bit numbers 0 to 31

By activating *Standard* in the entries of the selection *Value/Range* and *Bit number*, you define the text or graphic to be output if the tag assumes an undefined value.

In the runtime settings under *Screens* you can define the response when several bits are set as the bit number: if the *Bit selection for text and graphics lists* check box is activated, the text or graphic that has been configured for a set bit with the least significance is shown. If the check box is not activated, the text or graphic that has been configured for a single set bit only is shown. If several bits are set, the text or graphic set as standard is then shown.



## Working with function lists

System functions are predefined functions which cannot be changed, for example for calculating the value of a tag, for setting a bit in the PLC station, or for changing the user.

An HMI object may have configurable events, e.g. a button has the *Click* event. A function list containing the system functions can be assigned to an event, which are then executed when the event occurs.

To configure a function list, select the object in the process screen and open the *Events* group in the *Properties* tab of the inspector window. Select an event, click in the function list on *<Add function>* and select the desired system function from the drop-down list.

It may be necessary to supply the system function with parameters. You can assign more than one system function to an event.

### 16.4.2 Working with alarms

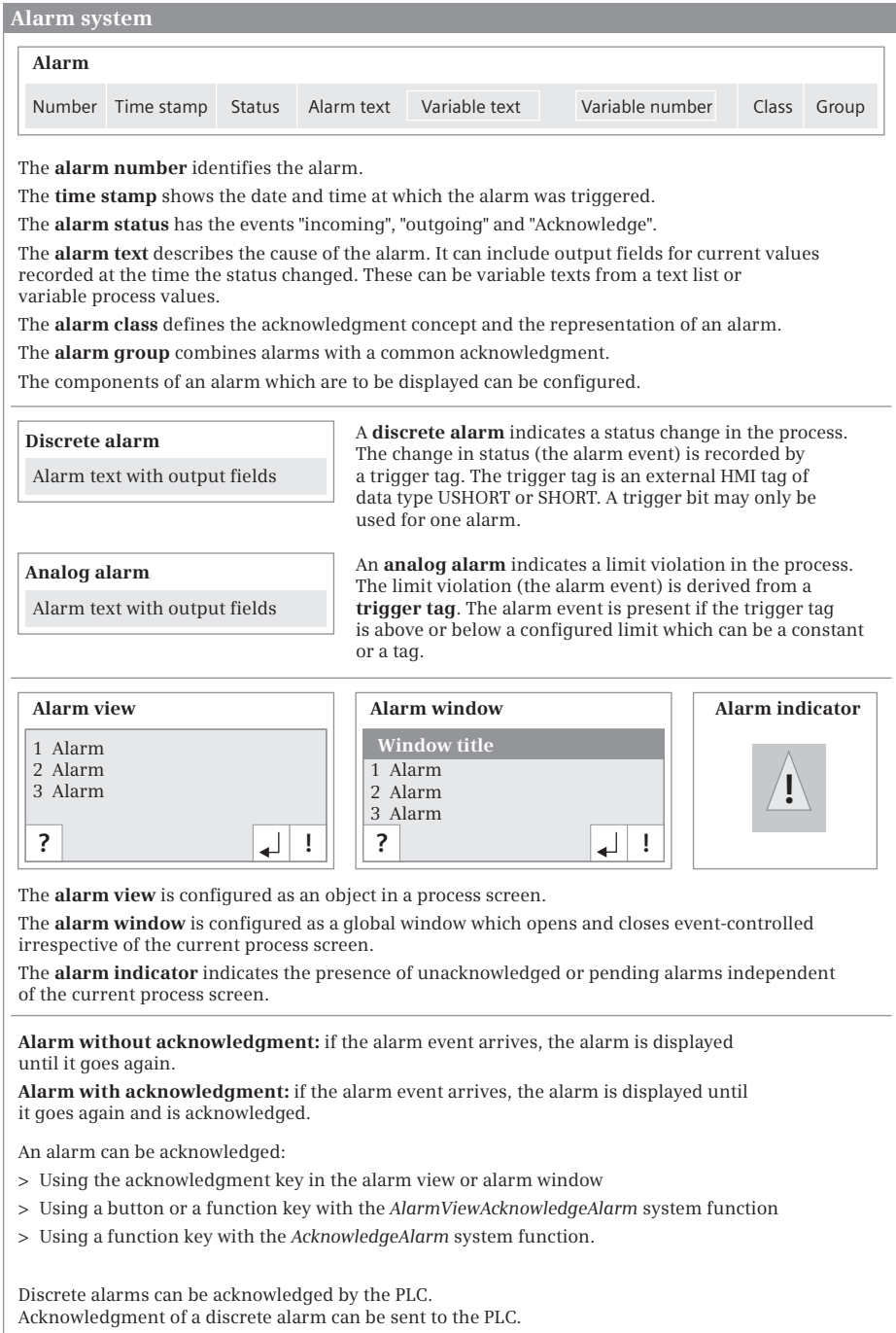
The alarm system distinguishes between *system-defined alarms* which do not require configuration and *user-defined alarms* which can be configured as analog alarms (display of limit violations) or discrete alarms (display of states or status changes).

System alarms indicate the status of the HMI station and of the communication between the HMI and PLC station. The type and number depend on the HMI station. You can set the display duration of system alarms in the project tree under *Runtime settings* and *Alarms*.

User-defined alarms visualize the process sequence and process states in the machine or plant. An alarm consists of an alarm number, the time of the event (date, time), the alarm text, the alarm status (incoming, outgoing), the alarm class, and possibly the alarm group (Fig. 16.10).

Alarms are assigned to certain *Alarm classes*: *System* (contains system-defined alarms for display of HMI station statuses), *Errors* (designed to display user-defined alarms for critical or dangerous statuses with an acknowledgment obligation) and *Warnings* (designed to display user-defined alarms for regular statuses without acknowledgment). User-defined alarm classes are configured with the desired representation and acknowledgment concept.

Alarms in an *Alarm group* are acknowledged together. Triggered alarm events are saved in an alarm buffer. The *Alarm view* shows selected alarm events from the alarm buffer in a process screen. If a new alarm is present, the *Alarm window* shows all pending alarms or alarms awaiting acknowledgment of a particular alarm class. With an incoming alarm, the *Alarm indicator* shows the defined alarm class, and whether unacknowledged alarms or alarms which have already been acknowledged but are still pending are still present.



**Fig. 16.10** Components of the alarm system

## Alarm statuses

Each alarm has an alarm status (Table 16.6):

- ▷ Incoming (I) the condition for triggering an alarm applies
- ▷ Outgoing (O): the condition for triggering an alarm no longer applies
- ▷ Acknowledge (A): the user has acknowledged the alarm.

The display text of the message statuses can be freely selected.

**Table 16.6** Alarm statuses

### Alarms without acknowledgment obligation

Display text	Status	Description
I	Incoming	The condition of an alarm applies.
IO	Outgoing	The condition of an alarm no longer applies.

### Alarms with acknowledgment obligation

Display text	Status	Description
I	Incoming	The condition of an alarm applies.
IO	Outgoing, not acknowledged	The condition of an alarm no longer applies. The user has not acknowledged the alarm.
IOA	Outgoing, then acknowledged	The condition of an alarm no longer applies. The user acknowledged the alarm after this point in time.
IA	Incoming, acknowledged	The condition of an alarm applies. The user has acknowledged the alarm.
IAO	Outgoing, but acknowledged first	The condition of an alarm no longer applies. The user acknowledged the alarm when the condition still applied.

## Acknowledging alarms

Alarms without an acknowledgment obligation indicate process states which are neither critical nor dangerous. Alarms without an acknowledgment obligation come and go without an acknowledgment.

Alarms with an acknowledgment obligation are used to ensure that the user has registered the occurrence of an alarm. By means of the acknowledgment, the user confirms that the status which triggered the alarm has been processed. The alarm is displayed until it has been acknowledged. Alarms with an acknowledgment obligation indicate critical or hazardous states.

An alarm can be acknowledged on the HMI station by an authorized user or automatically by the system, e.g. by means of a tag value, a system function in a function list, or the program in the PLC station.

Alarms combined in an alarm group are acknowledged together. An alarm group can contain alarms from different classes. Examples of alarm groups are those caused by the same fault or those from a particular machine unit or subprocess.

## Configuring alarms

The steps for configuring alarms are as follows:

- 1) Create and edit alarm class (representation and acknowledgment concept)
- 2) If required: Create alarm groups (combination of alarms into groups that are acknowledged together)
- 3) Create tags (define tags and limits which are to trigger alarms)
- 4) Create alarms (configure discrete and analog alarms, and assign the tags, alarm class, alarm group etc. to be monitored)
- 5) Configure the alarm output (create HMI objects in process screens, and assign alarms)
- 6) If required: Configure loop-in-alarm (following the arrival of an alarm, the process screen in which the alarm event is present is selected)

To configure alarms, double-click on the *HMI alarms* editor in the project tree under the HMI station.

### *Creating an alarm class*

Select the *Alarm classes* tab in the *HMI alarms* window. The table already contains the predefined alarm classes *Errors*, *System*, and *Warnings*. To enter a new alarm class, double-clicking on <Add new> in the next line. You can specify its properties in the inspector window as required, in particular the acknowledgment concept and the representation. You can configure enabling for the colored representation of the alarm classes in the project tree under *Runtime settings* > *Alarms* and *General* > *Alarm class colors*.

### *Creating an alarm group*

Select the *Alarm groups* tab in the *HMI alarms* window. Double-click on <Add new> in the table to enter a new alarm group whose name you can change in the table cell or inspector window.

### *Configuring a discrete alarm*

Select the *Discrete alarms* tab in the *HMI alarms* window (Fig. 16.11). To enter a new discrete alarm in the table, double-click on <Add new>. You can specify its properties in the inspector window as required.

In the *General* section you can assign the alarm to an alarm class as well as to an alarm group if required. Enter the event text which is to be displayed. You can insert tag output fields or text list output fields in the event text which display current values or selected texts when the alarm is output. To do this, use the right mouse button to select the position in the event text where the insertion is to take place, and then select the output field in the shortcut menu. Then define the tag and the output format or the text list and the selection tag in the dialog.

In the *Trigger* section, select the tag whose change in status is to trigger the alarm. To do this, enter an HMI tag with the data type INT or UINT or a PLC tag with the

The screenshot shows a software window titled "Project1200 > Gate handling [KTP600 Basic PN] > HMI alarms". It has four tabs: "Discrete alarms", "Analog alarms", "Alarm classes", and "Alarm groups". The "Discrete alarms" tab is active, displaying a table with the following data:

ID	Event text	Alarm class	Trigger tag	Trigger bit	HMI acknowledgment tag
1	Main gate is blocked (entry)	Errors	GateFault	0	HMI_BitAck
2	Main gate is blocked (exit)	Errors	GateFault	1	HMI_BitAck
3	Main gate is closes	Warnings	Data100_GateControl	0	<No tag>
<Add new>					

**Fig. 16.11** Configuring discrete alarms

data type INT or WORD and a bit number under *Settings*. Use this tag only as the trigger tag.

In the *Acknowledgment* section you can define the HMI tag which saves whether the alarm has been acknowledged. If the alarm is to be acknowledged by the PLC program, the trigger tag is an external HMI tag and simultaneously also the acknowledgment tag which you specify under *Acknowledgment* in the *PLC* field. Select different bits for triggering and acknowledging an alarm.

In the *Events* tab you define the functions to be executed as required for the *Incoming*, *Outgoing*, *Acknowledge*, and *Loop-in-alarm* events in one function list each. For the *Loop-in-alarm*, select the *ActivateScreen* system function and enter the screen name of a previously configured screen.

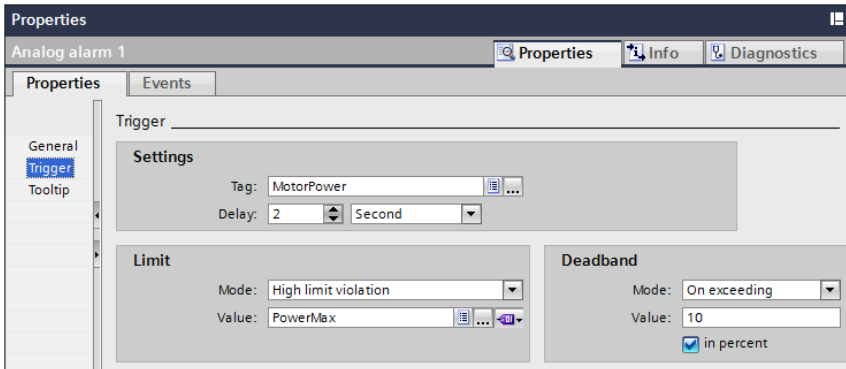
You can also configure a discrete alarm immediately when creating an HMI tag. Start configuration of tags using the *HMI tags* editor, open the *HMI tags* tab in the working window, select the tag in the top table, and configure the associated discrete alarm in the bottom table. The configured discrete alarm is imported into the alarm list of the alarm editor.

### Configuring an analog alarm

Select the *Analog alarms* tab in the *HMI alarms* window. To enter a new analog alarm in the table, double-click on <Add new>. You can specify its properties in the inspector window as required.

In the *General* section you can assign the alarm to an alarm class as well as to an alarm group if required. Enter the event text which is to be displayed. You can insert tag output fields or text list output fields in the event text to display current values or selected texts when the alarm is output. To do this, use the right mouse button to select the position in the event text where the insertion is to take place, and then select the output field in the shortcut menu. Then define the tag and the output format or the text list and the selection tag in the dialog.

In the *Trigger* section, specify the tag whose value is to be monitored for a limit, and define the delay time: the alarm is only triggered if the trigger condition is still present following expiry of the delay time. Specify under *Mode* when an alarm is to be triggered (High limit/low limit violation), and define the limit either as a



**Fig. 16.12** Properties of an analog alarm (trigger conditions)

constant or as a tag. Synchronization indicates how the limits are synchronized with this alarm with regard to the limits configured for the tag (configured using the tag editor, see next section). By means of a deadband (hysteresis) you can prevent repeated generation of the alarm if the tag value oscillates around the limit (Fig. 16.12).

In the *Events* tab you define the functions to be executed as required for the *Incoming*, *Outgoing*, *Acknowledge*, and *Loop-in-alarm* events in one function list each. For the *Loop-in-alarm*, select the *ActivateScreen* system function and enter the screen name of a previously configured screen.

#### *Configuring an analog alarm using the tag editor*

You can also configure an analog alarm immediately when creating an HMI tag. Start configuration of tags using the *Show all tags* editor, open the *HMI tags* tab in the working window, and select the tag in the top table. In the inspector window, enter high and low limits in the *Properties* tab under *Range* either as constants or tags. Configure the associated analog alarm in the bottom table in the working window. The configured analog alarm is imported into the alarm list of the alarm editor.

### **Configuring the alarm output**

In order to display alarms, the Basic Panels feature the alarm view as an object in a process screen, the alarm window, and the alarm indicator.

#### *Configuring an alarm view*

An alarm view is configured in a process screen. You can configure the alarm states (pending or unacknowledged alarms) and the alarm classes in this alarm view.

