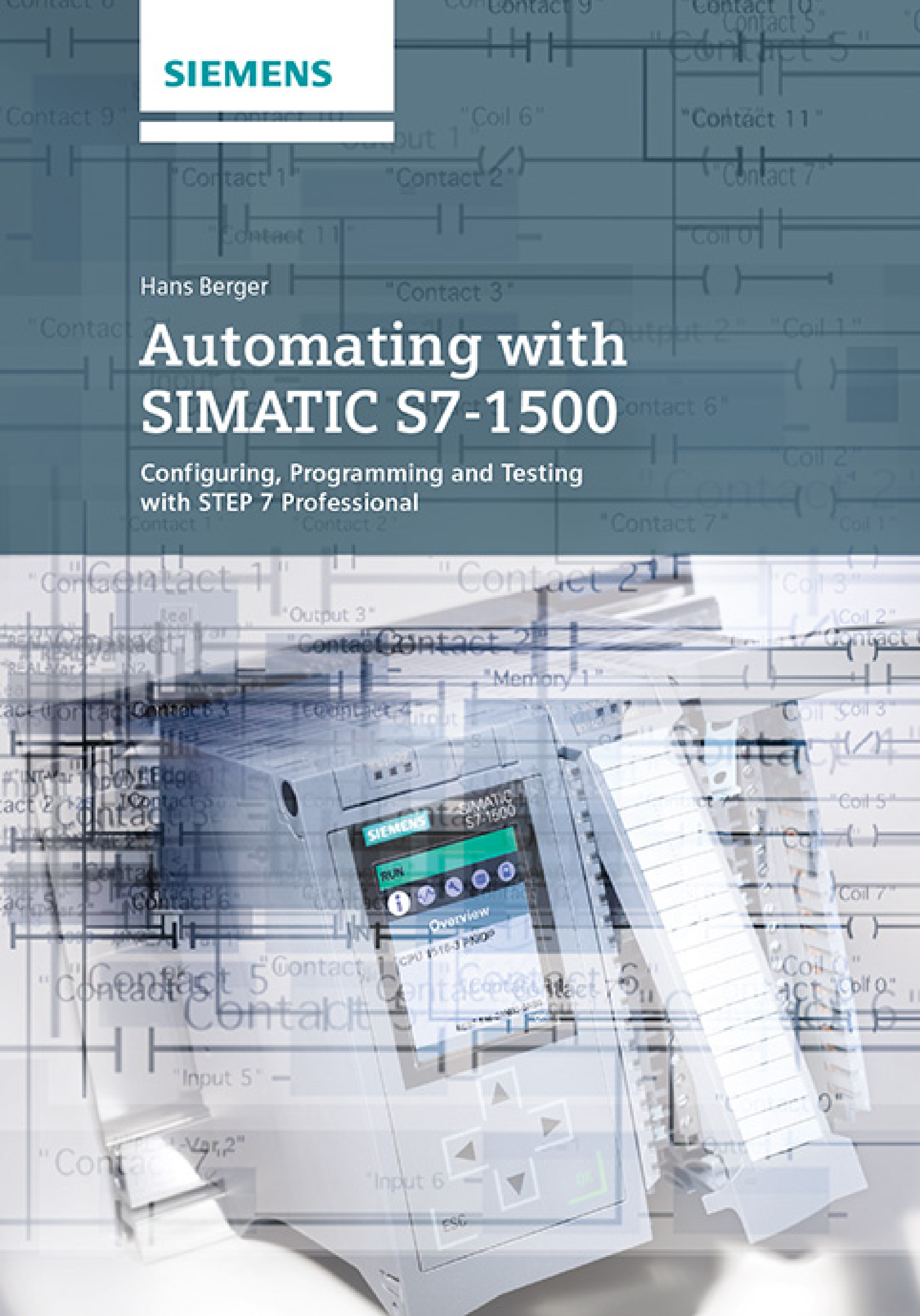
SIEMENS

Hans Berger

Automating with SIMATIC S7-1500

Configuring, Programming and Testing with STEP 7 Professional



Berger Automating with SIMATIC S7-1500

Automating with SIMATIC S7-1500

Configuring, Programming and Testing
with STEP 7 Professional

by Hans Berger

Publicis Publishing

Bibliographic information from 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

Print ISBN 978-3-89578-404-0
ePDF ISBN 978-3-89578-919-9

Editor: Siemens Aktiengesellschaft, Berlin and Munich
Publisher: Publicis Publishing, Erlangen

Printed in Germany

Preface

The SIMATIC automation system unites all of 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* (TIA) 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 S7-1500 automation system. S7-1500 controllers are compact in design and can be modularly expanded. The CPUs feature integrated bus interfaces for communicating with other automation systems via Industrial Ethernet and, depending on the type of module, via PROFIBUS DP as well.

The STEP 7 Professional engineering software in the TIA Portal makes it possible to use the complete functionality of the S7-1500 controllers. STEP 7 Professional is the common tool for hardware configuration, generation of the user program, and for program testing and diagnostics.

STEP 7 Professional provides five programming languages for generation of the user program: Ladder logic (LAD) with a graphic representation similar to a circuit diagram, function block diagram (FBD) with a graphic representation based on electronic circuitry systems, a high-level Structured Control Language (SCL) similar to Pascal, statement list (STL) with formulation of the control task as a list of commands at machine level, and finally GRAPH as a sequencer with sequential processing of the user program.

STEP 7 Professional supports testing of the user program by means of watch tables for monitoring, control and forcing of tag values, by representation of the program with the current tag values during ongoing operation, and by offline simulation of the programmable controller.

This book describes the configuration, programming, and testing of the S7-1500 automation system using the engineering software STEP 7 V12 SP 1 in connection with a CPU 1500 with firmware version V1.1 and the simulation software PLCSIM version V12 SP 1.

Erlangen, May 2014 Hans Berger

The contents of the book at a glance

Start

Overview of the SIMATIC S7-1500 automation system.

Introduction to the SIMATIC STEP 7 Professional V12 engineering software.

The basis of the automation solution: Creating and editing a project.

SIMATIC S7-1500 automation system

Overview of the SIMATIC S7-1500 modules: Design of an automation system, CPUs, signal, technology and communication modules.

Device configuration

Configuration of a station, parameterization of modules, and networking of stations.

Tags, addressing, and data types

The properties of inputs, outputs, I/O, bit memories, data, and temporary local data as operand areas, and how they are addressed: absolute, symbolic, and indirect.

Description of elementary and structured data types, data types for block parameters, pointers, user and system data types.

Program execution

How the CPU responds in the STARTUP, RUN, and STOP modes.

How the user program is structured with blocks, what the properties of these blocks are, and how they are called.

How the user program is executed: startup characteristics, main program, interrupt processing, troubleshooting, and diagnostics.

The program editor

Working with the PLC tag table, creating and editing code and data blocks, compiling blocks, and evaluating program information.

The ladder logic programming language LAD

The characteristics of LAD programming; series and parallel connection of contacts, the use of coils, standard boxes, Q boxes, and EN/ENO boxes.

The function block diagram programming language FBD

The characteristics of FBD programming; boxes for binary logic operations, the use of standard boxes, Q boxes, and EN/ENO boxes.

The structured control language SCL

The characteristics of SCL programming; operators and expressions, working with binary and digital functions, control of program execution using control statements.

The statement list programming language STL

The characteristics of STL programming; programming of binary logic operations, application of digital functions, and control of program execution.

The sequential control programming language GRAPH

What a sequential control is, and what its elements are: sequencers, steps, transitions, and branches. How a sequential control is configured using GRAPH.

Description of the control functions

Basic functions: Functions for binary signals: binary logic operations, memory functions, edge evaluations, SIMATIC timer/counter functions, IEC timer/counter functions.

Digital functions: Functions for digital tags: transfer, comparison, arithmetic, math, conversion, shift, and logic functions.

Program control: Branching in the program using jump functions, calling and ending functions and function blocks, ARRAY and CPU data blocks.

Online operation and program test

Connecting a programming device to the PLC station, switching on online mode, transferring the project data, and protecting the user program.

Loading, modifying, deleting, and comparing the user blocks.

Working with the hardware diagnostics and testing the user program.

Distributed I/O

Overview: The ET 200 distributed I/O system.

How a PROFINET IO system is configured, and what properties it has.

How a PROFIBUS DP master system is configured, and what properties it has.

Communication

The communication functions used to implement open user communication.

The properties of S7 communication and with what communication functions it is programmed.

How PtP communication is implemented.

Appendix

How external source files are created and imported for STL and SCL blocks.

How a project created using STEP 7 V5.x is migrated to the TIA Portal.

How the Web server is configured in the CPU, and what features it offers.

Technology functions: counting, measuring, motion control, PID control

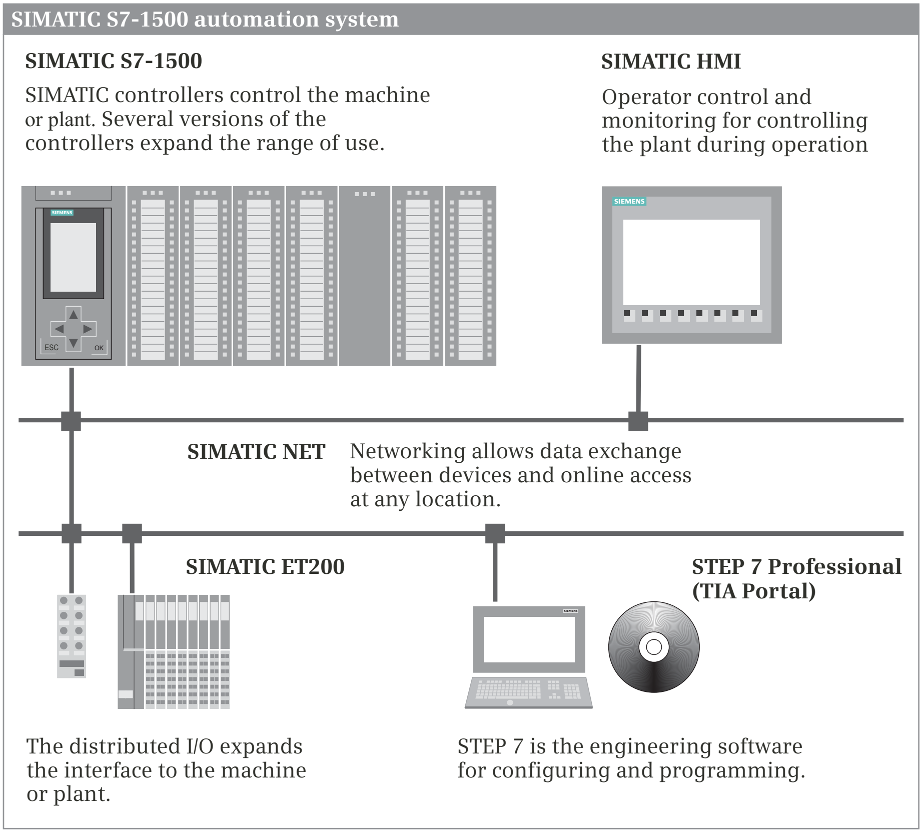
How the user program is tested offline using the S7-PLCSIM simulation software.

Table of contents

1 Introduction

1.1 Overview of the S7-1500 automation system

SIMATIC S7-1500 is the modular automation system for the medium and upper performance ranges. Different versions of the controllers allow the performance to be matched to the respective application. Depending on the requirements, the programmable controller can be expanded by input/output modules for digital and analog signals and technology and communication modules. The SIMATIC S7-1500 automation system is seamlessly integrated in the SIMATIC system architecture (Fig. 1.1).



SIMATIC S7-1500 automation system

SIMATIC S7-1500

SIMATIC controllers control the machine or plant. Several versions of the controllers expand the range of use.

SIMATIC HMI

Operator control and monitoring for controlling the plant during operation

SIMATIC NET Networking allows data exchange between devices and online access at any location.

SIMATIC ET200

The distributed I/O expands the interface to the machine or plant.

STEP 7 Professional (TIA Portal)

STEP 7 is the engineering software for configuring and programming.

Fig. 1.1 Components of the SIMATIC S7-1500 automation system

The SIMATIC ET200 distributed I/O allows for additional expansion using input/output modules which are connected to the central controller via PROFIBUS DP or PROFINET IO. The distributed stations can be installed in a control cabinet or – if provided with special designs for increased mechanical requirements – directly on the machine or system.

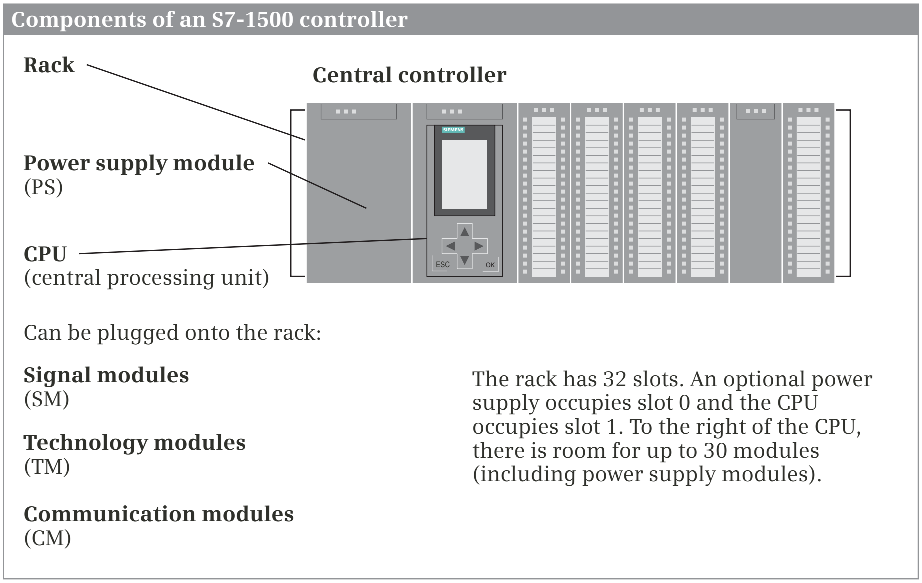
SIMATIC HMI (HMI = Human Machine Interface) is used to control and monitor a machine or plant and its function. Depending on their version, the devices can provide control functions via process images, display system status and alarm messages, and manage the automation data in the form of recipes or measured value archives.

SIMATIC NET handles the exchange of data via various bus systems between the SIMATIC controllers, the distributed I/O, the HMI devices, and the programming device. The programming device can be a personal computer, an industrial PC, or a notebook with a Microsoft Windows operating system.

The SIMATIC components are configured, parameterized, and programmed using the STEP 7 Engineering Software. The TIA Portal (TIA = Totally Integrated Automation) is the central tool for managing automation data and the associated editors in the form of a hierarchically structured project.

1.1.1 SIMATIC S7-1500 programmable controller

The most important components of an S7-1500 programmable controller are shown in Fig. 1.2.



The rack has 32 slots. An optional power supply occupies slot 0 and the CPU occupies slot 1. To the right of the CPU, there is room for up to 30 modules (including power supply modules).

Fig. 1.2 Components of an S7-1500 controller

The **CPU** contains the operating system and the user program. The user program is saved powerfail-proof on the *SIMATIC Memory Card*, which is inserted in the CPU. The user program is executed in the CPU's work memory. The bus interfaces present on the CPU establish the connection to other programmable controllers.

Signal modules (SM) are responsible for the connection to the controlled machine or plant. These input and output modules are available for digital and analog signals with different voltages and currents.

Technology modules (TM) are signal-preprocessing, “intelligent” I/O modules which prepare and process signals coming from the process independent of the CPU and either return them directly to the process or make them available at the CPU's internal interface. Technology modules are responsible for handling functions which the CPU cannot usually execute quickly enough such as counting pulses.

Communication modules (CM) allow data traffic in excess of the functionality provided by the standard interfaces on the CPU with regard to protocols and communication functions.

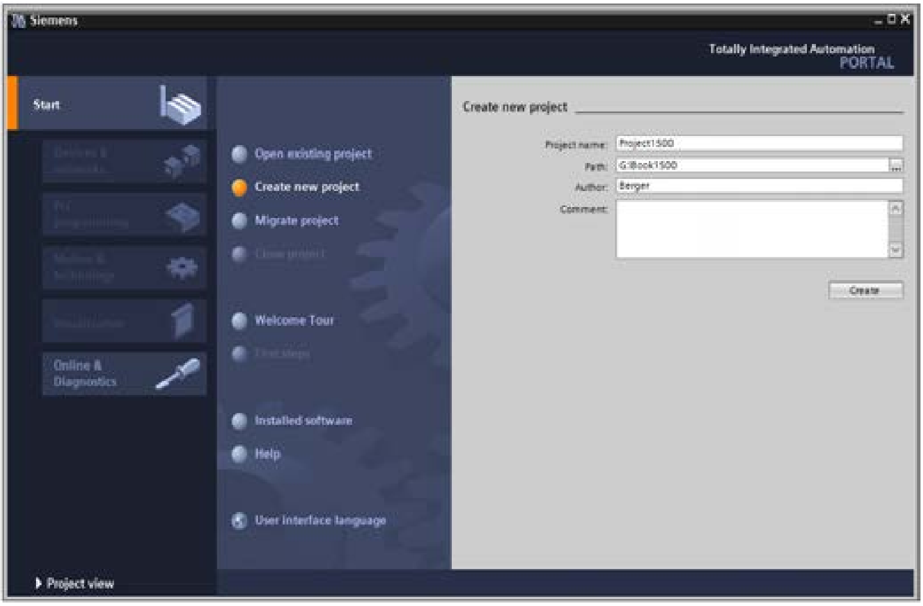
The **(system) power supply modules** provide the internal voltages required by the programmable controller. Up to three system power supply modules can be used in the programmable controller as needed. Load voltages or load currents are provided via external load current supplies (power modules, PM), which can also provide 24 V primary voltage for system power supply modules.

1.1.2 Overview of STEP 7 Professional V12

STEP 7 is the central automation tool for SIMATIC. STEP 7 requires authorization (licensing) and is executed on the current Microsoft Windows operating systems. Configuration of an S7-1500 controller is carried out in two views: the Portal view and the Project view.

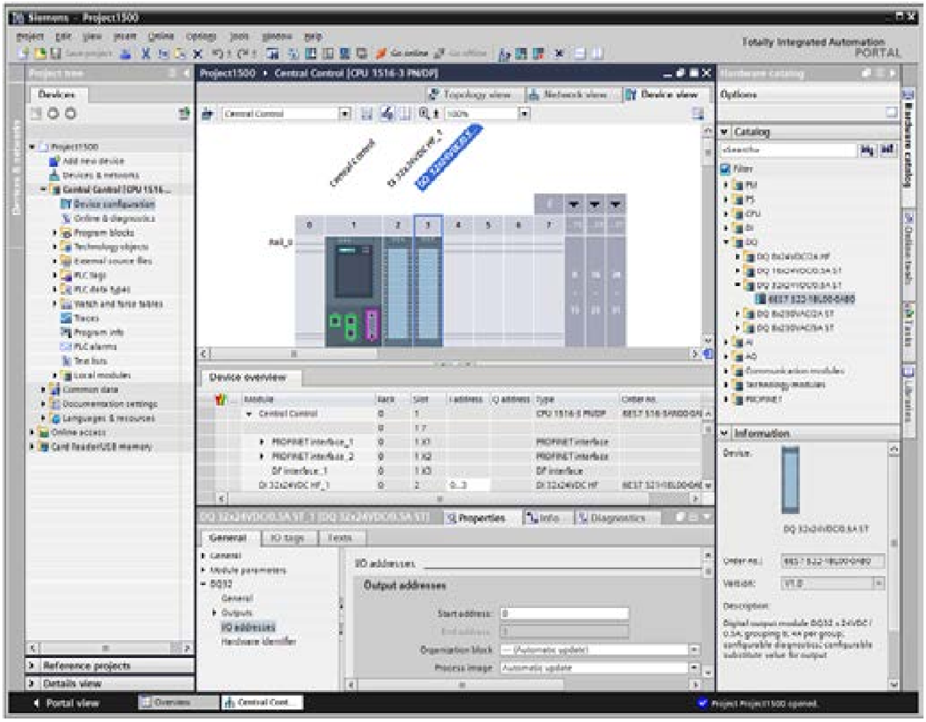
The **Portal view** is task-oriented. In the *Start portal* you can open an existing project, create a new project, or migrate a project. A “project” is a data structure containing all the programs and data required for your automation task. The most important STEP 7 tools and functions can be accessed from here via further portals: The *Devices & networks* portal for hardware configuration, the *PLC programming* portal for processing the user program, the *Motion & technology* portal for generating technology objects, the *Visualization* portal for configuring HMI systems, and the *Online & Diagnostics* portal for the online mode of the programming device (Fig. 1.3).

The **Project view** is an object-oriented view with several windows whose contents change depending on the current activity (Fig. 1.4). 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. A further window – the inspector window – displays the properties of the selected module, and the task card provides support by means of the hardware catalog with the available modules. The Network view allows networking between PLC and HMI stations.



Project name: Project1500
Path: G:\Book1500
Author: Berger
Comment:

Fig. 1.3 Tools in the Start portal of STEP 7 Professional V12



Module	Rack	Slot	I address	Q address	Type	Order no.
Central Control	0	1			CPU 1516-3 PN/DP	6ES7 516-3AN00-0A[illegible]
	0	1 Y				
PROFINET interface_1	0	1 X1			PROFINET interface	
PROFINET interface_2	0	1 X2			PROFINET interface	
DP interface_1	0	1 X3			DP interface	
DI 32x24VDC HF_1	0	2	0...3		DI 32x24VDC HF	6ES7 521-1BL00-0A[illegible]

DQ 32x24VDC/0.5A ST_1 [DQ 32x24VDC/0.5A ST]

I/O addresses

Output addresses

Start address: 0
End address: 3
Organization block: --- (Automatic update)
Process image: Automatic update

Device: DQ 32x24VDC/0.5A ST
Order no.: 6ES7 522-1BL00-0AB0
Version: V1.0
Description: Digital output module DQ32 x 24VDC / 0.5A; grouping 8; 4A per group; configurable diagnostics; configurable substitute value for output

Project Project1500 opened.

Fig. 1.4 Example of a Project view: Working area of the device configuration

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 card contains the program elements catalog with the available program elements and statements. The same applies to the processing of PLC tags or to online program testing using watch tables.

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.

1.1.3 Various programming languages

You can select between five programming languages for the user program: ladder logic (LAD), function block diagram (FBD), structured control language (SCL), statement list (STL), and sequential control (GRAPH).

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 and coils (Fig. 1.5). Complex functions such as arithmetic functions are represented by boxes which you arrange like contacts or coils in the ladder logic.

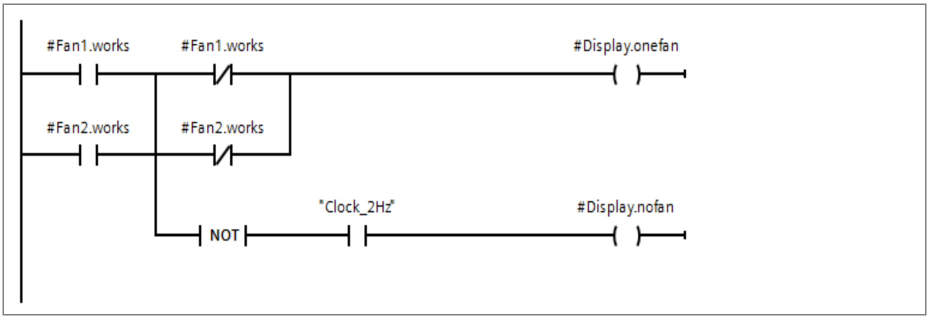


Fig. 1.5 Example of representation in ladder logic

Using the **function block diagram**, you program the control task based on electronic circuitry systems. Binary operations are implemented by linking AND and OR functions and are terminated by memory boxes (Fig. 1.6). Complex boxes are used to handle the operations on digital tags, for example with arithmetic functions.

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).

Using the **statement list**, you program the control task using a sequence of statements. Every STL statement contains the specification of what has to be done, and possibly an operand with which the operation is executed. STL is equally suitable for binary and digital operations and for programming complex open-loop control tasks (Fig. 1.8).

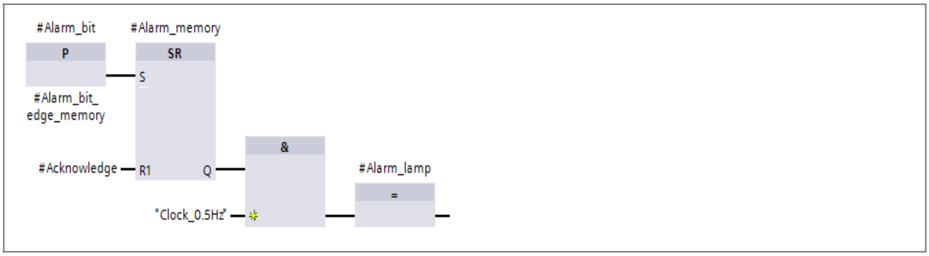
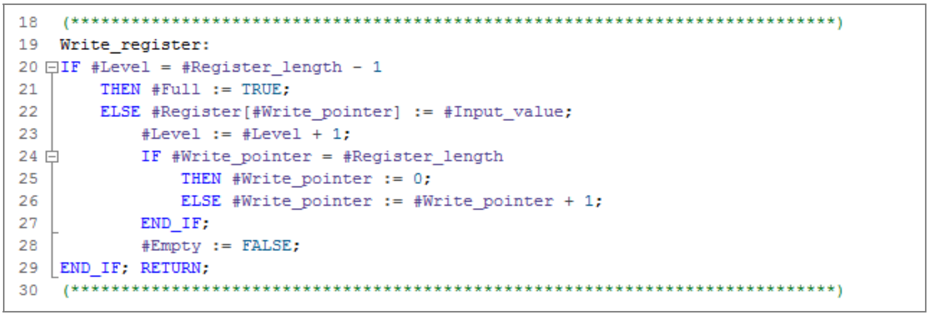
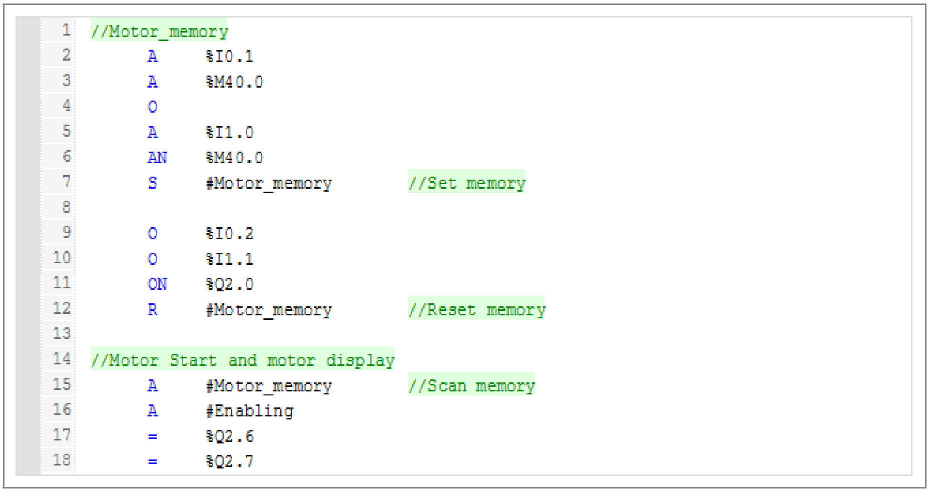


Fig. 1.6 Example of representation in function block diagram



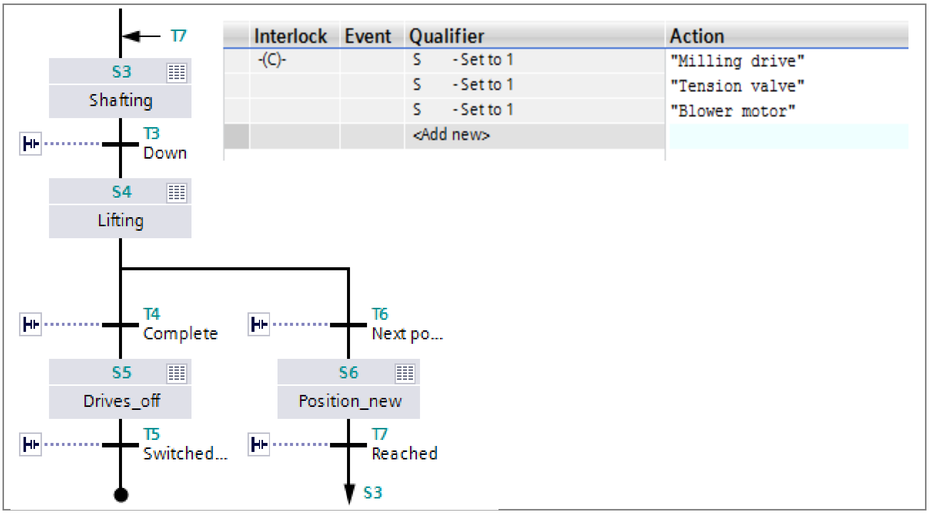
```
(****************************************************************************)
Write_register:
IF #Level = #Register_length - 1
    THEN #Full := TRUE;
    ELSE #Register[#Write_pointer] := #Input_value;
        #Level := #Level + 1;
        IF #Write_pointer = #Register_length
            THEN #Write_pointer := 0;
            ELSE #Write_pointer := #Write_pointer + 1;
        END_IF;
        #Empty := FALSE;
END_IF; RETURN;
(****************************************************************************)
```

Fig. 1.7 Example of SCL statements



```
//Motor_memory
      A     %I0.1
      A     %M40.0
      O
      A     %I1.0
      AN    %M40.0
      S     #Motor_memory        //Set memory

      O     %I0.2
      O     %I1.1
      ON    %Q2.0
      R     #Motor_memory        //Reset memory

//Motor Start and motor display
      A     #Motor_memory        //Scan memory
      A     #Enabling
      =     %Q2.6
      =     %Q2.7
```

Fig. 1.8 Example of STL statements



Interlock	Event	Qualifier	Action
-(C)-		S - Set to 1	"Milling drive"
		S - Set to 1	"Tension valve"
		S - Set to 1	"Blower motor"
		<Add new>	

Fig. 1.9 Example of a GRAPH sequencer and step configuration

Using **GRAPH**, you program a control task as a sequential control in which a sequence of actions prevails. The individual steps and branches are enabled by step enabling conditions which can be programmed using LAD or FBD (Fig. 1.9).

1.1.4 Execution of the user program

After the power supply has been switched on, the control processor checks the existing hardware and parameterizes the modules. A startup program is then executed once, if present. The startup program belongs to the user program which you produce. Modules can be initialized, for example, by the startup program.

The user program is usually divided into individual sections called “blocks”. 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. 1.10).

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. selection of tag values), but with different parameters sets (e.g. for different calculations) (Fig. 1.11).

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 and are the tag memory of the function block. Global data blocks contain data which is not assigned to any block.

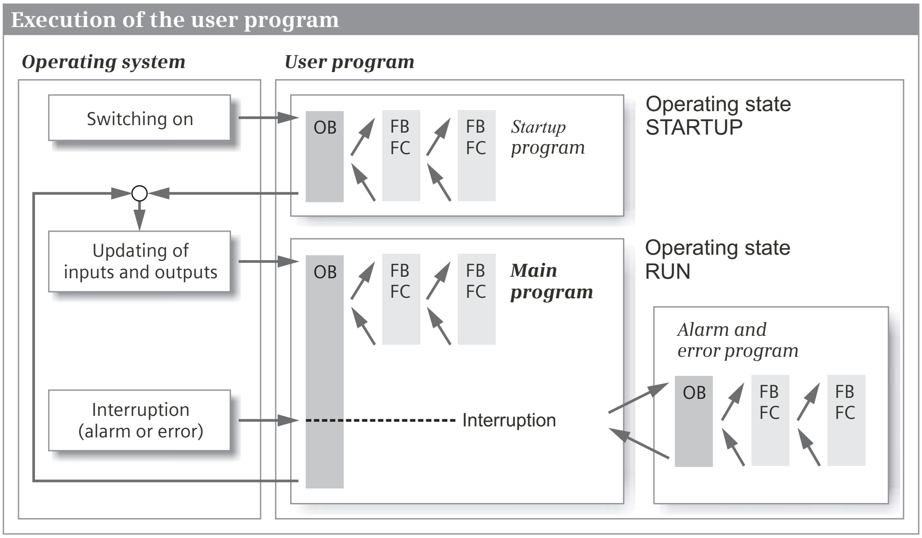


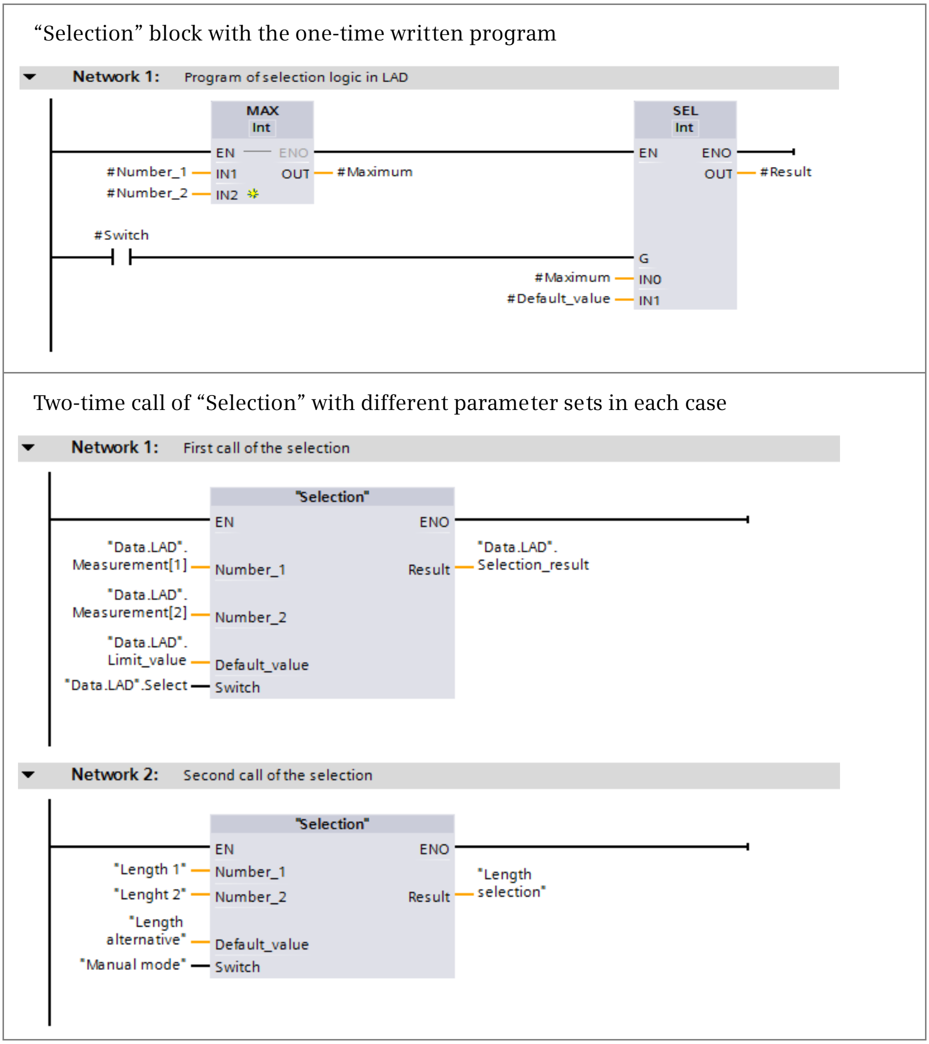
Fig. 1.10 Execution of the user program

Following a startup, 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. 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 logic controllers. The user program is even executed if no actions are requested “from outside”, e.g. 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 controller has a characteristic like, for example, 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. Once the interrupt program has been executed, the control processor continues execution of the main program from the point of interruption.



“Selection” block with the one-time written program

Two-time call of “Selection” with different parameter sets in each case

Fig. 1.11 Multiple use of a block 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. First of all, there is the programming device. All automation data of a STEP 7 project is saved on its hard disk. Configuration and programming of the project data with STEP 7 are carried out in the main memory of the programming device (Fig. 1.12).

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

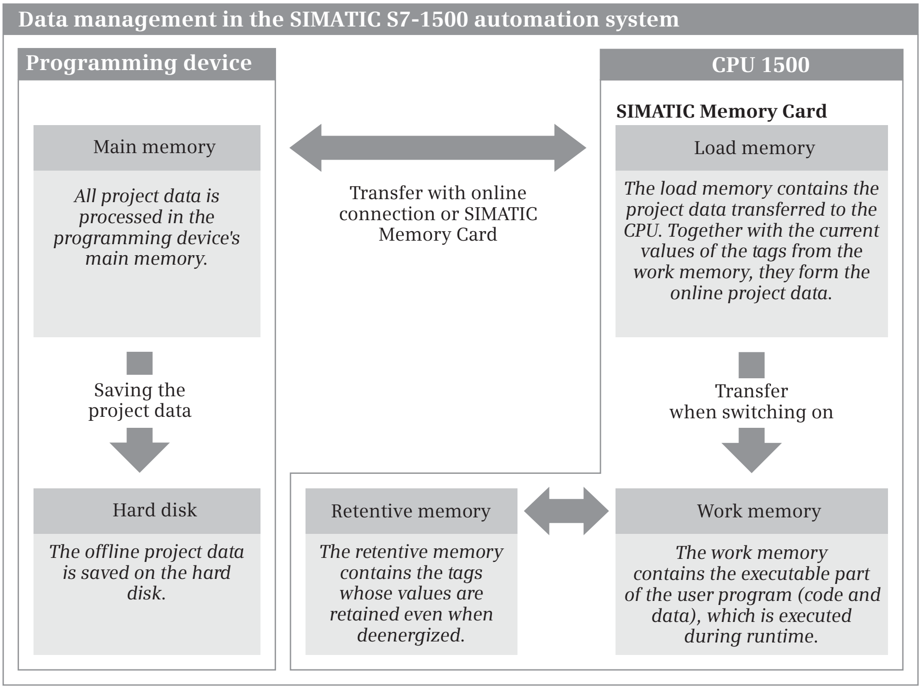

Data management in the SIMATIC S7-1500 automation system
Programming device
CPU 1500
SIMATIC Memory Card
Main memory
All project data is processed in the programming device's main memory.
Transfer with online connection or SIMATIC Memory Card
Load memory
The load memory contains the project data transferred to the CPU. Together with the current values of the tags from the work memory, they form the online project data.
Saving the project data
Transfer when switching on
Hard disk
The offline project data is saved on the hard disk.
Retentive memory
The retentive memory contains the tags whose values are retained even when deenergized.
Work memory
The work memory contains the executable part of the user program (code and data), which is executed during runtime.

Fig. 1.12 Data management in the SIMATIC S7-1500 automation system

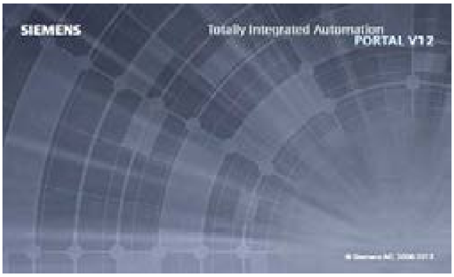
The user memory on the CPU is divided into two components: The *load memory* on the SIMATIC Memory Card – an SD memory card – contains the complete user program with the configured initial data, including the configuration data. The *work memory* contains the executable user program with the current control data.

The project data can be transferred between the programming device and CPU using the SIMATIC Memory Card. The normal case is an online connection for transfer, testing, and diagnostics.

1.2 Introduction to STEP 7 Professional V12

1.2.1 Installing STEP 7

STEP 7 Professional V12 is executed on the operating systems Windows XP Professional SP3, Windows 7 (Professional, Enterprise, Ultimate) SP1 (32-bit and 64-bit), Windows 2003 Server R2 Standard Edition SP2, and Windows 2008 Server Standard Edition SP2. 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.



In order to be able to work with STEP 7, you need a programming device with at least one Core i5, 2.4 GHz processor or a comparable processor. The main memory should have a minimum size of 3 GB for a 32-bit operating system and 8 GB for a 64-bit operating system. On the hard disk, STEP 7 Professional requires approximately 2 GB of free space in the system drive.

For the online connection to the programmable controller, an interface module is required on the programming device for the connection to Industrial Ethernet. If you want to work on the programming device using an SD memory card, you need a corresponding card reader.

Installation, repair, and uninstalling are carried out using the setup program *start.exe* on the DVD. You can also uninstall STEP 7 Professional normally in Windows using the *Software* application (Windows XP) or the *Programs and functions* application (Windows 7) in the Windows Control Panel.

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 Professional. The license for STEP 7 Professional (license key) is provided on a USB flash drive. You will be requested to provide authorization during installation if a license key is not yet present on the hard disk. You can also carry out the authorization following installation of STEP 7.

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. If you lose the license key, e.g. due to a defective hard disk, you can revert to the trial license delivered with STEP 7, which is valid for a limited duration.

The Automation License Manager also manages license keys of other SIMATIC products such as STEP 7 V5.5 or WinCC.

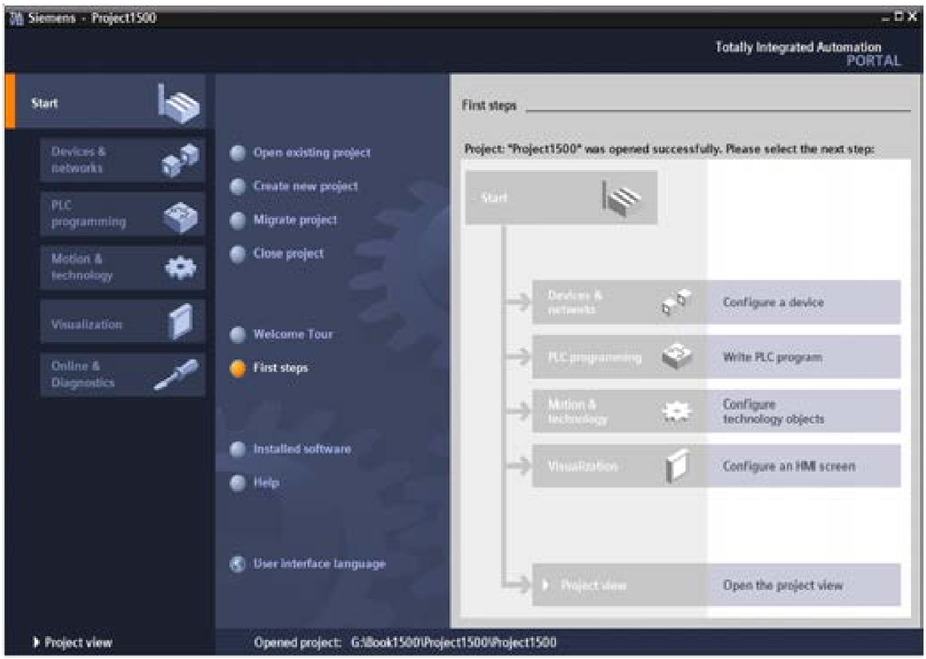
1.2.3 Starting STEP 7 Professional

You start STEP 7 Professional either using the Start button of Windows and *Programs > Siemens Automation > TIA Portal V12*, or by double-clicking on the icon on the Windows desktop. The *Totally Integrated Automation Portal* (TIA Portal) is the software framework in which STEP 7 is embedded. TIA Portal may also contain other applications that use the same database, such as WinCC Professional V12.

TIA Portal V12

1.2.4 Portal view

Following initial starting-up, STEP 7 Professional displays the Start portal. A *portal* provides 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 Professional V12 permits selection of the following portals (Fig. 1.13):



Project: "Project1500" was opened successfully. Please select the next step:

Opened project: G:\Book1500\Project1500\Project1500

Fig. 1.13 Portal view: First steps after opening a project

- ▷ 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.
- ▷ In the *Motion & technology* portal, you create technology objects, such as a PID temperature regulator or a high-speed counter.
- ▷ In the *Visualization* portal, you generate the operator control and monitoring interface for HMI stations. Here you can configure, for example, the process images, the control elements, and alarms.
- ▷ Using the *Online & Diagnostics* portal, you can connect the programming device to a programmable controller, transfer and test programs, and search for (and detect) faults in the automation system.

Additional functions included in the Start portal are: *Create new project*, *Open existing project*, and *Migrate project*. The *Welcome Tour* and *First steps* provide an introduction to STEP 7. *Installed software* provides an overview of further SIMATIC applications that are currently available on the programming device. You can call *Help* in every portal. The *User interface language* allows you to set the language for working with STEP 7.

1.2.5 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.

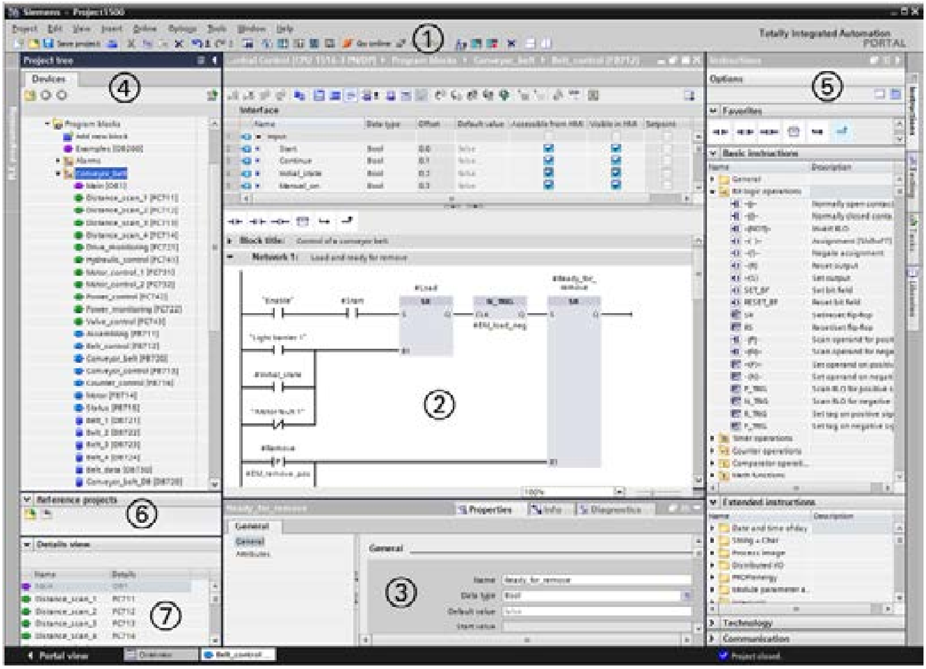


Fig. 1.14 Components of Project view using example of block programming

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* > *Settings* in the main menu, you can adapt the user interface. For example, under *General* you can define the user interface language in which

STEP 7 is used, and the mnemonics (the representation of the operands: “I” for international input, 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 (stations and modules) 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 contains the program. You use the working window to configure the hardware of the automation system, generate the user program, or configure the process images for an HMI device.

③ Inspector window

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

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, HMI and PC 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, for example, 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 by means of 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 Reference projects

The *Reference projects* palette shows the reference projects that are open in addition to the current project. Using the *View > Reference projects* command from the main menu, you can switch the palette display on and off.

⑦ 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. Click on a tab to display its contents in the top level of the working window. This makes it easy to change quickly between different window contents. The status bar on the far right indicates the current status of project execution.

1.2.6 Help information system

During programming, the help function of STEP 7 provides you with comprehensive support for solving your automation task.

To call the help function, click on *Help* in the Portal view or select the *Help > Show help* command in the main menu in the Project view. A window appears with the help information system (Fig. 1.15).

The online help is roughly divided according to the project execution 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 not be included in the online help. A comprehensive description of all available basic and extended statements can be found under *Programming a PLC > References*.

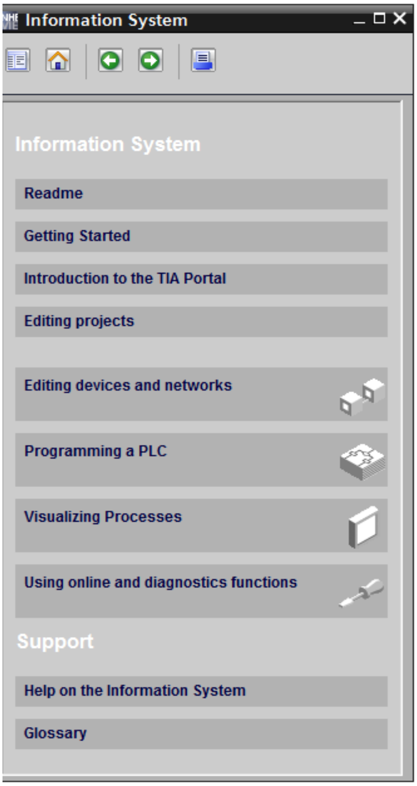

Information System
Information System
Readme
Getting Started
Introduction to the TIA Portal
Editing projects
Editing devices and networks
Programming a PLC
Visualizing Processes
Using online and diagnostics functions
Support
Help on the Information System
Glossary

Fig. 1.15 Start page of the information system

1.2.7 Adapting the user interface

The language of the user interface can be changed. In the main menu, select the *General* section under *Options > Settings*. In the *User 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.

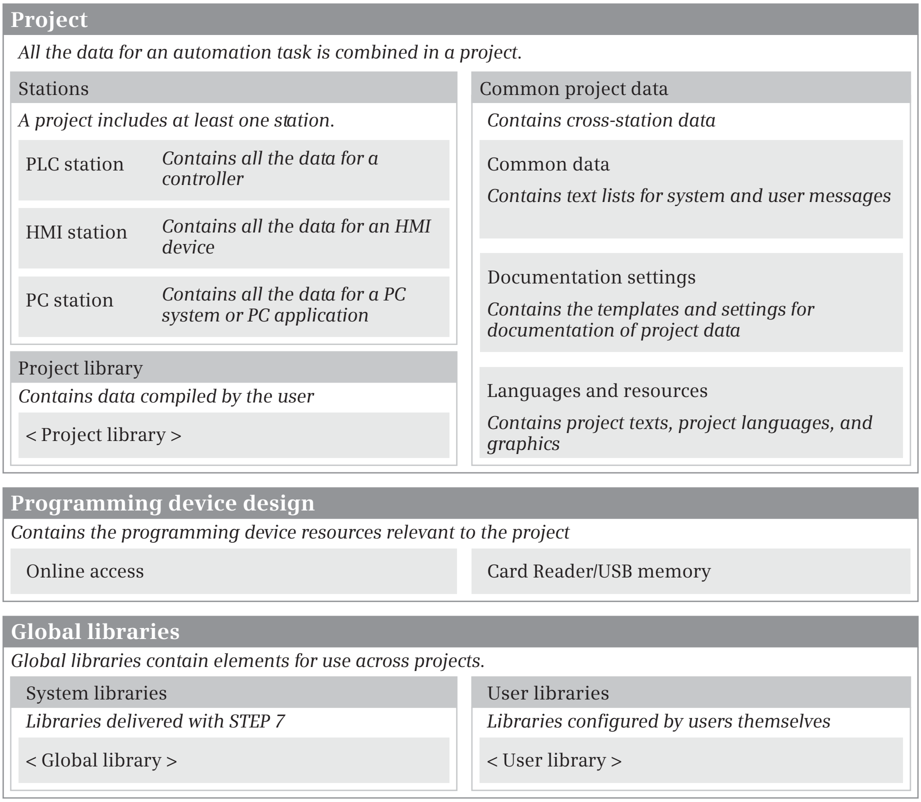
You can show or hide the displayed windows using the menu command *View*. You can always change the size of windows by dragging on its edge with the mouse. Windows can be minimized into symbols which appear in one of the navigation bars in the left, bottom or right margin of the screen.

You can separate the working window completely from the Project view so that it is displayed as a separate window (symbol for *Float* in the title bar of the working window), and also insert it again (symbol for *Embed*). Using the symbol for *Maximize*, all other windows are closed and the working window is displayed in maximum size. 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. 1.16 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



Project

All the data for an automation task is combined in a project.

Stations

A project includes at least one station.

PLC station *Contains all the data for a controller*

HMI station *Contains all the data for an HMI device*

PC station *Contains all the data for a PC system or PC application*

Project library

Contains data compiled by the user

< Project library >

Common project data

Contains cross-station data

Common data

Contains text lists for system and user messages

Documentation settings

Contains the templates and settings for documentation of project data

Languages and resources

Contains project texts, project languages, and graphics

Programming device design

Contains the programming device resources relevant to the project

Online access

Card Reader/USB memory

Global libraries

Global libraries contain elements for use across projects.

System libraries

Libraries delivered with STEP 7

< Global library >

User libraries

Libraries configured by users themselves

< User library >

Fig. 1.16 Project components, libraries, and programming device design

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

A *project library* can be 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 (e.g. LAN adapters) and memory card readers.

1.3.1 Structured representation of project data

The project tree in the Project view displays the project data and the programming device design in a tree structure (Fig. 1.17).

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.

1.3.2 Project data and editors for a PLC station

If you add a PLC station (an S7-1500 controller) to the project, STEP 7 creates the corresponding structure in the project data (Fig. 1.18). A station is always required for editing in a project so that STEP 7 can create the data structures required for programming or configuration. If you wish to write a user program without previously selecting a specific CPU, you can select the “unspecified CPU 1500” from the hardware catalog and replace it later with a “real” CPU 1500.

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 (“main program”, 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 can be made visible using the *Program info* editor (further down in the project tree) in a call and dependency structure.

The *Program blocks* folder contains a *System blocks* subfolder with the system and standard blocks used in the program. This is created automatically when a block of this type is used.

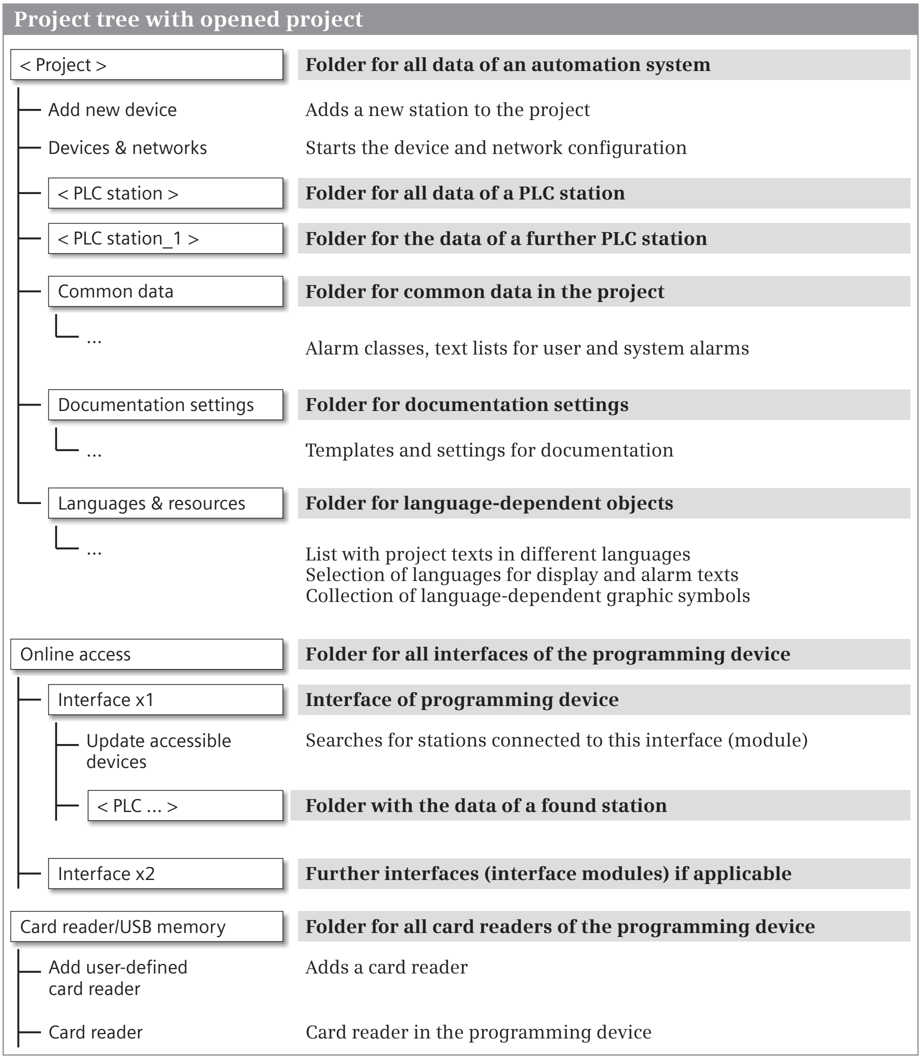
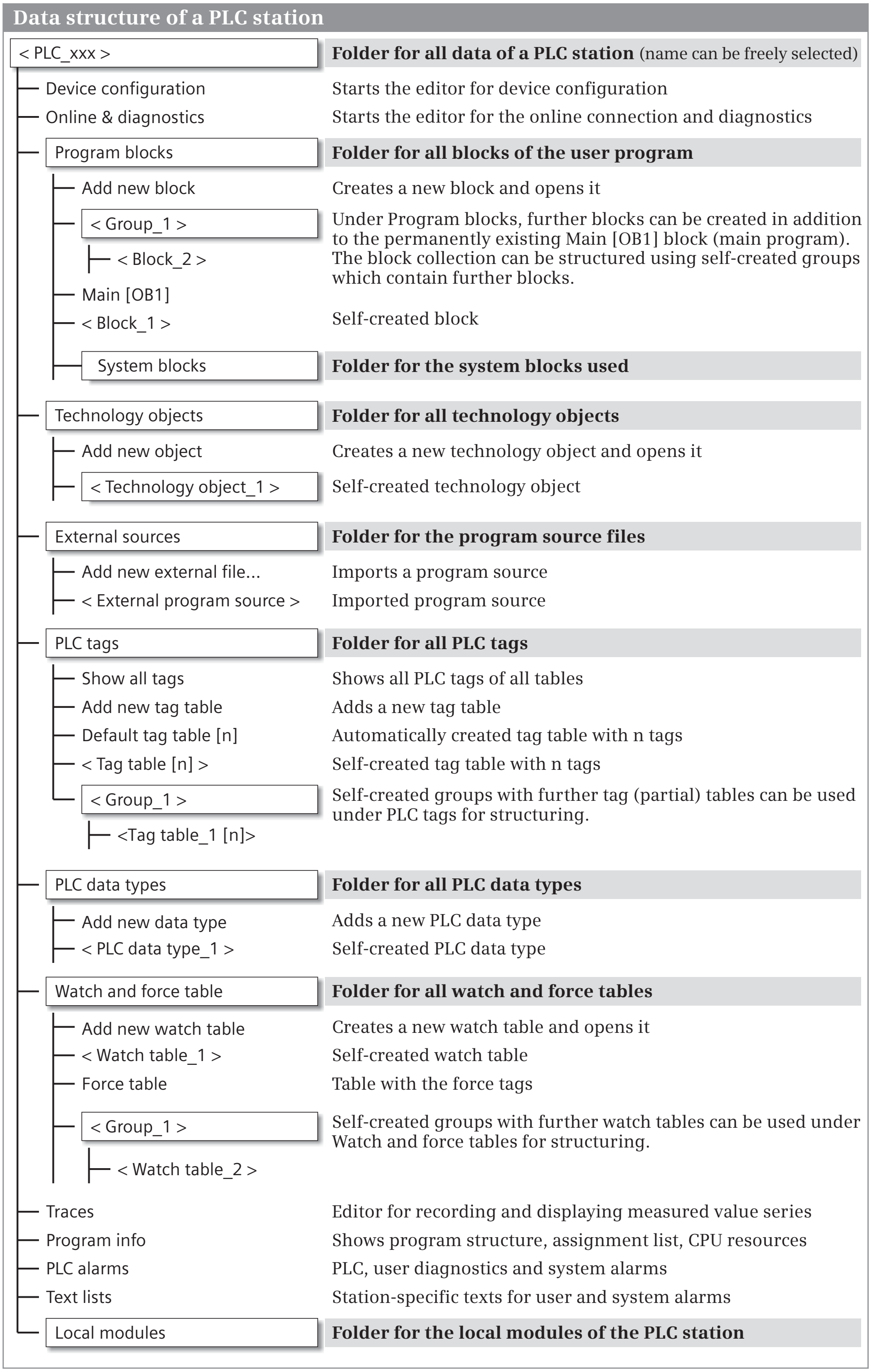

Project tree with opened project
< Project >
Folder for all data of an automation system
Add new device
Adds a new station to the project
Devices & networks
Starts the device and network configuration
< PLC station >
Folder for all data of a PLC station
< PLC station_1 >
Folder for the data of a further PLC station
Common data
Folder for common data in the project
...
Alarm classes, text lists for user and system alarms
Documentation settings
Folder for documentation settings
...
Templates and settings for documentation
Languages & resources
Folder for language-dependent objects
...
List with project texts in different languages
Selection of languages for display and alarm texts
Collection of language-dependent graphic symbols
Online access
Folder for all interfaces of the programming device
Interface x1
Interface of programming device
Update accessible devices
Searches for stations connected to this interface (module)
< PLC ... >
Folder with the data of a found station
Interface x2
Further interfaces (interface modules) if applicable
Card reader/USB memory
Folder for all card readers of the programming device
Add user-defined card reader
Adds a card reader
Card reader
Card reader in the programming device

Fig. 1.17 Project structure in the project tree

The *Technology objects* folder contains the configuration data for the objects of axis controls, control loops (PID controllers), and high-speed counters. A new technology object can be generated using the *Add new object* editor.

The *External sources files* folder contains the program sources for STL and SCL blocks. The *Add new external file* editor is used to import a program source and to save it in this folder. The *External sources files* folder can be configured using self-created subfolders.

The *PLC tags* folder contains the assignment of the absolute address to the symbolic address (name) of inputs, outputs, and bit memories, as well as SIMATIC timer func-



Data structure of a PLC station

< PLC_xxx >	**Folder for all data of a PLC station** (name can be freely selected)
Device configuration	Starts the editor for device configuration
Online & diagnostics	Starts the editor for the online connection and diagnostics
Program blocks	**Folder for all blocks of the user program**
Add new block	Creates a new block and opens it
< Group_1 >	Under Program blocks, further blocks can be created in addition to the permanently existing Main [OB1] block (main program). The block collection can be structured using self-created groups which contain further blocks.
< Block_2 >	
Main [OB1]	
< Block_1 >	Self-created block
System blocks	**Folder for the system blocks used**
Technology objects	**Folder for all technology objects**
Add new object	Creates a new technology object and opens it
< Technology object_1 >	Self-created technology object
External sources	**Folder for the program source files**
Add new external file...	Imports a program source
< External program source >	Imported program source
PLC tags	**Folder for all PLC tags**
Show all tags	Shows all PLC tags of all tables
Add new tag table	Adds a new tag table
Default tag table [n]	Automatically created tag table with n tags
< Tag table [n] >	Self-created tag table with n tags
< Group_1 >	Self-created groups with further tag (partial) tables can be used under PLC tags for structuring.
<Tag table_1 [n]>	
PLC data types	**Folder for all PLC data types**
Add new data type	Adds a new PLC data type
< PLC data type_1 >	Self-created PLC data type
Watch and force table	**Folder for all watch and force tables**
Add new watch table	Creates a new watch table and opens it
< Watch table_1 >	Self-created watch table
Force table	Table with the force tags
< Group_1 >	Self-created groups with further watch tables can be used under Watch and force tables for structuring.
< Watch table_2 >	
Traces	Editor for recording and displaying measured value series
Program info	Shows program structure, assignment list, CPU resources
PLC alarms	PLC, user diagnostics and system alarms
Text lists	Station-specific texts for user and system alarms
Local modules	**Folder for the local modules of the PLC station**

Fig. 1.18 Structure of the project data for a PLC station

tions and SIMATIC counter functions. Example: The symbolic address "Switch on motor" can be assigned to the input with the absolute address %I1.0. A PLC tag is applicable throughout the CPU, it is a "global" tag. The *PLC tags* folder can be configured using self-created subfolders. A subset of the PLC tags is listed in a tag table. The *Show all tags* editor lists all PLC tags used from all tag tables.

The *PLC data types* folder contains user-defined data types. A PLC data type combines various data types in the form of a named data structure. A PLC data type can be assigned to a local tag in a block or serve as a template for the structure of a data block. The *PLC data types* folder can be configured using self-created subfolders.

All created watch tables and the force table 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 table* can be used to assign a fixed value to peripheral inputs and outputs. The *Watch and force tables* folder can be configured using self-created subfolders.

Using the *Traces* editor, the recording of measured value series is planned, the corresponding tasks are sent to the CPU, and the recordings are displayed and managed in tables and graphs in the form of a curve chart.

Program info provides information about

- the *call structure* – which block calls which other block
- the *dependency structure* – which block is called by which other block
- the *assignment list* – which global operands are already used and which addresses are still unused
- the *resources* – how much space is required by the program in the load and work memory

Under *PLC alarms* you see an overview of which program alarms and system alarms are currently present and edit them.

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 alarms 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 of the station. 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. When you start the editor, the data is displayed in the form of a pictorial representation of the programmable controller in the working window and in a register-oriented representation of the module properties in the inspector window. The bottom section of the working window additionally displays the configuration table with the modules as a drop-down list.

Online & diagnostics starts the editor for the online connection and online functions. For example, you can use a (software) control panel in online mode to control the operating states 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 on *Create new project* in the Start portal. 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 are 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 1500 (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 add a new block and open it for editing.
- ▷ Configure an HMI screen (using the supplied WinCC Basic)
  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 alarms, and the HMI simulator. If WinCC Comfort, Advanced or Professional is installed, it is started under this portal.
- ▷ Open the project view
  STEP 7 changes to the Project view in which you can perform the next steps such as adding another PLC station, modifying the configuration of an existing PLC station, adding and programming a block, or configuring the process images for an HMI station.

In the Project view 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. In the Start portal, 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 project*.

Compiling and downloading project data

Before project data can be downloaded to a station, it must be made readable for the respective 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 (only changes)* shortcut menu only compiles those software components which have been changed since the last compilation.

The same applies to downloading of the compiled data to a station. You can select for a PLC station whether you wish to download only the hardware configuration, or only the user program, or both.

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. The settings in the main menu under *Options* > *Settings* and *General* > *Print settings* apply to all projects in the TIA Portal. The templates for the project circuit manual are saved in the project tree in the *Documentation settings* folder. You can add your own templates or change existing ones.

In the global *Documentation templates* library under *Master copies* in the *Document information* group, you can find the templates to design a circuit manual, in the *Frames* group are the templates for the page frames, and in the *Cover Pages* group are the cover page templates. To copy templates to the project, in the *Libraries* task card, open the *Documentation templates* library and drag a template from the *Document information* folder, for example *DocuInfo_ISO_A4_Portrait*, to the *Document information* folder under *Documentation settings*. Copy a cover page from the *Cover Pages* folder to the *Cover pages* folder and a frame from the *Frames* folder to the *Frames* folder.

Double-clicking on a template in the project tree opens the template for editing. For example, you add a new text field or graphical symbol to the cover page. You are supported by the *Toolbox* task card, which contains object templates for a text box, a date/time field, a field for the page number, a field for free text, and a graphic placeholder. In the frame template you complete the title block and in the document information template you enter the data for the circuit manual.

You select the objects to be printed in the project tree or in a library. To display the print preview, select *Print preview...* from the shortcut menu or *Project* > *Print preview... from the main menu*. In the dialog window you can set the document information to be used, select the printout of the cover page and table of contents, and specify whether all project data or a compact selection should be displayed in the print preview.

To print, select the objects to be printed and click on the *Print* icon in the toolbar or select *Project > Print…* in the main menu or *Print… in the shortcut menu*. In the dialog window, you then specify the printer, the document layout, and compact or full printout.

Archiving and retrieving a project

You can reduce the size of the project on the hard disk in two ways:

- ▷ You create a minimized project. This reduces the opened project to its essential components and saves it as a copy. You can open and continue to edit a minimized project as usual.
- ▷ You create a project archive. This reduces the opened project to its essential components and compresses it. The compressed project archive can only be edited further after it is retrieved.

To archive a project, open it. If you make changes to the project, save it before you archive it. Then select the command *Project > Archive…* from the main menu. In the dialog window under *File type*, select either *TIA Portal project minimized* or *TIA Portal project archives* from the drop-down menu. If you want to create a minimized project copy, save the copy under a different name and/or a different directory. A project archive is saved with the file extension .zap12. The project name and project path can be retained.

To retrieve a project, close any open projects and select the command *Project > Retrieve* from the main menu. In the dialog window, specify the name of the project archive with the file extension .zap12 and, in the next dialog window, specify the directory in which the retrieved project is to be saved. Then the retrieved project is opened.

1.3.4 Working with reference projects

You have the capability of opening projects in addition to the current project. These projects are write-protected, i.e. they cannot be modified. You can import individual objects from these “reference projects” into the current project and you can compare a PLC station of a reference project to a station of the current project or a different reference project.

You open a reference project using the *Open reference project* icon in the project tree on the *Reference projects* palette. Select the desired project from the subsequent dialog window and open it.

The read-only reference project is opened. You can open individual objects of this project, but you cannot change them. You can copy individual objects of the reference project into the current project: Select the object in question, press and hold the mouse button, and “drag” the object into the current project. You can process the copied object further here.

To compare two PLC stations, select the station and then select the command *Compare > Offline/offline* from the shortcut menu. The station is displayed in the left pane of the compare editor. Now press and hold the mouse button and “drag” the PLC station to be compared into the header of the right pane. This can be a station from a reference project or from a library. The compare editor marks different objects with symbols (green circle: no differences, semi-circles in various colors: differences exist, unfilled semi-circle: object does not exist). You can select individual objects and start a detailed comparison via the shortcut menu if the type of the object allows it. Actions such as overwriting an object are not possible for a reference project. You can compare additional stations by “dragging” the corresponding station into the header of one of the panes.

1.3.5 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. The library contents can be listed with the symbol open or close the element view in the *Elements* pallet in the *Details mode*, *List mode*, or *Overview mode*. The *Info* pallet shows further information on the selected library element.

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 in read-only mode.

To create a global library, open the *Libraries* task card in the task window and click on the *Create new global library* icon in the *Global libraries* palette. In the dialog window, specify the name and path of the library before you click on the *Create* button. Using the other symbols in the *Global libraries* palette, you can open a global library, save the changes to the library, and close the library.

2 SIMATIC S7-1500 automation system

2.1 S7-1500 station components

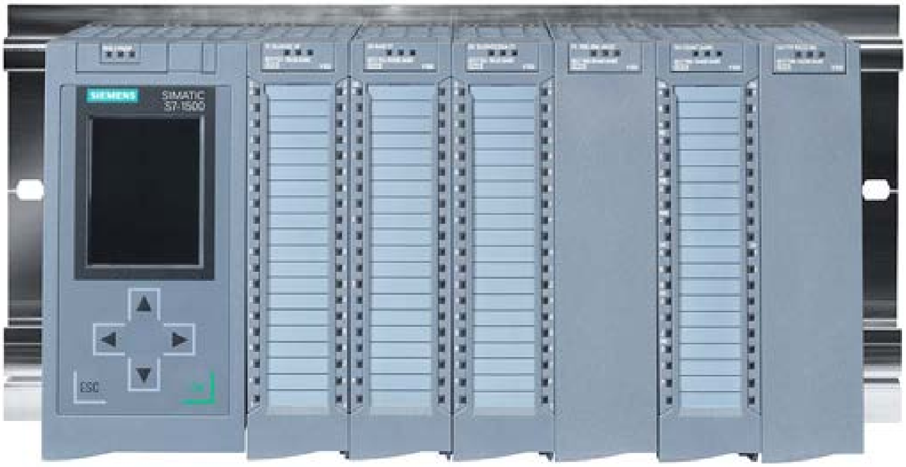


Fig. 2.1 S7-1500 station with CPU 1516-3 PN/DP

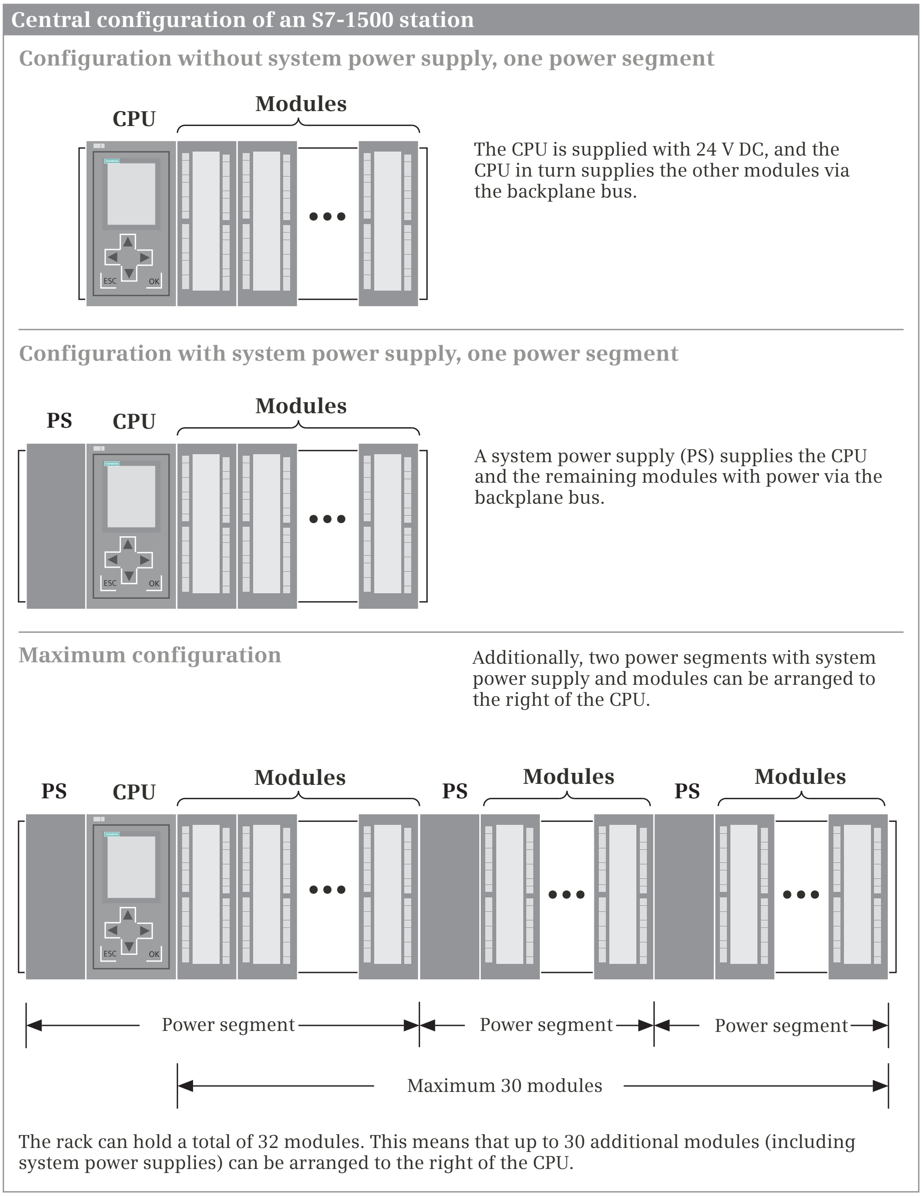
A programmable controller including all I/O modules is referred to as a “station”. An S7-1500 station can contain the following components:

- ▷ Rack
- ▷ Power supply (PS)
- ▷ Central processing unit (CPU)
- ▷ Input/output modules (signal modules, SM)
- ▷ Technology modules (TM)
- ▷ Communication modules (CM)

A station can also encompass distributed I/O which is connected to the CPU or a communication module via a PROFINET IO or PROFIBUS DP bus system.

Design variants

An S7-1500 station comprises one rack with a maximum of 32 slots. It can be divided into as many as three “power segments”. A power segment comprises a current source (PS or CPU) and the modules to be supplied as current sinks. The number of modules a power segment encompasses depends on the electrical power that is provided and consumed. An additional load current supply is needed for supplying the sensors and actuators (Fig. 2.2).


Central configuration of an S7-1500 station
Configuration without system power supply, one power segment
CPU
Modules
ESC
OK
Configuration with system power supply, one power segment
PS
CPU
Modules
ESC
OK
Maximum configuration
PS
CPU
Modules
PS
Modules
PS
Modules
ESC
OK
Power segment
Power segment
Power segment
Maximum 30 modules

The CPU is supplied with 24 V DC, and the CPU in turn supplies the other modules via the backplane bus.

A system power supply (PS) supplies the CPU and the remaining modules with power via the backplane bus.

Additionally, two power segments with system power supply and modules can be arranged to the right of the CPU.

The rack can hold a total of 32 modules. This means that up to 30 additional modules (including system power supplies) can be arranged to the right of the CPU.

Fig. 2.2 Design variants of an S7-1500 station

If a power supply module is used for the first power segment, it is plugged into the first slot on the far left (slot 0). The CPU is always plugged into slot 1 next to it. To the right of the CPU, there is room for another 30 modules, including any additional system power supply modules. Each module occupies one slot independent of its width. The modules must be inserted without gaps.

The power supply for the module electronics and the data exchange between the modules is accomplished via the backplane bus. The backplane bus is made up of “U-connectors” between the modules. One U-type-connector is needed for each module.

2.2 S7-1500 CPUs

2.2.1 CPU versions

CPUs for S7-1500 are available in several versions for different applications. Common to all CPUs is the scope of control functions (operands, tag types, data types, binary logic operations, fixed-point and floating-point arithmetic, etc.). Within the versions, the CPUs differ in their memory size, the range of operands, and the processing speed (Table 2.1).

Standard controllers

Three versions of standard-design controllers are currently available: CPU 1511-1 PN, CPU 1513-2 PN, and CPU 1516-3 PN/DP.

It is possible to connect to Industrial Ethernet using the PN interface. Each CPU can be both an IO controller and an “intelligent” IO device on PROFINET IO. A CPU with a DP interface can be the DP master on PROFIBUS DP.

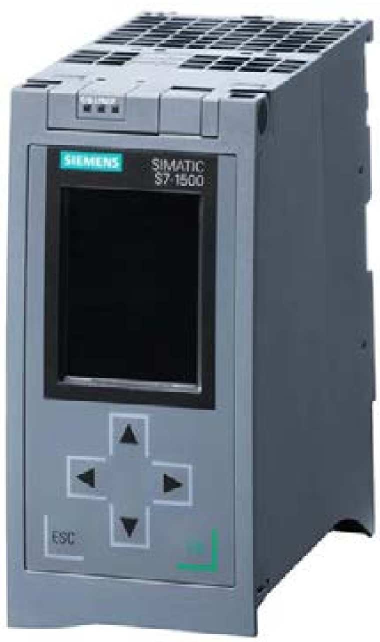


Fig. 2.3 CPU 1516-3 PN/DP

2.2.2 Control and display elements

The control panel with the display and status LEDs above the control panel are located on the front side of the CPU. The mode switch, slot for the SIMATIC Memory Card, and interface connections are located behind the control panel.

Table 2.1 Selected data of a CPU 1500 with Firmware V1.1

	CPU 1511-1 PN	CPU 1513-2 PN	CPU 1516-3 PN/DP
User memory			
Work memory			
for program	150 KB	300 KB	1 MB
for data	1 MB	1.5 MB	5 MB
Retentive memory	128 KB	128 KB	128 KB
Load memory on the memory card up to	2 GB	2 GB	2 GB
Hardware configuration			
Racks	1	1	1
Modules per rack	max. 32	max. 32	max. 32
Address ranges			
in the process image	32 KB inputs, 32 KB outputs	32 KB inputs, 32 KB outputs	32 KB inputs, 32 KB outputs
per IO subsystem	8 KB inputs, 8 KB outputs	8 KB inputs, 8 KB outputs	8 KB inputs, 8 KB outputs
Blocks			
Number (total)	2000	2000	6000
OB/FB/FC size	150 KB	300 KB	512 KB
DB size	1 MB	1.5 MB	5 MB
Bit memory	16 KB	16 KB	16 KB
SIMATIC timers	2048	2048	2048
SIMATIC counters	2048	2048	2048
Temporary local data			
per priority class	64 KB	64 KB	64 KB
per block	16 KB	16 KB	16 KB
Interfaces			
PROFINET	1 (IO controller/device)	1 (IO controller/device)	1 (IO controller/device) 1 (Industrial Ethernet)
PROFIBUS	–	–	1 (DP master)
Interfaces via CM	4 (PROFINET + PROFIBUS)	6 (PROFINET + PROFIBUS)	8 (PROFINET + PROFIBUS)
Number of connections			
maximum	96	128	256
reserved for PG, HMI, and Web server	10	10	10
via integrated interfaces	64	88	128
Execution times			
for binary operations	60 ns/statement	40 ns/statement	10 ns/statement
for word operations	72 ns/statement	48 ns/statement	12 ns/statement
for fixed-point arithmetic	96 ns/statement	64 ns/statement	16 ns/statement
for floating-point arithmetic	384 ns/statement	256 ns/statement	64 ns/statement

Status LEDs

The operating state of the CPU is indicated by LEDs on the front side above the control panel:

STOP/RUN	Continuous yellow light in STOP operating state Continuous green light in RUN operating state Flashing light in STARTUP operating state
ERROR	Flashing red light in the event of an error Continuous red light if hardware is defective
MAINT	Continuous yellow light indicates a maintenance request

Display and control keys on the control panel

The color display shows – structured in several menus – the status and properties of the CPU, diagnostics alarms, the date/time, and information about the inserted modules.

The control keys are designed as membrane keyboard. These can be used to select the menus in the display and to set the date, time, access protection, language, and IP address. The memory of the CPU can also be reset to the factory settings.

The control panel can be replaced during ongoing operation.

The control panel can be secured using a lead-wire seal or with a bracket lock in order to prevent unauthorized operation of the mode switch or unauthorized removal of the memory card.

Mode switch

The mode switch is designed as a toggle switch with the positions RUN, STOP, and MRES. In the RUN position, the user program is executed and the programming device has unlimited access to the CPU.

The user program is not executed in the STOP position, but the CPU retains its communication capability. For example, a new user program can be downloaded using the programming device or the diagnostics buffer can be read out with the CPU at STOP.

In the MRES position (master reset), the CPU parameters are reset. MRES functions like a pushbutton. A memory reset can be carried out for the CPU using a special input sequence, or it can be reset to the delivered state.

2.2.3 SIMATIC Memory Card

The SIMATIC Memory Card is an SD memory card (secure digital memory card), which is pre-formatted by Siemens.

The data is stored retentive on the memory card, but can be read, written, and deleted like with a RAM. This feature permits data backup without a battery.

The complete load memory is present on the memory card, meaning that a memory card is always required to operate a CPU 1500.

The memory card can be used as a portable storage medium for user programs or firmware updates. You can apply the user program to read or write data blocks on the memory card by means of special system functions, for example read recipes from the memory card or create a measured value archive on the memory card and supply it with data.

The SIMATIC Memory Card is available for various memory capacities up to 2 GB. Please note that formatting the memory card using Windows tools makes it unusable for a CPU 1500.

Fig. 2.4 SIMATIC Memory Card

The SIMATIC Memory Card has a serial number to which you can “tie” program blocks. This means that the user program is only loaded into the CPU if the “correct” memory card is inserted (copy protection similar to a dongle).

2.2.4 Memory areas in an S7-1500 station

Fig. 2.5 shows the memory areas in the programming device, in the CPU, and in the signal modules which are important for the user program.

The programming device contains the offline data. This consists of the user program (program code and user data), the system data (e.g. hardware, network and connection configuration), and further project-specific data such as the PLC tag table.

The signal modules contain memories for the signal states of the input and output signals.

The online data consists of the user program and the system data which is located in three memory areas: in the load memory, in the work memory, and in the system memory.

Load memory

The load memory contains the complete user program including configuration data (system data). The load memory is located entirely on the SIMATIC Memory Card. The user program is always initially transferred from the programming device to the load memory, and then from there to the work memory. The program in the load memory is not executed as the user program.

Data blocks that contain recipes, for example, can be identified as “not relevant to execution”, and in this case they are not transferred to the work memory. These data blocks can be accessed from the user program using system functions.

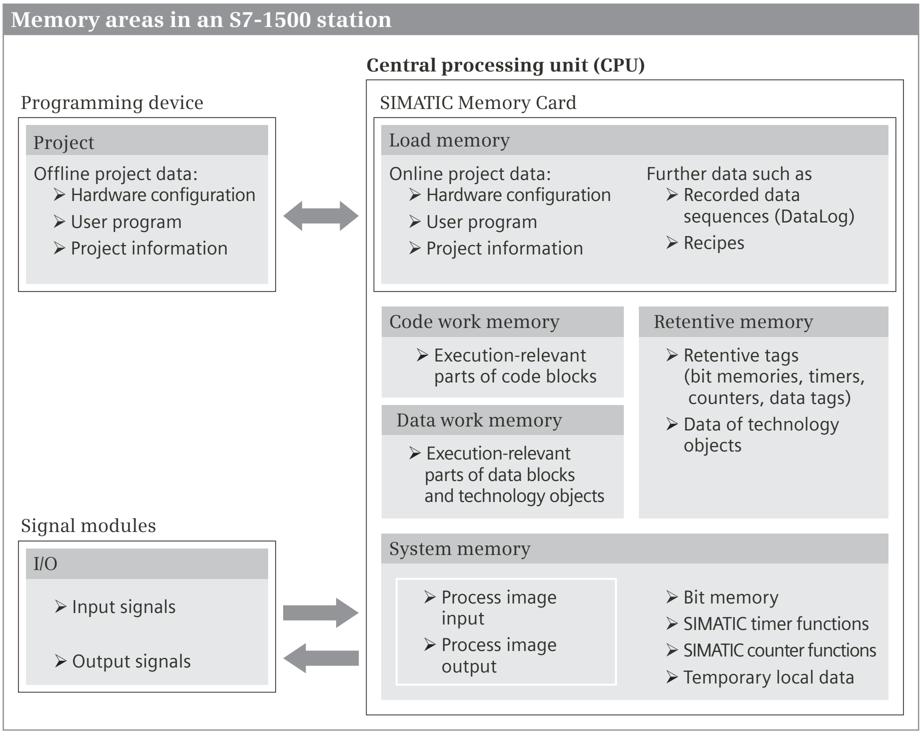


Fig. 2.5 Memory areas for the user program

Work memory

The work memory is designed as a fast RAM completely integrated in the CPU. The CPU's operating system copies the “execution-relevant” program code and the user data into the work memory. “Execution-relevant” is a property of the existing objects, and is not tantamount to the fact that a specific code block is actually called and executed. The “actual” user program is executed in the work memory.