Open the process screen and drag the *Alarm view* object from the *Tools* task card under the *Control* category into the screen. With the alarm view selected, you can design its properties in the inspector window in accordance with your requirements. You can make various settings in the *Properties* tab:

- ▷ Under *General* you can select the alarm states to be displayed (pending and/or unacknowledged alarms) and activate the alarm classes to be displayed in this alarm view.
- ▷ Under *Display* you can define the buttons to be shown in the alarm view (infotext, acknowledge, loop-in alarm) and whether a scroll bar is to be present.
- ▷ Under *Layout* you can set the number of lines for an alarm and how many alarms are to be visible. You can also set the position and size of the alarm view here if you have not already done this with the mouse in the process screen. Use *Layout* to define the (background) colors and, under *Text format*, the font size to be used.
- ▷ Under *Columns* you can define the visible columns and the sorting of alarms.

### *Configuring an alarm window*

The alarm window shows the current alarms. It is opened independent of the current screen. The HMI station can still be used, even if alarms are present and displayed. An alarm window is configured in the global screen and displayed like an alarm view.

To configure the alarm window, double-click in the project tree under the HMI station in the *Screen management* folder on *Global screen*. The global screen opens. It contains as standard the alarm window for unacknowledged alarms at screen layer 1, for system alarms (active system events) at screen layer 1, and for pending (active) alarms at screen layer 3. You can also drag an additional alarm window from the *Tools* task card under the *Controls* category into the global screen.

In order to be able edit the alarm window better, position it in a separate screen layer, and hide the other layers (see Section 16.3.3 “Working with screen layers” on page 519).

With the alarm view selected, you can design its properties in the inspector window in accordance with your requirements. The settings are made as for the alarm view (see above). You can additionally select the properties of the window (Display automatically, Closable, Modal) and the inscription in the window title in the *Properties* tab under *Mode*.

### *Configuring the alarm indicator*

The alarm indicator indicates by means of a warning triangle that alarms are pending or require acknowledgment. The alarm indicator flashes if at least one unacknowledged alarm is present, and lights up continuously if at least one of the acknowledged alarms is not yet gone. The number of pending alarms is displayed.

To configure the alarm indicator, double-click in the project tree under the HMI station in the *Screen management* folder on *Global screen*. The global screen opens. It contains the warning triangle of the alarm indicator as standard. If the alarm indicator is not present, drag it from the *Tools* task card under the *Controls* category into the open window. Only one alarm indicator is permissible.

With the alarm indicator selected, you set its properties in the inspector window:

- ▷ You select the alarm classes for the indicator in the *Properties* tab under *General*. The alarms from these alarm classes activate the indicator during runtime. Specify whether pending alarms and/or alarms to be acknowledged are to be displayed by the indicator.
- ▷ In the *Events* tab you can define whether the *ShowAlarmWindow* system function is to be executed with *Click* or *Click when flashing*.

During runtime, the alarm indicator can only be accessed per touch screen.

### Configuring the acknowledgment of alarms

Whether and how an alarm is acknowledged is defined in the alarm class to which the alarm is assigned (see Section “Creating an alarm class” on page 531). A pending alarm can be acknowledged individually or in an alarm group. Acknowledgment can be performed in the following ways:

- ▷ Click the *Acknowledge* button in the alarm view or window.
- ▷ Configure a separate button or function key with the *AlarmViewAcknowledgeAlarm* system function for the *Click* or *Press* key event, and specify the alarm view whose alarm is to be acknowledged under *Screen object*.
- ▷ The acknowledgment of a discrete alarm by the PLC station, and the sending of an acknowledgment bit to the PLC station, are described in Section “Configuring a discrete alarm” on page 531.

If acknowledgment of alarms is only to be allowed for a limited circle of users, configure a button or function key with a corresponding operator authorization (see Section 16.4.4 “Working with the user administration” on page 539).

### 16.4.3 Working with recipes

Recipes include associated data, for example data for a certain production batch. A recipe consists of recipe data records. These differ in terms of their values, but not in their structure. A recipe data record consists of recipe elements (Fig. 16.13).

You can enter the recipe data – if known – during the configuration phase. Recipes are saved in the HMI station and entered, modified or deleted in the recipe view during runtime.

The displayed recipe data record can be transferred to the PLC station. A recipe data record can also be transferred from the PLC station to the HMI station. The trigger for this comes from a user or from the PLC station's program.

If there is a risk that recipe data could be mutually overwritten by the HMI and PLC stations, the transfer can be synchronized by means of the *Data record* area pointer.



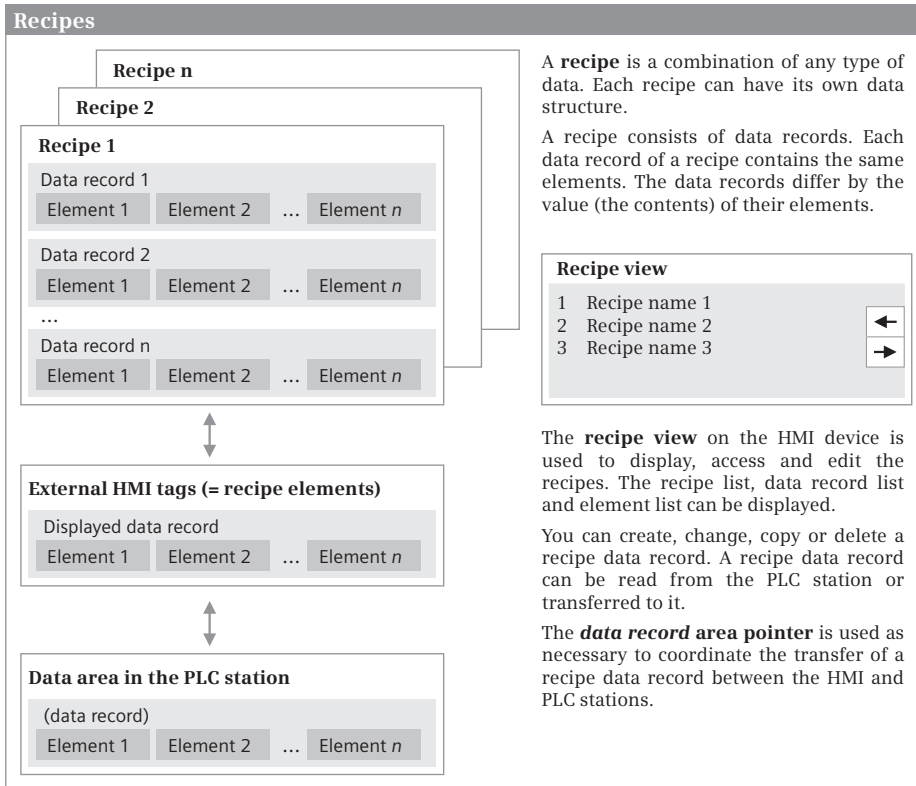


Fig. 16.13 Recipe components

### General procedure for configuring recipes

The structure of the recipe should be defined prior to the configuration of recipe management: What recipe elements are required per data record? How many data records does the recipe have? In which PLC tags are the values of the current data record to be saved?

Then create the PLC tags or the external HMI tags. Create a new recipe, define the recipe elements, and enter the data records with the values for each recipe element.

Configure a process screen with the recipe view. If the transfer of the current recipe data record to the PLC station is to be synchronized, create a *Data record* area pointer.

### Creating a recipe

To configure a new recipe, first create the PLC tags for a recipe data record. If you directly select the PLC tags during the configuration phase when assigning the PLC tags to the recipe data record elements, external HMI tags are created automatically. You can also create the external HMI tags in advance and then select them during configuration. Then create the new recipe, assign recipe elements to it, and enter the values for the elements in the data records.

Double-click on the *Recipes* editor in the project tree under the HMI station. The displayed recipe list has two parts: The top part *Recipes* lists the recipes. In the bottom part you can enter the elements of a recipe data record in the *Elements* tab and in the *Data records* tab you can define the number of data records in the recipe.

Create a new recipe in the *Recipes* table. In the properties of the recipe in the inspector window, you define under *Synchronization* whether data exchange is to be synchronized and via which HMI connection this is to be carried out. If the *Coordinated data transfer* checkbox is activated, the *Data record* area pointer must be configured.

Enter the recipe elements in the *Elements* table. A double-click on <Add> creates a line with a new element. In the *Tag* column, assign either an external HMI tag or a direct PLC tag to the recipe element. During runtime, these tags save the value of the recipe element in the current recipe data record. The value in the *Default value* column is used as the default entry when creating a data record. A recipe element can be assigned a text list whose text, depending on the value of the recipe element, is displayed during runtime in an output field. Enter as many elements as present in a recipe data record.

In the *Data records* table, define the values of the recipe elements for each individual data record. A double-click on <Add> creates a line with a new data record with a column for each recipe element. Enter the values for the recipe elements here.

### **Configuring a recipe view**

The recipe view is an off-the-shelf display and control element used to manage recipe data records.

To configure the recipe view in a process screen, use the mouse to drag the recipe view from the *Tools* task card under the *Controls* category into the open screen.

If you only wish to display a particular recipe in the view, enter the recipe under *General* in the properties of the recipe view in the inspector window. You can specify a tag in the *Recipe data record* section in which – depending on the tag's data format – the number or name of the currently displayed recipe data record is to be saved. If you deactivate the *Editing mode* check box, you can suppress the editing of recipe data, and the data is only displayed.

You define the display of the recipe data under *Simple view*. Under *Toolbar* you can define which buttons are to be available in the recipe view during runtime.

### **Operating the recipe view during runtime**

You can display the list of all existing recipes, the data record list of a recipe, and the element list for a data record in the recipe view. The view always begins with the recipe list. Depending on the configuration, you can create a new recipe data record, modify, copy or delete an existing record, transfer a record to the PLC station, or read a record from the PLC station.

The values of a data record modified in the recipe view must be transferred to the PLC station to enable them to be processed there. To carry out the transfer, open the desired recipe and the elements list of the data record whose values you wish to transfer. Select the *Down* command in the shortcut menu.

In order to read the values of a recipe data record from the PLC station, open the recipe and the elements list of the data record whose values are to be imported from the PLC station. Select the *Up* command in the shortcut menu. The new values are displayed.

You can also assign the commands of the recipe view to another screen object. For example, you can assign the *RecipeViewSetDataRecordToPLC* system function to a button or function key.

### **Controlling transfer from PLC station's program**

Using the *Job mailbox* area pointer you can read a recipe data record from the PLC station (job number 69) or write one into the PLC station (job number 70). Working with area pointers is described in Section 16.2.3 “Creating an area pointer” on page 515. The first parameter of the area pointer contains the recipe number, the second parameter the data record number. During transfer from the PLC station to the HMI station (job number 69) you can specify in the third parameter whether the existing data record is to be overwritten (with 1) or not (with 0).

The recipe view is not updated automatically. For example, if recipe data is transferred from the PLC station while the recipe is being displayed, the new values are only displayed when the associated data record is selected again.

### **Synchronized transfer to the PLC station**

There are two possibilities for the transfer: transfer without synchronization and transfer with synchronization via the *Data record* area pointer. You can use transfer without synchronization if the transfer of a data record is to be exclusively triggered by an operator input on the HMI station. Synchronization is necessary if the transfer is triggered from both the HMI station and PLC station. This prevents uncontrolled mutual overwriting of data.

If the initiative for transfer comes from the HMI station, it enters a value of 2 into the status word, changes the recipe and data record numbers, reads or writes the data record values, and sets the status word to a value of 4. The PLC station must set the status word to 0 in order to enable the next transfer.

If the initiative comes from the PLC station, this must trigger the HMI station to transfer by means of control job no. 69 or no. 70 (as described above). The further sequence is as with the initiative from the HMI station.

A bit in the status word may only be set by the HMI station. The PLC station may only reset the status word back to 0.

Data record area pointer		
The data record area pointer is used to synchronize the transfer of a recipe data record. It can be created in the PLC station as a tag of data type ARRAY [1..5] of UINT.		
<b>Data record area pointer</b>		
Word 1	Current recipe number (1 ... 999)	
Word 2	Current data record number (1 ... 65 535)	
Word 3	Reserved	
Word 4	Status (0, 2, 4, 12)	
Word 5	Reserved	
<b>Value of status word</b>		
Decimal	Binary	Meaning
0	0000 0000	Transfer permissible
2	0000 0010	Transfer busy
4	0000 0100	Successfully completed
12	0000 1100	Completed with errors

**Fig. 16.14** Structure of the *Data record* area pointer

#### 16.4.4 Working with the user administration

You configure user groups, users, and authorizations with the user administration. Authorizations restrict security-related operations to specific user groups. To do this, set up access protection for the operating element; then only a user assigned to a user group with the relevant access rights can perform the operation using this object (Fig. 16.15).

Examples of user groups with different privileges include: *Administrators* or *Service engineers* who have unlimited access, *Technicians* who are permitted to make settings in the process or on the machine, and *Operators* who are responsible for the production process.

#### Configuration procedure

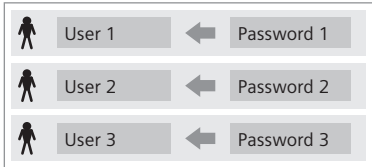
- 1) User groups are defined during the configuration phase. If users are already known at this point in time, they can be included in a user group. Users can also be assigned to the user group during runtime by means of the *User view*.
- 2) Define privileges, and assign the corresponding privileges to each user group.
- 3) During configuration of the operator-accessible object, set the privilege with which this object can be accessed in the properties under *Security*. This means that a user can only operate this object during runtime if he or she is included in the corresponding user group.

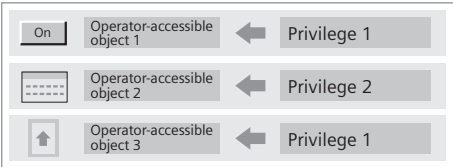
#### Configuring users and user groups

To configure the user administration, double-click in the project tree under the HMI station on *User administration* and select the *User groups* tab in the working window. The *Administrator group* and *Users* user groups are always present in the top

### User administration

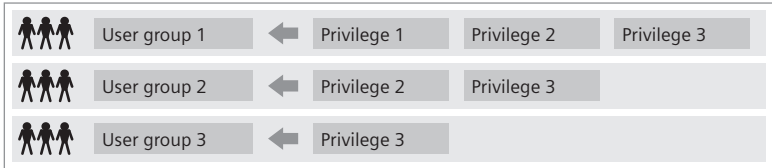
An HMI station is provided with **access protection** which protects against unauthorized operation during runtime. Safety-related operations can be restricted to special user groups.





Each **user** has a password.  
 Each user is assigned to a user group.  
 A user can have any name.

A **privilege** is assigned to each operator-accessible object during configuration.  
 Different objects can have the same privileges.  
 A privilege can have any name.



Privileges are assigned to each **user group**. Several privileges can be assigned to a user group.  
 A user group can have any name.

If a user logs in with his or her password during runtime, the user group to which he or she is assigned allows him or her to access those objects which have the same privileges.

**User view**

User 1	User group 1	▲
User 2	User group 2	▲
User 3	User group 3	▼
		▼

The user view is used to administer the users during runtime (create and delete users, assign privileges).

**Fig. 16.15** User administration elements

table *Groups*. The privileges *Operate*, *User administration* and *Monitor* are always present in the lower *Privileges* table.

To enter a new privilege, double-click on <Add> in the *Privileges* table. You can set the privilege properties in the inspector window. The name and number must be unique in the HMI station. The name of a privilege is freely-selectable, but should indicate the access privilege. The name is displayed in the user administration. Explain the privilege in the comment field.