The work memory of a CPU 1500 consists of two parts: The code work memory contains the program code. The data work memory contains the user data and the data of the technology objects.

When uploading the complete user program to the programming device, the blocks are fetched from the load memory, supplemented by the current values of the data from the work memory.

System memory

The system memory contains the process images for the inputs and outputs. These are copies of the input and output signals from the modules. The system memory also contains the operand areas Bit memories, SIMATIC timer/counter functions, and Temporary local data. The temporary local data are intermediate memories for program execution in the blocks of the user program.

Retentive memory

The retentive memory contains the bit memories, SIMATIC timer-/counter functions and data tags that are defined as retentive. The values in the retentive memory are retained after a power failure or if the power supply is switched off and on. The values are deleted if the memory is reset or if the CPU is reset to the factory settings.

2.2.5 Bus interfaces

Each CPU 1500 has an integrated PROFINET interface (PN interface) with two ports for setting up a linear topology. The CPU 1516-3 PN/DP also has an additional PROFINET interface with a port for connecting to Industrial Ethernet and a PROFIBUS DP interface (Fig. 2.6).

The first PN interface connects the CPU to a PROFINET IO system. The CPU can be operated as IO controller or as IO device. The PN interface has two ports which are interconnected by a switch. This permits simple configuration of a quasi-linear topology. A programming device or an operator control and display unit can also be connected to the PN interface. Data transfer to other devices is possible using open user communication over Industrial Ethernet.

The second PN interface connects the CPU to Industrial Ethernet. It has its own IP address, which makes it possible to connect to a company network separately from the process subnetwork. A programming device or an operator control and display unit can also be connected to the port of this PN interface. Data transfer to other devices is possible using open user communication over Industrial Ethernet.

The DP interface connects the CPU to the PROFIBUS DP bus system. The CPU is the DP master.

Routing of data records is possible via the PN and DP interfaces, i.e. data can be transmitted beyond the limits of subnets. These interfaces also support time synchronization.

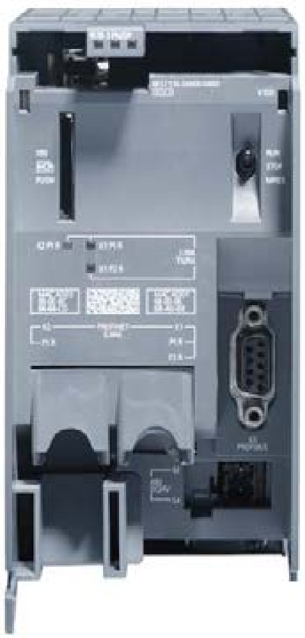


Fig. 2.6
The bus connections and control elements under the front flap of a CPU 1516-3PN/DP

The bus interfaces are numbered: X1 for the first interface (PN) with ports P1 and P2, with CPU 1516: X2 for the second interface (Ethernet) and X3 for the DP interface.

2.3 Signal modules

Signal modules (SM) are peripheral input/output modules which establish the connection between the CPU and the machine or process. The following types of module are available for SIMATIC S7-1500:

▷ SM 521	Digital input modules
▷ SM 522	Digital output modules
▷ SM 531	Analog input modules
▷ SM 532	Analog output modules

A signal module can be inserted in the rack at one of the slots 2 to 31.

Common properties

A green RUN LED and a red error LED indicate the operating state of the module. On most of the modules, a green power LED indicates the presence of the load voltage.

Correspondingly configured modules provide a statement about the validity of the process signal along with the value status. If the value status is activated, the information in the process image input is available. In the value status, one bit per process channel indicates with signal state “0” that the assigned process signal or the assigned analog value is invalid. For input modules, the value status lies in the connection to the user data in the process image. For output modules, input bytes are also occupied for the value status.

2.3.1 Digital input modules

The digital input modules are used by the CPU to record the operating states of the controlled machine or plant. These modules are signal conditioners for binary process input signals. Process signals present with a DC or AC voltage level from 24 V to 230 V are converted into signals with an internal level.

If a module converts a positive (DC) voltage at the input into signal state “1”, it is called a “sinking input”. A “sourcing input” converts a positive voltage at the input into signal state “0”. Further information can be found in Chapter 12.1.2 “Working with binary signals” on page 504.

Depending on the module, the input channels are isolated either individually or in groups. There are simple input modules and modules with diagnostic capability with hardware and diagnostic interrupt triggering (Table 2.2).

The digital input modules have two or four bytes, corresponding to 16 or 32 input signals. The presence of a process signal is indicated by an LED at the input channel.

Table 2.2 Overview of digital modules

Module type	Short designation	Description
SM521 digital input modules	DI 16 × 24 V DC HF	Diagnostic interrupt, hardware interrupts, isochronous mode, value status, sinking input
	DI 32 × 24 V DC HF	
	DI 16 × 24 V DC SRC BA	Sourcing input
	DI 16 × 230 V AC BA	–
SM522 digital output modules	DQ 16 × 24 V DC / 0.5 A ST	Transistor output, diagnostic interrupt, isochronous mode, value status
	DQ 32 × 24 V DC / 0.5 A ST	
	DQ 8 × 24 V DC / 2 A HF	Transistor output, diagnostic interrupt, value status
	DQ 8 × 230 V AC / 2 A ST	Triac output
	DQ 8 × 230 V AC / 5 A ST	Relay output, diagnostic interrupt, value status

2.3.2 Digital output modules

The digital output modules are used by the CPU to control the connected machine or plant. These modules are signal conditioners for binary process output signals (Fig. 2.7). The internal signals are amplified and output in the following current and voltage ranges (rated values):

- With electronic amplifiers from 24 V DC and a current of 0.5 A and 2 A
- With electronic amplifiers from 120 V to 230 V AC and a current of 2 A
- With relay contacts with a DC voltage of 24 V or an alternating voltage of 230 V and a current of up to 5 A

Depending on the module, the output channels are isolated either individually or in groups. The module types include simple digital output modules, digital output modules with diagnostic capability, and modules with or without integral short-circuit protection (Table 2.2).

The digital output modules have two or four bytes, corresponding to 16 or 32 output signals. All modules indicate a delivered process signal by means of an LED on the output channel.

The digital output modules are disabled in the STOP and STARTUP operating states. In this case they deliver either a configured substitute value or retain the last output value.

Fig. 2.7 SM 522 digital output module, DQ 32 × 24 V DC / 0.5 A ST

2.3.3 Analog input modules

The CPU can use analog input modules to process analog measured variables after they have been converted into digital values by the modules. These modules are signal conditioners for analog process input signals (Fig. 2.8).

Voltage and current transmitters, thermocouples, resistors or thermoresistors can be connected to the modules depending on the design. The measuring range can be set as desired per channel or per channel group. The resolution is 16 bits including sign. An analog value (a channel) occupies 16 bits, in other words two bytes. The analog input modules have 8 channels, corresponding to an address range of 16 bytes (Table 2.3).

One status LED per channel indicates whether the channel is deactivated, working properly, or an error has occurred.

The input channels are not isolated from each other. There is galvanic isolation between the channels and the backplane bus and between the channels and the internal power supply.

Fig. 2.8
SM 531 analog input module, AI 8 × U/I/RTD/TC ST

Table 2.3 Overview of analog modules

Module type	Short designation	Description
SM531 analog input modules	AI 8 × 16 bit ST	For the measurement types voltage, current, resistor, thermoresistor, and thermocouple Diagnostic interrupt, hardware interrupt
	AI 8 × 16 bit HS	For the measurement types voltage and current Diagnostic interrupt, hardware interrupt, isochronous mode
SM532 analog output module	AO 4 × 16 bit ST	For the output types voltage and current Diagnostics interrupt
	AO 4 × 16 bit HS	For the output types voltage and current Diagnostic interrupt, isochronous mode

2.3.4 Analog output modules

The CPU can use analog output modules to continuously provide actuators with analog setpoints. These modules are signal conditioners for analog process output signals (Table 2.3).

The modules can output a voltage value in the ranges of 0 to 10 V, 1 to 5 V, or -10 to +10 V or a current value in the ranges of 0 to 20 mA, -20 to +20 mA, or 4 to 20 mA. The resolution is 16 bits including sign. The output channels are not isolated from each other. There is galvanic isolation between the channels and the backplane bus and between the channels and the load voltage L+.

An analog value (an analog channel) occupies 16 bits, in other words two bytes. The analog output modules have 8 channels, corresponding to an address range of 16 bytes. One status LED per channel indicates whether the channel is deactivated, working properly, or a diagnosis event has occurred.

The analog output modules are disabled in the STOP and STARTUP operating states. In this case they deliver either a configured substitute value or retain the last output value.

2.4 Technology modules

Technology modules (TM) are signal-preprocessing, “intelligent” modules which prepare and process signals coming from the process independent of the CPU, and either return them to the process or make them available to the user program at the CPU's internal interface. They are responsible for handling functions which the CPU cannot usually execute quickly enough, such as counting pulses (Fig. 2.9).

The following technology modules are available:

▷ TM Count 2 × 24 V (6ES7 550-1AA0-0AB0)

Technology module for counting pulses, for measuring a frequency, time period or velocity, and for position detection for motion control; with two 32-bit counter channels and a maximum signal frequency of 200 kHz (this corresponds to max. 800 kHz with fourfold evaluation); for connection of 24 V incremental encoders with and without N signal, 24 V incremental encoders with and without direction signal, and 24 V incremental encoders with separate signals for counting up and counting down; three configurable digital inputs per channel for starting, stopping, synchronizing or saving the count value (capture function), and two configurable digital outputs for outputting a comparison result.

Fig. 2.9 TM Count 2 × 24 V counter module

▷ TM PosInput 2 (6ES7 551-1AB00-0AB0)

Technology module for counting pulses, for measuring a frequency, time period or velocity, and for position detection for motion control; with two 32-bit counter channels and a maximum signal frequency of 1 MHz (this corresponds to max. 4 MHz with fourfold evaluation); for connection of SSI absolute value encoders, RS 422/TTL incremental encoders with and without N signal, RS 422/TTL incremental encoders with and without direction signal, and RS 422/TTL incremental encoders with separate signals for counting up and counting down; two configurable digital inputs per channel for starting, stopping, synchronizing or saving the count value (capture function), and two configurable digital outputs for outputting a comparison result.

2.5 Communication modules

The communication modules (CM) relieve the CPU of 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 communication services for the CPU and the user program (Fig. 2.10).

The following communication modules are available:

▷ CM PTP RS232 BA (6ES7 540-1AD00-0AA0)

Communication module for point-to-point connection to an interface; physical transmission characteristics RS 232 with up to 19.2 Kbit/s; maximum frame length: 1 KB; supported protocols: Freeport and 3964 (R), USS protocol via system functions.

▷ CM PTP RS422/485 BA (6ES7 540-1AB00-0AA0)

Communication module for point-to-point connection to an interface; physical transmission characteristics RS 422/485 with up to 19.2 Kbit/s; maximum frame length: 1 KB; supported protocols: Freeport and 3964 (R), USS protocol via system functions.

▷ CM PTP RS232 HF (6ES7 541-1AD00-0AB0)

Communication module for point-to-point connection to an interface; physical transmission characteristics RS 232 with up to 115.2 Kbit/s; maximum frame length: 4 KB; supported protocols: Freeport, 3964 (R) as well as Modbus RTU Master and Modbus RTU Slave, USS protocol via system functions.

Fig. 2.10
CM PtP RS232 BA communication module

▷ CM PTP RS422/485 HF (6ES7 541-1AB00-0AB0)

Communication module for point-to-point connection to an interface; physical transmission characteristics RS 422/485 with up to 115.2 Kbit/s; maximum frame length: 4 KB; supported protocols: Freeport, 3964 (R) as well as Modbus RTU Master and Modbus RTU Slave, USS protocol via system functions.

▷ CM 1542-5, PROFIBUS (6GK7 542-5DX00-0XE0)

Communication module for PROFIBUS; physical transmission characteristics RS 485 with up to 12 Mbit/s; operation as DPV1 master or DPV1 slave; PG/OP communication, S7 communication; open user communication.

▷ CP 1543-1, Industrial Ethernet (6GK7 543-1AX00-0XE0)

Communication module for Industrial Ethernet up to 1000 Mbit/s; TCP/IP, ISO, UDP, IP broadcast/multicast, open user communication, among others; addressing with IPv4/IPv6; can be used for safety applications.

2.6 Other modules

2.6.1 System power supply modules

The system power supply modules (PS) provide the operating voltage for the modules in the rack.

Depending on the power supply module, the primary voltage is either an alternating voltage of 120/230 V (PS 507) with an output power of 60 W or a 24 V direct voltage (PS 505) with an output power of 25 W and 60 W.

A green RUN LED and a red error LED indicate the operating state of the module. A yellow MAINT LED signals a maintenance request.

Fig. 2.11
PS 505 24 V/25 W
power supply

2.6.2 Load power supply modules

The load power supply modules (power modules, PM) provide 24 V direct voltage, which can be used as the supply voltage for sensors and actuators (load power supply of the I/O modules), CPUs, and system power supply modules. The modules are in the S7-1500 design.

The primary voltage of the PM 1507 load power supply modules is 120/230 V AC with an output power of 70 W and 190 W.

A green RUN LED and a red error LED indicate the operating state of the module. A yellow MAINT LED indicates the stand-by state.

3 Device configuration

3.1 Introduction

Device configuration entails planning the hardware design of the 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 1500 standard controller and provides an overview of the networking options. Configuration of the distributed I/O is described in Chapters 16.3 “PROFINET IO” on page 701 and 16.4 “PROFIBUS DP” on page 716.

Starting

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*.

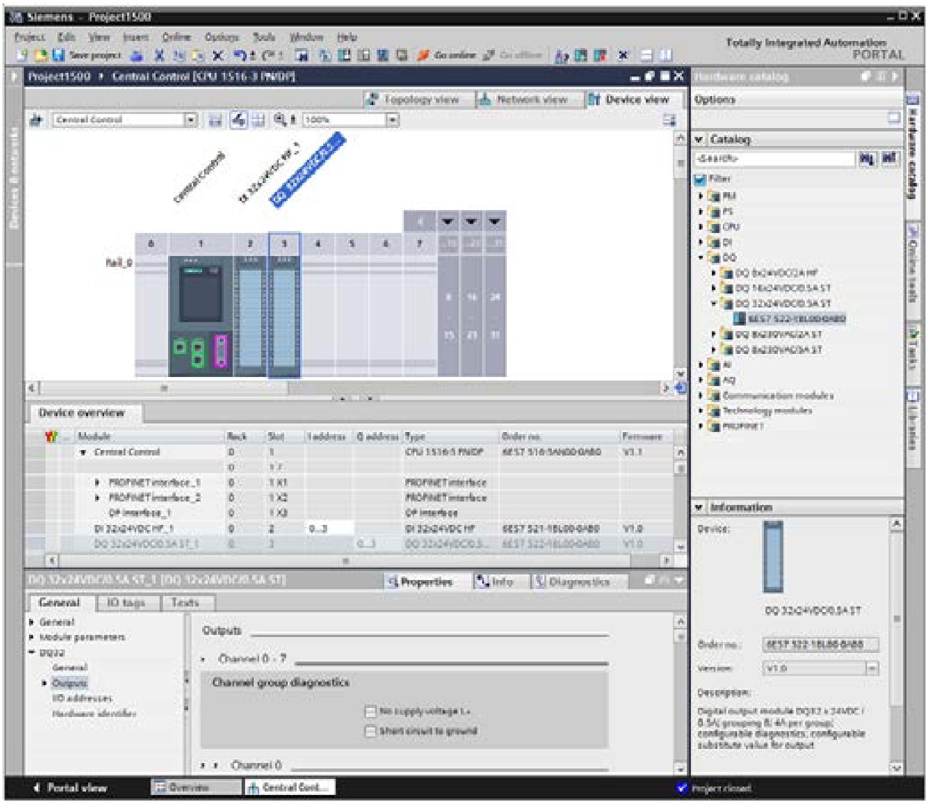
In the Project view, you can start the device configuration in the project tree either by double-clicking on the *Devices & networks* editor under the project or on the *Device configuration* editor under the PLC station.

Working area of the device configuration

Fig. 3.1 shows the working area of the device configuration in the Project view (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 as a table with the existing stations and their interconnections in the bottom part. Further information can be found in Chapter 3.4 “Configuring a network” on page 73.
- ▷ 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



Device overview

Module	Rack	Slot	I address	Q address	Type	Order no.	Firmware
▾ Central Control	0	1			CPU 1516-3 PN/DP	6ES7 516-3AN00-0AB0	V1.1
	0	1 Y					
▸ PROFINET interface_1	0	1 X1			PROFINET interface		
▸ PROFINET interface_2	0	1 X2			PROFINET interface		
DP interface_1	0	1 X3			DP interface		
DI 32x24VDC HF_1	0	2	0...3		DI 32x24VDC HF	6ES7 521-1BL00-0AB0	V1.0
DQ 32x24VDC/0.5A ST_1	0	3		0...3	DQ 32x24VDC/0.5...	6ES7 522-1BL00-0AB0	V1.0

DQ 32x24VDC/0.5A ST_1 [DQ 32x24VDC/0.5A ST]

Properties | Info | Diagnostics

General | IO tags | Texts

- General
- Module parameters
- DQ32
  - General
  - Outputs
  - I/O addresses
  - Hardware identifier

Outputs

Channel 0 - 7

Channel group diagnostics

□ No supply voltage L+

□ Short circuit to ground

Channel 0

Description:

Digital output module DQ32 x 24VDC / 0.5A; grouping 8; 4A per group; configurable diagnostics; configurable substitute value for output

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

the bottom part. Further details on the Topology view are described in Chapter 16.3.5 “Real-time communication in PROFINET” on page 710.

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 alarm 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 “fold shut” all windows except the working window and thus provide more space for the latter. The working window can also be maximized and displayed as a separate window.

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, 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 using *Online > Download to device*.

Upgrading and support

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

To subsequently install support packages, for example hardware support packages (HSP) for new modules, select *Options > Support packages* in 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 from the file system.

3.2 Configuring a station

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

3.2.1 Adding a PLC station

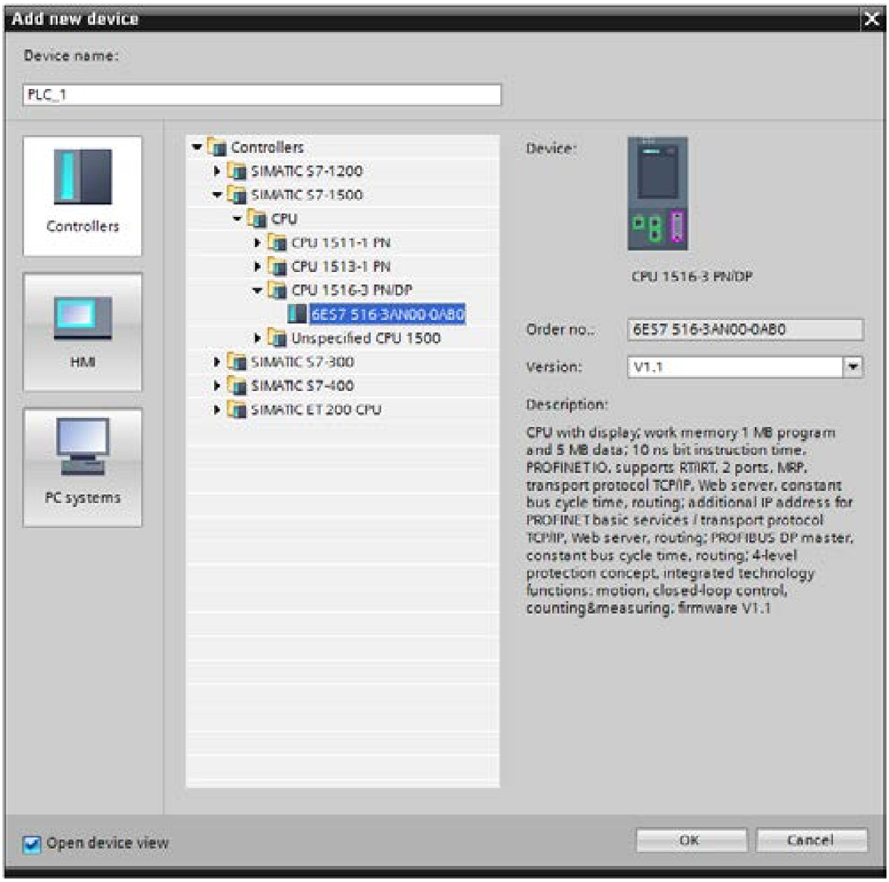
When creating a new project, you normally add a PLC station at the same time. You can add further PLC stations in both the Portal view and the Project view. In the Portal view, you can add a new station in the *Devices & networks* portal using the *Add new device* command. In the Project view, double-click 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* checkbox is activated in the window at the bottom left (Fig. 3.2).

You have now configured a rack with a CPU inserted in slot 1. Slot 0 on the far left is intended for a power supply module.

3.2.2 Adding a module

If you have not already done so, open the PLC station in the Device view. To insert a 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



Device name: PLC_1

Order no.: 6ES7 516-3AN00-0AB0

Version: V1.1

Description:
CPU with display; work memory 1 MB program and 5 MB data; 10 ns bit instruction time, PROFINET IO, supports RT/IRT, 2 ports, MRP, transport protocol TCP/IP, Web server, constant bus cycle time, routing; additional IP address for PROFINET basic services / transport protocol TCP/IP, Web server, routing; PROFIBUS DP master, constant bus cycle time, routing; 4-level protection concept, integrated technology functions: motion, closed-loop control, counting&measuring; firmware V1.1

☑ Open device view

Fig. 3.2 Selection window *Add new device*

are highlighted. You position the new module by double-clicking on the module symbol or by dragging it with the mouse to the rack.

If you activate the *Filter* checkbox in the hardware catalog, only modules from the selected device family will be shown; in our case only the modules for SIMATIC S7-1500.

The I/O modules can be arranged as desired during the configuration on slots 2 to 31, even with gaps. For the compilation, however, the modules must be inserted without gaps.

The modules are supplied with power from the backplane bus of the CPU or a system power supply (see Section "Design variants" on page 48). Observe the power consumption in a power segment when arranging the modules. Any imbalance in the power will be reported as an error during the compilation.

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

3.3 Parameterization of modules

“Parameterization” or “assigning parameters” 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 operation 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.

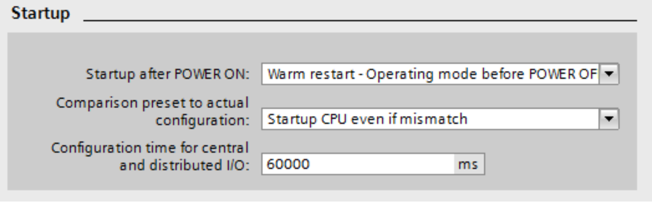
To parameterize the CPU properties, select the CPU in the working window of the device configuration. If the project contains several stations, select the desired station in the toolbar of the working window.

Set the name of the PLC station in the **General** section under *Project information* and the module ID under *Identification & Maintenance*. Using the higher level designation, you can identify the CPU according to its function in the plant, for example, and you can use the location designation – which can be part of the equipment designation – to describe the arrangement of the PLC station on the machine or within the plant.

In the **PROFINET interface** section you set the connection to an Ethernet subnet and define the IP address, the subnet mask, and the PROFINET device name. For more information on the format of the IP address, refer to Chapter 3.4.6 “Configuring a PROFINET subnet” on page 80. Under *Operating mode* you can activate the operation as an IO device. The IO controller mode is a fixed default setting. Further settings define, for example, the real-time properties of the PROFINET IO communication, the interconnection of the ports, and the activation of the web server via this interface.

In the **DP interface** section (for CPU 1516) you can define the connection to a PROFIBUS subnet, the node address, and other properties such as the properties of the SYNC/FREEZE groups. The DP master mode is a fixed default setting. You change the network parameters in the Properties tab of the inspector window in the Network view with the PROFIBUS subnet selected. For more information, refer to Chapter 3.4.7 “Configuring a PROFIBUS subnet” on page 84.

You can set the startup characteristics of the CPU under **Startup** (Fig. 3.3). As *Startup after POWER ON* you can select between *No restart (remain in STOP mode)*, *Warm restart - RUN*, and *Warm restart - Operating mode before POWER OFF*.



Startup

Startup after POWER ON: Warm restart - Operating mode before POWER OF

Comparison preset to actual configuration: Startup CPU even if mismatch

Configuration time for central and distributed I/O: 60000 ms

Fig. 3.3 Startup parameters with a CPU 1500

During startup, the CPU compares the modules that are actually inserted to the configuration. You can set the strictness of the check under *Comparison preset to actual configuration*: *Start up CPU only if compatible* or *Startup CPU even if mismatch*.

The duration for distributed I/O parameterization is monitored during a startup; you can set the parameterization time. A module is considered to be absent if the monitoring time for it expires.

In the **Cycle** section, you define the cycle monitoring time under *Maximum cycle time*. It is signaled if the cycle monitoring time is exceeded and this can lead to the STOP operating state. You can also specify the *Minimum cycle time*, which indicates the minimum duration of program cycle execution.

In the **Communication load** section, you set the time share for communication under *Cycle load due to 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 with this parameter (communication load). The cycle time is then extended by the factor 100 / (100 – communication load).

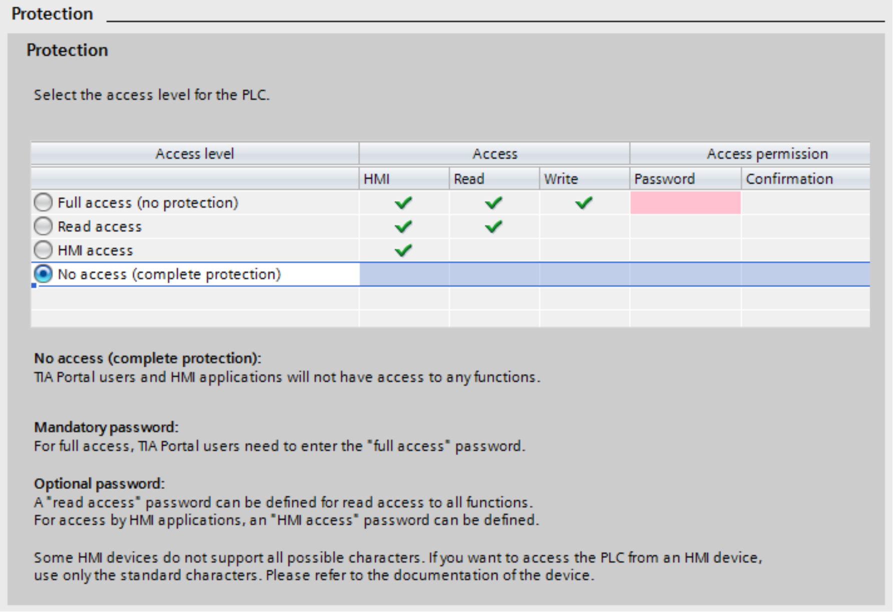
System and clock memory are operands controlled by the operating system which can be scanned in the user program. For example, there is a bit memory which indicates the occurrence of a diagnostic event, or a bit memory which changes its signal state at a frequency of 2 Hz. During parameterization of the CPU, you activate the system memories and/or the clock memories and assign an address 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 90.

In the **System diagnostics** section, you activate the system diagnostics and define the category of the alarms to be output and whether the alarms must be acknowledged.

In the **Web server** section, you can activate the web server and set its properties. Further details can be found in Chapter 18.3 “Web server” on page 796. The properties of the display in the front flap of the CPU are set in the **Display** section. Such properties are, for example, waiting times for standby and energy-saving mode and password-protected display. The language in which the project texts are displayed in the web server and on the display can be defined in the **User interface languages** section.

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

In the **Protection** section, you can protect the program in the CPU from unauthorized access. Here, you can assign a password for each type of access (read access, full access, HMI access) or completely block access to the CPU (Fig. 3.4). Further details can be found in Chapter 15.2.3 "Protecting the user program" on page 657. In this properties group, you can also set the access permission for S7 communication with GET and PUT.



Protection

Protection

Select the access level for the PLC.

Access level	Access			Access permission	
	HMI	Read	Write	Password	Confirmation
○ Full access (no protection)	✓	✓	✓		
○ Read access	✓	✓			
○ HMI access	✓				
● No access (complete protection)					

No access (complete protection):
TIA Portal users and HMI applications will not have access to any functions.

Mandatory password:
For full access, TIA Portal users need to enter the "full access" password.

Optional password:
A "read access" password can be defined for read access to all functions.
For access by HMI applications, an "HMI access" password can be defined.

Some HMI devices do not support all possible characters. If you want to access the PLC from an HMI device, use only the standard characters. Please refer to the documentation of the device.

Fig. 3.4 Settings for access protection

The **System power supply** section shows the power balance of the first power segment in the rack. If to the left of the CPU there is a system power supply module which supplies the CPU, place a checkmark in the box *Connection to supply voltage L+*.

The **Connection resources** section shows the number and distribution of the reserved and available resources for the configured connections of the station and the communication-capable modules in the station. Further details are described in Chapter 3.4.5 “Configuring a connection” on page 78.

The assigned inputs and outputs are shown in the **Overview of addresses**. The module addresses, the assigned process image partitions with any assigned organization blocks, and the slots are displayed. The display encompasses the modules that are both centrally and decentrally configured via PROFINET and PROFIBUS.

3.3.2 Addressing modules

Slot address, geographic address

Every slot in a PLC station has a fixed address. A module is unequivocally defined by the slot address (“geographic address”). The CPU in an S7-1500 station has, for example, the slot address of Rack 0, Slot 1.

If interface submodules are present on the module, each submodule is assigned an additional module address. In this manner, every binary signal, every analog signal, and every serial connection in the system can be addressed unequivocally.

In the same manner, modules of the distributed I/O also have a “geographic” address. In this case, the number of the PROFINET IO system or DP master system and the station number replace the rack number.

By positioning a module on a rack in the hardware configuration, you automatically define the slot address. The CPU's operating system requires the slot address in order to explicitly address a specific module, e.g. during parameterization. The slot address is not usually required in the user program, and is not used either.

Hardware identifier

The configuration editor assigns a hardware identifier for each object, including modules. This is a constant value which cannot be changed by the user. With the hardware identifier, you can address a module in the user program, for example when querying the module status (see Chapter 4.4 “Addressing of hardware objects” on page 107). All of the hardware identifiers used in the PLC station are listed in the default tag table in the *System constants* tab (Chapter 4.12 “Hardware data types” on page 143).

Logical address, user data address

Every peripheral byte is addressed by a number, the “logical” address. This logical address defines the slot, and this corresponds to the absolute address. This is also referred to as the user data address since you can use this address to access the user data of the input/output modules in the user program, either via the process image (inputs I and outputs Q) or directly on the modules (peripheral inputs I:P and peripheral outputs Q:P). The range of the logical addresses starts with zero and ends with 32 767, for inputs and outputs respectively.

Module start address

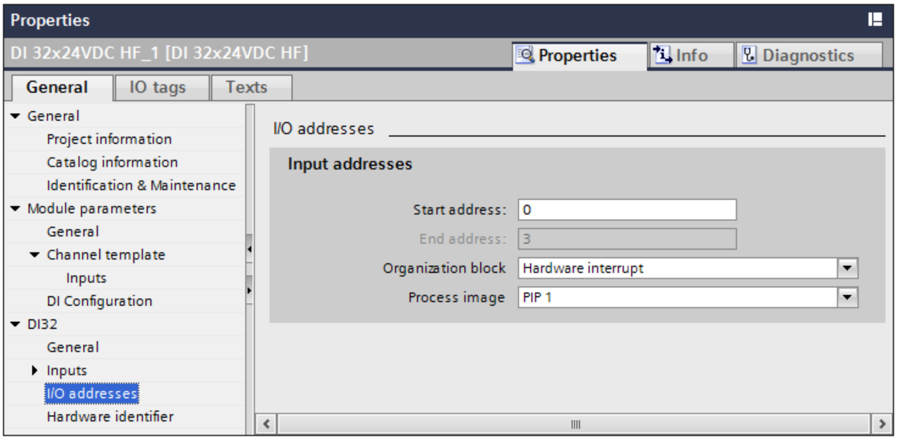
The module start address is the smallest logical (user data) address of a module; it identifies the relative byte zero of the module. The following module bytes are then occupied consecutively with the logical addresses.

Using the hardware configuration you determine the position of the user data addresses of a module in the address volume of the CPU by specifying the module start address. The lowest logical address is the module start address, also for modules of the distributed I/O and even for the virtual slots in the user data interface of an intelligent IO device or an intelligent DP slave.

The module start address is used in some cases to identify a module. It has no special significance beyond that.

Configuring user data addresses

When configuring the modules, STEP 7 automatically assigns a module start address starting with zero. 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 under *I/O addresses*. You can change the automatically assigned start address (Fig. 3.5).



I/O addresses

Input addresses

Start address: 0

End address: 3

Organization block: Hardware interrupt

Process image: PIP 1

Fig. 3.5 Example of parameterization of I/O addresses of a digital input module

The logical addresses of the individual modules – independent of whether they are centrally located or belong to the distributed I/O – must not overlap. For the input and output modules, the logical addresses are assigned separately so that an input byte can have the same number as an output byte.

All of the inputs and outputs of a module have a process image in the system memory of the CPU. During addressing, you set the way in which the process image of the module is to be updated in the *I/O addresses* properties group. The entry

Automatic update means that the process image of this module will be automatically updated before calling organization block OB 1. You can set a process image partition and link it to a specific organization block. This process image partition will then be updated when this organization block is called. If you do not assign an organization block, the process image update can be initiated in the user program using system blocks. If you do not assign a process image (entry *None*), you must address the user data of the module directly via the I/O operand area. Further details are described in Chapter 5.6.2 “Process image updating” on page 179.

3.3.3 Assigning parameters to signal modules

To parameterize the module properties, select the module in the working window of the Device view and set the properties in the inspector window on the *General* tab.

Common properties

In the *General* section of the module properties, you can enter a name under *Project information* of the module and enter the higher level designation and location designation under *Identification & Maintenance*.

The configuration of modules with many I/O channels is simplified by the *Channel template*. By default, all of the channels have the properties that are set in the channel template. Deviations from the template can be individually configured for the corresponding channel.

In the section *Module parameters > General*, you configure the behavior during a startup in the *Comparison preset to actual module* drop-down list if the actual module differs from the configured module: *From CPU, Startup CPU only if compatible*, or *Startup CPU even if mismatch*.

For correspondingly configured modules, you can configure a *value status*, which indicates the validity of an assigned I/O signal with one bit each. For additional details, see Section “Value status” on page 90. You activate the value status under *Module parameters > xx Configuration* (xx = depending on the module type DI, DQ, AI, or AQ).

The start address of the module, the assignment to a process image and to an organization block can be set under *xx > I/O addresses* (xx = depending on the type of module DI*n*, DQ*n*, AI*n*, or AQ*n*; *n* = number of I/O channels). An example is shown in Fig. 3.5 on page 69.

Hardware identifier shows the hardware identifier of the module, which is assigned and listed in the *System constants* tab of the default tag table by the configuration editor.

Digital input modules

With correspondingly configured modules, you can parameterize additional properties in the section *DIn > Inputs > Channel*. To adjust the settings, select *Manual* under *Parameter settings*.

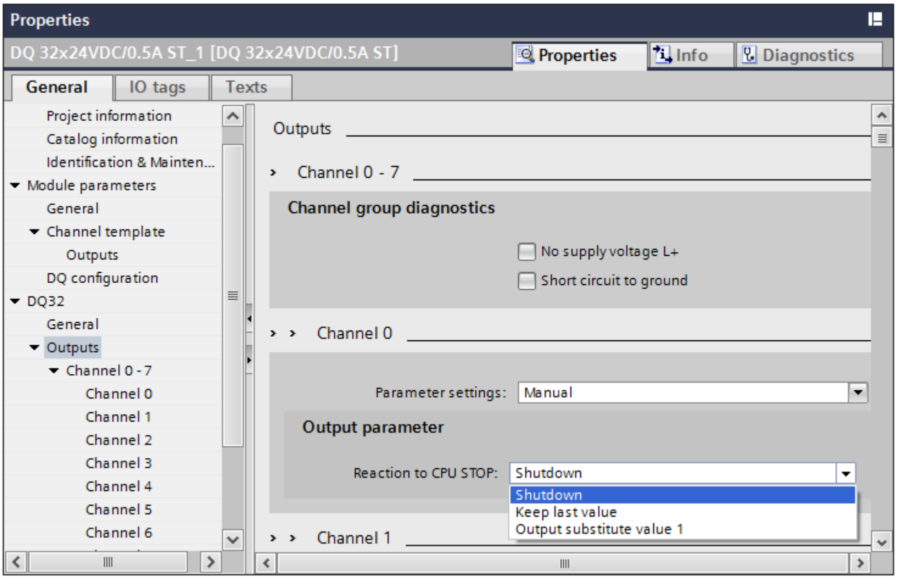
In the *Diagnostics* field, activate the events (No supply voltage L+, Wire break) which cause a diagnostic alarm to be sent when they occur, e.g. to the diagnostics buffer.

The input delay can be set in the *Input parameters* field. The longer the input delay, the more immune the input signal is to high-frequency interfering signals. However, this also increases the period until a change in the input signal is recognized by the module.

The *Hardware interrupts* field contains the assignment of a hardware interrupt to a status change of the input signal. You can assign a hardware interrupt organization block to each signal edge and specify its processing priority.

Digital output modules

With correspondingly configured modules, you can parameterize additional properties in the section *DQn > Outputs > Channel* (Fig. 3.6):



□ No supply voltage L+
□ Short circuit to ground

Parameter settings: Manual

Reaction to CPU STOP: Shutdown

Fig. 3.6 Example of parameterization of a digital output channel

In the *Channel group diagnostics* field or in the *Diagnostics* field, activate the events (No supply voltage L+, Short circuit to ground) which cause a diagnostic alarm to be sent when they occur, e.g. to the diagnostics buffer.

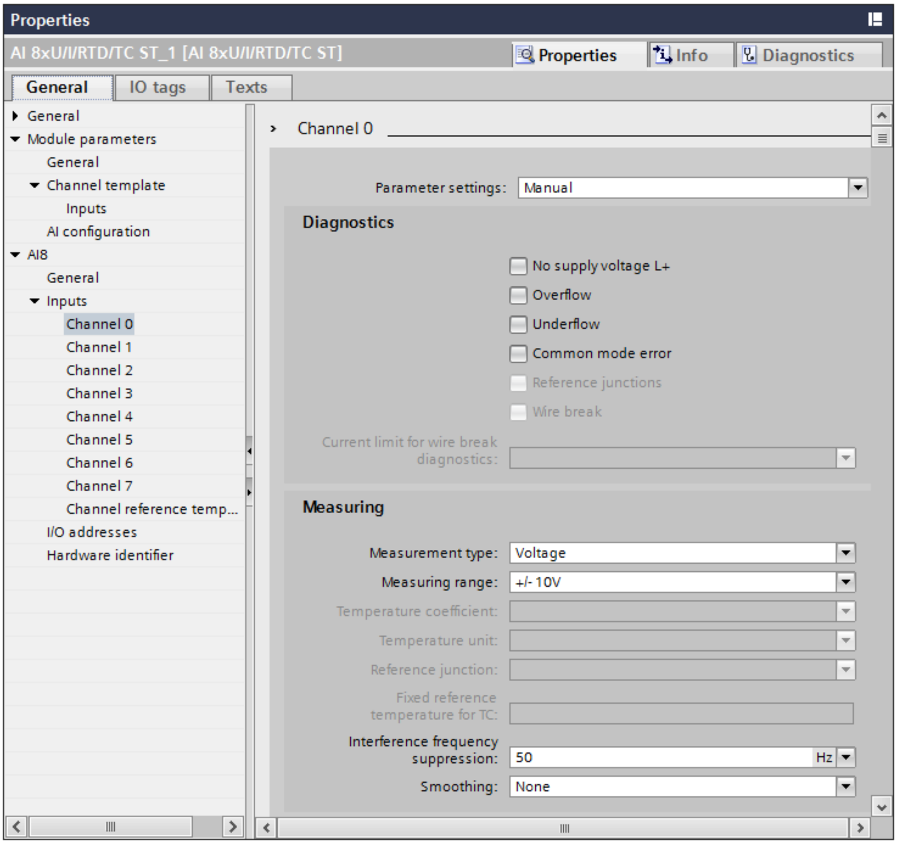
In the *Output parameter* field there is the entry from the channel template if the *Parameter settings* are set to *From template*. If you set the parameter settings to *Manual*, you can select the response of the output channel during the transition to the STOP operating state from a drop-down list: *Shutdown*, *Keep last value*, or *Output substitute value 1*.

Analog input modules

With correspondingly configured modules, you can parameterize additional properties in the section *AIn* > *Inputs* > *Channel*. To adjust the settings, select *Manual* under *Parameter settings*.

In the *Diagnostics* field, activate the events (No supply voltage L+, Overflow, Underflow, Common mode error, Reference junctions, Wire break) which cause a diagnostic alarm to be sent when they occur, e.g. to the diagnostics buffer.

In the *Measuring* field, set the type of measurement (e.g. Voltage, Current, Resistor, Thermocouple) and the associated parameters (Fig. 3.7).



Channel 0

Parameter settings: Manual

Diagnostics

□ No supply voltage L+
□ Overflow
□ Underflow
□ Common mode error
□ Reference junctions
□ Wire break

Current limit for wire break diagnostics:

Measuring

Measurement type: Voltage
Measuring range: +/- 10V
Temperature coefficient:
Temperature unit:
Reference junction:
Fixed reference temperature for TC:
Interference frequency suppression: 50 Hz
Smoothing: None

Fig. 3.7 Example of parameterization of an analog input channel (diagnostics, measurement type)

The *Hardware interrupts* field contains the assignment of a hardware interrupt to a limit violation of the input signal (the exceeding or undershooting of two configurable limits). You can assign a hardware interrupt organization block to each limit violation and specify its processing priority.

Analog output modules

With correspondingly configured modules, you can parameterize additional properties in the section *AQn* > *Outputs* > *Channel*. To adjust the settings, select *Manual* under *Parameter settings*.

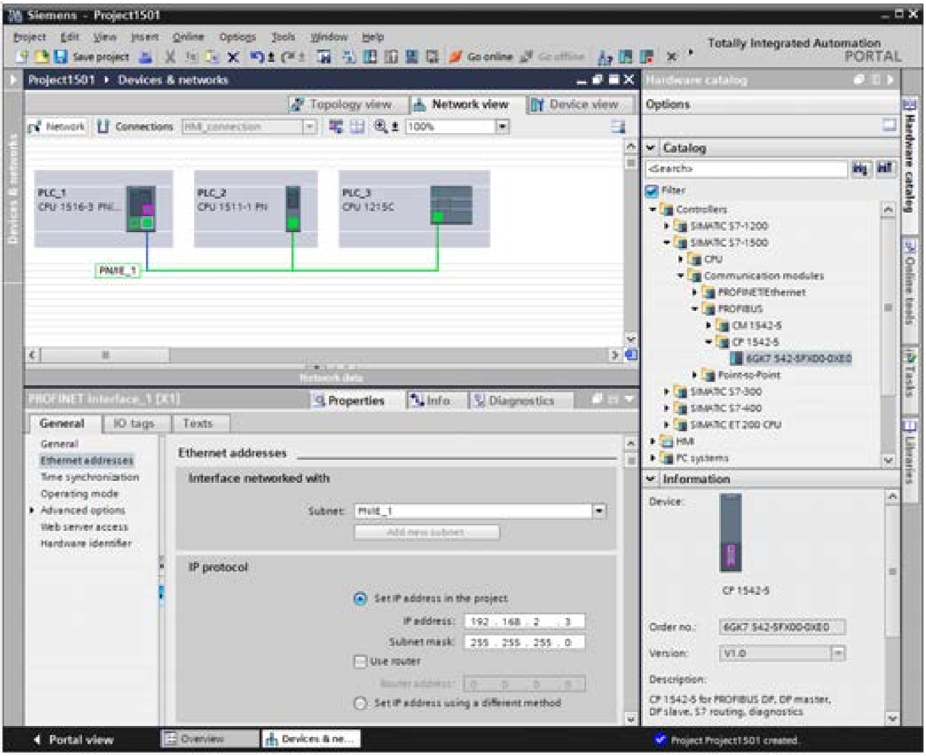
In the *Diagnostics* field, activate the events (No supply voltage L+, Wire break, Short circuit to ground, Overflow, Underflow) which cause a diagnostic alarm to be sent when they occur, e.g. to the diagnostics buffer.

In the *Output* field, set the type of measurement (Deactivated, Voltage, Current) and the associated parameters and the response of the output channel when transitioning to the STOP operating state (*Shutdown*, *Keep last value*, or *Output substitute value*).

3.4 Configuring a network

3.4.1 Introduction, overview

The network configuration permits the graphic display (on screen) and graphic documentation (on paper) of the configured networks and their stations. Configu-



Ethernet addresses

Interface networked with

Subnet: PN/IE_1

IP protocol

● Set IP address in the project

IP address: 192 . 168 . 2 . 3

Subnet mask: 255 . 255 . 255 . 0

□ Use router

Router address: 0 . 0 . 0 . 0

○ Set IP address using a different method

Device:

Order no.: 6GK7 542-5FX00-0XE0

Version: V1.0

Description:
CP 1542-5 for PROFIBUS DP, DP master, DP slave, S7 routing, diagnostics

Project Project1501 created.

Fig. 3.8 Example of working area of network configuration (network view)

ration of the networking is part of the device configuration. If a PLC station is operated on its own, without HMI station and without data communication to other PLC stations, the network configuration is not required. Connection of a programming device for transfer of the user program and for program testing does not require configuration either.

You can access 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. 3.8).

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 lower part of the working window (closed and not visible in the figure) contains the tabs *Network overview*, *Connections*, *I/O communication*, and *VPN*. You can drag further stations with the mouse from the hardware catalog into the working area and thus add them to the project. Information on the selected object is displayed below the hardware catalog. If you select an object in the working window, the inspector window displays the properties of the object.

3.4.2 Networking a station

“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 via the cable. The logical connection defines the transmission parameters between the modules.

The working window of the configuration editor shows the existing stations with the modules with communication capability. The interfaces for the subnets are highlighted.

Adding a station in the network configuration

In the hardware catalog under *Controllers > SIMATIC S7-1500 > CPU > [folder: CPU 15xx...] > [CPU]*, select the desired CPU and drag it with the mouse into the working area. The graphic shows the CPU with the existing bus interfaces as a representative for the complete PLC station.

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 *Controllers > SIMATIC S7-1500 > 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.

A CM module added in this manner is positioned by the configuration editor in the lowest vacant slot in the rack.

If you drag the CM module to an existing subnet and if the CM module has an interface matching the subnet, the interface is directly connected to the subnet when adding and the CM module is displayed individually as a graphic. In the Device view, the CM module is then positioned in a rack which is otherwise empty.

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

Click on the *Network* button in the toolbar of the working window in order to network stations.

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 1500 has two ports which are interconnected by an integrated switch. A linear network can thus quasi be set up.

The individual ports are shown in the topology view and you can then interconnect them and set their properties.

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 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. 3.9).

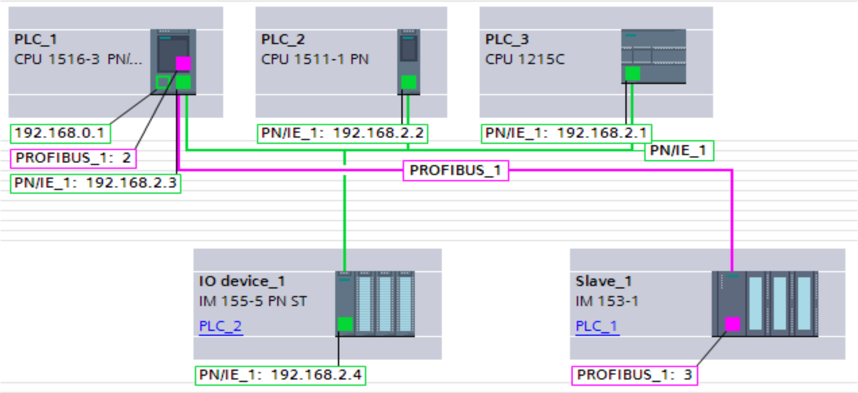


Fig. 3.9 Display of node addresses in the network view

Setting node addresses

When networking a module, the configuration editor automatically claims 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 Communication services and types of connection

The connection type specifies the protocol for the data exchange. You select the connection type during the configured setup of the connection, depending on the communication service to be carried out. The connection type is determined by the communication functions during the programmed setup (Table 3.1).

PG communication is used for connecting a programming device to a PLC station. The necessary connection is automatically set up during the setup of the online mode. Data can be transferred via a PROFINET or PROFIBUS subnet and via a subnet gateway.

Table 3.1 Communication services and types of connection (selection)

Communication service	Connection type	Connection setup		Subnet	
PG communication	–		automatic	PN	DP
HMI communication	HMI connection	configured	automatic	PN	DP
Open user communication					
via TCP/IP	TCP connection	configured	programmed	PN	
via ISO-on-TCP	ISO-on-TCP connection	configured	programmed	PN	
via UDP	UDP connection	configured	programmed	PN	
via ISO [1]	ISO connection	configured		PN	
via FDL [2]	FDL connection	configured			DP
E-mail	–		programmed	PN	
FTP [1]	–		programmed	PN	
S7 communication	S7 connection	configured		PN	DP
Point-to-point communication	–		programmed	PtP	

[1] via CP 1543-1
[2] via CM 1542-5

HMI communication is used for connecting an HMI device to a PLC station. When the online mode with the PLC station is initiated by the HMI device, the necessary connection is automatically set up. A configured HMI connection is necessary in order to be able to configure the data exchange between the PLC station and the HMI station.

Open user communication is used for data exchange between PLC stations or with external devices. It takes place via PROFINET (exception: open user communication via FDL). A TCP, ISO-on-TCP and a UDP connection can be programmed with both the configuration editor and with the communication functions. The communication functions are needed in both cases. Configured connections are set up statically and are permanently assigned to the connection resources. Programmed connections can be set up dynamically and the connection resources can be released after the data transfer. Programming open user communication via TCP, ISO-on-TCP and UDP is described in Chapter 17.2 “Open user communication” on page 751.

Open user communication via ISO is used for transferring data via Industrial Ethernet. The CP 1543-1 communication module can exchange data with devices that support the ISO transport connection. The communication service is suitable for large volumes of data which are acknowledged upon receipt. The interfaces in the user program of an S7 station are SEND/RECEIVE and FETCH/WRITE. For PC stations, there are C-functions for ISO transport services.

Open user communication via FDL (Fieldbus Data Link) is used for transferring data via PROFIBUS FDL. The CM 1542-5 communication module can exchange data with devices that support the sending or receiving of data according to the SDA function (Send Data with Acknowledge). The receipt of data is confirmed by an acknowledgement. The interface in the user program of an S7 station is SEND/RECEIVE. For PC stations, there are C functions for the FDL services.

Using e-mail communication, process data can be sent as an e-mail via Industrial Ethernet. The necessary connection is programmed with the communication function TMAIL_C (see Chapter 17.2.6 "Further functions of open user communication" in Section "Send e-mail" on page 758).

S7 communication is used for the exchange of data between PLC stations via PROFINET or PROFIBUS. An S7 connection is configured using the configuration editor. The communication functions for a unilaterally configured connection are PUT and GET. For a bilaterally configured connection, they are BSEND/BRCV and USEND/URCV. Chapter 17.3 "S7 communication" on page 761 describes how you can program S7 communication.

Point-to-point communication transfers data via a serial point-to-point connection. The CM PtP communication modules handle the data traffic via an RS 232 or RS 422/485 port. The PtP connection is programmed using communication functions (see Chapter 17.4 "Point-to-point communication" on page 767).

3.4.5 Configuring a connection

In order to configure a connection, 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 highlighted in the Network view.

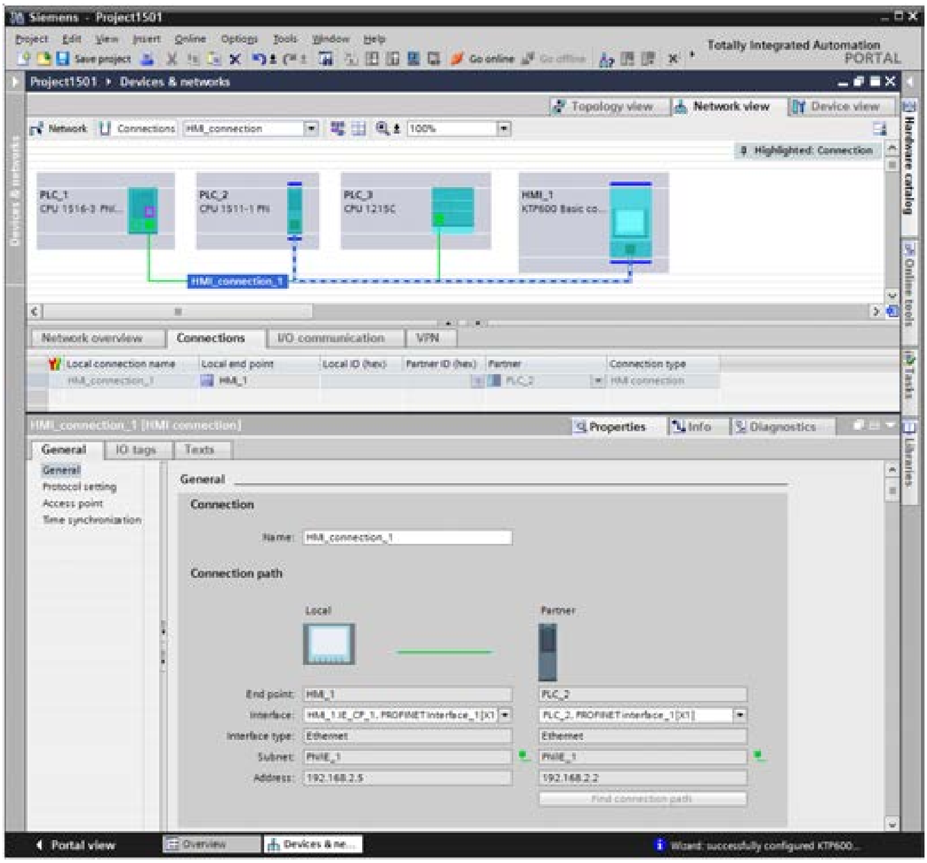
Click with the left mouse button on a station, drag the connection line with the mouse button pressed to the other station, and release the button. A connection with the connection name is displayed as a blue/white patterned line. If no networking has been configured, a suitable subnet will be automatically created. Several logical connections can be created using one cable. These connections are then also present in the connection table in the *Connections* tab in the bottom part of the working window (Fig. 3.10).

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

A connection is defined unequivocally by means of the "connection ID". In the communication functions program, this connection ID specifies the connection via which the data is to be transmitted. The connection ID can have different values in the two connection partners.

The connection partners, the connection path, and the node addresses are displayed in the inspector window in the *Properties* tab under *General*. Fig. 3.10 shows the connection properties for an HMI connection. If a station has several suitable interfaces, you can select the appropriate one from a drop-down list. In the lower part of the property sheet, you can set additional connection properties depending on the type of connection.



Network overview | Connections | I/O communication | VPN

Local connection name	Local end point	Local ID (hex)	Partner ID (hex)	Partner	Connection type
HMI_connection_1	HMI_1			PLC_2	HMI connection

HMI_connection_1 [HMI connection]

General

Connection

Name: HMI_connection_1

Connection path

	Local	Partner
End point:	HMI_1	PLC_2
Interface:	HMI_1.IE_CP_1, PROFINET interface_1[X1	PLC_2, PROFINET interface_1[X1]
Interface type:	Ethernet	Ethernet
Subnet:	PN/IE_1	PN/IE_1
Address:	192.168.2.5	192.168.2.2

Wizard: successfully configured KTP600...

Fig. 3.10 Representation of an HMI connection in the network configuration

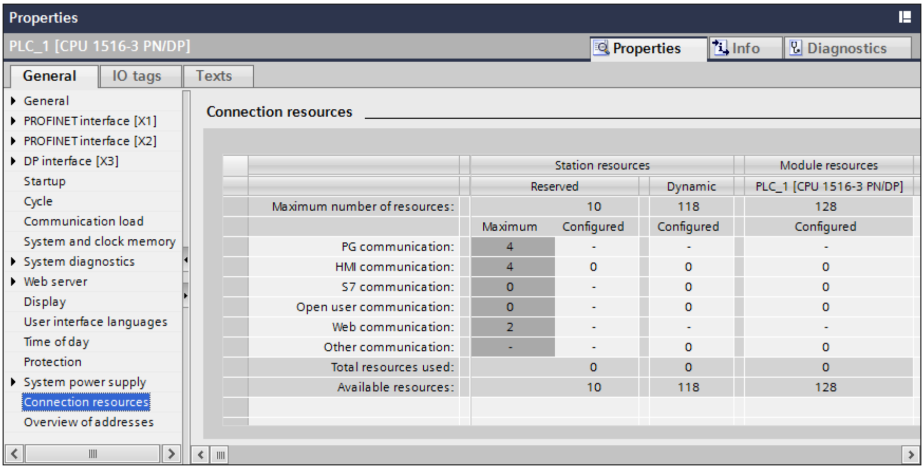
Connection resources

Every connection requires connection resources (memory areas in the operating system of the module) for the end point of the connection and for the transition point in a CM/CP module. For example, one connection is occupied in the CPU if S7 functions are executed over a bus interface of the CPU; the same functions over the bus interface of the CP module occupies one connection resource each in the CP module and in the CPU.

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. Connections are reserved for PG communication, HMI communication, and communication with the web server. These cannot be used for any other purpose.

The available connection resources depend on the CPU and the communication modules used and must not exceed a defined upper limit for the PLC station.

The connection resources of an S7-1500 station are displayed in the properties of the CPU. The display also contains the connection resources of the existing communication modules (Fig. 3.11).



Connection resources

	Station resources			Module resources
	Reserved		Dynamic	PLC_1 [CPU 1516-3 PN/DP]
Maximum number of resources:		10	118	128
	Maximum	Configured	Configured	Configured
PG communication:	4	-	-	-
HMI communication:	4	0	0	0
S7 communication:	0	-	0	0
Open user communication:	0	-	0	0
Web communication:	2	-	-	-
Other communication:	-	-	0	0
Total resources used:		0	0	0
Available resources:		10	118	128

Fig. 3.11 Connection resources of an S7-1500 station

3.4.6 Configuring a PROFINET subnet

The X1 interface of a CPU 1500 is a PROFINET interface, which can function in the IO controller and IO device modes in addition to transferring data via Industrial Ethernet. The interface has two ports which are interconnected by a switch. The CPU 1516 has a second PN interface X2 only for transferring data via Industrial Ethernet. Therefore, this interface lacks the setting options for PROFINET IO. This interface has only one port. In addition, CP 1543-1 communication modules can be operated in an S7-1500 station for the transfer of data to Industrial Ethernet.

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

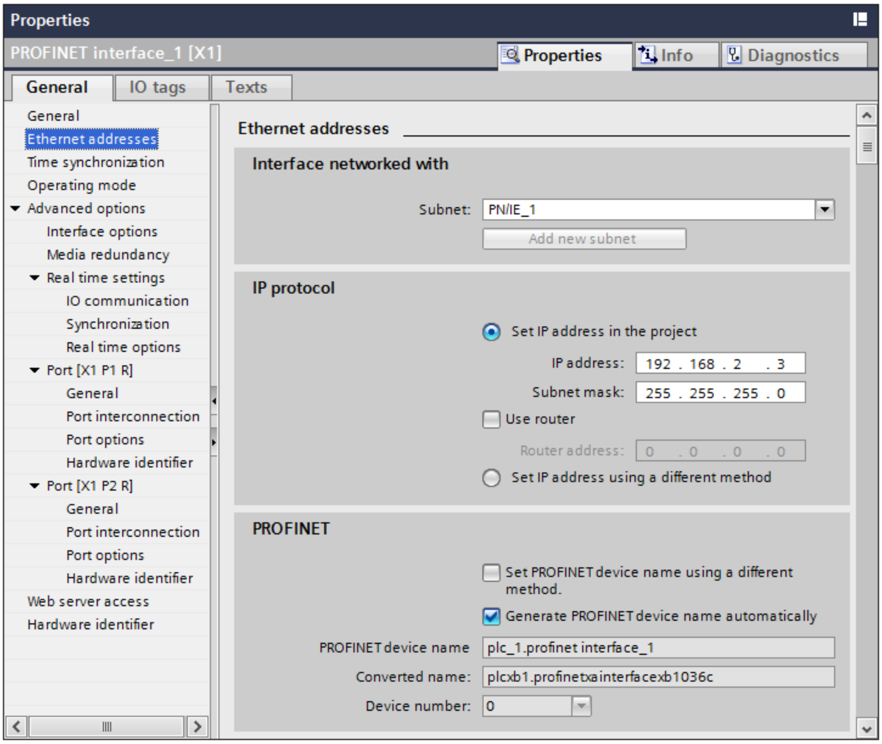
To set the properties, select the PROFINET subnet and then the *Properties* tab in the inspector window. Under *General*, you can assign a different name to the subnet and also change the subnet ID if appropriate.

In the *Domain management* section, you compile the node groups for real-time communication (*Sync domains*) and media redundancy (*MRP domains*). You can find more details in Chapters 16.3.5 “Real-time communication in PROFINET” on page 710 and “Media redundancy” on page 716. Under *Overview isochronous mode* you can view an overview of all of the components involved in isochronous mode and the relevant parameters. You can find the description for this in Chapter 16.7.2

"Isochronous mode with PROFINET IO" on page 739.

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 (Fig. 3.12).



Ethernet addresses

Interface networked with

Subnet: PN/IE_1

IP protocol

● Set IP address in the project

IP address: 192 . 168 . 2 . 3

Subnet mask: 255 . 255 . 255 . 0

□ Use router

Router address: 0 . 0 . 0 . 0

○ Set IP address using a different method

PROFINET

□ Set PROFINET device name using a different method.

☑ Generate PROFINET device name automatically

PROFINET device name plc_1.profinet interface_1

Converted name: plcxb1.profinetxainterfacexb1036c

Device number: 0

Fig. 3.12 Example of the properties window of a PN interface for PROFINET IO

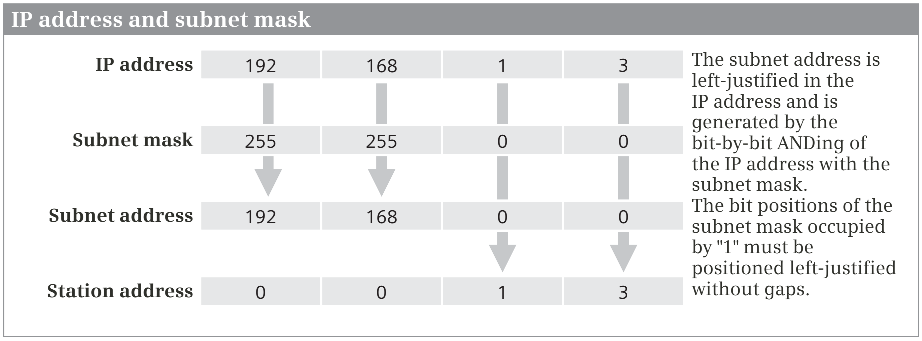
Ethernet address (MAC address)

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), where 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. 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.

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. 3.13).



IP address and subnet mask

IP address	192	168	1	3
Subnet mask	255	255	0	0
Subnet address	192	168	0	0
Station address	0	0	1	3

The subnet address is left-justified in the IP address and is generated by the bit-by-bit ANDing of the IP address with the subnet mask.
The bit positions of the subnet mask occupied by "1" must be positioned left-justified without gaps.

Fig. 3.13 Example of the structure of an IP address

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 extension for PROFINET device name* checkbox in the properties of the PROFINET IO system.

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.

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 an actual 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 and 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.

For operating on PROFINET IO, you can set the operating mode to IO Device, the assigned IO controller, and the structure of the transfer areas in addition to the permanently set IO controller mode in the *Operating mode* section. Under *Advanced options* you can set, among others, the options for real-time mode. Refer to Chapter 16.3.3 “Configuring PROFINET IO” on page 705 for how to configure a PROFINET IO system.

Release this PN interface for access under *Web server access*. You can configure the activation of the web server in the properties of the CPU.

Hardware identifier shows the hardware identifier of the interface, which is assigned and listed in the *System constants* tab of the default tag table by the configuration editor.

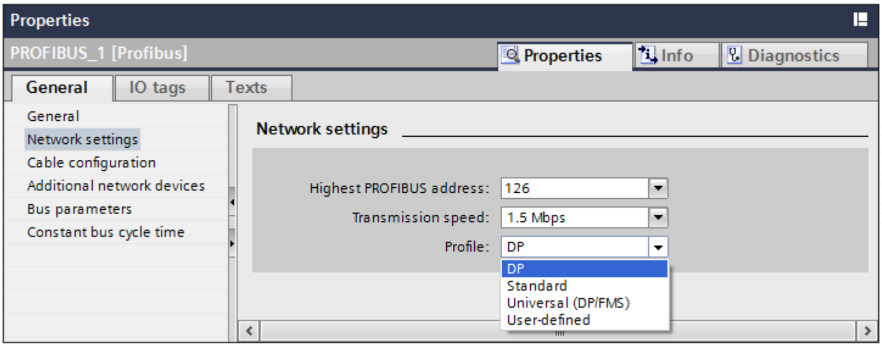
3.4.7 Configuring a PROFIBUS subnet

The third bus interface X3 of a CPU 1516 is a DP interface for operating as a DP master (firmware version 1.0). In each S7-1500 station, CM 1542-5 communication modules can be operated. They can be either a DP master or DP slave.

To configure a PROFIBUS subnet, drag the DP interface of one station to the DP interface of the other station with the mouse. 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. Observe the technical specifications of the involved modules when doing this (Fig. 3.14).



Highest PROFIBUS address: 126

Transmission speed: 1.5 Mbps

Profile: DP

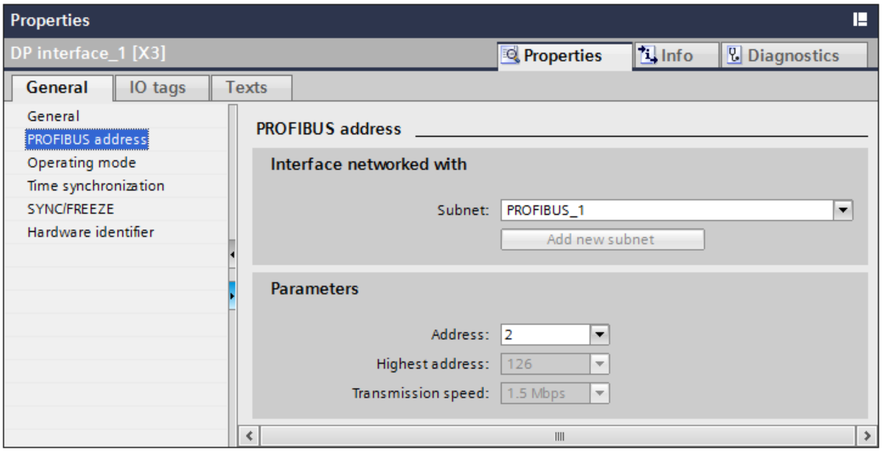
Fig. 3.14 Example of network settings on the PROFIBUS

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 (Fig. 3.15).



PROFIBUS address

Interface networked with

Subnet: PROFIBUS_1

Parameters

Address: 2

Highest address: 126

Transmission speed: 1.5 Mbps

Fig. 3.15 Example of the properties window of a DP interface

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.

The configuration editor assigns node addresses from 2 upwards as standard. It is recommendable to assign the addresses without gaps.

Under *Operating mode* you set whether the module is to be operated as a DP master or DP slave. There is only one DP master in a DP master system.

Under *Time synchronization* you set the synchronization mode for the real-time clock. As master, the real-time clock synchronizes the clocks in other devices; as slave, the real-time clock is synchronized by a clock in another device. This setting is independent of the mode as DP master or DP slave.

SYNC/FREEZE is a function for simultaneous output (SYNC) and/or reading-in (FREEZE) of the signal states of the DP slaves involved. Here you set which SYNC or FREEZE group the module is to belong to. Further details can be found in Chapter 16.4.5 “Special PROFIBUS configurations” on page 728.

Hardware identifier shows the hardware identifier of the interface, which is assigned and listed in the *System constants* tab of the default tag table by the configuration editor.

4 Tags, addressing, 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):

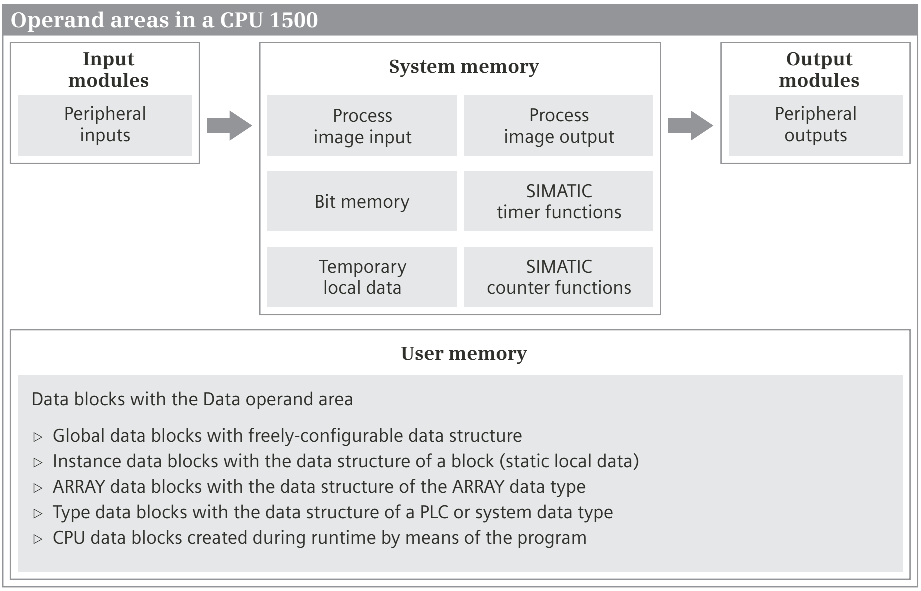

Operand areas in a CPU 1500
Input modules
Peripheral inputs
System memory
Process image input
Process image output
Bit memory
SIMATIC timer functions
Temporary local data
SIMATIC counter functions
Output modules
Peripheral outputs
User memory
Data blocks with the Data operand area
▷ Global data blocks with freely-configurable data structure
▷ Instance data blocks with the data structure of a block (static local data)
▷ ARRAY data blocks with the data structure of the ARRAY data type
▷ Type data blocks with the data structure of a PLC or system data type
▷ CPU data blocks created during runtime by means of the program

Fig. 4.1 Operand areas in a PLC station

- *Peripheral inputs* are the memory areas on the input modules. They constitute the direct interface to the controlled machine or plant, 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 work memory. These are normally processed by the user program when signal states of the machine or plant 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 work 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 for signal states and numerical values.

- The SIMATIC timer/counter functions save their data – contrary to the IEC timer/counter functions – at a fixed position in the system memory. Therefore the SIMATIC timer/counter functions have a fixed number range and their number depends on the memory space provided by a CPU for this purpose. You can find a description of these functions in Chapters 12.4 “SIMATIC timer functions” on page 524 and 12.6 “SIMATIC counter functions” on page 545.

- *Temporary local data* refers to memory areas assigned by the CPU to a code 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.

- The term *Data* describes tags in the user memory which are compiled in *data blocks* with various structures. A data block, which is assigned to a code block (instance data block), contains the operand area *static local data*.

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 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 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), one doubleword (32 bits), or one long word (64 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

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 system program into the process image input prior to each processing cycle of the user program (see Chapter 5.6.2 “Process image updating” on page 179).

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.

The peripheral inputs are read-only. Inputs can be read and written. Inputs not occupied by peripheral inputs can be used as additional intermediate memories like the bit memories.

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 system program 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 updating” on page 179).

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 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 process image output, 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 address areas: a user data area which can be directly addressed by loading and transferring, and a system data area for the transfer of data records.

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 technology and communication modules. The amount of user data is module-specific. There are modules which occupy one, two, four, or more bytes in this area. Occupation always commences at the relative byte 0. The address of the relative byte 0 is the module start address, which is defined by the hardware configuration.

The user data represents the *I/O* operand area, divided into peripheral inputs and peripheral outputs depending on the transfer direction. The data transfer between the I/O area and the process images can be controlled from the system program or from the user program.

Consistent user data transfer

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 1500 retains the data consistency for tags with all elementary data types and system data types. This means a read or write operation for a tag with one of these data types cannot be interrupted.

The following system functions are available for the consistent transfer of large volumes of data: UBLKMOV, which performs an uninterruptable transfer of a tag or an absolutely addressed data area, UMOVE_BLK, which performs an uninterruptable transfer of elements of an ARRAY data field, and UFILL_BLK, which fills an area of an ARRAY data field without interruption. Since the copy process using these system functions cannot be interrupted by other actions of the operating system, the response time to an interrupt that occurs during the transfer can be increased.

Interrupt events can be blocked and released using the system functions DIS_IRT and EN_IRT. The system functions DIS_AIRT and EN_AIRT allow higher-priority interrupt events to be delayed. During the blocking or delaying phase, the processing of the program cannot be interrupted (this includes any data transfer that was initiated by the user program during this time).

On the system side, communication with the programming device or with other PLC stations can interrupt the execution of the program, because the communication takes place in “time slice mode” during program execution. Communication has priority 15. A program in an organization block which has an execution priority higher than 15 thus cannot be interrupted by communication functions of the operating system.

Data between the CPU and an IO device or a DP standard slave can be consistently transferred using the system functions described in Chapter 16.5.1 “Read and write user data” on page 730.

During data exchange between PLC stations, the volume of the consistently transferred data depends on the communication functions used. The coordination of access to the transferred data in the user program can take place using the control parameters of the communication functions. For the unilaterally configured data transfer with GET and PUT, during which the operating system takes over the data transfer in the server CPU, the maximum volume of consistently transferred data is 462 bytes for a CPU 1500.

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

Value status

The value status (quality information, QI) indicates the validity of an I/O signal. The value status occupies one bit per I/O channel in the process image input. If this bit has signal state “0”, the value of the assigned I/O channel is invalid.

The value status is activated for correspondingly configured modules with the hardware configuration. For digital input modules, the value status is then saved at the user data address in the process image. Depending on the number of output signals, additional input bytes are assigned for digital output modules. The process is similar for analog modules: The value status of analog input modules is then saved at the user data address in the process image input (one bit per analog channel). For analog output modules, an additional input byte is assigned for the value status.

The value status is transferred along with the user data.

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 and 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 “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 Chapter 5.1.4 “Retentive behavior of operands” on page 148.

System memory

A CPU 1500 provides a memory byte whose signal states are 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 system memory byte is activated. You can change the default identifiers. The individual bits have the following meanings:

- Bit 0: Is set to signal state “1” if the main program is executed for the first time after the CPU is switched on. For all of the other execution cycles, it has signal state “0”.

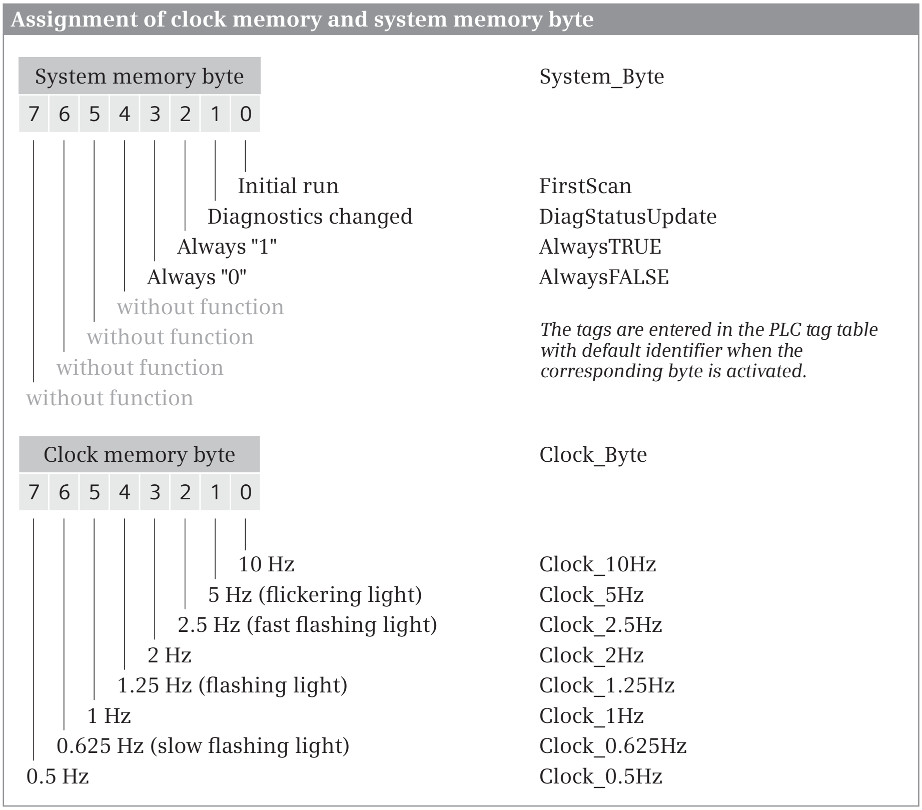


Fig. 4.2 Assignment of the system and clock memory byte

- ▷ 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 memories

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

Clock memories are memories whose signal state changes periodically with a pulse-to-pause ratio of 1:1. The clock memories 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 several 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. Certain system blocks also use instance data blocks.
- ▷ ARRAY data blocks have the structure of the ARRAY data type: They consist of tags with similar data types. The index starts with zero and has a configurable upper limit.
- ▷ 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.
- ▷ CPU data blocks are data blocks that are created with CREATE_DB during runtime by the user program.

Data blocks are global objects which are addressed in absolute mode using a 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 block, ARRAY data block), function block (instance data block), data type (type data block), or template (CPU data block). 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 write-protected in the 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 Chapter 5.1.4 “Retentive behavior of operands” on page 148.

4.1.5 Operand area: temporary local data

Temporary local data includes operands that are saved in the local data stack (L stack) in the CPU's system memory. Temporary local data is available in each code 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 from other blocks.

The operating system of a CPU 1500 provides 64 KB of temporary local data for each priority class (e.g. in the main program or in the hardware interrupt program) and a maximum of 16 KB for processing an individual block.

Organization blocks with standard access (the block attribute *Optimized block access* is deactivated) provide 20-byte long start information in the temporary local data. The general structure is described in Chapter 4.11.4 “Start information” on page 142. A supplemental description on the contents of the start information can be found at the corresponding organization block. In some cases, organization blocks with the activated *Optimized block access* attribute provide start information in the block interface as input parameters.

The amount of temporary local data required by a block which has already been compiled 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.

Use of temporary local data

The tags in the operand area Temporary local data are declared in the block interface. They can accept all of the elementary, structured, PLC and most system data types. All operations which also apply to the bit memories are permissible for the temporary local data. Note, however, that the values of the temporary local data lose their validity when block processing is finished. For example, a temporary local data bit is not suitable as an edge memory bit since it does not retain its signal state beyond block processing.

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

For blocks with standard access, the temporary local data have a quasi-random value before they are written for the first time. For blocks with the *Optimized block access* attribute activated, tags with an elementary data type or components of structured tags with an elementary data type have the default value preset. For temporary local data with a PLC data type, the components are given the start value from the declaration of the PLC data type (exception: STRING tags). STRING tags are provided with the correct length specifications and the characters are preset with '$00'.

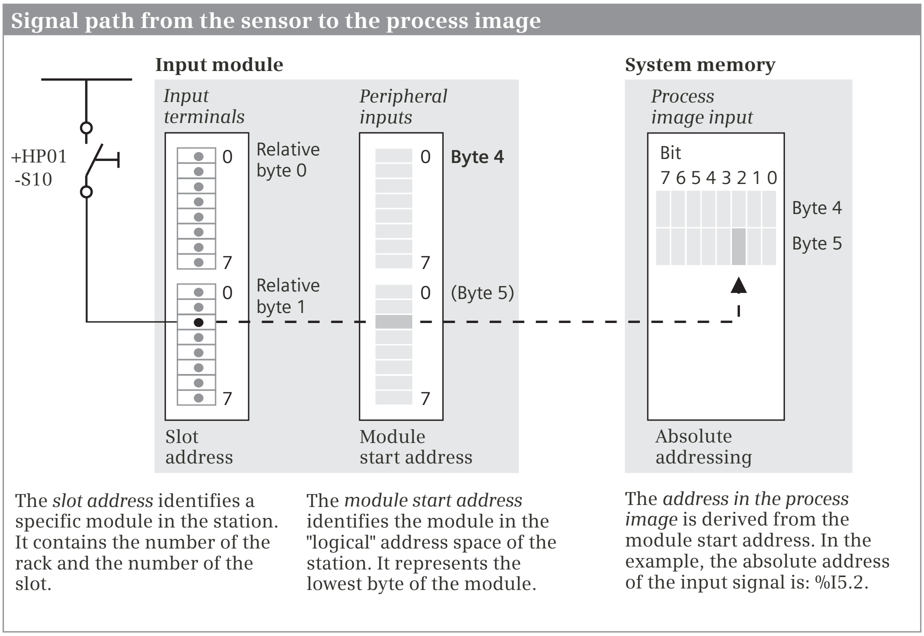
Temporary local data is addressed symbolically. The exceptions for STL are described in Chapter 10.7.6 “Absolute addressing of temporary local data” on page 454.

4.2 Addressing of operands and tags

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 also use the hardware configuration to set the module start address with which the signals are addressed by the module in the user program. This setting is simultaneously the address in the process image.



Signal path from the sensor to the process image

The *slot address* identifies a specific module in the station. It contains the number of the rack and the number of the slot.

The *module start address* identifies the module in the "logical" address space of the station. It represents the lowest byte of the module.

The *address in the process image* is derived from the module start address. In the example, the absolute address of the input signal is: %I5.2.

Fig. 4.3 Signal path from sensor to process image

The CPU copies the signal from the input module into the process image input by default every time before program execution is started, where it is then addressed as the operand "Input" (e.g. %I 5.2). The expression "%I 5.2" 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.

4.2.2 Absolute addressing

During absolute addressing, a signal state or a numerical value is addressed directly via 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. An absolute address is displayed with a preceding percent sign (%).

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. Fig. 4.4 clarifies this using an example of memory bytes %MB24 to %MB27.

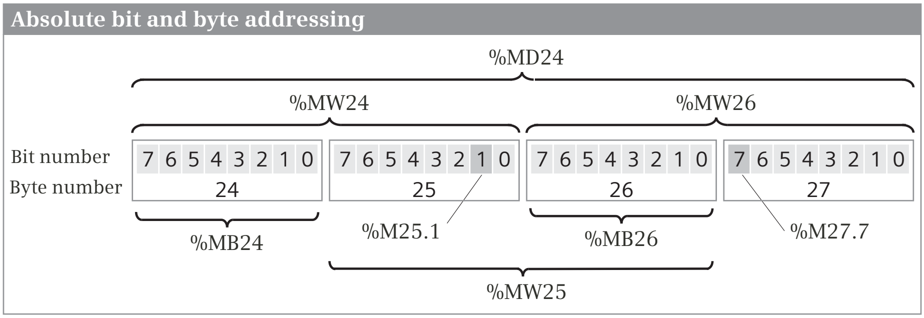


Fig. 4.4 Example of bit and byte assignments

The absolute addressing of a 64-bit wide tag (the absolute addressing of a long word) is not possible.

Absolute addressing of inputs, outputs, and bit memories

The addresses of the peripheral inputs and outputs (the input and output channels on the modules) are defined during configuration of the station design using the hardware configuration. The assigned inputs and outputs in the process image have the same addresses. To identify a peripheral address, ":P" is appended to the input or output address (Table 4.1).

Table 4.1 Absolute addressing of inputs, outputs, and bit memories

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	The input operand is expanded with :P	%Iy.x:P	%IBy:P	%IWy:P	%IDy:P
Output	Q	%Qy.x	%QBy	%QWy	%QDy
Peripheral output	The output operand is expanded with :P	%Qy.x:P	%QBy:P	%QWy:P	%QDy:P
Bit memory	M	%My.x	%MBy	%MWy	%MDy

y = byte address; x = bit address

A peripheral address is only considered to be present if the correspondingly addressed module is also present. Access to a non-existent peripheral address triggers an error. The operand areas Inputs, Outputs, and Bit memories are present in the complete, CPU-specific length. Therefore inputs and outputs which are not assigned to a module can also be addressed. In this case they behave like bit memories.

Absolute addressing of data operands

Addressing of tags in an ARRAY data block is described in Chapter 4.3.3 "Indirect addressing of a tag in an ARRAY DB" on page 102.

A data operand is a local tag within a data block. If addressing of the data operand is carried out in conjunction with the data block, the data operand is unique on the CPU, in other words it is a global tag. In the case of this "complete addressing", the data block precedes the data operand. For example, %DB10.DBW4 addresses data word 4 in data block 10. The data operand itself can be addressed with a width of bit, byte, word or doubleword (Table 4.2).

Table 4.2 Absolute complete addressing of data operands

Operand area	Operand ID	Bit (1 bit)	Byte (8 bits)	Word (16 bits)	Doubleword (32 bits)
Data	DB	%DBz.DBXy.x	%DBz.DBBy	%DBz.DBWy	%DBz.DBDy

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

The numbering of the data blocks commences at 1 and ends at a CPU-specific upper limit. Data block DB 0 does not exist. The number and size of the data blocks depends on the CPU used. A data block for a CPU 1516 can be 5 MB in size, for example.

Data operands can only be addressed in absolute mode if the *Optimized block access* block attribute is deactivated in the data block. The absolute address of a data operand is shown in the *Offset* column of the block interface once the data block has been compiled.

The STL programming language gives you the capability of opening the corresponding data block beforehand and then only addressing the data operands themselves ("partial addressing"). This option is described in Chapter 10.7.5 "Partial addressing of data operands" on page 453.

You can also address a data operand in a data block which was transferred via a block parameter with the data type DB_ANY. Further details can be found in Chapters 4.8.6 "Parameter type DB_ANY" on page 133 and 4.3.4 "Indirect addressing of a data block" on page 102.