To enter a new user group, double-click on <Add> in the *Groups* table. Set the user group properties in the inspector window. The name and number must be unique in the HMI station. The name of a user group is freely-selectable, but should indicate the group characteristics. The name is displayed in the user administration. Describe the user group in the comment field.

To assign privileges to a user group, select the user group and activate the corresponding privileges in the *Active* column in the *Privileges* table.

In the *Users* tab, configure the users in the top table *Users*. One user *Administrator* is already present. Double-click on <Add> and enter the properties of the next user in the inspector window. The name and number must be unique in the HMI station. Assign a password and confirm it. A user can change his or her own password during runtime.

To assign the user to a user group, select the user and then the user group in the *Member of* column in the *Groups* table. You can assign a user to exactly one user group. In the inspector window, you can set in the user properties under *Automatic logoff* the number of minutes after which automatic logging off is to take place.

### **Configuring access protection for control elements**

The privileges created in the user administration must be assigned to the control elements protected against unauthorized access. Objects with access protection are the date/time field, the I/O field, the graphic and symbolic I/O fields, the switch, the button, and the recipe view.

In order to configure access protection for a control element, open the process screen and select the control element. In the inspector window, select the privilege under *Properties* and *Security*, and define whether operator access is permissible.

### **Configuring the user view**

The user view is used to configure and administer users and privileges during runtime. The user view is configured in a process screen. Open the screen and use the mouse to drag the user view from the *Tools* task card under *Controls* into the screen. Set the user view properties in the inspector window.

### **Runtime settings for user administration**

Use the *Runtime settings* editor under an HMI station in the project tree to configure the security settings of the user administration during runtime. Start by double-clicking on the editor, and select the *User administration* section in the runtime settings.

Under *General* you set the number of permissible invalid login attempts by a user before the user is assigned to the *Unauthorized* group. If the *Logon only with password* checkbox is activated, users are not required to enter a user name when logging in.

Under *Hierarchy level* you can activate the group-specific privileges for the user administration. This means that an administrator can only administer those users during runtime whose group number is smaller than or equal to his or her own number, and only assign a user to a user group whose number is smaller than or equal to his or her own group number.

Under *Password*, you can activate the password aging. You can set the number of days for which a password is valid, and the preliminary warning time before a change in password becomes necessary. *Password generation* is understood to be the number of past passwords before a certain password can be repeated. With password aging activated, the *Password aging* column can be edited for the user groups (*User administration* editor in the *Users* tab and *Groups* table).

Under *Password complexity* you can set the minimum password length and also specify whether a password must contain digits and/or special characters.

## 16.5 Completing HMI configuration

### 16.5.1 Compiling the HMI configuration (Consistency test)

The entered configuration data must be compiled so that execution in the HMI station is possible. During configuration the data is compiled continuously in the background. Further compilation takes place automatically when downloading to the HMI station.

You can also compile the configuration data in between times in order to check its consistency. The faulty positions are listed in the inspector window and can be directly selected by double-clicking on the error message in order to eliminate the error.

To compile the entire configuration data for an HMI station, select the station and then the *Compile > Software (rebuild all)* command in the shortcut menu. The *Compile > Software* command is used to compile only the configuration data which has been changed since the last compilation. If the PLC tags were changed during compilation of the HMI device, or if new ones have been added, it is advisable to also compile the corresponding data blocks in the PLC station prior to compilation of the HMI configuration.

### 16.5.2 Simulation of HMI configuration

You can use the simulator on the programming device to test the response of the HMI configuration. The simulation can be carried out together with the PLC station or with the tag values of a simulation table, in the latter case without a PLC station.

To carry out the simulation, select the HMI station in the project tree and then the *Online > Simulation > ...* command from the main menu.

The command *Online > Simulation > Start* starts the simulation with the networked PLC station. The simulator program establishes a connection to the active PLC station and imports its tag values for the simulation. The configured process screens are displayed in the simulator window. Use the mouse for operations instead of the touch screen and function keys.

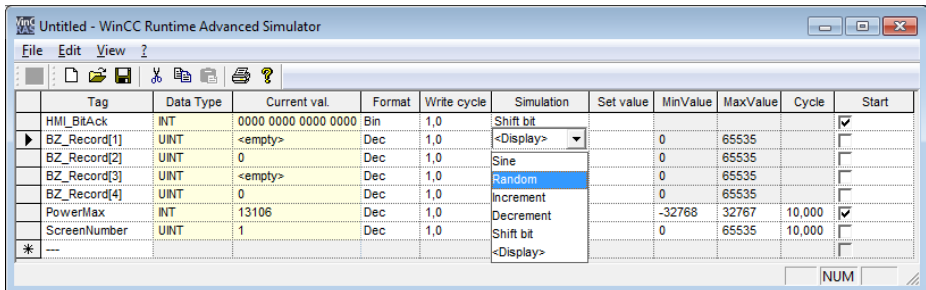
The command *Online > Simulation > With tag simulator* starts the simulation with simulation table. Two windows are opened: simulation of the HMI configuration

(display of the process screens) and the simulation table. Use the <Alt + Tab> key combination to switch between the two windows.

Enter all tags in the simulation table with which you wish to simulate the HMI configuration. The *Format* column defines the display format, the *Set value* column the start value. In the *Simulation* column, select the type of simulation, e.g. *Increment*, from the drop-down list. Enter the value by which the tag value is to change in the *Cycle* column. Define the rate of change in the *Write cycle* column: the value entered corresponds to the interval in seconds with which the tag value is to be changed. In the *MinValue* and *MaxValue* columns you can define the range of values for tags. In order to carry out simulation with a tag, activate the *Start* check box.

Depending on the data type of the tag, the simulator provides six different modes (Fig. 16.16):

- ▷ *Sinus*: changes the value in the form of a sinusoidal curve
- ▷ *Random*: provides randomly generated values
- ▷ *Increment*: increases the tag value commencing with the start value up to the maximum value, and commences again with the minimum value
- ▷ *Decrement*: decreases the tag value commencing with the start value down to the minimum value, and commences again with the maximum value
- ▷ *Shift bit*: shifts a set bit continuously by one position
- ▷ <Display>: shows the current tag value in static form



Tag	Data Type	Current val.	Format	Write cycle	Simulation	Set value	MinValue	MaxValue	Cycle	Start
HMI_BitAck	INT	0000 0000 0000 0000	Bin	1,0	Shift bit					<input checked="" type="checkbox"/>
BZ_Record[1]	UINT	<empty>	Dec	1,0	<Display>		0	65535		<input type="checkbox"/>
BZ_Record[2]	UINT	0	Dec	1,0	Sine		0	65535		<input type="checkbox"/>
BZ_Record[3]	UINT	<empty>	Dec	1,0	Random		0	65535		<input type="checkbox"/>
BZ_Record[4]	UINT	0	Dec	1,0	Increment		0	65535		<input type="checkbox"/>
PowerMax	INT	13106	Dec	1,0	Decrement		-32768	32767	10,000	<input checked="" type="checkbox"/>
ScreenNumber	UINT	1	Dec	1,0	Shift bit		0	65535	10,000	<input type="checkbox"/>
* ---					<Display>					<input type="checkbox"/>

Fig. 16.16 Example of a simulation table

You can save the settings of the simulation table using the menu command *File > Save*. Several simulation tables with different contents can be saved. Use *File > Open* to open a saved simulation table again.

### 16.5.3 Downloading configuration to the HMI station

You have created a project with a PLC and an HMI station. The PLC and HMI station are networked and prepared for data exchange with at least one HMI connection. The user program in the PLC station is complete, compiled, and downloaded to the PLC station (Chapter 13.2 “Transferring project data” on page 425). Configuration



of the HMI station has been completed, the configuration data has been compiled, and – as far as possible – successfully simulated.

### **Preparation for loading the configuration data**

The standard settings for loading are defined in the properties of the HMI station. Select the HMI station in the project tree and then the *Properties* command in the shortcut menu. The settings which you already made when networking the PLC station are present in the properties window in the *Ethernet addresses* section.

You can assign an IP address online to an HMI station if you start the *Online & diagnostics* editor in the project tree under the HMI station. You set the IP address and subnet mask in the working window under *Functions* and *Assign IP address*. Use the *Assign IP address* button to transfer the settings to the HMI station.

### **Checking the connection data on the HMI station**

You can check the connection settings on the HMI station in the Control Panel. If you start the HMI station (connect the power supply) the loader window is displayed during booting. If configuration data is already loaded and the root screen is displayed, terminate the runtime (the current program on the HMI device) using a correspondingly configured function key or using the system screen *Different jobs* in order to display the loader window.

Click in the loader window on the *Control Panel* button. Briefly click twice to reach the next dialogs in the Control Panel: *Profinet* and *Transfer* contain the network settings, e.g. the IP address. To change an entry, click on the associated field and enter the new value using the on-screen keyboard which appears. Exit the *Control Panel* by clicking on the exit cross at the top right in the title bar.

### **Downloading configuration data**

To download, select the HMI station in the project tree and choose the command *Download to device > Software (all)* from the shortcut menu. In the *Extended download to device* dialog, you can specify the IP address of the HMI station to be loaded to (Fig. 16.17).

The configuration data is compiled on downloading. Displayed warnings do not prevent downloading; the function may be restricted at runtime. If there are errors, the configuration data is not loaded. Clear any faults before trying to download again.

After successful compilation, the downloading preview is displayed. You see the individual loading steps and can change specific settings as required. If the compilation is completed without errors – and you have activated the settings to overwrite the user administration and recipe data, if required – click on the *Load* button. The messages that appear during loading are displayed in the inspector window in the *Info* tab.

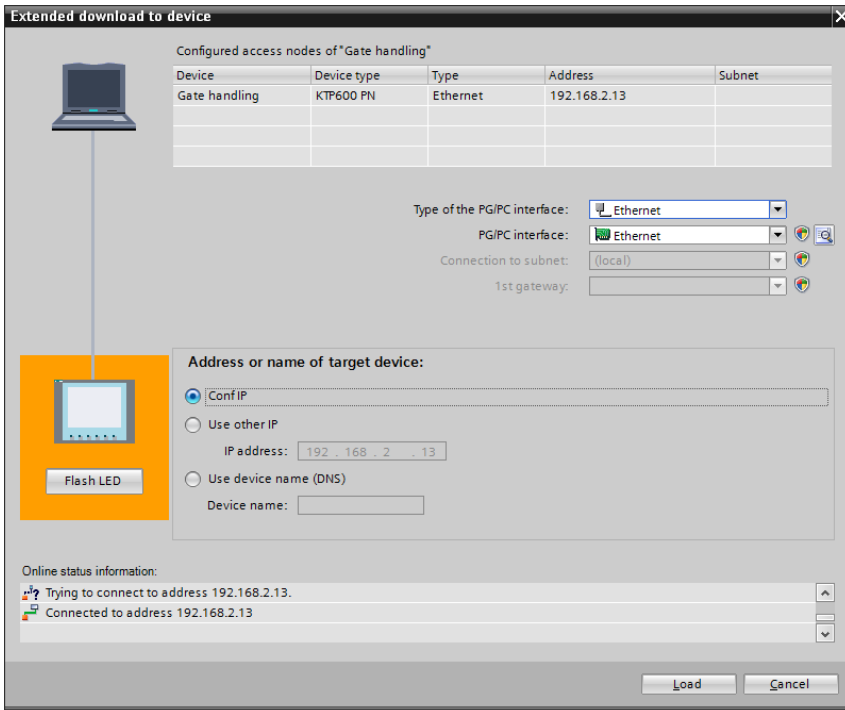


Fig. 16.17 Downloading to an HMI station

Any program running on the HMI station is exited when downloading is started, and the transfer window appears displaying the progress. Following successful downloading, the operating program is started automatically (if so configured).

The loader window is displayed in the case of an HMI station which is not in runtime, for example following startup without configuration data. In this case you must prepare the HMI station for downloading: click the *Transfer* button on the HMI station before you continue downloading by means of the *Download* button on the programming device. The HMI station then shows the downloading progress in the transfer window.

### Updating the operating system during downloading

When downloading the configuration data, a check is carried out to establish whether the configured operating system version of the HMI station agrees with the version upon which the configuration software is based. If this is not the case, it is possible to immediately update the operating system version during downloading.

Note that the recipe and user data in the HMI station could be inadvertently deleted when updating the operating system. Save this data if necessary, and then start the download procedure again with updating of the operating system. Restore the saved data after updating.

Saving and restoring are described in the next Section 16.5.4 “Maintenance of the HMI station”. Updating of the operating system can also be carried out independent of downloading.

### Starting the HMI station

Following successful downloading of the configuration data, the runtime starts following a certain delay, if so configured. You can configure the delay time on the HMI station: click in the loader window on the *Control Panel* button, double-click in the Control Panel on *OP*, and set the delay time in seconds in the *OP properties* window in the *Display* tab (click on the input field).

The runtime also starts if you click on the *Start* button in the loader window.

### 16.5.4 Maintenance of the HMI station

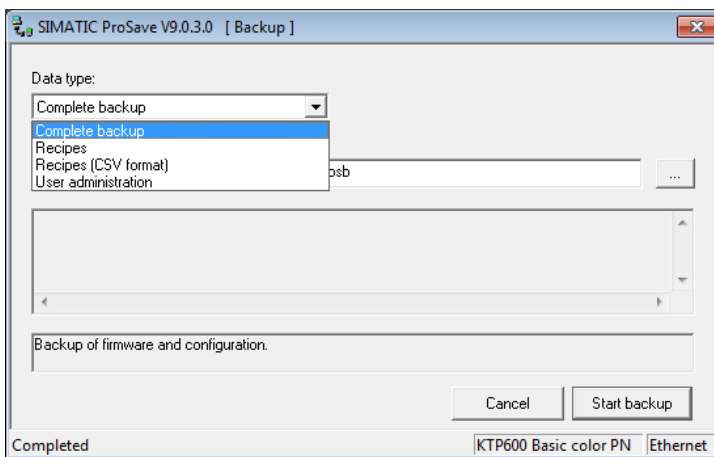
Maintenance of the HMI station comprises the following functions:

- ▷ Save configuration data of HMI station
- ▷ Restore configuration data of HMI station
- ▷ Authorization/licensing (does not apply to Basic Panels)
- ▷ Update operating system
- ▷ Options (does not apply to Basic Panels)

To start a maintenance function, select an HMI station and use the main menu command *Online > Device maintenance > ...* in the project tree.

### Data backup

Using a connected programming device you can save the following HMI station data: Complete backup (all data including operating system), recipes, and user administration (Fig. 16.18).



**Fig. 16.18** Saving configuration data and operating system

Start STEP 7 Basic, open the project with the HMI station whose data is to be saved, and select the HMI station in the project tree. Check that the settings for downloading are correct in the properties of the HMI station (in the *Properties* shortcut menu, and then select the *Download* section).

In the main menu, select *Online > HMI device maintenance > Backup*. Perform the save operation for the desired data type. To restore the data, select *Online > HMI device maintenance > Restore* in the main menu.

### **Updating the operating system of the HMI station**

When updating the operating system, all data on the HMI station is deleted. You must save the data for recipes or user administration if you wish to use them again following updating of the operating system.

Start the operating system update by selecting an HMI station in the project tree and choosing *Online > HMI device maintenance > Update operating system* from the main menu.