Absolute addressing of static local data

The static local data – just like the block parameters – are local tags in a function block which are addressed symbolically. The exceptions for STL are described in Chapter 10.7.5 "Partial addressing of data operands" on page 453.

The values of the block parameters and the static local data of a function block are present in a data block, and therefore these tags can be addressed by each code block like "normal" data tags. For data blocks with the *Optimized block access* attribute deactivated, absolute addressing can be used.

Absolute addressing of temporary local data

The temporary local data are local tags in a code block which are addressed symbolically. The exceptions for STL are described in Chapter 10.7.6 "Absolute addressing of temporary local data" on page 454.

Absolute addressing of SIMATIC timer/counter functions

The SIMATIC timer/counter functions present in the system memory are addressed by a number starting at 0. The upper limit of the numbering – according to the maximum number of timer and counter functions – is CPU-specific. The timer and counter functions can be selected as desired within the quantity framework. Example of absolute addressing: %T15 corresponding to the timer function number 15. Table 4.3 shows the operand IDs of these functions.

Table 4.3 Absolute addressing of SIMATIC timer and counter functions

Operand area	Operand ID	Address
SIMATIC timer function	T	n
SIMATIC counter function	C	n

n = number

4.2.3 Symbolic addressing

During symbolic addressing, an operand is assigned an alphanumeric identifier (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 operand areas Inputs, Outputs, Bit memories, SIMATIC timer functions, and SIMATIC counter functions. The name of a peripheral input is derived from the name of the input. The name of a peripheral output is derived from the name of the output.

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 its 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 code 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 character (#). If the name includes special characters, it is additionally indicated in quotation marks.

Symbolic addressing of data tags

Symbolic addressing of data tags is carried out during complete addressing. Symbolic partial addressing is not possible. With complete addressing, the data tag is given the characteristic of a global tag. Example: The tag name *Activate_motor* can be present in both data blocks *"Motor_1"* and *"Motor_2"*. The address *"Motor_1".Activate_motor* addresses a different tag than the address *"Motor_2".Activate_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 preceding number character, is specified: *#Local data*. For a function block, the instance data are local tags.

4.2.4 Addressing of a tag area

It is possible to address an area within a tag ("slice access"). This area can be a bit, byte, word, or doubleword.

If the block attribute *IEC check* is activated, the tag must have a bit-serial data type. If the block attribute *IEC check* is deactivated, it can also be a fixed-point data type. You address the areas within the tags as follows:

- A bit: *Tag_name.X<bit number>*
- A byte: *Tag_name.B<byte number>*
- A word: *Tag_name.W<word number>*
- A doubleword: *Tag_name.D<doubleword number>*

The numbering begins with zero at the least significant subarea 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 most significant bit (the sign bit of data type INT) is addressed with *"Store".Temperature.X15*.

4.2.5 Addressing a constant

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 4.6 on page 114). 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 a 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 another constant, 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.

The constants created by the configuration editor or the program editor are listed in the default tag table in the *System constants* tab. They can be used with their name or with the numeric value (see Chapter 4.12 "Hardware data types" on page 143).

4.3 Indirect addressing

4.3.1 Overview

Indirect addressing allows you to address operands whose addresses are only defined during runtime. You can also use indirect addressing to repeatedly execute program sections, e.g. in a loop, and address different operands in each cycle.

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!*

You have the following options for indirect addressing:

- ▷ Indirect addressing of ARRAY components.
  The index of an ARRAY tag is variable and can be calculated during runtime.
- ▷ Indirect addressing of components in an ARRAY data block.
  A data tag in an ARRAY data block can be addressed via an index with a variable value.
- ▷ Indirect addressing of a data block.
  The number of a data block that is transferred via a block interface can be changed during runtime.
- ▷ Indirect addressing using the “variable” ANY pointer.
  The value of a tag in the temporary local data with the data type ANY can be changed during runtime and used as an address for any operand area.
- ▷ For SCL: with PEEK and POKE.
  PEEK reads a value from an operand with an address than can be dynamically preset. PEEK writes a value to an operand with an address than can be dynamically preset.
- ▷ For STL: with address registers.
  Addressing via an address register allows access to operands with an address that is only calculated during runtime. The description can be found in Chapters 10.7.8 “Memory-indirect addressing” on page 458 and 10.7.9 “Register-indirect addressing” on page 461.

4.3.2 Indirect addressing of ARRAY components

A tag with the data type ARRAY comprises a fixed number of components with the same data type. Each array component can be individually addressed via an index with limits that are defined when the ARRAY tags are declared. Example: A tag with the name *#Measurement_series* and the data type ARRAY [1..12] OF INT consists of 12 components. The first component is addressed with *#Measurement_series[1]*.

An array component can also be addressed with an index tag with a value that is only calculated during runtime. Example: *#Measurement_series[#index]* addresses the array component with an index that is the value of the *#index* tag.

The index tag can be an absolutely or symbolically addressed global or local tag with a fixed-point data type (except for LINT and ULINT for LAD, FBD and STL). This indirect addressing is also possible with multi-dimensional arrays and with the addressing of partial arrays. An example is shown in Fig. 4.5.

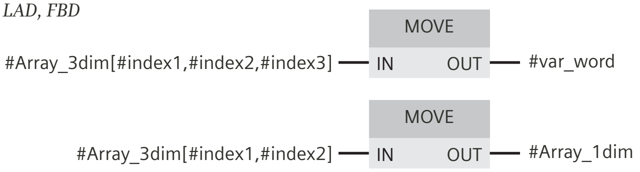
Examples of dynamic indexing of array components

Static local tags in the interface of a function block

Name	Declaration	Data type	Description
Array_1dim	Static	ARRAY [1..4] OF WORD	One-dimensional array
Array_3dim	Static	ARRAY [1..4,1..4,1..4] OF WORD	Three-dimensional array
index1	Static	INT	Index tag
index2	Static	INT	Index tag
index3	Static	INT	Index tag
var_int	Static	INT	INT tag
var_word	Static	WORD	WORD tag

Dynamic addressing of a component in a three-dimensional array
Assignment of a partial array

LAD, FBD


MOVE
#Array_3dim[#index1,#index2,#index3]
IN
OUT
#var_word
MOVE
#Array_3dim[#index1,#index2]
IN
OUT
#Array_1dim

SCL

```
#var_word  := #Array_3dim[#index1, #index2, #index3];
#Array_1dim := #Array_3dim[#index1, #index2];
```

STL

```
L   #Array_3dim[#index1, #index2, #index3]
T   #var_word

CALL BLKMOV
   Variant
   SRCBLK  := #Array_3dim[#index1, #index2]
   RET_VAL := #var_int
   DSTBLK  := #Array_1dim
```

Fig. 4.5 Examples of dynamic indexing of array components

Note: For an indirectly addressed ARRAY component which is created as an actual parameter on an in/out parameter, a change to the index tags in the block program has no effect. The value is written back into the same ARRAY component from which it was read.

4.3.3 Indirect addressing of a tag in an ARRAY DB

An ARRAY data block consists of data tags which all have the same data type and are addressed via an index. The general syntax for addressing a tag in an ARRAY data block is as follows:

- For LAD, FBD, and STL: "<DB_name>".THIS[<Index>].<Component_name>
- For SCL: "<DB_name>"."THIS"[<Index>].<Component_name>.

<DB_name> is the name of the ARRAY data block. <Index> addresses a component (a data tag) in the ARRAY data block. <Index> can be a constant or a tag with a fixed-point data type. <Index> always begins with zero and ends at a configurable upper limit. <Component_name> is the name of a component. If the component has a structured data type, the individual elements of the data type – separated by a dot – can be addressed directly.

The way in which an ARRAY data block is added is described in Chapter 14.4.3 "ARRAY data blocks" on page 641. Functions are available for accessing an ARRAY data block in the load memory. These are described in Chapter 14.4.4 "System blocks for access to ARRAY data blocks" on page 642. With these blocks, the data block can also be indirectly addressed.

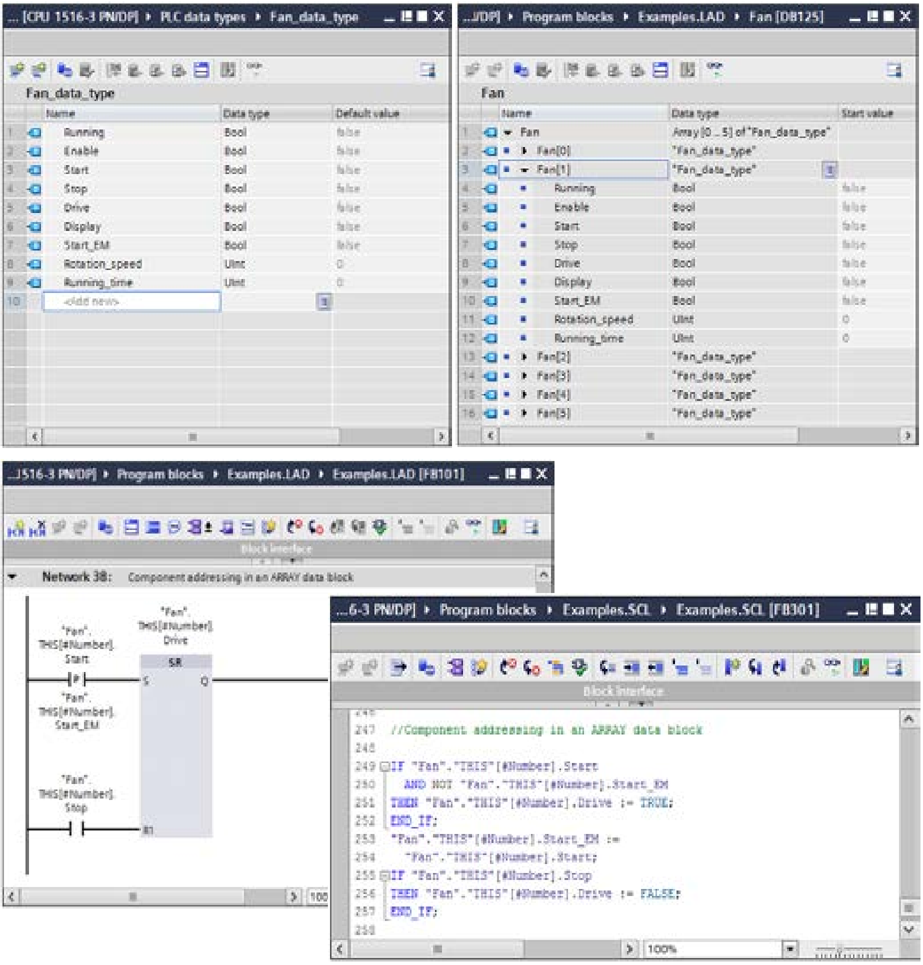
Fig. 4.6 shows an example of the structure of the data tags and their addressing in an ARRAY data block. A PLC data type with the name *Fan_data_type* has been created. Based on this PLC data type, an ARRAY data block with 6 components (numbered 0 to 5) has been added. All of the components have the PLC data type *Fan_-data_type*. The tags for controlling the fan are addressed in the program (*Start*, *Start_EM*, *Stop*, *Drive*). The fans to be controlled are selected using the *#Number* tag.

4.3.4 Indirect addressing of a data block

A data block is transferred to the called block via a block parameter with the data type DB_ANY (see Chapter 4.8.6 "Parameter type DB_ANY" on page 133). In the program of the called block, you can access a data operand of this transferred data block, although the data block and thus its structure is not known at the time the program is created.

LAD, FBD, SCL: A data operand in a transferred data block is addressed with the absolute address. If, for example, the block parameter with the data type DB_ANY has the name *#Data*, you can address a bit with *#Data.%DBXy.x*, a byte with *#Data.%DBBy*, a word with *#Data.%DBWy*, and a doubleword with *#Data.%DBDy* with *y* as byte number and *x* as bit number. Note that you can only determine the absolute address of a data operand – after the compilation – in a data block with the *Optimized block access* attribute deactivated.

STL: A data block that was transferred via a block parameter can be opened in the statement list both via the DB register and via the DI register. Then the data operands of this data block can be completely or partially addressed. Further details can be found in Chapters 10.7.4 "Working with the data block registers" on page 451 and 10.7.5 "Partial addressing of data operands" on page 453.


... [CPU 1516-3 PN/DP] ▸ PLC data types ▸ Fan_data_type
...1/DP] ▸ Program blocks ▸ Examples.LAD ▸ Fan [DB125]
...1516-3 PN/DP] ▸ Program blocks ▸ Examples.LAD ▸ Examples.LAD [FB101]
Block interface
...6-3 PN/DP] ▸ Program blocks ▸ Examples.SCL ▸ Examples.SCL [FB301]
100%

Fan_data_type

	Name	Data type	Default value
1	Running	Bool	false
2	Enable	Bool	false
3	Start	Bool	false
4	Stop	Bool	false
5	Drive	Bool	false
6	Display	Bool	false
7	Start_EM	Bool	false
8	Rotation_speed	UInt	0
9	Running_time	UInt	0
10	<Add new>		

Fan

	Name	Data type	Start value
1	▾ Fan	Array [0 .. 5] of "Fan_data_type"	
2	▸ Fan[0]	"Fan_data_type"	
3	▾ Fan[1]	"Fan_data_type"	
4	Running	Bool	false
5	Enable	Bool	false
6	Start	Bool	false
7	Stop	Bool	false
8	Drive	Bool	false
9	Display	Bool	false
10	Start_EM	Bool	false
11	Rotation_speed	UInt	0
12	Running_time	UInt	0
13	▸ Fan[2]	"Fan_data_type"	
14	▸ Fan[3]	"Fan_data_type"	
15	▸ Fan[4]	"Fan_data_type"	
16	▸ Fan[5]	"Fan_data_type"	

Network 38: Component addressing in an ARRAY data block

```
//Component addressing in an ARRAY data block

IF "Fan"."THIS"[#Number].Start
  AND NOT "Fan"."THIS"[#Number].Start_EM
THEN "Fan"."THIS"[#Number].Drive := TRUE;
END_IF;
"Fan"."THIS"[#Number].Start_EM :=
  "Fan"."THIS"[#Number].Start;
IF "Fan"."THIS"[#Number].Stop
THEN "Fan"."THIS"[#Number].Drive := FALSE;
END_IF;
```

Fig. 4.6 Example of tag addressing in an ARRAY data block

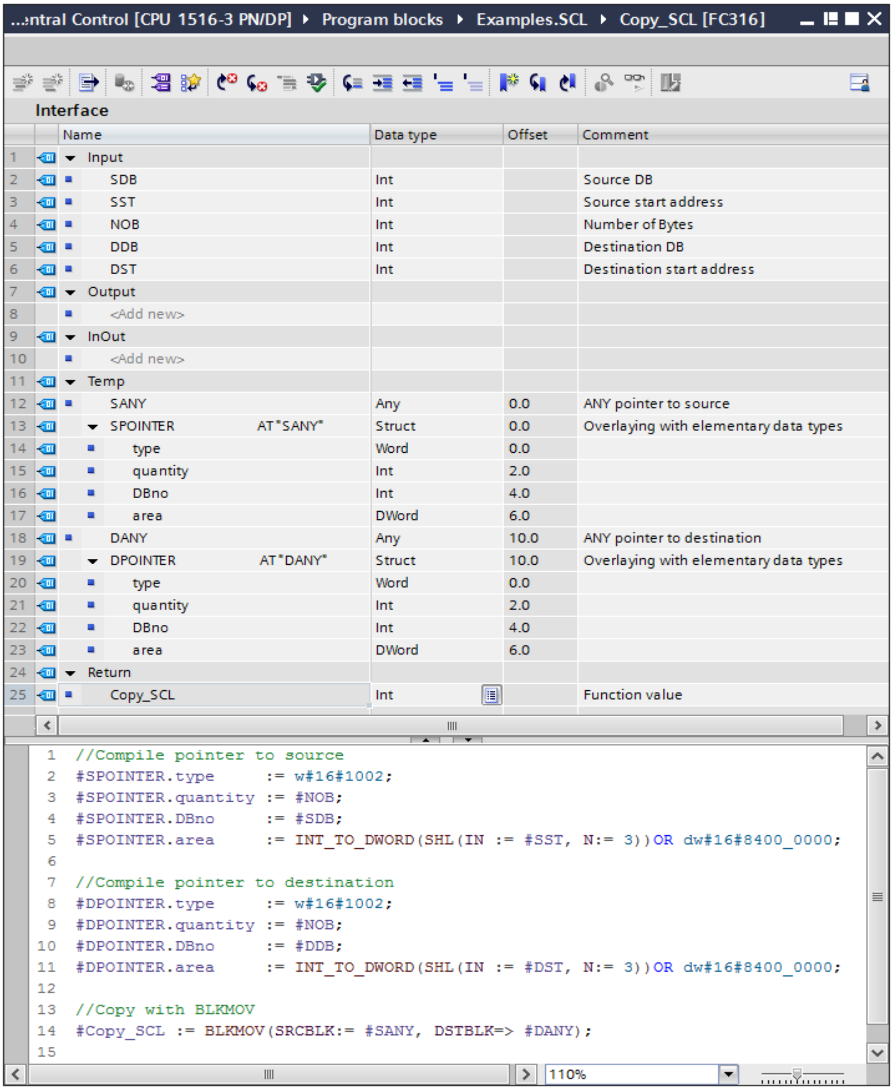
4.3.5 Indirect addressing with an ANY pointer

An ANY pointer in the representation of a constant *P#Operand Type Quantity* points to an operand area with fixed address. Even if you provide a tag for a block parameter with the data type ANY, the program editor will generate a constant ANY pointer to this tag. In neither case is it possible to change the tag or data area during runtime.

An exception is made by the program editor if the actual parameter itself is in the temporary local data and is of the type ANY. No other ANY pointer is then produced; in this case the program editor interprets this ANY tag as an ANY pointer to an actual parameter. This means that the ANY tag must be formatted like an ANY pointer and written with the required values in the user program prior to its use. The structure of an ANY pointer is described in Chapter 4.9.4 “ANY pointer” on page 135.

You can use the “variable” ANY pointer in a block with the *Optimized block access* attribute deactivated. To do this, first create a tag with the data type ANY in the temporary local data of the block interface. Overlay this tag with a data structure, which makes it possible to provide the individual components of the ANY pointer with values. You can now create the ANY pointer that is compiled in this way on a block parameter with the data type ANY of a block or of a (program) function.

Fig. 4.7 shows an example of application of the “variable” ANY pointer in the SCL programming language. It can also be formulated in LAD, FBD, or STL. The example contains the program for a function (FC) with the name “Copy”, which transfers a



...ntral Control [CPU 1516-3 PN/DP] ▸ Program blocks ▸ Examples.SCL ▸ Copy_SCL [FC316]

Interface

	Name		Data type	Offset	Comment
1	▾ Input				
2	SDB		Int		Source DB
3	SST		Int		Source start address
4	NOB		Int		Number of Bytes
5	DDB		Int		Destination DB
6	DST		Int		Destination start address
7	▾ Output				
8	<Add new>				
9	▾ InOut				
10	<Add new>				
11	▾ Temp				
12	SANY		Any	0.0	ANY pointer to source
13	▾ SPOINTER	AT"SANY"	Struct	0.0	Overlaying with elementary data types
14	type		Word	0.0	
15	quantity		Int	2.0	
16	DBno		Int	4.0	
17	area		DWord	6.0	
18	DANY		Any	10.0	ANY pointer to destination
19	▾ DPOINTER	AT"DANY"	Struct	10.0	Overlaying with elementary data types
20	type		Word	0.0	
21	quantity		Int	2.0	
22	DBno		Int	4.0	
23	area		DWord	6.0	
24	▾ Return				
25	Copy_SCL		Int		Function value

```
//Compile pointer to source
#SPOINTER.type     := w#16#1002;
#SPOINTER.quantity := #NOB;
#SPOINTER.DBno     := #SDB;
#SPOINTER.area     := INT_TO_DWORD(SHL(IN := #SST, N:= 3))OR dw#16#8400_0000;

//Compile pointer to destination
#DPOINTER.type     := w#16#1002;
#DPOINTER.quantity := #NOB;
#DPOINTER.DBno     := #DDB;
#DPOINTER.area     := INT_TO_DWORD(SHL(IN := #DST, N:= 3))OR dw#16#8400_0000;

//Copy with BLKMOV
#Copy_SCL := BLKMOV(SRCBLK:= #SANY, DSTBLK=> #DANY);

```

Fig. 4.7 Example of a “variable” ANY pointer in the SCL programming language

data area from one data block to another, where the addresses and the length of the area can be changed during runtime. The values for the source area and destination area that are individually specified via block parameters are compiled into two ANY pointers. At the end, the copy process is started with the BLKMOV function. The error information from BLKMOV is passed on to the function value (return value) of the “Copy” function and can be evaluated in the program of the calling block.

4.3.6 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. 4.8).

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. The byte address is provided 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 default data type is BYTE; it is used to read one byte. If two bytes should be read, note the statement 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 an operand area defined with the parameters 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 bit memory address area begins at the address *#M_addr* and is *#M_dis* bytes long. Both tags are declared with the INT data type.

```
FOR #i := #M_addr TO #M_addr + #M_dis - 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 control tag in the FOR statement and contains the address of the memory byte that is currently overwritten with 16#00.

Indirect addressing of an operand

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

SCL

```
#tag    := PEEK_Data type(
  AREA       := ... ,
  DBNUMBER   := ... ,
  BYTEOFFSET := ... );

#bit_tag := 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      := ... );

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

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

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, and at the data operand area with the data block number.
DBNUMBER, BYTEOFFSET, and BITOFFSET have a fixed-point data type, VALUE has a bit-serial data type.

Indirect addressing of an operand area

POKE_BLK transfers an indirectly addressed operand area to another indirectly addressed operand 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 operand area (SRC) and writes them to the destination operand area (DEST).

Data types:
The operand area is defined at the AREA_xxx 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, and at the operand area data with the data block number.
DBNUMBER_xxx, BYTEOFFSET_xxx, and COUNT have a fixed-point data type.

Fig. 4.8 Indirect operand addressing with PEEK and POKE

4.4 Addressing of hardware objects

The configuration editor assigns an unambiguous ID (the “hardware identifier”) for each hardware object. Thus, for example, each station, each module, each interface, or even each transfer area of an I-device can be addressed. The hardware identifier is specified in the object properties. It is a constant and cannot be changed. For example, the hardware identifier of a signal module does not change if the user data addresses (the logical addresses) are changed. Exception: For organization blocks, the value of the hardware identifier corresponds to the number of the organization block and can be changed later.

The data type of the hardware identifier follows that of the referenced object. Every hardware identifier has a name which you can change in the object properties (under *General*). The name, value, and data type of the current hardware identifiers are listed in the *System constants* tab of the default tag table.

Fig. 4.9 shows an example of the addressing of a module. When the module is “inserted” into the rack, the configuration editor assigns a name (displayed in the object properties under *General*, can be changed) and a value (under *Hardware identifier*, cannot be changed). The name and the value are listed in the *System constants* tab. When the module is addressed (in the example with the system block LOG2GEO), the parameter LADDR is provided with the value or the name from the system constants table.

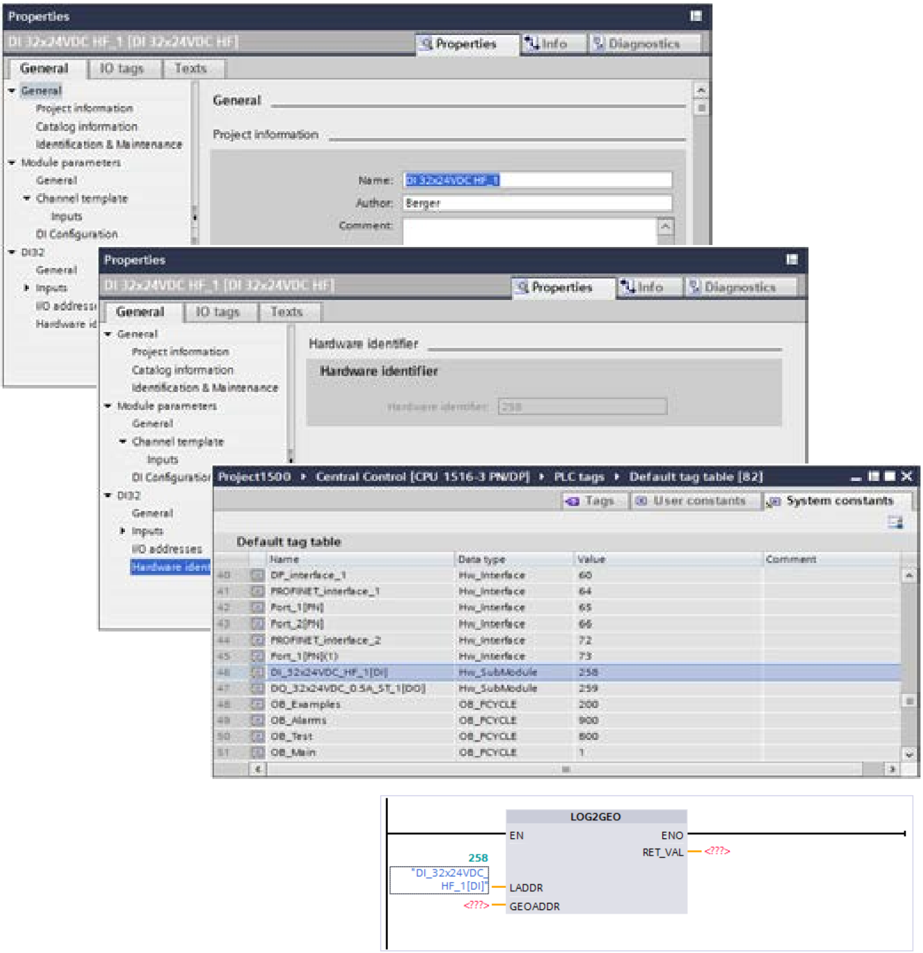
4.5 General information on data types

4.5.1 Overview of data types

Data types define the properties of tags, basically the representation of the contents and the permissible value range. 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 pre-defined and cannot be further subdivided
- Structured data types, which are predefined and comprise a combination of elementary data types
- Parameter types as predefined, additional data types for the transfer of tags to block parameters of functions and function blocks
- PLC data types, which a user can compile from existing data types
- System data types, which are provided by the program editor in STEP 7 and have a fixed structure
- Hardware data types, which are defined by the configuration editor in STEP 7

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 acti-



Name: DI 32x24VDC HF_1

Author: Berger

Comment:

Hardware identifier: 258

Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ PLC tags ▸ Default tag table [82]
Tags
User constants
System constants

Default tag table

	Name	Data type	Value	Comment
40	DP_interface_1	Hw_Interface	60	
41	PROFINET_interface_1	Hw_Interface	64	
42	Port_1[PN]	Hw_Interface	65	
43	Port_2[PN]	Hw_Interface	66	
44	PROFINET_interface_2	Hw_Interface	72	
45	Port_1[PN](X1)	Hw_Interface	73	
46	DI_32x24VDC_HF_1[DI]	Hw_SubModule	258	
47	DIQ_32x24VDC_0.5A_ST_1[DO]	Hw_SubModule	259	
48	OB_Examples	OB_PCYCLE	200	
49	OB_Alarms	OB_PCYCLE	900	
50	OB_Test	OB_PCYCLE	800	
51	OB_Main	OB_PCYCLE	1	

Fig. 4.9 Using the hardware identifier

vated, the test is stricter. The block attribute *Optimized block access* can also play a role in the application of data types.

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 13.6 “Conversion functions” on page 586).

4.5.2 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. If the *IEC check* attribute is activated, 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 an input or output parameter which expects the data type WORD (destination data type). Conversely, it is possible to apply a WORD tag to an input or output parameter with the data type DWORD. The programmed bit length must agree with the expected bit length for an in-out parameter. The block attribute *IEC check* is used to set the strictness of the compatibility check (see Table 4.4).

For LAD and FBD, the implicit conversion is marked with a split rectangle symbol on the function input or output. At a transition from dark gray to light gray, a data type

Table 4.4 Implicit data type conversion

to from	BOOL	BYTE	WORD	DWORD	LWORD	USINT	UINT	UDINT	ULINT	SINT	INT	DINT	LINT	REAL	LREAL	S5TIME	TIME	LTIME	DATE	TOD	LTOD	DT	LDT	DTL	CHAR	STRING
BOOL																										
BYTE			X	X	X	O	O	O	O	O	O	O	O												O	
WORD		O		X	X	O	O	O	O	O	O	O	O			O			O						O	
DWORD		O	O		X	O	O	O	O	O	O	O	O	O			O			O					O	
LWORD		O	O	O		O	O	O	O	O	O	O	O		O			O			O		O		O	
USINT		O	O	O	O		X	X	X	O	X	X	X	X	X										O	
UINT		O	O	O	O	O		X	X	O	O	X	X	X	X				O						O	
UDINT		O	O	O	O	O	O		X	O	O	O	X	O	X		O			O					O	
ULINT		O	O	O	O	O	O	O		O	O	O	O	O	O			O			O		O		O	
SINT		O	O	O	O	O	O	O	O		X	X	X	X	X										O	
INT		O	O	O	O	O	O	O	O	O		X	X	X	X				O						O	
DINT		O	O	O	O	O	O	O	O	O	O		X	O	X		O			O					O	
LINT		O	O	O	O	O	O	O	O	O	O	O		O	O			O			O		O		O	
REAL				O		O	O	O	O	O	O	O	O		X											
LREAL					O	O	O	O	O	O	O	O	O	O												
S5TIME			O																							
TIME				O				O				O						X		O						
LTIME					O				O				O				O				O		O			
DATE			O				O				O													O		
TOD				O				O				O					O				X					
LTOD					O				O				O					O		O						
DT																							X	X		
LDT					O				O				O					O				O		X		
DTL																						O	X			
CHAR		O	O	O	O	O	O	O	O	O	O	O	O													X
STRING																									X	

Implicit data type conversion is possible: X Independent of attribute *IEC check*
O With deactivated attribute *IEC check*

conversion is possible without any loss of accuracy. At a transition from dark gray to white, an error may occur during the conversion. The ENO output is then set to “0”.

In Fig. 4.10, the display of the implicit data type conversion for LAD is displayed just as it is used for FBD. The *IEC check* attribute is deactivated. The addition is carried out according to the characteristic of the INT data type. A conversion from SINT to INT is possible without any loss of accuracy. An error can occur during a conversion from DINT to INT.

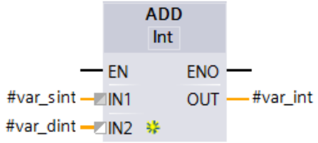


Fig. 4.10 Implicit data type conversion

For SCL, an implicit data type conversion is displayed with a yellow underscore if loss of accuracy can occur. A warning is then issued during compilation. 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);
```

Implicit data type conversion is not possible in the programming language STL. STL interprets the contents of accumulators according to the executed operation and independent of the significance of the bit assignments, i.e. independent of the (actual) data type. For example, the +I operation (integer addition) interprets the contents of the accumulators as numbers with data format INT and adds them together according to the integer rules. The programmer is responsible for ensuring that numbers with data format INT are actually present in the accumulators during execution of the operation.

An error is reported if the permissible numerical range of the destination data type is left or the sign is lost during implicit conversion.

Implicit conversion of bit-serial data types

Implicit conversion is not possible for the data type BOOL.

If the length of the source tag is equal to or shorter than the destination tag for the data types BYTE, WORD, DWORD and LWORD, the bit pattern is entered right-justified in the destination tag and the free bit positions are filled with “0”. If the source tag is longer than the destination tag, the bit pattern is entered in the destination tag starting from the right and the “excess” bit positions are ignored.

During a conversion to a floating-point data type, the value of the source tag is converted into the format of the destination data type. Example: DW#16#0000_000A is converted into REAL#10.0 (DW#16#4120_0000).

Implicit conversion of fixed-point data types

During a conversion from a fixed-point data type to a floating-point data type, the value of the source tag is converted into the format of the destination tag. During a conversion from a fixed-point data type to a fixed-point data type, the value of the source tag is transferred without changes and right-justified to the destination tag and the sign is updated.

Implicit conversion of floating-point data types

During a conversion from a floating-point data type to a fixed-point data type, the value of the source tag is rounded and converted into the format of the destination tag. Example: LREAL#317.8 is converted to INT#318 and this value is then converted to USINT#62.

Implicit conversion of duration data types

If the value of the source tag lies outside the value range of the destination type, the value of the destination tags is not changed. In all other cases, the bit pattern of the source tag is transferred unchanged to the destination tag.

Implicit conversion of date and time

For a conversion to a date/time data type, the value of the source tag is entered in the destination tag in the proper format, with a possible loss of accuracy. For a conversion to a different data type, the bit pattern of the source tag is transferred unchanged to the destination tag.

4.5.3 Overlaying tags (data type views)

A tag can be “overlaid” by further data types. It is then possible to address the contents of tags completely or partially using various data types. 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). Table 4.5 shows which combinations are permitted when overlaying.

You can program overlaying only in the interface of code blocks. In addition, the attribute *Optimized block access* must be deactivated. Exception: If the retentivity setting for a tag in a function block is *Set in IDB*, this tag can also be overlaid with another data type. For an FC block, the overlaying tag must have the same width as the “original” tag.

You initially declare the tag with the “original” data type and with any default setting. In the next line you write the tag which is to “overlay” the one above it. You then write the keyword AT in the *Data type* column to indicate that this is a “overlaid” data type definition, and then complete the input using the RETURN key. You subsequently assign this tag with the additional data type envisaged for it.

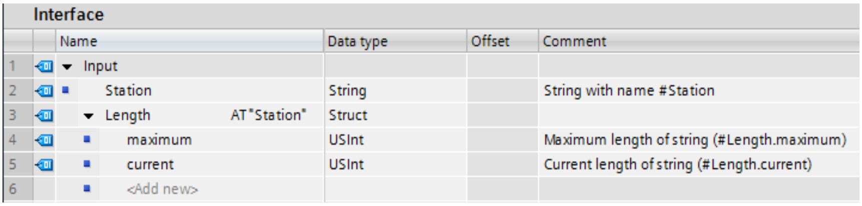
You can overlay a tag with several data type definitions which you differentiate by different names. A default setting with fixed values (initialization) is not possible.

Example: You declare an input parameter in the block interface of a function block 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 *maximum* and *current*, each with the data type USINT (Fig. 4.11). You can now address the current length of the tag *#Station* with *#Length.current* in the block program. You can address an individual character with *#Station[<index>]*, for example, *#Station[1]* for the first character.

Table 4.5 Permitted combinations when overlaying data types

Declaration section (FB)	**“Original” data type**	**can be overlaid with data type**		
Input	Elementary Structured ANY, POINTER DB_ANY	Elementary Elementary – Elementary	Structured Structured Structured Structured	– ANY, POINTER – –
Output, Static	Elementary Structured ANY, POINTER DB_ANY	Elementary Elementary – Elementary	Structured Structured – Structured	– – – –
InOut	Elementary Structured ANY, POINTER DB_ANY	Elementary – – –	– Structured – –	– – – –
Temp	Elementary Structured ANY, POINTER DB_ANY	Elementary Elementary – –	Structured Structured Structured –	– ANY, POINTER – –
Declaration section (FC)	**“Original” data type**	**can be overlaid with data type**		
Input, Output, InOut	Elementary Structured ANY, POINTER DB_ANY	elementary 1) – – elementary 1)	– structured 1) – structured 1)	– – – –
Temp	Elementary Structured ANY, POINTER DB_ANY	Elementary Elementary – –	Structured Structured Structured –	– ANY, POINTER – –
Declaration section (OB)	**“Original” data type**	**can be overlaid with data type**		
Temp	Elementary Structured ANY, POINTER DB_ANY	Elementary Elementary – –	Structured – Structured –	– ANY, POINTER – –

1) only with the same width



Interface

	Name		Data type	Offset	Comment
1	Input				
2	Station		String		String with name #Station
3	Length	AT "Station"	Struct		
4	maximum		USInt		Maximum length of string (#Length.maximum)
5	current		USInt		Current length of string (#Length.current)
6	<Add new>				

Fig. 4.11 Example of declaration of a “overlaid” data type

You use a tag with an 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 *Station*; the overlaying data type definition as a byte structure is not accessible to it.

4.6 Elementary data types

Elementary data types are pre-defined data types which cannot be further subdivided. You can find an overview of the elementary data types in Table 4.6. The data types BCD16 and BCD32 are not data types in the closer sense – they cannot be assigned to a tag; they are only relevant to data type conversion. The elementary data types can be used together with tags from all operand areas.

4.6.1 Bit-serial data types BOOL, BYTE, WORD, DWORD, and LWORD

Fig. 4.14 shows the structure of the data types BYTE, WORD, DWORD, and LWORD.

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.

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

A tag with data type WORD occupies 16 bits. The individual bits have no significance. The hexadecimal notation for constants is W#16#0000 to W#16#FFFF. A constant of word width can also be written as a 16-bit binary number (2#0000_..._0000 to 2#1111_..._1111).

A tag with data type DWORD occupies 32 bits. The individual bits have no significance. The hexadecimal notation for constants is DW#16#0000_0000 to DW#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).

A tag with data type LWORD (long word) occupies 64 bits. The individual bits have no significance. The hexadecimal notation for constants is LW#16#0000_0000_0000_0000 to LW#16#FFFF_FFFF_FFFF_FFFF. A constant of long word width can also be written as a 64-bit binary number (2#0000_..._0000 to 2#1111_..._1111).

4.6.2 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. The character is entered in single quotation marks. Example of the notation: ’A’ or CHAR#’A’. Special characters can be entered with a preceding dollar sign; Fig. 4.13 shows a selection.

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.

Table 4.6 Overview of elementary data types

Bit-serial data types			
BOOL	1 bit	1-bit binary value	0, 1, FALSE, TRUE
BYTE	8 bits	8-bit binary value	B#16#00 to B#16#FF
WORD	16 bits	16-bit binary value	W#16#0000 to W#16#FFFF
DWORD	32 bits	32-bit binary value	DW#16#0000 0000 to DW#16#FFFF FFFF
LWORD	64 bits	64-bit binary value	LW#16#0000 0000 0000 0000 to LW#16#FFFF FFFF FFFF FFFF
Characters			
CHAR	8 bits	A character in ASCII code	'a', 'A', '1', ...
BCD numbers [1]			
BCD16	16 bits	3 decades with sign	−999 to +999
BCD32	32 bits	7 decades with sign	−9 999 999 to +9 999 999
Unsigned fixed-point numbers			
USINT	8 bits	Unsigned 8-bit fixed-point number	0 to 255
UINT	16 bits	Unsigned 16-bit fixed-point number	0 to 65 535
UDINT	32 bits	Unsigned 32-bit fixed-point number	0 to 4 294 967 296
ULINT	64 bits	Unsigned 64-bit fixed-point number	0 to 18 446 744 073 709 551 615
Fixed-point numbers with sign			
SINT	8 bits	8-bit fixed-point number	−128 to +127
INT	16 bits	16-bit fixed-point number	−32 768 to +32 767
DINT	32 bits	32-bit fixed-point number	−2 147 483 648 to +2 147 483 647
LINT	64 bits	64-bit fixed-point number	−9 223 372 036 854 775 808 to +9 223 372 036 854 775 807
Floating-point numbers			
REAL	32 bits	32-bit floating-point number	approx. $\pm 1.18\times 10^{-38}$ to $\pm 3.40\times 10^{38}$
LREAL	64 bits	64-bit floating-point number	approx. $\pm 2.23\times 10^{-308}$ to $\pm 1.80\times 10^{308}$
Durations			
S5TIME	16 bits	Duration in SIMATIC format (in intervals of 10 ms)	S5T#0h0m0s0ms to S5T#2h46m30s0ms
TIME	32 bits	Duration in IEC format (number of milliseconds)	T#−24d20h31m23s648ms to T#+24d20h31m23s647ms
LTIME	64 bits	Duration in IEC format (number of nanoseconds)	LT#−106751d23h47m16s854ms775us808ns to LT#+106751d23h47m16s854ms775us807ns
Points in time (date and time of day)			
DATE	16 bits	Date (number of days)	D#1990-01-01 to D#2168-12-31
TOD	32 bits	Time of day (number of milliseconds)	TOD#00:00:00.000 to TOD#23:59.59.999
LTOD	64 bits	Time of day (number of nanoseconds)	LTOD#00:00:00.000000000 to LTOD#23:59:59.999999999
LDT	128 bits	Date and time of day (number of nanoseconds)	LDT#1970-01-01-0:0:0.000000000 to LDT#2262-04-11-23:47:16.854775807

[1] Not data types in a narrower sense; only relevant to data type conversion

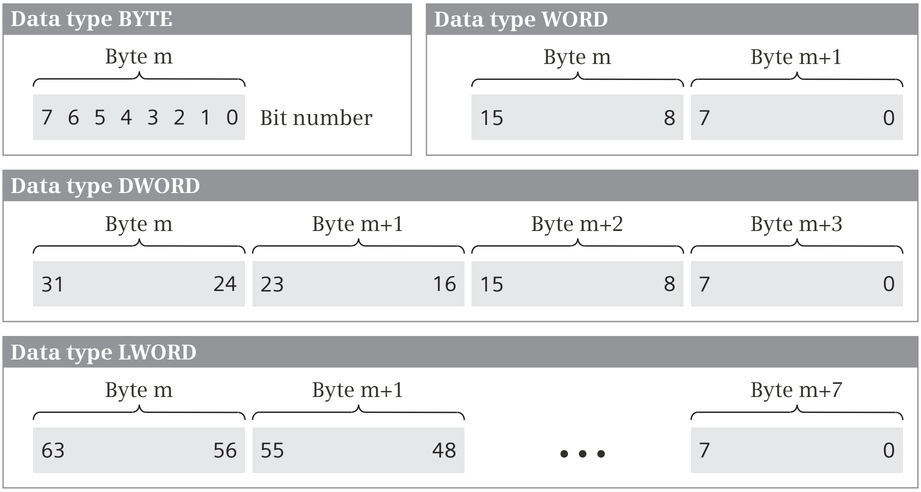
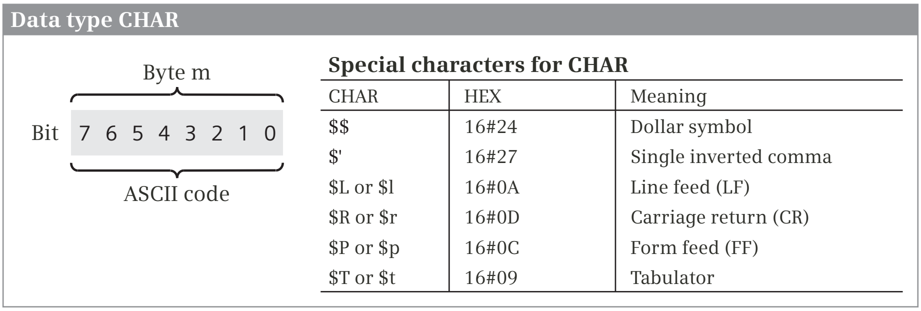


Fig. 4.12 Assignment of bit-serial data types



Special characters for CHAR

CHAR	HEX	Meaning
$$	16#24	Dollar symbol
$'	16#27	Single inverted comma
$L or $l	16#0A	Line feed (LF)
$R or $r	16#0D	Carriage return (CR)
$P or $p	16#0C	Form feed (FF)
$T or $t	16#09	Tabulator

Fig. 4.13 Structure of the CHAR data type

4.6.3 BCD numbers BCD16 and BCD32

BCD 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 the hexadecimal form W#16#xxxx or DW#16#xxxx_xxxx. For a positive, three-decade decimal number you can also use the notation C#0 to C#999.

BCD numbers are used, for example, in association with the conversion functions. The sign of a BCD number is located in the left-justified (highest) decade. Thus one decade is lost in the number range (Fig. 4.14).

The sign of a BCD number present in a 16-bit word is in the bits 12 to 15, where only bit 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.

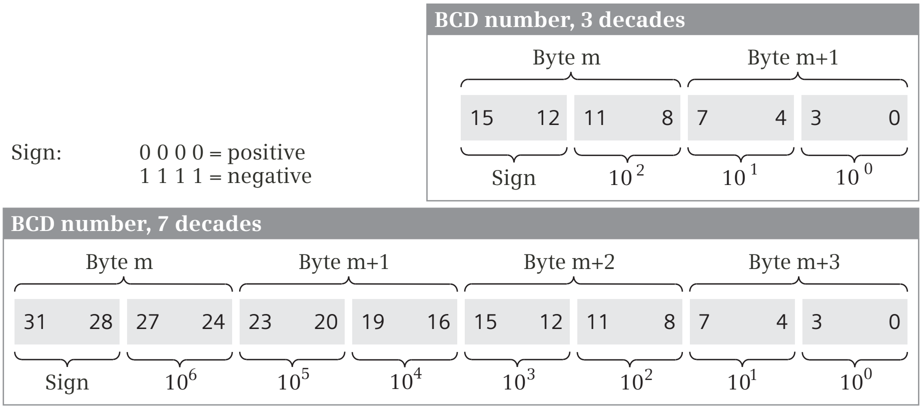


Fig. 4.14 Structure of BCD data types

The sign of a BCD number present in a 32-bit word is in the bits 28 to 31.

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

4.6.4 Fixed-point data types without sign USINT, UINT, UDINT, ULINT

The data type USINT (unsigned short integer, unsigned short 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 B#16#00 to B#16#FF (Fig. 4.15).

The data type UINT (unsigned integer, unsigned 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 W#16#0000 to W#16#FFFF.

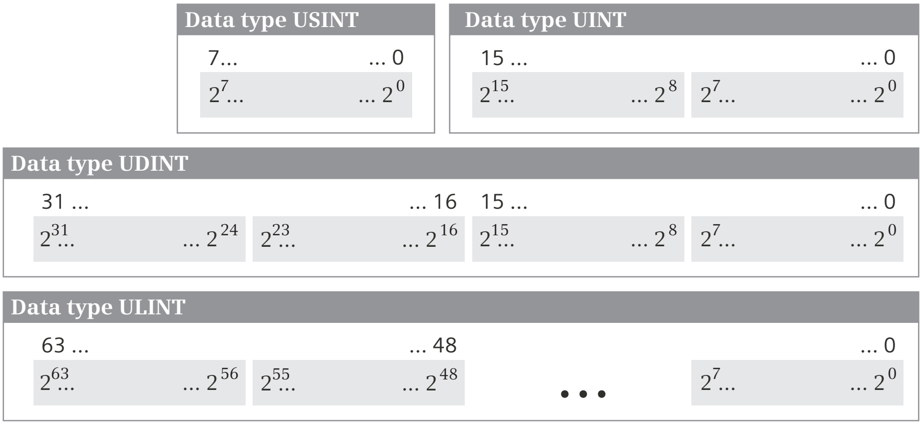


Fig. 4.15 Bit assignment of data types USINT, UINT, UDINT, and ULINT

The data type UDINT (unsigned double integer or unsigned, double-width 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 DW#16#0000 0000 to DW#16#FFFF FFFF.

The data type ULINT (unsigned long integer, unsigned long fixed-point number) occupies one long word. The numerical range extends from 20 to 2^{64}–1, i.e. from 0 to 18 446 744 073 709 551 615, or in hexadecimal notation from LW#16#0000 0000 0000 0000 to LW#16#FFFF FFFF FFFF FFFF.

4.6.5 Fixed-point data types with sign SINT, INT, DINT, and LINT

With the fixed-point data types with sign, the signal state of the highest bit represents the sign. 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, short 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 B#16#80 to B#16#7F (Fig. 4.16).

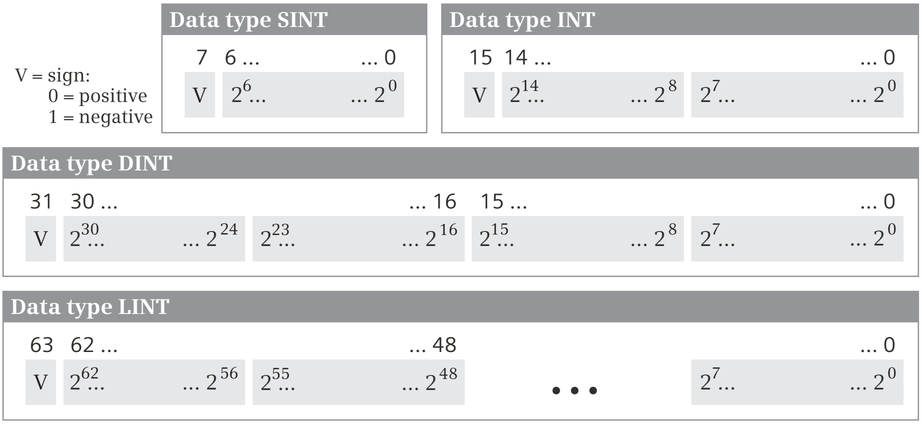

V = sign:
0 = positive
1 = negative
Data type SINT
7 6 0
V 2^{6}... ... 2^{0}
Data type INT
15 14 0
V 2^{14}... ... 2^{8} 2^{7}... ... 2^{0}
Data type DINT
31 30 ... ... 16 15 0
V 2^{30}... ... 2^{24} 2^{23}... ... 2^{16} 2^{15}... ... 2^{8} 2^{7}... ... 2^{0}
Data type LINT
63 62 ... ... 48 ... 0
V 2^{62}... ... 2^{56} 2^{55}... ... 2^{48} • • • 2^{7}... ... 2^{0}

Fig. 4.16 Bit assignment of data types SINT, INT, DINT, and LINT

The data type INT (integer, 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 W#16#8000 to W#16#7FFF.

The data type DINT (double integer, double-width 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 DW#16#8000 0000 to DW#16#7FFF FFFF.

The data type LINT (long integer, long fixed-point number) occupies one long word. The numerical range extends from -2^{63} to $+2^{63}-1$, i.e. from –9 223 372 036 854 775 808 to +9 223 372 036 854 775 807, or in hexadecimal notation from
LW#16#8000 0000 0000 0000 to LW#16#7FFF FFFF FFFF FFFF.

4.6.6 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 4.7 shows the internal range limits of a floating-point number.

Table 4.7 Internal range limits of a floating-point number

Sign	Exponent for REAL	Exponent for 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)

For floating-point numbers, a distinction is made between "normalized" floating-point numbers, which can be shown with complete accuracy, and "denormalized" floating-point numbers with limited accuracy. A CPU 1500 cannot calculate with denormalized floating-point numbers. A denormalized floating-point number is interpreted like a zero. If a calculated result falls in this range, it is displayed as zero and an downward violation of the numerical range is reported.

Data type REAL

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

$-3.402\,823 \times 10^{+38}$ to $-1.175\,495 \times 10^{-38}$
±0
$+1.175\,495 \times 10^{-38}$ to $+3.402\,823 \times 10^{+38}$

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. 4.17). A number in the REAL format is displayed by STEP 7 rounded to seven decimal points.

Data type LREAL

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

$-1.797\,693\,134\,862\,3158 \times 10^{+308}$ to $-2.225\,073\,858\,507\,2014 \times 10^{-308}$
±0
$+2.225\,073\,858\,507\,2014 \times 10^{-308}$ to $+1.797\,693\,134\,862\,3158 \times 10^{+308}$

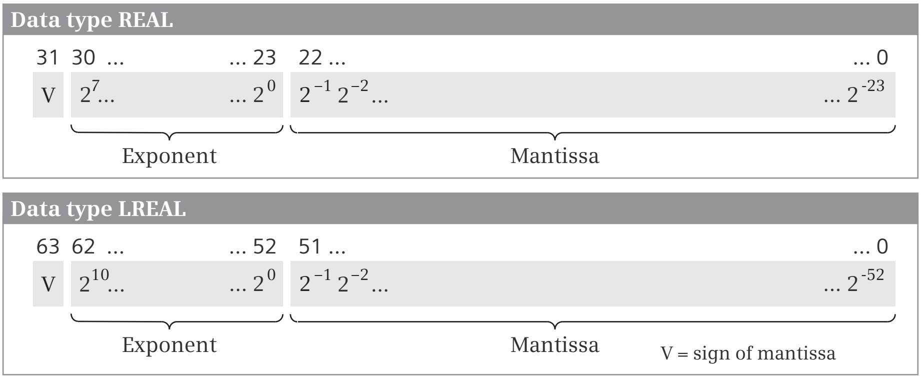


Fig. 4.17 Bit assignment of data types REAL and LREAL

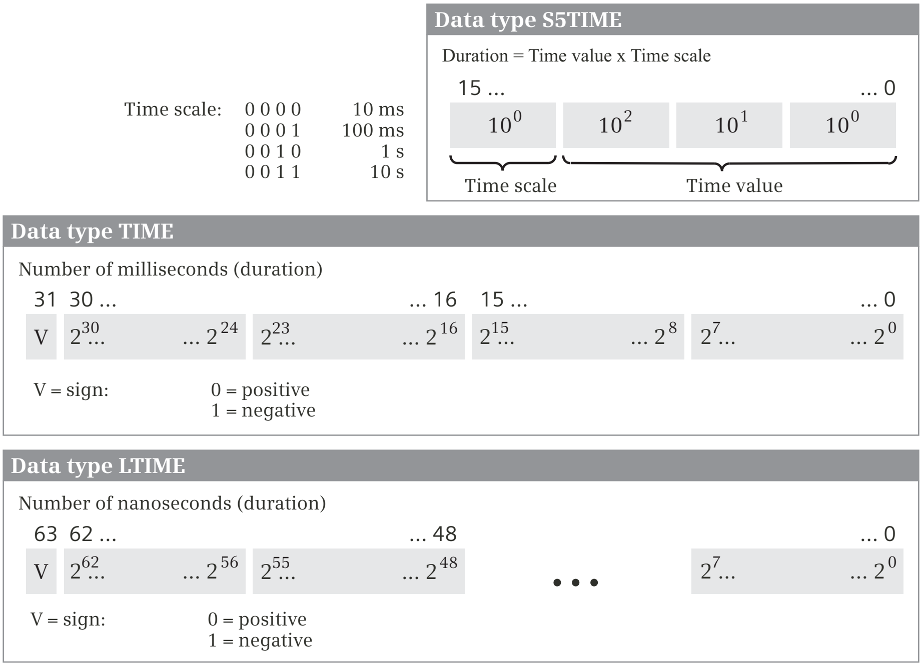
A tag with data type LREAL consists internally of three components: the sign, the 11-bit exponent to base 2, and the 52-bit mantissa. The sign can have the values "0" (positive) or "1" (negative).

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. A number in the LREAL format is displayed by STEP 7 rounded to 15 decimal points.

4.6.7 Data types for durations

Data type S5TIME

A tag with data type S5TIME is used for the duration of a SIMATIC timer function. It occupies a 16-bit word with 1+3 decades (Fig. 4.18).



Data type S5TIME

Duration = Time value x Time scale

15 ...			... 0
10^0	10^2	10^1	10^0
Time scale	Time value		

Time scale:		
	0 0 0 0	10 ms
	0 0 0 1	100 ms
	0 0 1 0	1 s
	0 0 1 1	10 s

Data type TIME

Number of milliseconds (duration)

31	30 16	15 ...	... 0
V	2^{30}... ... 2^{24} / 2^{23}... ... 2^{16}	2^{15}... ... 2^8	2^7... ... 2^0

V = sign: 0 = positive
1 = negative

Data type LTIME

Number of nanoseconds (duration)

63	62 48		... 0
V	2^{62}... ... 2^{56} / 2^{55}... ... 2^{48}	• • •	2^7... ... 2^0

V = sign: 0 = positive
1 = negative

Fig. 4.18 Bit assignment of data types S5TIME, TIME, and LTIME

The time duration is displayed in hours, minutes, seconds, and milliseconds. Conversion into the internal representation is handled by STEP 7. The internal representation is a BCD number from 000 to 999. The time frame can adopt the following values: 10 ms (0000), 100 ms (0001), 1 s (0010), and 10 s (0011). The duration is the product of the time frame and time value. Depending on the time scale, different limits result for the time value:

Time scale	10 ms	100 ms	1 s	10 s
Time value	S5T#10 ms to S5T#9s990ms	S5T#100ms to S5T#1m39s900ms	S5T#1s to S5T#16m39s	S5T#10s to S5T#2h46m30s

Examples: S5TIME#500ms (= W#16#0050)
S5T#2h46m30s (= W#16#3999)

Data type TIME

A tag with data type TIME (duration) occupies a doubleword. The representation contains the data for days (d), hours (h), minutes (m), seconds (s) and milliseconds (ms), whereby individual time units can be omitted. If only one time unit is specified, a decimal representation is possible. 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. 4.19).

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
T#–24d20h31m23s648ms to T#24d20h31m23s647ms.

Examples:	TIME#2h30m	(= DW#16#0089_5440)
	T#2.25h	(= DW#16#007B_98A0)

Data type LTIME

A tag with data type LTIME (duration) occupies a long word. The representation contains the data for days (d), hours (h), minutes (m), seconds (s), milliseconds (ms), microseconds (us), and nanoseconds. Individual time units can be omitted. If only one time unit is specified, a decimal representation is possible. If more than one time unit is specified, the values for the time units are limited: Days from 0 to 106 751, hours from 0 to 23, minutes and seconds from 0 to 59, and milliseconds, microseconds and nanoseconds from 0 to 999 (Fig. 4.19).

The content of the tag is interpreted as nanoseconds (ns) and saved as a 64-bit fixed-point number with sign. The range of values extends from
LT#–106751d23h47m16s854ms775us808ns to
LT#+106751d23h47m16s854ms775us807ns

Examples:	LTIME#2h20s	(= LW#16#0000_0691_0989_0800)
	LT#15.25h	(= LW#16#0000_31EE_66FF_8800)

4.6.8 Data types for points in time

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. 4.19). The range of values extends from D#1990-01-01 to D#2168-12-31.

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

TIME_OF_DAY (TOD)

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 o'clock) as an unsigned fixed-point number. The representation contains the data for hours, minutes, and sec-

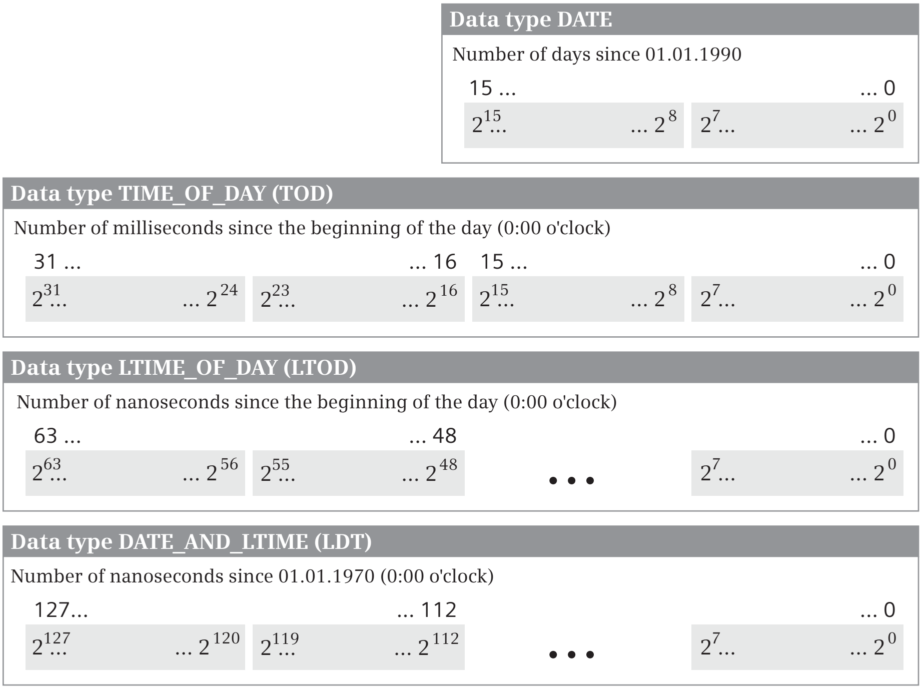


Fig. 4.19 Bit assignment of data types DATE, TOD, LTOD, and LDT

onds, each separated by a colon. The specification of milliseconds, which follows the seconds and is separated by a dot, can be omitted (Fig. 4.19). The range of values extends from TOD#00:00:00.000 to TOD#23:59.59.999.

Examples:	TIME_OF_DAY#00:00:00	(= DW#16#0000_0000)
	TOD#23:59:59.999	(= DW#16#0526_5BFF)

LTIME_OF_DAY (LTOD)

A tag with data type LTIME_OF_DAY occupies a long word. It contains the number of nanoseconds since the beginning of the day (0:00 o'clock) 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, microseconds and nanoseconds, which follows the seconds and is separated by a dot, can be omitted (Fig. 4.19). The range of values extends from
LTOD#00:00:00.000_000_000 to LTOD#23:59:59.999_999_999.

Examples:	LTOD#12:05:00	(= LW#16#0000_2790_220C_3800)
	LTOD#23:59:59.999_999_999	(= LW#16#0000_4E94_914E_FFFF)

DATE_AND_LTIME (LDT)

A tag with data type DATE_AND_LTIME occupies a long word. It contains the number of nanoseconds since 01.01.1970 (0:00 o'clock) as an unsigned fixed-point number.

The representation contains the year, month and day, each separated by a hyphen. After a colon come the hours, minutes and seconds, each separated by a colon. The specification of milliseconds, microseconds and nanoseconds, which follows the seconds and is separated by a dot, can be omitted (Fig. 4.19). The range of values extends from LDT#1970-01-01-0:0:0.000_000_000 to LDT#2262-04-11-23:47:16.854_775_807.

Examples:

LDT#2012-07-23-11:55:00	(= LW#16#12A3_73DB_640A_C800)
LDT#2262-04-11-23:47:16.854_775_807	(= LW#16#7FFF_FFFF_FFFF_FFFF)

4.7 Structured data types

Structured data types consist of a combination of elementary data types under one name (Table 4.8). These data types can only be used locally in the interface of code blocks and in data blocks; they are not approved for the operand areas Inputs (I), Outputs (Q), and Bit memories (M) in the PLC tag table.

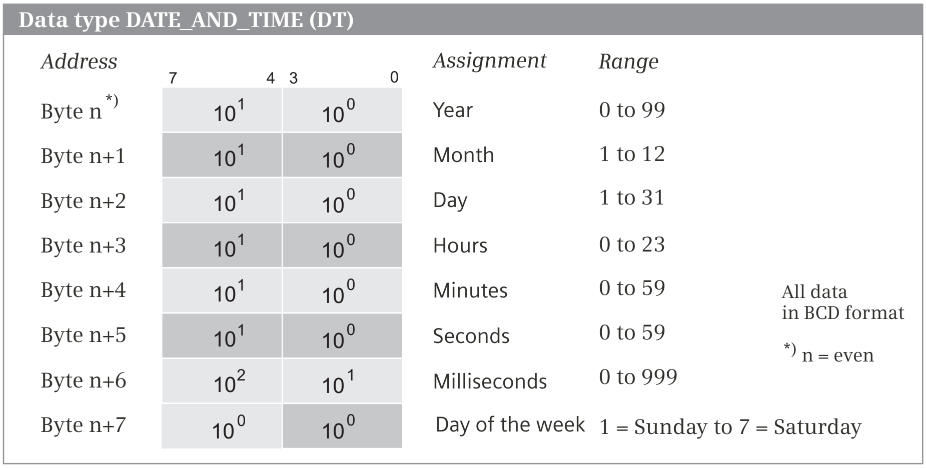
Table 4.8 Overview of structured data types

Data type	Length	Meaning, remark
DATE_AND_TIME	8 bytes	Date and time (accuracy: milliseconds) Example: DT#1990-01-01-00:00:00
DATE_AND_LTIME	16 bytes	Date and time (accuracy: nanoseconds) Example: DTL#1970-01-01-00:00:00.000_000_000
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 *Setpoint* 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 *Valve* has the data type STRUCT. It can then contain the components: Valve.Switch_on; Valve.Switch_off; Valve.Fault; etc.

4.7.1 Date and time DATE_AND_TIME (DT)

The data type DATE_AND_TIME (DT) represents a specific point in time consisting of the date and time, with the accuracy of one millisecond. The representation contains the year, month and day, each separated by a hyphen. After another hyphen come the 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.

A tag with data type DATE_AND_TIME occupies 8 bytes. Saving in the memory commences at a byte with even address. All values are present in BCD format (Fig. 4.20). The range of values extends from DT#1990-01-01-00:00:00.000 to DT#2089-12-31-23:59:59.999.



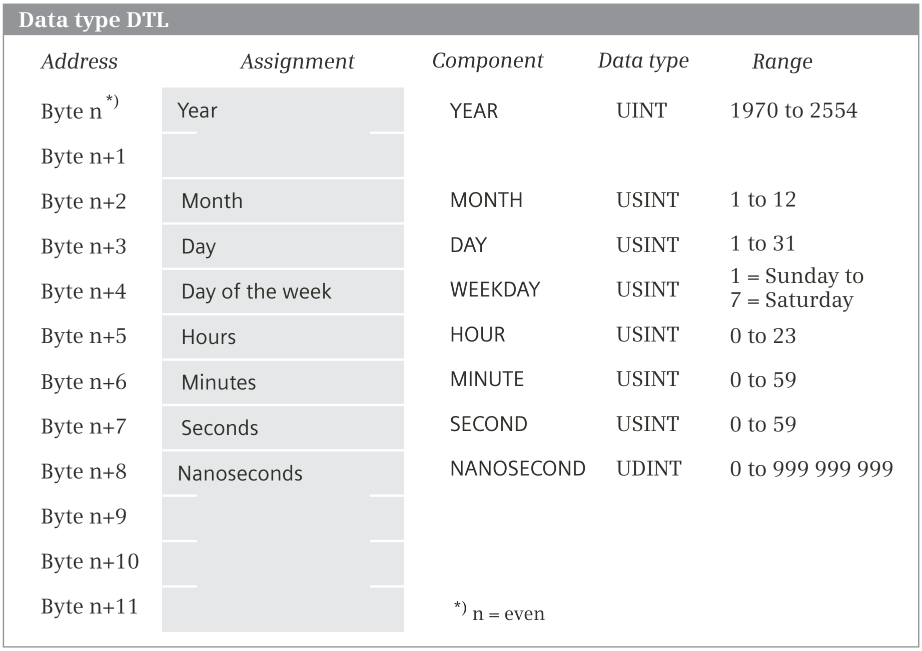
Data type DATE_AND_TIME (DT)

Address	7 4	3 0	Assignment	Range
Byte n *)	10^1	10^0	Year	0 to 99
Byte n+1	10^1	10^0	Month	1 to 12
Byte n+2	10^1	10^0	Day	1 to 31
Byte n+3	10^1	10^0	Hours	0 to 23
Byte n+4	10^1	10^0	Minutes	0 to 59
Byte n+5	10^1	10^0	Seconds	0 to 59
Byte n+6	10^2	10^1	Milliseconds	0 to 999
Byte n+7	10^0	10^0	Day of the week	1 = Sunday to 7 = Saturday

All data in BCD format

*) n = even

Fig. 4.20 Structure of data type DATE_AND_TIME (DT)



Data type DTL

Address	Assignment	Component	Data type	Range
Byte n *)	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	Day of the week	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				

*) n = even

Fig. 4.21 Structure of data type DATE_AND_LTIME (DTL)

4.7.2 Date and time DATE_AND_LTIME (DTL)

The data type DATE_AND_LTIME (DTL) represents a specific point in time consisting of the date and time, with the accuracy of one nanosecond. The representation contains the year, month and day, each separated by a hyphen. After another hyphen come the hours, minutes and seconds, each separated by a colon. The specification of nanoseconds, which follows the seconds and is separated by a dot, can be omitted.

A tag with data type DTL occupies 12 bytes. Saving in the memory commences at a byte with even address. The values are available in the form of an unsigned fixed-point number (Fig. 4.21). The range of values extends from DTL#1970-01-01-00:00:00.000_000_000 to DTL#2554-12-31-23:59:59.999_999_999.

Each component of a tag in DTL format can also be addressed individually. If a DTL tag has the name *#Start_time*, the hour can be addressed with *#Start_time.HOUR* and the minutes can be addressed with *#Start_time.MINUTE*. Both components have the data type USINT.

4.7.3 STRING data type

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

If a STRING tag is saved as a value, the maximum length can be defined in square brackets when the tag is declared. This corresponds to the maximum number of characters in ASCII code. If the length specification is omitted, the standard length of 254 characters is defined. When saved as a pointer (block parameter for an FC block, in/out parameter for a function block), only the standard length of 254 characters is accepted.

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. 4.22).

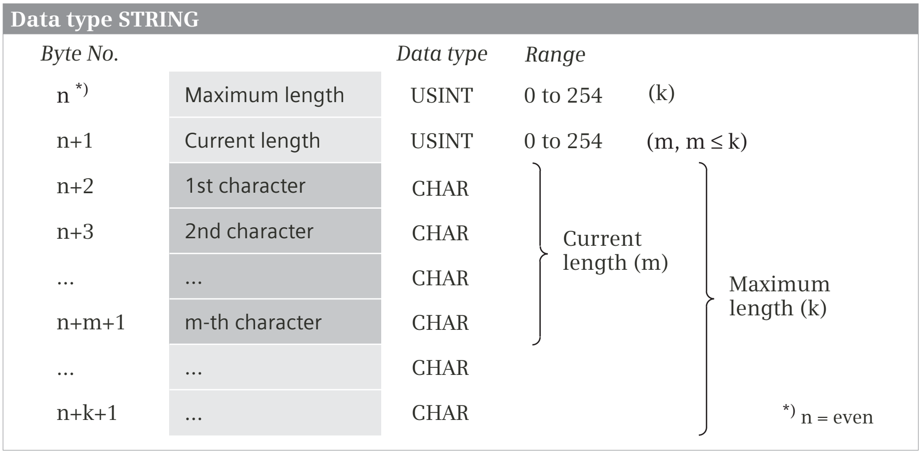
Example: The tag *Machine* is to be defined with a maximum length of 12 characters and should have 'Drill' as the default setting.

```
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.

A constant with data type STRING is written with single quotation marks, for example 'Hans Berger'. Special characters are entered with a preceding dollar sign. Fig. 4.13 on page 115 shows a selection.

A STRING tag cannot be assigned a default value when declared in the temporary local data. In order to use STRING tags in the temporary local data for meaningful



Data type STRING

Byte No.		*Data type*	*Range*		
n [*)]	Maximum length	USINT	0 to 254	(k)	
n+1	Current length	USINT	0 to 254	(m, m ≤ k)	
n+2	1st character	CHAR			
n+3	2nd character	CHAR	Current length (m)		
…	…	CHAR		Maximum length (k)	
n+m+1	m-th character	CHAR			
…	…	CHAR			
n+k+1	…	CHAR			[*)] n = even

Fig. 4.22 Structure of STRING data type

purposes, they must be written before being read. For blocks with standard access, the contents of the range lengths and characters are quasi-random before they are written for the first time. For blocks with the *Optimized block access* attribute activated, the range lengths have plausible values and the characters have the value '$00'.

The characters in a STRING tag can also be addressed individually (not with SCL). 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 example above, the tag *Machine[3]* has the character 'h'. The index can also be a tag in fixed-point format.

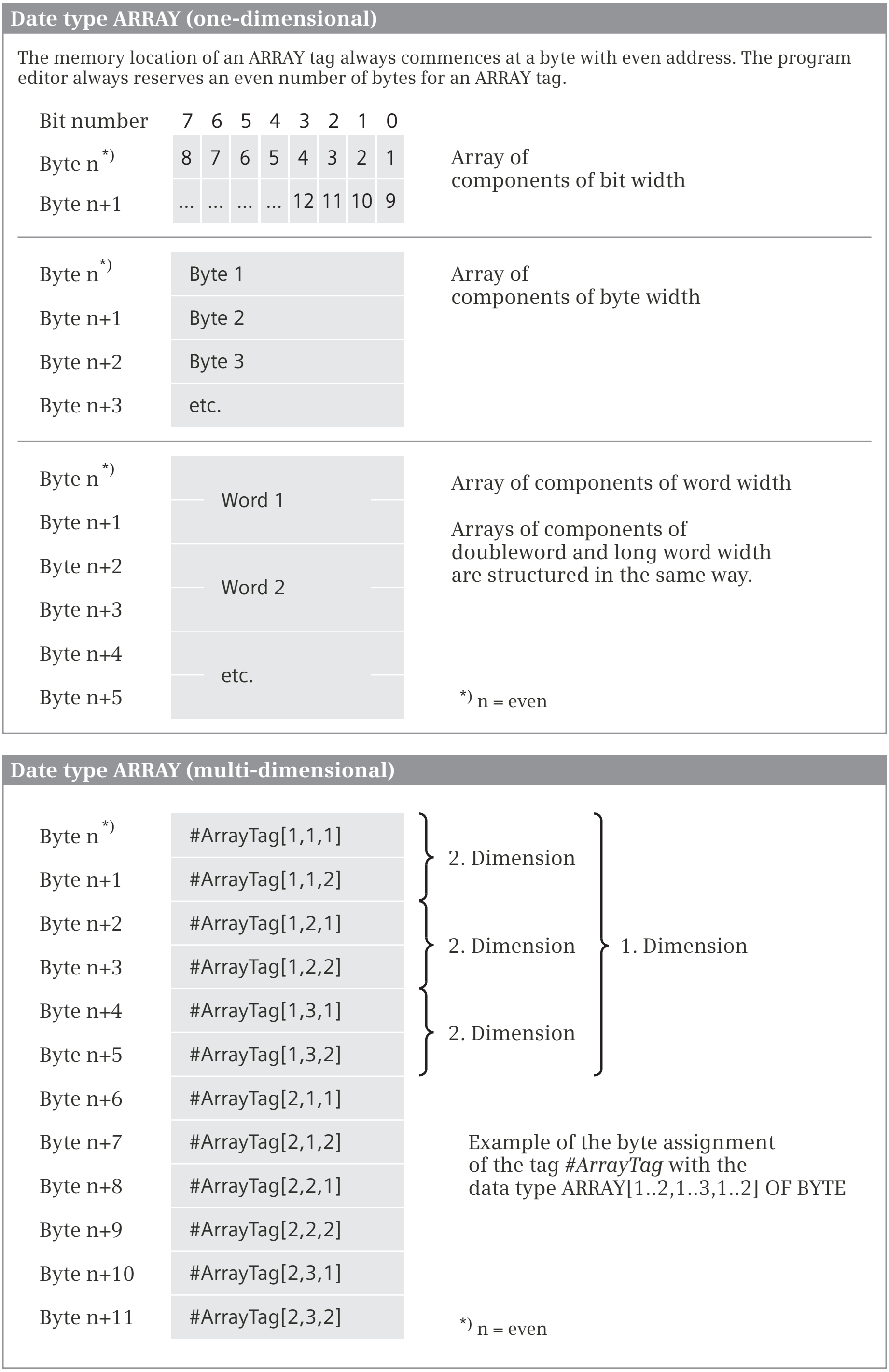
Special functions are available for processing STRING tags, for example to separate a partial string or to combine two STRING tags into a single one (see Chapter 13.9 "Processing of strings (data type STRING)" on page 615).

4.7.4 Data type ARRAY

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

A tag with data type ARRAY commences 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 an ARRAY tag (Fig. 4.23).

When creating an ARRAY tag, the number range of the components is specified in square brackets, and the data type following the keyword OF. Example: A tag with



Date type ARRAY (one-dimensional)

The memory location of an ARRAY tag always commences at a byte with even address. The program editor always reserves an even number of bytes for an ARRAY tag.

Bit number	7	6	5	4	3	2	1	0
Byte n *)	8	7	6	5	4	3	2	1
Byte n+1	...	...	...	...	12	11	10	9

Array of components of bit width

Byte n *)	Byte 1
Byte n+1	Byte 2
Byte n+2	Byte 3
Byte n+3	etc.

Array of components of byte width

Byte n *)	Word 1
Byte n+1	
Byte n+2	Word 2
Byte n+3	
Byte n+4	etc.
Byte n+5	

Array of components of word width

Arrays of components of doubleword and long word width are structured in the same way.

*) n = even

Date type ARRAY (multi-dimensional)

Byte n *)	#ArrayTag[1,1,1]
Byte n+1	#ArrayTag[1,1,2]
Byte n+2	#ArrayTag[1,2,1]
Byte n+3	#ArrayTag[1,2,2]
Byte n+4	#ArrayTag[1,3,1]
Byte n+5	#ArrayTag[1,3,2]
Byte n+6	#ArrayTag[2,1,1]
Byte n+7	#ArrayTag[2,1,2]
Byte n+8	#ArrayTag[2,2,1]
Byte n+9	#ArrayTag[2,2,2]
Byte n+10	#ArrayTag[2,3,1]
Byte n+11	#ArrayTag[2,3,2]

Example of the byte assignment of the tag *#ArrayTag* with the data type ARRAY[1..2,1..3,1..2] OF BYTE

*) n = even

Fig. 4.23 Structure of data type ARRAY

the name *Measured value* is to have 16 components of data type INT, which are numbered commencing with 1.

```
Measured value : ARRAY[1..16] OF INT
```

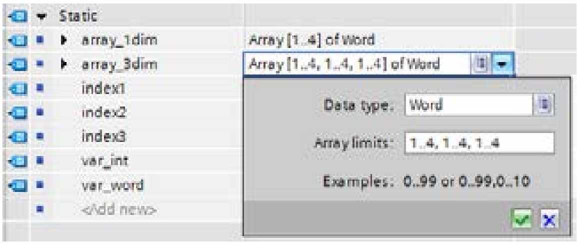
The number range extends from –2 147 483 648 to 2 147 483 647. The lower range value must be smaller than the upper value. The maximum number of components depends on the data type of a component and on the memory space available in the block in which the ARRAY tag is created.

The components of an ARRAY tag can 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.

During addressing, the index can also be a tag with a fixed-point data type and thus allow indirect addressing which is only defined during runtime. Further details are described in Chapter 4.3.2 "Indirect addressing of ARRAY components" on page 100.

Multi-dimensional arrays

ARRAY tags can have up to 6 dimensions. The same applies as to one-dimensional arrays. The dimension areas are written in the declaration in square brackets, each separated by a comma. Fig. 4.24 shows an example of the declaration of a three-dimensional array.



Static	
array_1dim	Array [1..4] of Word
array_3dim	Array [1..4, 1..4, 1..4] of Word
index1	
index2	
index3	
var_int	
var_word	
<Add new>	

Data type: Word

Array limits: 1..4, 1..4, 1..4

Examples: 0..99 or 0..99,0..10

Fig. 4.24 Example of the declaration of an array tag

In the multi-dimensional arrays, the components are saved starting with the first dimension. With bit and byte components, a new dimension always commences in the next byte, with components of other data types always in the next word (in the next byte with even address, see Fig. 4.23).

When addressing an array component, you can specify a constant or a tag with a fixed-point data type for the index. For SCL, it is also permissible to specify an expression with a fixed-point data type. Example:

```
#var_word := #Array_3dim[12, #index, 2*#index];
```

Addressing of partial arrays is also possible: With multi-dimensional arrays, you can handle the partial arrays like correspondingly dimensioned tags: You omit array indices starting from the right, and obtain a partial area of the original array with a smaller dimension. An example is shown in Chapter 4.3.2 “Indirect addressing of ARRAY components” on page 100.

4.7.5 Data type STRUCT

The STRUCT data type represents a data structure comprising a fixed number of components with different data types. All data types are permissible for the components.

A tag with STRUCT data type commences at a byte with even address, followed by the components in the declared sequence. Components with the BOOL data type commence in the least significant bit of the next vacant byte, components with the BYTE or CHAR data type in the next vacant byte. Components with other data types commence at a byte with even address. The program editor reserves an even number of bytes for a STRUCT tag (Fig. 4.25).

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* (BOOL), *switch_off* (BOOL), *speed* (INT), and *delay* (TIME). Fig. 4.26 shows the declaration of the tags.

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* can be created on a block parameter with the INT data type.

Nested structures

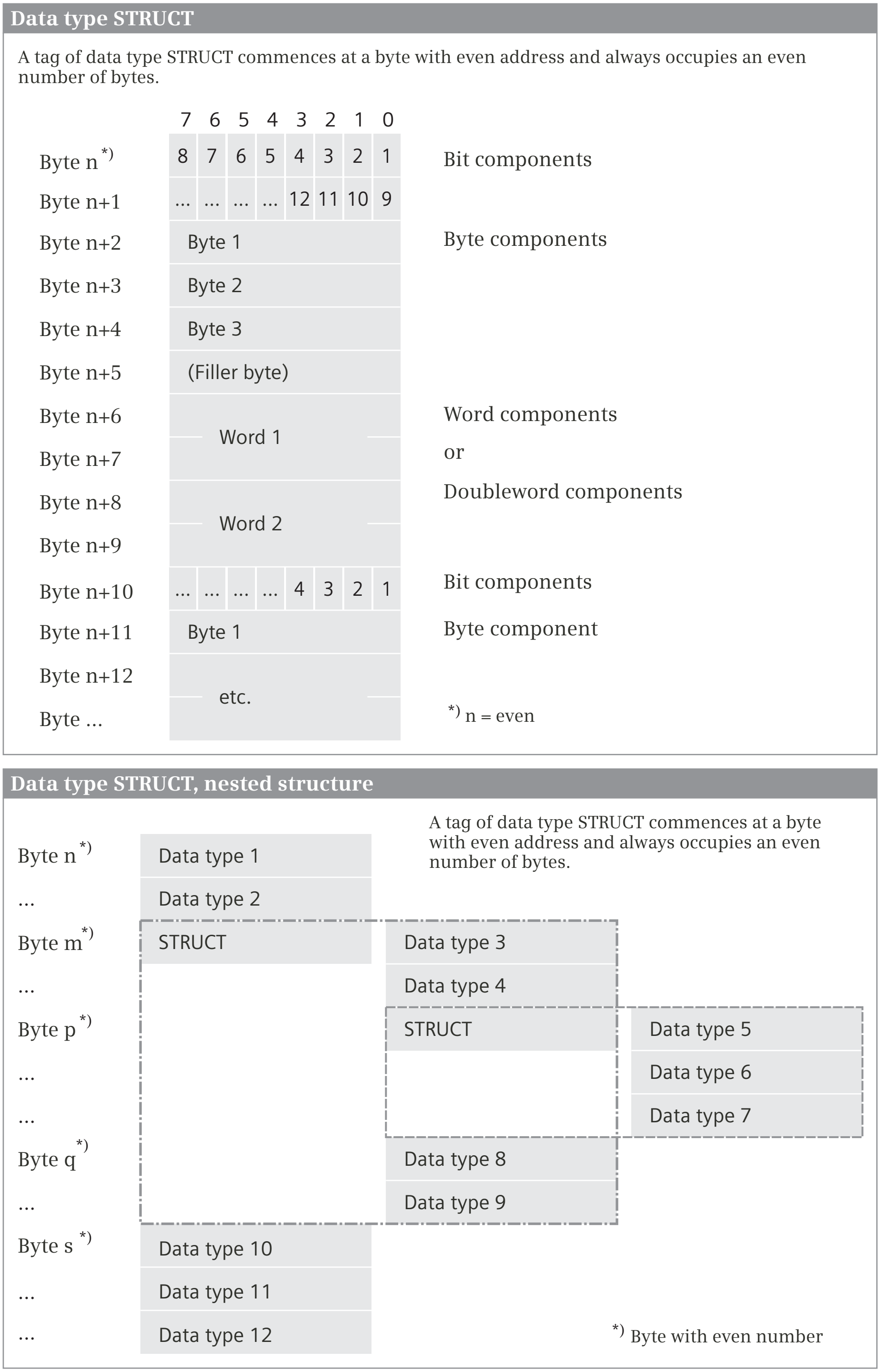
A nested structure contains at least one further structure as component. A nesting depth of up to 8 levels is possible. All components can be addressed individually. The individual names are each separated by a dot.

Example: *StructureTag.Structure_Level2.Component_Level2*.

4.8 Parameter types

4.8.1 Overview

The parameter types are additional data types for block parameters. In addition to the data types shown in Table 4.9, there are the PLC data types, the system data types, and the hardware data types, which – with certain restrictions – can also be used in the block interface.



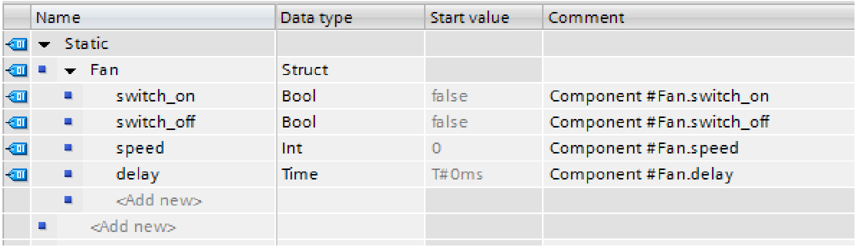
Data type STRUCT

A tag of data type STRUCT commences at a byte with even address and always occupies an even number of bytes.

Data type STRUCT, nested structure

A tag of data type STRUCT commences at a byte with even address and always occupies an even number of bytes.

Fig. 4.25 Structure of STRUCT data type



Name	Data type	Start value	Comment
▾ Static			
▾ Fan	Struct		
switch_on	Bool	false	Component #Fan.switch_on
switch_off	Bool	false	Component #Fan.switch_off
speed	Int	0	Component #Fan.speed
delay	Time	T#0ms	Component #Fan.delay
<Add new>			
<Add new>			

Fig. 4.26 Example of the declaration of a tag with STRUCT data type

Table 4.9 Overview of parameter types

Parameter type	Description	Examples of actual parameters
TIMER	SIMATIC timer function	%T15 or name
COUNTER	SIMATIC counter function	%C16 or name
*Function*_xTIME	IEC timer function With the function TP, TON, TOF, or TONR and different time value lengths	The actual parameter is a > Local instance: #Instance name > Single instance: "Data block name"
*Function*_xCOUNTER	IEC counter function with the function CTU, CTD or CTUD and different count value lengths	The actual parameter is a > Local instance: #Instance name > Single instance: "Data block name"
BLOCK_FC	Function	%FC17 or name (FC without block parameter!)
BLOCK_FB	Function block	%FB18 or name (FB without block parameter!)
DB_ANY	Data block	%DB19 or name or UINT tag
VOID	No function value (without data type)	Without actual parameter (only with functions FC)
POINTER	DB pointer	As pointer: P#M10.0 or P#DB20.DBX22.2 As operand: %MW20 or %I1.0 or #Name
ANY	ANY pointer	As area: P#DB10.DBX0.0 WORD 20 or any (complete) tag
VARIANT	VARIANT pointer	As area: P#DB10.DBX0.0 WORD 20 or any (complete) tag or type data block

4.8.2 TIMER and COUNTER parameter types

The SIMATIC timer and counter functions transferred at the block interface are of parameter types TIMER and COUNTER. These types of block parameter can only be declared in the declaration section *Input*. The content of the block parameter is the number of the transferred timer and counter operands.