A connection to the HMI station is established using the *Device status* button. The relevant device data is displayed: the boot loader and operating system versions as well as the sizes of the flash and RAM memories.

If you wish to reset the HMI station to the default settings, e.g. because updating of the operating system was interrupted, activate the *Reset to default settings* check box in the ProSave window.

# 17 Appendix

## 17.1 Integral and technological functions

### 17.1.1 High-speed counter (HSC)

Contrary to a counter function, a high-speed counter (HSC) counts pulses independent of the cycle time of the user program. A counting frequency up to 200 kHz is possible. The counting range corresponds to the range of values of a DINT tag (–2 147 483 648 to +2 147 483 647).

A high-speed counter is an integral component of the CPU and must be activated and configured prior to use. The number of available high-speed counters depends on the CPU: three counters for the CPU 1211, four counter for the CPU 1212, and six counters for the CPU 1214 and CPU 1215. The number of high-speed counters with the CPU 1211 and CPU 1212 can be increased by using a signal board with digital input channels.

A high-speed counter can be used as a single-phase or two-phase counter (A/B quadrature). Please note that – depending on the operating mode – specific input channels are permanently assigned to a high-speed counter and that this may result in limitations when using the counter. The assignment of the counter inputs affects the *peripheral inputs* (the input terminals). A change in the (logical) input addresses has no influence on this assignment.

With correspondingly designed signal boards, the maximum counting frequency of 100 kHz achievable with the onboard inputs of the CPU can be increased to 200 kHz.

The data required for execution of a counter function is saved in a data block. When calling the counter function as a single instance, this is a separate data block per call, when calling as a local instance in a function block, the instance data block of the function block can be used for data saving (multi-instance).

### Assignment of inputs to high-speed counters

Depending on the operating mode, specific inputs integrated on the CPU or on the signal board are assigned to a high-speed counter (Table 17.1). If a counter uses the inputs, they cannot be used elsewhere. Unused inputs can be used like “normal” inputs for other purposes. Inputs used by a high-speed counter cannot be forced (assigned a fixed value).

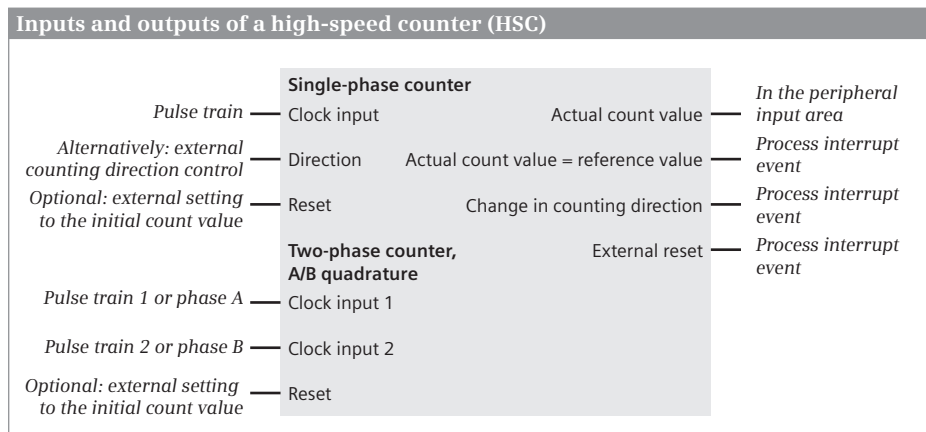
As the table shows, the assignable inputs of HSC 1 and HSC 2 (I 0.1, I 0.3) as well as of HSC 3 and HSC 4 (I 0.5, I 0.7) overlap, meaning that not all counters can be used in every mode. If both counters are to be used in each case, it is possible to change

**Table 17.1** Assignment of onboard inputs to counters (without signal board)

CPU input	HSC No.	For single-phase counter mode	For two-phase counter mode	For A/B quadrature mode
I 0.0	HSC 1	<b>Clock input</b>	<b>Clock input up</b>	<b>Clock phase A</b>
I 0.1	HSC 1 HSC 2	Direction Reset	Clock input down Reset	Clock phase B Clock phase Z
I 0.2	HSC 2	<b>Clock input</b>	<b>Clock input up</b>	<b>Clock phase A</b>
I 0.3	HSC 1 HSC 2	Reset Direction	Reset Clock input down	Clock phase Z Clock phase B
I 0.4	HSC 3	<b>Clock input</b>	<b>Clock input up</b>	<b>Clock phase A</b>
I 0.5	HSC 3 HSC 4	Direction Reset	Clock input down Reset	Clock phase B Clock phase Z
I 0.6	HSC 4	<b>Clock input</b>	<b>Clock input up</b>	<b>Clock phase A</b>
I 0.7	HSC 3 HSC 4	Reset Direction	Reset Clock input down	Clock phase Z Clock phase B
I 1.0	HSC 5	<b>Clock input</b>	<b>Clock input up</b>	<b>Clock phase A</b>
I 1.1	HSC 5	Direction	Clock input down	Clock phase B
I 1.2	HSC 5	Reset	Reset	Clock phase Z
I 1.3	HSC 6	<b>Clock input</b>	<b>Clock input up</b>	<b>Clock phase A</b>
I 1.4	HSC 6	Direction	Clock input down	Clock phase B
I 1.5	HSC 6	Reset	Reset	Clock phase Z

back to the inputs of the signal board or adapt the operating modes. For example, control of the counting direction and resetting can also be carried out with the CTRL\_HSC statement as an alternative to the external inputs.

Fig. 17.1 shows which inputs and outputs are used by a high-speed counter depending on the operating mode.

**Fig. 17.1** Inputs and outputs of a high-speed counter

## Operating modes of a high-speed counter

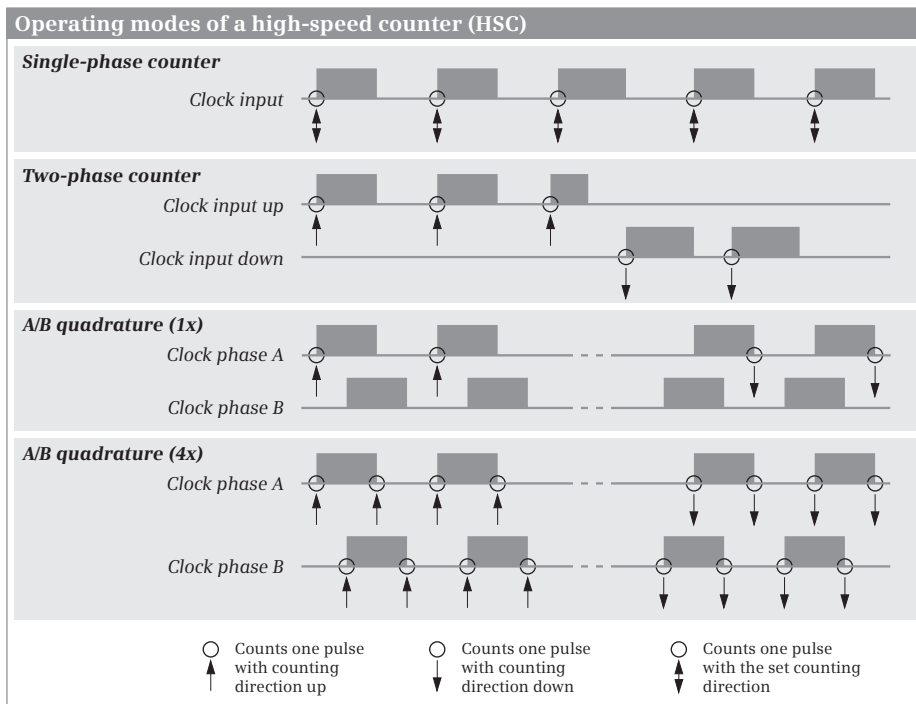
Fig. 17.2 shows the possible counting modes of a high-speed counter.

*Single-phase counter:* the counter is controlled by a single pulse train. The counting direction is specified either internally with the CTRL\_HSC statement or externally via an input. Each rising edge of a pulse increases or decreases the count value depending on the actual counting direction.

*Two-phase counter:* The counter is controlled by two independent pulse trains, one for the up counting direction and one for the down counting direction. Each rising edge of a pulse increases or decreases the count value depending on the pulse train. Please note that with the first pulse of the other input in each case, there is a change in the counting direction and therefore – if activated – the associated process interrupt will be triggered.

*A/B quadrature with single speed:* The counter is controlled by two pulse trains offset by 90° (“Phase A” and “Phase B”). If the pulse train of phase B has signal state “0” (“between” the pulses), counting is enabled: then each rising edge of the pulses of phase A results in an increase in the count value, and each falling edge a decrease.

*A/B counter with quadruple speed:* The counter is controlled by two pulse trains offset by 90° (“Phase A” and “Phase B”). Each edge of each pulse train is counted.



**Fig. 17.2** Operating modes of a high-speed counter

### Functional principle of a high-speed counter

A high-speed counter can be operated with three counting modes: counting, frequency meter or motion axis.

Pulses are counted in the *Counting* mode, either from one or two pulse trains depending on the operating mode. With only one pulse train for the counter input, the counting direction is controlled either internally or externally by an input. The initial count value and a reference value can be defined. The count value can be reset to the initial value via an input or per program. A fast counter in *Counting* mode delivers the actual count value as well as interrupt events if the actual count value is equal to the reference value, if the counting direction is changed, and if the counter is reset externally.

The number of changes in the count value per time interval is counted in the *Frequency* mode. The frequency value output is a mean value over the time interval. With only one pulse train for the counter input, the counting direction is controlled either internally or externally by an input. The measuring period (the time interval) can be defined. A fast counter in *Frequency* mode delivers the mean value of the frequency over the measured period as well as interrupt events if the actual count value is equal to the reference value, if the counting direction is changed, and if the counter is reset externally.

In the *Motion axis* mode, the high-speed counter is used by the technological object *Axis* and therefore cannot be used for other purposes (applies to counters HSC 1 and HSC 2).

### Actual count value

The current count value is not made available as a block parameter at the counter box, but is stored by the counter in the process image input (Table 17.2). Note, however, that the actual count value is no longer up-to-date when it is read and processed by the user program if the counter is counting at high speed.

**Table 17.2** Memory addresses for the current count values of the high-speed counters

High-speed counter HSC	1	2	3	4	5	6
Current count value in	%ID1000	%ID1004	%ID1008	%ID1012	%ID1016	%ID1020

A comparison between the actual count value and a “target value” can be carried out indirectly by loading the “target value” as a reference value and – when the “target value” has been reached – evaluating the attainment of the “target value” in the interrupt routine (react accordingly) and loading a new reference value (the next “target value”).

The mean value of the frequency is output in Hertz (changes per second) in the *Frequency* mode, independent of the time interval of the measuring period.



## Configuring a high-speed counter

A high-speed counter must be activated using the hardware configuration editor. In order to configure a high-speed counter, start the *Device configuration* editor under the PLC station in the project tree. Select a high-speed counter in the properties of the CPU module in the inspector window, and activate it using the *Enable this high-speed counter for use* check box.

You set the counting mode (counting, frequency, or axis of motion) under *Function* and under *Operating phase* you set the manner in which the count pulses are to be made available and counted (single-phase, two-phase, A/B counter 1X, A/B counter 4X). If a signal board is plugged in, you can select the input source (integrated CPU input, signal board input) for the HSC 1, HSC 2, HSC 5, and HSC 6 counters. The inputs of a counter (counter input, direction input, reset input) are either all on the CPU or all on the signal board. If you have selected *Single-phase* for the operating phase, set under *Counting direction is specified by* whether the counting direction is to be defined by the user program (internal direction control) or by an input (external direction control). Define the initial counting direction.

Under *Reset to initial values* you can define the initial count value and the initial reference value and define whether resetting to these values is to be carried out by an external input and at which signal level. Resetting is active for as long as the set signal level is present. You can change the initial count value and the reference value during runtime using the CTRL\_HSC statement.

Under *Event configuration* you can set the event at which the counter is to generate a process interrupt, as well as the name of the interrupt event. The events are: *Generate interrupt for counter value equals reference value event*, *Generate interrupt for external reset event* and *Generate interrupt for direction change event*. Provide the event with a name, and assign a process interrupt organization block to it. The *HW interrupt* drop-down list contains all previously created organization blocks with the start event *Hardware interrupt*. You can create a new process interrupt OB using the *Add object* button.

*Hardware inputs* lists those inputs occupied by counters, together with the maximum achievable counting frequency.

Under *I/O addresses/HW identifier* you can set the addresses of the peripheral inputs at which the counter is to output the actual count value. In the *Process image* drop-down list you can set whether the transfer is to be to the input process image (*Cyclic PI*). The hardware ID of the counter is also specified in this tab (*HW ID*); use this for assigning the configured counter to the CTRL\_HSC statement on the HSC parameter.

## CTRL\_HSC statement

The CTRL\_HSC statement controls a high-speed counter. CTRL\_HSC can be found in the program elements catalog under *Technology* and *Counters*. To call the statement, drag CTRL\_HSC with the mouse into the open block. Each CTRL\_HSC call

requires an instance data record, which can be either in a separate block (single instance) or – if the call is made in a function block – in the instance data block of the calling function block (multi-instance).

The CTRL\_HSC is executed if “1” is present at the enabling input or if “current” flows into the EN input or if EN is not connected. The enabling output ENO is then “1”. If execution of the function is not enabled (EN = “0”) or if an error occurs during execution of the statement, ENO is set to signal state “0”.

The HSC ID can be found either in the *System constants* tab in the default tag table or the properties of the CPU in the *High-speed counters (HSC)* group under the activated and applied counter under *General* and *Name*. At the HSC parameter, enter this name or select it from the drop-down list. You can also specify the numerical value of the HSC ID, which is shown in the *System constants* tab or in the counter properties under *Hardware ID*, as a constant or variable.

With signal state “1”, the DIR parameter sets the counting direction which is specified on the NEW\_DIR parameter (+1 = up, -1 = down). Signal state “0” on the DIR pa-

**Controlling a high-speed counter (HSC)**

*Instance data*

CTRL_HSC	
— HSC	BUSY —
— DIR	STATUS —
— CV	
— RV	
— PERIOD	
— NEW_DIR	
— NEW_CV	
— NEW_RV	
— NEW_PERIOD	

A **high-speed counter (HSC)** is a function integrated in the CPU module. A high-speed counter permits the counting of pulses and a frequency measurement up to 100 kHz or 200 kHz.

The CTRL\_HSC statement controls a high-speed counter. Calling of CTRL\_HSC requires an instance data record which can be either in a separate data block (single instance) or in the instance data block of the calling function block (multi-instance).

Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
HSC	INPUT	HW_HSC	HSC ID (1 ... 6, is created when activating the HSC)
DIR	INPUT	BOOL	Set a new counting direction (with signal state “1”)
CV	INPUT	BOOL	Set a new count value (with signal state “1”)
RV	INPUT	BOOL	Set a new reference value (with signal state “1”)
PERIOD	INPUT	BOOL	Set a new time interval (with signal state “1”)
NEW_DIR	INPUT	INT	New counting direction (+1 = up, -1 = down)
NEW_CV	INPUT	DINT	New count value
NEW_RV	INPUT	DINT	New reference value
NEW_PERIOD	INPUT	INT	New time interval
BUSY	OUTPUT	BOOL	Job being processed (with signal state “1”)
STATUS	OUTPUT	WORD	Job status

**Fig. 17.3** Call box for a high-speed counter HSC

parameter has no effect. The parameter is only effective if the counter is configured with internal specification of the counting direction.

With signal state “1”, the CV parameter sets the count value to the initial count value which is specified on the NEW\_CV parameter. Signal state “0” on the CV parameter has no effect.

With signal state “1”, the RV parameter sets the reference value to the value which is specified on the NEW\_RV parameter. Signal state “0” on the RV parameter has no effect.

With signal state “1”, the PERIOD parameter sets the time interval for the measuring period to the value which is specified on the NEW\_PERIOD parameter. Signal state “0” on the PERIOD parameter has no effect. The PERIOD parameter is only effective if the counter is configured with the *Frequency* counting mode.

The BUSY parameter shows with signal state “1” that a triggered job is still being processed following completion of the CTRL\_HSC statement (should not occur with the high-speed counters of a CPU 1200). The STATUS parameter shows the job status and specifies any error messages.

### **17.1.2 Pulse generator**

A pulse generator generates pulses at a special 24-V output channel. The max. frequency is 100 kHz, or 200 kHz when using an appropriately designed signal board.

A pulse generator is an integral component of the CPU and must be activated and configured prior to use. Four pulse generators are available per CPU with firmware version V3.0. A signal board (SB) with digital output channels is required for CPUs with relay outputs. A pulse generator has two modes of operation: PTO (pulse train output) and PWM (pulse-width modulation).

#### **PTO mode**

A pulse generator in PTO mode is required for the technological object *Axis*. The data achievable in this mode are:

- ▷ Minimum frequency 2 Hz
- ▷ Maximum frequency 100 kHz (pulse input in CPU), 20 kHz (pulse input in a standard signal board), and 200 kHz (pulse input in a “high-speed” signal board),
- ▷ Minimum change in frequency (acceleration/deceleration) 0.28 Hz/s
- ▷ Maximum change in frequency (acceleration/deceleration) 9500 MHz/s

#### **PWM mode**

The data required for the control function of a pulse generator in PWM mode is saved in a data block. When calling the control function as a single instance, this is a separate data block per call, when calling as a local instance in a function block, the instance data block of the function block is used for data saving (multi-instance).

The type of output pulses is defined during configuration of the pulse generator. The period is defined by the configuration (time base, cycle time); the pulse width can be changed during runtime. It is specified in units of the pulse width format: as hundredths (percent, 0 to 100), as thousandths (0 to 1000), as ten thousandths (0 to 10 000) or in S7 analog format (0 to 27 648). A value of 0 means that no pulse is output (the output is always “0”), the maximum value means that the output is always “1”.

You enter the initial pulse width during configuration. The pulse width can be changed during runtime. When changing from STOP to RUN, the configured initial pulse width is entered in an output word (Table 17.3).

The CTRL\_PWM statement is used to activate and deactivate the pulse generator in PWM mode.

### Outputs assigned to the pulse generators

The pulse generators are assigned to specific digital outputs depending on the operating mode. The address data in Table 17.3 corresponds to the standard configuration. Please note that the pulse output is assigned to the peripheral output (the “output channel”). The pulse output cannot be connected to a different output channel by assigning a different (logical) address. If an output is not used for a pulse generator, it is available for other purposes. The force function cannot be used for the digital outputs assigned to a pulse generator.

**Table 17.3** Inputs and outputs for the pulse generators

Pulse generator in the operating mode	PTO 0	PTO 1	PTO 2	PTO 3
on the CPU				
Pulse output	%Q0.0	%Q0.2	%Q0.4 1)	%Q0.6 2)
Direction output	%Q0.1	%Q0.3	%Q0.5 1)	%Q0.7 2)
on the signal board				
Pulse output	%Q4.0	%Q4.2	%Q4.0	%Q4.2
Direction output	%Q4.1	%Q4.3	%Q4.1	%Q4.3
Output of the pulse width	%QW1000	%QW1002	%QW1004	%QW1006
Pulse generator in the operating mode	PWM 0	PWM 1	PWM 2	PWM 3
Pulse output on the CPU	%Q0.0	%Q0.2	%Q0.4 1)	%Q0.6 2)
Pulse output on the signal board	%Q4.0	%Q4.2	%Q4.1	%Q4.3

1) Not with CPU 1211C

2) Not with CPU 1211C and CPU 1212C

### Configuring a pulse generator

A pulse generator must be activated using the hardware configuration editor. In order to configure a pulse generator, start the *Device configuration* editor under the PLC station in the project tree. Select a pulse generator in the properties of the CPU

module in the inspector window, and activate it using the *Enable this pulse generator for use* check box.

You set the operating mode under *Parameterization*: PTO, e.g. if it is to be configured for a technology object *Axis*, or PWM. The integral CPU output can be specified as the output source or – if present – the output of the signal board. In PWM mode you also set the pulse shape (pulse width) here.

The assigned pulse output is shown under *Hardware outputs* and, in PTO mode, also the direction output and the high-speed counter used for the technological object *Axis*.

The HW ID for the pulse generator is shown under *Hardware ID*. In PWM mode, the output addresses used are also shown for the pulse width under *I/O addresses*.

### CTRL\_PWM statement

The CTRL\_PWM statement activates and deactivates a pulse generator integrated in the CPU in PWM mode (Fig. 17.4). CTRL\_PWM is present in the program elements catalog under *Extended statements* and *Pulse*. To call the statement, drag CTRL\_PWM with the mouse into the open block. Each CTRL\_PWM call requires an instance data record, which can be either in a separate block (single instance) or – if the call is made in a function block – in the instance data block of the calling function block (multi-instance).

You create the ID for the pulse generator at the PWM parameter. These can be found either in the *System constants* tab in the default tag table or the properties of the CPU in the group of the activated and applied pulse generator under *General* and *Name*. At the PWM parameter, enter this name or select it from the drop-down list. You can also specify the numerical value of the PWM ID, which is shown in the *System constants* tab or in the pulse generator properties under *Hardware ID*, as a constant or variable.

Instance data		Name	Declaration	Data type	Description
CTRL_PWM		EN	–	BOOL	Enabling input
— PWM	BUSY —	ENO	–	BOOL	Enabling output
— ENABLE	STATUS —	PWM	INPUT	HW_PWM	PWM ID
		ENABLE	INPUT	BOOL	Activate pulse generator with "1"
		BUSY	OUTPUT	BOOL	Job being processed with "1"
		STATUS	OUTPUT	WORD	Job status

A **pulse generator** is a function integrated in the CPU module. In **PWM mode** (pulse-width modulation) it is controlled by the CTRL\_PWM statement.

Calling up the statement CTRL\_PWM requires an instance data set, which is either in its own data lock (single instance) or in the instance data block of the calling function block (multi-instance).

**Fig. 17.4** CTRL\_PWM statement

If signal state “1” is present on the ENABLE parameter, the pulse generator is in operation (pulses are being generated); no pulses are generated with signal state “0” and in the STOP operating mode.

### 17.1.3 Technology objects for motion control

The technology objects *TO\_Axis\_PTO* and *TO\_CommandTable\_PTO* are available in a CPU 1200 for controlling and monitoring stepper motors and servomotors with pulse interface.

The technology object *TO\_Axis\_PTO* (“Axis”) is the interface between the user program and the drive. The motion control statements in the user program process the technology object. The technology object *TO\_CommandTable\_PTO* allows you to create motion profiles for a drive that is controlled with *TO\_Axis\_PTO*.

In a CPU 1211C, 1212C and 1214C, a maximum of two *TO\_Axis\_PTO* technology objects can be configured, in a CPU 1215C a maximum of four technology objects. A *TO\_Axis\_PTO* technology object requires a pulse generator in PTO mode. One 24 V output channel is required per drive; a signal board (SB) with a digital output channel must therefore be present for CPUs with relay outputs. An input channel with interrupt capability is required if a reference point switch is used.

#### Configuring the technology object *TO\_Axis\_PTO*

The *Axis* technology object requires an activated pulse generator in PTO mode. You activate and configure the pulse generator in the properties of the CPU in the hardware configuration (see Chapter 17.1.2 “Pulse generator” on page 554).

The *Technology objects* folder in the project tree under the PLC station contains the technology objects. Double-click on *Add new object* to generate a new technology object. In the dialog window, select the *Motion* button and then *TO\_Axis\_PTO*. The technology object stores the instance data in a data block whose number you can set. If the *Add new and open* checkbox is activated, the technology object is opened for configuration. Click *OK*.

A technology object is stored in the project tree in the *Technology objects* folder. The axis contains the entries *Configuration*, *Commissioning*, and *Diagnostics*. Double-click on *Configuration* to open the configuration window (Fig. 17.5). In the working window, select the desired parameter range and enter the configuration data for the axis.

To control an axis in the user program, the statements referred to in Section “Programming the Axis technological axis” on page 559 are available.

#### Configuring the technology object *TO\_CommandTable\_PTO*

With *TO\_CommandTable\_PTO* you can configure a command table for a motion sequence. Double-click on *Add new object* in the *Technology objects* folder to create a new technology object. In the dialog window, select the *Motion* button and then *TO\_CommandTable\_PTO*. The technology object stores the instance data in a data

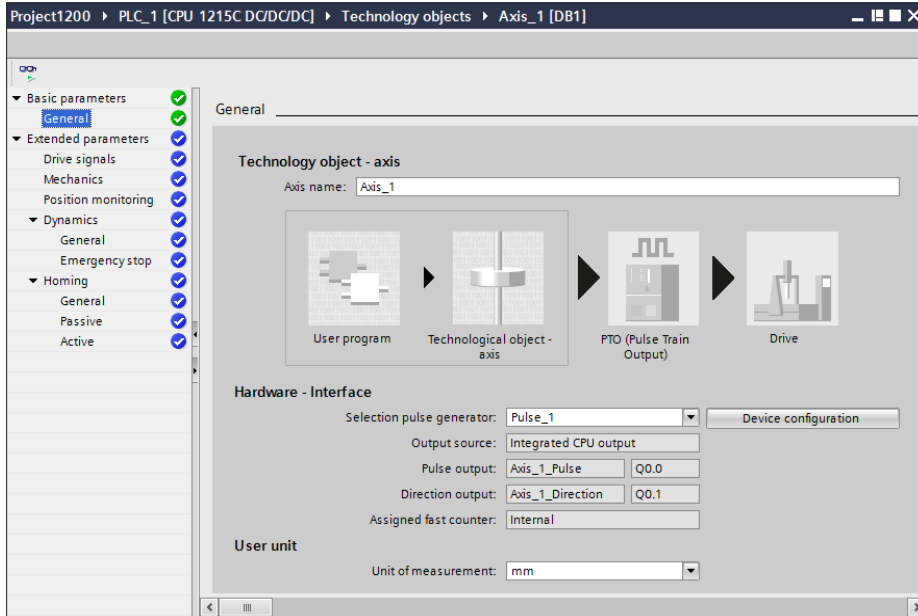


Fig. 17.5 Configuring the technology object *TO\_Axis\_PTO*

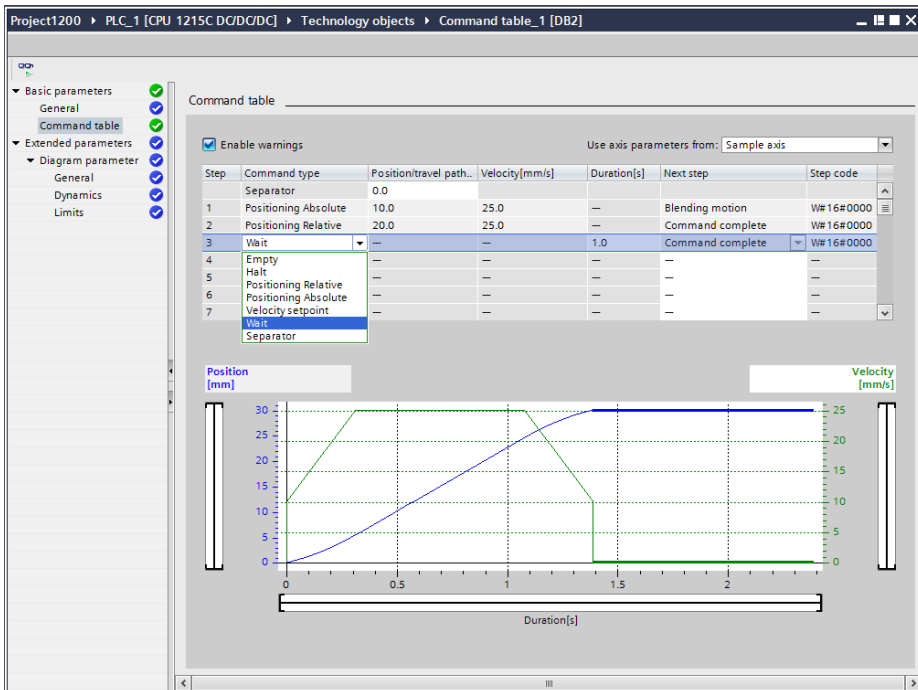


Fig. 17.6 Configuring the technology object *TO\_CommandTable\_PTO*

block whose number you can set. If the *Add new and open* checkbox is activated, the technology object is opened for configuration. Click *OK*.

To configure the command table, double-click *Configuration* in the project tree under the command table used. In the configuration window, enter the motion profile (Fig. 17.6).

For a motion sequence, the commands listed in Table 17.4 are available. Either “Complete job” (the axis decelerates to a stop) or “Continue movement” (the axis maintains speed) can be configured as a completion of a positioning command.

**Table 17.4** Commands for the motion control of an axis

Command	Action
Positioning Absolute	The axis is moved at the configured speed to the specified position.
Positioning Relative	The axis is moved the specified distance at the specified speed.
Velocity Setpoint	The axis is moved at the specified speed for the specified duration.
Hold	The axis is stopped (only after a Velocity Setpoint command).
Wait	There is a wait of the specified time between two commands.
Empty	This command is used as a placeholder for any other command
Separator	This command separates different motion profiles in the command table.

The commands in the command table are implemented in the user program with the statement *MC\_CommandTable*.

### Programming the Axis technological axis

You control the *Axis* technology object and thus the drive with the user program using the motion control statements. The statements are available in the program elements catalog under *Technology* and *Motion control*. To call a statement, drag it with the mouse into the open block. Each call requires an instance data record, which can be either in a separate block (single instance) or – if the call is made in a function block – in the instance data block of the calling function block (multi-instance). Fig. 17.7 shows the calls of the motion control statements.

**MC\_Power** enables and disables an axis for motion control.

**MC\_Reset** resets all errors, and acknowledges all errors which can be acknowledged.

**MC\_Home** sets a homing point, i.e. the (mechanical) positioning system of the axis is matched with the (logical) coordinate system in the controller.