TIMER and COUNTER are also used in the PLC tag table as data types for SIMATIC timer and counter functions.

4.8.3 Parameter types for IEC timer functions

The data types in Table 4.10 are available for the transfer of IEC timer functions to the block interface. The structure of the data types corresponds to the structure of the system data type IEC_TIMER (see Chapter 4.11.1 “System data types for IEC timer functions” on page 139).

Table 4.10 Parameter types for IEC timer functions

Timer function	Parameter type with TIME duration	Parameter type with LTIME duration
Pulse generation	TP_TIME	TP_LTIME
ON delay	TON_TIME	TON_LTIME
OFF delay	TOF_TIME	TOF_LTIME
Accumulating ON delay	TONR_TIME	TONR_LTIME

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.

4.8.4 Parameter types for IEC counter functions

Depending on the counter type and the data type of the count value, there are the data types shown in the table for the transfer of IEC counter functions to the block interface. The structure of the data types corresponds to the structure of the system data type IEC_xCOUNTER (see Chapter 4.11.2 “System data types for IEC counter functions” on page 140).

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.

4.8.5 Parameter types BLOCK_FC and BLOCK_FB (STL)

Code blocks FC and FB, which are programmed using STL, can be transferred to a block with the STL program via block parameters. With the data type BLOCK_FB, a function block can be transferred via a block parameter. With the data type

Table 4.11 Parameter types for IEC counter functions

Counter function	Parameter type	Data type of the count value	Parameter type	Data type of the count value
Up counter	CTU_SINT CTU_INT CTU_DINT CTU_LINT	SINT INT DINT LINT	CTU_USINT CTU_UINT CTU_UDINT CTU_ULINT	USINT UINT UDINT ULINT
Down counter	CTD_SINT CTD_INT CTD_DINT CTD_LINT	SINT INT DINT LINT	CTD_USINT CTD_UINT CTD_UDINT CTD_ULINT	USINT UINT UDINT ULINT
Up/down counter	CTUD_SINT CTUD_INT CTUD_DINT CTUD_LINT	SINT INT DINT LINT	CTUD_USINT CTUD_UINT CTUD_UDINT CTUD_ULINT	USINT UINT UDINT ULINT

BLOCK_FC, a function (FC) can be transferred. An FC block that is transferred in this way must not have any block parameters. A function block must not have any instance data blocks and thus no block parameters and no static local data. The operations UC and CC that are used for the call are described in Chapter 10.6.2 “Block call function in the statement list” on page 438.

A block parameter with the BLOCK_FB or BLOCK_FC parameter type can only be declared in the declaration section *Input*. The content of the block parameter is the number of the transferred block.

4.8.6 Parameter type DB_ANY

A data block can be transferred to the called block via a block parameter with the data type ANY_DB. The actual parameter can be the absolute address of a data block (e.g. %DB10), the symbolic address of a data block (e.g. “Station data”), or a tag with the data type UINT.

A block parameter with the DB_ANY parameter type can only be declared in the declaration section *Input*. The content of the block parameter is the number of the transferred block.

A block parameter with the data type DB_ANY cannot be the instance data block of a function block or of a system block.

A data tag in this data block can be addressed in the program of the called block with *#BlockParameterName.%DataOperand*. In this way, a data block whose number is only known during runtime can be addressed. Further details can be found in Chapter 4.3.4 “Indirect addressing of a data block” on page 102.

4.8.7 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. Additional information on the function value can be found in section “Using a function value of a function (FC)” on page 167.

4.8.8 Parameter types POINTER, ANY, and VARIANT

POINTER parameter type

A tag with elementary data type is transferred at a block parameter of the type POINTER. Such a block parameter can be declared in the declaration sections *Input* and *InOut*, and with functions (FC) also in the subsection *Output*. The content of the block parameter is a DB pointer which points to the actual parameter to be transferred (see Chapter 4.9 "Pointer" on page 134).

ANY parameter type

A tag with any data type or a data area is transferred at a block parameter of the type ANY. Such a block parameter can be declared in the declaration sections *Input* and *InOut*, and with functions (FC) also in the subsection *Output*. The content of the block parameter is an ANY pointer which points to the actual parameter to be transferred (see Chapter 4.9 "Pointer" on page 134).

Parameter type VARIANT

A block parameter with data type VARIANT contains a pointer to a tag or a data area. VARIANT can be used in the declaration sections *Input*, *InOut*, and *Output*. Actual parameters of all data types are approved for a block parameter of type VARIANT. The actual parameter can be an absolutely or symbolically addressed tag, an operand area that is absolutely addressed with an ANY pointer, or a type data block.

You can "pass on" a block parameter with the parameter type VARIANT to a block parameter of a called block that also has the parameter type VARIANT.

4.9 Pointer

4.9.1 Introduction

A pointer is a reference to a tag, an operand, or an operand area. It is structured in such a way that it contains the bit address, the byte address, the operand ID if applicable, the area length, and the data type. STEP 7 knows the following types of pointers:

- Area pointers; these have a length of 32 bits and contain an address and possibly the operand ID.
- DB pointers; these have a length of 48 bits and contain the number of the data block in addition to the area pointer.
- ANY pointers; these have a length of 80 bits and contain further data such as the data type of the operand in addition to the DB pointer.

For STL, the area pointer is used for indirect addressing. The DB pointer is used as actual parameter for a block parameter with the data type POINTER or VARIANT, and the ANY pointer is used as actual parameter for a block parameter with the data type ANY or VARIANT.

4.9.2 Area pointer

An area pointer is used for the indirect addressing for STL.

The area pointer contains the operand address and possibly also the operand area. Without an operand area, it is an *area-internal* pointer. If the pointer also contains the operand area, one refers to a *cross-area* pointer. The two types of pointer are distinguished by the assignment of bit 31 (Fig. 4.27).

You can load an area pointer as a constant into accumulator 1 or into one of the address registers. The notation for this is as follows:

P#y.x for an area-internal pointer (e.g. P#22.0) and

P#Zy.x for a cross-area pointer (e.g. P#M22.0)

where x = bit address, y = byte address, and Z = area. Specify the operand ID as the area (I, Q, M, DBX, DIX, L, and P). The operand area I/O cannot be reached using a pointer.

The area pointer always has a bit address which also always has to be specified for digital operands; the bit address is 0 (zero) for digital operands. You can use the area pointer P#M22.0, for example, to address the memory bit M 22.0, but also the memory byte MB 22, the memory word MW 22, or the memory doubleword MD 22.

4.9.3 DB pointer

A DB pointer is used for transferring a tag to a block parameter or function parameter.

A DB pointer also contains, supplementary to the area pointer, a data block number as UINT number. It specifies the data block if the area pointer contains the operand areas Global data (DBX) or Instance data (DIX). In all other cases, zero is present instead of the data block number (Fig. 4.27).

You have already become acquainted with the pointer's notation in the complete addressing of data operands. The data block and the data operand are also specified here separated by a dot: P#Data_block.Data_operand.

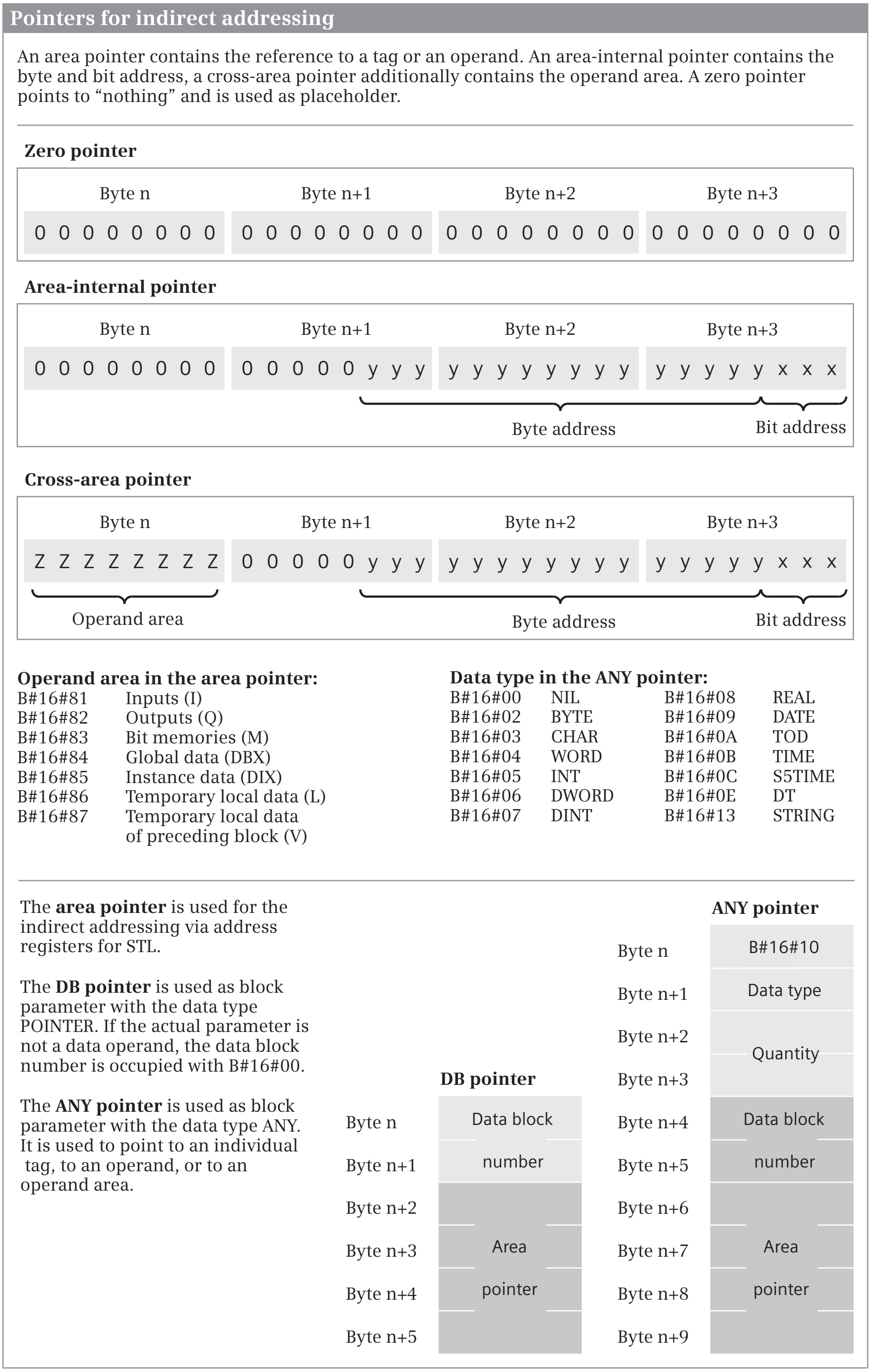
Examples:

▷ P#DB10.DBX20.5	Data bit 20.5 in data block 10
▷ P#DB102.DBD250.0	Data doubleword 250 in data block 102

4.9.4 ANY pointer

The ANY pointer is used for transferring a tag or an operand area to a block parameter or function parameter.

Supplementary to the DB pointer, the ANY pointer also contains the data type and a repetition factor. It is thus possible to additionally point to an (absolutely addressed) operand area. The representation of a constant is:
P#[Data_block.]Operand Type Quantity.



Pointers for indirect addressing

An area pointer contains the reference to a tag or an operand. An area-internal pointer contains the byte and bit address, a cross-area pointer additionally contains the operand area. A zero pointer points to “nothing” and is used as placeholder.

Zero pointer

Byte n	Byte n+1	Byte n+2	Byte n+3
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0

Area-internal pointer

Byte n	Byte n+1	Byte n+2	Byte n+3
0 0 0 0 0 0 0 0	0 0 0 0 0 y y y	y y y y y y y y	y y y y y x x x

Cross-area pointer

Byte n	Byte n+1	Byte n+2	Byte n+3
Z Z Z Z Z Z Z Z	0 0 0 0 0 y y y	y y y y y y y y	y y y y y x x x

Operand area in the area pointer:

B#16#81	Inputs (I)
B#16#82	Outputs (Q)
B#16#83	Bit memories (M)
B#16#84	Global data (DBX)
B#16#85	Instance data (DIX)
B#16#86	Temporary local data (L)
B#16#87	Temporary local data of preceding block (V)

Data type in the ANY pointer:

B#16#00	NIL	B#16#08	REAL
B#16#02	BYTE	B#16#09	DATE
B#16#03	CHAR	B#16#0A	TOD
B#16#04	WORD	B#16#0B	TIME
B#16#05	INT	B#16#0C	S5TIME
B#16#06	DWORD	B#16#0E	DT
B#16#07	DINT	B#16#13	STRING

The **area pointer** is used for the indirect addressing via address registers for STL.

The **DB pointer** is used as block parameter with the data type POINTER. If the actual parameter is not a data operand, the data block number is occupied with B#16#00.

The **ANY pointer** is used as block parameter with the data type ANY. It is used to point to an individual tag, to an operand, or to an operand area.

DB pointer

Byte n	Data block
Byte n+1	number
Byte n+2	
Byte n+3	Area
Byte n+4	pointer
Byte n+5	

ANY pointer

Byte n	B#16#10
Byte n+1	Data type
Byte n+2	Quantity
Byte n+3	
Byte n+4	Data block
Byte n+5	number
Byte n+6	
Byte n+7	Area
Byte n+8	pointer
Byte n+9	

Fig. 4.27 Structure of the pointers

Examples:

▷ P#DB11.DBX30.0 INT 12	Area with 12 words in the %DB11 starting at %DBB30
▷ P#M16.0 BYTE 8	Area with 8 bytes starting at %MB16
▷ P#I18.0 WORD 1	Input word %IW18
▷ P#I1.5 BOOL 1	Input %I1.5

The program editor then applies an ANY pointer which agrees with the data in the representation of the constant with regard to type and quantity. Note that the operand address in the ANY pointer must always be a bit address.

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

You can create a “zero pointer” with P#P0.0 VOID 0.

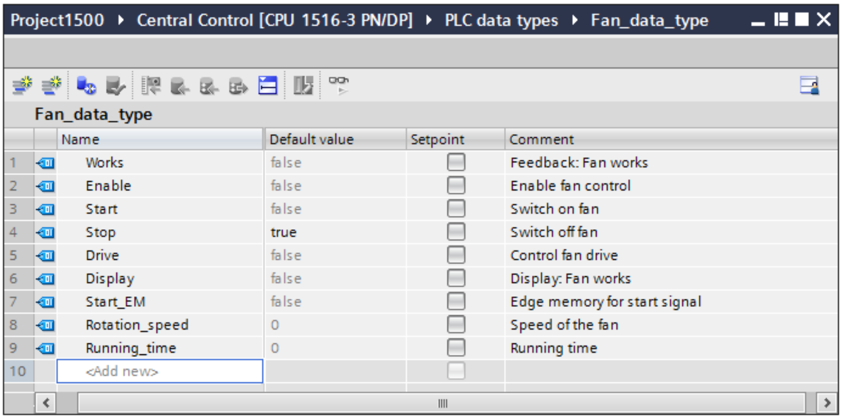
The ANY_pointer is a constant which permanently points to a tag or an operand area. How to create a “variable” ANY pointer is described in Chapter 4.3.5 “Indirect addressing with an ANY pointer” on page 103.

4.10 PLC data types

A PLC data type is one with its own name. It is structured like the STRUCT data type, i.e. it consists of individual components which can have 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).

4.10.1 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*



Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ PLC data types ▸ Fan_data_type

Fan_data_type

	Name	Default value	Setpoint	Comment
1	Works	false	□	Feedback: Fan works
2	Enable	false	□	Enable fan control
3	Start	false	□	Switch on fan
4	Stop	true	□	Switch off fan
5	Drive	false	□	Control fan drive
6	Display	false	□	Display: Fan works
7	Start_EM	false	□	Edge memory for start signal
8	Rotation_speed	0	□	Speed of the fan
9	Running_time	0	□	Running time
10	<Add new>		□	

Fig. 4.28 Example of programming a PLC 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. 4.28).

You can change the standard name *User_data_type_n*, where *n* is the 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 user constant, or a block. The operand ID is UDT (user-defined data type), the number is assigned by the program editor.

4.10.2 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 code block. The default setting for the PLC data type can be changed. You then address the individual components of the tag using *#tag_name.comp_name*.

You can also assign a PLC data type to an input or an output in a PLC tag table. Then, however, the PLC data type cannot have any components with the data type STRING. Specify the bit 0 in the lowest byte as the absolute address for the operand area. Example: %I64.0.

You can nest PLC data types. A (different) PLC data type can be used in 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.

4.10.3 Comparing PLC data types

The PLC data types of the opened PLC station can be compared to the PLC data types in the CPU (offline/online comparison) or to the PLC data types in a different station from the same project, from a reference project, or from a library (offline/offline comparison). To perform the comparison, select the PLC station in the project tree and choose the command *Compare* > *Offline/online* or *Compare* > *Offline/offline*. An online connection to the CPU is required for the offline/online comparison.

This starts the compare editor, which shows the PLC station with the contained objects on the left side, including the PLC data types. For the offline/online comparison, the user program objects from the CPU are displayed on the right side. For an offline/offline comparison, use the mouse to move a PLC station from the same project, from a reference project, or from a library into the title bar on the right side of the compare editor. You can move other PLC stations into the title bar on one of the two sides at any time in order to carry out further comparisons.

The “Status and action area” is located between the two tables. Above this is the switchover button with the scale. In the automatic comparison (the switchover button with the scale is white), the PLC data types are automatically assigned on the left and right side based on their names and the comparison icons are displayed in the center. Activate the manual comparison by clicking on the switchover button. The switchover button is now gray. Manually assign the PLC data types to be compared by selecting them using the mouse. The result of the comparison is displayed in the bottom area of the comparison window in the “Property comparison”. The lower area can be opened and closed using the arrow buttons.

For a detailed comparison, select a PLC data type and click on the *Start detailed comparison* icon. The compared PLC data types are displayed next to each other. The columns *Status* and *Action* are located between the lists. You can select the desired action from a drop-down list.

A filled green circle means that the objects are identical. A blue-orange semicircle (offline/online comparison) or a blue-gray semicircle (offline/offline comparison) indicates that the objects differ. If one half of the circle is not filled, the corresponding object is missing. An exclamation mark in a gray circle indicates an object with differences in the identified folder.

In the *Action* column, you can select an action from a drop-down list for different objects, for example copying with an arrow in the direction in which you are copying. Clicking on the *Execute actions* icon starts the set actions. Note that you can neither add, delete, nor overwrite objects in reference projects.

4.11 System data types

System data types (SDT) are pre-defined data types which, like the data type STRUCT, consist of a fixed number of components which can have different elementary data types each. System data types are provided with STEP 7 and cannot be changed. The system data types can only be used together with certain functions or statements.

4.11.1 System data types for IEC timer functions

For the instance data of an IEC timer function, a CPU 1500 has two system data types: IEC_TIMER for durations with the data type TIME, and IEC_LTIMER for durations with the data type LTIME.

If you use one of the statements TP, TON, TOF or TONR, 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_TIMER or IEC_LTIMER. You can also create a type data block or a local instance with the data type IEC_TIMER or IEC_LTIMER yourself. IEC_TIMER or IEC_LTIMER consists of the components shown in Table 4.12.

Table 4.12 Structure of the system data types IEC_TIMER and IEC_LTIMER

Name	Designation	Data type for IEC_TIMER	Data type for IEC_LTIMER
ST	(internal)	TIME	LTIME
PT	Preset time	TIME	LTIME
ET	Elapsed time	TIME	LTIME
RU	(internal)	BOOL	BOOL
IN	Start input	BOOL	BOOL
Q	Timer status	BOOL	BOOL

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*.

4.11.2 System data types for IEC counter functions

For the instance data of an IEC counter function, a CPU 1500 has eight system data types – depending on the data type of the count value (Table 4.13).

Table 4.13 System data types for IEC counter functions

IEC data type	Data type of the count value	IEC data type	Data type of the count value
IEC_SCOUNTER IEC_COUNTER IEC_DCOUNTER IEC_LCOUNTER	SINT INT DINT LINT	IEC_USCOUNTER IEC_UCOUNTER IEC_UDCOUNTER IEC_ULCOUNTER	USINT UINT UDINT ULINT

Table 4.14 Structure of the system data types IEC_xCounter

Name	Designation	Data type
CU	Up counter input (count up)	BOOL
CD	Down counter input (count down)	BOOL
R	Reset input	BOOL
LD	Load input	BOOL
QU	Status up	BOOL
QD	Status down	BOOL
PV	Preset value	SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT *)
CV	Count value	SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT *)

*) dependent on the system data type (IEC_SCOUNTER, IEC_COUNTER, IEC_DCOUNTER, etc.)

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 4.14.

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*.

4.11.3 Data type ERROR_STRUCT

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 an error that occurred is output with this structure (Table 4.15). 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 4.16). 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*.

The assignment of the ERROR_ID and handling of the error evaluation is described in Chapter 5.8.2 "Local error handling" on page 213.

Table 4.15 Structure of ErrorStruct data type

Name	Data type	Note, assignment
ERROR_ID	WORD	Error ID (see text)
FLAGS	BYTE	16#00
REACTION	BYTE	Reaction 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	
BLOCK_TYPE	BYTE	Type of block in which the error occurred 16#01: OB, 16#02: FC, 16#03: FB
CODE_BLOCK_NUMBER	UINT	Number of block in which the error occurred
OFFSET	UDINT	Internal memory address at which the error occurred
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	
AREA	BYTE	Addressed memory area on occurrence of error (C) (see text)
DB_NUMBER	UINT	Number of data block on occurrence of error, otherwise zero (D) (see text)
OFFSET	UDINT	Bit offset on occurrence of error (E) (see text)

Table 4.16 Information output depending on access type MODE

MODE	(A)	(B)	(C)	(D)	(E)
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

Memory area	Assignment of AREA component
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 memories (M)	16#83
Data operands (DB)	16#84, 16#85, 16#8A, 16#8B

4.11.4 Start information

If the attribute *Optimized block access* is deactivated for an organization block, the operating system of the CPU transfers start information in the temporary local data when the organization block is called. The start information can only be directly scanned in the program of the organization block. The system block RD_SINFO also permits access to the start information from the blocks called in the organization block.

The program editor automatically configures the start information when adding an organization block to the user program. The tag names and comments in English can be adapted according to your requirements.

This start information is 20 bytes long for every organization block and practically identical. The “standard structure” of the start information shown in Table 4.17 can be found as a basic framework in all organization blocks. Individual tags can have different names and different data types for some organization blocks. If the additional information and the data ID contain relevant information, this is specified in the description of the individual organization blocks.

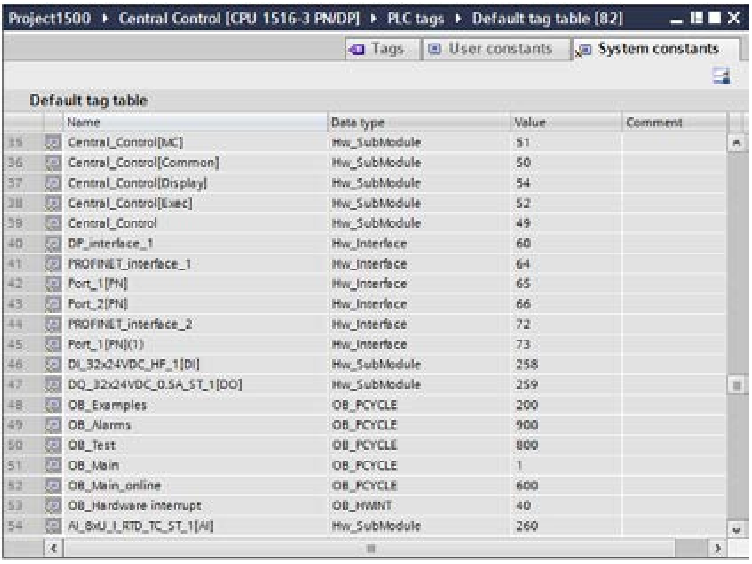
For an organization block with the *Optimized block access* attribute activated, the CPU operating system transfers any existing start information in the *Input* declaration section. This start information is described for the corresponding organization blocks.

Table 4.17 Structure of the start information

Byte	Data type	Tags	Meaning, remark
0	BYTE	EV_CLASS	Bits 0 to 3: Event identifier Bits 4 to 7: Event class
1	BYTE	EV_NUM STRT_INF	Event number
2	BYTE	PRIORITY	Priority class, number of execution level
3	BYTE	NUM	OB number B#16#FF for an OB number >254
4	BYTE	TYP2_3	Data ID 2_3: identifies the information entered in ZI2_3
5	BYTE	TYP_1	Data ID 1: identifies the information entered in ZI1
6 … 7	WORD	ZI1	Additional information 1
8 … 11	DWORD	ZI2_3	Additional information 2_3
12 … 19	DATE_AND_TIME	DATE_TIME	Beginning of event

4.12 Hardware data types

Hardware data types refer to all data types which can accept the hardware identifiers in the default tag table in the *System constants* tab. A hardware object is addressed in the program with a hardware identifier. The data type and the value are predefined, the name can be changed in the object properties (see also Chapter 4.4 “Addressing of hardware objects” on page 107). Fig. 4.29 shows the *System constants* tab with a selection of hardware data types.


Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ PLC tags ▸ Default tag table [82]
Tags
User constants
System constants

Default tag table

	Name	Data type	Value	Comment
35	Central_Control[MC]	Hw_SubModule	51	
36	Central_Control[Common]	Hw_SubModule	50	
37	Central_Control[Display]	Hw_SubModule	54	
38	Central_Control[Exec]	Hw_SubModule	52	
39	Central_Control	Hw_SubModule	49	
40	DP_interface_1	Hw_Interface	60	
41	PROFINET_interface_1	Hw_Interface	64	
42	Port_1[PN]	Hw_Interface	65	
43	Port_2[PN]	Hw_Interface	66	
44	PROFINET_interface_2	Hw_Interface	72	
45	Port_1[PN](1)	Hw_Interface	73	
46	DI_32x24VDC_HF_1[DI]	Hw_SubModule	258	
47	DQ_32x24VDC_0.5A_ST_1[DO]	Hw_SubModule	259	
48	OB_Examples	OB_PCYCLE	200	
49	OB_Alarms	OB_PCYCLE	900	
50	OB_Test	OB_PCYCLE	800	
51	OB_Main	OB_PCYCLE	1	
52	OB_Main_online	OB_PCYCLE	600	
53	OB_Hardware interrupt	OB_HWINT	40	
54	AI_8xU_I_RTD_TC_ST_1[AI]	Hw_SubModule	260	

Fig. 4.29 Examples of hardware data types

5 Program execution

5.1 Operating states of the CPU

A CPU 1500 recognizes the following operating states in which it is ready for operation:

- STOP, when the user program is not being executed
- STARTUP, when the startup program is being executed
- RUN, when the main program and the interrupt routine are being executed

If the CPU is not ready to operate, there is either no power supply or the CPU is defective.

Fig. 5.1 illustrates the operating state transitions:

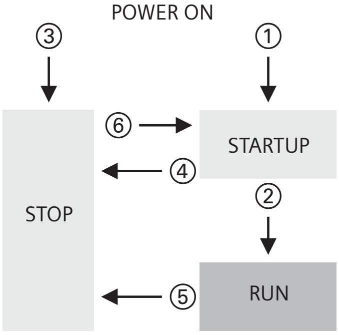
① After it is switched on, the CPU switches over to the STARTUP operating state if *Warm restart* is set as the startup type and the online project data is consistent.

② After a successful startup, the CPU switches to the RUN operating state.

③ If *No startup* is set as the startup type, the CPU switches to the STOP operating state after being switched on.

④ ⑤ The CPU switches to the STOP operating state if a “serious” error occurs during STARTUP or RUN, the system block STP is being executed, or the CPU is stopped by the operator.

Operating states of a CPU 1500



After it is switched on, the CPU – if the corresponding conditions are fulfilled – switches to the STARTUP operating state ① and then to RUN ② .
The CPU switches to STOP if the startup conditions are not fulfilled after switching on ③, or if a “serious” error occurs during STARTUP ④ or RUN ⑤ .
The CPU can switch from the STOP operating state via STARTUP ⑥ to RUN ②.

Fig. 5.1 Operating states and operating state transitions of a CPU 1500

⑥ The CPU switches from STOP to the STARTUP operating state if it is started by the operator and the online project data is consistent.

Using the mode switch on the CPU, you can activate the operating states STOP and RUN. The operating states are displayed on the CPU display and are indicated by the RUN/STOP LED: In the STOP operating state, the LED displays a continuous yellow light. In the RUN operating state, the LED is green.

On the programming device, you can control the operating states in online mode using the online tools and you can display them in various ways, e.g. with the online tools or in the inspector window in the tab *Diagnostics* > *Device information*.

5.1.1 STOP operating state

The STOP operating state is reached

- ▷ after the power supply is switched on at the CPU and *No restart* is configured as the startup type,
- ▷ after changing the mode switch from RUN to STOP,
- ▷ if a "serious" error occurs during program execution,
- ▷ if the STP system block is executed in the user program, and
- ▷ following a stop request from the programming device.

The CPU enters the cause of the STOP operating state into the diagnostics buffer. In this operating state 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 the STOP operating state. 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 operating state, the CPU can execute passive one-way communication functions if, for example, data is requested or sent by another station via S7 communication. The real-time clock continues to run in the STOP operating state.

You can parameterize the CPU in the STOP operating state, 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 operating state (OD or BASP signal, output disable or command output disable). Disabled output modules output a zero signal or – if configured accordingly – a parameterized substitute value.

5.1.2 STARTUP operating state

The STARTUP is executed when the CPU changes from the STOP operating state to the RUN operating state. In the STARTUP operating state, the CPU initializes itself and the modules controlled by it.

In the STARTUP operating state, the CPU updates the SIMATIC timer functions, the clock memories, the runtime meters and the real-time clock, and executes the user program in a startup organization block.

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.

A CPU 1500 carries out a warm restart when started up (Fig. 5.2).

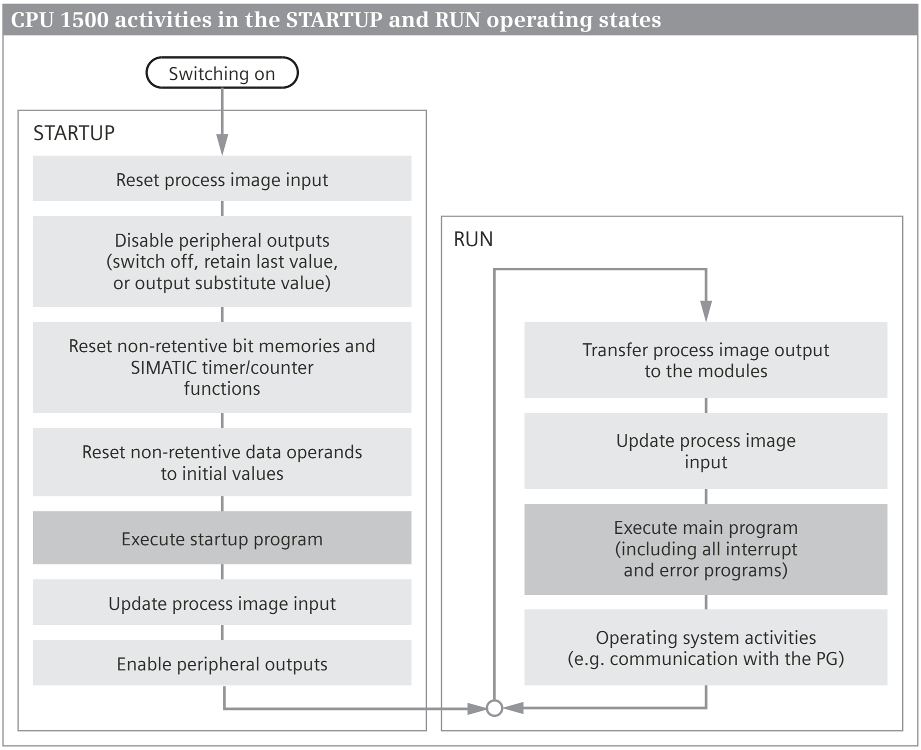

CPU 1500 activities in the STARTUP and RUN operating states
Switching on
STARTUP
Reset process image input
Disable peripheral outputs (switch off, retain last value, or output substitute value)
Reset non-retentive bit memories and SIMATIC timer/counter functions
Reset non-retentive data operands to initial values
Execute startup program
Update process image input
Enable peripheral outputs
RUN
Transfer process image output to the modules
Update process image input
Execute main program (including all interrupt and error programs)
Operating system activities (e.g. communication with the PG)

Fig. 5.2 CPU activities in the STARTUP and RUN operating states

Warm restart startup type

A *manual warm restart* is triggered

▷ by the mode switch on the CPU on a transition from STOP to RUN, or

- ▷ by operator input on the programming device; the mode switch must be at RUN for this.

A manual warm restart can always be triggered, except if the CPU requests a memory reset.

An *automatic warm restart* is triggered by switching on the power supply if

- ▷ the CPU was not at STOP when the voltage was switched off, the mode switch is set to RUN, and the startup type *Warm restart* was parameterized in the CPU properties under *Startup after POWER ON*, or
- ▷ the CPU was interrupted during a warm restart by a power failure.

With a warm restart, the CPU deletes the process image input and initializes the process image output and the peripheral outputs, i.e. the outputs and the peripheral outputs are switched off, retain their last value, or output a parameterized substitute value depending on the parameterization. This is followed by disabling of the peripheral outputs by the OD or BASP signal (output disable or command output disable).

The CPU deletes the non-retentive bit memories, SIMATIC timers, and SIMATIC counters, and sets the non-retentive data operands to the start values from the load memory. The values of the operands set as retentive are retained. The current program and the retentive data in the work memory are retained, as are the data blocks generated per system block.

Hardware compatibility

The modules are parameterized as was defined by the hardware configuration. If a module other than the configured module is plugged in, you can configure the startup behavior of the CPU:

In the module properties of the CPU, under *Startup* and *Comparison preset to actual configuration*, you can choose between *Start up CPU only if compatible* and *Startup CPU even if mismatch*.

In the properties of a signal module, under *Module parameters > General*, you can set the individual startup behavior in the *Comparison preset to actual module* field depending on the module: *From CPU*, *Start up CPU only if compatible*, or *Startup CPU even if mismatch*.

Startup program

If startup organization blocks are available, they are called one-time (see Chapter 5.5 “Startup program” on page 169). The peripheral inputs can be accessed directly during the startup program and the outputs and peripheral outputs can be controlled. However, the signal states at the output terminals are not yet changed because the peripheral outputs are still disabled.

The process image input is updated following execution of the startup program, and the process image output is transferred to the I/O. Disabling of the peripheral outputs is then canceled. Following a warm restart, execution of the main program always commences at the beginning.

5.1.3 RUN operating state

The RUN operating state is reached from STARTUP operating state. In the RUN operating state, the user program is executed and the PLC station controls the machine or process.

The following activities are executed cyclically by the CPU (see also Fig. 5.2 on page 146):

- ▷ Transmission of process image output to the output modules
- ▷ Updating of the process image input
- ▷ Execution of the main program, including interrupt and error programs

The main program is present in organization block OB 1 and in further organization blocks of the *Program cycle* event class. If further organization blocks are present for the main program, they are executed following the OB 1 in order of their numbering.

In the RUN operating state, 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 the RUN operating state can be found in Chapter 5.6 “Main program” on page 177 (including process images, cycle time, response time, time-of-day), in Chapter 5.7 “Interrupt processing” on page 192 (time-of-day, time-delay, cyclic and hardware interrupts), and in Chapter 5.8 “Error handling” on page 212 (including OB 82 *diagnostics interrupt* and 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 values of the set retentive tags are stored in the retentivity memory, which has a CPU-specific size. It can accommodate bit memories, SIMATIC timer functions, SIMATIC counter functions, and tags from data blocks.

Retentivity settings for bit memories and SIMATIC timer/counter functions

You set the retentive memory area for the bit memories and the SIMATIC timer/counter functions in the PLC tag table or in the assignment list. Click on the symbol for retentivity in the toolbar of the working window and enter the number of retentive memory bytes and the number of retentive timer and counter functions. The retentivity area always begins with the address zero. A memory tag occupying more than one byte must not exceed the limit between the retentive and non-retentive areas.

A retentive operand is marked with a checkmark in the *Retain* column of the PLC tag table. You can activate/deactivate the retentivity mark in the assignment list via an icon in the toolbar of the working window.

Retentivity settings for global data tags

If the *Optimized block access* attribute is activated in a 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 retentivity 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-retain
The tag in the instance data block is always non-retentive.

▷ Retain
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 bock.

The standard setting is “Non-retain”. 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 during the draft phase by adaptation to technological and functional conditions; this is important for program creation, testing, and startup. 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 can be divided into so-called networks (not with SCL), each of which has part of the block program. 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 the order of their numbering, but can also be bypassed depending on conditions. The program can be tested 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.

In Fig. 5.3, a simple example is used to compare linear program structures with modular program structures. 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 block.

Blocks can also be used repeatedly. Let us assume in the example 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 independently of one another.

Practice-oriented program organization

In the block at the highest position in the call hierarchy (in the main program), you should call the blocks located “underneath” in such a manner that you achieve rough structuring of your program. Program structuring is possible according to technological or functional aspects.

The following explanations can only present a rough and very general view which can provide a beginner with food for thought with regards to program structuring and ideas for realization of his control task. Advanced programmers usually have enough experience to allow them to find a program structure appropriate to the specific control task.

Technological program structuring is strongly based on the structure of the plant to be controlled. The individual parts of the plant or the process to be controlled correspond to the individual program sections. Subordinate to this rough

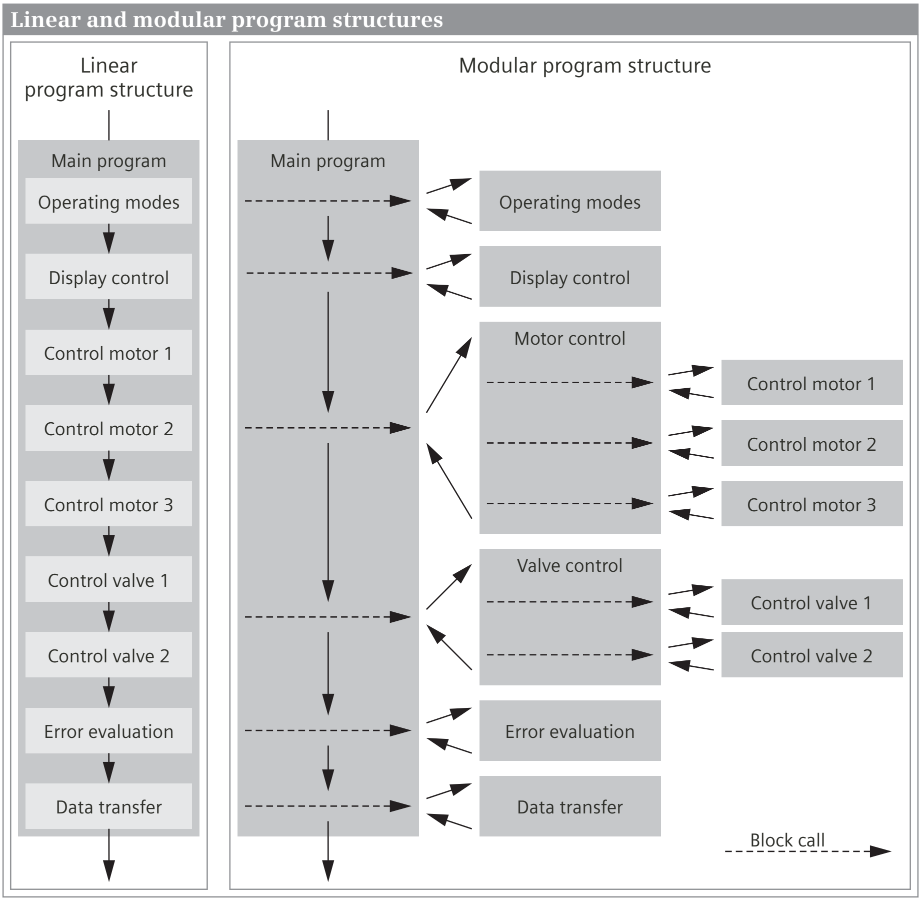


Fig. 5.3 Comparison between linear and modular program structures

structuring is the scanning of limit switches and HMI devices and the control of final controlling elements and display units (specific to each plant unit). Signal exchange between the individual plant units (or better: program sections) takes place by means of global tags.

Functional program structuring is based on the control function to be executed. This type of program structuring does not initially take into account the structure of the plant to be controlled. The division of the plant only becomes visible in the subordinate blocks if the control function achieved using the rough structuring is divided further.

In practice, mixed forms of the two structuring concepts are usually present. Fig. 5.4 shows an example: The *operating mode program* and the *data processing program* reflect a plant-independent division of functions. The program sections *Feed 1*, *Feed 2*, *Process*, and *Remove* base their technological structuring on the plant units to be controlled.

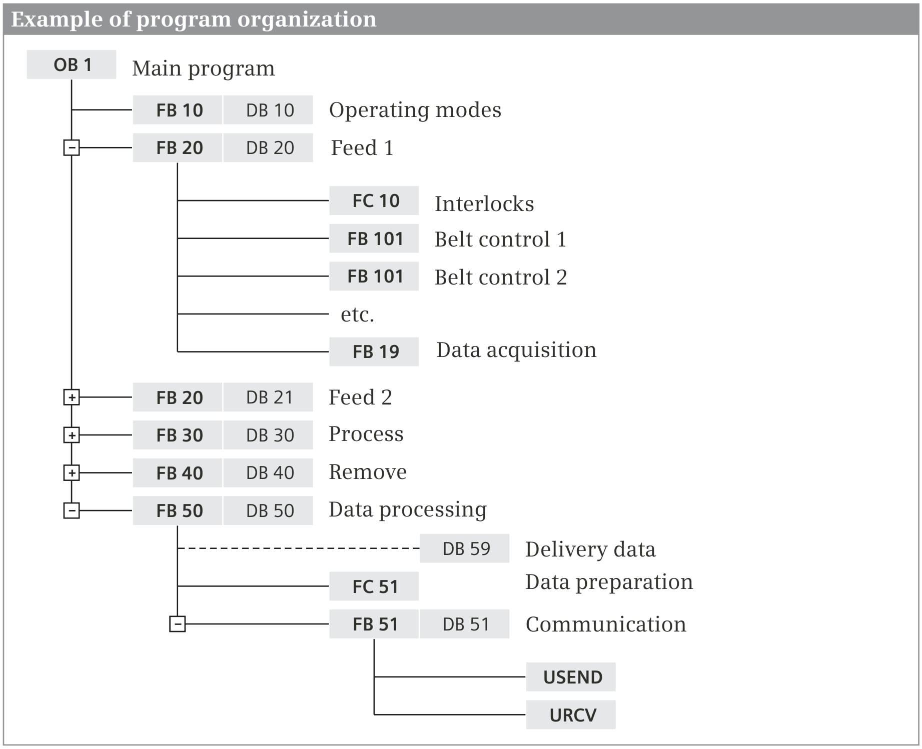


Fig. 5.4 Example of program organization

The example also shows the use of different types of block (further information on the types of block can be found in Chapter 5.3.1 “Block types” on page 155). The organization block OB 1 contains the main program; the blocks for the operating modes, for the individual plant units, and for data processing are called in it. These blocks are function blocks (FB) with an instance data block (DB) as the data memory. *Feed 1* and *Feed 2* have an identical structure; FB 20 is used to control a feeder unit, in the case of *Feed 1* with DB 20 as the instance data block, and in the case of *Feed 2* with DB 21.

In the *Feed controller*, the function FC 20 processes the interlocks; it scans inputs or bit memories, and controls the local data of FB 20. Function block FB 101 contains a conveyor belt control; it is called once per conveyor belt. The call is carried out as a local instance so that its local data is present in the instance data block DB 20. The same applies to the data acquisition FB 29.

Data processing FB 50 with DB 50 processes the data acquired with FB 29 (and other blocks) which is present in the global data block DB 60. Function FC 51 prepares this data for transmission. Transmission is controlled by FB 51 (with DB 51 as the instance data block), in which the USEND and URCV are called for communication with another station. The system blocks store their instance data in the “higher-level” DB 51 in this case as well.

Block nesting depth

A further block can be called within a block, and then another one in this, etc. The number of such “horizontal” call levels, the nesting depth, is limited. In Fig. 5.4, for example, block FB 20 is called in block OB 1 (nesting depth 1), and then block FC 20 in FB 20. This corresponds to a nesting depth of 3.

The maximum nesting depth is 24 per priority level for a CPU 1500. If more blocks are called in a “horizontal” level, the CPU will generate a program execution error.

Blocks which are called in succession (linear, “vertical”) do not generate a new call level and therefore do not affect the nesting depth.

5.2.2 Program execution

The complete program of a CPU comprises the operating system and the user program (control 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 FLASH 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 types

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. 5.5). 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.

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.

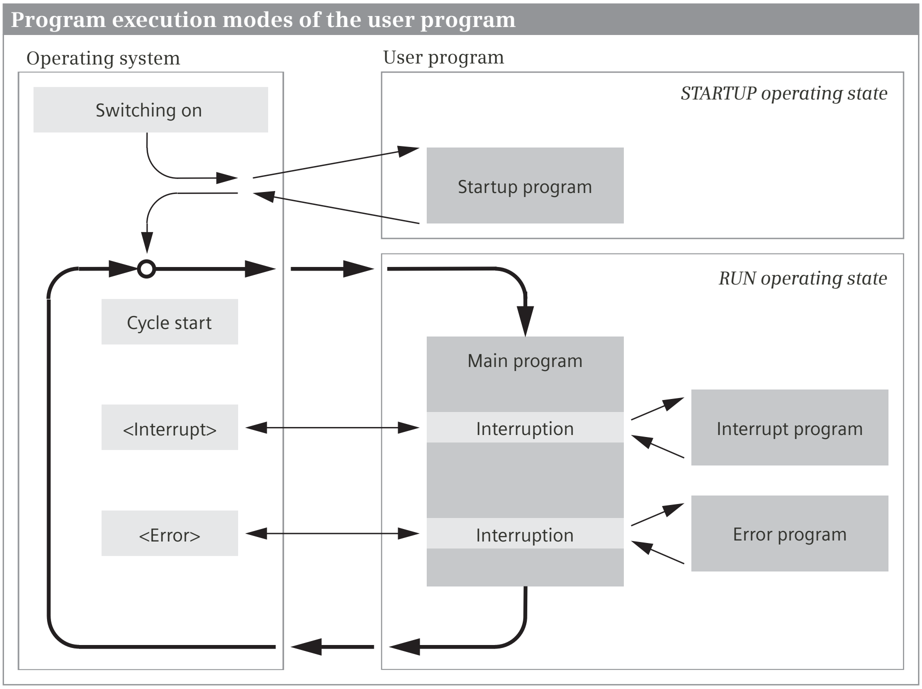


Fig. 5.5 Program execution modes of a SIMATIC user program

Startup program

Program execution commences in the CPU with the startup program in the STARTUP operating state, after switching on the power supply, for example. The startup program is optional. If you wish to create a startup program, use organization block OB 100 (*Startup*). You have the capability of assigning additional organizational bocks to the startup program. These are then processed in the order of their OB number after OB 100. Additional code blocks can be called up in a startup organization block. Following execution of the startup program, the CPU commences with execution of the main program.

Main program

The main program is present as standard in organization block OB 1 (*Program cycle*), which is always executed by the CPU. The first statement in OB 1 is identical to the program start of the main program. You have the capability of assigning additional organizational bocks to the main program. These are then processed in the order of their OB number after OB 1. Additional code blocks can be called up in a main program organization block. The main program is the totality of all of the cyclically processed 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 and the organization blocks assigned to the main program again.

Interrupt routine and error program

Events that can interrupt the main program are interrupts and errors. Interrupts have their origin in the plant to be controlled (hardware interrupt), in the CPU (time-of-day, time-delay and cyclic interrupts), or originate from the modules (diagnostics interrupt).

A distinction is made between asynchronous and synchronous errors. An asynchronous error is an error which is independent of program execution, for example a power supply failure in a station of the distributed I/O. A synchronous error is caused by the program execution, for example the addressing of a non-existent operand or an error during conversion of a data type.

5.3 Programming blocks

5.3.1 Block types

You can divide your program into individual sections as required. These program parts are called “blocks”. A block is a part of the user program that is defined by its function, structure or application. Each block should feature a technological or functional framework.

User 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 a hardware interrupt or cyclic interrupt. The main program is located in organization block OB 1 by default. There are organization blocks with a fixed number and a fixed assignment to an event and there are organization blocks with a freely selectable number and a freely selectable assignment to an event. When calling, the organization blocks make 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 – the instance data block. 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 global data block, instance data block, ARRAY data block, type data block, or CPU data block. With a global data block, you program the data tags directly in the data block. For an instance data block, the programming of the assigned function block defines which data tags the data block has. An ARRAY data block consists of tags which all have the same data type. A type data block has the structure of a PLC data type and a CPU data block is created during runtime in the user program with CREATE_DB.

The number of organization blocks and their block numbers are defined in part by the operating system. The block numbers of the other types of block can be assigned as desired within the permissible range. Note that the number range is larger than the number of permissible blocks. Blocks should preferably be symbolically addressed using a name.

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, for example manipulating the internal CPU clock or the communication 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 configuration data, for example module parameters. These blocks are created and managed by STEP 7 itself. As a rule, system data blocks are located in the load memory. You can only access the contents of system data blocks in special cases, such as when parameterizing modules with the aid of system blocks.

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, for example as extended statements or in the global libraries. 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 Block properties

Table 5.1 shows the modifiable block properties. The block attributes that are specially intended for blocks with the GRAPH program or for blocks in connection with the engineering tool Continuous Function Chart (CFC) are not included in the table. Additional properties can be set for some organization blocks, e.g. the phase offset for cyclic interrupt processing. These are described in the corresponding chapters.

Each block has other properties which cannot be set by the user and which provide information about the status of the block, for example the time of creation and compilation.

You configure the properties of a block using the program editor when the block is created, as described in Chapters 6.3.3 “Specifying code block properties” on page 256 and 6.4.3 “Defining properties for data blocks” on page 272. These chapters also contain a detailed description of the block attributes.

5.3.3 Block interface

The block interface contains the declarations of the local tags that are used solely within the block. These are the block parameters and the temporary and static local data. The block interface is shown as a table in the top part of the working window and contains – depending on the block type – the sections shown in Table 5.2.

Input parameters

An input parameter transfers a value to the program in the block and may only be read in the block program. 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 STL and 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. The data type TIMER can be used to transfer a SIMATIC timer function, and the data type COUNTER to transfer a SIMATIC counter function. Some organization blocks with the activated block attribute *Optimized block access* provide start information in the block interface as input parameters.

A block parameter with the data type DB_ANY transfers a data block to the called block. For blocks with the STL program, a block parameter with the data type BLOCK_FB transfers a function block and a block parameter with the data type

BLOCK FC transfers a function to the called block. The transferred code blocks must not have any block parameters themselves.

Table 5.1 Configurable block properties

Property	Block	Description
Block type, number, name	OB, FB, FC, DB	A block is unambiguously defined by the type and number (e.g. FB 10). A configurable ID (user-specific ID) can also be assigned to each block under "Name".
Constant name, event class, priority	OB	An organization block is assigned to a specific event class and is addressed in the user program with its constant name. For some organization blocks, the execution priority can be set.
Process image partition number	OB	Indicates the process image partition assigned to the organization block.
Language	OB, FB, FC	Setting of the programming language for the block program (LAD, FBD, SCL, STL, GRAPH).
Block information	OB, FB, FC, DB	The block information includes the block title (the "header"), the block comment, the author, the version, the block family, and the user-defined ID.
Protection	OB, FB, FC DB	A code block can be provided with access protection ("know-how protection") and copy protection. A data block can be provided with access protection ("know-how protection").
Download without reinitialization	FB, DB	Allows specific changes to the block interface during operation, without resetting the current values.
Block attributes	**Block**	**Description**
IEC check	OB, FB, FC	Defines how strictly the data type test is carried out.
Handle errors within block	OB, FB, FC	Defines how the operating system should treat a program execution error in the block.
Block can be used as a know-how protected library element	OB, FB, FC	Indicates whether the block can be used with know-how protection in a library.
Optimized block access	OB, FB, FC, DB	Defines the storage and access to the block-local tags. For organization blocks, this defines the type of the start information.
Only store in load memory	DB	The data block is not transferred to the work memory (for global, ARRAY and type data blocks) or is only generated in the load memory (for CPU data blocks).
Data block write-protected in the device	DB	The data block cannot be written from the user program (for global, ARRAY, type and CPU data blocks).
Set data in the standard area to retentive	Instance DB	Activates the retentivity setting for all tags which have the setting *Set in IDB* in the block interface (for instance data blocks).
Set ENO automatically	FB, FC (SCL)	Generates additional program code for error monitoring during the compilation (only for SCL).
Parameter passing via registers	FB, FC (STL)	Allows a special type of parameter assignment (only for STL).

Table 5.2 Declaration sections in the block interface

Section	Type, function, and data types	Included in
Input	Input parameters may only be read in the program of the block Elementary and structured data types, PLC, system and hardware data types, TIMER, COUNTER, DB_ANY, POINTER, ANY, VARIANT FB: STRING of adjustable length FC: STRING of standard length 254 Blocks with STL program: BLOCK_FC, BLOCK_FB	FC and FB
Output	Output parameters may only be written in the program of the block Elementary and structured data types, PLC data types, DB_ANY FB: STRING of adjustable length FC: STRING of standard length 254, POINTER, ANY, VARIANT	FC and FB
InOut	In/out parameters may be read and written in the program of the block Elementary and structured data types, PLC and system data types, STRING of standard length 254, DB_ANY, POINTER, ANY, VARIANT	FC and FB
Temp	Temporary local data may be read and written in the program of the block, are only valid during current block processing Elementary and structured data types, PLC data types, STRING of adjustable length, VARIANT OB, FC: DB_ANY Blocks with standard access: ANY	FC, FB and OB
Static	Static local data may be read and written in the program of the block, is saved in the instance data block and remains valid even following block processing Elementary and structured data types, PLC and system data types, STRING of adjustable length, DB_ANY	FB
Return	Function value may only be written in the program of the block, is an output parameter with the return value of a function Elementary data types, DTL, PLC data types, STRING of adjustable length, DB_ANY, POINTER, ANY (not with SCL), VOID	FC

Output parameters

An output parameter transfers a value to the calling block and may only be written in the block program. 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 STL and 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 in the case of 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 STL and 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 of an FC block is an output parameter which is handled in a special manner. It bears the name of the block with the default data type VOID (= no type). The function value is used with the SCL programming language. It is possible here to integrate FC blocks with function value in formulae (in expressions). The function value then corresponds to the value used for calculation in the formula. Any programming language can be used for the FC block in this case. An example is shown in the section “Using a function value of a function (FC)” on page 167.

For LAD, FBD and STL, you can ignore the function value if the data type VOID is set in the interface description. You can also assign a different data type to the function value, and this is then displayed as the first output parameter with the name *Ret_Val*. 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. It is used as intermediate memory for the block program 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 93.

The temporary local data is addressed symbolically. The local data can also be absolutely addressed only for STL (see Chapter 10.7.6 “Absolute addressing of temporary local data” on page 454).

Static local data

The static local data is stored in the instance data of the called function block – when called as a single instance in the assigned instance data block and when called as a local instance in the instance data of the calling function block. It can be read and written in the program of the called block. Static local data 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 addressed symbolically. The local data can also be absolutely addressed only for STL (see Chapter 10.7.5 “Partial addressing of data operands” on page 453).

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, e.g. using *“Data_block_name”.tag_name*.

5.3.4 Programming block parameters

By means of *block parameters* you enable parameterization of the processing specification (the block function) present in a block.

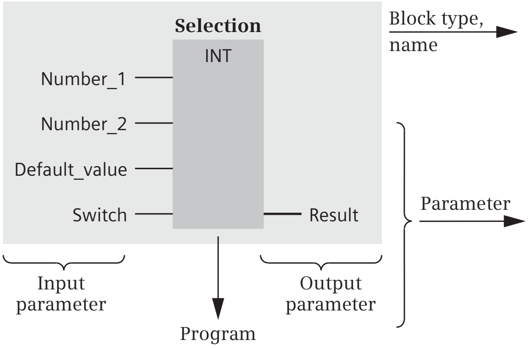
The example shows the programming of a block, which selects a value from three digital values according to the following criteria: The maximum value of two digital values *Number_1* and *Number_2* is searched for and, depending on the signal state of the binary tag *Switch*, this maximum or a substitute value will be output. The block is to be used multiple times in the user program with different tags. The tags are therefore transferred as block parameters – in our example, four input parameters and one output parameter. Since the selection logic need not permanently save values internally, a function FC is suitable as the block type (Fig. 5.6).

The values to be transferred are declared as input parameters in the *Input* section with name and data type, the selected value as an output parameter in the *Output* section, also with name and data type. An additional tag is required as an intermediate memory for the maximum value. This is declared in the *Temp* section, since its value is not required outside the block.

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.

After the “Selection” block is programmed, it can be called in the user program. Different values are transferred to “Selection” 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 formal parameters are replaced with the actual parameters. Section “Example of a block call” on page 163 shows how the selection block programmed here is called and supplied with current tags.

Programming block parameters



Block properties

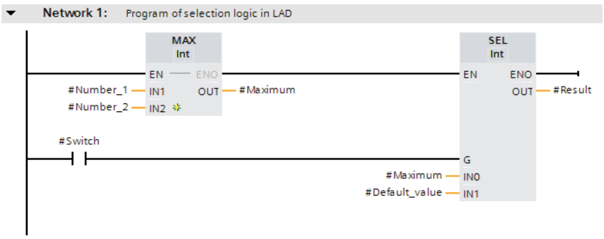
Name:	Selection
Type:	FC
Number:	311

Interface

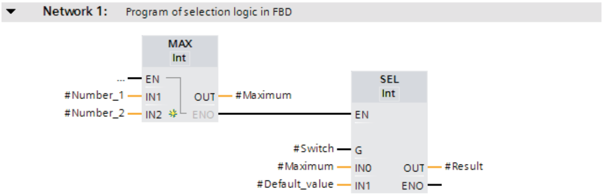
Declaration	Name	Data type
Input	Number_1	INT
	Number_2	INT
	Default_value	INT
	Switch	BOOL
Output	Result	INT
Temp	Maximum	INT

In the block program, the block parameters are called *formal parameters*. They are used like tags which have the same data type. The number symbol (#) in front of the name identifies the formal parameters and the other tags of the block interface as (block-)local tags.

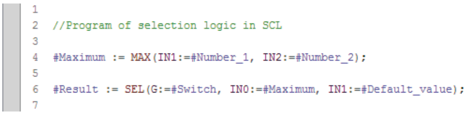
Block program in ladder logic



Block program in function block diagram

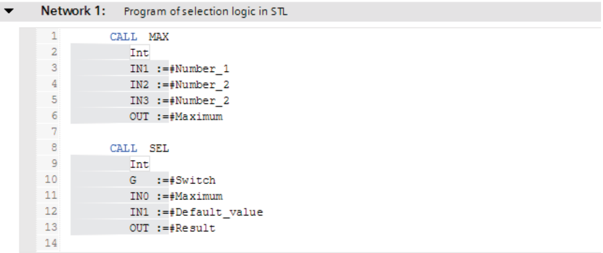


Block program in Structured Control Language



```

//Program of selection logic in SCL

#Maximum := MAX(IN1:=#Number_1, IN2:=#Number_2);

#Result := SEL(G:=#Switch, IN0:=#Maximum, IN1:=#Default_value);

```

Block program in statement list



Network 1: Program of selection logic in STL

```
      CALL  MAX
         Int
         IN1  :=#Number_1
         IN2  :=#Number_2
         IN3  :=#Number_2
         OUT  :=#Maximum

      CALL  SEL
         Int
         G    :=#Switch
         IN0  :=#Maximum
         IN1  :=#Default_value
         OUT  :=#Result

```

Fig. 5.6 Example of programming with block parameters

5.4 Calling blocks

5.4.1 General information on calling of code blocks

If blocks are to be processed, they must first be called in the program. The organization blocks which are started by the operating system when certain events occur are an exception.

With LAD and FBD, the call functions are boxes with an enable input EN and an enable output ENO. A conditional block call 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. With STL, this “EN/ENO mechanism” can be mapped using STL statements.

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 283.

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 14.2 “Calling of code blocks” on page 631.

Example of a block call

Chapter 5.3.4 “Programming block parameters” on page 161 shows how a block (an FC function) is programmed with a block parameter. You can now call the “*Selection”* 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. 5.7).

During runtime, the control processor replaces the formal parameters by the actual parameters. When calling the “*Selection”* block in the example, the maximum value of the “*Measurement 1”* and “*Measurement 2”* tags is selected and with signal state “0” of the “*Test mode”* tag, it is transferred to the “*Temperature”* tag. If “*Test mode”* has signal state “1”, the “*Test value”* tag is copied to the “*Temperature”* tag.

The “*Selection”* block can also be called multiple times in the user program, each time with a different parameter assignment. The existing program is then processed multiple times with various tags.

Example of calling a block

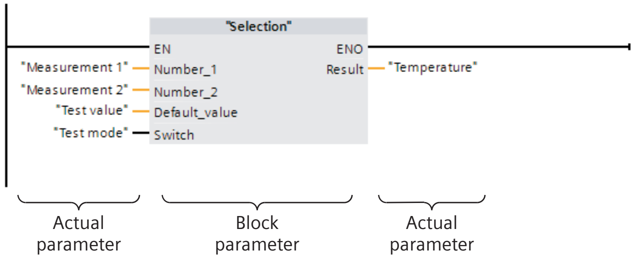
During the block call, the block parameters are supplied with tags with which the program in the block is to work. These tags are called *actual parameters*.

It can be called after programming the “Selection” block. To program the block call, open the block in which the “Selection” is to be called and drag the “Selection” block from the project tree into the working window. Then write the actual parameters to the block parameters.

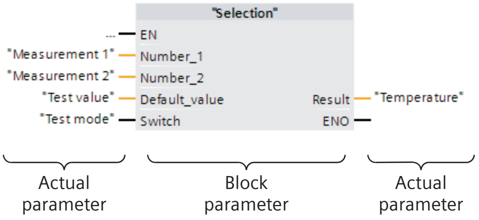
During runtime, the actual parameters (the current tags) are used instead of the formal parameters ("placeholders") used in the block.

The “Selection” block can also be called multiple times with different actual parameters. The same function is thus processed with different tags in each case.

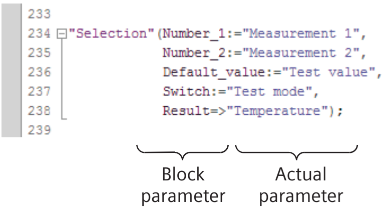
Block call in LAD



Block call in FBD

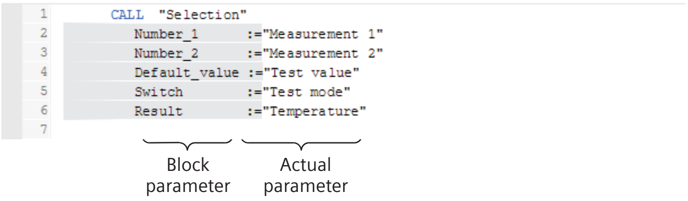


Block call in SCL



```
"Selection"(Number_1:="Measurement 1",
            Number_2:="Measurement 2",
            Default_value:="Test value",
            Switch:="Test mode",
            Result=>"Temperature");
```

Block call in STL



```
      CALL  "Selection"
         Number_1      :="Measurement 1"
         Number_2      :="Measurement 2"
         Default_value :="Test value"
         Switch        :="Test mode"
         Result        :="Temperature"
```

Fig. 5.7 Example of a block call with block parameters

5.4.2 Supplying the block parameters

The data type of the actual parameter must compatible with the data type of the block parameter. The data type test can be controlled in the calling block using the *IEC check* attribute. If the attribute is activated, the test is conducted using stricter criteria. If possible, the program editor uses implicit data type conversion.

Assigning elementary data types to block parameters

At block parameters with elementary data type, you can use tags from the Inputs, Outputs, Bit memories, Data, Temporary local data and Static local data operand areas. A data tag in a data block must be addressed completely with the (*"Data_block".Data_tag*) data block.

Constants and peripheral inputs are only permissible for input parameters, peripheral outputs only for output parameters.

At a block parameter with an elementary data type, you can also create a component of an array (ARRAY) or of a structure (STRUCT, DTL, PLC and system data type) if the data type of the component is compatible with the block parameter.

Assigning structured data types to block parameters

At block parameters with structured data type, you can use tags from the operand areas Data, Temporary local data and Static local data.

Input parameters with the data type DT, DTL or STRING can be provided with a constant. When calling a function block for an input and output parameter with the data type STRING, the maximum length of the actual parameter must match the maximum length of the block parameter, because the value of the actual parameter is saved in the instance data. In all of the other cases, the block parameter consists of a pointer, which points to the actual parameter so that the maximum length of the actual parameter is not defined (up to 254 characters).

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 plausible values before being used as an actual parameter (at least the current and maximum length). This is handled by the program editor for a block with the *Optimized block access* attribute activated.

For supplying a block parameter with the ARRAY data type, an actual parameter with exactly the same structure is allowed – the number and data type of the components must match. A partial array (part of a multidimensional array) can also be used as actual parameter.

For supplying a block parameter with one of the data types STRUCT, PLC data type or system data type, actual parameters with exactly the same structure are allowed – the arrangement and the data type of the components must match.

Assigning a parameter type to block parameters

On a block parameter with the data type TIMER, a SIMATIC timer function (T) is permitted as the actual parameter. On a block parameter with the data type COUNTER, a SIMATIC counter function (C) is permitted.

At a block parameter of the type IEC timer/counter function, an instance data block of an IEC timer/counter function, a local instance of an IEC timer/counter function, or an in/out parameter of the type IEC timer/counter function can be created.

At an (input) block parameter with the data type DB_ANY, a data block can be created as an actual parameter. With LAD, FBD and SCL, data tags in this transferred data block can be addressed in the program of the called block. With STL, this data block can be opened with the statement OPN via the DB register and with the statement OPNDI via the DI register, e.g. for the partial addressing of data operands.

At a block parameter with the data type BLOCK_FC or BLOCK_FB, a block (FC or FB) which does not have its own block parameters can be created. This block can be called with one of the STL statements UC or CC if the block attribute *Parameter passing via register* is activated.

On a block parameter with the parameter type POINTER, a tag with elementary data type or a pointer (e.g. P#DB10.DBX20.5) is permitted. The tag can also be a completely addressed data tag or a component of an array or of a data structure and must be located in a memory area with standard access. A zero pointer (a pointer to "nothing") is specified with P#0.0.

Tags of all data types are approved for a block parameter with the parameter type ANY. The tags which must be connected to the block parameters or which are meaningful are defined by the programming within the called block. You can also specify a constant with the format of the ANY pointer "P#[Data block.]Operand Data_type Quantity", and thus define an absolutely addressed area. Supplying with temporary local data of data type ANY is handled separately (see Chapter 4.3.5 "Indirect addressing with an ANY pointer" on page 103).

Tags of all data types are allowed on a block parameter with the parameter type VARIANT, including operand areas addressed with an ANY pointer. An entire data block can only be an actual parameter if 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.

5.4.3 Calling a function (FC)

For an FC block, the block parameters are pointers to the actual operands. Therefore all of the block parameters must be supplied with actual operands when calling a function.

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.

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 a different data type, it is handled like an output parameter.

When calling the block, the function value is represented as the first output parameter in LAD, FBD, and STL – 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. 5.8 shows an example: The block "Call2" is programmed like the block "Call" in the Fig. 5.6 on page 162, with the difference that the result of the selection is assigned to the function value instead of to an output parameter. The function value has the data type INT. It can be further processed directly in an expression in SCL. The example shows an arithmetic expression.

Example of application of the function value

In the "Selection2" function, the function value is declared in the block interface in the *Return* section, in the example with the data type INT.

Block interface

Declaration	Name	Data type
Input	Number_1	INT
	Number_2	INT
	Default_value	INT
	Switch	BOOL
Temp	Maximum	INT
Return	**Selection2**	**INT**

The block interface of the called block contains the four input parameters and the function value as result of the selection of the three digital tags. The program in the "Selection2" block can be written in any programming language.
When called in LAD, FBD and STL, the function value is handled like an output parameter.

Block call in an expression

```
"Temperature" := "Selection2" (Number_1      := "Measurement_1",
                               Number_2      := "Measurement_2",
                               Default_value := "Test_value",
                               Switch        := "Test_mode") + "Correction_value";
```

The "Selection2" 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. 5.8 Use of the function value with SCL

5.4.4 Calling a function block (FB)

Call type of a function block

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 have the following options: 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 also repeatedly call a function block in another function block as a local instance using different names in each case.

Chapter 10.7.13 “Data storage of a local instance in a multi-instance” on page 468 describes how the block parameters and the static local data are saved when calling as a local instance in a multi-instance.

Supplying block parameters of an FB

The block parameters of a function block are located in the instance data.

Block parameters with saved values do not have to be supplied when the function block is called. If the supply is omitted, the function block works with the “old” values from its last call or with the default settings.

Block parameters which are saved as pointer to the actual parameter must be supplied with an actual parameter when called. These are block parameters with the parameter type POINTER, ANY or VARIANT and in/out parameters with a structured data type. During programming, the program editor uses three question marks to indicate that a block parameter must be supplied or three periods to indicate that a supply can be omitted.

You can supply the EN enable input and ENO enable output as required.

5.4.5 “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 generally also applies here that the actual parameter must have a data type that is compatible with the formal parameter, the testing of which is controlled by the attribute *IEC check*. Note in this context that the maximum length may have to be considered in the validity check for 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. Exception: An in/out parameter with the data type POINTER or ANY cannot be created at an output parameter.

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 outputs.

5.5 Startup program

A CPU 1500 carries out a warm restart when started up. The activities carried out during the warm restart are described in Chapter 5.1.2 "STARTUP operating state" on page 146.

5.5.1 Startup organization blocks

A CPU 1500 provides 100 organization blocks with the numbers OB 100 and from OB 123 for the startup program. A startup organization block is assigned to the *Startup* event class. It is of hardware data type OB_STARTUP. The constant names and the values are listed in the *System constants* tab of the default tag table. The name of the constant can be changed in the block properties under *General*.

Start information

A startup organization block with the attribute *Optimized block access* activated provides the start information shown in Table 5.3 in the *Input* declaration section. A startup organization block with the attribute *Optimized block access* deactivated (OB with standard access) provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in Chapter 4.11.4

Table 5.3 Start information for a startup organization block

Declaration	Tag name	Data type	Description
The *Optimized block access* attribute is activated:			
Input	LostRetentive	BOOL	= "1" if retentive data areas have been lost
Input	LostRTC	BOOL	= "1" if the time of the real-time clock has been lost
The *Optimized block access* attribute is deactivated (standard access):			
Temp	STOP	WORD	Number of the STOP event
Temp	STRT_INFO	DWORD	Additional information for the current startup

“Start information” on page 142. This contains the tags specified in Table 5.3. Using these tags, you can determine which event triggered the last STOP and with which event the CPU has been started, e.g. with a manual startup using the mode switch (see reference of the startup organization blocks in the STEP 7 help). With this information you can create an event-triggered startup program.

Using the startup program

The startup program is executed one time. The startup organization blocks are called in the order of their numbering. A startup program is not essential. If no startup program is required, simply omit the organization blocks with the *Startup* event class.

The startup program can have any length. There is no time limit for executing the startup program; the cycle time monitoring is not active. 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”.

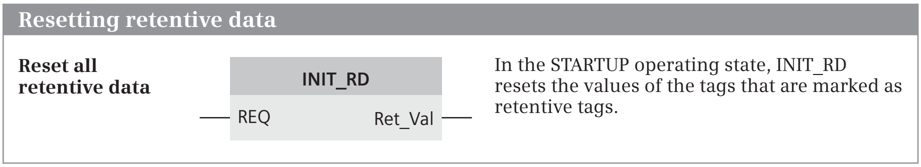
No interrupt events – except errors – are processed during execution of the startup program. Interrupts occurring during the startup are executed after the startup but before the main program.

Configuring the startup program

To configure the startup program, add an organization block with the event class *Startup* and enter the name, programming language, and number. The defined execution priority 1 cannot be changed.

5.5.2 Resetting retentive data

INIT_RD resets the values of the retentive tags (bit memories, data tags, and SIMATIC timer/counter function). Resetting is executed if the signal state at parameter REQ is “1”. The Ret_Val parameter outputs error information. INT_RD can only be called in a startup organization block. Fig. 5.9 shows the graphic representation of INT_RD.



Resetting retentive data

Reset all retentive data

In the STARTUP operating state, INIT_RD resets the values of the tags that are marked as retentive tags.

Fig. 5.9 System block for resetting the retentive data

5.5.3 Determining a module address

Signal modules, or more precisely the user data on input/output modules, are addressed in two manners: You use the *logical address* in the user program to address the inputs and outputs. This corresponds to the absolute address and can be made easier to read by using symbols. The smallest logical address of a module is the base address or module start address. The CPU addresses the modules using the *geographic address*. You require the geographic address if you wish to learn the module's slot. A module can also be addressed via its hardware identifier, which can be found in the module properties and in the *System constants* tab in the default tag table.

The following system blocks convert the various addressing options of a module (logical address, geographic address, hardware identifier):

- ▷ GEO2LOG Determine the hardware identifier from the geographic address
- ▷ LOG2GEO Determine the geographic address from the hardware identifier
- ▷ LOG2MOD Determine the hardware identifier from the logical address
- ▷ IO2MOD Determine the hardware identifier from an address list
- ▷ RD_ADDR Determine the address range of a module

Fig. 5.10 shows the graphic representation of the system blocks. These system blocks can be called in all priority classes, i.e. in the program of all organization blocks. You find them in the program elements catalog under *Extended instructions* > *Addressing*.

The following blocks are also available for the migration of S7-300/400 programs:

- ▷ GEO_LOG Determine logical start address
- ▷ GADR_LGC Determine logical address of a module channel
- ▷ RD_LGADR Determine all logical addresses of a module
- ▷ LOG_GEO Determine geographic address
- ▷ LGC_GADR Determine slot address of a module

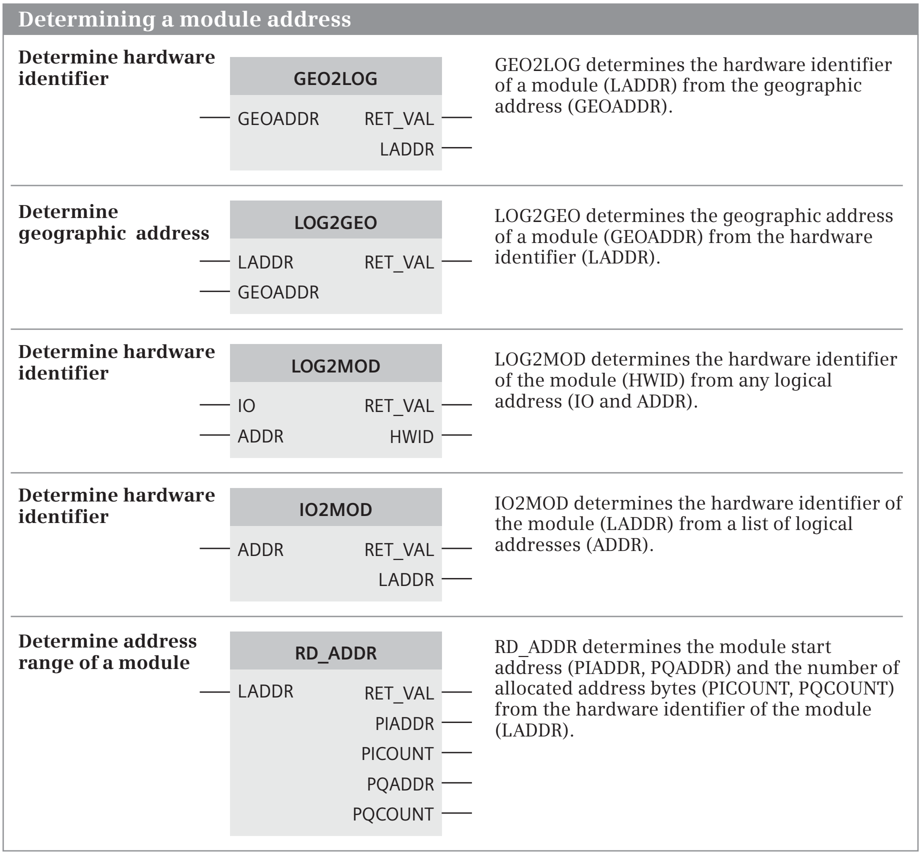
These system blocks are not intended for new applications and are therefore not described in further detail in the following.

GEO2LOG Determine the hardware identifier from the geographic address

GEO2LOG provides the hardware identifier of an object, with a geographic address that is specified as the parameter GEOADDR. Table 5.4 shows the structure of the GEOADDR system data type.

The component HWTYPE of the parameter GEOADDR defines the object with the hardware identifier that is output at the LADDR parameter. The components of GEOADDR which are used to define the object are placed in parentheses:

- ▷ HWTYPE = 1: Hardware identifier of the PROFINET IO system (IOSYSTEM)
- ▷ HWTYPE = 2: Hardware identifier of the IO device (IOSYSTEM, STATION)


Determining a module address
Determine hardware identifier
GEO2LOG
GEOADDR
RET_VAL
LADDR
Determine geographic address
LOG2GEO
LADDR
RET_VAL
GEOADDR
Determine hardware identifier
LOG2MOD
IO
RET_VAL
ADDR
HWID
Determine hardware identifier
IO2MOD
ADDR
RET_VAL
LADDR
Determine address range of a module
RD_ADDR
LADDR
RET_VAL
PIADDR
PICOUNT
PQADDR
PQCOUNT

GEO2LOG determines the hardware identifier of a module (LADDR) from the geographic address (GEOADDR).

LOG2GEO determines the geographic address of a module (GEOADDR) from the hardware identifier (LADDR).

LOG2MOD determines the hardware identifier of the module (HWID) from any logical address (IO and ADDR).

IO2MOD determines the hardware identifier of the module (LADDR) from a list of logical addresses (ADDR).

RD_ADDR determines the module start address (PIADDR, PQADDR) and the number of allocated address bytes (PICOUNT, PQCOUNT) from the hardware identifier of the module (LADDR).

Fig. 5.10 System blocks for determination of module addresses

- HWTYPE = 3: Hardware identifier of the rack (IOSYSTEM, STATION)
- HWTYPE = 4: Hardware identifier of the module (IOSYSTEM, STATION, SLOT)
- HWTYPE = 5: Hardware identifier of the submodule (IOSYSTEM, STATION, SLOT, SUBSLOT)

The AREA component is not evaluated. If the addressed object does not exist, error information is output at the RET_VAL parameter.

LOG2GEO Determine the geographic address from the hardware identifier

At parameter GEOADDR, LOG2GEO provides the geographic address of an object with a hardware identifier that is specified at the parameter LADDR. Table 5.4 shows the structure of the GEOADDR parameter.

If the hardware type of LOG2GEO is not supported, the value 0 is output at the component HWTYPE. If the addressed object does not exist, error information is output at the RET_VAL parameter.

Table 5.4 Structure of the GEOADDR parameter

Name of the component	Data type	Assignment, note
HWTYPE	UINT	Hardware type 1: IO system 2: IO device 3: Rack 4: Module 5: Sumodule
AREA	UINT	Area code 0: Central module
IOSYSTEM	UINT	PROFINET IO system 0: Central controller
STATION	UINT	Station number With AREA = 0: number of the rack
SLOT	UINT	Slot number of the rack
SUBSLOT	UINT	Slot number of the submodule

LOG2MOD Determine the hardware identifier from a logical address

LOG2MOD determines the hardware identifier from any logical address of the module. You define the type of logical address at the IOID parameter:

- B#16#00 Input/output is defined by bit 15 of ADDR
- B#16#54 Input address
- B#16#55 Output address

You specify any logical address of the module at the ADDR parameter. The parameter HWID provides the hardware identifier of the module. Error information is output at the RET_VAL parameter if the specified address is invalid.

IO2MOD Determine the hardware identifier from an address list

IO2MOD determines the hardware identifier of a module from a logical address or from a list of logical addresses at parameter ADDR (data type VARIANT). The hardware identifier is output at the LADDR parameter. The hardware identifier is generated from the first list entry. If this is invalid, error information is output at the RET_VAL parameter. The remaining list entries are ignored.

RD_ADDR Determine the address range of a module

RD_ADDR returns the module start address and the number of allocated address bytes of a module with a hardware identifier that is located at parameter LADDR. The start addresses are output at the parameters PIADDR (inputs) and PQADDR (outputs), the number of bytes allocated by the module are output at PICOUNT (input bytes) ad PQCOUNT (output bytes). If the hardware identifier is invalid, error information is output at the RET_VAL parameter.

5.5.4 Parameterization of modules

Most S7 modules can be parameterized, i.e. properties can be set on the module which are different from the default settings. To assign parameters, open the module in the hardware configuration and configure the module properties in the inspector window. When started, the CPU automatically transfers the configured module parameters to the modules and for the distributed I/O following the “return” of a station.

Specific module parameters can be changed during runtime with the aid of system blocks. Note that with a renewed startup the parameters set on the modules by the system blocks are overwritten by the parameters set (and saved on the CPU) using the hardware configuration.

Asynchronously working system blocks

Asynchronous execution means that the result of the block function is not immediately available following calling of the block. Execution of the function extends over several calls and is triggered by the block parameter REQ = “1”. The BUSY parameter has signal state “1” during job execution, and the error information has the value W#16#7001 (job being executed). The error information is located either in the parameter RET_VAL or in bytes 2 and 3 of the STATUS parameter.

A specific task is specified by the hardware identifier of the module and the data record number. As long as BUSY = “1”, a renewed call for the same job with REQ = “1” has no effect and the error information is set to W#16#7002.

BUSY has signal state “0” when the job has been completed. If completed without errors, the error information has the value W#16#0000; with the system function RD_REC, the number of transmitted bytes is present in RET_VAL. In the event of an error, the error information contains the error code.

You can use a program loop in which the asynchronous system block is called in the startup program to “wait” for the end of job processing. You are advised not to do this in the main, interrupt or error program, since it can result in undesirable delays in the cycle processing time and thus in the response time, and the cycle monitoring time may then be triggered.

Please note that the maximum number of simultaneously running asynchronous system blocks depends on the CPU used.

Module and data record addressing

Use the hardware identifier of the module (module ID) for addressing for the data record transfer. The hardware identifier is located in the module properties under *Hardware identifier* and in the *System constants* tab of the default tag table. During runtime, you can determine the hardware identifier using the system block LOG2-MOD from the logical module address.

Data records have a number from 0 to 240. You can consult the manual of the module to learn whether a module can be parameterized during runtime and, if so, which number and which configuration the data records have. The Table 5.5 shows a basic overview of the system data records for module parameterization and module diagnostics.

Table 5.5 Overview of system data records for modules

Writing data records			**Usable system block**
Number	**Number**	**Size per data record**	
0	Parameter	–	WR_DPARM
1	Parameter	–	WR_DPARM
2 to 127	User data	Up to 240 bytes	WR_DPARM, WRREC
128 to 240	Parameter	Up to 240 bytes	WR_DPARM, WRREC
Reading data records			**Usable system block**
Number	**Content**	**Size per data record**	
0	Module-specific diagnostic data	4 bytes	RDREC
1	Channel-specific diagnostic data	4 to 220 bytes	RDREC
2 to 127	User data	Up to 240 bytes	RDREC
128 to 240	Diagnostic data	Up to 240 bytes	RDREC

System blocks for the transmission of data records

The following system blocks for transferring data records to and from modules are available for a CPU 1500:

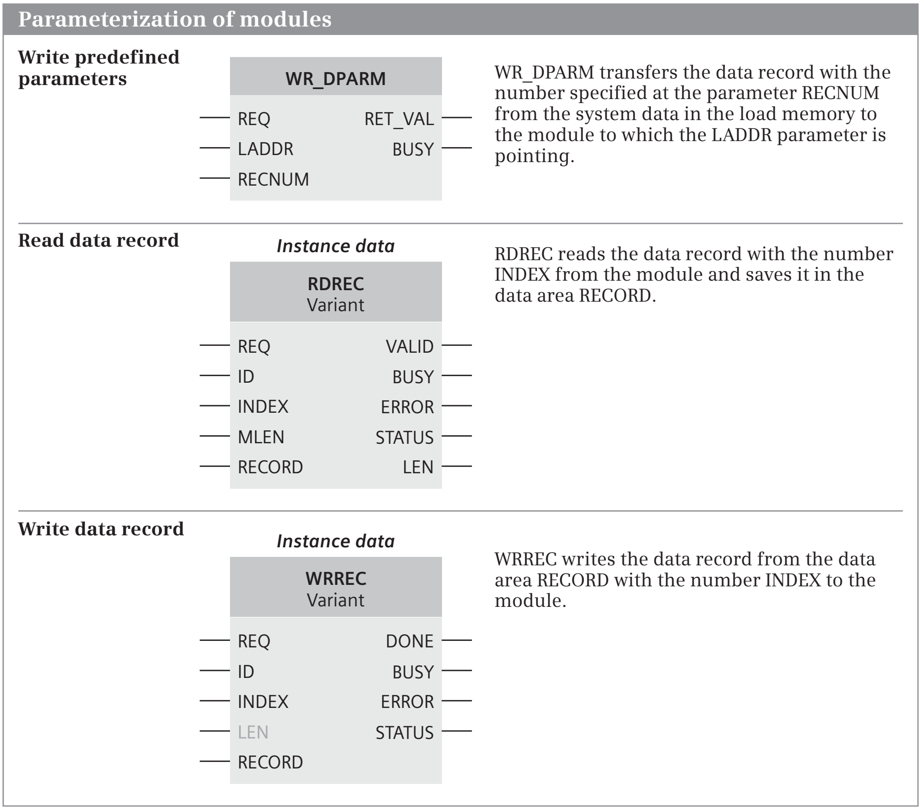
- WR_DPARM Write configured parameters to the module
- RDREC Read data record
- WRREC Write data record

You find WR_DPARM in the program elements catalog under *Extended instructions > Module parameter assignment*. You find RDREC and WRREC in the program elements catalog under *Extended instructions > Distributed I/O*.

The following blocks are also available for the migration of S7-300/400 programs:

- RD_DPAR Read predefined parameters
- RD_DPARA Read predefined parameters
- RD_DPARM Read predefined parameters
- RD_REC Read data record
- WR_REC Write data record

These system blocks are not intended for new applications and are therefore not described in further detail in the following. Fig. 5.11 shows the graphic represen-



Parameterization of modules

Write predefined parameters

WR_DPARM transfers the data record with the number specified at the parameter RECNUM from the system data in the load memory to the module to which the LADDR parameter is pointing.

Read data record

RDREC reads the data record with the number INDEX from the module and saves it in the data area RECORD.

Write data record

WRREC writes the data record from the data area RECORD with the number INDEX to the module.

Fig. 5.11 System blocks for module parameterization and transmission of data records

tation of the system blocks for module parameterization described in the following.

Using the hardware identifier at the parameter LADDER or ID, you can specify the module whose parameter data is to be read or written. You create the data record number at the parameter RECNUM or INDEX. RECORD specifies the data area in which the read data record is to be stored or from which the data record that is to be written will be transferred. The actual parameter at RECORD can be any tag or an operand area that is absolutely addressed with the ANY pointer.

WR_DPARM Write predefined parameters

WR_DPARM transfers a data record from the system data located in the load memory to the module. You specify the data record number at the RECNUM parameter. You address the module with the hardware identifier at the LADDR parameter.

The data record is completely read for the task initiation with REQ = “1”; the transfer can be distributed across several program cycles. The parameter BUSY has signal state “1” during the transfer.

RDREC Read data record

RDREC reads a data record from the module whose hardware identifier is in the ID parameter. You specify the data record number at the INDEX parameter. The read data record is stored in the target area, which is specified by the RECORD parameter. This can be a tag or an operand area that is absolutely addressed with the ANY pointer. The target area must be large enough to accommodate the data record. The MLEN parameter specifies how many bytes are to be read.

The job is triggered with “1” at the REQ parameter. The transfer can be divided between several program cycles; the BUSY parameter has signal state “1” during the transfer.

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 the module whose hardware identifier is in the ID parameter. You specify the data record number at the INDEX parameter. The data record that is to be read is taken from the source area, which is specified by the RECORD parameter. This can be a tag or an operand area that is absolutely addressed with the ANY pointer. The LEN parameter specifies how many bytes are to be written.

The job is triggered with “1” at the REQ parameter. The transfer can be divided between several program cycles; the BUSY parameter has signal state “1” during the transfer.

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.

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 Main program organization blocks

A CPU 1500 provides 100 organization blocks with the numbers OB 1 and from OB 123 for the main program. A main program organization block is assigned to the *Program cycle* event class. It is of hardware data type OB_PCYCLE. The constant names and the values are listed in the *System constants* tab of the default tag table. The name of the constant can be changed in the block properties under *General*.

Start information

A main organization block with the attribute *Optimized block access* activated provides the start information shown in Table 5.6 in the *Input* declaration section. A main program organization block with the attribute *Optimized block access* deactivated (OB with standard access) provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in Chapter 4.11.4 "Start information" on page 142. This contains the tags specified in Table 5.6. Using these tags you can determine the current cycle time and its fluctuation per program.

Table 5.6 Start information for a main program organization block

Declaration	Tag name	Data type	Description
The *Optimized block access* attribute is activated:			
Input	Initial_Call	BOOL	= "1" for the first call of the organization block
Input	Remanence	BOOL	= "1" if retentive data is available
The *Optimized block access* attribute is deactivated (standard access):			
Temp	PREV_CYCLE	INT	Runtime of previous cycle (in ms)
Temp	MIN_CYCLE	INT	Minimum cycle time since last startup (in ms)
Temp	MAX_CYCLE	INT	Maximum cycle time since last startup (in ms)

Using the main program

The main program is cyclically executed, i.e. if the execution has reached the end of the main program, a minimum cycle time is waited through (if configured) before the execution of the main program is started again from the beginning. The main program organization blocks are called in the order of their numbering.

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 (see Chapters 5.7 "Interrupt processing" on page 192 and 5.8 "Error handling" on page 212).

The execution of the main program with all interrupt events that occurred in the current processing cycle is monitored by the cycle time monitoring (see Chapter 5.6.3 "Cycle time" on page 182).

Configuring the main program

To configure the main program, add an organization block with the event class *Program cycle* and enter the name, programming language, and number. The defined execution priority 1 cannot be changed.

5.6.2 Process image updating

The process image is part of the CPU's internal system memory (see Chapter 4.1 “Operands and tags” on page 86). 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 32 KB per area. The user data of all of the modules is located in the address area of the process image.

The process image input and output can be comprised of several process image partitions, independent of one another. The updating of the process image (partitions), i.e. the data transfer from and to the modules, can take place automatically or be controlled by system functions via the user program.

Benefits of a process image

The use of a process image has many benefits:

- The scanning of an input or the controlling of an output is significantly faster than the addressing of an input or output module, e.g. the setting times at the I/O bus are omitted and the response times of the system memory are shorter than the response times of the module. This means that the program is executed faster.
- Inputs can also be set and reset since they are stored in a Random Access Memory. Digital input modules can only be read. The setting of the inputs can simulate encoder statuses during the program test or commissioning, thereby simplifying the program test.
- Outputs can also be scanned since they are stored in a Random Access Memory. Digital output modules can only be written. The scanning and linking of the outputs does away with the additional saving of output bits to be scanned by the user.
- The signal state of an input is the same throughout the entire program cycle (data consistency during a program cycle). If a bit on an input module changes, the change of the signal state is transferred to the input at the start of the next program cycle.
- A multiple signal state change of an output during a program cycle has no effect on the bit on the output module. The signal state of the output at the end of the program cycle is transferred to the module.

The downside of these benefits is an increased response time of the program; see Chapter 5.6.3 “Cycle time” on page 182 for more details.

Activating the automatic process image update

If the value *Automatic update* is entered in the *Process image* field under *I/O addresses*, the user data of this module is updated before the main program is executed. Initially, the process image output is transferred to the output modules. After this, the signal states of the input modules are imported into the process image input. This is then followed by the execution of the main program (Fig. 5.12).

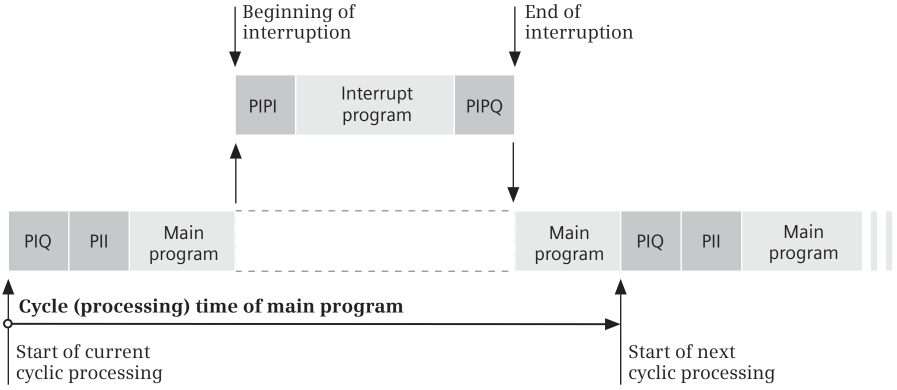
Automatic update of the process images

Automatic update of the main program process image

The user data of all modules with the value *Automatic update* as process image is updated before the main program is executed. First, the process image output is sent to the output modules. Then the signal states of the input modules are transferred to the process image input. This is then followed by the execution of the main program. The main program process image comprises the addresses of the module, which are updated automatically and are not assigned to any (other) process image partition.

Automatic update of an assigned process image partition

If the user data of a module are located in a process image partition and the process image partition is assigned to an organization block, the processing of the organization block is initiated with the update of the process image partition input. The program of the organization block is executed afterwards. Processing of the interrupt ends with the transmission of the process image output to the output modules.


Beginning of interruption
End of interruption
PIPI
Interrupt program
PIPQ
PIQ
PII
Main program
Main program
PIQ
PII
Main program
Cycle (processing) time of main program
Start of current cyclic processing
Start of next cyclic processing

PIQ = process image output of main program
PII = process image input of main program
PIPQ = process image partition output
PIPI = process image partition input

Fig. 5.12 Automatic process image update

Process image partitions

With a CPU 1500, you can divide the process image into as many as 31 process image partitions. You can assign a module to a process image partition in the module properties. Enter a value *PIP1* to *PIP31* from the drop-down list in the *Process image* field under *I/O addresses*.

Carry out the assignment separately for the process image input and process image output. A module can only be assigned to a single process image partition and only with all of its addresses. All of the modules addressed in the process image which you do not assign to a process image partition PIP1 to PIP31 are in the “residual process image” (process image of main program).

Assigning a module to a process image partition makes sense if all of the user data of the module is processed in an interrupt routine or if specific program sections are to be provided with their own process image. If you use the isochronous mode interrupt, you must assign the participating modules to a process image partition.

You can also simultaneously assign an organization block while assigning the process image partition. Then the assigned process image partition is automatically updated when the organization block is executed (Fig. 5.12). You can assign the organization block in the module properties under *I/O addresses* in the *Organization block* field. Choose an existing organization block from a drop-down list or create a new one using the *Add object* button.

Deactivating the automatic process image update

If you choose the entry *None* in the *Process image* field under *I/O addresses* in the module properties, the user data of this module will not be updated. The user data of the module must then be accessed via the I/O operand area.

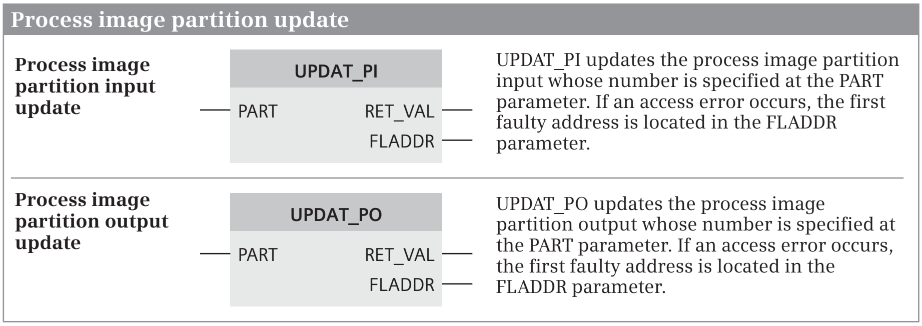
If you select a process image partition in the *Process image* field and do not select an organization block in the *Organization block* field, the user data of this module will not be automatically updated. You then have the option of initiating the updating of the process image partition with system blocks (or to access the user data of the module via the I/O operand area).

Process image update with system blocks

The system functions UPDAT_PI and UPDAT_PO are available for updating the process image partitions via the user program. In the isochronous interrupt organization blocks, you can use the system functions SYNC_PI and SYNC_PO for updating the process image partitions (see Chapter 16.7.4 “Isochronous mode interrupt” on page 745).

UPDAT_PI Update process image partition of inputs
UPDAT_PO Update process image partition of outputs

UPDAT_PI updates a process image partition of the inputs, UPDAT_PO a process image partition of the outputs. You cannot update the process image of the main program with these system blocks. You find the system blocks in the program elements catalog under *Extended instructions* > *Process image* (Fig. 5.13).



Process image partition update

Process image partition input update

UPDAT_PI updates the process image partition input whose number is specified at the PART parameter. If an access error occurs, the first faulty address is located in the FLADDR parameter.

Process image partition output update

UPDAT_PO updates the process image partition output whose number is specified at the PART parameter. If an access error occurs, the first faulty address is located in the FLADDR parameter.

Fig. 5.13 System blocks for a process image partition update

You can call these system functions at any point in the user program. You enter the number of the process image partition (1 to 31) at the PART parameter. The selected process image partition may not be automatically updated (no assignment to an organization block or deactivation of automatic updating of the main program process image) and it may not be updated with the system blocks SYNC_PI or SYNC_PO.

The updating of a process image can be interrupted by an organization block with a higher priority class. If an error occurs during the updating of the process image, e.g. because a module can no longer be addressed, it is reported back via the function value RET_VAL of the system block. The first error-causing address is then located in the FLADDR parameter.

5.6.3 Cycle time

Cycle monitoring time

Execution of the main program is monitored with regard to timing by means of the *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 (Fig. 5.14).

Cycle processing time and minimum cycle time

Cycle processing time

The cycle monitoring time comprises the complete duration of main program execution, including all interruptions.

Cycle monitoring time

The cycle monitoring time monitors the duration of main program execution.
If it expires, the CPU operating system signals an error.
The cycle monitoring time can be extended (“retriggered”).

Minimum cycle time

If a minimum cycle time is activated, cyclic execution of the main program begins only after the minimum cycle time has expired. This must be greater than the cycle processing time.

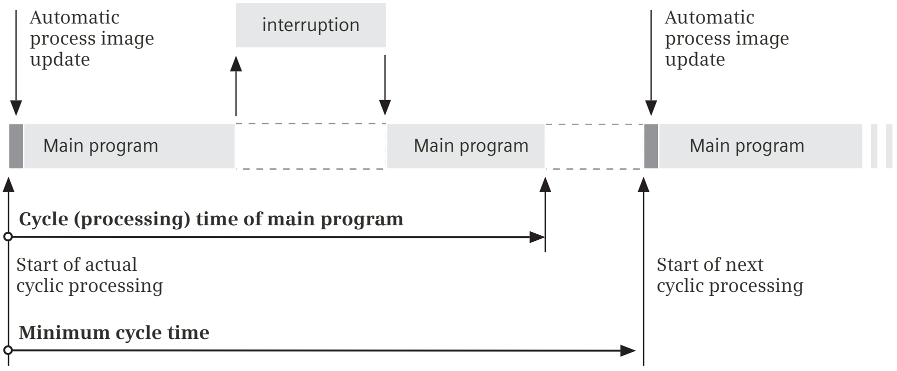

Automatic process image update
interruption
Automatic process image update
Main program
Main program
Main program
Cycle (processing) time of main program
Start of actual cyclic processing
Start of next cyclic processing
Minimum cycle time

Fig. 5.14 Cycle processing time and minimum cycle time

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!).

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 is not present, a CPU 1500 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.

Minimum cycle time

In addition to the maximum cycle (monitoring) time, you can also set a minimum cycle time in the CPU properties. The minimum cycle time must be longer than the cycle (processing) time and shorter than the cycle (monitoring) time.

When the minimum cycle time is activated, the CPU waits at the end of the main program execution until the minimum cycle time has elapsed and only then it begins with a new program cycle. If execution of the main program takes longer than the set minimum cycle time, this has no further effects.

With a minimum cycle time, you can reduce large fluctuations in the processing time and thus large fluctuations in the response time. While the CPU waits for the minimum cycle time to elapse, it can perform communication tasks.

RE_TRIGR Restart cycle monitoring time

RE_TRIGR restarts the cycle monitoring time. This then starts with the value set during CPU parameterization. ENO has signal state "1". RE_TRIGR does not have any parameters (Fig. 5.15).

You can find RE_TRIGR in the program elements catalog under *Basic instructions* > *Program control operations*. The RE_TRIGR function is only executed if it is called in the main program. The cycle monitoring time is not restarted by a call in the startup program or in an interrupt routine, and ENO has the signal state "0".

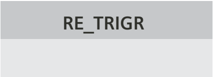
Retrigger cycle monitoring time		
Retrigger cycle monitoring time		The call retriggers the cycle monitoring time with the configured duration.

Fig. 5.15 System block for retriggering the cycle monitoring time

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 maximum 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. 50% is set by default.

If *k* represents the communication load in percent, the execution 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.

The processing of organization blocks with a processing priority higher than 15 is not interrupted by communication.

Cycle statistics

If you are connected online to the programming device with a running CPU, you can use the *Online & diagnostics* command from the project tree to start the task card with the online tools. The *Cycle time* section shows the shortest, current, and longest cycle (processing) time in milliseconds and presents these graphically.

You can also obtain data on the current cycle time of the last cycle as well as the minimum and maximum cycle times since the last startup from the start information of a main program organization block if the *Optimized block access* attribute is deactivated.

RUNTIME Runtime measurement

RUNTIME measures the program runtime between two calls. The same actual parameters that were applied to the parameters during the first call must be applied during the second call. RUNTIME is available in the SCL and STL programming languages. You can find RUNTIME in the program elements catalog under *Basic instructions > Program control operations* (Fig. 5.16).

If RUNTIME is called once in a main program organization block, RUNTIME will output the current runtime of the main program at the parameter RET_VAL after the second program cycle and after each additional one.

5.6.4 Response 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 cycle execution time (in short: cycle time). The response time lies between one and two cycle times, as demonstrated in the following example (Fig. 5.17).

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

Measure runtime with RUNTIME

Measure runtime

SCL

```
#var_retval := RUNTIME(#var_mem);
```

STL

```
CALL  RUNTIME
      Ret_Val  := #var_retval
      MEM      := #var_mem
```

Data types:
MEM: LREAL
Ret_Val: LREAL

The first call starts the runtime measurement and saves the start time in the MEM array. The second call determines the difference between the start time (saved in MEM) and the time of the call and outputs the runtime in seconds at the parameter RET_VAL.

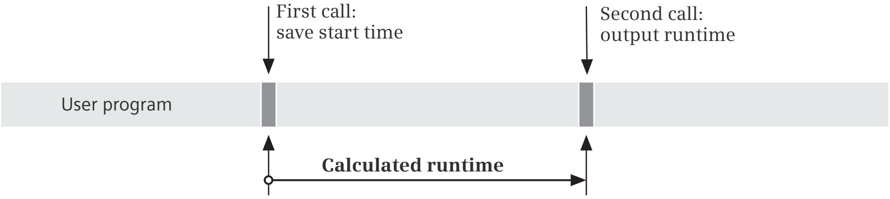
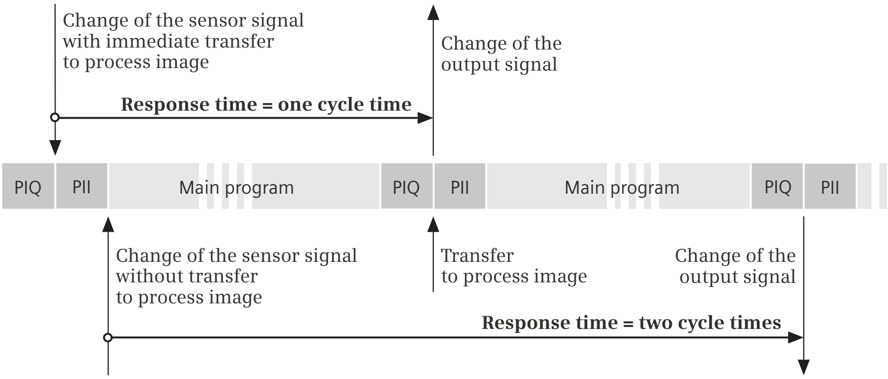


Fig. 5.16 System block for measuring the runtime

Response times when using process images



PII = process image input
PIQ = process image output

Fig. 5.17 Response times of the main program

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. 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 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 for the input and output modules, the switching times of contactors, and so on.

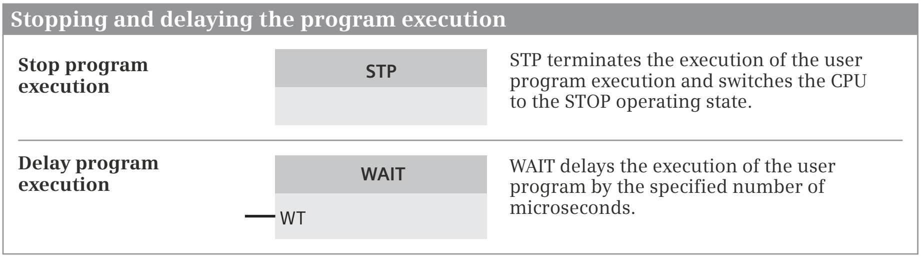
In certain cases you can reduce the response times by addressing the I/O directly or by calling program sections depending on events (hardware interrupt).

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 cyclic interrupt program. Program execution isochronous with the processing cycle of a PROFINET IO system or PROFIBUS DP master system also results in calculable response times.

5.6.5 Stopping and delaying the program

STP Stop program execution

The system function STP terminates program execution; the CPU then switches to the STOP operating state. STP has no parameters (graphic representation is shown in Fig. 5.18).



STP terminates the execution of the user program execution and switches the CPU to the STOP operating state.

WAIT delays the execution of the user program by the specified number of microseconds.

Fig. 5.18 System blocks for stopping or delaying program execution

The CPU terminates processing of the user program and updates the process image output. In the module properties of correspondingly designed modules, you can set the signal states of the digital and analog outputs which the CPU is to output in the STOP operating state: *Shutdown*, *Keep last value*, or *Output 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 state, the CPU continues communication with the programming device and the diagnostics activities.

WAIT Delay program execution

The system function WAIT holds program execution for a defined duration (Fig. 5.18).

The system function WAIT has the input parameter WT with data type INT in which you can specify the hold time in microseconds (µs). The maximum hold time is 32 767 µs, the smallest possible hold time corresponds to the CPU-dependent execution time of the system function.

WAIT can be interrupted by events of higher priority.

5.6.6 Time

Each CPU 1500 has a real-time clock with a typical deviation of 2 s per day. The duration of the buffering is approx. 6 weeks at an ambient temperature of 40 °C. The time is not affected by a memory reset. When it is reset to the factory settings, the clock starts at DTL#2012-01-01-00:00:00 GMT.

The clock can be synchronized and can be set or queried in the user program using the display of the CPU, a programming device, or system blocks.

Module time, local time, daylight saving/standard time

The time set in the CPU's real-time clock is the module time (basic 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 module time, RD_SYS_T reads the module time. The module time can also be set using the display of the CPU or online using the programming device. The time is converted to UTC time.

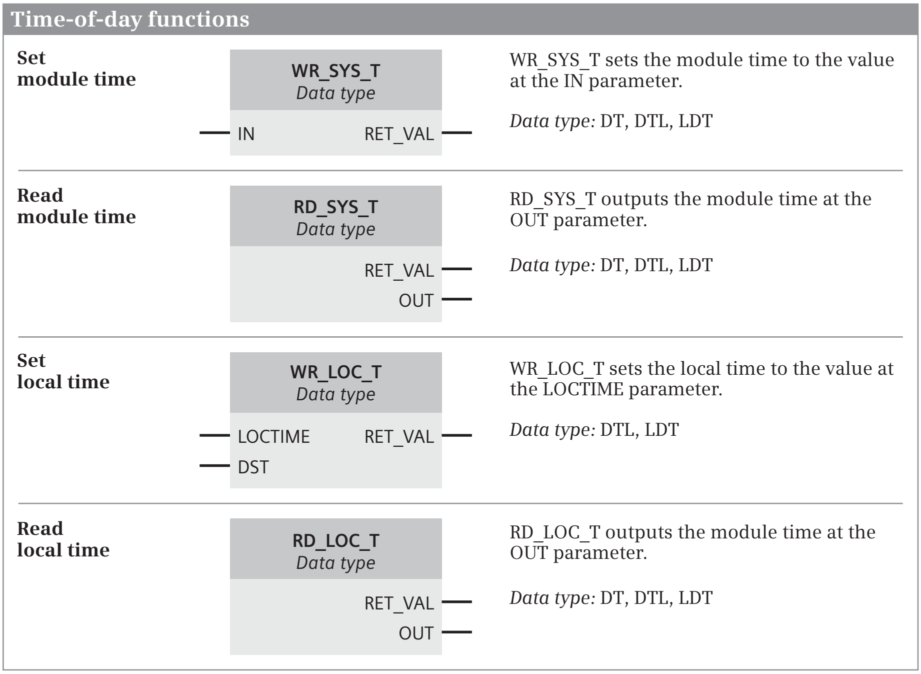
The local time is set by addition of a correction factor which can also be negative. Configuration is carried out when parameterizing the CPU with the device configuration. The local time can be used to visualize time zones. It is read with RD_LOC_T and set with WR_LOC_T.

WR_SYS_T Set module time

WR_SYS_T (Write System Time) sets the CPU's clock to the value specified at the IN parameter (Fig. 5.19). The module time can be defined in the data format DATE_AND_TIME (DT) or DATE_AND_LTIME (DTL or LDT). 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”.

RD_SYS_T Read module time

RD_SYS_T (Read System Time) reads the CPU's current module time and outputs it in the OUT parameter (Fig. 5.19). The module time can be output in the data format DATE_AND_TIME (DT) or DATE_AND_LTIME (DTL or LDT). This value does not



Set module time

WR_SYS_T sets the module time to the value at the IN parameter.

Data type: DT, DTL, LDT

Read module time

RD_SYS_T outputs the module time at the OUT parameter.

Data type: DT, DTL, LDT

Set local time

WR_LOC_T sets the local time to the value at the LOCTIME parameter.

Data type: DTL, LDT

Read local time

RD_LOC_T outputs the module time at the OUT parameter.

Data type: DT, DTL, LDT

Fig. 5.19 System blocks for the time functions

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".

Configuring the local time

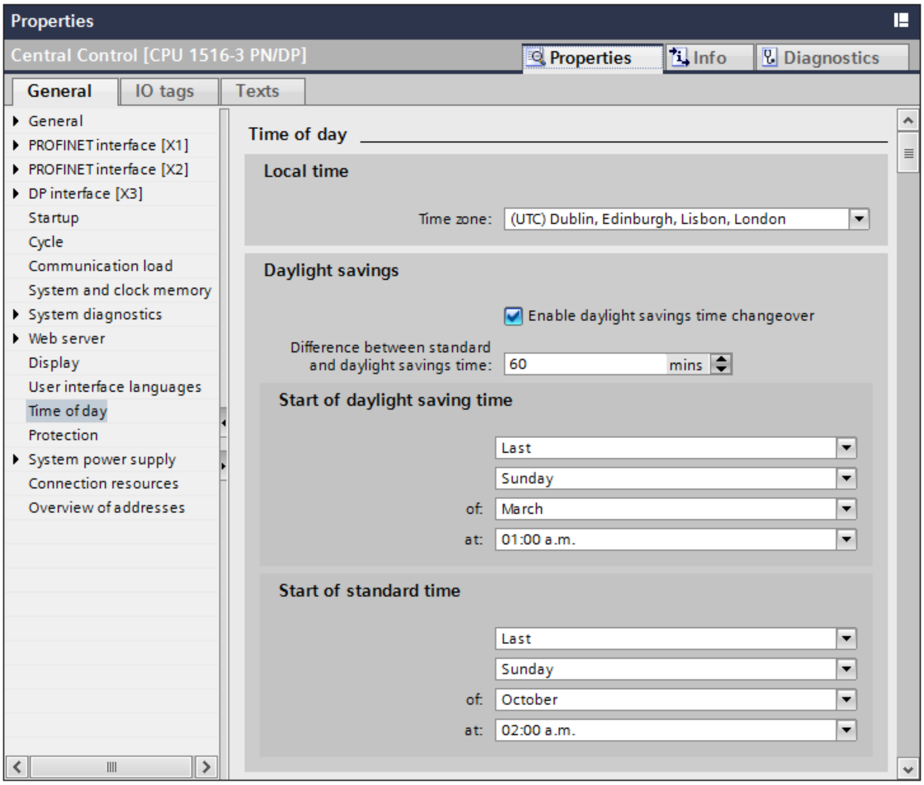
The time zone and the switching over between daylight saving and standard time is set in the properties of the CPU: Select the CPU in the device configuration, and open the *Time of day* section in the *Properties* tab in the inspector window. Set the time zone (local time), check the *Enable daylight savings time changeover* box, specify the time difference between daylight saving and standard time and also the dates and times of changeover (Fig. 5.20).

WR_LOC_T Set local time

WR_SYS_T (Write Local Time) sets the CPU's clock (Fig. 5.19). You specify the local time at the LOCTIME parameter. WR_LOC_T then calculates the module time based on the setting to local time in the properties of the CPU and uses it to set the clock.

The local time is defined in the data format DATE_AND_LTIME (DTL or LDT). The RET_VAL parameter (0 = no error) outputs the error information. In the event of an error, ENO is set to signal state "0".

If, during the changeover to standard time, the clock is set during the hour which exists twice, the hour is specified at the parameter DST: with signal state "1", the



Time of day

Local time

Time zone: (UTC) Dublin, Edinburgh, Lisbon, London

Daylight savings

☑ Enable daylight savings time changeover

Difference between standard and daylight savings time: 60 mins

Start of daylight saving time

Last
Sunday
of: March
at: 01:00 a.m.

Start of standard time

Last
Sunday
of: October
at: 02:00 a.m.

Fig. 5.20 Parameterization of local time and daylight saving/standard time changeover

first hour (still daylight saving time) and with signal state = “0” the second hour (already standard time).

RD_LOC_T Read local time

RD_LOC_T (Read Local Time) reads the CPU's current local time and outputs it at the OUT parameter with the DATE_AND_TIME (DT) or DATE_AND_LTIME (DTL or LTD) data type (Fig. 5.19). The local time is calculated based on the setting to the local time in the properties of the CPU.

The RET_VAL parameter (0 = no error) outputs the error information. 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 available system functions are described in Chapter 13.4.2 “Arithmetic functions for date and time” on page 576.

Setting the time via the CPU display

In the main menu of the CPU display, select the *Settings* icon and then the *Date & time* submenu. You can enter the date and time as local time.

Setting the time on the CPU online

You can read and set the system time (module time) on the CPU using the programming device online. To do this, open the project and start the *Online & diagnostics* editor in the project tree under the PLC station. To establish the online operation, click on the *Go online* icon in the toolbar of the project view or on the *Go online* button in the *Online access* section of the dialog window.

In the *Functions* section of the diagnostics window, select the *Set time* command. The current time of the programming device and the module time of the CPU are displayed. You can import the programming device time as the module time or you can set the module time itself.

Time synchronization via PROFINET

The time of the CPU can be synchronized via Ethernet. A time server is required, which is synchronized with the time of other 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 command *Time synchronization* in the properties of the inspector window. Check the checkbox *Enable time synchronization via NTP server*, specify the IP addresses of the participating servers, and select the updating interval.

Time synchronization via PROFIBUS

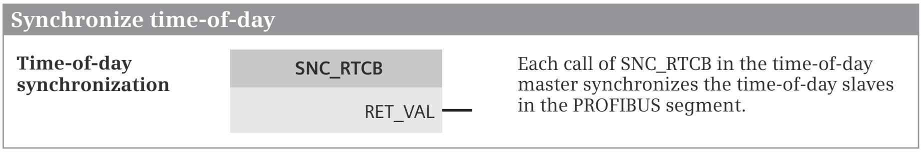
The clocks of all CPUs whose stations are connected to each other in a PROFIBUS segment can be synchronized. Parameterize one of the clocks as the master clock and parameterize the others as slave clocks.

Activate the time synchronization in the properties of the DP interface using the hardware configuration. Under *Time synchronization*, specify the *Synchronization type* (Master, Slave) and parameterize the *Update cycle*. The time synchronization takes place after each setting of the master clock in the parameterized interval.

You can synchronize the clocks of all time slaves in the bus segment, independently of the set interval, by calling the system function SNC_RTCB in the user program of the time master.

SNC_RTCB Synchronize time-of-day

You can synchronize the clocks of all time slaves in the bus segment, independently of the set interval, by calling the system function SNC_RTCB in the user program of the time master. Fig. 5.21 shows the graphic representation of the system function.



Each call of SNC_RTCB in the time-of-day master synchronizes the time-of-day slaves in the PROFIBUS segment.

Fig. 5.21 System block for synchronizing the time

5.6.7 Read system time

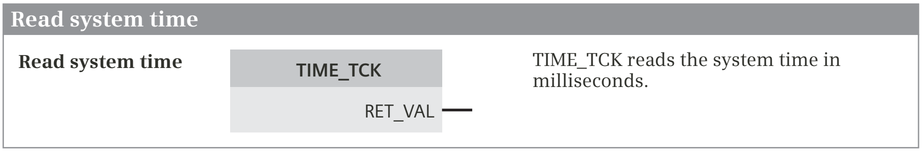
The system time is updated at an interval of one millisecond for a CPU 1500. The system time starts when the CPU is switched on. The system time runs for as long as the CPU is in the STARTUP or RUN operating state. The current value of the system time is “frozen” when at STOP. A warm restart resets the system time.

The system time is present in the data format TIME, where only positive values are possible: TIME#0ms to TIME#24d20h31m23s647ms. In the event of an overflow, the system time restarts at TIME#0.

You can use the system time, for example to determine the current runtime of the CPU or to calculate the duration between two TIME_TCK calls by generating the difference.

TIME_TCK Read system time

TIME_TCK reads the current system time. The RET_VAL parameter contains the read system time in the TIME data format. Fig. 5.22 shows the graphic representation of the system function.



TIME_TCK reads the system time in milliseconds.

Fig. 5.22 System function for reading the system time

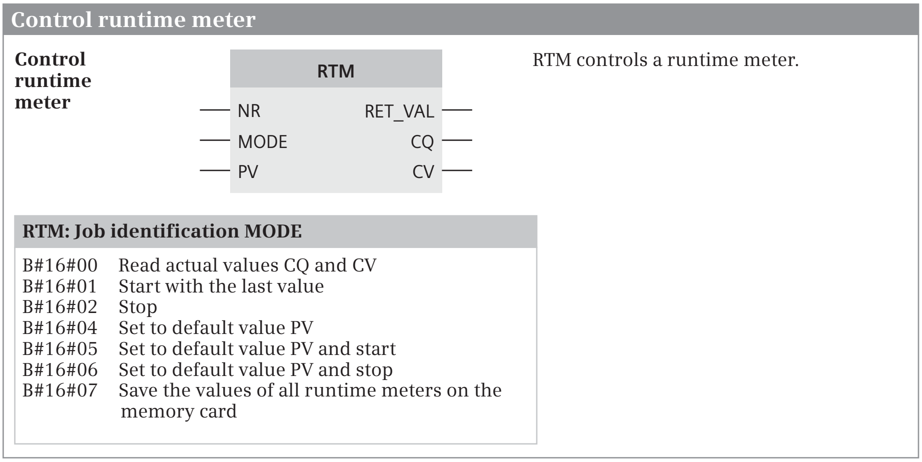
5.6.8 Runtime meter

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 runtime meter has a value range of 32 bits (2^{31}–1 hours). 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.

A 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 a runtime meter is retained on restart and after a memory reset. Resetting to the factory settings also resets a runtime meter to zero.

RTM Control runtime meter

RTM controls a runtime meter. Fig. 5.23 shows the graphic representation of the system function.



RTM controls a runtime meter.

RTM: Job identification MODE	
B#16#00	Read actual values CQ and CV
B#16#01	Start with the last value
B#16#02	Stop
B#16#04	Set to default value PV
B#16#05	Set to default value PV and start
B#16#06	Set to default value PV and stop
B#16#07	Save the values of all runtime meters on the memory card

Fig. 5.23 System block for controlling the runtime meter

RTM controls the runtime meter whose number is specified at 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 CQ parameter signals with signal state "1" if 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. Note that the number of write accesses to the memory card is physically limited.

5.7 Interrupt processing

5.7.1 Introduction to interrupt processing

Events

The response of the operating system is based on events. Events can be, for example, the one-time start of the startup, the cyclically recurring start of main program execution, a hardware interrupt, or a programming error.

If an organization block is assigned to the event, the operating system calls this organization block when the event occurs. If no organization block is assigned to an event, the operating system executes the preset system response when the event occurs: The operating system ignores the event or it changes to the STOP operating state or it carries out block-local error processing.

Execution order

A priority scheduler controls the execution order if events occur virtually simultaneously. Events with the same priority are processed in the order in which they occurred.

An event of higher priority interrupts execution of the program in an organization block to which an event with lower priority has been assigned. Such an interruption can take place after every operation (statement). Once this program with higher priority has been executed, the operating system resumes execution of the lower-priority program at the point of interruption.

Example: If a hardware interrupt occurs while the main program is executing, the operating system will interrupt the execution of the main program and call the organization block that is assigned to the hardware interrupt. When the interrupt routine has been executed, the execution of the main program will continue at the point where it was interrupted.

You can influence the interruption of a program by events of higher priority using system blocks (Chapter 5.8.6 “Disable, delay, and enable interrupts and asynchronous errors” on page 223).

Available organization blocks

Table 5.7 shows the organization blocks present with CPU 1500 with their execution priority.

The organization blocks for time-of-day, time-delay, cyclic and hardware interrupts are described in this chapter. The other organization blocks are described in the following chapters:

- Chapter 5.5.1 “Startup organization blocks” on page 169,
- Chapter 5.6.1 “Main program organization blocks” on page 177,
- Chapter 5.8.3 “Global error handling (synchronous error)” on page 215,
- Chapter 5.8.5 “Asynchronous errors” on page 220,
- Chapter 5.9.1 “Diagnostics interrupt” on page 226,
- Chapter 16.6 “DPV1 interrupts” on page 737 and
- Chapter 16.7.4 “Isochronous mode interrupt” on page 745.

Execution priorities

The main program has the fixed priority 1 and can be interrupted by all interrupt and error events. The startup program belongs to the same priority class as the main program: The operating system prevents both of them from being called at the same time. Interrupt events that occur during the startup phase are saved in a queue and are processed before the main program after the transition to the RUN operating state.

Table 5.7 Organization blocks of a CPU 1500

OB No.	Priority (default)	Start event	Event class	Number of OBs
1, ≥123	1	Start of main program	Program cycle	0 to 100
10 to 17, ≥123	2 to 24 (2)	Time-of-day interrupt	Time of day	0 to 20
20 to 23, ≥123	2 to 24 (2)	Time-delay interrupt	Time delay interrupt	0 to 20
30 to 38, ≥123	2 to 24 (8 to 17) *)	Cyclic interrupt	Cyclic interrupt	0 to 20
40 to 47, ≥123	2 to 26 (18)	Hardware interrupt	Hardware interrupt	0 to 50
55 56 57	2 to 24 (4) 2 to 24 (4) 2 to 24 (4)	Status interrupt Update interrupt Manufacturer-spec. interrupt	Status Update Profile	0 or 1 0 or 1 0 or 1
61 to 64, ≥123	16 to 26 (25)	Isochronous mode interrupt	Synchronous cycle	0 to 2
80 82 83 86	22 2 to 26 (5) 2 to 26 (6) 2 to 26 (6)	Time error Diagnostics interrupt Insert/remove module interrupt Rack failure	Time error interrupt Diagnostic error interrupt Pull or plug of modules Rack or station failure	0 or 1 0 or 1 0 or 1 0 or 1
91 92	17 to 26 (25) 16 to 26 (24)	MC servo interrupt MC interpolator interrupt	MC-Servo MC-Interpolator	0 or 1 0 or 1
100, ≥123	1	Warm restart	Startup	0 to 100
121 **) 122 **)	2 to 26 (7) 2 to 26 (7)	Programming error I/O access error	Programming error IO access error	0 or 1 0 or 1

*) depending on the call interval
**) only for global error handling

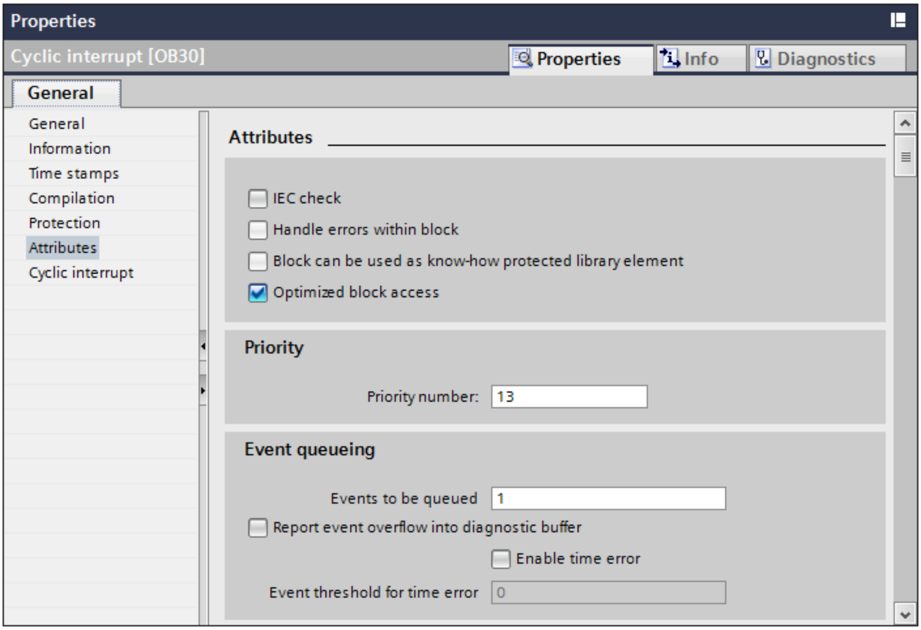
The communication with the programming device or the exchange of data with other PLC stations takes place in the "time slice mode" for a CPU 1500. While the user program is executing, the operating system carries out the communication "slice by slice" in a grid, which can be influenced using the CPU parameter *Communication load*. This communication has priority 15 and can thus interrupt the program execution in an organization block that has the same or lower priority.

How to influence a program interruption during runtime by means of higher-priority events with system functions is described in Chapter 5.8.6 "Disable, delay, and enable interrupts and asynchronous errors" on page 223.

Overload behavior

If several such events follow each other at such short intervals that execution "cannot keep up", the events are saved in a queue and are processed in succession. Each event type (each priority class) has its own queue. If the queue is full, the next equivalent event is counted and discarded.

In the properties of an organization block in which an overload can occur, you can set the response to an overload response under *Attributes* and *Event queuing* (Fig. 5.24).



Attributes

□ IEC check
□ Handle errors within block
□ Block can be used as know-how protected library element
☑ Optimized block access

Priority

Priority number: 13

Event queueing

Events to be queued 1
□ Report event overflow into diagnostic buffer
□ Enable time error
Event threshold for time error 0

Fig. 5.24 Setting the overload behavior in the block properties

In the *Events to be queued* field, you can define how many events of the operating system will list in the associated queue and process in succession. The default value is 1, i.e. exactly one event is buffered. If the selected value is too large, an overload situation can be made worse if organization blocks with the same or lower priority cannot be processed in a timely manner. Under some circumstances, it may be better to discard the corresponding events and respond to it in the program of the organization block.

If the queue is full when an event occurs, it is counted and then discarded. If the attribute *Optimized block access* is activated, the number of discarded events is located in the *Event_Count* tag in the start information of the organization block.

If the checkbox *Report event overflow into diagnostics buffer* in the block properties is checked, the event ID DW#16#0002_3507 will be entered in the diagnostics buffer when an event occurs that leads to an overflow of the queue. Another diagnostics buffer entry with this event ID will only be made if all of the events in the queue have been processed and then a new overflow occurs.

If you activate the *Enable time error* checkbox, the operating system calls organization block OB 80 *Time error* if the number of the events pending in the queue reaches the number entered in the *Event threshold for time error* field. At the same time, the event ID DW#16#0002_ 3502 is entered in the diagnostics buffer. In the program of the time error organization block, you now have the capability of responding to an impending overflow of the queue.

Current start and interrupt information

Every organization block with standard access – the *Optimized block access* attribute is deactivated – contains information concerning the start event in the first 20 bytes of the temporary local data. You default structure of this start information can be found in Chapter 4.11.4 "Start information" on page 142. Organization blocks with the attribute *Optimized block access* activated can have start information, which is provided by the operating system in the *Input* declaration section. The specific start information of an organization block is described in the description of the organization block.

In many cases the interrupt-triggering component provides additional information which you can read in the interrupt organization block with the system function block RALRM (see Chapter 5.7.7 "Reading additional interrupt information" on page 210).

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 process image output 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. To this end, you can insert, for example, system blocks for process image partitions, which update the inputs before the program execution begins and transfer the outputs to the modules after the program execution ends (see section "Process image partitions" on page 180).

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-of-day interrupts

A time-of-day interrupt is executed at a configured time, either one-time or periodically (e.g. daily). A CPU 1500 provides 20 organization blocks with the numbers OB 10 to OB 17 and after OB 123 for processing a time-of-day interrupt.

A time-of-day interrupt organization block is assigned to the event class *Time of day*. It is of hardware data type *OB_Time of day*. The constant names and the values are listed in the *System constants* tab of the default tag table. The name of the constant can be changed in the block properties under *General*.

Start information

A time-of-day interrupt organization block with the attribute *Optimized block access* activated provides the start information shown in Table 5.8 in the *Input* declaration section. A time-of-day interrupt organization block with the attribute *Optimized block access* deactivated (OB with standard access) provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in 4.11.4 "Start information" on page 142. This contains the tag specified in Table 5.8 with the processing interval. This is the interval with which the organization block is processed (see PERIOD parameter of the system function SET_TINTL further below).

Table 5.8 Start information for a time-of-day interrupt organization block

Declaration	Tag name	Data type	Description
The *Optimized block access* attribute is activated:			
Input	Caughtup	BOOL	= "1" for caught up call because clock was set forward
Input	SecondTime	BOOL	= "1" for repeated call because clock was set back
The *Optimized block access* attribute is deactivated (standard access):			
Temp	PERIOD_EXE	WORD	Execution interval

Using a time-of-day interrupt

To start a time-of-day interrupt, you must first set the start time and then activate the time-of-day interrupt. You can carry out both activities separately in the block properties or also with system functions. Note that activation in the block properties means that the time-of-day interrupt is automatically started.

You can start a time-of-day interrupt once or periodically. The time-of-day interrupt is canceled following a single call of the time-of-day interrupt OB. You can also cancel an active time-of-day interrupt using CAN_TINT. If you wish to reuse a canceled time-of-day interrupt, you must set the start time again and activate the time-of-day interrupt. You can query the status of a time-of-day interrupt with QRY_TINT.

Execution of the time-of-day interrupt is disabled with DIS_IRT and EN_IRT and delayed with DIS_AIRT and EN_AIRT. You can set the behavior for time-of-day interrupts that follow each other too closely (overload behavior).

Configuring a time-of-day interrupt

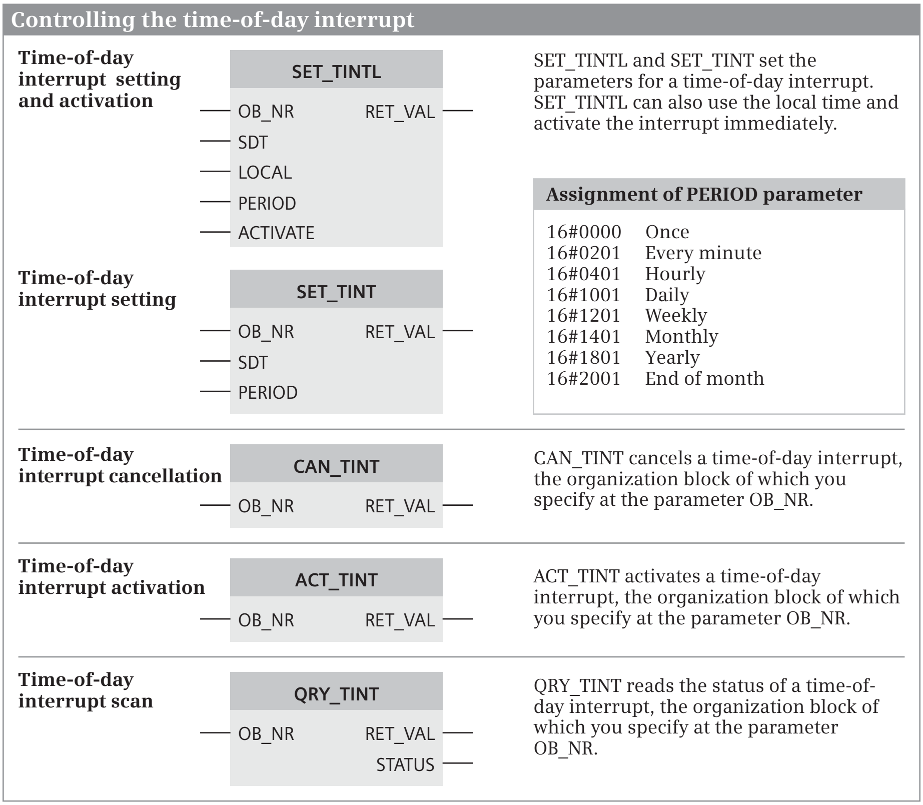
To configure a time-of-day interrupt, add an organization block with the event class *Time of day* and enter the name, programming language, and number. In addition to the general information and the attributes, you can set the following block properties in the properties of the organization block, under *Time of day interrupt*:

- ▷ Interval of execution: Never, Once, Every minute, Hourly, Daily, Weekly, Monthly, Yearly, and at the End of the month,
- ▷ Start date and Time of day,
- ▷ Time basis: System time or Local time.

You set the execution priority in the block attributes under *Priority*. You can change the default priority 2 from 2 to 24.

System functions for processing a time-of-day interrupt

You can use system functions to set, cancel, and activate a time-of-day interrupt and also to query the status. You can find the functions for the time-of-day interrupt in the program elements catalog under *Extended instructions* > *Interrupts*. Fig. 5.25 shows the graphic representation of the system functions.


Controlling the time-of-day interrupt
Time-of-day interrupt setting and activation
SET_TINTL
OB_NR
RET_VAL
SDT
LOCAL
PERIOD
ACTIVATE
Time-of-day interrupt setting
SET_TINT
OB_NR
RET_VAL
SDT
PERIOD
Time-of-day interrupt cancellation
CAN_TINT
OB_NR
RET_VAL
Time-of-day interrupt activation
ACT_TINT
OB_NR
RET_VAL
Time-of-day interrupt scan
QRY_TINT
OB_NR
RET_VAL
STATUS

SET_TINTL and SET_TINT set the parameters for a time-of-day interrupt. SET_TINTL can also use the local time and activate the interrupt immediately.

Assignment of PERIOD parameter	
16#0000	Once
16#0201	Every minute
16#0401	Hourly
16#1001	Daily
16#1201	Weekly
16#1401	Monthly
16#1801	Yearly
16#2001	End of month

CAN_TINT cancels a time-of-day interrupt, the organization block of which you specify at the parameter OB_NR.

ACT_TINT activates a time-of-day interrupt, the organization block of which you specify at the parameter OB_NR.

QRY_TINT reads the status of a time-of-day interrupt, the organization block of which you specify at the parameter OB_NR.

Fig. 5.25 System blocks for controlling the time-of-day interrupt

SET_TINTL determines the start time for a time-of-day interrupt. The parameter ACTIVATE specifies whether the start of the time-of-day interrupt OB should be carried out immediately (TRUE) or only when the function ACT_TINT is called (FALSE).

The start time is present in the SDT parameter in the format DATE_AND_LTIME, e.g. DTL#2011-01-01-08:30:00. The operating system ignores any specified seconds and milliseconds and sets these values to zero. For a monthly interval, only days 1 through 28 are possible start dates. When setting the start time, any old value of the start time is overwritten. A current time-of-day interrupt is canceled, i.e. the time-of-day interrupt must be activated again.

SET_TINT determines the start time for the time-of-day interrupt. SET_TINT only sets the start time; the time-of-day interrupt must be activated by ACT_TINT in order to start the time-of-day interrupt OB. The start time is present in the SDT parameter in the format DATE_AND_TIME, e.g. DT#2011-01-01-08:30:00. The operating system ignores any specified seconds and milliseconds and sets these values to zero. For a monthly interval, only days 1 through 28 are possible start dates. When setting the start time, any old value of the start time is overwritten. A current time-of-day interrupt is canceled, i.e. the time-of-day interrupt must be activated again.

CAN_TINT deletes a set start time and thus deactivates a time-of-day interrupt. The time-of-day interrupt OB is no longer called. If you wish to reuse this time-of-day interrupt, you must first set the start time again and then activate the time-of-day interrupt.

ACT_TINT activates a time-of-day interrupt. Activation is only possible if a time has been set for the time-of-day interrupt. ACT_TINT signals an error if the start time for a single start is in the past. In the case of a periodic start, the operating system calls the time-of-day interrupt OB at the next due time. A single time-of-day interrupt is quasi deleted following processing; you can set and activate it again (at a different start time).

QRY_TINT provides information on the status of a time-of-day interrupt. The STATUS parameter contains the desired information and the individual bits have the significance shown in Table 5.9.

Table 5.9 STATUS parameter of system function QRY_TINT

Bit	Meaning with signal state “0”	Meaning with signal state “1”
0	The CPU is in RUN.	The CPU is in STARTUP.
1	The interrupt is enabled.	The interrupt has been disabled by DIS_IRT.
2	The interrupt is not active or has expired.	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.
5	Always “0”	
6	The start time is based on the system time.	The start time is based on the local time.
Other	Always “0”	

Behavior during startup

During a warm restart, the operating system deletes all settings you have made using a system function. The settings parameterized in the block properties are retained.

You can obtain information in the startup program on the status of a time-of-day interrupt using QRY_TINT and cancel or reset and activate the time-of-day interrupt as required. Processing of a time-of-day interrupt organization block only takes place in the RUN operating state.

Error response

If the start time lies in the past for a one-time execution, the time-of-day interrupt will not be started.

If the start time for periodic processing lies in the past, the time-of-day interrupt will be started the next time it is due after the current time.

If the date for the monthly interval does not exist (e.g. February 30), the time-of-day interrupt will not be started.

If the clock is set forward by less than 20 s, any skipped time-of-day interrupt will be caught up. The *CaughtUp* tag is set in the start information during the caught up processing.

If the clock is set forward by 20 s or more and one or more time-of-day interrupts have been skipped, the time error organization block OB 80 is called for each priority class. The *Fault_ID* tag in the start information then has the value B#16#05 "Time-of-day interrupt expired due to time skip". If the time-of-day interrupt is not deleted in the time error OB, the first skipped time-of-day interrupt will be caught up and the *CaughtUp* tag will be set in the start information.

If the clock is set back by less than 20 s, a previously executed or still active time-of-day interrupt will not be repeated.

If the clock is set back by 20 s or more, all of the skipped time-of-day interrupts will be repeated. The start time of the first time-of-day interrupt to be repeated is recalculated if the time correction is longer than the period of this time-of-day interrupt. The *SecondTime* tag is set in the start information for a repeated time-of-day interrupt.

5.7.3 Time-delay interrupts

A time-delay interrupt implements a delay time independent of the timer functions and asynchronous to cyclic program execution. A CPU 1500 provides 20 organization blocks with the numbers OB 20 to OB 23 and after OB 123 for processing a time-delay interrupt.

A time-delay interrupt organization block is assigned to the event class *Time delay interrupt*. It is of hardware data type *OB_Delay*. The constant names and the values are listed in the *System constants* tab of the default tag table. The name of the constant can be changed in the block properties under *General*.

Start information

A time-delay interrupt organization block with the attribute *Optimized block access* activated provides the start information shown in Table 5.10 in the *Input* declaration section. A time-delay interrupt organization block with the attribute *Optimized block access* deactivated (OB with standard access) provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in 4.11.4 “Start information” on page 142. This contains the tags specified in Table 5.10.

Table 5.10 Start information for a time-delay interrupt organization block

Declaration	Tag name	Data type	Description
The *Optimized block access* attribute is activated:			
Input	Sign	WORD	Job ID (parameter SIGN from SRT_DINT)
The *Optimized block access* attribute is deactivated (standard access):			
Temp	SIGN	WORD	Job ID (parameter SIGN from SRT_DINT)
Temp	DTIME	TIME	Parameterized delay time (ms)

Using a time-delay interrupt

You start a time-delay interrupt by calling the system function SRT_DINT; this system function also passes on the delay interval and the delay organization block. When the time delay has expired, the organization block is called.

The time between the call of the SRT_DINT function and the start of the organization block is a maximum of one millisecond less than the parameterized delay time if 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. You can query the status of a time-delay interrupt with QRY_DINT.

Execution of the time-delay interrupt is disabled with DIS_IRT and EN_IRT and delayed with DIS_AIRT and EN_AIRT.

Configuring a time-delay interrupt

Configuration of the time-delay interrupts is carried out in two steps:

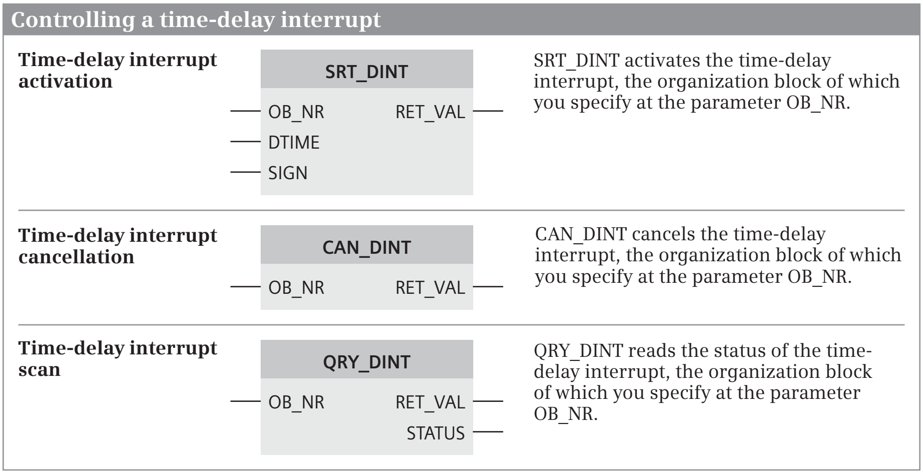
- ▷ You create an organization block for a time-delay interrupt.
- ▷ Then program the SRT_DINT function and possibly the CAN_DINT and QRY_DINT functions and assign the number of the time-delay interrupt OB to the OB_NR parameter.

To configure a time-delay interrupt, add an organization block with the event class *Time delay interrupt* and enter the name, programming language, and number. Set the priority in the properties of the organization block under *Attributes*. You can change the default priority 3 from 3 to 24.

Enter the function SRT_DINT into your program. Then click on the selection symbol in the input box of the OB_NR parameter and then select the time-delay interrupt OB from the list. You program the CAN_DINT and QRY_DINT functions in the same manner.

System functions for processing a time-delay interrupt

You can use system functions to activate and cancel a time-delay interrupt and also to query the status. You can find the functions for the time-delay interrupts in the program elements catalog under *Extended instructions* > *Interrupts*. Fig. 5.26 shows the graphic representation of the system functions.


Controlling a time-delay interrupt
Time-delay interrupt activation
SRT_DINT
OB_NR
RET_VAL
DTIME
SIGN
Time-delay interrupt cancellation
CAN_DINT
OB_NR
RET_VAL
Time-delay interrupt scan
QRY_DINT
OB_NR
RET_VAL
STATUS

SRT_DINT activates the time-delay interrupt, the organization block of which you specify at the parameter OB_NR.

CAN_DINT cancels the time-delay interrupt, the organization block of which you specify at the parameter OB_NR.

QRY_DINT reads the status of the time-delay interrupt, the organization block of which you specify at the parameter OB_NR.

Fig. 5.26 Start, cancel, and query a time-delay interrupt

SRT_DINT activates a time-delay interrupt. The call is simultaneously the start time for the parameterized period. Once the delay time has expired, the CPU calls the parameterized OB and transfers the job ID (configured in the SIGN parameter) in the start information for this OB. You can set the delay time in intervals of 1 ms. The accuracy of the delay time is also 1 ms.

Note that processing of the time-delay interrupt OB may be delayed if organization blocks of higher priority are being processed when the OB is called. You can overwrite a current delay time by a new value by calling SRT_DINT again. The new delay time then commences when called.

CAN_DINT cancels an activated time-delay interrupt. The parameterized organization block is not called in this case.

QRY_DINT provides information on the status of the time-delay interrupt. You select the time-delay interrupt using the OB number. The STATUS parameter contains the desired information and the individual bits have the significance shown in Table 5.11.

Table 5.11 STATUS parameter of system function QRY_DINT

Bit	Meaning with signal state "0"	Meaning with signal state "1"
0	The CPU is in RUN.	The CPU is in STARTUP.
1	The interrupt is enabled.	The interrupt has been disabled by DIS_IRT.
2	The interrupt is not active or has expired.	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"	

Behavior during startup

During a warm restart, 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 state 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.

Error response

If the time-delay interrupt OB is missing in the user program when called, the operating system ignores the event.

If the delay time has expired and the associated OB is still being processed, the operating system calls the organization block OB 80 *Time error* or ignores the event if OB 80 is not present.

5.7.4 Cyclic interrupts

A cyclic interrupt is an interrupt triggered at periodic intervals which 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. A CPU 1500 provides 20 organization blocks with the numbers OB 30 to OB 38 and after OB 123 for processing a cyclic interrupt.

A cyclic interrupt organization block is assigned to the *Cyclic interrupt* event class. It is of hardware data type *OB_Cyclic*. The constant names and the values are listed in the *System constants* tab of the default tag table. The name of the constant can be changed in the block properties under *General*.

Start information

A cyclic interrupt organization block with the attribute *Optimized block access* activated provides the start information shown in Table 5.10 in the *Input* declaration section. A cyclic interrupt organization block with the attribute *Optimized block access* deactivated (OB with standard access) provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in 4.11.4 "Start information" on page 142. This contains the tags specified in Table 5.10.

Table 5.12 Start information for a cyclic interrupt organization block

Declaration	Tag name	Data type	Description
The *Optimized block access* attribute is activated:			
Input	Initial_Call	BOOL	= "1" for the first call of the organization block in the RUN operating state and after reloading
Input	Event_Count	INT	Number of discarded start events since the last start of this organization block
The *Optimized block access* attribute is deactivated (standard access):			
Temp	PHS_OFFSET	INT	Phase offset *)
Temp	EXC_FREQ	INT	Parameterized time interval *)

*) for a cyclic interrupt cycle < 1 ms in microseconds (μs), otherwise milliseconds (ms)

Using a cyclic interrupt

For a cyclic interrupt that is configured with the hardware configuration, the start time for the call interval is the transition into the RUN operating state.

With the system function SET_CINT, the call interval (the cycle clock) and the phase offset can be changed per user program. You query the status of the cyclic interrupt with QRY_CINT.

Execution of the cyclic interrupt is disabled with DIS_IRT and EN_IRT and delayed with DIS_AIRT and EN_AIRT. You can set the behavior for cyclic interrupts that follow each other too closely (overload behavior).

Configuring a cyclic interrupt

To configure a cyclic interrupt, add an organization block with the event class *Cyclic interrupt* and enter the name, the programming language, the number, and the cycle clock (the call interval) in microseconds. In addition to the general information and the attributes, you can set the phase offset in the properties of the organization block, under *Cyclic interrupt*. You can also change the cycle clock here.

You set the execution priority in the block attributes under *Priority*. The default priority is 8 to 17, depending on the time interval, and can be changed in the range of 2 to 24.

Cycle clock

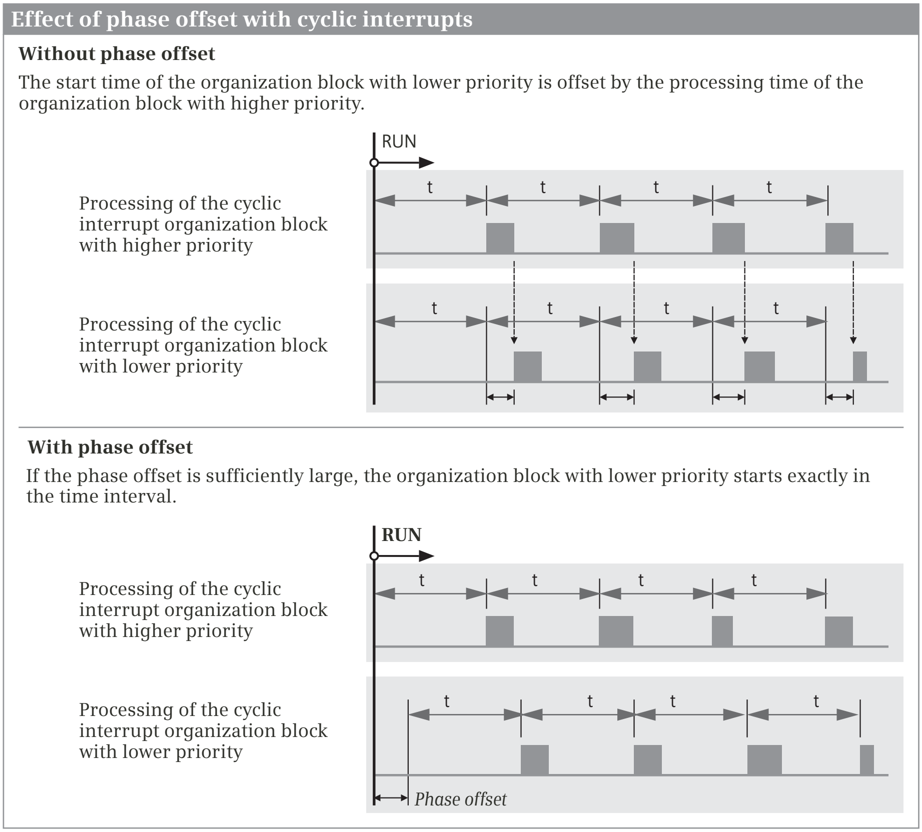
The cycle clock can be set in the range from 500 µs to 60,000,000 µs in intervals of 1 µs. When a cyclic interrupt organization block is added, the processing priority is preset depending on the cycle clock (Table 5.13). You can change the default priority in the block properties.

Table 5.13 Default setting of the priority depending on the cycle clock

Cycle clock (ms)	≤ 2	> 2 ... ≤ 5	> 5 ... ≤ 10	> 10 ... ≤ 100	> 100 ... ≤ 200	> 200 ... ≤ 500	> 500 ... ≤ 1000	> 1000 ... ≤ 2000	> 2000
Priority	17	16	14	13	12	11	10	9	8

Phase offset

You can use the phase offset to process cyclic interrupt programs in a precise time frame even if they have the same time interval or a common multiple thereof. This



Effect of phase offset with cyclic interrupts

Without phase offset

The start time of the organization block with lower priority is offset by the processing time of the organization block with higher priority.

With phase offset

If the phase offset is sufficiently large, the organization block with lower priority starts exactly in the time interval.

RUN
Processing of the cyclic interrupt organization block with higher priority
t
t
t
t
Processing of the cyclic interrupt organization block with lower priority
t
t
t
t
RUN
Processing of the cyclic interrupt organization block with higher priority
t
t
t
t
Processing of the cyclic interrupt organization block with lower priority
t
t
t
t
Phase offset

Fig. 5.27 Processing of cyclic interrupts with and without phase offset

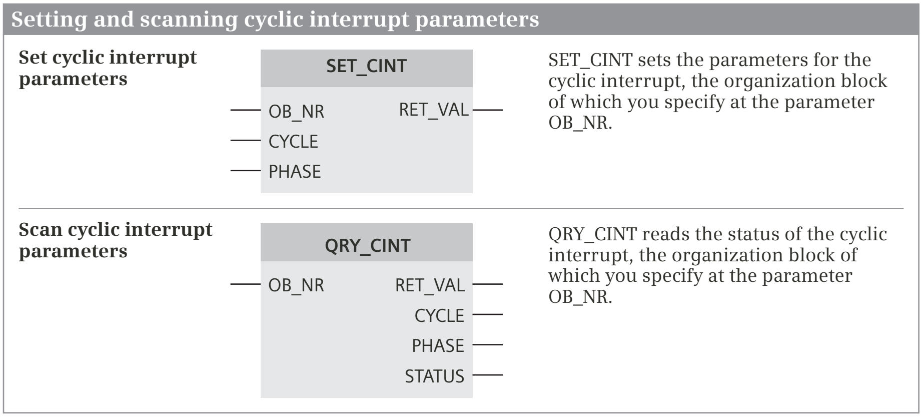
results in higher accuracy of the processing intervals. The phase offset can be set in the range from 0 µs to 15,000 µs in intervals of 1 µs.

The start time of the cycle clocks and the phase offset is the transition from the STARTUP operating state to RUN. The call instant for a cyclic interrupt OB is thus the cycle clock plus the phase offset. An example is shown in Fig. 5.27. No phase offset is set in the upper section, and consequently start of processing of the lower priority organization block is delayed by the current processing time of the higher priority organization block in each case.

If, on the other hand, a phase shift is configured and it is greater than the maximum processing time of the higher-priority organization block, the lower-priority organization block is processed in the precise time frame.

System functions for processing a cyclic interrupt

You can set and query the parameters for processing a cyclic interrupt with system functions. You can find the functions for the cyclic interrupt in the program elements catalog under *Extended instructions* > *Interrupts*. Fig. 5.28 shows the graphic representation of the functions.



Setting and scanning cyclic interrupt parameters

Set cyclic interrupt parameters

SET_CINT sets the parameters for the cyclic interrupt, the organization block of which you specify at the parameter OB_NR.

Scan cyclic interrupt parameters

QRY_CINT reads the status of the cyclic interrupt, the organization block of which you specify at the parameter OB_NR.

Fig. 5.28 System blocks for setting and querying the cyclic interrupt parameters

SET_CINT sets the parameters for a cyclic interrupt. This is the cycle clock with which the cyclic interrupt is triggered, and the phase offset. Enter the cycle clock 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 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 state of the selected cyclic interrupt organization block is output at parameter STATUS (Table 5.14).

Table 5.14 STATUS parameter of system function QRY_CINT

Bit	Meaning with signal state "0"	Meaning with signal state "1"
0	The CPU must be in the RUN mode.	The CPU must be 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"	

Behavior during startup

Processing of cyclic interrupts is not possible in the startup program. The cycle clocks only commence upon transition to the RUN state.

Error response

If the cyclic interrupt OB is missing in the user program when called, the operating system ignores the event.

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.

5.7.5 Hardware interrupts

With a hardware interrupt, there can be an immediate response with a corresponding program to events in the controlled process or on a module. A CPU 1500 provides 50 organization blocks with the numbers OB 40 to OB 47 and after OB 123 for processing a hardware interrupt.

A hardware interrupt organization block is assigned to the *Hardware interrupt* event class. It is of hardware data type *OB_HWINT*. The constant names and the values are listed in the *System constants* tab of the default tag table. The name of the constant can be changed in the block properties under *General*.

Start information

A hardware interrupt organization block with the attribute *Optimized block access* activated provides the start information shown in Table 5.15 in the *Input* declaration section. A hardware interrupt organization block with the attribute *Optimized block access* deactivated (OB with standard access) provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in 4.11.4 "Start information" on page 142. This contains the tags specified in Table 5.15.

Table 5.15 Start information for a time-of-day interrupt organization block

Declaration	Tag name	Data type	Description
The *Optimized block access* attribute is activated:			
Input	LADDR	HW_IO	Hardware identifier of the module triggering the interrupt
Input	USI	WORD	(ID for future expansions)
Input	ICHANNEL	USINT	Number of the channel triggering the interrupt
Input	EventType	BYTE	Event type (see module description)
The *Optimized block access* attribute is deactivated (standard access):			
Temp	I/O_FLAG	BYTE	I/O identification (16#54 = input, 16#55 = output)
Temp	MDL_ADDR	WORD	Module start address
Temp	POINT_ADDR	DWORD	Interrupt information

Using a hardware interrupt

A hardware interrupt is triggered on a module designed for this. This can be, for example, a digital or analog input module or a technology module. Triggering of a hardware interrupt is initially disabled by default. When you parameterize the module that triggers the interrupt using the hardware configuration, you enable the hardware interrupt event.

Only one hardware interrupt organization block can be assigned to a hardware interrupt event, but several events can be assigned to one hardware interrupt organization block.

At runtime, the assignment between a hardware interrupt event and an organization block can be made or removed in the user program using ATTACH and DETACH.

Execution of the hardware interrupt is disabled with DIS_IRT and EN_IRT and delayed with DIS_AIRT and EN_AIRT.

Configuring a hardware interrupt

To configure a hardware interrupt, enable the hardware interrupt event on the module that triggers the interrupt. Assign a hardware interrupt organization block to this event – either one that you have already created or one that you are creating now.

When parameterizing the module that triggers the interrupt with the hardware configuration, activate the hardware interrupt event, for example at the input of a correspondingly configured digital input module. Specify when a hardware interrupt is to be triggered, during a rising edge for example. The event is given a name, which you can change and which is entered in the *System constants* tab of the default tag table. You can use this name to address the hardware interrupt event in the user program, for example when using ATTACH to make an assignment to an organization block. In addition, you can change the default processing priority 16 from 2 to 24 in the module properties.

When assigning the organization block, choose an existing block or create a new one using the *Add object* button. When adding, give the organization block a name and set the programming language and number. In the properties of the organization block, under *Triggers*, you will find a table with the hardware interrupt events that were assigned during the hardware configuration.

Behavior during startup

During a warm restart, the operating system deletes all of the assignments made between an interrupt event and an organization block using a system function. The assignments configured with the hardware configuration are retained.

Interrupt handling commences with the transition to the RUN operating state. Hardware interrupts present during the transition are lost. Hardware interrupt organization blocks are only called in the RUN operating state.

Error response

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 hardware interrupt is lost. A new hardware interrupt is only acquired when processing of the old hardware interrupt has finished. If the event to which the same hardware 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 once again after processing the hardware interrupt OB.

If the hardware interrupt OB is missing in the user program when called, the operating system ignores the event.

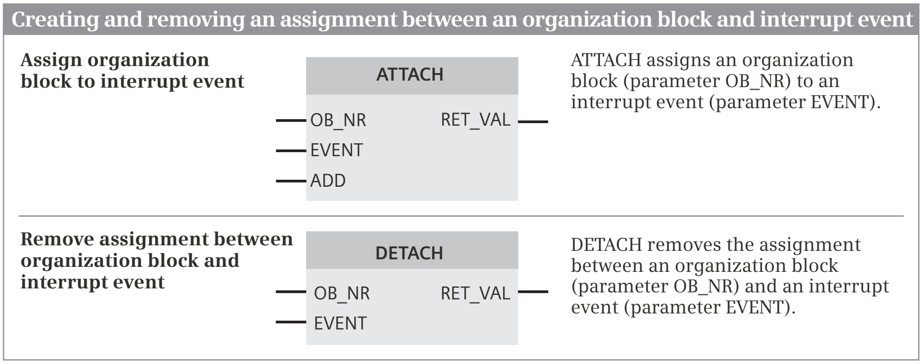
5.7.6 Assigning interrupts during runtime

With the following system 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

You find the system functions in the program elements catalog under *Extended instructions > Interrupts*. Calling of these functions is shown in Fig. 5.29.

At the parameter OB_NR you specify the number of the organization block, the hardware identifier (data type OB_HWINT) from the *System constants* tab in the default tag table, or an INT tag. At the parameter EVENT you specify the number or the name of the interrupt event (Data type Event_HwInt) from the *System constants* tab in the default tag table or a DWORD tag.



Creating and removing an assignment between an organization block and interrupt event

Assign organization block to interrupt event

ATTACH assigns an organization block (parameter OB_NR) to an interrupt event (parameter EVENT).

Remove assignment between organization block and interrupt event

DETACH removes the assignment between an organization block (parameter OB_NR) and an interrupt event (parameter EVENT).

Fig. 5.29 System blocks for the assigning hardware interrupts and canceling hardware interrupt assignments

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.

After the assignment has been made, the organization block is called and executed when the event occurs. The parameter ADD defines whether the previous assignments to other events will be retained (with “1” or TRUE) or whether they will be deleted (with “0” or FALSE).

The enable output ENO has signal state “0” for the following errors: OB does not exist (RET_VAL = 8090), OB is of the wrong type (RET_VAL = 8091), and 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 at the parameter EVENT, the assignment of this event is removed. If zero is assigned to the parameter EVENT, all of the assignments to the OB located at parameter OB_NR are deleted.

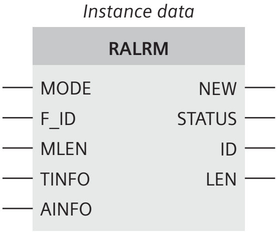
If the requested assignment does not exist, the enable output ENO has signal state “0” and the value 1 is output at the parameter RET_VAL. Further errors: OB does not exist (RET_VAL = 8090), OB is of the wrong type (RET_VAL = 8091), and event does not exist (RET_VAL = 8093).

5.7.7 Reading additional interrupt information

The system block RALRM reads additional interrupt information from the interrupt-triggering components (modules or submodules). It is called in an interrupt organization block or in a block called within this. Processing of RALRM is synchronous, i.e. the requested data is available at the output parameters immediately following the call. You find RALRM in the program elements catalog under *Extended instructions > Distributed I/O*. Fig. 5.30 shows the graphic representation of RALRM.

Reading additional interrupt information

Read additional interrupt information



RALRM reads additional information on an interrupt event.

RALRM: Assignment of MODE parameter	
0	Indicates the interrupt-triggering component in the ID parameter and sets NEW to signal state "1".
1	Writes all output parameters of RALRM.
2	Checks whether the component specified in the F_ID parameter has triggered the interrupt; if not, NEW is set to signal state "0", otherwise to signal state "1" and all output parameters are written.

Fig. 5.30 System blocks for reading additional interrupt information

RALRM can always be called in all organization blocks or execution levels for all events. If you call it in an organization block whose start event is not an interrupt from the I/O, correspondingly less information is available. Different information is entered in the destination areas specified by the TINFO and AINFO parameters depending on the respective organization block and the interrupt-triggering component.

In bytes 0 to 19, the destination area TINFO contains the complete (default) start information of the organization block in which RALRM was called, independent of the nesting depth in which it was called. The system block RALRM thus partially replaces the system function RD_SINFO. Address and management information is present in bytes 20 to 31, e.g. which component has triggered the interrupt.

In bytes 0 to 3 (bytes 0 to 25 with PROFINET), the destination area AINFO contains the header information, e.g. the number of received bytes of the additional interrupt information or interrupt type. Bytes 4 to 199 (bytes 26 to 1431 with PROFINET) contain the component-specific additional interrupt information itself.

The assignment of the MODE parameter determines the mode of the system block RALRM. With Mode = 0, the system block shows you the interrupt-triggering component in the ID parameter; NEW is assigned TRUE. With Mode = 1 all output parameters are written. With Mode = 2, check whether the component specified by the F_ID parameter was the interrupt-triggering one. If this applies, the NEW parameter has the value TRUE, and all other output parameters are written.

In order to work correctly, RALRM requires separate instance data for each call in the various organization blocks, e.g. a separate instance data block in each case.

5.8 Error handling

5.8.1 Causes of errors and error responses

The CPU can detect and signal errors in the program execution and from the modules. The response of the operating system depends on the type of error and on the configurable settings: The error is ignored, an error organization block is called, or it is left up to the user program to respond to the error. In the event of serious errors, e.g. the cycle monitoring time has elapsed twice in one program cycle, the CPU immediately goes into the STOP operating state.

Errors during runtime 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, a diagnostics alarm on the CPU display, an entry in the diagnostics buffer, or by starting the diagnostics interrupt (see Chapter 15.4 “Hardware diagnostics” on page 672). Using system blocks, you can respond to status and error messages of modules in the user program (see Chapter 5.9.3 “Diagnostic functions in the user program” on page 228).

System blocks that are prone to errors when executed report this error via the return value (function value), which is generally called the RET_VAL or STATUS. This feedback message can be evaluated in the user program and the error can be responded to. System blocks with an ENO output report a faulty execution in the block with signal state “0” or FALSE at this output. The ENO output can also be used for error reporting for self-written function blocks (FBs) and functions (FCs). Further details can be found in Chapters 7.6.4 “EN/ENO mechanism in the ladder logic” on page 320, 8.6.4 “EN/ENO mechanism in the function block diagram” on page 356, 9.6.2 “EN/ENO mechanism with SCL” on page 381, and 10.7.2 “EN/ENO mechanism in the statement list” on page 447.

Program execution errors can be programming errors, e.g. calling a non-existent block, and access errors, e.g. querying a non-existent peripheral input. The response to one of these so-called synchronous errors can be the system-internal error response (see Chapter 5.8.3 “Global error handling (synchronous error)” on page 215) or a user-specific error response (see Chapter 5.8.2 “Local error handling”). The operating system responds to errors are not related to the program execution (“asynchronous errors”) by calling an organization block (see Chapter 5.8.5 “Asynchronous errors” on page 220).

5.8.2 Local error handling

You can program local error handling in organization blocks (OB), function blocks (FB), and functions (FC). It only applies to the corresponding block. Local error handling is not taken from the calling block nor is it passed on to the called block. If the local error handling is not programmed, the system settings will apply if an error occurs (ignore error or STOP).

If local error handling is activated, the default responses are:

- For a write error: The error is ignored and program execution is continued.
- For a read error: The substitute value “0” or zero is read and program execution is continued.
- For an execution error: The execution of the faulty statement (function) is aborted and program execution is continued with the next statement.

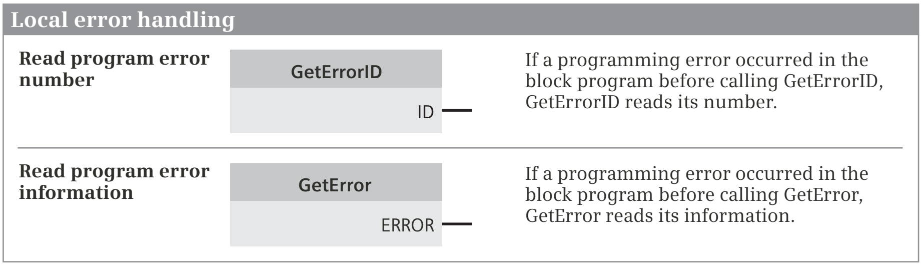
Local error handling is automatically activated by inserting the statement GET_ERROR or GET_ERROR_ID in the block and displayed in the block properties with the attribute *Handle errors within block* (cannot be edited).

Evaluating program errors

Two functions are available in the block for the error evaluation for local error handling (Fig. 5.31):

- GET_ERR_ID (read program error number) provides the error number (ID) in the event of a program execution error.
- GET_ERROR (read program error information) provides the corresponding information in a predefined data structure in the event of a program execution error.

In the event of a program execution error, the CPU enters the error into the diagnostics buffer by default and switches to STOP. If the function GET_ERROR or GET_ERR_ID is programmed in the block, there is no entry into the diagnostics buffer and no switch to STOP. Instead, the error is reported via GET_ERROR or GET_ERR_ID.



If a programming error occurred in the block program before calling GetErrorID, GetErrorID reads its number.

If a programming error occurred in the block program before calling GetError, GetError reads its information.

Fig. 5.31 System blocks for local error handling

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 GET_ERROR or GET_ERR_ID, the call is preferably positioned in the last network or at the end of the program in the monitored block.

GET_ERROR and GET_ERR_ID can also be called multiple times. A call of GET_ERROR or GET_ERR_ID re-initiates the error detection. The next call of GET_ERROR or GET_ERR_ID outputs the first error after the previous call of GET_ERROR or GET_ERR_ID. The progress of the error 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 5.16). 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).

Table 5.16 Error numbers with program execution errors

ERROR_ID		Error	ERROR_ID		Error
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 write-protected)	16#2943	10563	Write error during direct access (output channel does not exist)
16#253A	9530	Global DB does not exist			

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 4.11.3 “Data type ERROR_STRUCT” on page 141. 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 output according to their priority (Table 5.17). Priority 1 is the highest priority, 12 is the lowest.

Table 5.17 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

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.

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.

5.8.3 Global error handling (synchronous error)

The CPU's operating system generates a synchronous error event if an error occurs in direct relationship with the program execution. Two types of error are distinguished: programming error and I/O access error. If local error handling is not activated, the CPU operating system responds to a programming error with the call of the organization block OB 121 and to an I/O access error with the call of the organization block OB 122.

Programming error organization block OB 121

The organization block OB 121 is called if a programming error occurs (also in the STARTUP operating state). This includes, for example, BCD conversion errors, errors with indirect addressing, and addressing of missing SIMATIC timer/counter functions or blocks. If the organization block OB 121 is not present when a programming error occurs, the CPU switches to STOP.

The programming error organization block is assigned to the *Programming error* event class. It is of hardware data type OB_ANY. The constant names and the values are listed in the *System constants* tab of the default tag table. The name of the constant can be changed in the block properties under *General*.

Start information OB 121

The programming error organization block with the attribute *Optimized block access* activated provides the start information shown in Table 5.18 in the *Input* declaration section. With the attribute *Optimized block access* deactivated (OB with standard access), it provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in 4.11.4 "Start information" on page 142. This contains the tags specified in Table 5.18, which give information on the cause and location of the error. Example: If the SW-FLT tag is occupied by B#16#32 (= access to a non-existent global data block), the FLT_REG tag contains the number of the missing data block.

I/O access error organization block OB 122

The organization block OB 122 is called if an I/O access error occurs (also in the STARTUP operating state). This is the case, for example, if a faulty module, a non-existent module, or an I/O address unknown on the CPU is accessed. If the organization block OB 122 is not present when an I/O access error occurs, the operating system ignores the error event.

The I/O access error organization block is assigned to the *IO access error* event class. It is of hardware data type OB_ANY. The constant names and the values are listed in the *System constants* tab of the default tag table. The name of the constant can be changed in the block properties under *General*.