**MC\_Halt** aborts all movements and stops the axis.

**MC\_MoveAbsolute** starts a positioning motion of the axis to an absolute position.

**MC\_MoveRelative** starts a positioning motion of the axis relative to the start position.



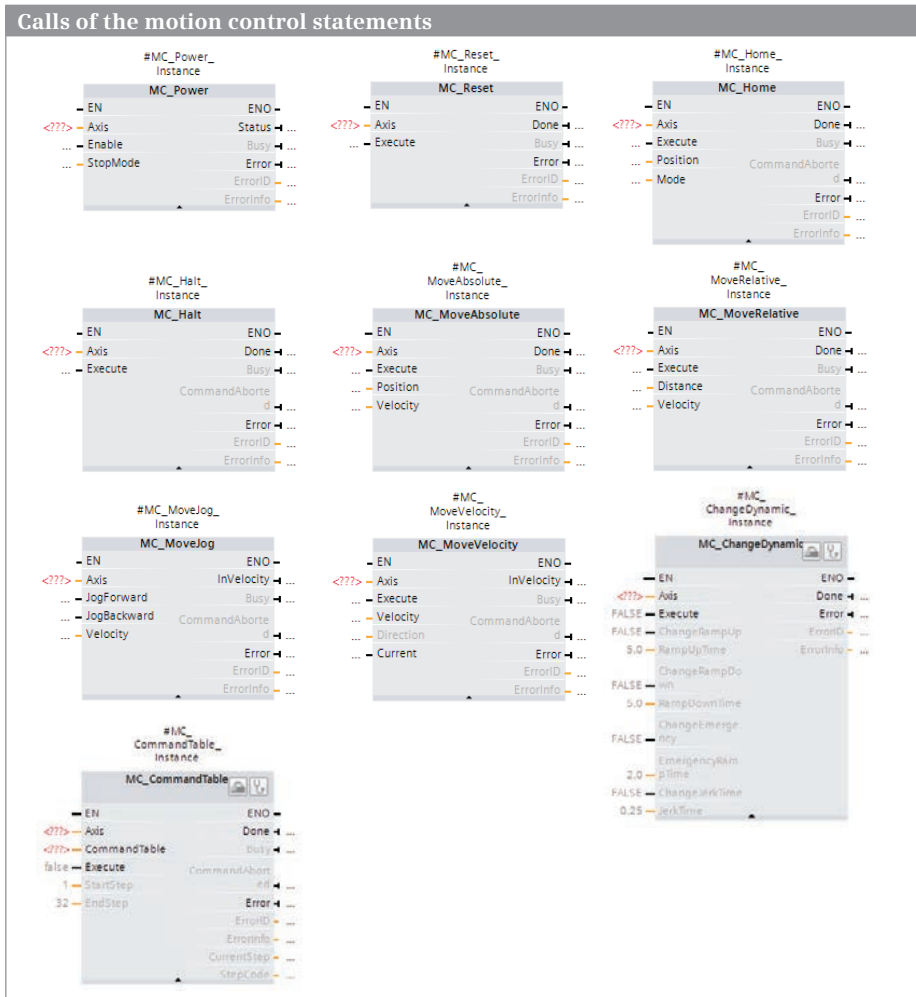
**MC\_MoveVelocity** starts the axis with the defined velocity.

**MC\_MoveJog** starts the axis in jog mode for testing and commissioning.

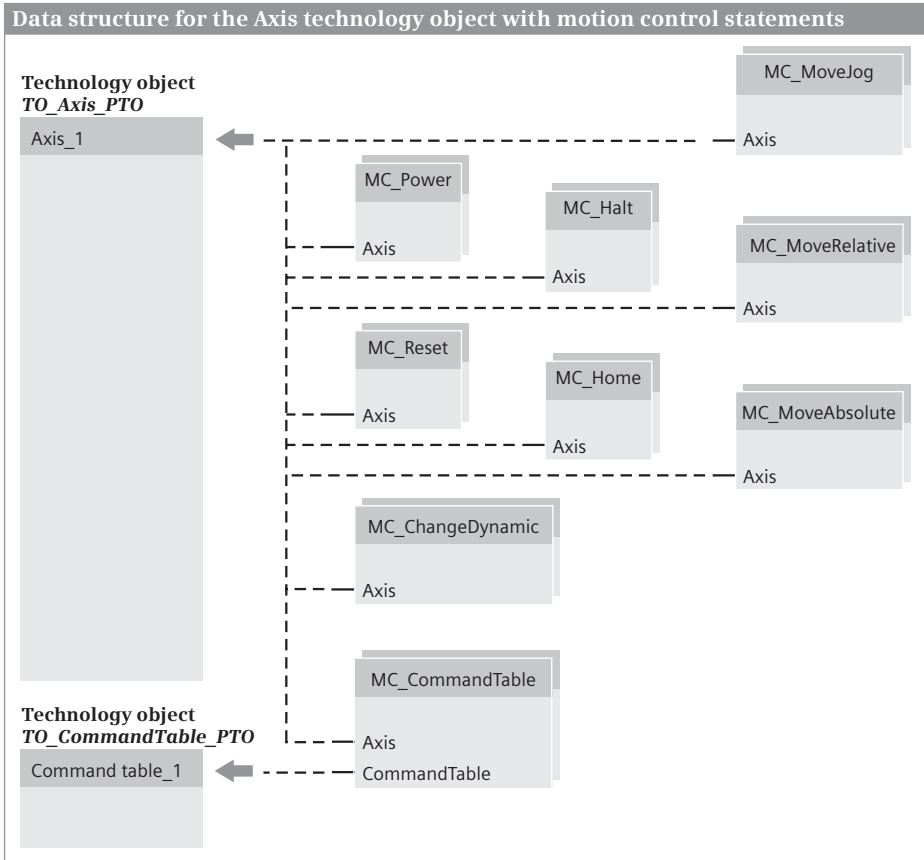
**MC\_ChangeDynamic** changes the dynamic settings of an axis of motion.

**MC\_CommandTable** implements the commands configured in the command table as a motion sequence.

If a motion control statement is still being executed, it must not be interrupted by the start of the same motion control statement. You should therefore call a motion control statement only once in the user program.



**Fig. 17.7** Calling the motion control statements in LAD representation



**Fig. 17.8** Data structure of the motion control technology objects

By assigning the `AXIS` parameter, you define the axis to be controlled by the motion control statements. At `AXIS`, specify the data block which was generated when configuring the technology object `TO_Axis_PTO` (Fig. 17.8). At the parameter `CommandTable` of the statement `MC_CommandTable`, create the data block that has been generated in the configuration of the technology object.

#### 17.1.4 Technology objects for PID control

With the technology objects for PID control you create control loops with PID action and self-tuning in manual and automatic mode. The statements `PID_Compact` (universal PID controller with self-optimization) and `PID_3STEP` (step controllers with 3-point mode with self-optimization for valves) are available.

A PID controller continuously records the measured *Input value* of the controlled variable in a control loop, and compares it with the desired *setpoint*. The PID con-

troller calculates a *manipulated variable* from the difference (the control deviation), and this matches the controlled variable to the setpoint.

With a PID controller, the controlled variable comprise three components: the *P component* which is proportional to the control deviation, the *I component* which is calculated by integration, increases along with the duration of the deviation, and finally results in compensation of the deviation, and the *D component* which increases along with the rate of change of the control deviation in order to minimize this as rapidly as possible.

The technology object *PID\_Compact* requires an analog input channel for the actual value and an analog output channel for the (analog) manipulated variable. If the manipulated variable is to be output as pulse width modulated signal, a digital output channel is required. The pulse generators integrated in the CPU are not required.

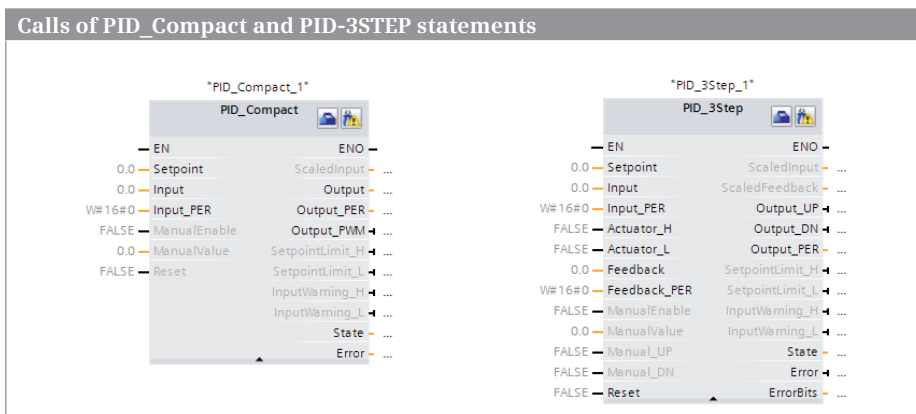
The technology object *PID\_3STEP* requires an analog input channel for the actual value and two digital outputs for “Control up” (e.g. open valve) and “Control down” (e.g. close valve).

### Configuring the PID control technology object

A PID controller can be created in the following ways: You first create a new technology object and then program the controller statement with the specification of the technology object as an instance data block, or you first program the controller statement, where the required instance data block is the technology object, and then you configure the technology object.

### Programming the controller statement

The controller must record the actual value at defined intervals – the scanning time – in order to be able to determine its time characteristics. Therefore the controller statement must be called in a cyclic interrupt organization block whose



**Fig. 17.9** Calls of the controller statements in LAD representation

call interval corresponds to the scanning time. The call can only be made as a single instance. The instance data block required corresponds to the technology object PID control.

To program the PID controller, create a cyclic interrupt OB with the desired scan time (see Chapter 5.7.3 “Cyclic interrupts” on page 159) and program the PID controller either directly in the cyclic interrupt organization block or in a block which is called in the cyclic interrupt OB. Drag the required statement from the *Statements* task card under *Technology* and *PID Control > Compact PID* into the open block and select the corresponding data block from the drop-down list – if you have already configured the technology object – or specify a new data block which is then created. Fig. 17.9 shows the call of the controller statements.

### Configuring the controller

You can also create a PID control technology object before programming one of the controller statements. In the project tree, open the *Technology objects* group and double-click on *Add new object*. Click on the *PID Control* button and select a controller type. If the technology object to be configured is already available, open it in the project tree and double-click on *Configuration*. Enter the desired parameters. Fig. 17.10 shows the basic settings of the configuration for a *PID\_3-STEP* 3-point controller.

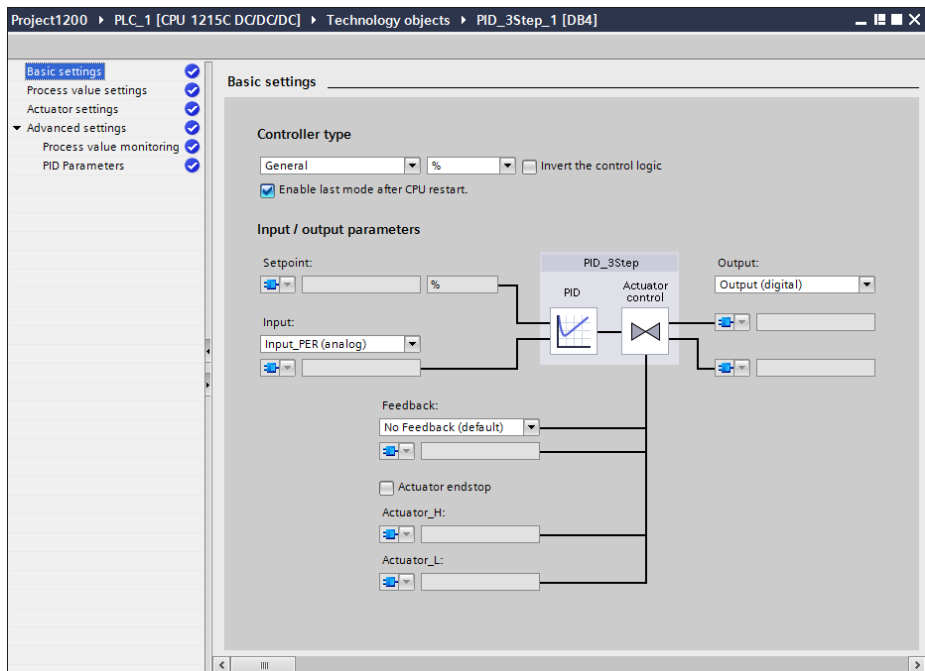
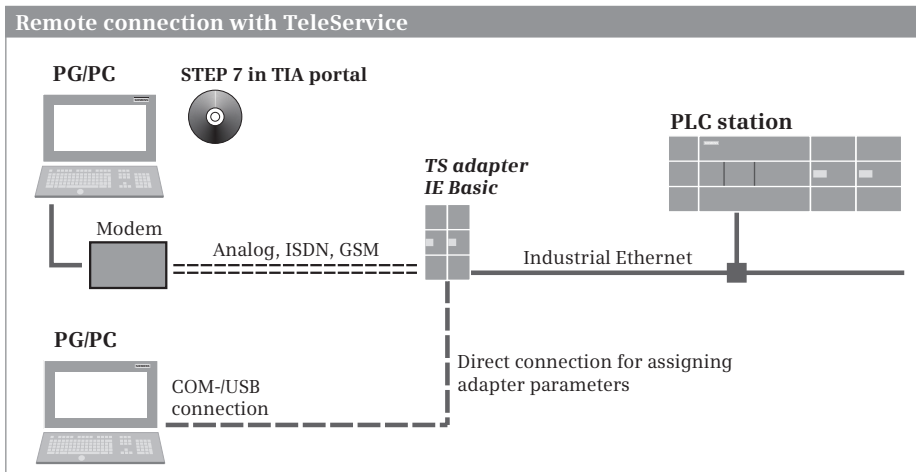


Fig. 17.10 Configuring the *PID controller* technology object

## 17.2 Telephone network connections with TeleService

You use the TeleService to connect a programming device or PC to a PLC or HMI station via the telephone network. This enables you to manage, control, and monitor remote machines or plants from a central point. TeleService is standalone software and does not require the installation of STEP 7. TeleService is included in the scope of delivery of STEP 7 in the TIA Portal. The connection to the central programming device is established via a modem.

A TS Adapter creates the connection on the station side. The TS Adapter IE Basic consists of the basic unit and a TS module with the modem or an interface for connecting to an external modem. The basic unit has an Ethernet interface for connecting to a programming device or programmable controller. The TS-Adapter IE Basic is parameterized with TeleService in the TIA Portal. Fig. 17.11 shows the basic structure of a remote connection using a TS Adapter.



**Fig. 17.11** Controlling a PLC station over a remote connection using TeleService

### Send an e-mail

The statement `TM_MAIL` sends an e-mail via the PN interface of the CPU to the dial-in server of the Internet service provider. If there is no direct Internet connection via the PN interface, the TS Adapter IE can be used to connect to the phone network. `TM_MAIL` uses SMTP (Simple Mail Transfer Protocol) as its transfer protocol.

For programming `TM_MAIL`, drag the statement from the program elements catalog under *Communication > TeleService* to the open block. Fig. 17.12 shows the call of `TM_MAIL` in LAD representation.

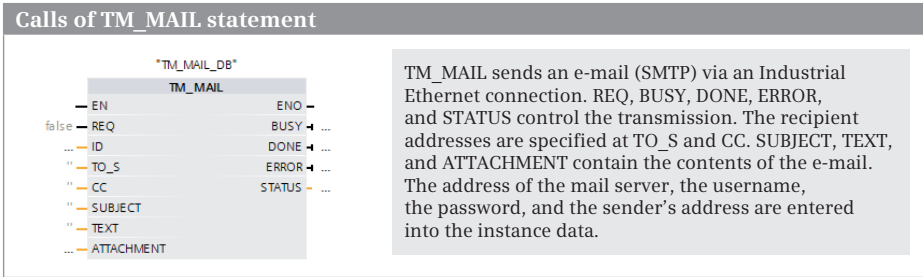


Fig. 17.12 Call of TM\_MAIL to LAD representation

### 17.3 Telecontrol with CP 1242-7

The telecontrol statements control the connection setup and data transfer with a CP 1242-7 communication module. You will find the communication functions in the program elements catalog under *Communication >Communication processor > GPRSComm:CP1242-7* after the CP 1242-7 communication module has been “subsequently installed” with a hardware support package. The calls of these functions are shown in Fig. 17.13.

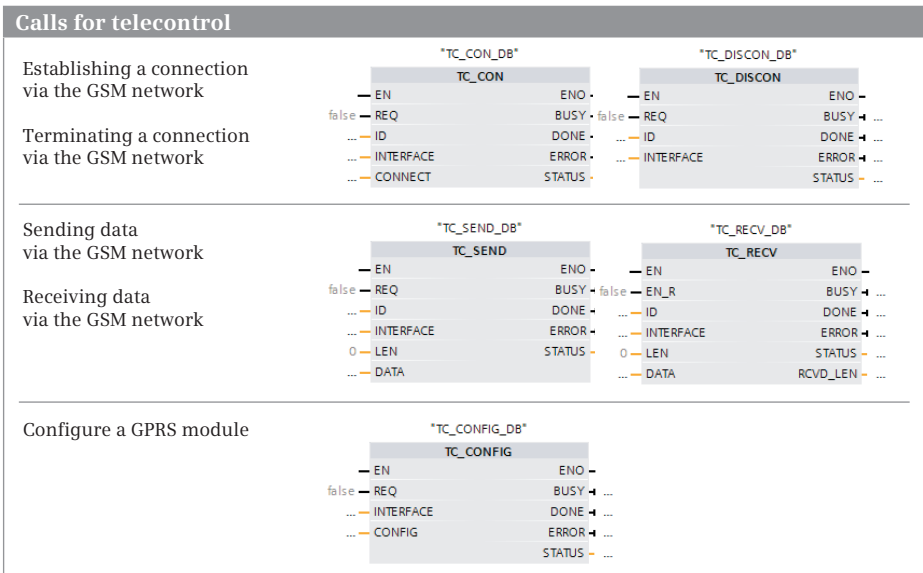


Fig. 17.13 Calls for telecontrol in LAD representation

### **Establishing and clearing a connection via the GSM network**

TC\_CON sets up a communication connection in an S7-1200 station with a CP 1242-7. The connection types ISO-ON-TCP (connection to a CP 1242-7), UDP (connection to any partner), SMS (connection to an SMS client), and telecontrol connection (connection to a telecontrol server) are available.

The parameters REQ, BUSY, DONE, ERROR, and STATUS control the establishment of a connection. At the ID parameter, enter the connection ID and at the parameter INTERFACE, enter the hardware ID of the CP 1242-7 from the *System constants* tab in the default tag table. The CONNECT parameter contains a pointer to the data block with the connection data. Depending on the type of connection used, this data block is derived from one of the system data types TCON\_ip\_rfc, TCON\_IP\_V4, TCON\_phone, or TCON\_WDC.

TC\_DISCON closes the connection established with TC\_CON. The parameters REQ, BUSY, DONE, ERROR, and STATUS control the establishment of a connection. At the ID parameter, enter the connection ID and at the parameter INTERFACE, enter the hardware ID of the CP 1242-7 from the *System constants* tab in the default tag table.

### **Sending data via the GSM network**

TC\_SEND sends data via a connection established with TC\_CON. The parameters REQ, BUSY, DONE, ERROR, and STATUS control the data transmission. At the ID parameter, enter the connection ID and at the parameter INTERFACE, enter the hardware ID of the CP 1242-7 from the *System constants* tab in the default tag table. The DATA parameter contains a pointer to the send data. The number of bytes to be sent is specified at the LEN parameter.

### **Receiving data via the GSM network**

TC\_RECV receives data via a connection established with TC\_CON. The parameters EN\_R, BUSY, NDR, ERROR, and STATUS control the data transmission. At the ID parameter, enter the connection ID and at the parameter INTERFACE, enter the hardware ID of the CP 1242-7 from the *System constants* tab in the default tag table. The DATA parameter contains a pointer to the receive mailbox. You enter the maximum number of bytes to be received at the parameter LEN. The actual number of bytes received is output at the parameter RCVD\_LEN.

### **Transferring configuration data to the CP 1242-7**

TC\_CONFIG transfers (new) configuration data to the CP 1242-7 communication module and thus overwrites (temporarily) the data configured with the hardware configuration. On the next startup, the initial configuration data is imported again. The parameters REQ, BUSY, DONE, ERROR, and STATUS control the transmission. At the parameter INTERFACE, enter the hardware ID of the CP 1242-7 from the *System constants* tab in the default tag table. The parameter CONFIG contains a pointer to a data area with the configuration data whose structure is specified by the system data type IF\_CONF.

## 17.4 Web server

A CPU 1200 has a web server that provides information from the CPU. To read out the information you require a web browser which displays the information on the HTML pages.

### 17.4.1 Enable web server

You enable the web server with the hardware configuration using the *Enable web server on this module* checkbox in the CPU properties under the *Web server* group.

By activating the *Permit access only via HTTPS* checkbox you limit access to the secure hypertext transmission protocol. You additionally require a valid and installed certificate, which you can download and install via the *download certificate* link on the initial page of the web server.

### 17.4.2 Reading out web information

In order to access the CPU's web server, the PC or PG must establish an Ethernet connection (TCP/IP) to the CPU. Start the web browser and enter the CPU's IP address as URL in the form *http://aaa.bbb.ccc.ddd* or – for a secure connection – *https://aaa.bbb.ccc.ddd*.

You can turn off automatic updating on or off and generate a print image of the updated website using the icons on the top right of each page. To enable logging on, two input boxes are provided for the user name and password on every page at the top left. Registration is not required for read access. To perform specific actions such as a firmware update for the CPU via the web server, it is necessary to log in as “admin” with a password configured for a protected CPU. The standard web pages use JavaScript and cookies, which you should release for unrestricted operation in the web browser.

### 17.4.3 Standard web pages

The first page displayed by the web server is the Welcome page. From here, click on ENTER to reach the Start page. The *Start page* shows a graphical representation of the CPU with the enabled LEDs, along with the general data and the status of the CPU (Fig. 17.14) .

The *Identification* page contains static information such as order number, serial number, and version numbers.

On the *Diagnostics Buffer* page you can see the contents of the diagnostics buffer with the most recent entries first. Select the group of 25 to be displayed from the drop-down list. Detailed information about the selected event is displayed.

The *Module Information* page shows the module status of the PLC station. From here you can call up the status of individual modules. Use the link in the “Head-





**Fig. 17.14** Start page of the web server

ing” to access a higher module level, the links in the table column *Name* to access lower levels.

The *Communication* page shows the network connection and used addresses in the *Parameters* tab and in the *Statistics* tab it shows statistical information on the data packets transmitted and received.

On the *Variable Status* page, you can enter operands in a table and display their statuses. Note the fixed time here for the automatic update. Clicking on *Monitor Value* updates the operand values immediately. If you are logged in as “admin”, you can also control the operand values.

On the *Data Logs* page you can transfer the data archives in CSV format created in the user program to the hard disk of the programming device.

The *User Pages* page shows a list of websites with user-specific web applications. When configuring the web server, you can specify the web pages in the CPU properties which you wish to load together with the other settings of the web server into the CPU.

The link to the *Update Firmware* page is only shown if you are logged in as “admin”.

A click on *Introduction* opens the Welcome page.

## WWW Initialize web server and synchronize web pages

WWW initializes user-defined pages in the web server of the CPU and synchronizes access between the pages and the user data. WWW is called cyclically in the user program. You can find WWW in the program elements catalog in the section *Communication* under *Web server* (Fig. 17.15).

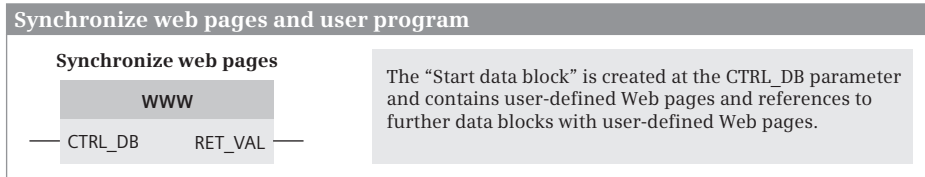


Fig. 17.15 Graphic representation of system function WWW

## 17.5 Data logging

### 17.5.1 Introduction

With data logging, selected process values from the user program are written to the data log file. The data log file is located in the load memory. This can be either the internal load memory of the CPU or the external load memory on the memory card.

A data log file stores the values in CSV format, i.e. each separated by a comma. The logged data can be read out with a web browser using the web server available in the CPU. If the data log file is on a memory card, the logged data can also be read out using an SD card reader on the programming device.

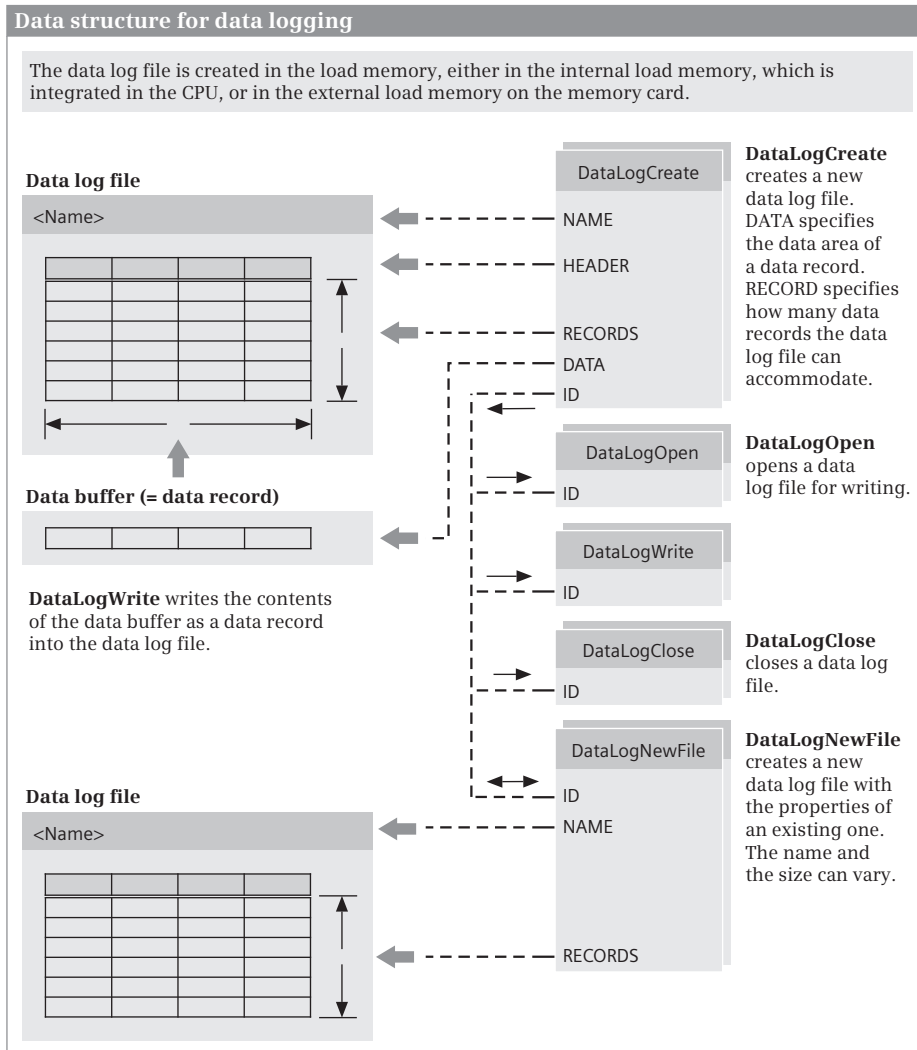
The data log file is designed as a ring buffer with a configurable number of data records. If the maximum number is reached, the oldest data record is overwritten. The size of a data log file must not exceed 25% of the load memory size. All data log files should not occupy more than a maximum of 50% of the load memory.

### 17.5.2 Using data logging

To use data logging, define a data buffer in a data area with any structure. You can write the contents of the data buffer as a data record into the data log file. This could be triggered, for example, at the end of a batch depending on production or with a time-controlled trigger in a specific timeframe (Fig. 17.16).

*DataLogCreate* creates a new (empty) data log file in the load memory. *DataLog Open* opens a data log file. The data records can then be written with *DataLog Write*. A maximum of 10 data log files can be opened simultaneously. *DataLog Close* closes a data log file so that no more data records can be written.

Each time a data record is written, the data log file is filled in. If it is full, the next data record to be written overwrites the oldest data record. *DataLogNewFile*



**Fig. 17.16** Data structure for data logging

allows an “expansion” of the data log file to avoid overwriting old records. The function creates a new (empty) data log file on the basis of the original data log file.

### 17.5.3 Functions for data logging

Fig. 17.17 shows the call of functions for data logging in LAD representation.

**DataLogCreate** creates a new data log file. The parameters REQ, DONE, BUSY, ERROR, and STATUS control the execution of the function. At the NAME parameter, enter the name of the data log file, following the requirements for Windows file

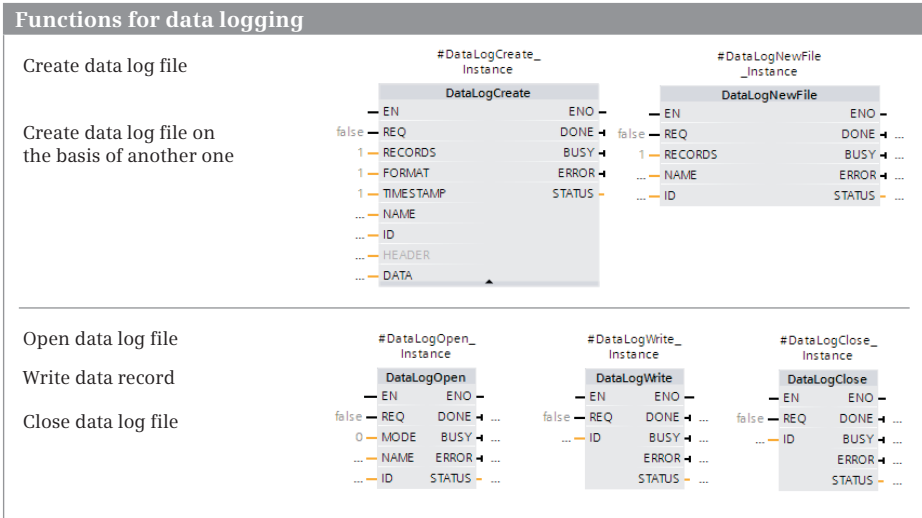


Fig. 17.17 Functions for data logging

names. Further information on the data log file is located at the parameters DATA (pointer to the data buffer with the data record), RECORDS (maximum number of data records), and HEADER (header in the data log file). A numerical value specifying the data log file is output at the ID parameter. You specify this numerical value at the other functions that access this data log file.