Start information OB 122

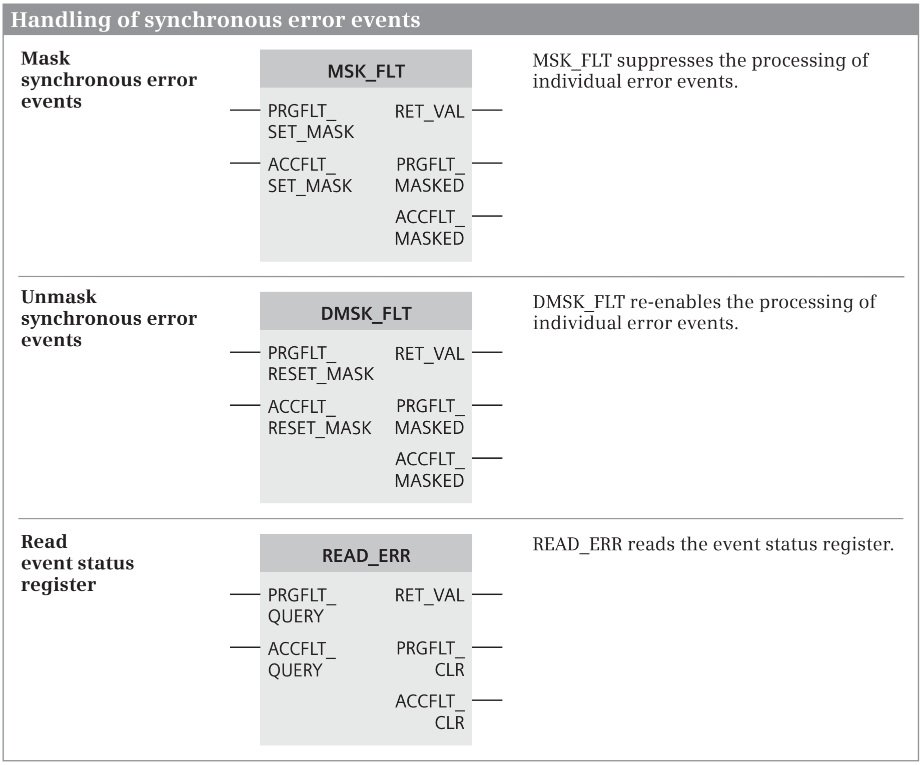
The programming error organization block with the attribute *Optimized block access* activated provides the start information shown in the Table 5.18 on page 217 in the *Input* declaration section. With the attribute *Optimized block access* deactivated (OB with standard access), it provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in 4.11.4 "Start information" on page 142. This contains the tags specified in Table 5.18, which give information on the cause and location of the error. In the SW_FLT tag, the value B#16#42 stands for a read operation and B#16#43 for a write operation. The type of access in the MEM_AREA tag can be bit access (with value 0), byte access (1), word access (2), and doubleword access (3). The memory area can be the I/O area I:P or Q:P (with value 0), the process image input, (1) or the process image output (2). The error-causing memory address is then in the MEM_ADDR tag.

Table 5.18 Start information for OB 121 and OB 122

Decl.	Tag name	Data type	Description	121	122
The *Optimized block access* attribute is activated:					
Input	BlockNo	UINT	Number of block in which the programming or I/O access error occurred	×	×
	Reaction	USINT	Reaction to error		
			0: Ignore error	×	×
			1: Replace faulty value	×	×
			2: Skip statement	×	×
			3: Programmed error handling	×	–
	Fault_ID	BYTE	Error code	×	×
	BlockType	USINT	Block type (OB: 16#88, FB: 16#8C, FC: 16#8E)	×	×
	Area	USINT	Operand area in which the erroneous access was located		
			Process image input: 16#01	×	–
			Process image output: 16#02	×	–
			Technology DB: 16#03	×	–
			Inputs: 16#81	×	×
			Outputs: 16#82	×	×
			Bit memory: 16#83	×	–
			Data (block): 16#84, 85, 8A, 8B	×	–
			Local data: 16#40 to 4E, 86, 87, 16#8E, 8F, C0 to CE	×	–
	DBNo	DB_ANY	DB number if AREA = 16#84, 85, 8A, 8B	×	–
	Csg_OBNo	OB_ANY	OB number (121 or 122)	×	×
	Csg_Prio	USINT	OB priority	×	×
	Width	USINT	Width of the access Bit: 16#00 Byte: 16#01 Word: 16#02 Doubleword: 16#03 Long word: 16#04	×	×
The *Optimized block access* attribute is deactivated (standard access):					
Temp	SW_FLT	BYTE	Start request for the organization block (error code)	×	×
	BLK_TYPE	BYTE	Type of block in which the error occurred	×	×
	FLT_REG	WORD	Error source depending on the error code	×	–
	MEM_AREA	BYTE	Memory area (bits 0 to 3) and type of access (bits 4 to 7)	–	×
	MEM_ADDR	WORD	Memory address at which the error occurred	–	×
	BLK_NUM	WORD	Number of block in which the error occurred	×	×
	PRG_ADDR	WORD	Relative address of the machine code causing the error	×	×

5.8.4 Enabling and disabling synchronous error processing

The processing of a synchronous error event can be disabled (“masked”), so that the error organization block is not called when the event occurs. An error mask defines which synchronous error events will be disabled. If an event occurs during the disabled state, it is recorded in the event status register. The processing of disabled synchronous error events can also be enabled again (“unmasked”). Fig. 5.32 shows the graphic representation of the system functions provided for this purpose.



MSK_FLT suppresses the processing of individual error events.

DMSK_FLT re-enables the processing of individual error events.

READ_ERR reads the event status register.

Fig. 5.32 System blocks for handling of synchronous error events

Error masks

The synchronous error processing is influenced via two error masks. A bit is present in the programming error mask for each detected programming error, and in the access error mask for each detected access error. When specifying the error mask, you set the bit which corresponds to the synchronous error you wish to mask, unmask, or query. The error masks returned by the system functions indicate the synchronous errors which are still masked or present by signal state “1”. Table 5.19 shows the assignment of the synchronous error masks.

Table 5.19 Assignment of the synchronous error masks

Programming error mask			
Bit	**Meaning**	**Bit**	**Meaning**
0	–	16	Write error global data block
1	BCD conversion error	17	Write error instance data block
2	Area length error when reading	18	Faulty number in DB register
3	Area length error when writing	19	Faulty number in DI register
4	Area error when reading	20	Faulty number of an FC block
5	Area error when writing	21	Faulty number of a function block
6	Faulty number of a timer function	22	–
7	Faulty number of a counter function	23	–
8	Address error when reading indirectly	24	–
9	Address error when writing indirectly	25	–
10	–	26	Data block does not exist
11	–	27	–
12	–	28	Called FC block does not exist
13	–	39	–
14	–	30	Called function block does not exist
15	–	31	–
Access error mask			
Bit	**Meaning**	**Bit**	**Meaning**
1	I/O access error when reading	2	I/O access error when writing

MSK_FLT Mask synchronous error events

By means of the error masks, the MSK_FLT disables calling of the synchronous error organization blocks. By means of signal state "1" you identify in the error masks for which synchronous errors the organization blocks are not to be called (the synchronous error events are "masked"). The specified masking is used in addition to the masking saved in the operating system. MSK_FLT signals in the function value whether a (saved) masking was already present (16#0001) for at least one bit for the masking specified in the input parameters.

MSK_FLT returns all currently masked events with signal state "1" in the output parameters.

If a masked synchronous error event occurs, the corresponding organization block is not called and the event is not entered in the event status register. Masking applies to the current priority class. If you mask the call of a synchronous error organization block in the main program, for example, the synchronous error organization block is nevertheless called if the error occurs in an interrupt routine.

DMSK_FLT Unmask synchronous error events

By means of the error masks, the DMSK_FLT enables calling of the synchronous error organization blocks. By means of signal state "1" you identify in the error masks the synchronous errors for which the organization blocks are to be called again (the synchronous error events are "unmasked"). The entries in the event status register corresponding to the specified unmasking are deleted. DMSK_FLT signals with W#16#0001 in the function value if no (saved) masking was present for at least one bit for the unmasking specified in the input parameters.

DMSK_FLT returns all currently masked events with signal state "1" in the output parameters.

If an unmasked synchronous error event occurs, the corresponding organization block is called and the event is entered in the event status register. Enabling applies to the current priority class.

READ_ERR Read event status register

READ_ERR reads the event status register. With signal state "1" you identify in the error masks the synchronous errors for which you wish to read the entries. READ_ERR signals with W#16#0001 in the function value if no (saved) masking was present for at least one bit for the selection specified in the input parameters.

READ_ERR returns the selected events with signal state "1" in the output parameters when they have occurred and deletes these events in the event status register when scanned. Synchronous errors which have occurred in the current priority class are signaled.

5.8.5 Asynchronous errors

Asynchronous errors are errors which can occur asynchronously to program execution. If an asynchronous error occurs, the operating system calls one of the following organization blocks:

OB 80	Time error
OB 82	Diagnostics interrupt
OB 83	Insert/remove module interrupt
OB 86	Rack error

The organization block OB 82 (diagnostics interrupt) is described in Chapter 5.9.1 "Diagnostics interrupt" on page 226.

Calling these asynchronous error organization blocks can be disabled and enabled using the system functions DIS_IRT and EN_IRT and delayed and enabled using DIS_AIRT and EN_AIRT.

Time error OB 80

The operating system calls the organization block OB 80 if one of the following errors occurs:

- The first instance of exceeding the cycle monitoring time in a program cycle
- OB request error (the called organization block is still being processed, or an organization block is called too frequently within a priority class)

- ▷ Time-of-day error interrupt (expired time-of-day interrupt through setting the time forward by more than 20 s or after transitioning to the RUN operating state)

If OB 80 is not present, the CPU switches to STOP in the event that the cycle monitoring time is exceeded. A different error event is ignored if the time error organization block is not present. The CPU switches to STOP even if the time error organization block is present if the cycle time is exceeded a second time in the same program cycle.

To program the error program, add the organization block with the event class *Time error interrupt* and enter the name, programming language, and number. The default priority 22 cannot be changed. The time error organization block is of the hardware data type OB_TIMEERROR. The constant name and the value are listed in the *System constants* tab of the default tag table. The name of the constant can be changed in the block properties under *General*.

The **Start information** of the time error organization block with the attribute *Optimized block access* activated contains the tags shown in Table 5.20 in the *Input* declaration section. The time error organization block with the attribute *Optimized block access* deactivated (OB with standard access) provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in Chapter 4.11.4 “Start information” on page 142. This contains the tags specified in Table 5.20.

Table 5.20 Start information for the time error organization block

Declaration	Tag name	Data type	Description
The *Optimized block access* attribute is activated:			
Input	Fault_ID	BYTE	Error code
	Csg_OBNo	OB_ANY	Number of the organization block causing the error
	Csg_Prio	UINT	Priority of the organization block causing the error
The *Optimized block access* attribute is deactivated (standard access):			
Temp	FLT_ID	BYTE	Error code
	ERR_EV_CLASS	BYTE	Error-triggering event class
	ERR_EV_NUM	BYTE	Error-triggering event number
	OB_PRIORITY	BYTE	Error information depending on the error code
	OB_NUM	BYTE	Error information depending on the error code

Insert/remove module interrupt OB 83

If a configured and activated module of the distribute I/O is inserted or removed, the operating system triggers an insert/remove module event and calls organization block OB 83 (also in the STARTUP operating state). The insert/remove module event is ignored if OB 83 is not present.

Inserting or removing a centrally arranged module always leads to a STOP of the CPU.

To program the error program, add the organization block with the event class *Pull or plug of modules* and enter the name, programming language, and number. Set the priority in the properties of the organization block under *Attributes*. You can change the default priority 6 in the range from 2 to 26. The insert/remove module organization block is of the hardware data type OB_ANY. The constant name and the value are listed in the *System constants* tab of the default tag table. The name of the constant can be changed in the block properties under *General*.

The **Start information** of the insert/remove module organization block with the attribute *Optimized block access* activated contains the tags shown in Table 5.21 in the *Input* declaration section. The insert/remove module organization block with the attribute *Optimized block access* deactivated (OB with standard access) provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in Chapter 4.11.4 “Start information” on page 142. This contains the tags specified in Table 5.21.

Table 5.21 Start information for an insert/remove module organization block

Declaration	Tag name	Data type	Description
The *Optimized block access* attribute is activated:			
Input	LADDR	HW_IO	Hardware identifier of the module
	Event_Class	BYTE	Event: B#16#38 : Module was inserted B#16#39 : Module was removed
	Fault_ID	BYTE	Error ID
The *Optimized block access* attribute is deactivated (standard access):			
Temp	FLT_ID	BYTE	Error code
	MDL_ID	BYTE	Interrupt-triggering I/O area B#16#54 : Peripheral inputs B#16#55 : Peripheral outputs
	MDL_ADDR	WORD	Start address of interrupt-triggering module
	RACK_NUM	INT	Number of the distributed station
	MDL_TYPE	WORD	Type of the interrupt-triggering module

Rack error OB 86

The operating system calls the organization block OB 86 if

- a DP master system or a PROFINET IO system fails or becomes available again,
- a distributed station (DP slave or IO device) fails or returns, and
- a distributed station (DP slave or IO device) is activated with the system function D_ACT_DP with MODE = 3 or deactivated with MODE = 4.

OB 86 is also called if one of the above-mentioned events occurs in the STARTUP operating state. The CPU ignores these events if OB 86 is not present.

To program the error program, add the organization block with the event class *Rack or station failure* and enter the name, programming language, and number. Set the priority in the properties of the organization block under *Attributes*. You can change the default priority 6 in the range from 2 to 26. The OB 86 is of the hardware data type OB_ANY. The constant name and the value are listed in the *System constants* tab of the default tag table. The name of the constant can be changed in the block properties under *General*.

The **Start information** of OB 86 with the attribute *Optimized block access* activated contains the tags shown in Table 5.22 in the *Input* declaration section. OB 86 with the attribute *Optimized block access* deactivated (OB with standard access) provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in Chapter 4.11.4 "Start information" on page 142. This contains the tags specified in Table 5.22.

Table 5.22 Start information for the rack error organization block

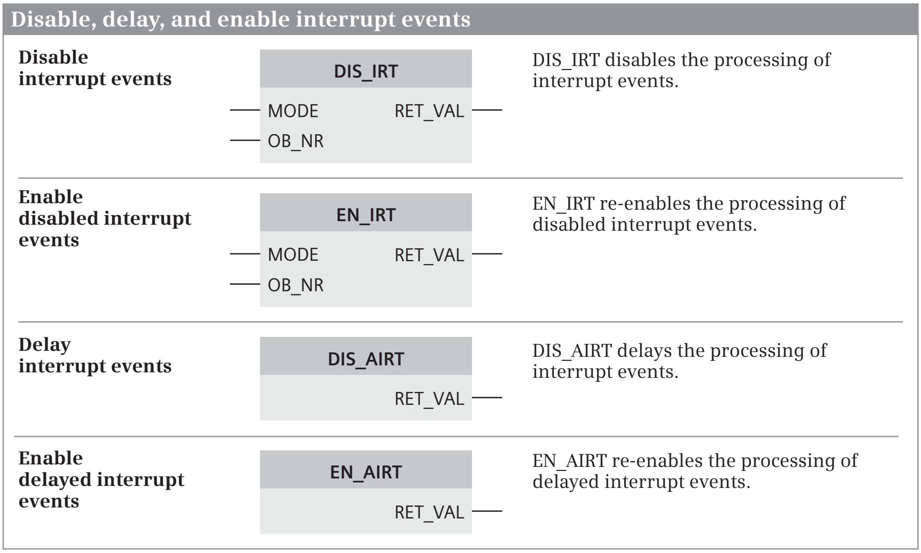
Declaration	Tag name	Data type	Description
The *Optimized block access* attribute is activated:			
Input	LADDR	HW_IO	Hardware identifier
	Event_Class	BYTE	Event: B#16#38 : Module was inserted B#16#39 : Module was removed
	Fault_ID	BYTE	Error ID
The *Optimized block access* attribute is deactivated (standard access):			
Temp	FLT_ID	BYTE	Error code
	MDL_ADDR	WORD	Depending on the error code
	RACKS_FLTD	ARRAY[0..31] of BOOL	Depending on the error code

5.8.6 Disable, delay, and enable interrupts and asynchronous errors

The processing of an interrupt or an asynchronous error organization block can be disabled so that no response occurs to the interrupt or asynchronous error. After enabling they are processed again. The processing can also be delayed so that it is processed after the enable. Fig. 5.33 shows the graphic representation of the system functions.

DIS_IRT Disable interrupt events

DIS_IRT disables the processing of new interrupt events and asynchronous error events. All new interrupts and asynchronous errors are rejected. If an interrupt or



DIS_IRT disables the processing of interrupt events.

EN_IRT re-enables the processing of disabled interrupt events.

DIS_AIRT delays the processing of interrupt events.

EN_AIRT re-enables the processing of delayed interrupt events.

Fig. 5.33 System blocks for handling interrupt events

asynchronous error occurs following disabling, the associated organization block is no longer processed; if the organization block does not exist, the CPU ignores the event.

Disabling of processing applies to all priority classes until canceled again by EN_IRT. The processing of all interrupts and asynchronous errors is enabled again following a warm restart.

You can use the MODE and OB_NR parameters to specify which interrupts and asynchronous errors are to be disabled (Table 5.23). Depending on the assignment of the MODE parameter, the disabled interrupt events are also entered into the diagnostics buffer (MODE = B#16#0x) or not (MODE = B#16#8x) when they occur.

EN_IRT Enable disabled interrupt events

EN_IRT enables processing of the interrupt events and asynchronous error events which had been disabled by DIS_IRT. Following enabling, the associated organization block is processed if an interrupt or asynchronous error occurs; if the OB does not exist, the preset system response takes place.

You can use the MODE and OB_NR parameters to specify which interrupts and asynchronous errors are to be enabled (Table 5.23).

DIS_AIRT Delay interrupt events

Following calling of DIS_AIRT, the program in the current organization block (in the current priority class) is not interrupted by an interrupt event of higher priority. The interrupts are processed with a delay, i.e. the operating system saves the

Table 5.23 Assignment of MODE parameter with DIS_IRT and EN_IRT

MODE	Meaning with DIS_IRT	Meaning with EN_IRT
B#16#00	All newly occurring interrupt events are disabled.	All newly occurring interrupt events are enabled.
B#16#01	The newly occurring interrupt events of an interrupt class are disabled.	The newly occurring interrupt events of an interrupt class are enabled.
B#16#02	The newly occurring interrupt events of an interrupt are disabled.	The newly occurring interrupt events of an interrupt are enabled.
B#16#80	All newly occurring interrupt events are disabled without entry into the diagnostics buffer.	–
B#16#81	The newly occurring interrupt events of an interrupt class are disabled without entry into the diagnostics buffer.	–
B#16#82	The newly occurring interrupt events of an interrupt are disabled without entry into the diagnostics buffer.	–

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

EN_AIRT 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.

5.9 Diagnostics in the user program

System diagnostics is the detection, evaluation, and reporting of errors that occur within the programmable controller. Some examples of such errors are errors on modules or a wire break for input signals.

This chapter describes how a program can respond to a diagnostic event. Further possibilities offered by the programming device in online mode are described in Chapter 15.4 “Hardware diagnostics” on page 672.

5.9.1 Diagnostics interrupt

A diagnostics interrupt is triggered if the diagnostics status of a correspondingly configured module changes, such as a missing power supply for signal modules or overflow and underflow for analog input modules. A CPU 1500 provides the organization block OB 82 for processing a diagnostics interrupt.

OB 82 is also called if the diagnosis event occurs in the STARTUP operating state. If OB 82 is not present when a diagnosis event occurs, the CPU ignores the diagnosis event. The occurrence of a diagnosis event is entered in the diagnostics buffer.

Start information

The diagnostics interrupt organization block with the attribute *Optimized block access* activated provides the start information shown in Table 5.24 in the *Input* declaration section. With the attribute *Optimized block access* deactivated (standard access), the diagnostics interrupt OB provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in 4.11.4 “Start information” on page 142. This contains the tags specified in Table 5.24.

Table 5.24 Start information for the diagnostics interrupt organization block

<table>
<tr><th>Declaration</th><th>Tag name</th><th>Data type</th><th>Description</th></tr>
<tr><td colspan="4"><b>The <i>Optimized block access</i> attribute is activated:</b></td></tr>
<tr><td>Input</td><td>IO_State</td><td>WORD</td><td>Diagnostics status of the hardware object
<table>
<tr><th>Bit</th><th>Meaning for “1”</th><th>Bit</th><th>Meaning for “1”</th></tr>
<tr><td>0</td><td>Good</td><td>4</td><td>Error</td></tr>
<tr><td>1</td><td>Deactivated</td><td>5</td><td>Not accessible</td></tr>
<tr><td>2</td><td>Need for maintenance</td><td>6</td><td>Qualified</td></tr>
<tr><td>3</td><td>Maintenance request</td><td>7</td><td>Not available</td></tr>
</table></td></tr>
<tr><td></td><td>LADDR</td><td>HW_ANY</td><td>Hardware identifier of the object triggering the interrupt</td></tr>
<tr><td></td><td>Channel</td><td>UINT</td><td>Channel number</td></tr>
<tr><td></td><td>MultiError</td><td>BOOL</td><td>With signal state “1”, more than one diagnosis event is present</td></tr>
<tr><td colspan="4"><b>The <i>Optimized block access</i> attribute is deactivated (standard access):</b></td></tr>
<tr><td>Temp</td><td>IO_FLAG</td><td>BYTE</td><td>I/O identifier
(B#16#54 = input, B#16#55 = output)</td></tr>
<tr><td></td><td>MDL_ADDR</td><td>WORD</td><td>Start address of interrupt-triggering module</td></tr>
<tr><td></td><td>&lt;Byte 8 ... 11&gt;</td><td>BOOL</td><td>Error messages</td></tr>
</table>

Configuring a diagnostics interrupt

Diagnostics interrupts are deactivated by default. You activate the diagnostics interrupt during the parameterization of a diagnostics-capable module with the hardware configuration.

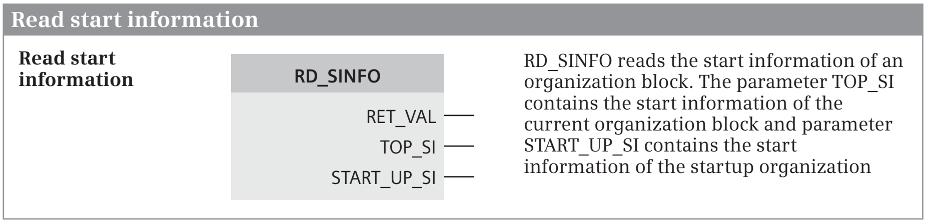
To program the diagnostics program, add the organization block with the event class *Diagnostic error interrupt* and enter the name and the programming language. Set the priority in the properties of the organization block under *Attributes*. You can change the default priority 5 in the range from 2 to 26.

The diagnostics interrupt organization block is of the hardware data type OB_DIAG. The constant name and the value are listed in the *System constants* tab of the default tag table. The name of the constant can be changed in the block properties under *General*.

5.9.2 Read start information

RD_SINFO Read start information

RD_SINFO provides the start information of the current organization block – this is the organization block at the top of the call tree – and also that of the last executed startup organization block on a lower call level (Fig. 5.34).



Read start information

Read start information

RD_SINFO reads the start information of an organization block. The parameter TOP_SI contains the start information of the current organization block and parameter START_UP_SI contains the start information of the startup organization

Fig. 5.34 Read start information RD_SINFO

Calling of RD_SINFO is not only permissible at any position within the main program but also in each priority class, including the program of an error organization block or in the startup. For example, if RD_SINFO is called in an interrupt organization block, TOP_SI contains the start information of the interrupt OB. TOP_SI and START_UP_SI have identical contents when calling in the startup.

For the information transfer to the parameters TOP_SI and START_UP_SI, there are the data structures listed in Table 5.24. Table 5.7 on page 194 shows the organization blocks that belong to the event class. The components of the individual structures are described in the online help for RD_SINFO. SI_classic corresponds to the start information as it is provided by an organization block with the attribute *Optimized block access* deactivated (byte 0 to 11).

Table 5.25 System data types for RD_SINFO (parameter TOP_SI)

The system data type	provides information for the event class	The system data type	provides information for the event class
SI_classic *)	All	SI_SynchCycle	Synchronous cycle
SI_none *)	–	SI_TimeError	Time error interrupt
SI_ProgramCycle	Program cycle	SI_DiagnosticInterrupt	Diagnostic error interrupt
SI_TimeOfDay	Time of day	SI_PlugPullModule	Pull or plug of modules
SI_Delay	Time delay interrupt	SI_StationFailure	Rack or station failure
SI_Cyclic	Cyclic interrupt	SI_Servo	MC-Servo
SI_HWInterrupt	Hardware interrupt	SI_Ipo	MC-Interpolator
SI_Submodule	Status Update Profile	SI_Startup *)	Startup
		SI_ProgIOAccessError	Programming error IO access error

*) Also permitted for the START_UP_SI parameter

For the application, create a tag in the local data of the code block or in a global data block and specify the name of the data structure as data type. Create this tag at the parameter TOP_SI or START_UP_SI as an actual parameter.

5.9.3 Diagnostic functions in the user program

The following functions are available to evaluate diagnostic data in the user program:

- LED Read status of an LED
- GET_DIAG Read diagnostic information
- GEN_DIAG Generate diagnostic information
- GET_NAME Read name of an IO device
- GetStationInfo Read out information of an IO device
- DeviceStates Read status of distributed I/O stations
- ModuleStates Read status of distributed I/O modules
- Get_IM_Data Read I&M data

Common LADDR parameter

The parameter LADDR specifies the hardware object to be addressed via the hardware identifier (hardware identifier, see Chapter 4.4 “Addressing of hardware objects” on page 107). You obtain the name and the value of the hardware identifier either from the *System constants* tab in the default tag table or, for a selected object, the object properties in the inspector window under *General* > *Project information* > *Name* or under *[Object type]* > *Hardware identifier*. You specify this name or value at the LADDR parameter.

Common DONE, BUSY, ERROR and STATUS parameters

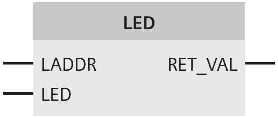
Asynchronously working system blocks cannot immediately end the task at the first call. The parameters DONE, BUSY, ERROR and STATUS provide information about the progress of the job processing: The job is being processed if BUSY = “1”. With DONE = “1” and ERROR = “0”, the job has been completed without errors; with DONE = “1” and ERROR = “1”, the job has been completed with one error. The error is then specified at the STATUS parameter.

LED Read status of an LED

LED reads the status of a module LED. The parameter LADDR specifies the CPU or the interface and the parameter LED specifies the light-emitting diode. RET_VAL indicates the current status of the specified LED. Fig. 5.35 shows the function call.

Read status of an LED

Read status of a light-emitting diode



LED reads the status of a light-emitting diode.

Parameter LED

Value	Light-emitting diode
1	STOP/RUN
2	ERROR
3	MAINT
4	Redundant
5	Link (green)
6	Rx/Tx (yellow)

Parameter RET_VAL

Value	Status of light-emitting diode
0	LED does not exist
1	Permanently switched off
2	Color 1 permanently switched on (e.g. green for STOP/RUN LED)
3	Color 2 permanently switched on (e.g. orange for STOP/RUN LED)
4	Color 1 flashes at 2 Hz
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. 5.35 Read status of an LED

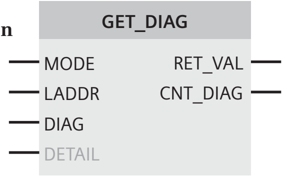
GET_DIAG Read diagnostic information

GET_DIAG reads the diagnostic information of a hardware object. You specify the hardware identifier at the LADDR parameter. With the MODE parameter, you select the type of diagnostic information that is output at the DIAG parameter. Fig. 5.36 shows the function call.

Via the MODE parameter you select the type of information to be output at the DIAG parameter. With MODE = 0 you 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 sup-

Read diagnostic information

Read diagnostic information



GET_DIAG reads the diagnostic information of the hardware object (e.g. of a module), the identifier of which is specified at the LADDR parameter.

Parameter MODE

Value	Meaning
0	Output of the diagnostic information supported by the hardware object to DIAG
1	Output of own diagnostics status to DIAG
2	Output of the diagnostics status of all subordinate modules to DIAG and output of the module status information to DETAIL
3	Output of the I/O status to DIAG, output of the number of additionally output details to CNT_DIAG, and output of the channel statuses to DETAIL

The diagnostic information output at the DIAG and DETAIL parameters depends on the selected hardware object (see operating instructions).

Fig. 5.36 Read diagnostic information

ported. 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.

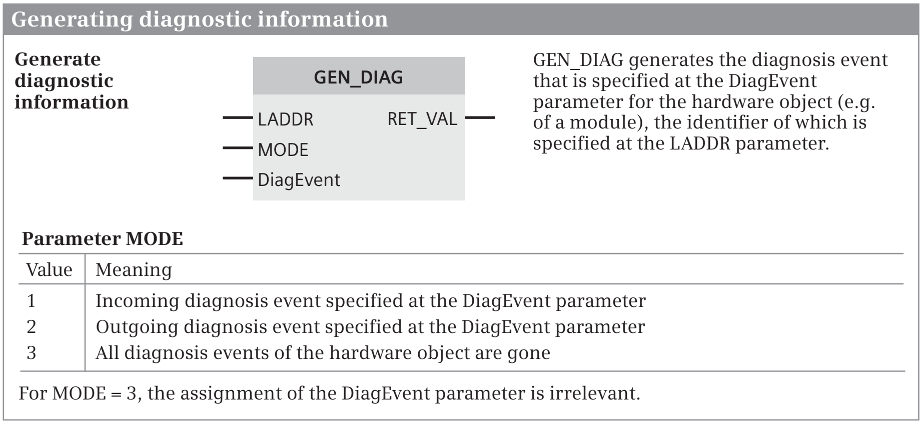
If MODE = 2, the diagnostic status of all the modules in the hardware object is output at parameter DIAG. CNT_DIAG is set to value 1, DETAIL contains module state information.

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.

GEN_DIAG Generate diagnostic information

GEN_DIAG generates a diagnostic event for a hardware object from a different manufacturer, which has been integrated beforehand with a GSD/GSDL/GSDXL file in the hardware catalog. The object, e.g. a module, is specified with the hardware identifier at the parameter LADDR. Fig. 5.37 shows the function call.

With the MODE parameter, you specify whether the diagnostic event is to be generated as an incoming or outgoing event. The diagnostic event is specified at the parameter DiagEvent, the structure of which can be found in the online help for GEN_DIAG. The type of information that can be generated depends on the hardware object that is addressed (see operating instructions).


Generating diagnostic information
Generate diagnostic information
GEN_DIAG
LADDR
MODE
DiagEvent
RET_VAL

GEN_DIAG generates the diagnosis event that is specified at the DiagEvent parameter for the hardware object (e.g. of a module), the identifier of which is specified at the LADDR parameter.

Parameter MODE

Value	Meaning
1	Incoming diagnosis event specified at the DiagEvent parameter
2	Outgoing diagnosis event specified at the DiagEvent parameter
3	All diagnosis events of the hardware object are gone

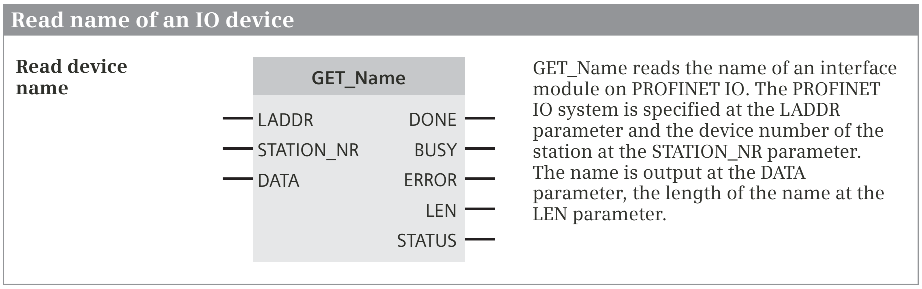
For MODE = 3, the assignment of the DiagEvent parameter is irrelevant.

Fig. 5.37 Generating diagnostic information

GET_NAME Read out name of an IO device

GET_NAME reads out the name of the interface module of a station in PROFINET IO. The PROFINET IO system is specified with the hardware identifier at the parameter LADDR and the interface module is specified with the device number at the parameter STATION_NR.

The name of the IO device is output at the DATA parameter. The length of the name (number of characters) is output at the LEN parameter. If the name is longer than the space provided at the DATA parameter, the name is limited to the maximum possible length (Fig. 5.38).



GET_Name reads the name of an interface module on PROFINET IO. The PROFINET IO system is specified at the LADDR parameter and the device number of the station at the STATION_NR parameter. The name is output at the DATA parameter, the length of the name at the LEN parameter.

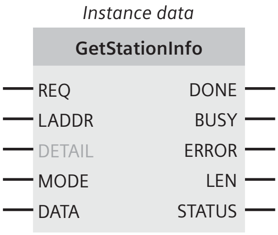
Fig. 5.38 Read name of an IO device

GetStationInfo Read out information of an IO device

GetStationInfo reads information from an IO device (currently the IPv4 address of the interface). Specify the hardware identifier of the station at the LADDR parameter (not the ID of the interface module). MODE is assigned the value 1, DETAIL is assigned the value 0. The address information is output at the DATA parameter with the structure shown in Fig. 5.39.

Read information of an IO device

Read IO device information



GetStationInfo reads the address information of an interface module in a PROFINET IO station, the hardware identifier of which is specified at the LADDR parameter. 1 is assigned to MODE. The information is output at the DATA parameter.

Structure IF_CONF_v4 of the parameter DATA (MODE = 1)

Byte	Name	Data type	Description
0..1	ID	UINT	ID of the structure IF_CONF_v4 (ID = 30)
2..3	Length	UINT	Length of the structure in bytes (length = 18)
4..5	Mode	UNIT	Mode = 0
6..9	InterfaceAddress	ARRAY [1..4] OF BYTE	IP address of the IO device
10..13	SubnetMask	ARRAY [1..4] OF BYTE	Subnet mask
14..17	DefaultRouter	ARRAY [1..4] OF BYTE	IP address of the router

Fig. 5.39 Read information of an IO device

DeviceStates Read status of distributed I/O stations

DeviceStates reads the status of the I/O stations in a PROFINET IO system or PROFIBUS DP master system. At the LADDR parameter you specify the hardware identifier of the PROFINET/PROFIBUS system. With the MODE parameter you select the type of status information that is displayed at the STATE parameter for all I/O stations. Fig. 5.40 shows the function call.

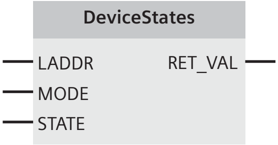
Via the MODE parameter you select the type of status information to be output at the STATE parameter. With a bit set to signal state "1", the bit array at the STATE parameter shows that the selected status information applies to the affected station. Example: If you want to determine

which stations are disrupted, assign the value 2 to the MODE parameter. Bit 0 at the STATE parameter has signal state "1" if 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 parameter STATE outputs the station status in a bit array, which is specified via the MODE parameter. For PROFINET, the length of the bit array is 1024 bits. For PROFIBUS, it is 128 bits. 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.

Read status of distributed stations

Read status of distributed stations



DeviceStates reads the status of distributed stations in a PROFINET IO system or PROFIBUS DP master system.

Parameter MODE

Value	Meaning
1	Configuration not completed
2	Station faulty
3	Station deactivated
4	Station present
5	Station has a problem

Parameter STATE

The STATE parameter represents every station with a bit (1024 for PROFINET, 128 for PROFIBUS). The number of the bit corresponds to the station or device number. If the respective bit has signal state "1", the status indicated at the MODE parameter applies to the stations. The status is valid across all stations in bit 0: If bit 0 has signal state "1", the scanned status applies to at least one station.

Fig. 5.40 Read status of distributed stations

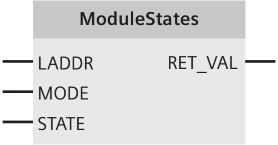
ModuleStates Read status of the modules in a station

ModuleStates reads the status of the modules in an IO device or in a DP slave. At the LADDR parameter you specify the hardware identifier of the station. With the MODE parameter you select the type of status information that is displayed at the STATE parameter for all modules. Fig. 5.41 shows the function call.

Via the MODE parameter you select the type of status information to be output at the STATE parameter. With a bit set to signal state “1”, the bit array at the STATE parameter shows that the selected status information applies to a module. Exam-

Read status of the modules of a distributed station

Read status of the modules of a distributed station



ModuleStates reads the status of the modules in an IO device or in a DP slave.

Parameter MODE

Value	Meaning
1	Configuration not completed
2	Module faulty
3	Module deactivated
4	Module present
5	Module has a problem

Parameter STATE

The STATE parameter represents every module of a distributed station with a bit (maximum 128). The number of the bit corresponds to the slot of the module. If the respective bit has signal state "1", the status specified at the MODE parameter applies to the module. The status is valid across all modules in bit 0: If bit 0 has signal state "1", the scanned status applies to at least one module.

Fig. 5.41 Read status of a central module

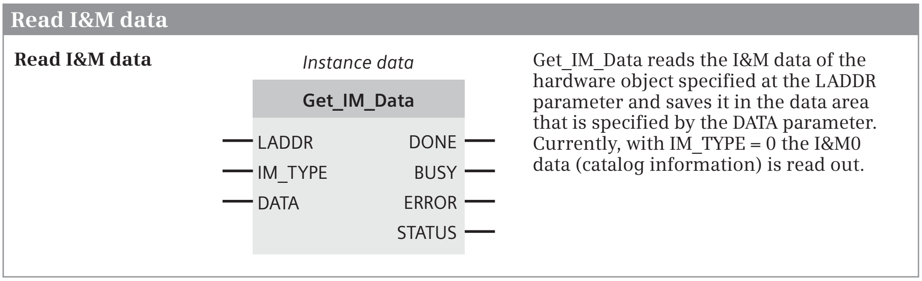
ple: If you want to determine which modules are disrupted, assign the value 2 to the MODE parameter. Bit 0 of the STATE parameter has signal state “1” if at least one module is disrupted. If bit 2 is set to signal state “1”, the module at slot 2 is disrupted.

The parameter STATE outputs the module status in a bit array, which is specified via the MODE parameter. The bit array has a maximum length of 128 bits. 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_IM_Data Read I&M data

Get_IM_Data reads the I&M data (Identification & Maintenance) of the hardware object (e.g. a module), the hardware identifier of which is specified at the LADDR parameter. At the parameter IM_TYPE, specify which information is to be read.

Fig. 5.42 shows the graphic representation of GET_IM_Data.



Get_IM_Data reads the I&M data of the hardware object specified at the LADDR parameter and saves it in the data area that is specified by the DATA parameter. Currently, with IM_TYPE = 0 the I&M0 data (catalog information) is read out.

Fig. 5.42 Read I&M data

Currently the I&M0 data is read with IM_TYPE = 0. This information is also displayed in online mode in the *Online & diagnostics* window.

The read information is stored in the data area, which is specified by the DATA parameter. You can create tags wit the data types STRING, STRUCT, ARRAY OF CHAR, or ARRAY OF BYTE at the DATA parameter. If the actual parameter is too small for the read data, the read data is entered up to the length of the actual parameter and an error message is output at the STATUS parameter. If the actual parameter is longer than the read data, the remainder is filled with zeroes. For a STRING tag, the actual length is adapted.

5.10 Configuring alarms

5.10.1 Introduction

Alarms, to put it plainly, indicate events. An event can be the signal state change of an input or of a bit memory, for example, or a specific status during the processing of the user program. An alarm is normally displayed on a display device (on an HMI station). An event-dependent alarm can be configured with a specific alarm text and alarm attributes and thus point to warnings or faults in the controlled process and their origin.

SIMATIC S7 distinguishes between the following types of alarm:

▷ System diagnostics alarms
System alarms report events on modules. They are activated or deactivated in the hardware configuration. They can be viewed, but not edited, in the alarm editor.

▷ Program alarms
Program alarms report events which occur synchronously with the processing of the user program. They are assigned to a respective block. They are created using the program editor and edited in the alarm editor.

▷ User diagnostics alarms
A user diagnostics alarm writes an entry in the diagnostics buffer and sends a corresponding alarm to a display unit. User diagnostics alarms cannot be programmed in STEP 7 V12 SP1 for a CPU 1500.

Alarm procedure

The alarm procedure defines the way in which alarms are configured, initiated, and displayed. The alarm procedure used must be available both in the PLC station and in the HMI station.

Bit messaging uses a bit in the PLC as alarm signal. If the signal state of the alarm signal changes, an alarm which has been configured in the HMI station is displayed on the HMI station.

Analog messaging monitors a digital value which, for example, is derived from an analog input module to detect exceeding or undershooting a limit value and generates the alarm signal from the limit violation. The alarms are configured in the HMI station.

During the *Message numbering* an alarm is initiated in the PLC station by calling an alarm block. The alarm number and associated alarm texts are configured in the PLC station, compiled, and then transferred to the HMI station. During runtime, the PLC station sends an alarm number and the time stamp to the HMI station. The display of the alarms is configured in the HMI station.

The configuration for the message numbering with block-related alarms in a PLC station is described in the following.

Components of an alarm

The displaying of an alarm depends on the alarm procedure, the alarm block, and display device. The possible components of an alarm are:

- The time stamp shows when the event occurred in the programmable controller.
- The alarm number is unique CPU-wide. It is assigned by the alarm editor and identifies an alarm.
- The alarm status shown the status of an alarm: incoming, outgoing, outgoing without acknowledgment, outgoing with acknowledgment.
- For PLC alarms, the alarm text is configured by the user.
- Associated values, which contain values from the controlled process, can be sent along with an alarm.

Alarm block

An alarm is generated by the *Program_Alarm* alarm block. For a positive and negative edge of the alarm signal, it sends an alarm with a time stamp for each. Up to ten associated values can be sent along with the alarm. Mandatory acknowledgement can be configured for the alarm.

The principle of programming for the alarm number procedure

An alarm block, e.g. Program_Alarm, generates an alarm if the binary alarm signal changes its status at the SIG parameter. The time stamp (TIMESTAMP parameter) and the associated values (SD_x parameter) are then added to the alarm.

The alarm block is called in a function block (any) as a local instance (Fig. 5.43). This “Alarm function block” contains the instance data of the alarm block in its local data. This is the alarm type. It serves as a template for the “actual” alarm, the alarm instance.

When the alarm function block is called, the instance data of the alarm types, the alarm instances, are located in its instance data – either in the instance data block or in the local data of the calling function block. An alarm instance is the “actual” alarm and contains the alarm number that applies CPU-wide. The name of the alarm instance is the alarm name.

The properties of the alarm type, such as the alarm group or the display class, are passed on to the alarm instance. If the properties in the alarm type are locked, they can no longer be changed in the alarm instance. To change locked properties, you must unlock them beforehand in the alarm type.

If, for example, there are only a few alarms and you call an alarm block in the alarm function block for each alarm and call the alarm function block only once, you can create the alarm signal directly on parameter SIG of the respective alarm block. Each alarm type then only has one alarm instance.

If you call the alarm function block multiple times, however, it makes sense to create an input parameter of the alarm function block at parameter SIG of the alarm

Principle of alarm programming

Programming an alarm

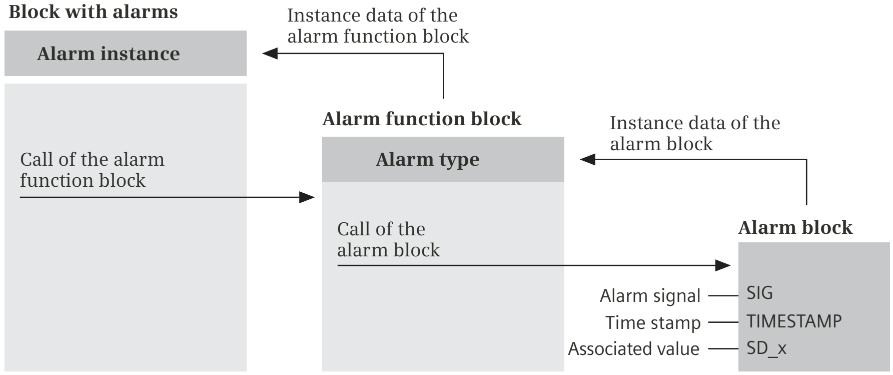
The alarms are programmed in a function block (the “alarm function block”). For an alarm you insert the alarm block as local instance (multi-instance) in the program of the alarm function block. You can insert several alarm blocks in the alarm function block and you can program several alarm function blocks.

Alarm types and alarm instances

The instance data of the alarm block in the alarm function block are the alarm type. The name of the local instance is the name of the alarm type. If you insert further alarm blocks, the alarm types are distinguished based on their names.

The alarm function block can be called either as single instance with own instance data block or as local instance with a different function block. The instance data of the alarm type is saved as alarm instance in the instance data of this call. This is then the "actual" alarm which contains the alarm number that applies CPU-wide. The name of the alarm instance is the alarm name.

You can call an alarm function block several times with different instance data in each case. Different alarm instances with different alarm numbers in each case are then generated from the alarm types.



Locking of alarm properties

All properties of an alarm type are passed on to the alarm instances. If the properties are “locked”, they can no longer be changed in the alarm instance (in the “actual” alarm).

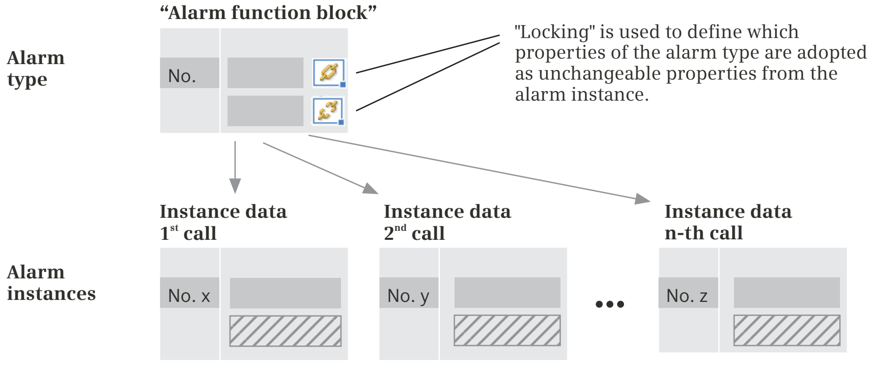


Fig. 5.43 Principle of alarm programming

block. With each call of the alarm function block you then supply the input parameters with the alarm signals. Thus each alarm instance is given its “own” alarm signal. Each alarm type then has the same number of alarm instances as the number of times the alarm function block is called.

Properties of alarms

An alarm can be provided with several properties which (also) depend on the display device used. Not all of the display devices support the properties listed in the following.

The *alarm text* should describe the reason for the alarm or its trigger event. *Associated values*, which contain the process values that were current at the time the alarm was initiated, can be inserted into the alarm text at any position. Certain display devices also accept an *info text* for an alarm, which can contain handling instructions for the machine operator, for example, and one or more *additional texts*.

If acknowledgement is mandatory, the alarm is displayed until it is acknowledged. This ensures that the alarm regarding a critical or hazardous process status has actually been registered by the operator. The *Priority* can be used to set the urgency with which the alarm must be acknowledged. Alarms that are acknowledged with an operator action can be grouped together into an *alarm group*. For example, this can be alarms which are all caused by the same fault or which all come from one machine unit or from one subprocess.

In an *alarm class*, alarms with the same level of importance can be grouped together, such as “warnings” or “errors”. An alarm class defines the representation on the display device and the mandatory acknowledgement.

A *display class* controls the assignment to the display unit. If, for example, several HMI stations are assigned to a PLC station, the display class can be activated in an HMI station, along with its alarms which are to be displayed at the station.

5.10.2 Configuring alarms according to the alarm number procedure

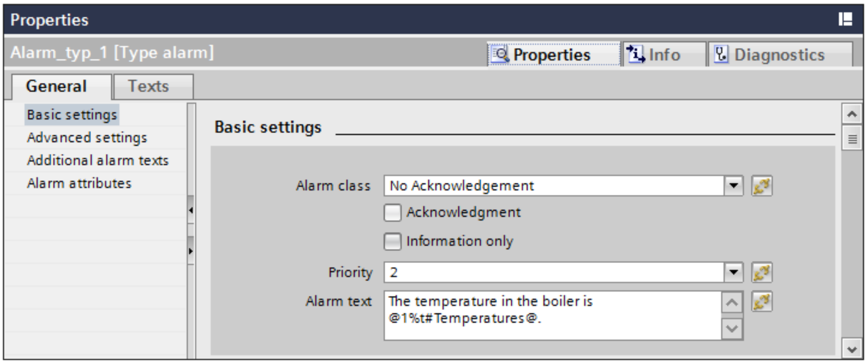
Programming alarm blocks

Open any function block and drag it from the program elements catalog under *Extended instructions* > *Alarms* into the program of the function block. The instance data of the alarm block is found in the instance data of the calling “alarm function block”. The parameters SIG and SD_x of the alarm block should be connected to the input/output parameters of the “alarm function block” in order to be able to individually supply them for multiple calls.

Repeat the inserting of the alarm block for each alarm. Several alarms can be programmed in the “alarm function block”. Supplement the program with your system-specific statements. You can also create more than one “alarm function block”.

Defining alarm properties in the alarm type

In the “alarm function block”, select the instance data of an alarm block call and set the alarm properties in the inspection window in the *Properties* > *Alarm* tab: Under *Basic settings*, you can enter the *Alarm class*, the *Priority* and the *Alarm text*, and you can activate the mandatory acknowledgement (depending on the acknowledging property of the alarm class). For alarms that are used for information, check the checkbox labeled *Information only*. Under *Additional alarm texts*, enter the info text and any additional texts as needed. Under *Advanced settings*, enter the display class and the group ID for the alarm group (Fig. 5.44).



Alarm class: No Acknowledgement

□ Acknowledgment

□ Information only

Priority: 2

Alarm text: The temperature in the boiler is @1%t#Temperatures@.

Fig. 5.44 Example of alarm properties in the inspector window

Clicking on the chain symbol for an alarm property allows you to lock (closed chain link) or unlock the property (open chain link). All of the alarm properties of the alarm type are passed on to the alarm instance. The locked properties can no longer be changed in the alarm instance. Unlocked ones can be changed.

In the “alarm function block” (at the alarm type), preferably set the alarm properties which are relevant to all of the alarm instances.

Setting alarm properties in the alarm instance

If you call up the “alarm function block” in the program of another block, specify the storage location of the instance data belonging to the call, either in its own instance data block or in the instance data block of the calling function block. You can also call up the “alarm function block” several times with different instance data in each case. The program editor generates an alarm instance for each call from the alarm type in the function block with its own CPU-wide unique alarm number. This is the “actual” alarm.

To set or change the properties of this “actual” alarm, open the instance data of the alarm type. If you have called the “alarm function block” as a single instance, it is the instance data block. If you have called the “alarm function block” as a local instance in a multi-instance, open the instance data block of the multi-instance and "open" the instance data of the local instance. You will find the programmed alarms in the instance data of the alarm type under *Static*. If you select an alarm, you can change the alarm properties which were unlocked in the alarm type in the *Properties* > *Alarm* tab in the inspector window.

After an alarm property is changed, a symbol ("type symbol") shows that the alarm property has changed compared to the alarm type. If you click on the type symbol, the original value is adopted again from the alarm type.

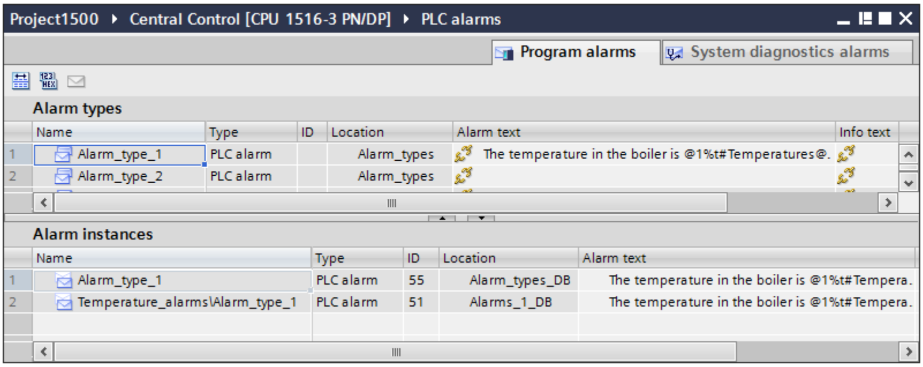
Setting alarm properties in the alarm editor

After you have programmed the alarm types (in the "alarm function block") and the alarm instances (when the "alarm function block" is called), you can also set the alarm properties using the alarm editor.

To start the alarm editor, double-click on *PLC alarms* in the project tree under the PLC station. In the *Program alarms* tab in the upper section of the working window in the *Alarm types* table, the alarm editor shows the programmed alarm types and it shows the alarm instances of the selected alarm type in the bottom section in the *Alarm instances* table. The alarm properties are displayed in both tables.

The example in Fig. 5.45 shows two alarm types in the upper *Alarm types* table. These are called *Alarm_type_1* and *Alarm_type_2*. The associated alarm blocks are programmed in the alarm function block, which is called *Alarm_types* in the example. Two alarm instances of the type *Alarm_type_1* exist. These are displayed in the lower *Alarm instances* table. For the first alarm instance (alarm number 55), the alarm function block is called as a single instance with the instance data block *Alarm types_DB*. For the second alarm instance (alarm number 51), the alarm function block has been called in the function block *Alarms_1* as a local instance with the name *Temperature_alarms*.

You can change the alarm properties directly in the tables of the alarm editor or in the properties tab of the inspector window. Individual columns can be hidden and shown: Right-click in a column title and then select the *Show/Hide* > ... command from the shortcut menu.



Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ PLC alarms
Program alarms
System diagnostics alarms

Alarm types

	Name	Type	ID	Location	Alarm text	Info text
1	Alarm_type_1	PLC alarm		Alarm_types	The temperature in the boiler is @1%t#Temperatures@.	
2	Alarm_type_2	PLC alarm		Alarm_types		

Alarm instances

	Name	Type	ID	Location	Alarm text
1	Alarm_type_1	PLC alarm	55	Alarm_types_DB	The temperature in the boiler is @1%t#Tempera.
2	Temperature_alarms\Alarm_type_1	PLC alarm	51	Alarms_1_DB	The temperature in the boiler is @1%t#Tempera.

Fig. 5.45 Example of program alarms in the alarm editor

Inserting associated values

In an alarm text, you can insert one or more associated values at any point. You define an associated value at the SD_n parameter of the alarm block. You then insert the following expression into the alarm text for each associated value: @<Associated value number><Format>@. A format entry is preceded by the percent character (%). Table 5.26 shows the permissible formats.

Table 5.26 Formats for associated values

Format	Display of the associated value as
%*n*X	Hexadecimal number with *n* places
%*n*u	Decimal number without sign with *n* places
%*n*d	Decimal number with sign and *n* places
%*n*b	Binary number with *n* places
%*n*.*m*f	Fixed-point number with sign and *n* total places, including *m* places after the decimal point; ".m" can also be omitted
%*n*s	Character string with *n* places (display up to the character value B#16#00)
%t#<Text list>	Access to a text list

Example: The expression @2%6.2f@ means that the associated value at parameter SD_2 ("2") with a total of 6 places including 2 decimal places ("%6.2") is to be displayed as a fixed-point number ("f").

If too few places are specified in the format specification, the associated value is nevertheless displayed in its full length. If the number of places is too great, leading spaces are inserted.

You can also insert text from a text list into an alarm as an associated value. The format for this is: @<Associated value number>%t#*name*@; *name* is the name of the text list.

Example: In the *Temperatures* text list, the text is selected based on decimal value ranges (Fig. 5.46). The configured alarm text

The temperature in the boiler is @1%t#Temperatures@.

is output as alarm text

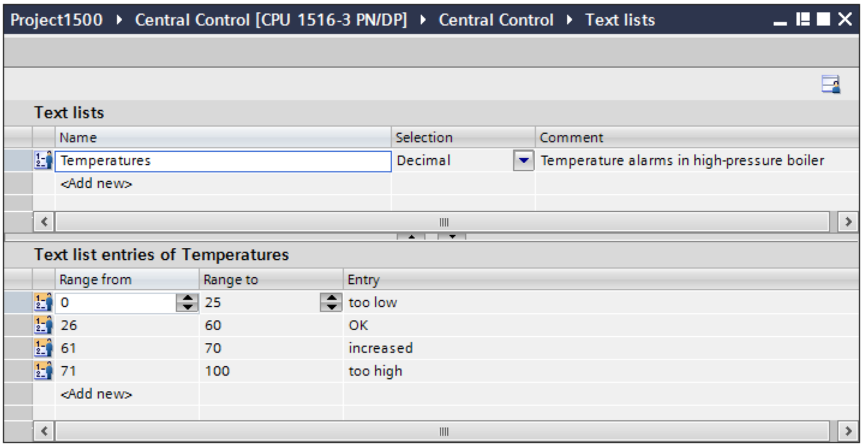
The temperature in the boiler is increased.

is output if the first associated value has the value 63.

Configuring text lists for alarm texts

Texts which are assigned to an individual value or a value range are managed in a text list. A text from a text list can thus be searched for (referenced) based on a value. Each text list has a unique name. A text list can be assigned to a station or to a project.

To create a new text list, double-click on *Text lists* in the project tree under the PLC station (station-assigned) or under the project and *Common data* (cross-sta-



Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ Central Control ▸ Text lists

Text lists

Name	Selection	Comment
Temperatures	Decimal	Temperature alarms in high-pressure boiler
<Add new>		

Text list entries of Temperatures

Range from	Range to	Entry
0	25	too low
26	60	OK
61	70	increased
71	100	too high
<Add new>		

Fig. 5.46 Example of a text list

tion). In the *Text lists* table in the upper section of the working window, add a new text list and give it a unique name (Fig. 5.46). In the *Selection* column, define the value range with which the texts of the text list will be referenced:

- "Decimal" if a decimal number or a range of decimal numbers (values 0 to 2^{16-1}) is the reference
- "Binary" if a bit or a bit range in a doubleword (bits 0 to 31) is the reference
- "Bit" if a bit ("0" or "1") is the reference

Select the text list. You define reference ranges in the *Text list entries of <list name>* table in the bottom section of the working window and the associated texts in the *Entry* column.

Configuring alarm classes

To configure an alarm class, double-click on *Alarm classes* in the project tree under the project and *Common data*. Two alarm classes have already been created in the *Alarm classes* table: *Acknowledgement* (for alarms with acknowledgement) and *No Acknowledgement* (for alarms without acknowledgement). To add a new alarm class, enter the name of the alarm class and the display name and activate or deactivate the mandatory acknowledgement for the alarm class. Now you can assign an alarm to the new alarm class when configuring the alarm properties.

5.10.3 Blocks for programming alarms

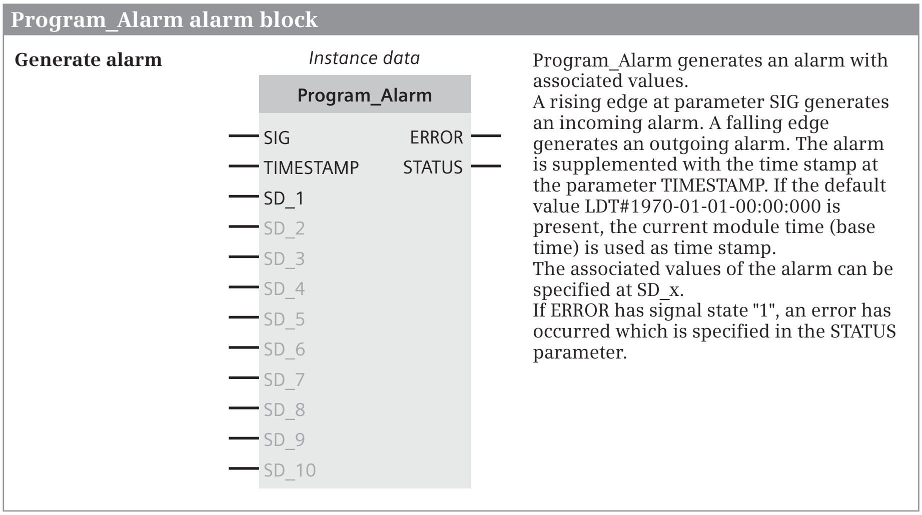
The following system blocks are available for programming alarms:

Program_Alarm	Create a program alarm with associated values
Get_AlarmState	Output an alarm state

You find these system blocks in the program elements catalog under *Extended instructions > Alarming*.

Program_Alarm

The alarm block *Program_Alarm* is called in a function block as a local instance. *Program_Alarm* generates an alarm with or without mandatory acknowledgement with up to ten associated values from a signal change. Fig. 5.47 shows the graphic representation of the alarm block.



Program_Alarm generates an alarm with associated values.
A rising edge at parameter SIG generates an incoming alarm. A falling edge generates an outgoing alarm. The alarm is supplemented with the time stamp at the parameter TIMESTAMP. If the default value LDT#1970-01-01-00:00:000 is present, the current module time (base time) is used as time stamp.
The associated values of the alarm can be specified at SD_x.
If ERROR has signal state "1", an error has occurred which is specified in the STATUS parameter.

Fig. 5.47 *Program_Alarm* alarm block

A rising signal edge at parameter SIG generates an incoming alarm. A falling signal edge generates an outgoing alarm. The time stamp is created at the parameter TIMESTAMP. This is used to output the alarm. If the parameter is provided with the default value LDT#1970-01-01-00:00:00, the module time at which the signal state change was detected is used as the time stamp. Otherwise, the alarm is given the time stamp at the parameter TIMESTAMP.

Up to 10 associated values, which are detected for an edge at parameter SIG and assigned to the alarm, can be sent along with the alarm.

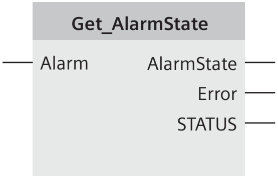
Get_AlarmState Output an alarm state

Get_AlarmState outputs the state of a program alarm. The instance data of the alarm block Program_Alarm (the alarm type) is created at the Alarm parameter. The status of the program alarm is output in one byte at the parameter AlarmState (Fig. 5.48).

If the Error parameter has signal state “1”, an error has occurred during execution of Get_AlarmState. This error is specified in more detail in the STATUS parameter.

Output an alarm state

Output alarm state



Get_AlarmState outputs the state of a program alarm.
The name of the instance data (the alarm type) is created at the Alarm parameter. The alarm state is then output at the AlarmState parameter.
Error and STATUS indicate the processing status.

AlarmState alarm state

Bit No.	Alarm state
0	"0" = Incoming, "1" = Outgoing
1	"1" = Incoming with acknowledgement
2	"1" = Outgoing with acknowledgment
3	"1" = Overflow for incoming alarms
4	"1" = Overflow for outgoing alarms
5	Reserved
6	Reserved
7	Alarm information ("0" = invalid, "1" = valid)

Fig. 5.48 Outputting an alarm state with Get_AlarmState

5.10.4 CPU alarm display

The CPU alarm display outputs the alarms saved in the CPU in online mode on a programming device. The alarm archive comprises system diagnostics alarms and program alarms. The alarm is displayed in the inspector window in the *Diagnostics > Alarm display* tab.

Setting the alarm archive

To set the alarm archive, select the command *Options > Settings* in the main menu. In the *Online & diagnostics* group, you can

- activate or deactivate the multi-line display in the inspector window
- activate or deactivate the automatic display of the current alarms
- select the size of the alarm archive from a drop-down list in increments in the range from 200 to 3000 alarms

If the alarm archive is full, the oldest alarm will be overwritten by the newly incoming alarm.

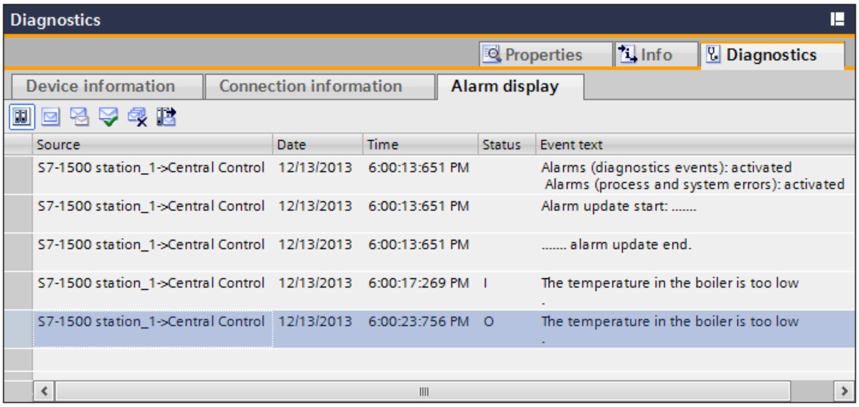
Receiving alarms

To display the alarms, activate the function *Receive alarms*. To do this, switch the programming device to online mode and,

- ▷ with the PLC station selected, select the command *Online* > *Receive alarms* from the main menu or *Receive alarms* from the shortcut menu, or
- ▷ double-click in the project tree under the PLC station on *Online & diagnostics* and activate the checkbox *Receive alarms* in the working window under *Online access* and *Alarms*.

Displaying alarms

The alarms are listed in the inspector window in the *Diagnostics* and *Alarm display* tabs (Fig. 5.49).



Source	Date	Time	Status	Event text
S7-1500 station_1->Central Control	12/13/2013	6:00:13:651 PM		Alarms (diagnostics events): activated Alarms (process and system errors): activated
S7-1500 station_1->Central Control	12/13/2013	6:00:13:651 PM		Alarm update start: ....
S7-1500 station_1->Central Control	12/13/2013	6:00:13:651 PM		.... alarm update end.
S7-1500 station_1->Central Control	12/13/2013	6:00:17:269 PM	I	The temperature in the boiler is too low .
S7-1500 station_1->Central Control	12/13/2013	6:00:23:756 PM	O	The temperature in the boiler is too low .

Fig. 5.49 Example of the alarm display of the CPU alarms

The table shows the alarms in the chronological order of their occurrence. To select the columns to be displayed, right-click in a column title and then select the *Show/hide columns* command from the shortcut menu. The order and width of the columns can be changed using the mouse.

You control the display with the symbols in the *Alarm display* tab. From left to right, the symbols are:

- ▷ *Archive view*: displays the alarms in the alarm archive in chronological order of occurrence.
- ▷ *Active alarms*: displays the currently pending alarms; alarms requiring acknowledgement are displayed in blue.
- ▷ *Ignore*: the displaying and archiving of the subsequent alarms are deactivated or activated; the activation or deactivation is displayed as an alarm.
- ▷ *Acknowledge*: acknowledges the selected alarm(s). Alarms requiring acknowledgement are displayed in blue.
- ▷ *Empty archive*: deletes all of the alarms in the alarm archive.
- ▷ *Export archive*: exports the alarm archive to a file in .xml format.

Acknowledge alarms

You can acknowledge the alarms requiring acknowledgement that were generated by the alarm blocks by selecting the relevant alarm(s) and clicking on the Acknowledge symbol or by pressing [Ctrl] + Q.

Status of the alarms

An alarm can have the following status in the archive view: Alarm came (I), Alarm came and was acknowledged (A), Alarm has gone (O), and Alarm was deleted (D). Alarms that are generated by the programming device such as a mode transition are displayed without a status.

In the “Active alarms” view, the alarm status is displayed as follows: I (Alarm came), IA (Alarm came and was acknowledged), and IO (Alarm has gone).

An “O” (Overflow) in red is displayed in the status column if more alarm events come in than the number of alarms that can be sent and displayed.

6 Program editor

6.1 Introduction

This chapter describes how you work with the program editor, with which the user program is written in the programming languages LAD, FBD, STL, SCL, and GRAPH. The special features of programming in the respective programming languages are described in Chapters 7 “Ladder logic LAD” on page 287, 8 “Function block diagram FBD” on page 323, 10 “Statement list STL” on page 395, 9 “Structured Control Language SCL” on page 359, and 11 “S7-GRAPH sequential control” on page 472.

The user program consists of blocks which are saved in the project tree under a PLC station in the *Program blocks* folder. Code 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. Ladder logic (LAD), function block diagram (FBD), structured control language (SCL), statement list (STL), and sequential control (GRAPH) are available as languages for programming the control function. You can define the programming language individually for each block. Blocks with the text-based programming languages SCL and STL can also be created as external source files as described in Chapter 18.1 “Working with source files” on page 780.

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. In the case of the code blocks, you start 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.

When you create the user program, you are supported by the cross-reference list, the assignment list, and the display of the call and dependency structure.

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

The user program works with operands, 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 absolute 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 279 for how to create a cross-reference list of the PLC tags. Monitoring of tags using the PLC tag table is described in Chapter 15.5.3 "Monitoring of PLC tags" on page 682.

6.2.1 Creating and editing a 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 a table. To obtain an overview of all tags and constants, double-click on *Show all tags* in the *PLC tags* folder. Fig. 6.1 shows an example of a PLC tag table.

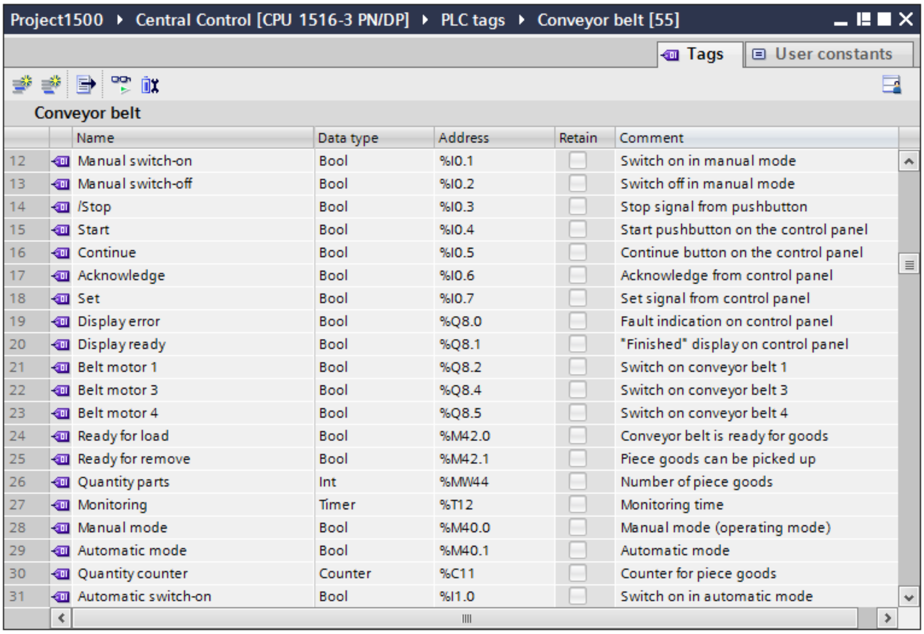
You can save an incomplete or faulty PLC tag table at any time and process it again later. However, the tag table must be error-free to enable compilation of the user program.

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*.

6.2.2 Defining and processing 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 to another PLC tag, a block, a symbolically addressed constant, or a PLC data type. 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 permissible as PLC tags. For a word or doubleword operand, specify the lowest respective byte number. For a long word operand, spec-


Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ PLC tags ▸ Conveyor belt [55]
Tags
User constants
Conveyor belt

	Name	Data type	Address	Retain	Comment
12	Manual switch-on	Bool	%I0.1	□	Switch on in manual mode
13	Manual switch-off	Bool	%I0.2	□	Switch off in manual mode
14	/Stop	Bool	%I0.3	□	Stop signal from pushbutton
15	Start	Bool	%I0.4	□	Start pushbutton on the control panel
16	Continue	Bool	%I0.5	□	Continue button on the control panel
17	Acknowledge	Bool	%I0.6	□	Acknowledge from control panel
18	Set	Bool	%I0.7	□	Set signal from control panel
19	Display error	Bool	%Q8.0	□	Fault indication on control panel
20	Display ready	Bool	%Q8.1	□	"Finished" display on control panel
21	Belt motor 1	Bool	%Q8.2	□	Switch on conveyor belt 1
22	Belt motor 3	Bool	%Q8.4	□	Switch on conveyor belt 3
23	Belt motor 4	Bool	%Q8.5	□	Switch on conveyor belt 4
24	Ready for load	Bool	%M42.0	□	Conveyor belt is ready for goods
25	Ready for remove	Bool	%M42.1	□	Piece goods can be picked up
26	Quantity parts	Int	%MW44	□	Number of piece goods
27	Monitoring	Timer	%T12	□	Monitoring time
28	Manual mode	Bool	%M40.0	□	Manual mode (operating mode)
29	Automatic mode	Bool	%M40.1	□	Automatic mode
30	Quantity counter	Counter	%C11	□	Counter for piece goods
31	Automatic switch-on	Bool	%I1.0	□	Switch on in automatic mode

Fig. 6.1 Example of a PLC tag table

ify the lowest byte number and – separated by a period – the bit number zero after the operand ID.

The peripheral operand area is addressed in the program by the extension ":P" on the tag name or on the operand. Therefore, it is sufficient to specify the corresponding input or output tags in the PLC tag table. A SIMATIC timer function and a SIMATIC counter function are addressed with a number.

The definition of a tag also includes the data type. This defines certain properties of the data identified by the name, basically the representation of the data content. An overview of the data types used with a CPU 1500 and the detailed description can be found in Chapter 4 "Tags, addressing, and data types" on page 86.

You can also assign a PLC data type to inputs and outputs. This PLC data type can contain all other types of data, except for STRING. Specify the operand ID, the lowest byte number, and the bit number zero as the operand.

One part of the operand areas bit memory and SIMATIC timer/counter functions can be set to retentive, i.e. this part retains the signal states and values when the power supply is restored. To set the retentive area, click on the icon for retentivity in the toolbar of the PLC tag table. In the dialog window that appears, enter the number of the retentive memory bytes and the number of SIMATIC timer/counter functions. A checkmark in the *Retain* column then identifies which bit memories and SIMATIC timer/counter functions are set to retentive.

Table 6.1 Approved data types and operands for PLC tags

Data types	Operand	Address
BOOL	Input bit Output bit Memory bit	%Iy.x %Qy.x %My.x
BYTE, SINT, USINT, CHAR	Input byte Output byte Memory byte	%IBy %QBy %MBy
WORD, INT, UINT, DATE, S5TIME	Input word Output word Memory word	%IWy %QWy %MWy
DWORD, DINT, UDINT, REAL, TIME, TOD	Input doubleword Output doubleword Memory doubleword	%IDy %QDy %MDy
LWORD, LINT, ULINT, LREAL, LTIME, LTOD, LDT	Input long word Output long word Memory long word	%Iy.0 %Qy.0 %My.0
TIMER COUNTER	SIMATIC timer function SIMATIC counter function	%Tn %Zn
PLC data type	Inputs Outputs	%Iy.0 %Qy.0

y = byte address, x = bit address, n = number

The properties of a PLC tag include the *Accessible from HMI* attributes (when activated, an HMI station can access this tag during runtime) and the *Visible in HMI* attribute (when activated, this tag is visible by default in the selection list of an HMI station).

Editing PLC tags

You can use *Insert row* from the shortcut menu to insert an empty line above 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 following the first click, in descending order following 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 left mouse button pressed.

If you enter the same name a second time, for example by 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 set or change the properties of a tag in the inspector window: Select the tag and choose the *Properties* tab in the inspector window.

You can also supplement, change, or delete the PLC tags when entering the user program (Chapter 6.3.7 “Editing tags” on page 267).

6.2.3 Comparing PLC tag tables

The PLC tags of a PLC station can be compared to the PLC tags of another station from the same project, from a reference project, or from a library. To perform the comparison, select the PLC station in the project tree and choose the command *Compare* > *Offline/offline* from the shortcut menu or alternatively the command *Tools* > *Compare* > *Offline/offline* from the main menu.

Compare editor

This starts the compare editor, which shows the PLC station with the contained objects on the left side. Using the mouse, drag the PLC station that is to be compared from a reference project, for example, into the title bar on the right side (labeled “Insert here to add a new object or replace an existing one...”). The “Status and action area” is located between the two tables. Above this is the switchover button with the scale.

In the automatic comparison (the switchover button with the scale is white), the tag tables are assigned on the left and right side based on their names and the comparison symbols are displayed in the center.

Activate the manual comparison by clicking on the switchover button. The switchover button is now gray. Manually assign the tag tables to be compared by selecting them using the mouse. The result of the comparison is displayed in the bottom area of the comparison window in the “Property comparison”. The lower area can be opened and closed using the arrow buttons.

Detailed comparison

Select a tag table and click on the *Start detailed comparison* icon. The PLC tags of both tag tables are individually listed and compared. The columns *Status* and *Action* are located between the lists. You can select the desired action from a drop-down list.

Comparison symbols and actions

A filled green circle means that the objects are identical. A blue-gray semicircle means that the objects differ. If one half of the circle is not filled, the corresponding object is missing. An exclamation mark in a gray circle indicates an object with differences in the identified folder.

In the *Action* column, you can select an action from a drop-down list for different objects, for example copying with an arrow in the direction in which you are copying. Clicking on the *Execute actions* icon starts the set actions. Note that you can neither add, delete, nor overwrite objects in reference projects.

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).

Table 6.2 Columns in the external file for the PLC tag table

PLC Tags worksheet						
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)	Comment	TRUE or FALSE	TRUE or FALSE
Constants worksheet						
Name	Path	Data Type	Value	Comment		
Name of constant	Group and name of PLC tag table	Data type of constant	Default value	Comment		

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 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.

6.2.5 Constants tables

A 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 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.

System constants

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 and 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 hardware identifier for a PROFINET IO system has the data type Hw_IoSystem and a value of, for example, 268. The name of the constant is defined in the properties of the PROFINET IO system using the hardware configuration.

The constants are used in the user program if a hardware or software object is to be addressed, for example if the status of I/O stations in a PROFINET IO system is to be read with DeviceStates. At the LADDR parameter, DeviceStates expects the hardware identifier for the PROFINET IO system, either as a constant or as a tag with the value of the constant or as a name.

The data types of the system constants are combined under the term “Hardware data types”. Chapter 4.12 “Hardware data types” on page 143 includes an example of a constants table.

6.3 Programming a code block

6.3.1 Creating a new code block

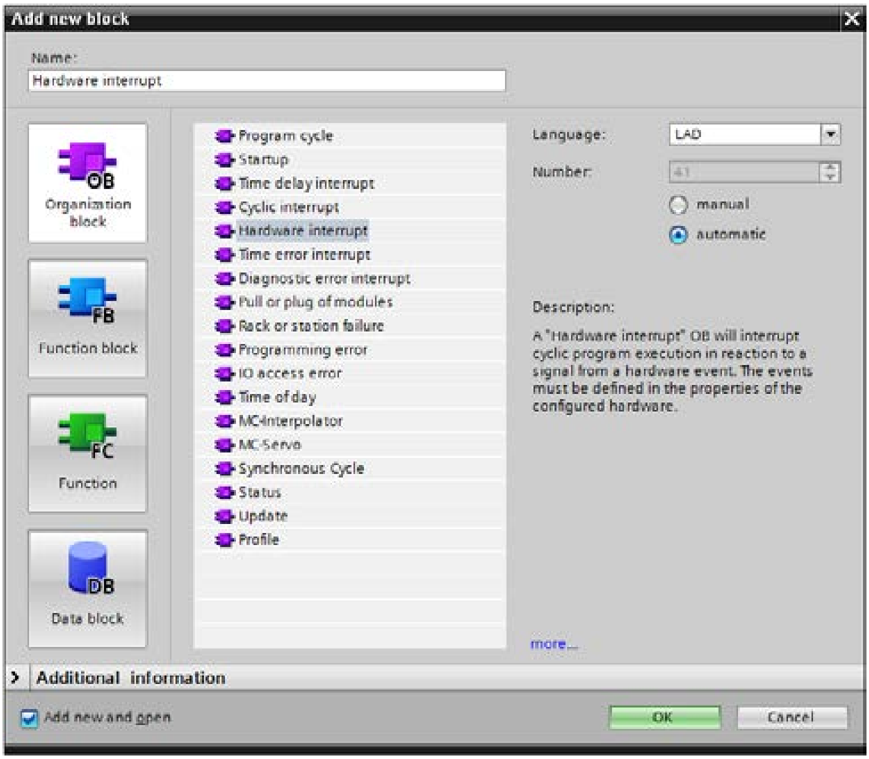
It is only possible to create a new block if a project with a PLC station has been opened. You can create a new block in either the Portal view or the Project view.

In the Portal view, click *PLC programming*. An overview window appears in which you can see the existing blocks. For 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 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.

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.



Add new block

Name: Hardware interrupt

Language: LAD

Number: 41

○ manual

● automatic

Description:

A "Hardware interrupt" OB will interrupt cyclic program execution in reaction to a signal from a hardware event. The events must be defined in the properties of the configured hardware.

☑ Add new and open

Fig. 6.2 *Add new block* window with organization block selected

You must assign an event class to an organization block, i.e. you define 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 192).

You set the default settings 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 settings for new blocks*, you can set the preselection for *IEC check*.

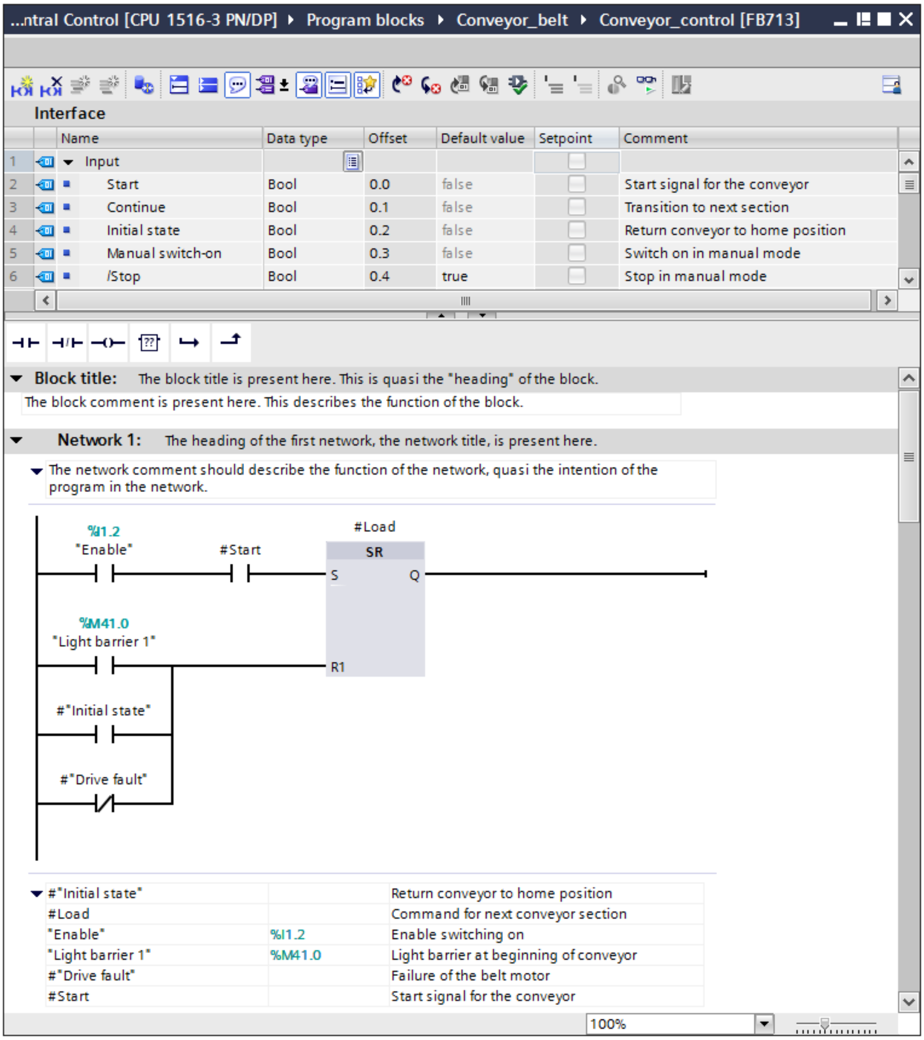
If the *Add new and open* checkbox is activated, the program editor is started by clicking on the *OK* button and programming of the newly created block can begin.

6.3.2 Working area of the program editor for code blocks

The program editor is automatically started when a block is opened. Open a block by double-clicking on its icon: 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.

You can adapt the properties of the program editor according to your requirements using the *Options* > *Settings* command in the main menu in the *PLC programming* section.

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 *Instructions* task card.



...ntral Control [CPU 1516-3 PN/DP] ▸ Program blocks ▸ Conveyor_belt ▸ Conveyor_control [FB713]

Interface

	Name	Data type	Offset	Default value	Setpoint	Comment
1	Input				□	
2	Start	Bool	0.0	false	□	Start signal for the conveyor
3	Continue	Bool	0.1	false	□	Transition to next section
4	Initial state	Bool	0.2	false	□	Return conveyor to home position
5	Manual switch-on	Bool	0.3	false	□	Switch on in manual mode
6	/Stop	Bool	0.4	true	□	Stop in manual mode

Block title: The block title is present here. This is quasi the "heading" of the block.

The block comment is present here. This describes the function of the block.

Network 1: The heading of the first network, the network title, is present here.

The network comment should describe the function of the network, quasi the intention of the program in the network.

#"Initial state"		Return conveyor to home position
#Load		Command for next conveyor section
"Enable"	%I1.2	Enable switching on
"Light barrier 1"	%M41.0	Light barrier at beginning of conveyor
#"Drive fault"		Failure of the belt motor
#Start		Start signal for the conveyor

Fig. 6.3 Example of the program editor's working window in ladder logic

The program editor's working window shows the following details:

- The toolbar
  contains the icons for the menu commands for programming, e.g. *Add network*, *Delete network*, *Go to next error,* etc. The significance of the icons is displayed if you hold the mouse pointer over the icon. Currently non-selectable icons are grayed out.
- 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 select or deselect *Display favorites in the editor.* To add an instruction to the favorites, select the instruction in the program elements catalog and drag it with the mouse into the favorites catalog or favorites bar. To remove an 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 working area 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*. 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

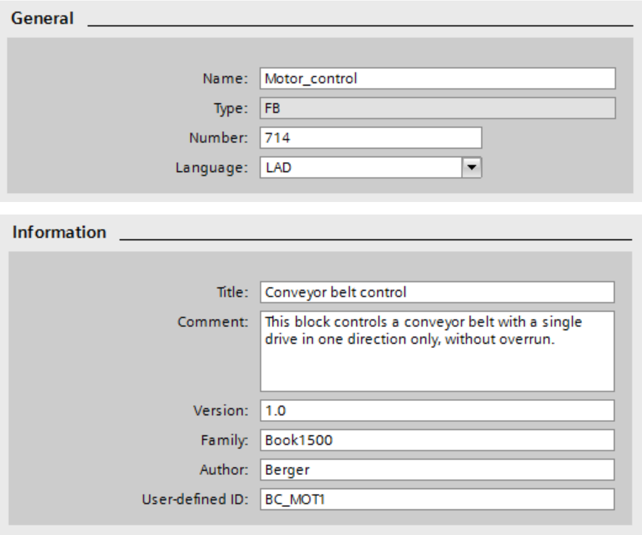
To set the block properties, select the block in the *Program blocks* folder, followed by the *Edit > Properties* command in the main menu or the *Properties* command in the shortcut menu.

Block properties which affect the block program when activated such as the test mode for data types (defined with the block attribute *IEC check*) should be set before the program is created. Some block properties can only be set when the block is created, for example the definition of the programming language SCL. To change these properties, you must create a new block.

Fig. 6.4 shows as example for the block properties the sections *General* and *Information* of a function block.