**DataLogOpen** opens the log file whose identifier is in the ID parameter. If you specify the name of the log file at the NAME parameter instead, the ID is output at the ID parameter. The parameters REQ, DONE, BUSY, ERROR, and STATUS control the execution of the function. Opening is the prerequisite for writing to the data log file. DataLogCreate and DataLogNewFile also open the newly created data log file. Use the MODE parameter to select whether the data records are deleted on opening (if MODE = 1).

**DataLogWrite** writes a data record to the data log file whose identifier is in the ID parameter. The data record is taken from the data buffer specified at the DATA parameter of DataLogCreate

**DataLogClose** closes the data log file whose identifier is at the ID parameter. The parameters REQ, DONE, BUSY, ERROR, and STATUS control the execution of the function. A data log file is also closed in the operating modes STARTUP and STOP.

**DataLogNewFile** creates a new data log file with the same properties as the data log file whose identifier is specified at the ID parameter. After execution, the identifier of the newly created data log file is at the ID parameter. At the NAME parameter, specify the name for the new data log file and at the RECORDS parameter, specify the maximum number of data records. The parameters REQ, DONE, BUSY, ERROR, and STATUS control the execution of the function.

# Index

## A

- ABS 368
- ACOS 373
- ADD 367
- Addressing 85
- Alarm system (HMI) 528
- AND (word logic operation) 392
- AND function
  - Description 331
  - with FBD 252
  - with LAD 215
  - with SCL 291
- Arc functions 373
- Area pointer (HMI) 515
- Arithmetic functions
  - for numerical values
    - Description 366
    - with FBD 273
    - with LAD 236
  - for time values
    - Description 369
    - with FBD 273
    - with LAD 236
  - with SCL 301
- ARRAY (data type) 104
- ASIN 373
- Assignment
  - Description 334
  - with FBD 259
  - with LAD 222
  - with SCL 294
- Assignment list 203
- ATAN 373
- ATH 389
- ATTACH 165
- Authorization 31

## B

- Basic Panels 508
- BCD16 (data type) 95
- BCD32 (data type) 95

- Binary logic operations
  - with FBD 249
  - with LAD 212
  - with SCL 288
- Bit memory 82
- Block
  - Calling 137
  - Comparing 432
  - Compile 198
  - Copy protection 132
  - Correct call 200
  - Editing
    - FBD elements 248
    - LAD elements 211
    - SCL statement 286
  - Interface
    - Correction 133
    - Description 133
    - Supplying 139
  - Know-how protection 132
  - Programming
    - Code block 183
    - Data block 194
  - Properties 128
- Block calls
  - Description 413
  - with FBD 282
  - with LAD 245
  - with SCL 316
- Block end function
  - Description 412
  - with FBD 282
  - with LAD 244
  - with SCL 316
- BOOL (data type) 95
- BYTE (data type) 95

## C

- Call structure 204
- CAN\_DINT 158
- CASE (SCL) 308
- CEIL 379
- CHAR (data type) 100
- CHARS\_TO\_STRG 387
- Clock memory bits 84

- Communication
  - Configuring 483
  - Open user
    - communication 484
    - Point-to-point communication 499
- Comparison functions
  - Description 364
  - with FBD 258
  - with LAD 219
  - with SCL 300
- Configuring function keys (HMI) 520
- Configuring process screen (HMI) 517
- Configuring the network 482
- Constants table 182
- Contacts
  - Comparison 219
  - Edge 218
  - NC contact 213
  - NO contact 212
  - OK contact 219
- CONTINUE (SCL) 314
- Control program
  - Cycle processing time 439
  - Programming
    - with FBD 246
    - with LAD 209
- Control statements (SCL) 307
- Controlling the program flow
  - with FBD 279
  - with LAD 241
  - with SCL 305
- CONV 377
- Conversion functions
  - Description 376
  - with FBD 275
  - with LAD 238
  - with SCL 299
- Copy protection 132
- COS 373
- Counter (HSC) 548

- Cross-reference list
    - for HMI objects 512
    - for the control program 201
  - CTD down counter 352
  - CTRL\_HSC 552
  - CTRL\_PWM 556
  - CTU up counter 351
  - CTUD up-down counter 353
  - Cycle processing time 437, 439
  - Cycle time monitoring 144
  - Cyclic interrupt 159
- D**
- Data addresses 84
  - Data management
    - in the CPU 435
    - with STEP 7 29
  - Data types
    - Classification 92
    - Elementary 92
    - Structured 101
  - DATE (data type) 100
  - DEC 369
  - DECO 394
  - DEMUX 395
  - Dependency structure 205
  - DETACH 166
  - Device name, device number 74
  - DeviceStates 173
  - Diagnostics buffer 437
  - Diagnostics interrupt OB 82 176
  - Digital functions
    - with FBD 270
    - with LAD 233
    - with SCL 298
  - DINT (data type) 98
  - DIS\_AIRT 166
  - Distributed I/O
    - AS-Interface 473
    - PROFIBUS DP 462
    - PROFINET IO 456
  - DIV 367
  - Downloading HMI program 543
  - DPNRM\_DG 471
  - DPRD\_DAT 472
  - DPWR\_DAT 472
  - DTL (data type) 101
  - DWORD (data type) 95
- E**
- Edge evaluation
    - Description 338
    - of a binary tag
      - with FBD 256
      - with LAD 218
    - of RLO
      - with FBD 266
      - with LAD 229
    - with pulse output
      - with FBD 261
      - with LAD 224
    - with SCL 295
  - EN\_AIRT 167
  - EN/ENO mechanism
    - Description 417
    - with FBD 418
    - with LAD 418
    - with SCL 306
  - Enable peripheral outputs 451
  - ENCO 395
  - ENO (tag, SCL) 305
  - ErrorStruct (data type) 112
  - Exclusive OR function
    - Description 333
    - with FBD 254
    - with SCL 292
  - EXIT (SCL) 316
  - EXP 374
  - Expressions (SCL) 288
  - EXPT 375
- F**
- FILL\_BLK 361
  - Filling of bit field
    - Description 336
    - with FBD 262
    - with LAD 225
  - FLOOR 380
  - FOR (SCL) 311
  - FRAC 375
  - Function lists (HMI) 528
- G**
- GET\_DIAG 175
  - GET\_ERROR 170
  - GET\_ERROR\_ID 170
- H**
- Hardware detection 434
  - Hardware diagnostics 436
  - High-speed counter 548
  - HMI device wizard 511
  - HMI tags 513
  - HTA 389
- I**
- IEC counter functions
    - Description 349
    - with FBD 268
    - with LAD 231
    - with SCL 297
  - IEC time functions
    - Description 344
  - IEC timer functions
    - with FBD 267
    - with LAD 230
    - with SCL 296
  - IF (SCL) 308
  - INC 369
  - Inputs 80
  - INT (data type) 98
  - Interrupt processing
    - Cyclic interrupt 159
    - Delaying and enabling 166
    - Introduction 153
    - Process interrupt 163
    - Time-delay interrupt 155
  - INV 394
  - IP address
    - Assigning to CPU 424
    - Configuring 73
    - of the programming device 421
- J**
- JMP\_LIST 409
  - Jump distributor
    - with FBD 281
    - with LAD 244
  - Jump functions
    - Description 406
    - with FBD 280
    - with LAD 242
  - Jump list
    - with FBD 281
    - with LAD 243
- L**
- Language setting 207
  - LED (function) 172
  - Library
    - Editing 42

LIMIT 398  
 LN 374  
 Local error handling 169  
 Logic functions  
   Description 392  
   with FBD 277  
   with LAD 240  
   with SCL 303  
 LREAL (data type) 98

**M**

Math functions  
   with FBD 274  
   with LAD 237  
 Mathematical functions  
   Description 372  
   with SCL 303  
 MAX 397  
 Memory card 428  
 Memory functions  
   Description 334  
   with FBD 265  
   with LAD 227  
   with SCL 294  
 Memory reset 439  
 Memory utilization  
   online 437, 439  
 MIN 397  
 Minimum cycle time 146  
 MOD 368  
 Modules  
   Assigning parameters 61  
   Properties 50  
   Status displays 436  
 ModuleStates 174  
 MOVE 356  
 MOVE\_BLK 360  
 MUL 367  
 MUX 395

**N**

NEG 369  
 Negating result of logic  
   operation  
     with FBD 255  
     with LAD 218  
 Negating the result of logic  
   operation  
     Description 329  
     with SCL 293  
 Nesting depth  
   Blocks 125  
 NORM\_X 381

Normally closed  
   contact 213  
 Normally open contact 212

**O**

OK test  
   Description 330  
   with FBD 257  
   with LAD 219  
 Online tools 439  
 Open user  
   communication 484  
 Operands 79  
 Operating mode  
   RUN 119  
   STARTUP 118  
   STOP 118  
 Operator control and display  
   objects (HMI) 525  
 Operators (SCL) 286  
 OR (word logic  
   operation) 392  
 OR function  
   Description 332  
   with FBD 253  
   with LAD 215  
   with SCL 291  
 Organization block  
   Cyclic interrupt 159  
   OB 1 main program 143  
   OB 100 startup  
     program 142  
   OB 80 time error 168  
   OB 82 diagnostics  
     interrupt 176  
   Process interrupt 163  
   Time-delay interrupt 155  
 Outputs 80-81

**P**

Parallel connection 215  
 PEEK (SCL) 90  
 Peripheral inputs 80  
 Peripheral outputs 80-81  
 PLC station  
   Adding 60  
   Parameterization 61  
 PLC tag table  
   Editing 178  
   Monitoring with 445  
 Point-to-point  
   communication 499  
 POKE (SCL) 90

Priority classes 154  
 Process image update 143  
 Process interrupt 163  
 PROFIBUS DP  
   Addressing 465  
   Configuring 467  
 PROFINET IO  
   Addressing 457  
   Configuring 459  
 Real-Time  
   Communication 461  
 Program execution  
   modes 124  
 Program status 441  
 Project  
   Editing 41  
   Object hierarchy 38  
 Pulse generator 554

**Q**

QRY\_CINT 161  
 QRY\_DINT 158

**R**

RALRM 473  
 RD\_LOC\_T 151  
 RD\_SYS\_T 150  
 RDREC 472  
 RE\_TRIGR 145  
 READ\_DBL 362  
 REAL (data type) 98  
 Recipes (HMI) 535  
 REPEAT (SCL) 313  
 Resources  
   Offline 206  
 Retentive behavior 121  
 ROL 392  
 ROR 391  
 ROUND 380  
 RTM 152

**S**

S\_CONV 383  
 SCALE\_X 381  
 Scanning of signal state  
   Description 329  
   with FBD 250  
   with LAD 212  
   with SCL 288  
 SEL 395  
 Series connection 215  
 SET\_CINT 161

- SET\_TIMEZONE 150
  - Setting and resetting
    - Description 335
    - with FBD 260
    - with LAD 223
    - with SCL 294
  - Shift functions
    - Description 389
    - with FBD 276
    - with LAD 239
    - with SCL 304
  - SHL 391
  - SHR 389
  - Simulation (HMI) 542
  - SIN 373
  - SINT (data type) 98
  - SQR 374
  - SQRT 374
  - SRT\_DINT 158
  - Start-up routine 142
  - STEP 7
    - Portal view 32
    - Project view 34
  - STP 147
  - STRG\_TO\_CHARS 387
  - STRG\_VAL 385
  - STRING (data type) 102
  - String functions
    - Description 398
    - with FBD 278
    - with LAD 240
  - STRUCT (data type) 104
  - SUB 367
  - SWAP 363
  - SWITCH 410
  - System memory bits 82
- T**
- T branch
    - with FBD 255
    - with LAD 217
  - T\_ADD 371
  - T\_CONFIG 495
  - T\_CONV 383
  - T\_DIFF 371
  - T\_SUB 371
- Tags**
- Addressing 85
  - Declaring data tags 198
  - Forcing 452
  - HMI tags 513
  - Introduction 79
  - Modifying 450
  - Monitoring with
    - Watch tables 449
  - monitoring with
    - PLC tag table 445
    - PLC tag table 178
- TAN** 373
- Technology objects**
- for motion control 557
  - for PID control 561
- TeleService** 564
- Text and graphics lists** (HMI) 526
- Time**
- Configuring 148
  - Setting online 151
- TIME** (data type) 100
- Time error OB** 80 168
- TIME\_OF\_DAY** (data type) 101
- Time-delay interrupt** 155
- TM\_MAIL** 564
- TOF off-delay** 347
- TON on-delay** 347
- TONR accumulating ON delay** 348
- TP pulse generation** 346
- Transfer functions**
- Description 356
  - with FBD 271
  - with LAD 235
  - with SCL 298
- Trigonometric functions** 373
- Troubleshooting** 167
- TRUNC** 380
- U**
- UDINT (data type) 97
  - UFILL\_BLK 361
- UINT** (data type) 97
- UMOVE\_BLK** 360
- User administration** (HMI) 539
- User data** 81
- User program**
- Download 425
  - Minimum cycle time 146
  - Process image 143
- Programming**
- General 189
  - with FBD 246
  - with LAD 209
  - with SCL 284
- Reaction time** 146
- Test with**
- Program status (LAD, FBD) 442
  - Program status (SCL) 444
  - Watch tables 447
- Troubleshooting** 167
- USINT** (data type) 97

**V**

- VAL\_STRG 387
- VARIANT (parameter type) 108
- VOID (parameter type) 109

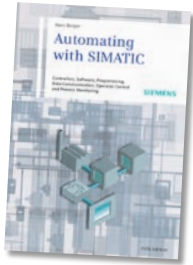
**W**

- Warm restart 119
- Watch tables 447
- Web server 567
- WHILE (SCL) 312
- WORD (data type) 95
- WR\_SYS\_T 148
- WRIT\_DBL 362
- WRREC 472
- WWW 569

**X**

- XOR (word logic operation) 392



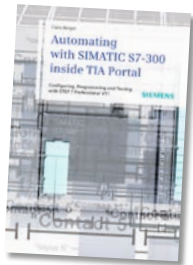


Hans Berger

## **Automating with SIMATIC**

**Controllers, Software, Programming,  
Data Communication, Operator Control  
and Process Monitoring**

5<sup>th</sup> revised and enlarged edition, 2012,  
284 pages, 140 illustrations, 49 tables, hardcover  
ISBN 978-3-89578-387-6, € 44.90

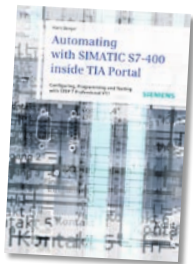


Hans Berger

## **Automating with SIMATIC S7-300 inside TIA Portal**

**Configuring, Programming and Testing  
with STEP 7 Professional V11**

2012, 709 pages, 429 illustrations,  
85 tables, hardcover  
ISBN 978-3-89578-382-1, € 69.90



Hans Berger

## **Automating with SIMATIC S7-400 inside TIA Portal**

**Configuring, Programming and Testing  
with STEP 7 Professional**

June 2013, ca. 760 pages,  
441 illustrations, 94 tables, hardcover  
ISBN 978-3-89578-383-8, € 69.90



Nicolai Andler

## **Tools for Project Management, Workshops and Consulting**

**A Must-Have Compendium of  
Essential Tools and Techniques**

2<sup>nd</sup> revised and enlarged edition, 2011,  
382 pages, 136 illustrations, 55 tables, hardcover  
ISBN 978-3-89578-370-8, € 39.90