Block properties in the *General* section

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



General

Name: Motor_control
Type: FB
Number: 714
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: Book1500
Author: Berger
User-defined ID: BC_MOT1

Fig. 6.4 Block properties: *General* and *Information* tabs

and lower case when checking the name. The *Type* of the block is defined when the block is created. The *Number* specifies the block number within the block type. For blocks with a program, the *Language* is: LAD, FBD, STL, SCL, or GRAPH. In the case of a function block with sequence control (GRAPH), you set the programming language in the networks (LAD or FBD) in the block properties.

When creating an organization block, you also define the *Event class* to which the organization block belongs. The program editor creates a hardware identifier for the organization block in the *System constants* tab of the default tag table. You use the hardware identifier to address the organization block in the program, e.g. for assignment to an event. The name of the hardware identifier can be changed in the *Constant name* field in the block properties under *General*. The value of the hardware identifier corresponds to the number of the organization block.

Block properties in the *Information* section

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 upstream of 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 block ID can each comprise up to 8 characters (without spaces).

Block properties in the sections *Time stamps, Compilation, and Protection*

The time data in the *Time stamps* section contain the date of creation of the block and the date of the last modification to the block, interface, and program.

The *Compilation* section provides information on the processing status of the block, and – following compilation – on the memory requirements of the block in the load and work memories.

In the section *Protection* you can set up know-how protection and copy protection for the block. Further details are described in Chapter 6.3.4 “Protecting blocks” on page 259.

Block properties in the *Download without reinitialization* section

Function blocks with the attribute *Optimized block access* activated can be downloaded into the CPU after a change of the interface in the RUN operating state without resetting the actual values to the start values (see Chapter 15.3.3 “Download without reinitialization” on page 665). The changes are entered in a special memory area in the block, the “memory reserve”. In this section of the block attributes, you set the size of the memory reserve and the size of the memory area reserved for this for retentive tag values.

Block attributes for code blocks

Table 5.1 on page 158 shows an overview of the block attributes for all blocks. Additional attributes for compilation of SCL blocks are described in Chapter 6.5.2 “Compiling SCL blocks” on page 277. The attributes for compilation as well as the sequence properties for the GRAPH function block are described in Chapter 11.3.6 “Attributes of the GRAPH function block” on page 494. The special block attributes for interrupt and error organization blocks are described for the blocks concerned.

The *IEC check* attribute defines how strict the data type test should be in the code block for the implicit data type conversion (see Chapter 4.5.2 “Implicit data type conversion” on page 108). With the attribute not activated, it is usually sufficient if the tags used have the data width required for execution of the function or statement; with the attribute activated, the data types of the tags must correspond to the required data types. It is recommendable to set the *IEC check* attribute prior to block programming.

The attribute *Handle errors within block* is activated once one of the functions *GetError* or *GetErrorID* is inserted when the program is created in the block. Then the system response to a programming error or access error is omitted in favor of a self-programmed error routine.

The attribute *Block can be used as a know-how protected library element* shows that the block can be used in a library with know-how protection.

The *Optimized block access* attribute defines the data storage in the block and access to block tags. If the attribute is activated, the tags are not saved in the order of the declaration but in a way that is memory-optimized. This has effects on the addressing and the retentivity of the tags. 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. Furthermore, with the attribute activated, individual tags can be set as retentive (in the associated function block for instance data blocks); only the complete block can be set if the attribute is not activated.

The attribute *Set ENO automatically* only concerns code blocks with SCL program. If the attribute is activated, the block-local tag ENO is set to the value FALSE in the event of an error and the value is passed on to the enable output ENO.

The attribute *Parameter passing via registers* only concerns code blocks with STL program. If the attribute is activated, block parameters can be transferred via tabs to a block that is called with UC or CC.

6.3.4 Protecting blocks

A block can be protected

- from unauthorized access with a password (know-how protection) and
- from unintended execution by binding to a specific memory card or CPU (copy protection).

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
- Parameters of the block interface
- Program structure
- Global tags (listed in the cross-reference list without specification of the point of use)

The following actions are also possible:

- Modify name and number in the block properties (necessary for copying and pasting the block)
- Copy and paste block (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.

Know-how protection with source files

In the case of STL and SCL blocks for a CPU 300/400, in a source file it is possible to protect a block against undesired access by using the keyword KNOW_HOW_PROTECT. This protection no longer exists for blocks for a CPU 1500. The keyword KNOW_HOW_PROTECT has no effect here (see Chapter 18.1 "Working with source files" on page 780).

Configuring copy protection

If a block has copy protection, processing of the block is dependent on a specific CPU or memory card. The block must then be provided with the know-how protection so that the copy protection cannot be removed.

When configuring the copy protection, 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 protect* 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.

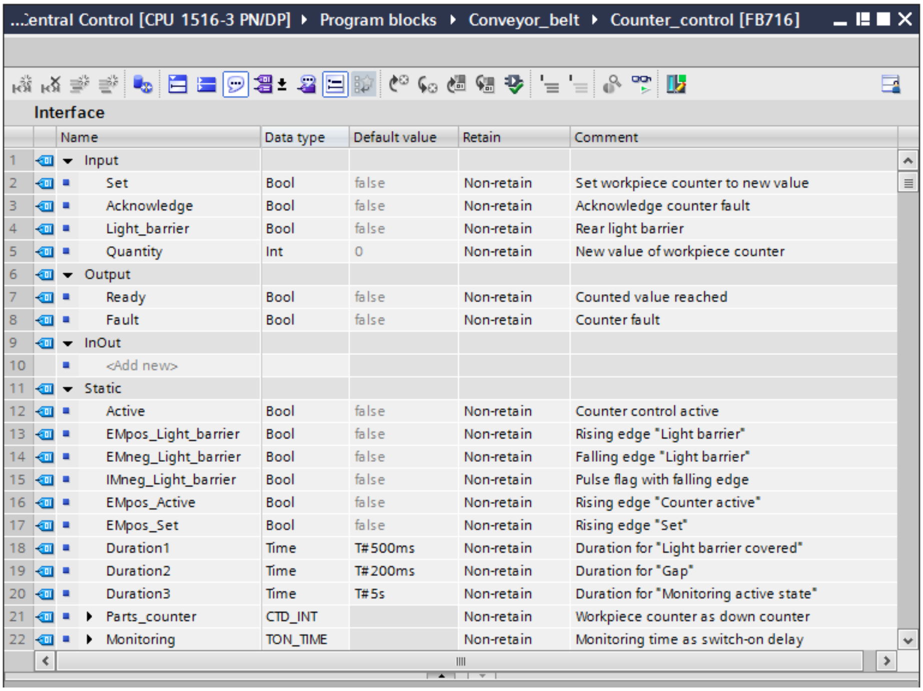
6.3.5 Programming a block interface

The block interfaces of the code 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 sections is described in detail in Chapter 5.3.3 "Block interface" on page 157.

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 or close the window. Fig. 6.5 shows an example of a function block interface.

Table 6.3 Declaration sections for code blocks

Declaration section	Meaning	Permissible with block type		
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	–



...Central Control [CPU 1516-3 PN/DP] ▸ Program blocks ▸ Conveyor_belt ▸ Counter_control [FB716]

Interface

	Name	Data type	Default value	Retain	Comment
1	▼ Input				
2	Set	Bool	false	Non-retain	Set workpiece counter to new value
3	Acknowledge	Bool	false	Non-retain	Acknowledge counter fault
4	Light_barrier	Bool	false	Non-retain	Rear light barrier
5	Quantity	Int	0	Non-retain	New value of workpiece counter
6	▼ Output				
7	Ready	Bool	false	Non-retain	Counted value reached
8	Fault	Bool	false	Non-retain	Counter fault
9	▼ InOut				
10	<Add new>				
11	▼ Static				
12	Active	Bool	false	Non-retain	Counter control active
13	EMpos_Light_barrier	Bool	false	Non-retain	Rising edge "Light barrier"
14	EMneg_Light_barrier	Bool	false	Non-retain	Falling edge "Light barrier"
15	IMneg_Light_barrier	Bool	false	Non-retain	Pulse flag with falling edge
16	EMpos_Active	Bool	false	Non-retain	Rising edge "Counter active"
17	EMpos_Set	Bool	false	Non-retain	Rising edge "Set"
18	Duration1	Time	T#500ms	Non-retain	Duration for "Light barrier covered"
19	Duration2	Time	T#200ms	Non-retain	Duration for "Gap"
20	Duration3	Time	T#5s	Non-retain	Duration for "Monitoring active state"
21	▸ Parts_counter	CTD_INT		Non-retain	Workpiece counter as down counter
22	▸ Monitoring	TON_TIME		Non-retain	Monitoring time as switch-on delay

Fig. 6.5 Example of 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, in the shortcut menu you can delete the line, insert an empty line above it, or add an empty line after it.

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. In the *Static* declaration section, tags can be identified as *Setpoints*. For a tag identified in this way, the value in the work memory can be overwritten in the operating state RUN and the current value from the work memory can be imported as a start value into the offline data management. Further details are described in Chapter 15.3.5 “Working with setpoints” on page 668.

In the case of a function (FC), the function value with the name of the block and data type VOID is displayed in the declaration section *Return*. The data type VOID prevents the display in the call box or call statement. If you specify a different data type here, the function value for LAD, FBD, and STL is displayed as the first output parameter. In 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 167).

An organization block (OB) can provide start information, which contains information about the call event, for example. If the *Optimized block access* attribute is activate, depending on the event class some organization blocks save start information in the declaration section *Input*. If *Optimized block access* is deactivated, each organization block provides start information (20 bytes long) in the *Temp* declaration section.

For code blocks with deactivated *Optimized block access* attribute, tags in the block interface can be superimposed with other data types as described in Chapter 4.5.3 “Overlaying tags (data type views)” on page 111.

6.3.6 Programming a control function

Working with networks

A network is part of a code block which, in the case of the LAD and FBD programming languages, contains a complete current path or a complete logic operation. The use of networks is optional for STL; it is recommendable to use networks for improved clarity. SCL and GRAPH do not use networks.

For a CPU 1500, it is possible to insert networks with STL program in a block with LAD or FBD program. Chapter 10.1.5 “STL networks in LAD and FBD blocks” on page 400 shows which special features must be observed here.

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.

Program elements catalog

All program elements permissible for the respective programming language (contacts, coils, boxes, statements, etc.) can be found for an open block 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 for technology modules)
- ▷ *Communication* (communication functions for data transmission and functions 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 logic (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 gray boxes indicate where the new program element may be positioned and – in green – where it is positioned when you “let go”.

In the ladder logic, the binary logic operations are implemented by series and parallel connections (Fig. 6.6). For the representation of the boxes, the Q or ENO output

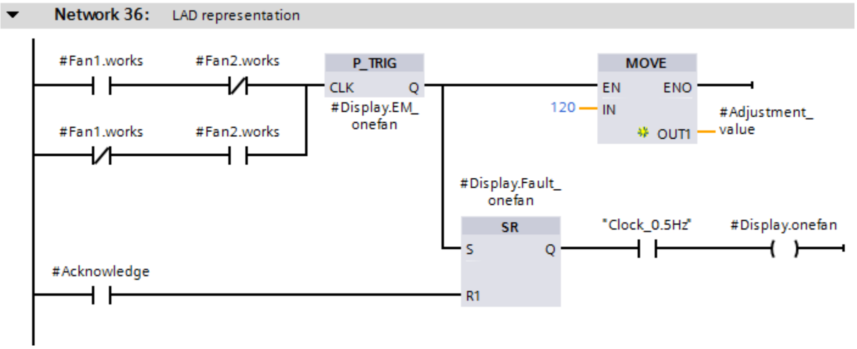


Fig. 6.6 Example of ladder logic representation with contacts, coils, and boxes

is positioned in the ladder logic representation at the top edge of the box in order to be able to "hang" the box into the current path. For many boxes, you have the choice to use the ENO output or to omit it. The structure of an LAD current path is described in Chapter 7 "Ladder logic LAD" on page 287.

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 "let go". 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. 6.7). The Q and ENO outputs are positioned at the bottom edge where they can be connected to the input of the following program ele-

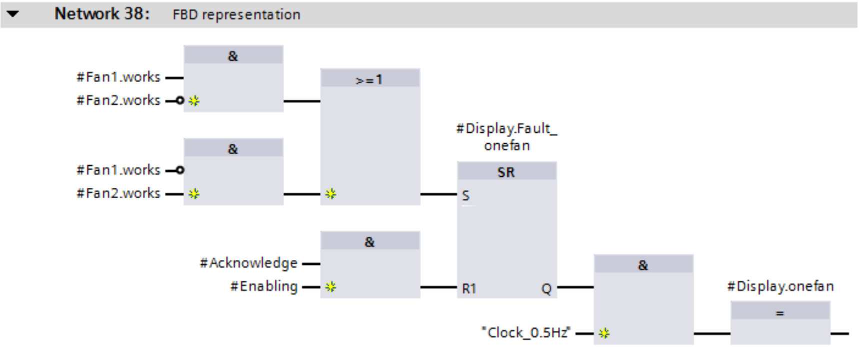


Fig. 6.7 Example of function block diagram with boxes

ment. For many boxes, you have the choice to use the ENO output or to omit it. The structure of an FBD logic operation is described in Chapter 8 “Function block diagram FBD” on page 323.

Selection of function and data types using 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 math 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 (Fig. 6.8). In this case, the drop-down list shows the data types permissible at this point, from which you can select the desired data type.

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.

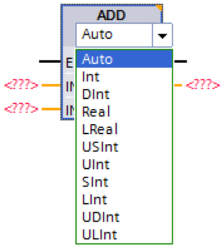
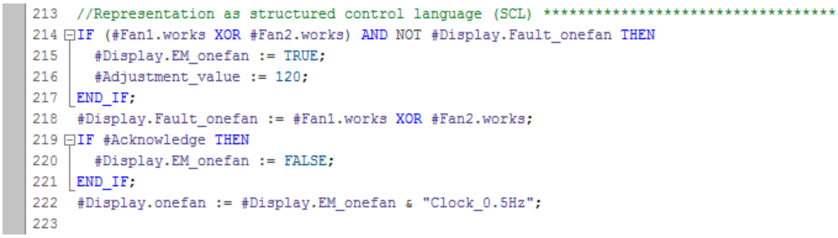


Fig. 6.8 Selection of data type using drop-down list

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 simple statements, for example a binary or digital operation, it is simpler to enter the statements as text via the keyboard.

Binary and digital logic operations are implemented in the SCL representation by expressions (Fig. 6.9). 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 359.



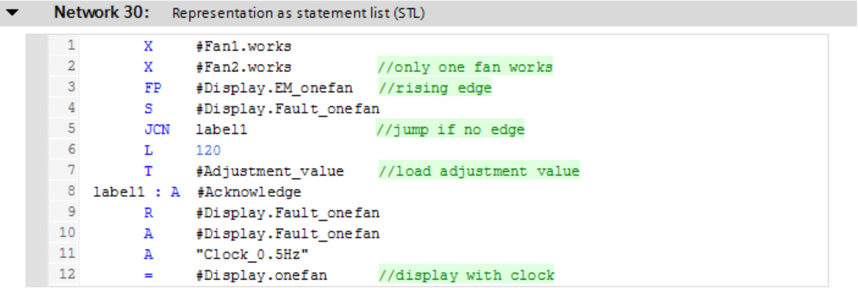
```
//Representation as structured control language (SCL) **********************************
IF (#Fan1.works XOR #Fan2.works) AND NOT #Display.Fault_onefan THEN
  #Display.EM_onefan := TRUE;
  #Adjustment_value := 120;
END_IF;
#Display.Fault_onefan := #Fan1.works XOR #Fan2.works;
IF #Acknowledge THEN
  #Display.EM_onefan := FALSE;
END_IF;
#Display.onefan := #Display.EM_onefan & "Clock_0.5Hz";

```

Fig. 6.9 Example of representation as structured control language (SCL)

Programming a control function with statement list (STL)

The control function is entered in STL line by line. Each line contains one statement. You can drag all statements from the program elements catalog into the working area. With simple statements, for example an AND logic operation, it is simpler to enter the statements line by line as text via the keyboard.

Binary logic operations are implemented in the representation as statement list by AND, OR, and exclusive OR logic operations (Fig. 6.10). The statements (operations and possibly operands) are written line by line. In the case of block calls and complex functions implemented as blocks, the block parameters are positioned underneath the call statement. The structure of an STL statement as well as processing of the statements are described in Chapter 10 “Statement list STL” on page 395.



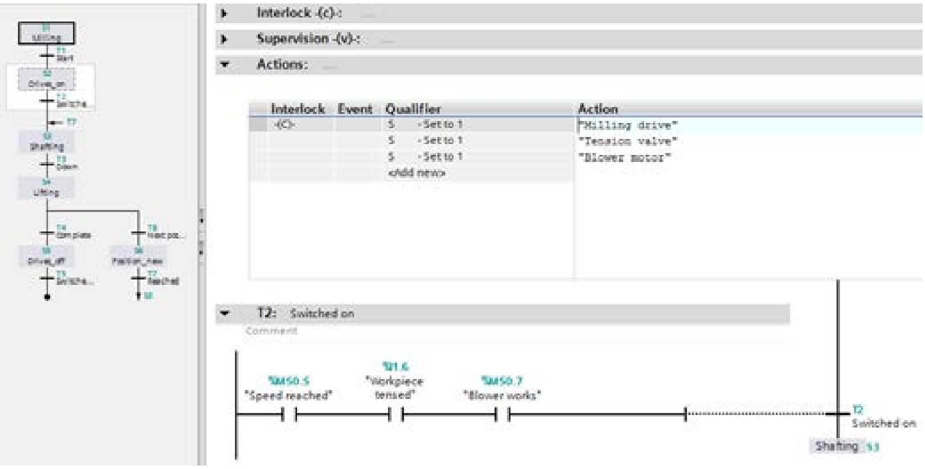
Network 30: Representation as statement list (STL)

```
          X      #Fan1.works
          X      #Fan2.works          //only one fan works
          FP     #Display.EM_onefan   //rising edge
          S      #Display.Fault_onefan
          JCN    label1               //jump if no edge
          L      120
          T      #Adjustment_value    //load adjustment value
label1 : A      #Acknowledge
          R      #Display.Fault_onefan
          A      #Display.Fault_onefan
          A      "Clock_0.5Hz"
          =      #Display.onefan      //display with clock
```

Fig. 6.10 Example of representation as statement list STL

Programming of a control function with sequential control (GRAPH)

You program a sequence control with the GRAPH programming language as a sequence of steps, transitions, and possibly branches and jumps. You create the



Interlock	Event	Qualifier	Action
-(C)-		S - Set to 1	"Milling drive"
		S - Set to 1	"Tension valve"
		S - Set to 1	"Blower motor"
		<Add new>	

Fig. 6.11 Example of representation as sequence control GRAPH

structure of the sequencer by “dragging” the corresponding element (step, transition, jump, etc.) from the programming elements catalog into the working area. To program the actions in a step or the links in a transition, select the desired item in the sequence tree on the left and program its contents in the working area (Fig. 6.11).

You can set or reset tags, program timer functions, create simple arithmetic logic operations, or call blocks in one step. The programming languages LAD and FBD are available for programming the logic operations, e.g. for transitions. The structure of a GRAPH sequence control is described in Chapter 11 “S7-GRAPH sequential control” on page 472.

6.3.7 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. 6.12 shows the insertion of an up/down counter as local instance (*#IEC_Counter_0_Instance*) in a function block. The example shows the representation in LAD, FBD, SCL, and STL.

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.

If you set the cursor to a block parameter or function parameter in STL, the declaration mode and the data type of the parameter are shown.

The program editor displays the global tags enclosed by quotation marks. Local tags are preceded by a number character (#); 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, symbolic 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* icon in the toolbar of the program editor.

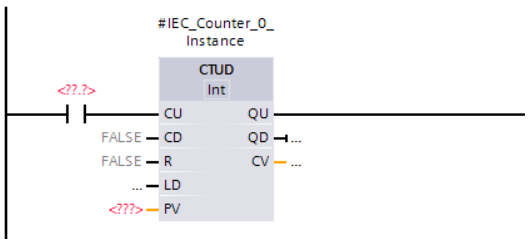
The data type of the tag must be compatible with the data type of the supply position. Use the block attribute *IEC check* to set how strict the program editor is when performing the check. Further details can be found in Chapter 4.5.2 “Implicit data type conversion” on page 108.

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 *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.

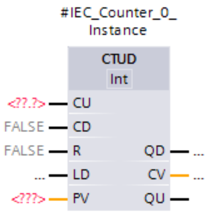
When programming the control function you can also enter the name of a tag which does not yet exist. The name of the tag is then underlined in red. By clicking with

Supply of an IEC counter function with tags

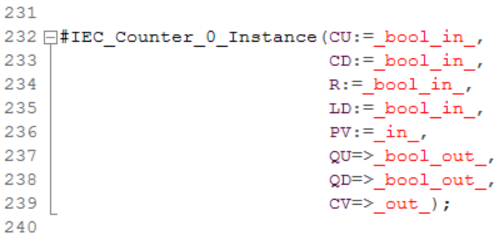
Call in LAD



Call in FBD

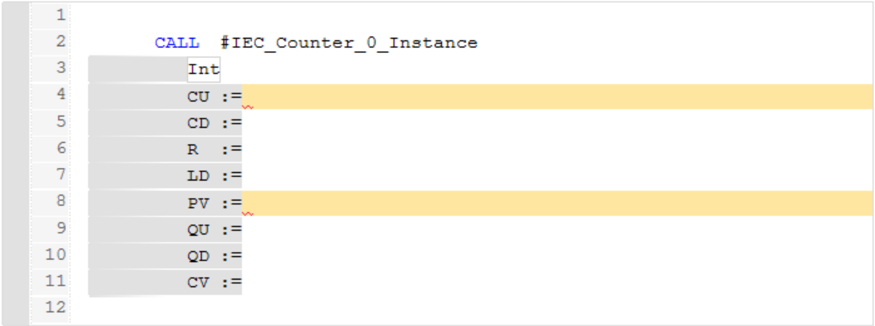


Call in SCL



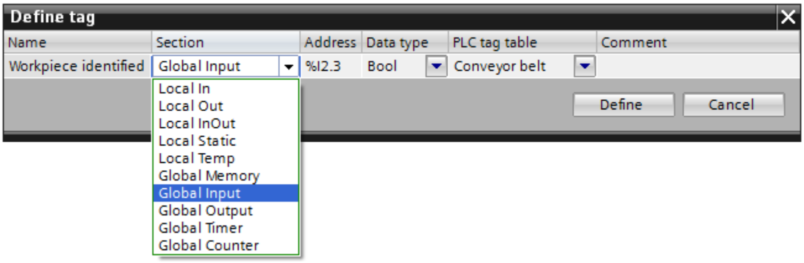
```
#IEC_Counter_0_Instance(CU:=_bool_in_,
                        CD:=_bool_in_,
                        R:=_bool_in_,
                        LD:=_bool_in_,
                        PV:=_in_,
                        QU=>_bool_out_,
                        QD=>_bool_out_,
                        CV=>_out_);
```

Call in STL



```
      CALL  #IEC_Counter_0_Instance
            Int
            CU  :=
            CD  :=
            R   :=
            LD  :=
            PV  :=
            QU  :=
            QD  :=
            CV  :=
```

Fig. 6.12 Supply with tags



Define tag

Name	Section	Address	Data type	PLC tag table	Comment
Workpiece identified	Global Input	%I2.3	Bool	Conveyor belt	

Fig. 6.13 Defining tags during program input

the right mouse button on the undefined tag and selecting *Define tag* from the shortcut menu you are provided with a new window in which you can define the tag (Fig. 6.13).

You can, for example, select the memory area in which the tag is to be positioned: Input, output or in/out parameter, static or temporary local data, bit memories, inputs, outputs, as well as peripheral inputs and peripheral outputs. You can also set the (existing) PLC tag table in which the tag is to be saved.

Showing and hiding tag information

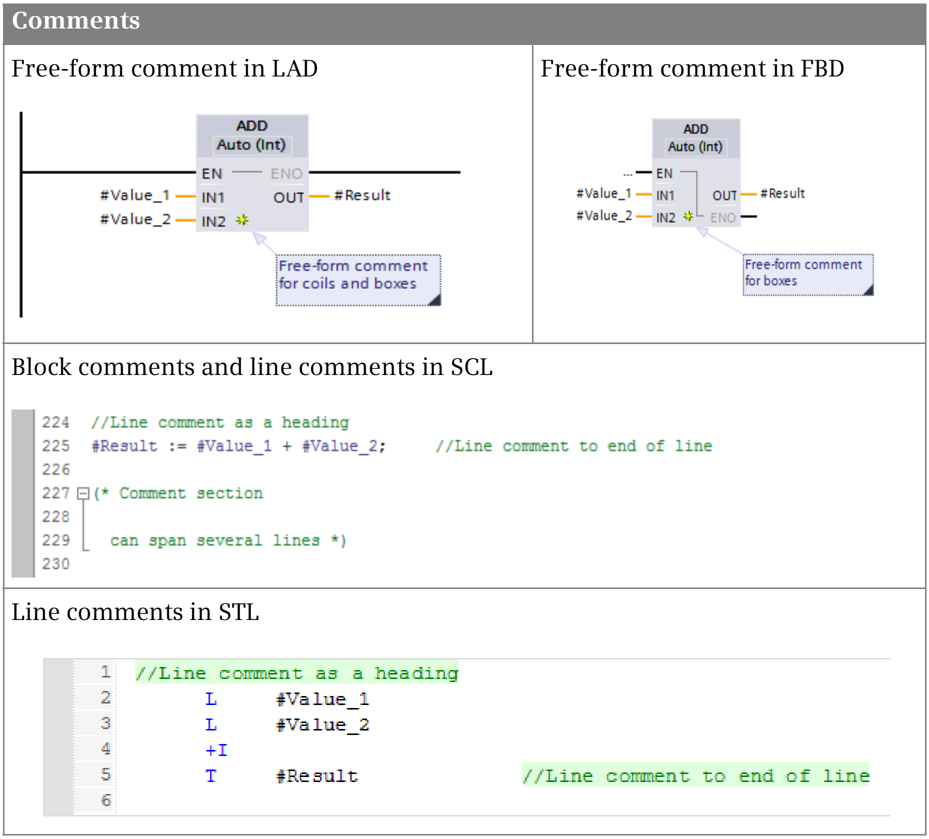
In LAD and FBD, you can display the name, address, and comments of the tags used in the network under the current path or the link. The general settings for all blocks are made in the main menu with the command *Options > Settings* in the group *PLC programming > General*. Here you can enable or disable the view with tag information. For the open block – and with the cursor in the program section – select *View > Display with > Tag information* or click in the toolbar of the working window on the *Tag information on/off* icon.

6.3.8 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. 6.14).

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 block comment starts with left parenthesis and asterisk and ends with an asterisk and right parenthesis; example: *(* This is a block comment *)*. It can extend over several lines. You can "comment out" code lines by positioning the cursor in the code line or by selecting several lines and clicking the *Disable code* icon in the toolbar of the working window. A line comment is then generated with the code line as content. You can undo the procedure using the *Enable code* icon.

With STL as the programming language you enter the comment following a double slash up to the end of the line. You can write the comment on its own in a line or position it after the STL statement. You can "comment out" code lines by positioning the cursor in the code line or by selecting several lines and clicking the *Disable code* icon in the toolbar of the working window. A line comment is then generated with the code line as content. You can undo the procedure using the *Enable code* icon.



Comments

Free-form comment in LAD

Free-form comment in FBD

Block comments and line comments in SCL

```
//Line comment as a heading
#Result := #Value_1 + #Value_2;      //Line comment to end of line

(* Comment section

  can span several lines *)

```

Line comments in STL

```
//Line comment as a heading
      L      #Value_1
      L      #Value_2
      +I
      T      #Result                //Line comment to end of line

```

Fig. 6.14 Comments in the various programming languages

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 with a PLC station has been opened. 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 icon 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) or system block. The data structure is defined during pro-

gramming of the block interface (for a function block) or is predefined (for a system block).

- An *ARRAY data block* has the structure of the ARRAY data type: It is a data field with components that all have the same data type. You define the data type and the upper array limit when you add the ARRAY data block.
- 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.

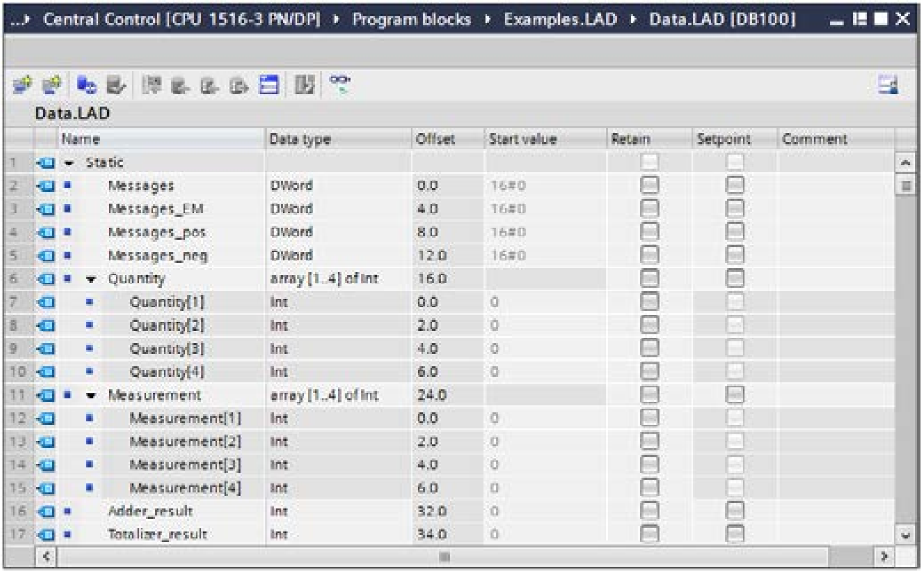
If the *Add new and open* checkbox is activated, the program editor is started by clicking on the *OK* button, and programming of the newly created block can begin.

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: 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. 6.15):

- The toolbar
  contains the icons (from left to right) for *Insert row*, *Add row*, *Reset start values*, *Update interface*, *Snapshot of the monitoring values*, *Copy all values from the "Snapshot" column to the "Start value" column*, *Copy all setpoints from the "Snapshot" column to the "Start value" column*, *Initialize setpoints*, *Expanded mode*, *Download without reinitialization and Monitor all*. The meaning of the icons is displayed if you hold the mouse pointer over the icon. 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* icon in the title bar, and embedded again using the icon for *Embed*. Display as a separate window is also possible: Click in the title bar on the icon for *Float*.



...› Central Control [CPU 1516-3 PN/DP] › Program blocks › Examples.LAD › Data.LAD [DB100]

Data.LAD

	Name	Data type	Offset	Start value	Retain	Setpoint	Comment
1	Static				□	□	
2	Messages	DWord	0.0	16#0	□	□	
3	Messages_EM	DWord	4.0	16#0	□	□	
4	Messages_pos	DWord	8.0	16#0	□	□	
5	Messages_neg	DWord	12.0	16#0	□	□	
6	Quantity	array [1..4] of Int	16.0		□	□	
7	Quantity[1]	Int	0.0	0	□	□	
8	Quantity[2]	Int	2.0	0	□	□	
9	Quantity[3]	Int	4.0	0	□	□	
10	Quantity[4]	Int	6.0	0	□	□	
11	Measurement	array [1..4] of Int	24.0		□	□	
12	Measurement[1]	Int	0.0	0	□	□	
13	Measurement[2]	Int	2.0	0	□	□	
14	Measurement[3]	Int	4.0	0	□	□	
15	Measurement[4]	Int	6.0	0	□	□	
16	Adder_result	Int	32.0	0	□	□	
17	Totalizer_result	Int	34.0	0	□	□	

Fig. 6.15 Example of the program editor's working window for data blocks

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.

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.

Block properties in the *General* section

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.

With data blocks, the designation *DB* together with the type of data block is present in the *Type* field: *Global DB* for a global data block, *Instance DB of <FB_name>* for an instance data block of the function block *<FB_name>*, *Array DB* for an ARRAY data block, and *Data block derived from <Type_name>* if the structure of the data block is based on the data type *<Type_name>*.

The block number of the data block is present in the *Number* field. The *language* for data blocks is always “DB”. For an ARRAY data block, the data type of the array component and the upper array limit (can be changed) are also displayed. The lower array limit is always zero.

Block properties in the *Information* section

The *Information* section contains the *Title* and the *Comment*. 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 block ID can each comprise up to 8 characters (without spaces).

Block properties in the sections *Time stamps*, *Compilation, and Protection*

The time data in the *Time stamps* section contain the date of creation of the block and the date of the last modification to the block, interface, and program.

The *Compilation* section provides information on the processing status of the block, and – following compilation – on the memory requirements of the block in the load and work memories.

In the *Protection* section you can set up know-how protection for the data block. Further details are described in Chapter 6.3.4 “Protecting blocks” on page 259.

Block attributes for data blocks

Table 5.1 on page 158 shows an overview of the block attributes for all blocks.

The *Optimized block access* attribute defines the data storage in the block. If the attribute is activated, the tags are not saved in the order of the declaration but in a way that is memory-optimized. This has effects on the addressing and the retentivity of the tags. If the attribute is activated, only symbolic addressing of the data tags is possible in the block. 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. Furthermore, with the attribute activated, individual tags can be set as retentive (in the associated function block for instance data blocks); only the complete block can be set if the attribute is not activated.

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. Data in the load memory can be read and also written using system functions. Data blocks with the *Only store in load memory* attribute activated are suitable for data which is only accessed rarely, e.g. recipes or data archives.

Data block write-protected in the device is an attribute for global and type data blocks. It means that you can only read from this data block by means of a program. Overwriting of the data is prevented and an error message is generated. The write protection applies to the data relevant to execution (actual values) in the work memory; the data in the load memory (start values) can be overwritten even if the data block is provided with write protection. Write protection must not be confused with block protection: A data block with block protection can be read and written by the program; however, its data can no longer be viewed using a programming or monitoring device.

The attribute *Set data in the standard area to retentive* concerns instance data blocks. When the attribute is activated, the retentivity of all of the tags with the default setting *Set in IDB* is switched on.

Block properties in the *Download without reinitialization* section

Global and instance data blocks with the attribute *Optimized block access* activated can be downloaded into the CPU after a change of the interface in the RUN operating state without resetting the actual values to the start values (see Chapter 15.3.3 “Download without reinitialization” on page 665). The changes are entered in a special memory area in the block, the “memory reserve”. In this section of the block attributes, you set the size of the memory reserve and the size of the memory area reserved for this for retentive tag values.

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, for example, to commence each call of the underlying function block (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 monitor value indicates the actual value of the tags in online mode. This is the value that is present in the work memory during scanning. This column is only displayed in *Monitoring* mode.
- Retain: A checkmark 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 checkmark in this column means that the tag is visible in the drop-down list of HMI stations by default.
- Accessible from HMI: A checkmark here indicates that an HMI device can access this tag.
- Setpoint: A checkmark in this column indicates that this value will be probably be set during commissioning. With tags marked in this way, the actual values can be imported into the offline data management system as start values. Further details are described in Chapter 15.3.5 “Working with setpoints” on page 668.
- 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. In expanded mode, the tags with structured data types (except for STRING) are “opened” so that the individual components can be displayed and – if permissible – assigned default values.

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 data type of a component. 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.

The drop-down list also shows the previously programmed PLC data types which you can also assign to a data tag. System data types are displayed if the corresponding statements (functions) have been programmed. When programming a tag with the system data type, for example *ErrorStruct* or *IEC_TIMER*, you cannot change the structure and you can only set defaults for individual components if it is permitted.

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 compilations while generating the user program to enable a quick response to any programming errors.

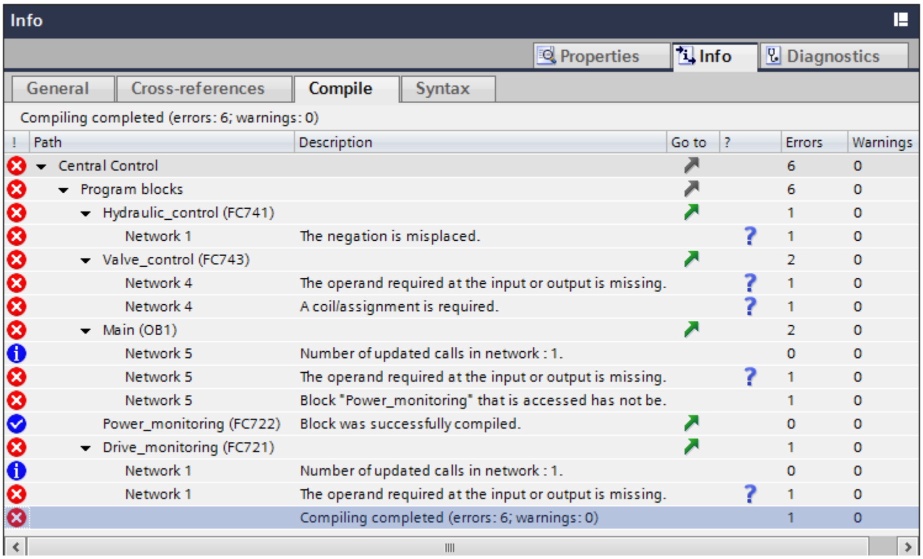
6.5.1 Starting the compilation

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 from the shortcut menu.
- ▷ 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 from the shortcut menu.
- ▷ To start the compilation process for the selected block, right-click a block in the *Program blocks* folder in the project tree followed by the *Compile > Software (only changes)* command from the shortcut menu.
- ▷ You can also select several blocks in a group in the *Program blocks* folder in the project tree and compile them together using the *Compile > Software (only changes)* command from the shortcut menu.
- ▷ By right-clicking on a group in the *Program blocks* folder, you can choose between *Compile > Software (only changes)* or *Compile > Software (reset memory reserve)* in 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 (only changes)*, ... *Software (rebuild all blocks)*, and ... *Software (reset memory reserve)*.

- If you select the *PLC station* folder and then *Compile* > ... from the shortcut menu, you can select between
  - ... *Hardware and Software (only changes)*
    Complete compilation of all project information relevant to execution
  - ... *Hardware (only changes)*
    Compilation of the device and network configuration
  - ... *Software (only changes)*
    Compilation of program changes since last compilation only
  - ... *Software (rebuild all blocks)*
    Compilation of entire user program
  - ... *Software (reset memory reserve)*
    Compile with resolution of the memory reserve (see Chapter 15.3.3 "Download without reinitialization" on page 665).

The result of the compilation is displayed in the inspector window in the *Info* tab under *Compile* (Fig. 6.16). 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.



Compiling completed (errors: 6; warnings: 0)

!	Path	Description	Go to	?	Errors	Warnings
✖	Central Control		↗		6	0
✖	Program blocks		↗		6	0
✖	Hydraulic_control (FC741)		↗		1	0
✖	Network 1	The negation is misplaced.		?	1	0
✖	Valve_control (FC743)		↗		2	0
✖	Network 4	The operand required at the input or output is missing.		?	1	0
✖	Network 4	A coil/assignment is required.		?	1	0
✖	Main (OB1)		↗		2	0
ℹ	Network 5	Number of updated calls in network : 1.			0	0
✖	Network 5	The operand required at the input or output is missing.		?	1	0
✖	Network 5	Block "Power_monitoring" that is accessed has not be.			1	0
✓	Power_monitoring (FC722)	Block was successfully compiled.	↗		0	0
✖	Drive_monitoring (FC721)		↗		1	0
ℹ	Network 1	Number of updated calls in network : 1.			0	0
✖	Network 1	The operand required at the input or output is missing.		?	1	0
✖		Compiling completed (errors: 6; warnings: 0)			1	0

Fig. 6.16 Example of compilation information in the inspector window

6.5.2 Compiling SCL blocks

If you activate the attribute *Automatically set ENO* in the properties of an SCL block, an additional program code is generated during the compilation which sets the enable output ENO to signal state "0" in the event of a program error during runtime.

You can also make additional settings in the main menu under *Options* > *Settings* and *PLC programming* > *SCL* > *Compile*:

- ▷ Create extended status information
  Permits monitoring of all tags in a block.
- ▷ Check ARRAY limits
  Checks the limits of ARRAY tags during runtime.

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 or clicking on the green arrow 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.

If you double-click on the error message, the program editor opens the network 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. 6.17).

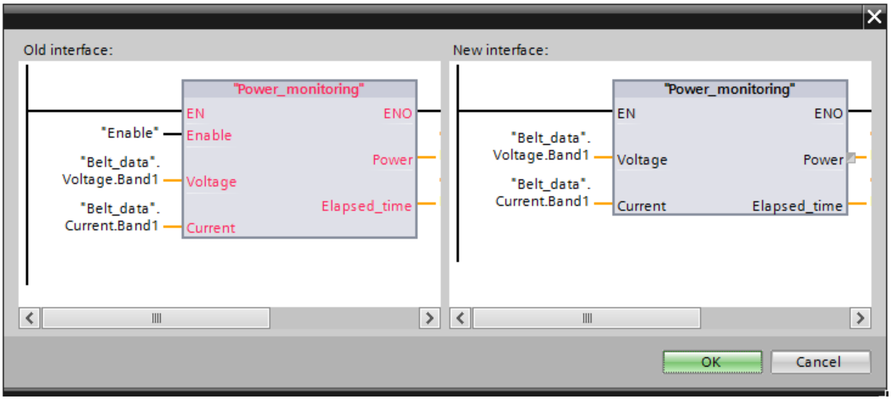


Fig. 6.17 Interface update in the case of faulty block calls

Under *Options* > *Settings* and *PLC programming* > *General* > *Compilation* you can select the *Delete actual parameters on interface update* option. The result is that an actual parameter is deleted when compiling or updating the interface if the associated block parameter has been deleted.

6.6 Program information

The following tools support you during programming and program testing:

- Cross-references
- Assignment list for inputs, outputs, bit memories, SIMATIC timer and SIMATIC counter functions
- Call and dependency structures
- Resources

You can start the individual tools at any time during programming, either in the main menu using the *Tools* > ... command or in the project tree by double-clicking *Program info* under a PLC station.

6.6.1 Cross-reference list

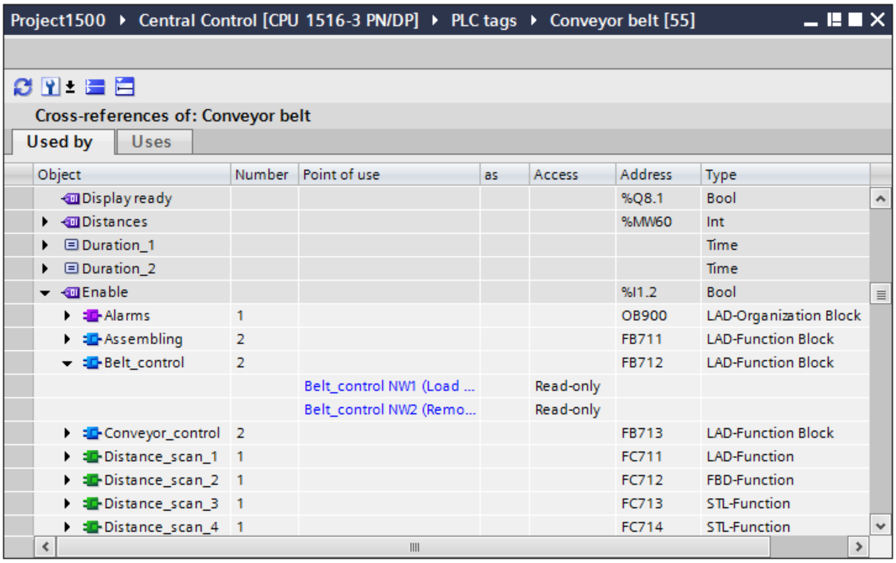
The cross-reference list indicates the use of tags and blocks in the user program. It provides an overview of

- Which objects have been used
- At which position in the program they have been used
- In what association they have been used, e.g. with which function a tag has been used

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-references* command from the shortcut 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. 6.18). 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 where the object is used. You can select the view options using the spanner icon in the toolbar of the cross-reference list: *Show used* and/or *Show unused*.



Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ PLC tags ▸ Conveyor belt [55]

Cross-references of: Conveyor belt

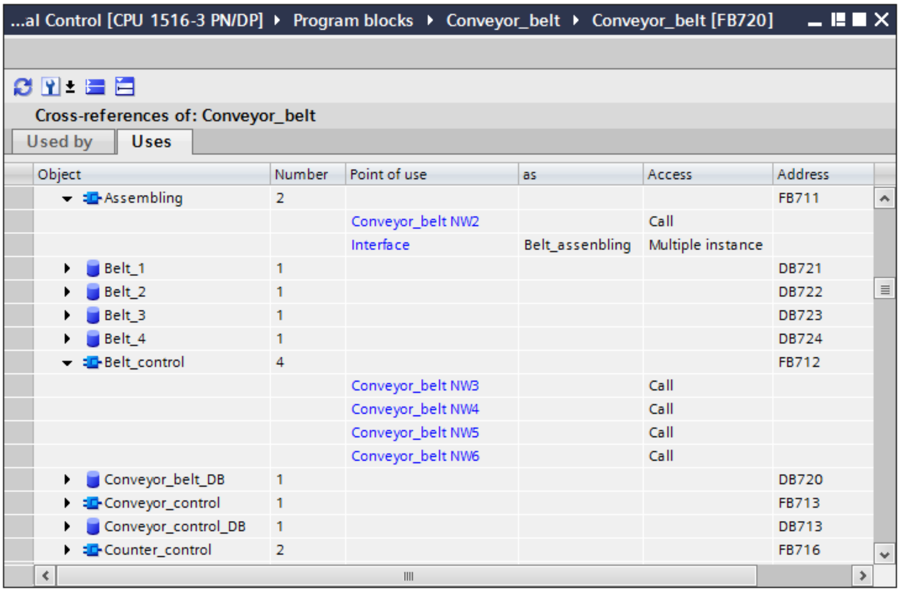
Used by | Uses

Object	Number	Point of use	as	Access	Address	Type
Display ready					%Q8.1	Bool
▸ Distances					%MW60	Int
▸ Duration_1						Time
▸ Duration_2						Time
▾ Enable					%I1.2	Bool
▸ Alarms	1				OB900	LAD-Organization Block
▸ Assembling	2				FB711	LAD-Function Block
▾ Belt_control	2				FB712	LAD-Function Block
		Belt_control NW1 (Load ...		Read-only		
		Belt_control NW2 (Remo...		Read-only		
▸ Conveyor_control	2				FB713	LAD-Function Block
▸ Distance_scan_1	1				FC711	LAD-Function
▸ Distance_scan_2	1				FC712	FBD-Function
▸ Distance_scan_3	1				FC713	STL-Function
▸ Distance_scan_4	1				FC714	STL-Function

Fig. 6.18 Example of a cross-reference list in the *Used by* view

Cross-reference list *Uses*

The *Uses* view displays the objects used by the referenced object. It shows which objects are used (Fig. 6.19). With a block, for example, it shows which blocks are called within it and which tags are used within it. If the list entries are expanded, a



...al Control [CPU 1516-3 PN/DP] ▸ Program blocks ▸ Conveyor_belt ▸ Conveyor_belt [FB720]

Cross-references of: Conveyor_belt

Used by | Uses

Object	Number	Point of use	as	Access	Address
▾ Assembling	2				FB711
		Conveyor_belt NW2		Call	
		Interface	Belt_assenbling	Multiple instance	
▸ Belt_1	1				DB721
▸ Belt_2	1				DB722
▸ Belt_3	1				DB723
▸ Belt_4	1				DB724
▾ Belt_control	4				FB712
		Conveyor_belt NW3		Call	
		Conveyor_belt NW4		Call	
		Conveyor_belt NW5		Call	
		Conveyor_belt NW6		Call	
▸ Conveyor_belt_DB	1				DB720
▸ Conveyor_control	1				FB713
▸ Conveyor_control_DB	1				DB713
▸ Counter_control	2				FB716

Fig. 6.19 Example of a cross-reference list in the *Uses* view

link in the *Point of use* 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 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), bit memories (M), SIMATIC timer functions (T), and SIMATIC counter functions (C). The use of operands as bit, byte, word or doubleword operands or tags is displayed. Peripheral inputs are assigned to the inputs operand area, and peripheral outputs to the outputs operand area.

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. 6.20).

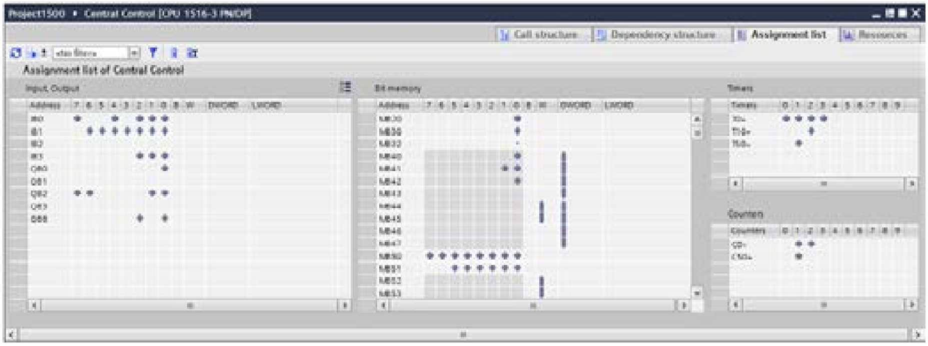

Project1500 ▸ Central Control [CPU 1516-3 PN/DP]
Call structure
Dependency structure
Assignment list
Resources
Assignment list of Central Control
Input, Output
Address 7 6 5 4 3 2 1 0 B W DWORD LWORD
IB0
IB1
IB2
IB3
QB0
QB1
QB2
QB3
IB8
Bit memory
Address 7 6 5 4 3 2 1 0 B W DWORD LWORD
MB30
MB30
MB32
MB40
MB41
MB42
MB43
MB44
MB45
MB46
MB47
MB50
MB51
MB52
MB53
Timers
Timers 0 1 2 3 4 5 6 7 8 9
Counters
Counters 0 1 2 3 4 5 6 7 8 9

Fig. 6.20 Example of an assignment list with I, Q, M, T, and C

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 doubleword operand, the entry has a gray background. You can use the *View options* icon 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. For the bit memories, symbols at the operands indicate up to what address the bit memories are retentive.

Display of timer and counter assignments

Use of the SIMATIC timer and counter functions is displayed in decades. All timer and counter operations are considered, e.g. also the scanning of a duration or the scanning of the counter status.

Filter

You can filter the display of the assignment list using the *Filter* icon in the toolbar. You specify which addresses (operands) you want to view: To select the operand area, activate the associated checkbox. You can select all addresses as the filter range (with an asterisk: *), an 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 the settings 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

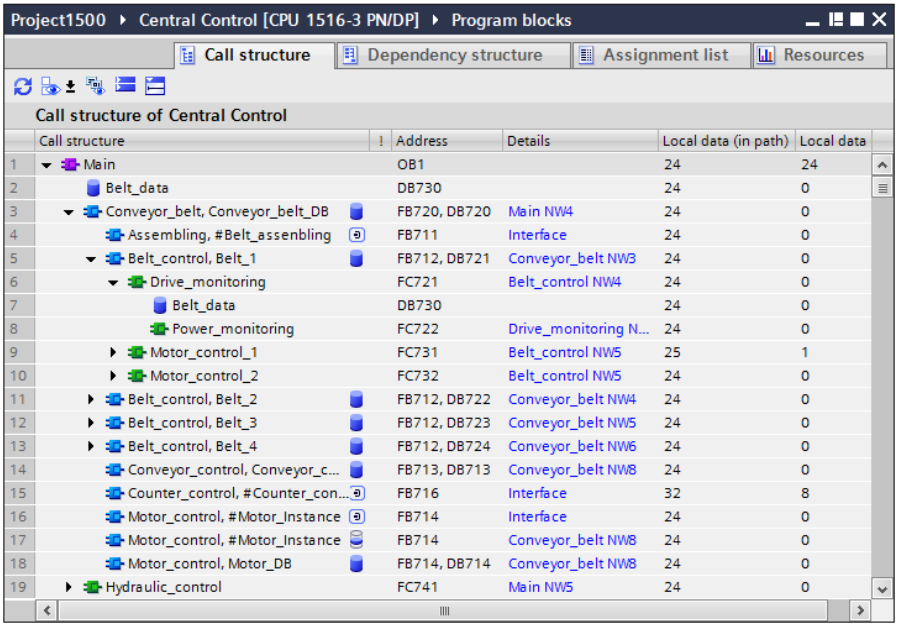
The call structure describes the call hierarchy of the blocks. To display the call structure, first select the *PLC station* or *Program blocks* folder for the entire program or for individual blocks, and then select *Call structure* from the shortcut menu or *Tools > Call structure* from the main menu.

The call structure shows the used blocks and the code blocks called from these blocks or the data blocks used in them (Fig. 6.21). The blocks which 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 editing with the program editor. The consistency check for the block calls is described in Chapter 6.6.5 “Consistency check” on page 283.

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 calls of non-existent blocks. *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.



Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ Program blocks
Call structure
Dependency structure
Assignment list
Resources

Call structure of Central Control

	Call structure	!	Address	Details	Local data (in path)	Local data
1	Main		OB1		24	24
2	Belt_data		DB730		24	0
3	Conveyor_belt, Conveyor_belt_DB		FB720, DB720	Main NW4	24	0
4	Assembling, #Belt_assenbling		FB711	Interface	24	0
5	Belt_control, Belt_1		FB712, DB721	Conveyor_belt NW3	24	0
6	Drive_monitoring		FC721	Belt_control NW4	24	0
7	Belt_data		DB730		24	0
8	Power_monitoring		FC722	Drive_monitoring N...	24	0
9	Motor_control_1		FC731	Belt_control NW5	25	1
10	Motor_control_2		FC732	Belt_control NW5	24	0
11	Belt_control, Belt_2		FB712, DB722	Conveyor_belt NW4	24	0
12	Belt_control, Belt_3		FB712, DB723	Conveyor_belt NW5	24	0
13	Belt_control, Belt_4		FB712, DB724	Conveyor_belt NW6	24	0
14	Conveyor_control, Conveyor_c...		FB713, DB713	Conveyor_belt NW8	24	0
15	Counter_control, #Counter_con...		FB716	Interface	32	8
16	Motor_control, #Motor_Instance		FB714	Interface	24	0
17	Motor_control, #Motor_Instance		FB714	Conveyor_belt NW8	24	0
18	Motor_control, Motor_DB		FB714, DB714	Conveyor_belt NW8	24	0
19	Hydraulic_control		FC741	Main NW5	24	0

Fig. 6.21 Example of the call structure

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.

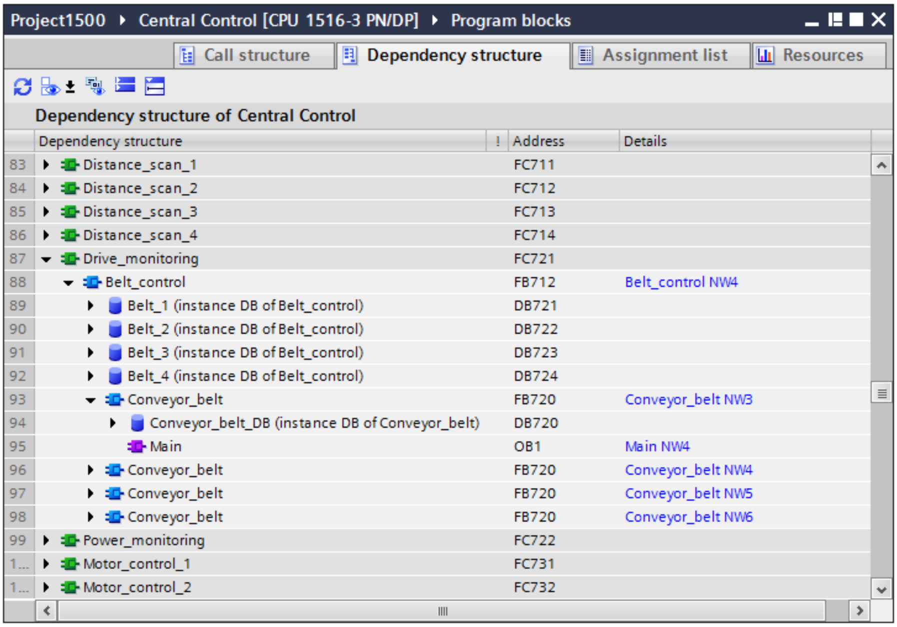
For each code block the dependency structure shows the block from which it is called, and for each data block the code block in which it is used (Fig. 6.22).

From the dependency structure, you can display the cross-reference information or open a block for processing with the program editor. The consistency check for the block calls is described in the next Chapter 6.6.5 “Consistency check” on page 283.

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 calls of non-existent blocks. *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.

6.6.5 Consistency check

Clicking on the *Consistency check* icon in the toolbar of the call or dependency structure displays block calls with an “interface conflict”. These are calls of blocks whose interface has been subsequently changed e.g. by assignment of a different data type to a block parameter or by modification of the static local data for function blocks.


Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ Program blocks
Call structure
Dependency structure
Assignment list
Resources

Dependency structure of Central Control

	Dependency structure	!	Address	Details
83	▸ Distance_scan_1		FC711	
84	▸ Distance_scan_2		FC712	
85	▸ Distance_scan_3		FC713	
86	▸ Distance_scan_4		FC714	
87	▾ Drive_monitoring		FC721	
88	▾ Belt_control		FB712	Belt_control NW4
89	▸ Belt_1 (instance DB of Belt_control)		DB721	
90	▸ Belt_2 (instance DB of Belt_control)		DB722	
91	▸ Belt_3 (instance DB of Belt_control)		DB723	
92	▸ Belt_4 (instance DB of Belt_control)		DB724	
93	▾ Conveyor_belt		FB720	Conveyor_belt NW3
94	▸ Conveyor_belt_DB (instance DB of Conveyor_belt)		DB720	
95	Main		OB1	Main NW4
96	▸ Conveyor_belt		FB720	Conveyor_belt NW4
97	▸ Conveyor_belt		FB720	Conveyor_belt NW5
98	▸ Conveyor_belt		FB720	Conveyor_belt NW6
99	▸ Power_monitoring		FC722	
1...	▸ Motor_control_1		FC731	
1...	▸ Motor_control_2		FC732	

Fig. 6.22 Example of dependency structure

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 eliminated 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 call block, the program editor shows what the updated call block will look like in the *Interface update* window. You can then carry out corrections and supplements in this window, for example if a new block parameter has been added.

6.6.6 Resources of the CPU

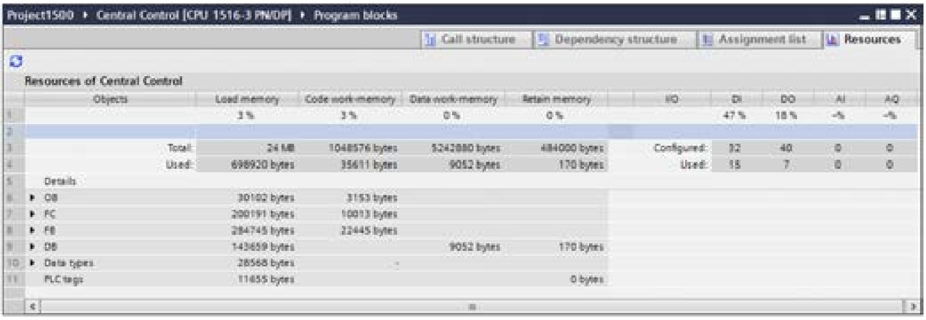
Under Resources you can see the utilization of the user memory and of the existing input/output modules (Fig. 6.23). To display the resources, select the folder *PLC station*, *Program blocks* or individual blocks and double-click on *Program info* in the project tree under the PLC station or select *Tools* > *Resources* from the main menu.

The resources function shows in four columns the maximum available and actually utilized storage space of the load memory, of the work memory for code and data, and of the retentive memory. You can see the utilization for each type of block, for individual blocks, for the PLC data types, and for the PLC tags. For blocks that have not yet been compiled, a question mark stands in place of the block size.

If a value for the memory utilization is displayed in red, the available memory space is overwritten. For the load memory, select the size of the memory card used from the drop-down list in the *Total* cell.

The existing (configured) input/output modules are divided according to DI, DO, AI and AQ, 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.


Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ Program blocks
Call structure
Dependency structure
Assignment list
Resources

Resources of Central Control

	Objects	Load memory	Code work-memory	Data work-memory	Retain memory	I/O	DI	DO	AI	AQ
1		3 %	3 %	0 %	0 %		47 %	18 %	-%	-%
2										
3	Total:	24 MB	1048576 bytes	5242880 bytes	484000 bytes	Configured:	32	40	0	0
4	Used:	698920 bytes	35611 bytes	9052 bytes	170 bytes	Used:	15	7	0	0
5	Details									
6	▸ OB	30102 bytes	3153 bytes							
7	▸ FC	200191 bytes	10013 bytes							
8	▸ FB	284745 bytes	22445 bytes							
9	▸ DB	143659 bytes		9052 bytes	170 bytes					
10	▸ Data types	28568 bytes	-							
11	PLC tags	11655 bytes			0 bytes					

Fig. 6.23 Example of display of resources

6.7 Language settings

STEP 7 gives you 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 station (of the HMI device)

The settings of the different languages are independent of one another.

Language setting of the operating system

If you are working with a multilingual version of the operating system (MUI variant), set the desired character set using the Windows control panel.

Language of the user interface and mnemonic

STEP 7 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* set “E” stands for “Eingang”, and with the *International* set, “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.

The 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. It provides examples of how the programming functions are represented in the ladder logic. You can find a description of the individual functions, e.g. comparison functions, in Chapters 12 “Basic functions” on page 503, 13 “Digital functions” on page 558, and 14 “Program control” on page 622.

Use of the program and symbol editor, which generally applies to all programming languages, is described in Chapter 6 “Program editor” on page 247.

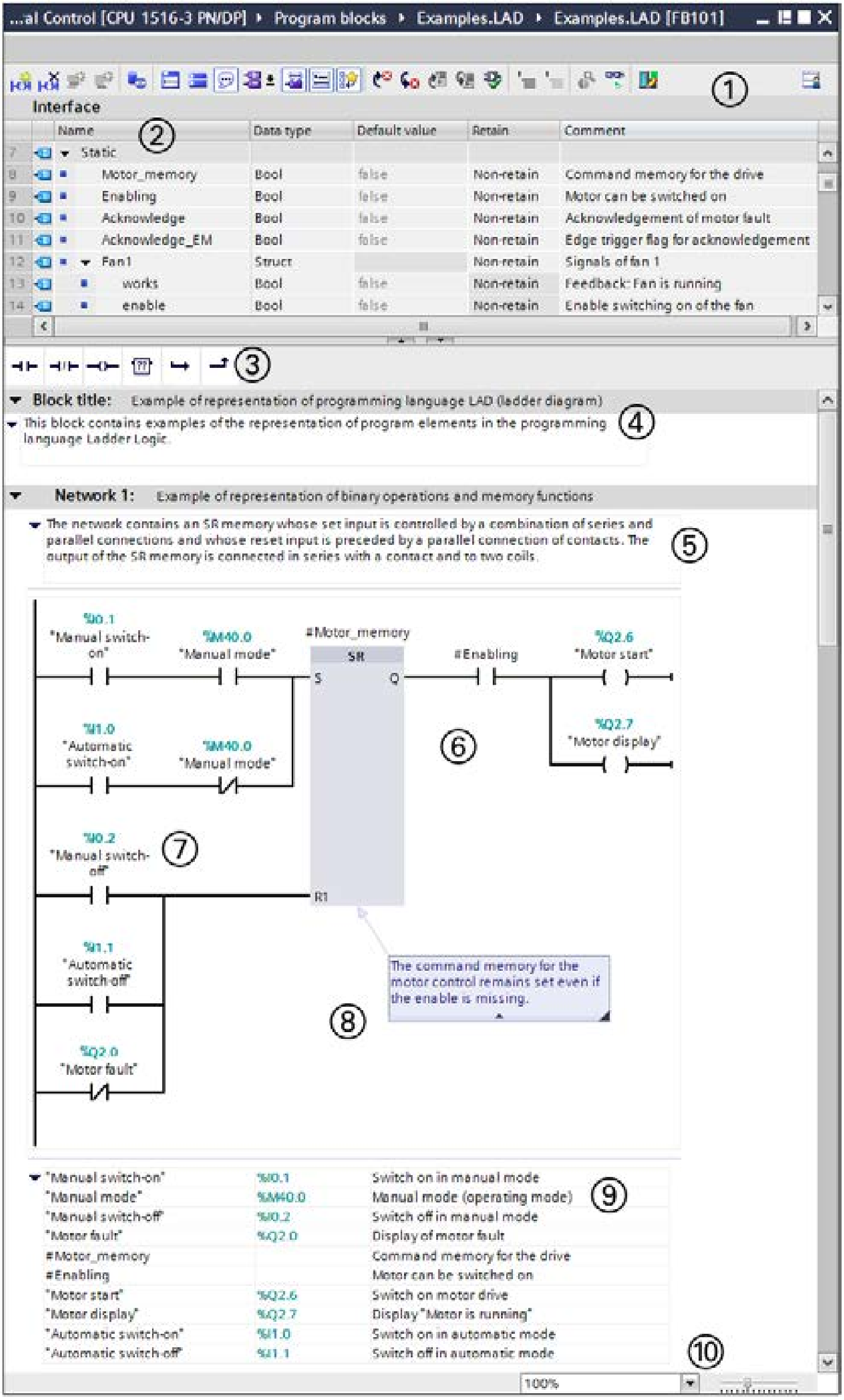
Ladder logic is used to program the contents of blocks (the user program). What blocks are, and how they are created, is described in Chapters 5.3.1 “Block types” on page 155 and 6.3 “Programming a code block” on page 253.

7.1.1 Programming with LAD in general

You use LAD to program the control function of the programmable controller – the user program (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 which may also have an extremely complex structure. Each network is terminated by at least one coil or box.

Fig. 7.1 shows the program editor's working window. The icons in the toolbar ① can be used to set the display of the working area, e.g. the display of the network comments and additional functions such as monitoring of the program status. The interface of the block ② in the upper part of the working window lists the block parameters and local data. The favorites bar ③ can be expanded by additional program elements. It can also be hidden. Each block has a heading, the block title, and a block comment ④, which can be used to explain the function of the block. These are followed by the first network with its number, heading and comment ⑤.

The control function, i.e. the interconnection of the program elements, is displayed in the working area ⑥. The tags can be displayed absolutely, symbolically, or with both addressing types ⑦. Each coil and each box can be assigned a “free-form” comment ⑧. The tag information ⑨ shows the tags used in the network with the tag comments. Like the network comment and the free comments, it can be hidden. With the zoom setting ⑩ the display of the control function can be adapted to the size of the working area.



Interface

	Name	Data type	Default value	Retain	Comment
7	▾ Static				
8	Motor_memory	Bool	false	Non-retain	Command memory for the drive
9	Enabling	Bool	false	Non-retain	Motor can be switched on
10	Acknowledge	Bool	false	Non-retain	Acknowledgement of motor fault
11	Acknowledge_EM	Bool	false	Non-retain	Edge trigger flag for acknowledgement
12	▾ Fan1	Struct		Non-retain	Signals of fan 1
13	works	Bool	false	Non-retain	Feedback: Fan is running
14	enable	Bool	false	Non-retain	Enable switching on of the fan

Block title: Example of representation of programming language LAD (ladder diagram)

This block contains examples of the representation of program elements in the programming language Ladder Logic.

Network 1: Example of representation of binary operations and memory functions

The network contains an SR memory whose set input is controlled by a combination of series and parallel connections and whose reset input is preceded by a parallel connection of contacts. The output of the SR memory is connected in series with a contact and to two coils.

The command memory for the motor control remains set even if the enable is missing.

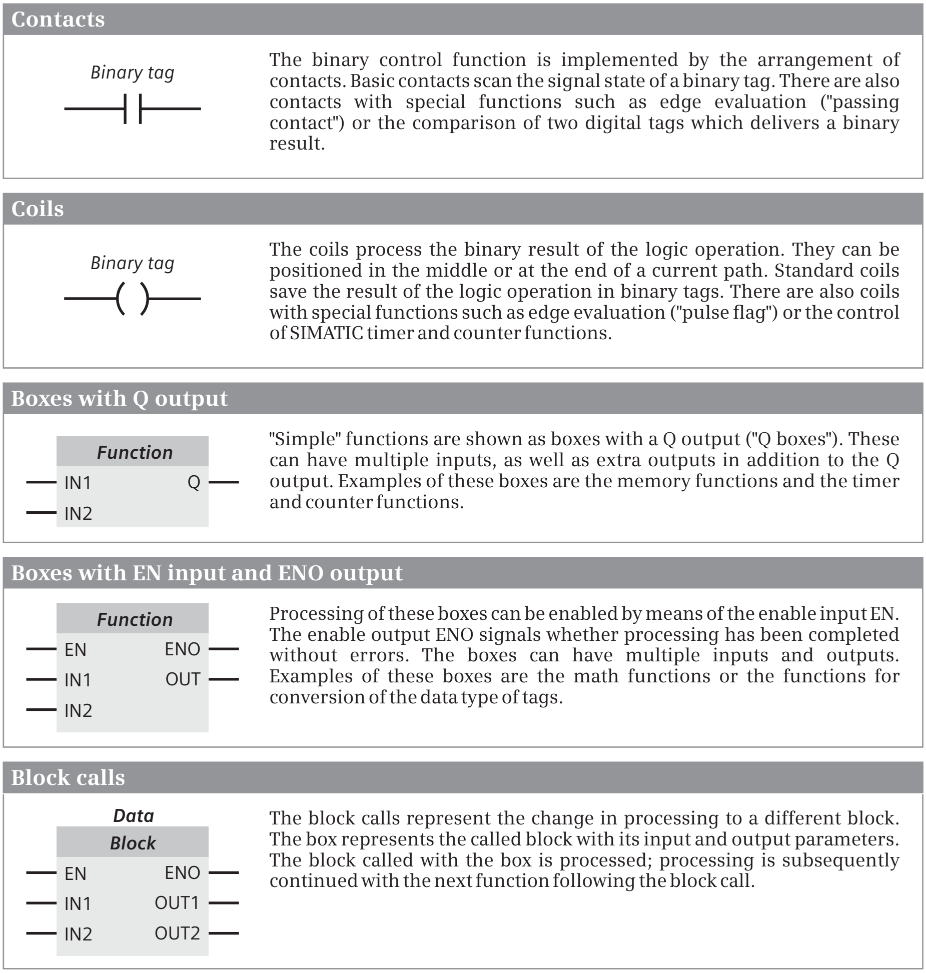
"Manual switch-on"	%I0.1	Switch on in manual mode
"Manual mode"	%M40.0	Manual mode (operating mode)
"Manual switch-off"	%I0.2	Switch off in manual mode
"Motor fault"	%Q2.0	Display of motor fault
#Motor_memory		Command memory for the drive
#Enabling		Motor can be switched on
"Motor start"	%Q2.6	Switch on motor drive
"Motor display"	%Q2.7	Display "Motor is running"
"Automatic switch-on"	%I1.0	Switch on in automatic mode
"Automatic switch-off"	%I1.1	Switch off in automatic mode

Fig. 7.1 Working window of the program editor for LAD programming

The program editor establishes an LAD network in accordance with the principle of the “main current path”: This is the highest branch which commences directly at 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.

“Open” parallel branches can lead out from the main current path. If they do not lead back to the main current path, they are called “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.



The binary control function is implemented by the arrangement of contacts. Basic contacts scan the signal state of a binary tag. There are also contacts with special functions such as edge evaluation ("passing contact") or the comparison of two digital tags which delivers a binary result.

The coils process the binary result of the logic operation. They can be positioned in the middle or at the end of a current path. Standard coils save the result of the logic operation in binary tags. There are also coils with special functions such as edge evaluation ("pulse flag") or the control of SIMATIC timer and counter functions.

"Simple" functions are shown as boxes with a Q output ("Q boxes"). These can have multiple inputs, as well as extra outputs in addition to the Q output. Examples of these boxes are the memory functions and the timer and counter functions.

Processing of these boxes can be enabled by means of the enable input EN. The enable output ENO signals whether processing has been completed without errors. The boxes can have multiple inputs and outputs. Examples of these boxes are the math functions or the functions for conversion of the data type of tags.

The block calls represent the change in processing to a different block. The box represents the called block with its input and output parameters. The block called with the box is processed; processing is subsequently continued with the next function following the block call.

Fig. 7.2 Overview of ladder logic program elements

A block is not terminated by a special network or function, you simply finish the program input. 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.

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 binary logic operations with LAD

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

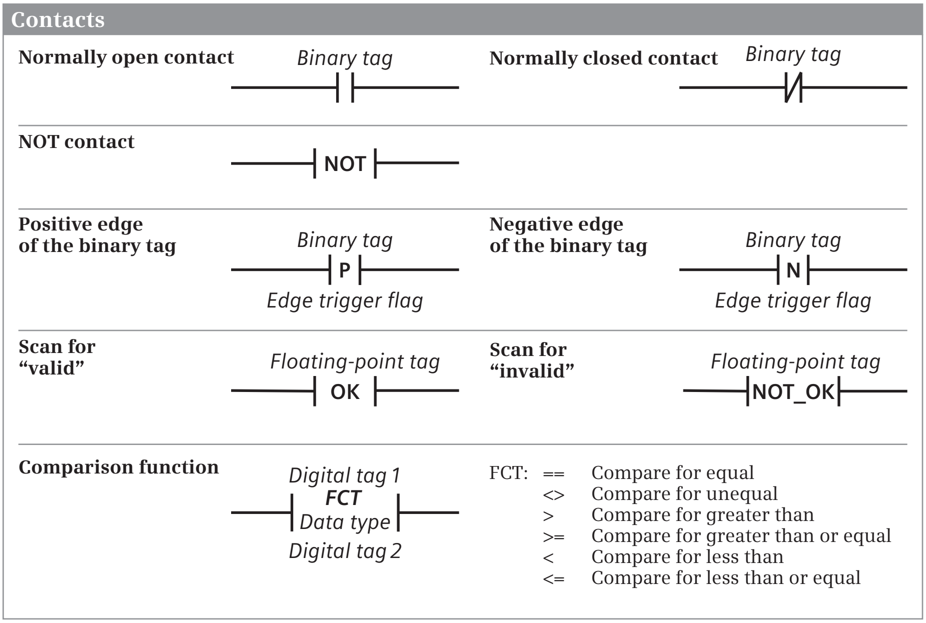


Fig. 7.3 Overview of the contacts described in this chapter

for a binary tag, validity checking of floating-point numbers, and the comparison function (Fig. 7.3).

7.2.1 NO and NC contacts

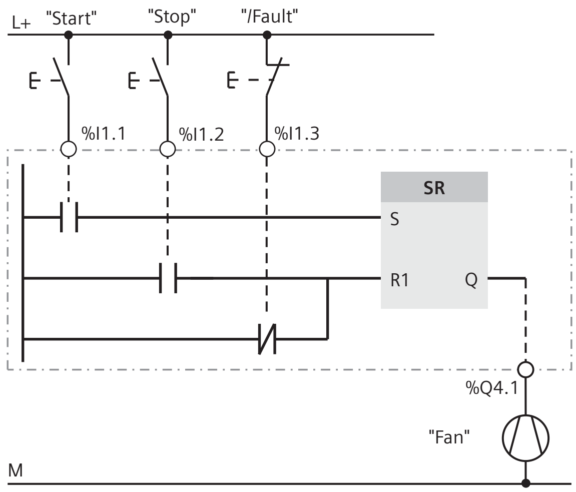
An NO or NC contact is used to scan the signal state of a binary tag. An NO contact passes on the scanned signal state directly to the logic operation, an NC contact first negates the signal state.

To program a contact, drag it with the mouse from the program elements catalog under *Basic instructions* > *Bit logic operation* to the working area. You can subsequently change the function (NO or NC contact) using a drop-down list which you can open using the small yellow triangle when the contact is selected.

You write the binary tag to be scanned above the contact. This can be an input, output, bit memory or data bit, or also a SIMATIC timer or SIMATIC counter function. Assignment with a constant (TRUE or FALSE) is not permissible.

The example in Fig. 7.4 shows the two "Start" and "Stop" pushbuttons. When pressed, they output the signal state "1" in the case of an input module with sinking input. The SR function is set or reset with this signal state.

Example of application of NO and NC contacts



When pressed, the "Start" and "Stop" pushbuttons switch the fan on and off. They are "1-active" signals.
If "/Fault" becomes active, the fan is to be switched off and remain off. "/Fault" is a "0-active" signal. In order to reset the SR function with signal state "1", this input is scanned with an NC contact (scan for signal state "0").
In the example, the 0-active signal is identified by a slash in front of the name.

Fig. 7.4 Principle of operation of NO and NC contacts

The "/Fault" signal is not active in the normal case. Signal state "1" is then present and is negated by scanning with an NC contact, and the SR function therefore remains uninfluenced. If "/Fault" becomes active, the SR function is to be reset. The active signal "/Fault" delivers signal state "0", which resets the SR function by means of the scan with an NC contact as signal state "1".

7.2.2 Series and parallel 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 ("AND function", see Chapter 12.1.3 "AND function, series connection" on page 507).

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 ("OR function", see Chapter 12.1.4 "OR function, parallel connection" on page 507).

Series and parallel connections can be combined. If contacts arranged in parallel are connected in series to other contacts arranged in parallel (series connection of parallel connections), this corresponds to an AND logic operation on OR functions. An OR logic operation on AND functions is the parallel connection of series connections.

To program a branch, use the mouse to drag the symbol for *Open branch* or *Close branch* from the program elements catalog under *Basic instructions* > *General* into the current path. Gray boxes indicate the permissible positioning, a green box identifies the position at which the branch will be opened or closed if you release the mouse button. You close a branch if you drag the end of the branch to the position at which it is to be closed.

Fig. 7.5 shows a simple example of the interconnection of contacts. Two fans signal with signal state "1" that they are running. A coil is to be activated for display is only one fan is running. In the upper current path, the logic operation is as follows: (*#Fan1.works* AND not *#Fan2.works*) OR (not *#Fan1.works* AND *#Fan2.works*). The lower current path solves the task with the logic operation (*#Fan1.works* OR *#Fan2.works*) AND (not *#Fan1.works* OR not *#Fan2.works*).

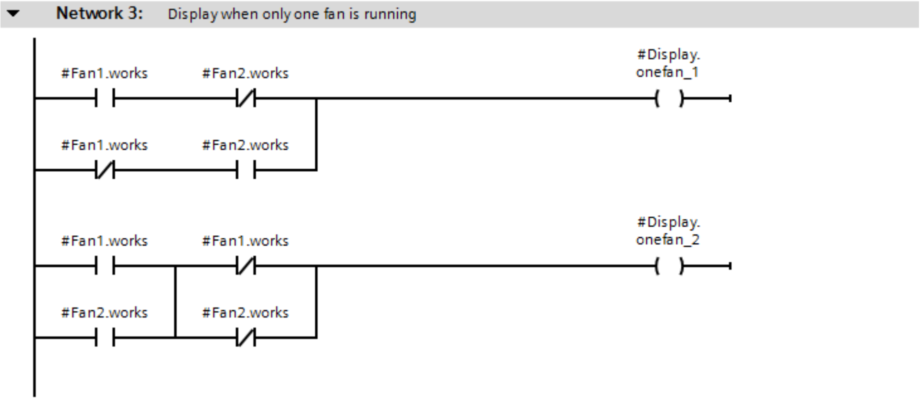


Fig. 7.5 Example of series and parallel connection of contacts

7.2.3 T branch, open parallel branch

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 having different logic operations, this is referred to as a “T branch” or an “open” parallel branch.

To program a T branch, use the mouse to drag the symbol for *Open branch* from the program elements catalog under *Basic instructions* > *General* to the position in the current path at which the T branch is to commence.

Fig. 7.6 shows a T branch. The parallel connection of *#Fan1.works* and *#Fan2.works* is followed by the branch in which a series connection of a NOT contact and an NO contact leads to a further coil.

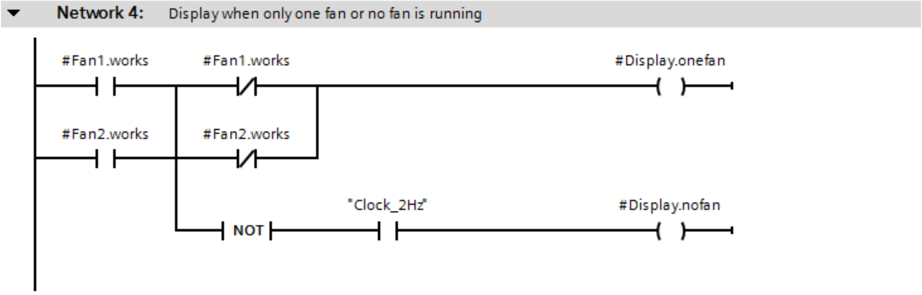


Fig. 7.6 Example of a T branch (open parallel branch) and the NOT contact

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.

7.2.4 Negate result of logic operation in the ladder logic

The NOT contact negates the result of the logic operation (the “current flow”).

To program a NOT contact, drag it with the mouse from the program elements catalog under *Basic instructions* > *Bit logic operation* to the working area.

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.

In Fig. 7.6 the parallel connection of *#Fan1.works* and *#Fan2.works* is negated. The resulting logic operation is: Not *#Fan1.works* AND not *#Fan2.works*. If no fan is working, the *#Display.nofan* tag flashes at 2 Hz.

7.2.5 Edge evaluation of a binary tag in the ladder logic

An edge evaluation detects the change in a binary signal.

To program an edge evaluation, drag the P or N contact with the mouse from the program elements catalog under *Basic instructions* > *Bit logic operation* to the working area.

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 “passing contact”. This “pulse” is linked to the result of the logic operation present prior to the contact.

The edge trigger flag is present underneath the edge contact. This is a memory 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 12.3 “Edge evaluation” on page 515).

The example in Fig. 7.7 shows an application of edge evaluation. Let us assume that an alarm has “arrived”, i.e. the alarm signal's state changes from “0” to “1”. Signal state “1” is then present after the P contact for one program cycle. The *#Alarm_memory* tag is set by this, and the *#Indicator_light* tag flashes at 0.5 Hz. The alarm memory can be reset using an *#Acknowledge* button. The alarm memory remains reset if *#Acknowledge* has signal state “0” again and *#Alarm_bit* is still present. *#Alarm_memory* is only set again by a further positive edge of *#Alarm_bit* (if *#Acknowledge* then no longer has signal state “1”).

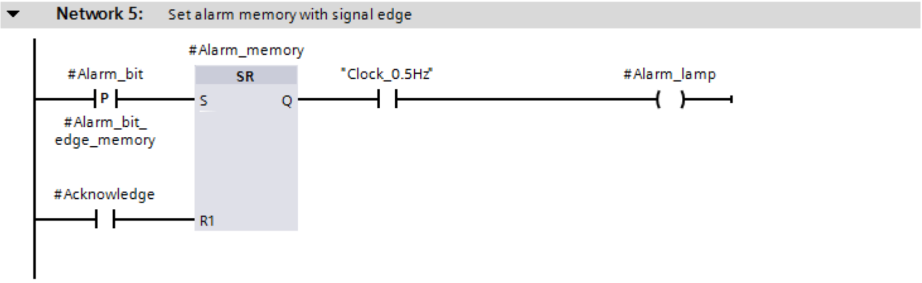


Fig. 7.7 Example of an edge evaluation of a binary tag

7.2.6 Validity check of a floating-point tag in the ladder logic

The OK contact checks a floating-point tag for validity, i.e. whether the range limits for the data type REAL or LREAL are adhered to. The contacts exists in two versions:

- ▷ The OK contact is closed if the tag is valid
- ▷ The NOT_OK contact is closed if the tag is outside the permissible value range

The OK/NOT_OK contact is programmed like a standard contact. You can find the OK/NOT_OK contact in the program elements catalog under *Basic instructions* > *Comparer*.

If in Fig. 7.8 the *#Measurement_from_sensor* tag is within the permissible value range for the data type REAL or LREAL and the *#Measurement.Registered* tag has signal state “1”, then *#Measured_value_ok* is set.

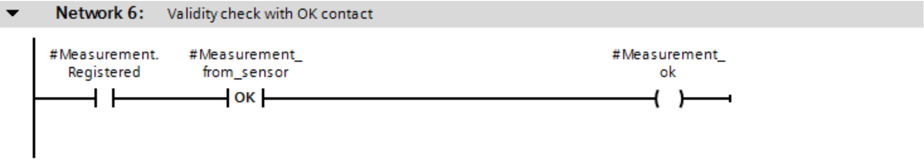


Fig. 7.8 Example of the validity check of a floating-point tag

7.2.7 Comparison contacts

A comparison contact compares two digital values and outputs a binary signal. 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. The comparison function is described in Chapter 13.3 “Comparison functions” on page 570.

To program a comparison function, drag it with the mouse from the program elements catalog under *Basic instructions* > *Comparator operations* to the working area. You position the comparison contact like a standard contact in the current path. You can then use drop-down lists to define the comparison (Fig. 7.9): If you select the comparison contact, you can set the comparison relation on the triangle in the upper right corner and the data type on the triangle in the lower right corner.

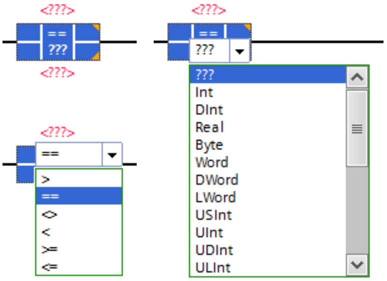


Fig. 7.9 Drop-down lists for setting the comparison mode and data type

Fig. 7.10 shows two comparison contacts connected in series. If the *#Measurement_temperature* tag is above a lower limit and below an upper limit, the coil is activated and the *#Measurement_in_range* tag is set.

7.3 Programming memory functions with LAD

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 with the negating coil.

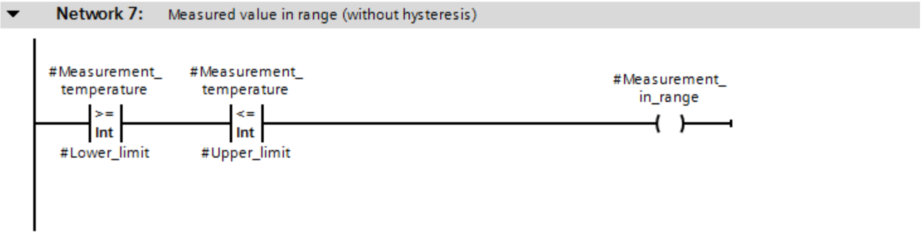


Fig. 7.10 Example of comparison contacts

There are coils with additional names and special functionalities such as the set and reset coils or the coil for pulse generation during evaluation of a signal edge. Coils can be used to set and reset bit arrays, start and reset timer functions, execute jumps in the program, and terminate blocks (Fig. 7.11). The jump functions and the block end function are described in Chapter 14 “Program control” on page 622.

7.3.1 Simple and negating coils

A simple coil directly assigns the current flow to the tag present on the coil: The tag is set to signal state “1” when current flows into the coil and is reset to signal state “0” when current no longer flows. The negating coil negates the current flow beforehand: The tag is set if no current flows into the coil and is reset if current flows.

For programming, drag the coil with the mouse from the program elements catalog under *Basic instructions > Bit logic operation* to the working area. Gray boxes indicate the permissible positioning, a green box identifies the position at which the coil will be inserted if you release the mouse button.

The simple and the negating coil require a preceding logic operation; they cannot be connected directly to the left-hand power rail. A coil can be positioned at the end of a current path, or in the middle. This also applies to a T branch. Positioning in a “closed” parallel branch is not permissible.

Simple and negating coils can be connected in series or – at the end of a current path – in parallel. Simple and negating coils do not change the result of logic operation (the “current flow”).

Fig. 7.12 shows the possible arrangements for a simple coil. In the current path, the *#Display.nofan*, *#Display.onefan* and *#Display.twofans* tags are controlled by simple coils. Two coils are connected in parallel at the end of the current path and respond in identical manners.

7.3.2 Set and reset coils

A set or reset coil is used to assign signal state “1” or “0” to a binary tag in the case of a result of logic operation “1”. A result of logic operation “0” has no effect.

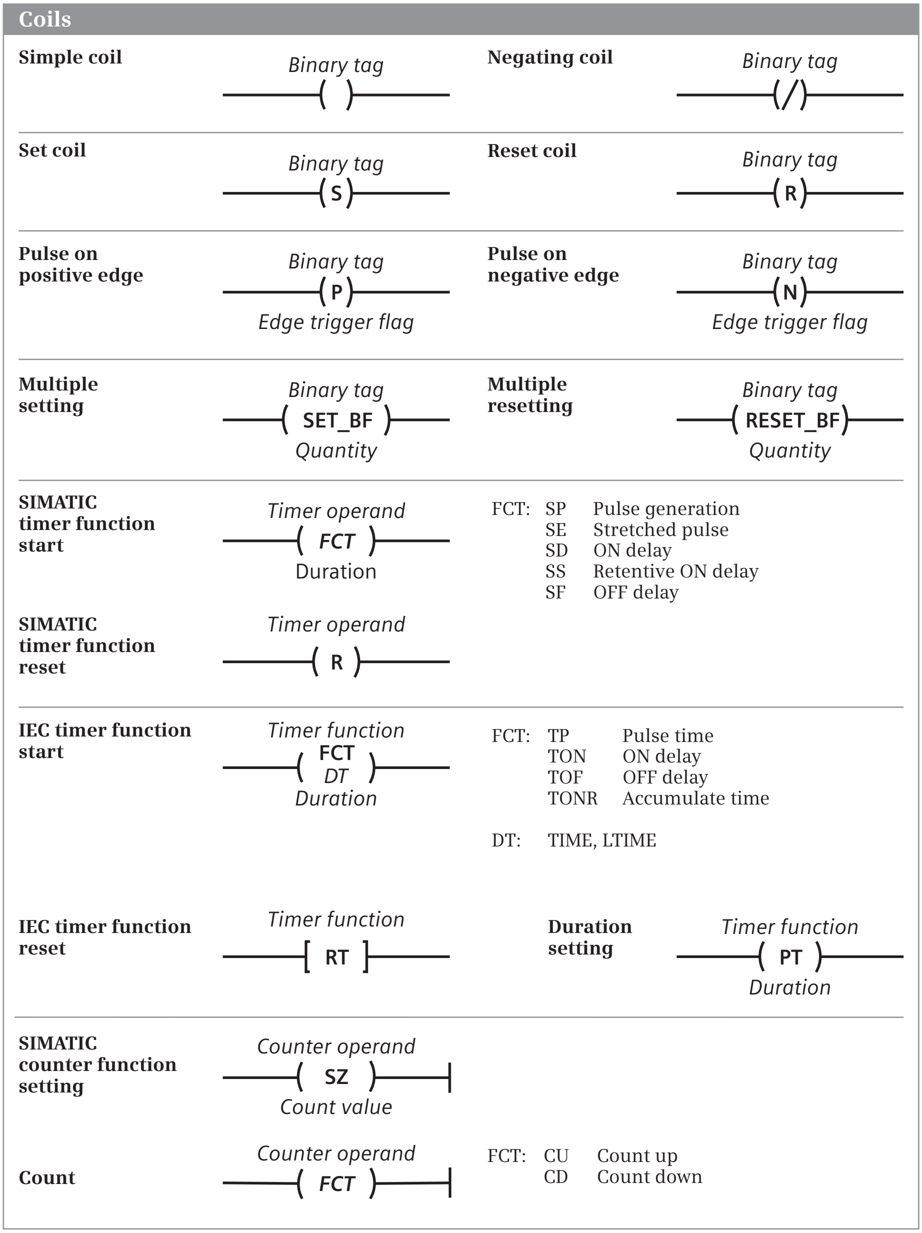


Fig. 7.11 Overview of the coils described in this chapter

For programming, drag the coil with the mouse from the program elements catalog under *Basic instructions > Bit logic operation* to the working area. Gray boxes indicate the permissible positioning, a green box identifies the position at which the coil will be inserted if you release the mouse button.

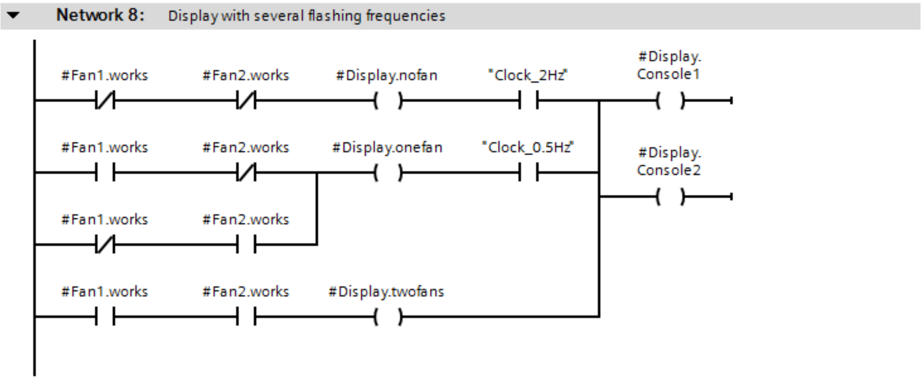


Fig. 7.12 Example of arrangement for a simple coil

Set and reset coils require a preceding logic operation and terminate a current path. The reset coil can also be used to reset a SIMATIC timer/counter function.

In Fig. 7.13, *#Fan1.start* with signal state “1” sets the *#Fan1.drive* tag. With signal state “1” at *#Fan1.stop*, *#Fan1.drive* is reset. As a result of positioning of the reset coil after the set coil, the memory response is “reset dominant”: If both contacts have signal state “1”, *#Fan1.drive* is reset or remains reset.

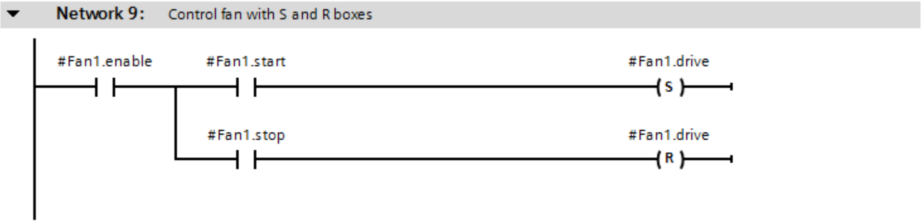


Fig. 7.13 Example of set and reset coils

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 logic. 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. 7.14 shows both types of memory function through latching, namely set dominant and reset dominant. Upper current path: If *#Fan2.start* closes, *#Fan2.drive* has signal state “1” and closes the contact parallel to *#Fan2.start*. If *#Fan2.start* then opens again, *#Fan2.drive* remains switched on. *#Fan2.drive* is switched off if *#Fan2.stop* opens. If signal state “1” is present at both *#Fan2.start* and *#Fan2.stop*,

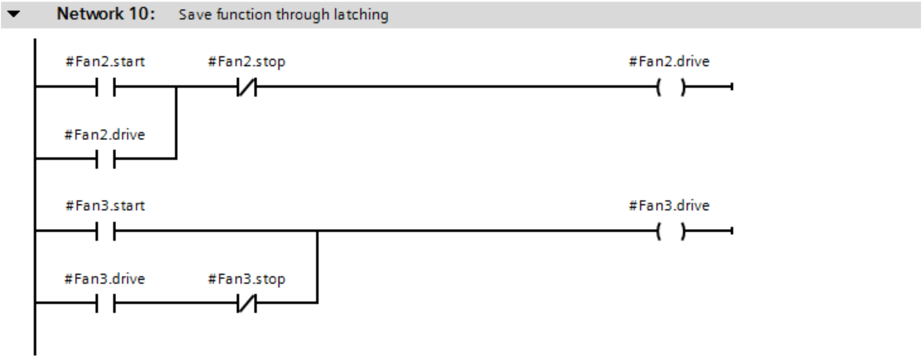


Fig. 7.14 Retentive response due to latching

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 *#Fan3.start* and *#Fan3.stop*, current flows into the coil (set dominant).

7.3.4 Edge evaluation with pulse output in the ladder logic

The P coil and N coil are available for edge evaluation with coils. The binary tag located above the P coil is set for the duration of one program cycle if the signal state changes from "0" to "1" before the P coil (rising edge). For the N coil, the binary tag located above the coil is set for the duration of one program cycle for a falling edge.

The binary tag present above the coil is referred to as a "pulse flag". Suitable for pulse flags are, for example, tags from the bit memory or data area. The edge trigger flag is present under the coil and must be a different tag for each edge evaluation (see Chapter 12.3 "Edge evaluation" on page 515).

Edge coils can be arranged inside a current path or they can terminate a current path. Edge coils can also be programmed following a T branch. A direct connection to the left-hand power rail does not make sense.

If additional program elements follow an edge coil, for example if the edge coil has been placed inside a current path, then the signal state at the input of the edge coil is passed on directly to the output of the coil.

In Fig. 7.15, if "current" is flowing through the series connection of *#Enable* and *#Measurement.Registered*, the *#Measurement.Load* tag has signal state "1" for the duration of one program cycle. The *#Measurement.Load_EM* tag is the edge trigger flag for edge evaluation.

7.3.5 Multiple setting and resetting (filling the bit array) in the ladder logic

If the result of logic operation is "1", the SET_BF coil sets the bits of a bit array to signal state "1". The bit array is defined by the start tag above the coil and the num-

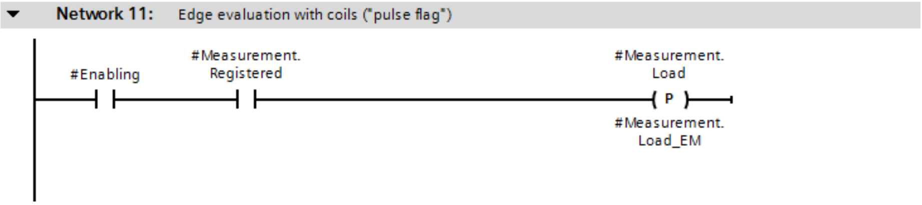


Fig. 7.15 Edge evaluation with coils ("pulse flag")

ber of bits indicated below the coil. If the result of logic operation is "1", the RESET_BF coil resets the bits in the bit array. With result of logic operation "0", there is no response in both cases.

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.

In Fig. 7.16, with a rising edge from *#Acknowledge*, 16 bits from *"Data.LAD". Alarm_bit[0]* are set to signal state "1" and 8 bits from *"Output 1"* are reset to signal state "0". *"Data.LAD".Alarm_bit* is a tag with the data type ARRAY OF BOOL, *"Output 1"* is a bit in the outputs operand area.

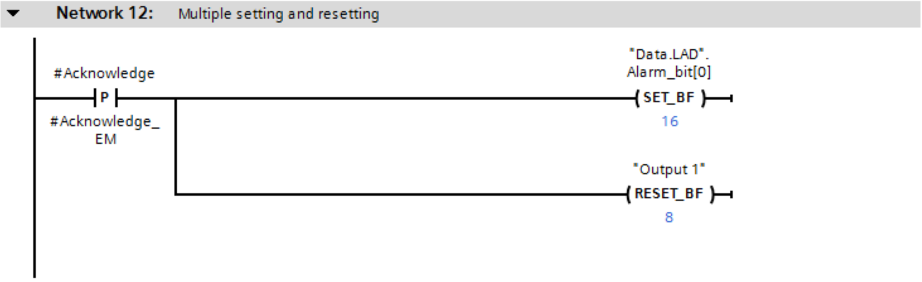


Fig. 7.16 Filling a bit array with SET_BF and RESET_BF

7.3.6 Coils with time response

Starting a SIMATIC timer function

A coil with a time response starts a SIMATIC timer function with a response as described in Chapter 12.4 "SIMATIC timer functions" on page 524. Available are starting as pulse (SP), as extended pulse (SE), as ON delay (SD), as retentive ON delay (SS), and as OFF delay (SF).

For programming, drag the corresponding with the mouse from the program elements catalog under *Basic instructions* > *Timer operations* to the working area.

A coil with time response requires a preceding logic operation. It can be placed in the middle or at the end of a current path and in a T branch. The timer operand from the SIMATIC timer functions (T) area is located above the coil. The time value is specified in the data format S5TIME underneath the coil.

A reset coil (R) can be used to reset a SIMATIC timer function. You can find the coil in the program elements catalog under *Basic instructions* > *Bit logic operation*. The reset coil can be positioned on the left-hand power rail, in the middle or at the end of a current path, and in a T branch.

In Fig. 7.17, the SIMATIC timer function *"Fan1.on-delay"* is started by the positive edge of *#Fan1.start* in the upper current path. Following expiry of the duration (3 s in the example), the fan *#Fan1.drive* is switched on. If *#Fan1.start* has signal state "0" prior to expiry of the duration, the fan is not switched on.

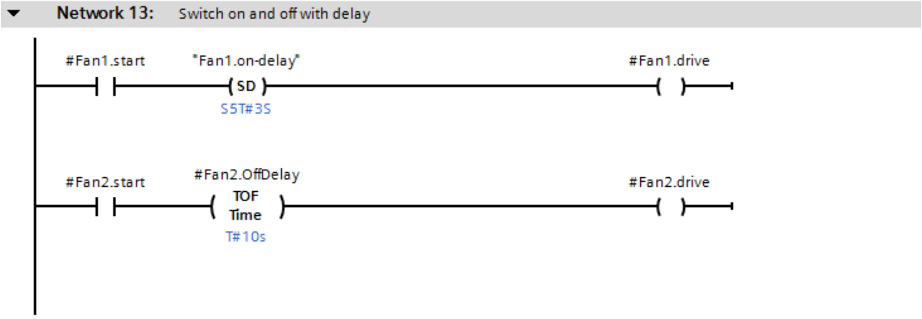


Fig. 7.17 Example of coils with time response

Controlling an IEC timer function

A coil with a time response controls an IEC timer function with a response as described in Chapter 12.4 "SIMATIC timer functions" on page 524. Starting as pulse (TP), as ON delay (TON), as OFF delay (TOF), as accumulating ON delay (TONR), and loading a time function with a duration (PT) and resetting a time function (RT) are available.

For programming, drag the corresponding with the mouse from the program elements catalog under *Basic instructions* > *Timer operations* to the working area.

A coil with time response requires a preceding logic operation. It can be placed in the middle or at the end of a current path and in a T branch. The name of the timer function, either the data block for a single instance or the instance name for a local instance, is located above the coil. The function for the time response and the data type of the duration, which is specified under the coil, are located in the coil.

With the PT coil, an IEC timer function is loaded with the duration that is indicated under the coil. The RT coil resets an IEC timer function. The RT coil and the PT coil can be positioned on the left-hand power rail, in the middle or at the end of a current path, and in a T branch.

In Fig. 7.17, the timer function “*Fan2.OffDelay*” is started as OFF delay by a positive edge at *#Fan2.start* in the bottom current path. The coil *#Fan2.drive* has signal state “1” if *#Fan2.start* is switched on and 10 seconds after the switch-off.

7.3.7 Coils with counter response

Controlling a SIMATIC counter function

A coil with a counter response controls a SIMATIC counter function with a response as described in Chapter 12.6 “SIMATIC counter functions” on page 545. Setting a counter (SC), counting up (CU), and counting down (CD) are available.

For programming, drag the corresponding coil with the mouse from the program elements catalog under *Basic instructions* > *Counter operations* to the working area.

A coil with counter response requires a preceding logic operation. It can be placed in the middle or at the end of a current path and in a T branch. The counter operand from the SIMATIC counter (C) area is located above the coil. The count value in data format WORD is specified underneath the SC coil, where the numerical range extends from W#16#0000 to W#16#0999 or from C#000 to C#999.

The reset coil (R) can be used to reset a SIMATIC counter function. You can find the reset coil in the program elements catalog under *Basic instructions* > *Bit logic operation*. The reset coil can be positioned on the left-hand power rail, in the middle or at the end of a current path, and in a T branch.

Fig. 7.18 counts the switch-on processes of *#Fan1.start* with the SIMATIC counter function “*Fan1.quantity*”. The *#Acknowledge* signal resets the counter to 0.

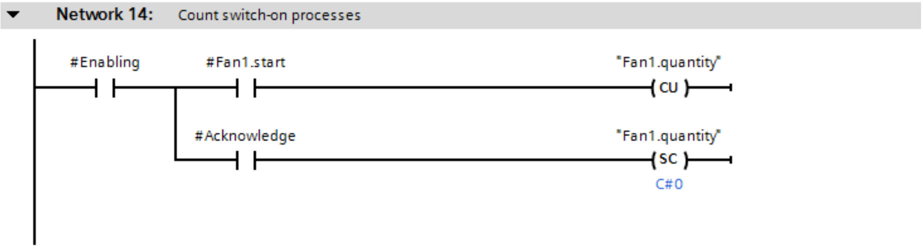


Fig. 7.18 Example of coils with counter response

7.4 Programming Q boxes with LAD

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. 7.19).

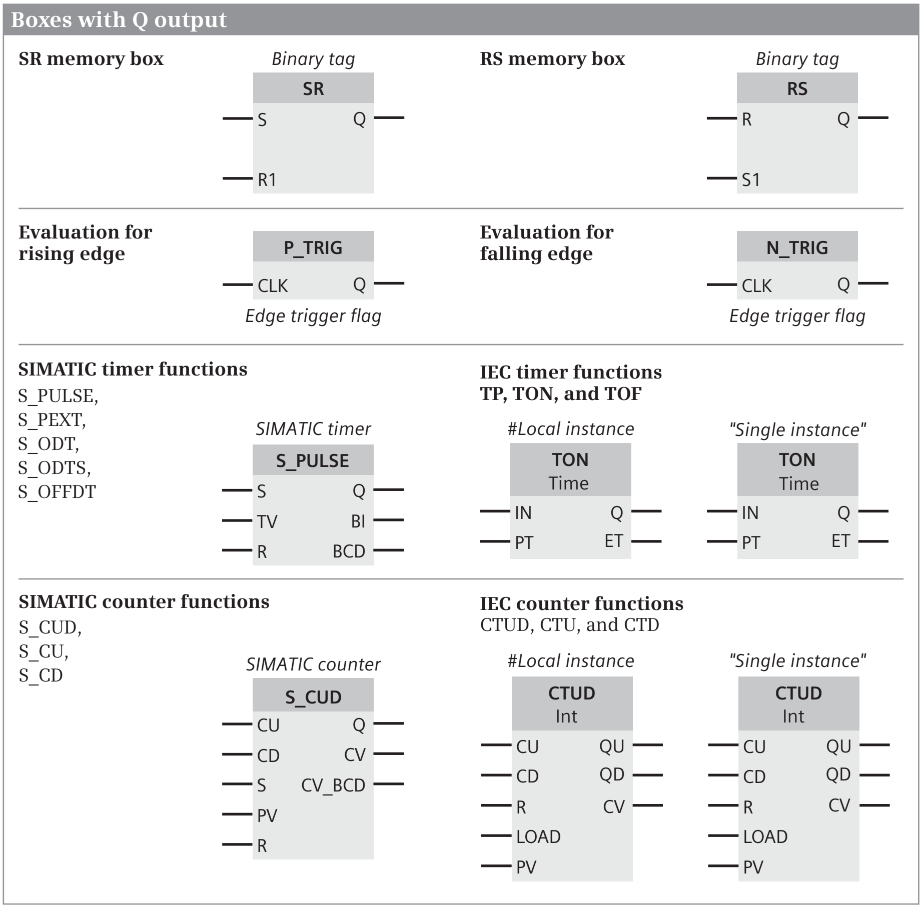


Fig. 7.19 Overview of Q boxes available with LAD

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.

When using Q boxes as program elements, you can:

- ▷ Program one single box per network, either within the current path or as its termination

- ▷ Arrange boxes in series by connecting the Q output of one box to a binary input of the following box
- ▷ Position boxes following T branches and in branches which commence on the left-hand power rail

7.4.1 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). With reset dominant, the memory function is reset or remains reset if both inputs have signal state "1". With set dominant, the memory function is set or remains set in such a case. The response of the memory box is described in Chapter 12.2 "Memory functions" on page 510.

For programming, drag the SR or RS symbol with the mouse from the program elements catalog under *Basic instructions* > *Bit logic operation* to the working area.

Fig. 7.20 shows a binary scaler: Each positive edge of the *#Bin_input* tag changes the signal state of *#Bin_output*. Thus half the input frequency is present at the output.

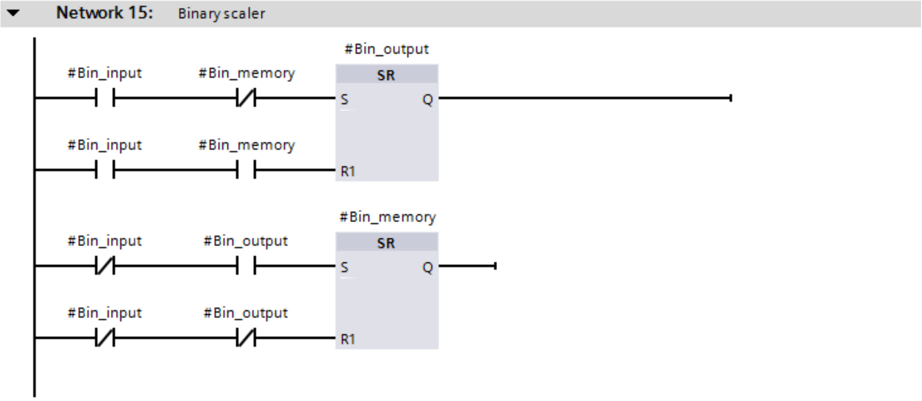


Fig. 7.20 Example of binary scaler

7.4.2 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. The response of the boxes for edge evaluation is described in Chapter 12.2 "Memory functions" on page 510.

For programming, drag the P_TRIG or N_TRIG symbol with the mouse from the program elements catalog under *Basic instructions* > *Bit logic operation* to the working area.

The edge boxes require a preceding logic operation and may only be positioned within a current path.

In Fig. 7.21, *#Measurement.Memory* is set if *#Measurement_temperature* reaches or exceeds an upper limit. In turn, the *#Measurement.Memory* tag sets the *#Measurement.Message* memory. Setting is carried out in both cases by a pulse with positive edge so that acknowledgment is also possible with a set signal present. Acknowledgment is also carried out by a pulse so that, with an acknowledgment signal present, the measured value memory and the alarm memory are set if the upper limit is exceeded again.

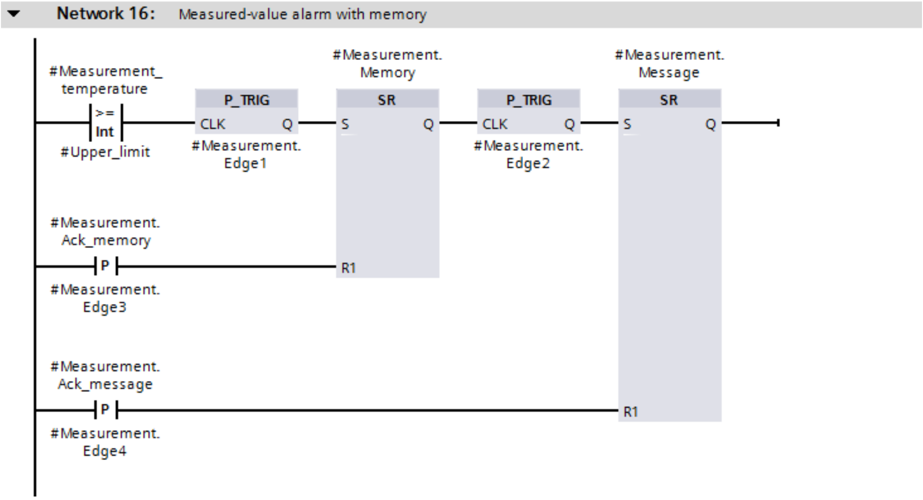


Fig. 7.21 Example of edge evaluations of current flow

7.4.3 SIMATIC timer functions in the ladder logic

A SIMATIC timer function can be started as pulse (S_PULSE), as extended pulse (S_PEXT), as ON delay (S_ODT), as retentive ON delay (S_ODTS), or as OFF delay (S_OFFDT). A detailed description of the timer response is provided in Chapter 12.4 “SIMATIC timer functions” on page 524.

For programming, drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions* > *Timer operations* to the working area. You can subsequently change the function using a drop-down list which you can open using the small yellow triangle when the box is selected.

The start input S and the time value TV must be connected; connection of the other box inputs and outputs is optional.

Fig. 7.22 shows a switch-on and switch-off delay. The timer function “*Fan3.on-delay*” is started by *#Fan3.start*. The output Q has signal state “1” after 3 s, which starts the timer function “*Fan3.off-delay*”. At the same time, the *#Fan3.drive* tag is set by the Q output of the box. The Q output continues to have signal state “1” for 10 s after *#Fan3.start* has signal state “0”.

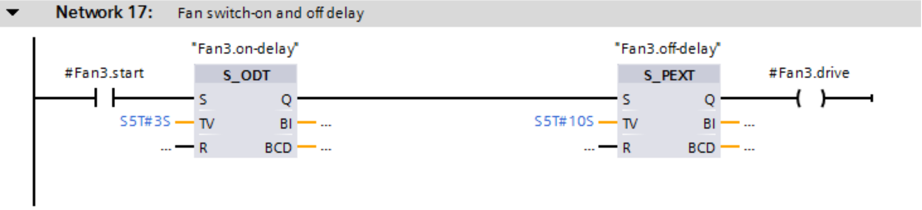


Fig. 7.22 Example of SIMATIC timer functions in the ladder logic

7.4.4 SIMATIC counter functions in the ladder logic

A SIMATIC counter function is available as up counter (S_CU), as down counter (S_CD), or as up/down counter (S_CUD). A detailed description of the counter response is provided in Chapter 12.6 “SIMATIC counter functions” on page 545.

For programming, drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions* > *Counter operations* to the working area. You can subsequently change the function using a drop-down list which you can open using the small yellow triangle when the box is selected.

At least one of the counter inputs (CU or CD) must be connected; connection of the other box inputs and outputs is optional.

Fig. 7.23 shows a down counter. The name of the SIMATIC counter used is positioned above the counter box. *#Quantity_set* sets the counter to the count value W#16#0120. The count value is reduced by 1 with each pulse from *#Workpart_identified*. Once zero has been reached, *#Quantity_reached* is set.

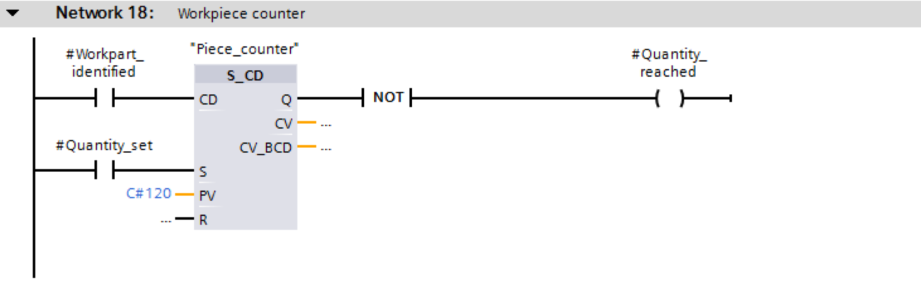


Fig. 7.23 Example of SIMATIC counter functions in the ladder logic

7.4.5 IEC timer functions in the ladder logic

An IEC timer function is available as pulse generation (TP), as ON delay (TON), as OFF delay (TOF), or as accumulating ON delay (TONR). A detailed description of the timer response is provided in Chapter 12.5 “IEC timer functions” on page 539.

For programming, drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions* > *Timer operations* to the working area. When positioning, you select either as single instance or – possible in a function block – as local instance (multi-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*.

You can subsequently change the timer function using a drop-down list which you can open using the small yellow triangle when the box is selected (not with TONR).

With the IEC timer functions, the IN input must have a preceding logic operation and a duration must be connected to the PT input. The Q output can be supplied with a coil, but cannot be linked further. You can also directly access the output parameters using the instance data, for example with *“<DB_name>”.Q* or *“<DB_name>”.ET* for a single instance.

Fig. 7.24 shows the IEC timer function *#MessageDelay*, which saves its data as local instance in the instance data block of the calling function block. If the *#Measurement_too_high* tag has signal state “1” for longer than 10 s, *#Message_too_high* is set.

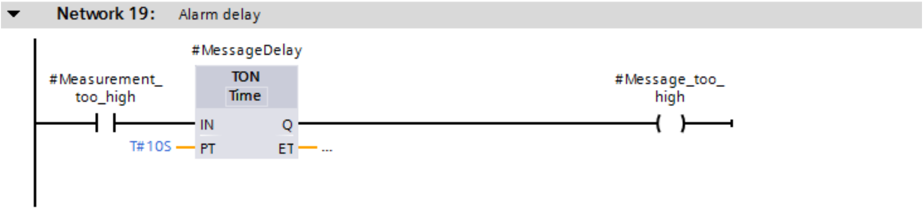


Fig. 7.24 Example of an IEC timer function

7.4.6 IEC counter functions in the ladder logic

An IEC counter function is available as up counter (CTU), as down counter (CTD), or as up/down counter (CTUD). A detailed description of the counter response is provided in Chapter 12.7 “IEC counter functions” on page 553.

For programming, drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions* > *Counter operations* to the working area. When positioning, you select either as single instance or – possible in a function block – as local instance (multi-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*.

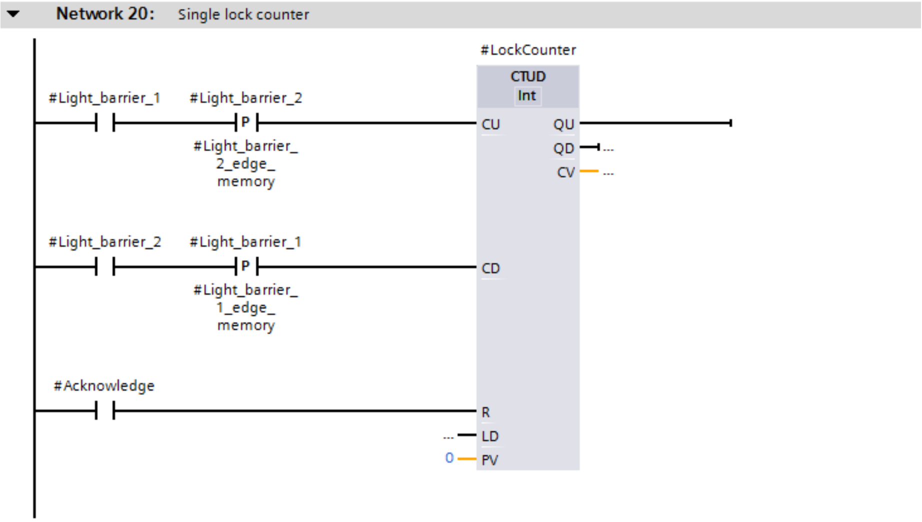


Fig. 7.25 Example of IEC counter functions

You can subsequently change the timer function using a drop-down list which you can open using the small yellow triangle when the box is selected.

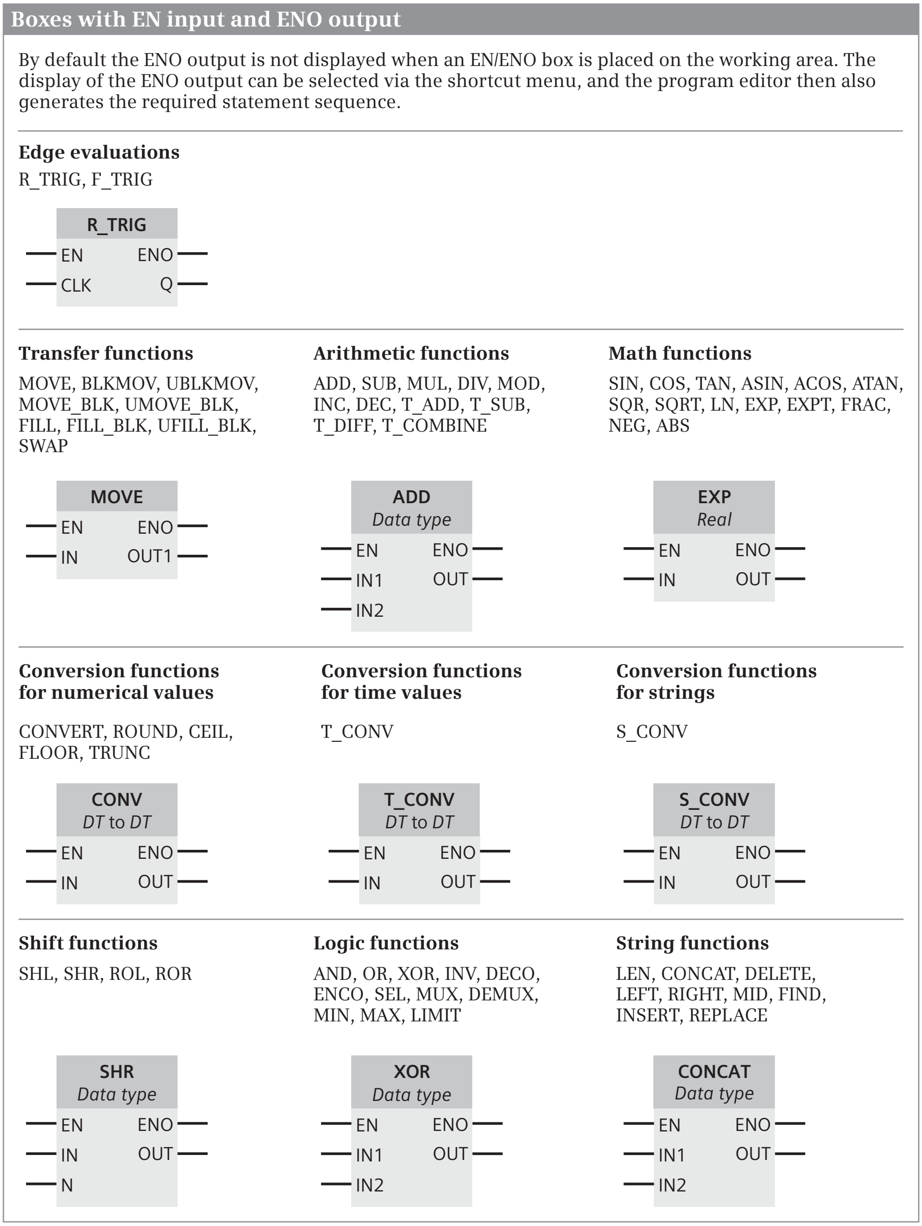
With the IEC counter functions, at least one counter input (CU or CD) must have a preceding logic operation. Connection of the other box inputs and outputs is optional. A coil can be positioned at the top output QU, but not a further logic operation. The QD output cannot be supplied, but can be scanned indirectly via the corresponding component *QD* of the counter structure. For the QU output, this would be the component *QU*.

You can also directly access the output parameters using the instance data, for example with *"<DB_name>".QD*.

Fig. 7.25 shows the IEC counter function *#LockCounter*, 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.

7.5 Programming EN/ENO boxes with LAD

EN/ENO boxes have an enable input EN and an enable output ENO. The enable input can be used to control processing of the box. If an error occurs while the box is being processed, this is displayed at the enable output. Fig. 7.26 provides an overview of the "basic" functions implemented with EN/ENO boxes.



Boxes with EN input and ENO output

By default the ENO output is not displayed when an EN/ENO box is placed on the working area. The display of the ENO output can be selected via the shortcut menu, and the program editor then also generates the required statement sequence.

Edge evaluations

R_TRIG, F_TRIG

Transfer functions

MOVE, BLKMOV, UBLKMOV, MOVE_BLK, UMOVE_BLK, FILL, FILL_BLK, UFILL_BLK, SWAP

Arithmetic functions

ADD, SUB, MUL, DIV, MOD, INC, DEC, T_ADD, T_SUB, T_DIFF, T_COMBINE

Math functions

SIN, COS, TAN, ASIN, ACOS, ATAN, SQR, SQRT, LN, EXP, EXPT, FRAC, NEG, ABS

Conversion functions for numerical values

CONVERT, ROUND, CEIL, FLOOR, TRUNC

Conversion functions for time values

T_CONV

Conversion functions for strings

S_CONV

Shift functions

SHL, SHR, ROL, ROR

Logic functions

AND, OR, XOR, INV, DECO, ENCO, SEL, MUX, DEMUX, MIN, MAX, LIMIT

String functions

LEN, CONCAT, DELETE, LEFT, RIGHT, MID, FIND, INSERT, REPLACE

Fig. 7.26 Overview of boxes with enable input EN and enable output ENO

The parameters of the EN/ENO boxes must all be connected. The enable input EN and the enable output ENO are not parameters of the box function. They are used for processing boxes and are added to the box function by the program editor.

By default, the majority of EN/ENO boxes are displayed without an ENO output when they are moved from the program elements catalog to the working area. Only when you select the command *Generate ENO* from the shortcut menu when the box is selected will the ENO output be displayed and the required statements will be generated during compilation. You can deselect an ENO output using the command *Do not generate ENO* from the shortcut menu.

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 7.6.4 “EN/ENO mechanism in the ladder logic” on page 320. The block calls in the ladder logic, which are also shown as EN/ENO boxes, are described in Chapter 14.2 “Calling of code blocks” on page 631.

7.5.1 Edge evaluation with an EN/ENO box

The R_TRIG box and the F_TRIG box are available for edge evaluation with an EN/ENO box. A detailed description of the edge evaluation is provided in Chapters 12.3 “Edge evaluation” on page 515 and 12.3.5 “Edge evaluation with an EN/ENO box (LAD, FBD)” on page 520.

For programming, drag one of the edge evaluations with the mouse from the program elements catalog under *Basic instructions* > *Bit logic operations* to the working area. When you release the mouse button, you will be prompted to specify a data area for the instance data. This can be a data block or, if the edge evaluation is programmed in a function block, a local instance (multi-instance) in the instance data block of the function block.

In Fig. 7.27, the *Start* tag is monitored for a rising edge. The instance data is located in the local data of the function block. It consists of the input CLK (in the example: *“Start”* tag), the output Q, and the edge trigger flag. The output Q can also be addressed directly: For a single instance, specify the data block (example: “DB_name”.Q). For a local instance, specify the instance name (example: #Instance_name.Q, in the figure: *#Edge_Start.Q*).

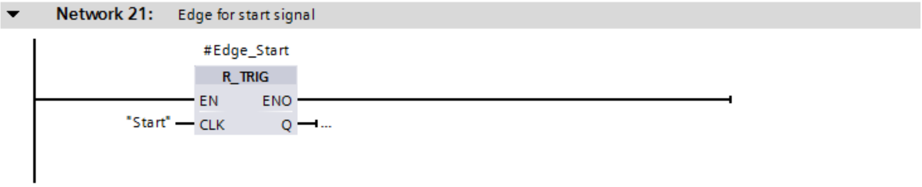


Fig. 7.27 Example of edge evaluation with EN/ENO box in the ladder logic

7.5.2 Transfer functions in the ladder logic

A detailed description of the transfer functions is provided in Chapter 13.2 “Transfer functions” on page 559.

The transfer function MOVE transfers the value of one tag to one or more other tags. MOVE_BLK and UMOVE_BLK transfer individual components from one ARRAY tag to another. BLKMOV and UBLKMOV transfer individual tags or absolutely addressed data areas. FILL_BLK and UFILL_BLK fill components of an ARRAY tag with a value. FILL fills a tag or an absolutely addressed data area with a value. SWAP swaps the order of the bytes in a tag.

For programming, drag the symbol of the transfer function with the mouse from the program elements catalog under *Basic instructions* > *Move operations* to the working area.

In Fig. 7.28, the *Messages* tag is transferred from the data block *“Data.LAD”* to the *“Alarm bits”* tag in the bit memory address area.

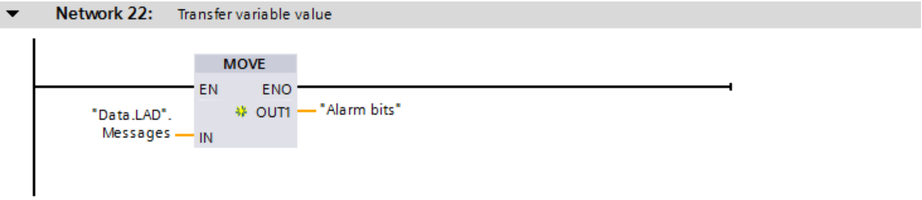


Fig. 7.28 Example of a transfer function in the ladder logic

7.5.3 Arithmetic functions in the ladder logic

A CPU 1500 provides arithmetic functions for numerical values and for time values.

Arithmetic functions for numerical values

An arithmetic function for numerical values implements the basic arithmetical operations with the data formats USINT, UINT, UDINT, ULINT, SINT, INT, DINT, LINT, REAL, and LREAL in the user program. A detailed description of these arithmetic functions is provided in Chapter 13.4 “Arithmetic functions” on page 574.

For programming, drag one of the arithmetic functions (ADD, SUB, MUL, DIV, or MOD) with the mouse from the program elements catalog under *Basic instructions* > *Math functions* to the working area. You can set the function and data types using drop-down lists which you can open using the small yellow triangle when the box is selected. The data type is also automatically set when the first actual value is created.

In Fig. 7.29, the upper limit of a measured value is monitored. A hysteresis is introduced to ensure that the *#Measurement_too_high* alarm does not “pulsate” when the measured value changes slightly in the upper limit range. The alarm *#Measurement_too_high* is only canceled when the measured value has dropped again below the upper limit by the magnitude of the hysteresis.

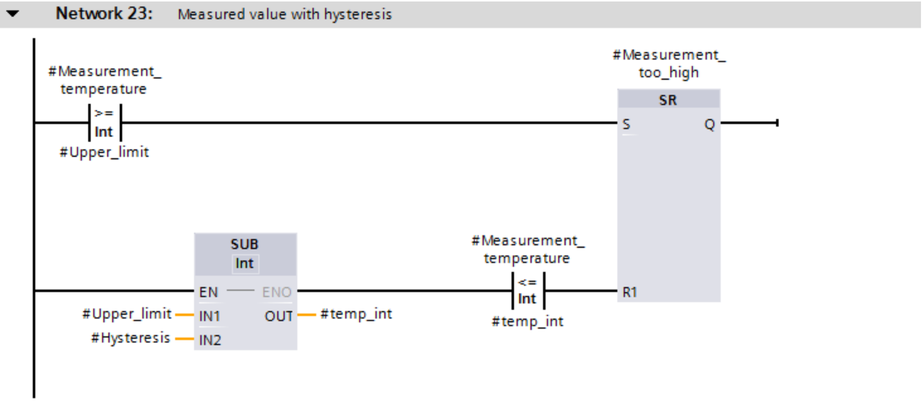


Fig. 7.29 Example of an arithmetic function in the ladder logic

Arithmetic functions for time values

An arithmetic function for time values adds two durations or one duration to a time (T_ADD), subtracts two durations or one duration from a time (T_SUB), formulates the difference of two times (T_DIFF), or combines a date and a time-of-day into a time (T_COMBINE). A detailed description of these arithmetic functions is provided in Chapter 13.4 “Arithmetic functions” on page 574.

For programming, drag one of the functions (T_ADD, T_SUB, T_DIFF, or T_COMBINE) with the mouse from the program elements catalog under *Extended instructions > Date and time-of-day* to the working area. You can set the data types using drop-down lists which you can open using the small yellow triangle when the box is selected.

7.5.4 Math functions in the ladder logic

The math functions comprise, for example, trigonometric functions, exponential functions, and logarithmic functions with tags in the data formats REAL and LREAL. A detailed description of these math functions is provided in Chapter 13.5 “Math functions” on page 578.

For programming, drag one of the math functions (SIN, COS, TAN, ASIN, ACOS, ATAN, SQR, SQRT, LN, EXP, EXPT, FRAC, NEG, ABS) with the mouse from the program elements catalog under *Basic instructions > Math functions* to the working area. You can set the function and data types using drop-down lists which you can open using the small yellow triangle when the box is selected.

Fig. 7.30 shows the calculation of the reactive power according to the equation *#Reactive_power* = *#Voltage* × *#Current* × sin(*#phi*).

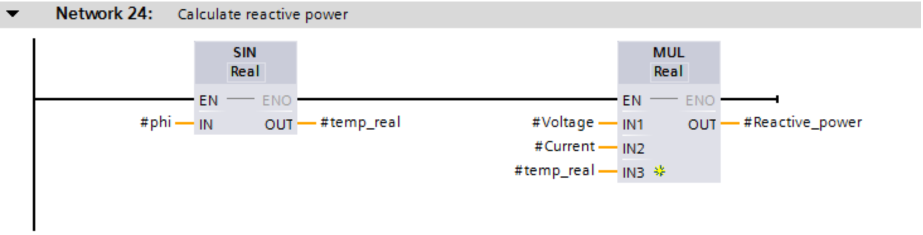


Fig. 7.30 Example of math functions in the ladder logic

7.5.5 Conversion functions in the ladder logic

The conversion functions convert the data formats of tags. A detailed description of the conversion functions is provided in Chapter 13.6 “Conversion functions” on page 586.

For programming, drag one of the conversion functions (CONVERT, ROUND, CEIL, FLOOR, TRUNC, SCALE_X, or NORM_X) with the mouse from the program elements catalog under *Basic instructions* > *Conversion operations* to the working area. You can set the function and data types using drop-down lists which you can open using the small yellow triangle when the box is selected. If the first actual value created has a permissible data type, the data type is also set automatically.

The conversion function T_CONV for data type conversion of date/time can be found in the program elements catalog under *Extended instructions* > *Date and time-of-day*. The conversion function for data type conversions of character strings (S_CONV, STRG_VAL, VAL_STRG, CHARS_TO_STRG, STRG_TO_CHARS, ATH, HTA) can be found in the program elements catalog under *Extended instructions* > *String + Char*.

Fig. 7.31 shows an example of the conversion functions. A measured value present in data format REAL is first converted into data format DINT and then converted into the BCD32 format.

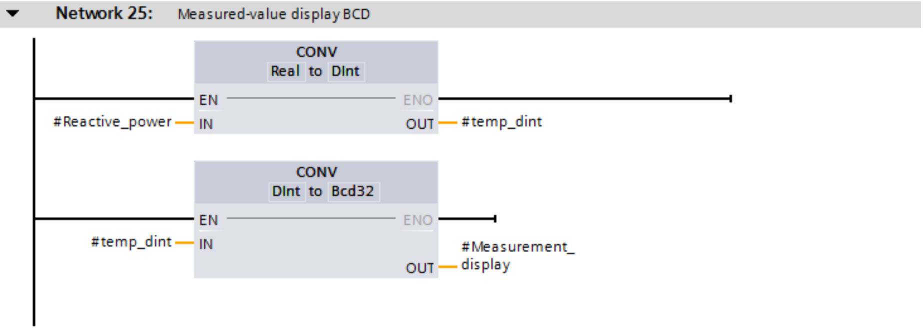


Fig. 7.31 Example of the conversion functions in the ladder logic

7.5.6 Shift functions in the ladder logic

The shift functions shift the content of tags bit-by-bit to the left or right. A detailed description of the shift functions is provided in Chapter 13.7 “Shift functions” on page 603.

For programming, drag one of the shift functions (SHL, SHR, ROL, or ROR) with the mouse from the program elements catalog under *Basic instructions* > *Shift and rotate* to the working area. You can set the function and data types using drop-down lists which you can open using the small yellow triangle when the box is selected. The data type is also automatically set when the first actual value is created.

In Fig. 7.32, the respective three decades of two numbers present in BCD16 format of a SIMATIC counter are joined without gaps. Using the shift function SHL – set to data type DWORD! – the *#Quantity_high* tag is shifted to the left by 12 bits, corresponding to three decades. A small square on the input parameter IN indicates that the data type of the applied tag (WORD in the example) does not agree with the data type of the function (DWORD in the example) and will be converted implicitly.

The bottom three decades (the *#Quantity_low* tag) are subsequently added by a doubleword logic operation according to OR and output to the *#Quantity_display* tag.

7.5.7 Logic functions in the ladder logic

The logic functions include the word logic operations AND, OR and XOR, the inversion INVERT, the coding functions DECO and ENCO, the selection functions SEL, MUX, DEMUX, MIN, MAX, and the limiting function LIMIT. A detailed description of the logic functions is provided in Chapter 13.8 “Logic functions” on page 607. In the program elements catalog, the logic functions are located under *Basic instructions* > *Word logic operations* (AND, OR, XOR, INVERT, DECO, ENCO, SEL, MUX, and DEMUX) and under *Basic instructions* > *Math functions* (MIN, MAX, and LIMIT).

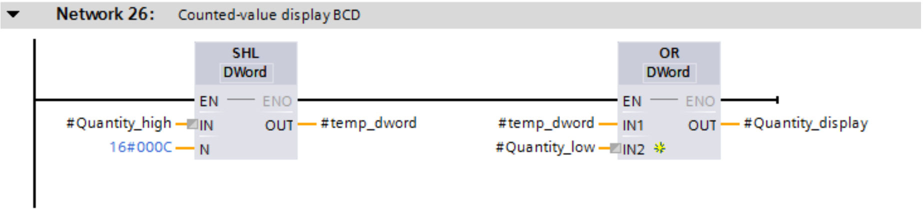


Fig. 7.32 Example of the shift functions in the ladder logic

Word logic operations

The word logic operations link each bit of two tags according to an AND, OR, or exclusive OR function. For programming, drag one of the word logic operations (AND, OR, XOR, INV) with the mouse from the program elements catalog under *Basic instructions* > *Word logic operations* to the working area. You can set the func-

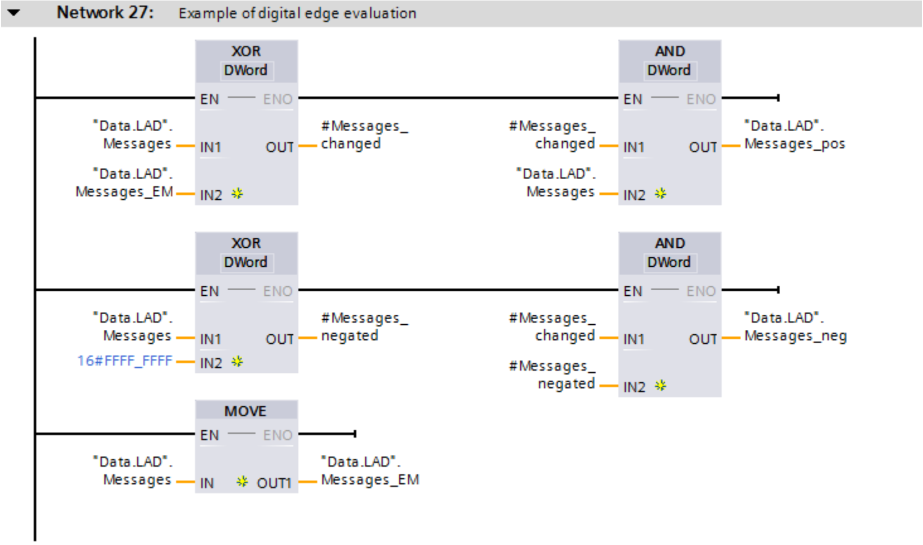


Fig. 7.33 Example of word logic operations in the ladder logic

tion (AND, OR and XOR with AND, OR and XOR, INV is fixed) and the data type (WORD and DWORD with AND, OR and XOR, INT and DINT with INV) via drop-down lists which you can open using the small yellow triangle when the box is selected. The data type is also automatically set when the first actual value is created.

Fig. 7.33 shows how you can program 32 edge evaluations simultaneously for rising and falling edges. The alarm bits are collected in a doubleword *Messages*, which is present in data block *"Data.LAD"*. 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 alarm bit has changed. In order to obtain the positive signal edges, the changes are linked to the alarms by an AND logic operation. The bit is set for a rising signal edge wherever the alarm and the change each have a "1". This corresponds to the pulse flag of the edge evaluation. If you do the same with the negated alarm bits – the alarm 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 alarms.

7.5.8 Functions for strings in the ladder logic

Character strings are tags with the data type STRING. With the functions for character strings, parts of a character string can be extracted (LEFT, RIGHT, MID), inserted (INSERT), replaced (REPLACE) or deleted (DELETE), two character strings can be combined (CONCAT), and the length of a character string (LEN) or the position of a character in a character string (FIND) can be determined.

A detailed description of these functions is provided in Chapter 13.9 "Processing of strings (data type STRING)" on page 615.

Fig. 7.34 shows the connection of the STRING tags *#Station.Name* and *#Station.Number* to form the tag *#Station.Identification*. In the program elements catalog, the string functions are located under *Extended instructions* > *String + Char*.

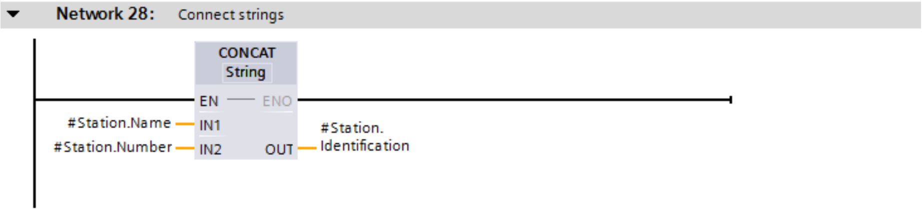


Fig. 7.34 Example of string functions in the ladder logic

7.6 Program control with LAD

You can influence execution of the user program by means of the program control functions. You use jump functions to exit linear program execution and continue at a different point in the block. Block call functions cause the continuation of program execution in a different block. A block end function terminates the execution in the block. The functions that are available in the ladder logic are shown in Fig. 7.35.

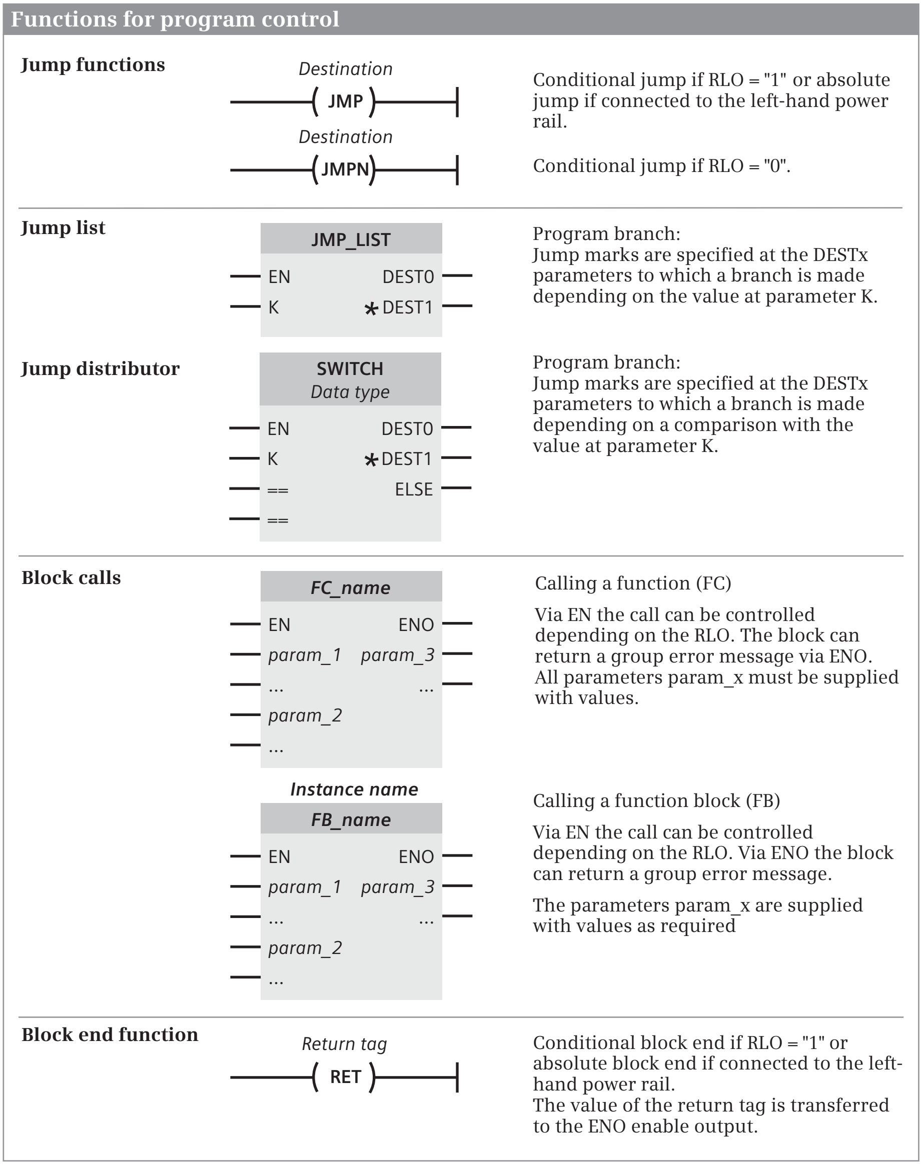
7.6.1 Jump functions in the ladder logic

A detailed description of the jump functions is provided in Chapter 14.1 "Jump functions" on page 623.

Jump functions JMP and JMPN

To program a jump function JMP or JMPN, drag a jump coil with the mouse from the program elements catalog under *Basic instructions* > *Program control operations* to the working area. You define the jump label (the jump destination) using the jump coil. To program the jump destination, use the mouse to drag the *Label* function to the start of the network with which processing of the program is to be continued from the program elements catalog under *Basic instructions* > *Program control operations* and write the designation of the label into the box.

You can subsequently set the jump function (JMP or JMPN) via a drop-down list which you can open using the small yellow triangle when the coil is selected. You can also directly connect the coil with the jump function JMP to the left-hand power rail. The jump is always carried out in this case (absolute jump). The jump function JMPN always requires a preceding logic operation.



Functions for program control	
Jump functions	Conditional jump if RLO = "1" or absolute jump if connected to the left-hand power rail. Conditional jump if RLO = "0".
Jump list	Program branch: Jump marks are specified at the DESTx parameters to which a branch is made depending on the value at parameter K.
Jump distributor	Program branch: Jump marks are specified at the DESTx parameters to which a branch is made depending on a comparison with the value at parameter K.
Block calls	Calling a function (FC) Via EN the call can be controlled depending on the RLO. The block can return a group error message via ENO. All parameters param_x must be supplied with values. Calling a function block (FB) Via EN the call can be controlled depending on the RLO. Via ENO the block can return a group error message. The parameters param_x are supplied with values as required
Block end function	Conditional block end if RLO = "1" or absolute block end if connected to the left-hand power rail. The value of the return tag is transferred to the ENO enable output.

Fig. 7.35 Overview of functions for program control in the ladder logic

It is only possible to jump within a block. A jump function cannot be programmed in association with a T branch. Only one jump function or block end function is permissible per network.

Example of loop jump

Fig. 7.36 shows a jump function using a program loop as an example. In a *#Quantity* array with 16 components from *#Quantity[0]* to *#Quantity[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 *#Quantity[#Index]*. If *#MaxValue* is less than *#Quantity[#Index]*, it is overwritten with the larger value of *#Quantity[#Index]*. *#Index* is then increased by +1. As long as *#Index* is less than or equal to 15, it is jumped to the beginning of the program loop (to the jump destination *MaxSearch*) and the program section is executed again.

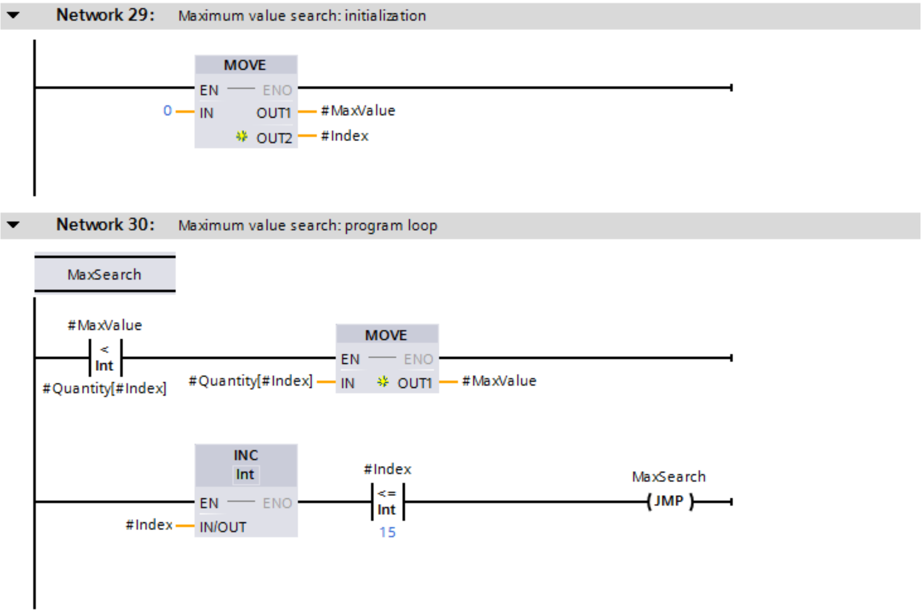


Fig. 7.36 Example of a conditional jump

Jump list JMP_LIST

The jump list is represented as a box. The box is only processed if the EN input signal state is “1”. The value of parameter K (0 to 99) determined 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.

Jump distributor SWITCH

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 comparison is fulfilled, program execution 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.

7.6.2 Block call functions in the ladder logic

Calling of a block is represented by an EN/ENO box. With a function (FC), the block name is present quasi as a function name in the box; with a function block, the instance name (the name of the instance data block or the name of the local instance) is additionally present above the box. A detailed description of the block calls is provided in Chapter 14.2 "Calling of code blocks" on page 631.

To call a code block, use the mouse to drag the block which has already been programmed from the project tree under *Program blocks* into the working area. With a logic operation preceding the EN input you can structure the block call depending on conditions.

The top network in Fig. 7.37 shows the call of a function (FC). The function name is

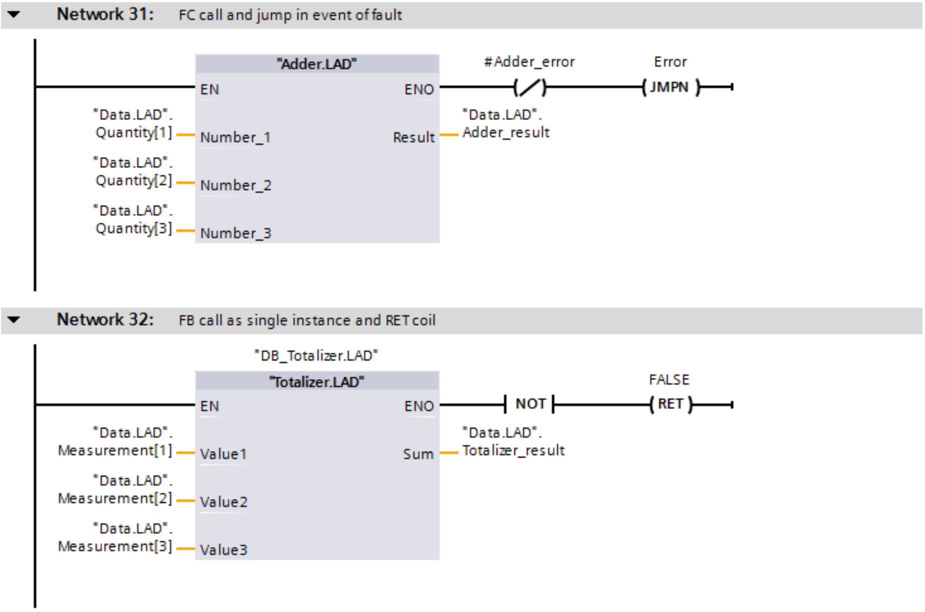


Fig. 7.37 Examples of functions for program control in the ladder logic

present as title in the call box. In the event of an error in the block (ENO is then "0"), *#Adder_error* is set to "1" and a jump is made to the network with the *Error* label.

In the next network, the call of a function block is present as a single instance. The name of the function block is present as the title in the call box, the instance name – in this case the name of the instance data block – is present above the box. If the block reports an error with ENO = “0”, the block is exited with the RET coil and the value FALSE.

7.6.3 Block end function in the ladder logic

To program the block end function, drag the RET coil with the mouse from the program elements catalog under *Basic instructions* > *Program control operations* to the working area. Now define which value the ENO output of the exited block should have. You select the RET coil and use the small yellow triangle to select the dropdown menu item

▷ *Ret*, then it is the current result of logic operation RLO (the “current flow”, i.e. signal state “1”),

▷ *Ret True*, then it is the signal state “1” (TRUE),

▷ *Ret False*, then it is the signal state “0” (FALSE),

▷ *Ret Value*, then it is the signal state that the return tag above the RET coil has.

A detailed description of the RET coil is provided in Chapter 14.3.1 “Block end function RET (LAD and FBD)” on page 636.

In the second network in Fig. 7.37, the block with the RET coil is left if the “*Totalizer.LAD”* block signals an error. The ENO output of the exited block is set then to signal state “0” (FALSE).

7.6.4 EN/ENO mechanism in the ladder logic

The EN/ENO mechanism allows the execution of program functions (statements) and blocks that are dependent upon the logic operation (“current flow”). The enable input EN enables the execution of a program function or a block. The enable output (ENO) reports an error in program execution that occurred during runtime. The enable input EN and the enable output ENO are both of data type BOOL.

EN and ENO are not function or block parameters; they are not declared in the block interface. They are statement sequences which the program editor generates before and after a function or block call.

EN/ENO mechanism with program functions (instructions)

The program editor in LAD displays program functions with the EN/ENO mechanism using EN/ENO boxes. These are the functions described in Chapter 7.5 “Programming EN/ENO boxes with LAD” on page 308, which can be found in the program elements catalog in the *Basic instructions* tab.

If the enable input EN is connected to the left-hand power rail or to a preceding logic operation which provides signal state “1” during runtime (if “current” is flowing), then the function is carried out. If the signal state is “0” at the EN input, the function is not carried out and program execution is continued with the next program element.

The enable output ENO provides signal state “1” if the function has been executed without any errors. If an error occurred during execution of the function or if signal state “0” was present at the EN input, resulting in the function not being executed, then the ENO output has signal state “0”.

The enable output ENO is not displayed by default in most functions, when you drag them from the program elements catalog to the working area. Then no additional program code is generated for error detection. Using the *Generate ENO* command from the shortcut menu, you can switch on error detection for the selected function, then the ENO output will be displayed and an additional program code will be generated during compilation. The ENO output is “deselected” using the command *Do not generate ENO*.

Controlling a processing sequence

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. In this manner it is possible, for example, to “switch off” the complete sequence, or the rest of the sequence is no longer processed if a box signals an error.

In the example in Fig. 7.38, neither of the boxes is processed if the *#Enabling* tag has signal state “0”. If an error occurs during processing of the ADD box, for example a numerical range is exceeded, the subsequent SQRT box is no longer processed.

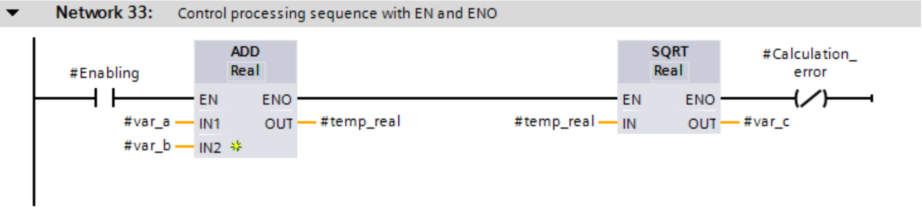

Network 33: Control processing sequence with EN and ENO

Fig. 7.38 Example of series connection of ENO and EN with LAD

EN/ENO mechanism with blocks

When calling blocks (FC functions and FB function blocks), the program editor always displays the input EN and the output ENO, regardless of which programming language the blocks are programmed in. In this context, blocks also include all of the functions in the program elements catalog that are not listed in the *Basic instructions* tab (called “system blocks” in the following).

If the enable input EN is connected to the left-hand power rail or to a preceding logic operation which provides signal state “1” during runtime (if “current” is flowing), then the block is called. If the signal state is “0” at the EN input, a block call is not carried out and the program execution is continued after the block call.

For “system blocks”, the enable output ENO provides signal state “1” if the function has been executed without any errors. If an error occurred during execution of the “system block” or if signal state “0” was present at the EN input, resulting in the “system block” not being executed, then the ENO output has signal state “0”.

For self-written blocks (FC functions and FB function blocks), it is the responsibility of the user to determine which signal state the enable output ENO provides. By default – without user action – the ENO output signal state is “1”. In the event of an error, if you would like to evaluate an error in the calling block, you must set the ENO output to signal state “0”.

Controlling the ENO output for self-written blocks

The signal state of the ENO output is controlled in the ladder logic with the RET coil (see Chapter 7.6.2 “Block call functions in the ladder logic” on page 319). In principle, you can end the execution in the block in the event of any detected error by using the RET coil and the return value FALSE and then no longer execute the remainder of the block program.

You can also keep a detected error in “error tags” and then, at the end of the block, terminate the block with FALSE in the event of an error.

An example is shown in Fig. 7.39. In the event of an error, the tag *#Measurement_ok* has signal state “0” and the tags *#Adder_error* and *#Calculation_error* have signal state “1”. In the event of an error, the tag *#temp_bool* is set to signal state “0”. Upon exiting the block with the RET coil, the signal state of the tag *#temp_bool* is assigned to the ENO output.

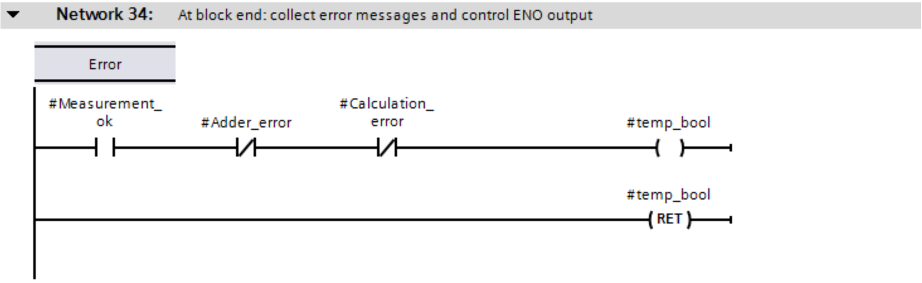


Fig. 7.39 Control of the ENO output

8 Function block diagram FBD

8.1 Introduction

This chapter describes programming with function block diagram. It provides examples of how the program functions are represented in the function block diagram. You can find a description of the individual functions, e.g. comparison functions, in Chapters 12 “Basic functions” on page 503, 13 “Digital functions” on page 558, and 14 “Program control” on page 622.

Use of the program and symbol editor, which generally applies to all programming languages, is described in Chapter 6 “Program editor” on page 247.

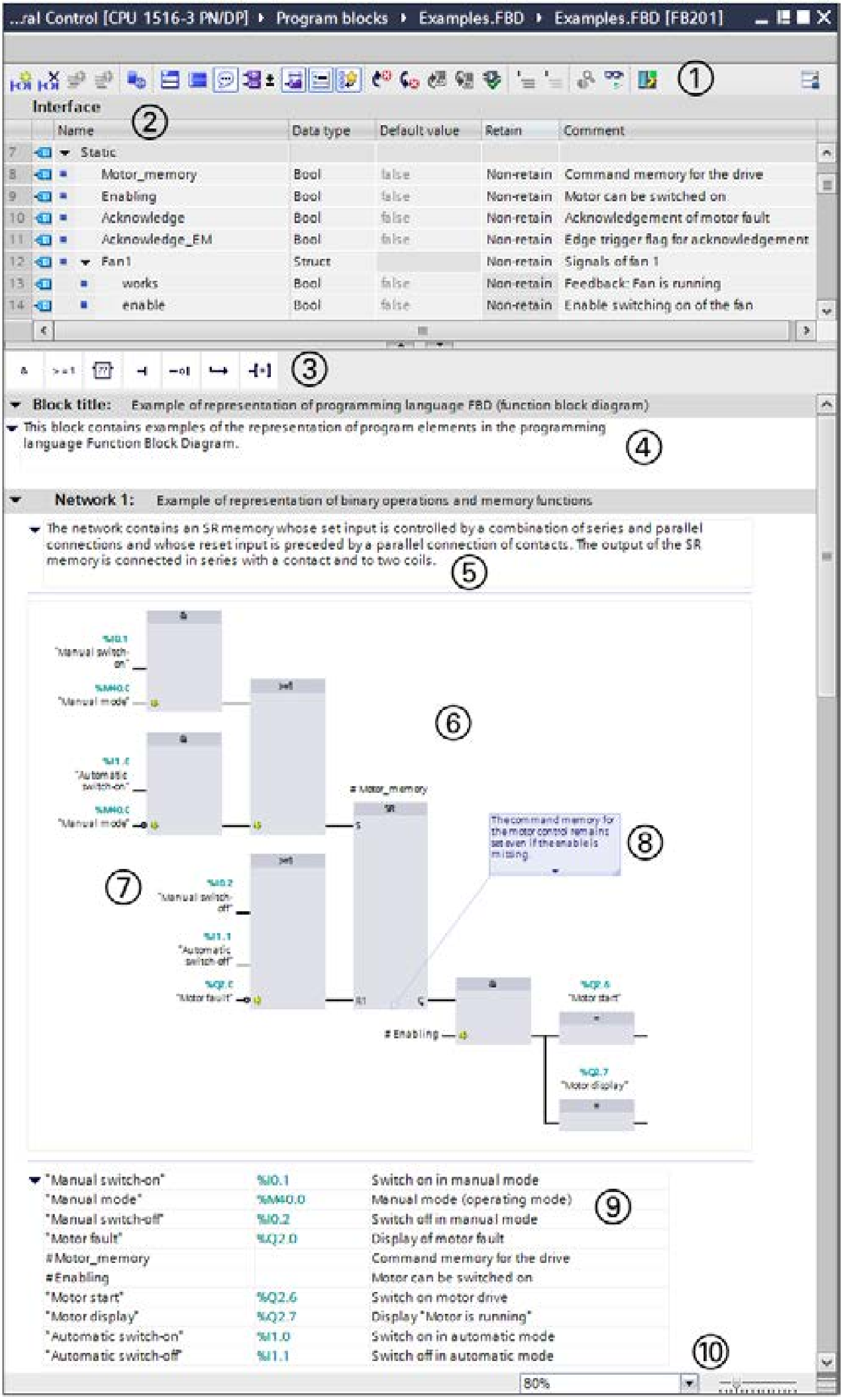
Function block diagram is used to program the contents of blocks (the user program). What blocks are, and how they are created, is described in Chapters 5.3.1 “Block types” on page 155 and 6.3 “Programming a code block” on page 253.

8.1.1 Programming with FBD in general

You use FBD to program the control function of the programmable controller – the user program (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 program editor's working window. The icons in the toolbar ① can be used to set the display of the working area, e.g. the display of the network comments and additional functions such as monitoring of the program status. The interface of the block ② in the upper part of the working window lists the block parameters and local data. The favorites bar ③ can be expanded by additional program elements. It can also be hidden. Each block has a heading, the block title, and a block comment ④, which can be used to explain the function of the block. These are followed by the first network with its number, heading and comment ⑤.

The control function, i.e. the interconnection of the program elements, is displayed in the working area ⑥. The tags can be displayed absolutely, symbolically, or with both addressing types ⑦. Each box can be assigned a “free-form” comment ⑧. The tag information ⑨ shows the tags used in the network with the tag comments. Like the network comment and the free comments, it can be hidden. With the zoom setting ⑩ the display of the control function can be adapted to the size of the working area.



...ral Control [CPU 1516-3 PN/DP] ▸ Program blocks ▸ Examples.FBD ▸ Examples.FBD [FB201]

Interface

	Name	Data type	Default value	Retain	Comment
7	▼ Static				
8	Motor_memory	Bool	false	Non-retain	Command memory for the drive
9	Enabling	Bool	false	Non-retain	Motor can be switched on
10	Acknowledge	Bool	false	Non-retain	Acknowledgement of motor fault
11	Acknowledge_EM	Bool	false	Non-retain	Edge trigger flag for acknowledgement
12	▼ Fan1	Struct		Non-retain	Signals of fan 1
13	works	Bool	false	Non-retain	Feedback: Fan is running
14	enable	Bool	false	Non-retain	Enable switching on of the fan

Block title: Example of representation of programming language FBD (function block diagram)

This block contains examples of the representation of program elements in the programming language Function Block Diagram.

Network 1: Example of representation of binary operations and memory functions

The network contains an SR memory whose set input is controlled by a combination of series and parallel connections and whose reset input is preceded by a parallel connection of contacts. The output of the SR memory is connected in series with a contact and to two coils.

The command memory for the motor control remains set even if the enable is missing.

"Manual switch-on"	%I0.1	Switch on in manual mode
"Manual mode"	%M40.0	Manual mode (operating mode)
"Manual switch-off"	%I0.2	Switch off in manual mode
"Motor fault"	%Q2.0	Display of motor fault
#Motor_memory		Command memory for the drive
#Enabling		Motor can be switched on
"Motor start"	%Q2.6	Switch on motor drive
"Motor display"	%Q2.7	Display "Motor is running"
"Automatic switch-on"	%I1.0	Switch on in automatic mode
"Automatic switch-off"	%I1.1	Switch off in automatic mode

Fig. 8.1 Working window of the program editor for FBD programming

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 the inputs and outputs. The boxes with binary logic operations can be extended by additional 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 T branches, there are certain limitations with regard to which permissible program elements can be selected.

A block is not terminated by a special network or function, you simply finish the program input. If 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 such as arithmetic functions.

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.

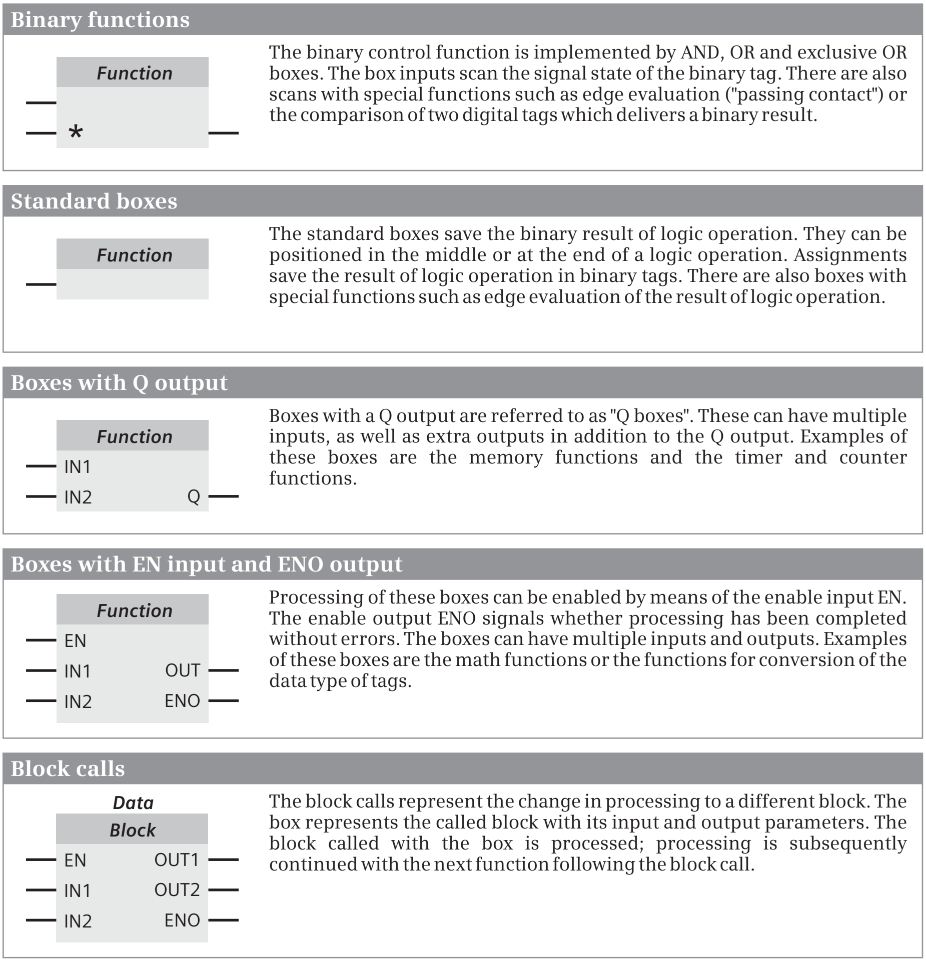
8.2 Programming binary logic operations with FBD

The binary logic operations are carried out in the function block diagram using the AND, OR, and exclusive OR boxes. The binary tags, e.g. the inputs, can be scanned for signal state “1” or signal state “0” before the result of the scan is linked further. The binary results of other boxes can also be included, e.g. the evaluation of a signal edge or the comparison of two digital tags (Fig. 8.3).

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.

The example in Fig. 8.4 shows the two “Start” and “Stop” pushbuttons. When pressed, they output the signal state “1” in the case of an input module with sinking input. The SR function is set or reset with this signal state.



Binary functions

The binary control function is implemented by AND, OR and exclusive OR boxes. The box inputs scan the signal state of the binary tag. There are also scans with special functions such as edge evaluation ("passing contact") or the comparison of two digital tags which delivers a binary result.

Standard boxes

The standard boxes save the binary result of logic operation. They can be positioned in the middle or at the end of a logic operation. Assignments save the result of logic operation in binary tags. There are also boxes with special functions such as edge evaluation of the result of logic operation.

Boxes with Q output

Boxes with a Q output are referred to as "Q boxes". These can have multiple inputs, as well as extra outputs in addition to the Q output. Examples of these boxes are the memory functions and the timer and counter functions.

Boxes with EN input and ENO output

Processing of these boxes can be enabled by means of the enable input EN. The enable output ENO signals whether processing has been completed without errors. The boxes can have multiple inputs and outputs. Examples of these boxes are the math functions or the functions for conversion of the data type of tags.

Block calls

The block calls represent the change in processing to a different block. The box represents the called block with its input and output parameters. The block called with the box is processed; processing is subsequently continued with the next function following the block call.

Fig. 8.2 Overview of program elements of the function block diagram

The “/Fault” signal is not active in the normal case. Signal state “1” is then present and is negated by scanning for signal state “0”, and the SR function therefore remains uninfluenced. If “/Fault” becomes active, the SR function is to be reset. The active signal “/Fault” delivers signal state “0”, which by scanning for signal state “0” resets the SR function as signal state “1”.

8.2.2 Programming a binary logic operation in the function block diagram

To program a binary logic operation, drag the corresponding symbol (&, >=1, X) with the mouse from the program elements catalog under *Basic instructions* > *Bit logic operations* to the working area. If a logic operation is already present in the working area, the program editor indicates with small gray boxes where the logic

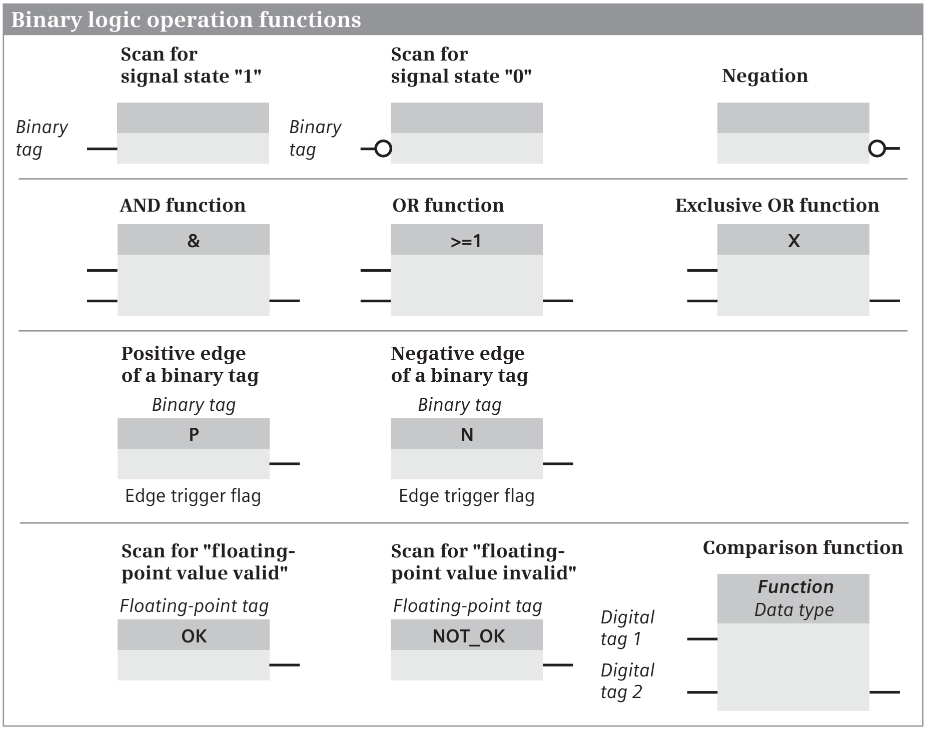
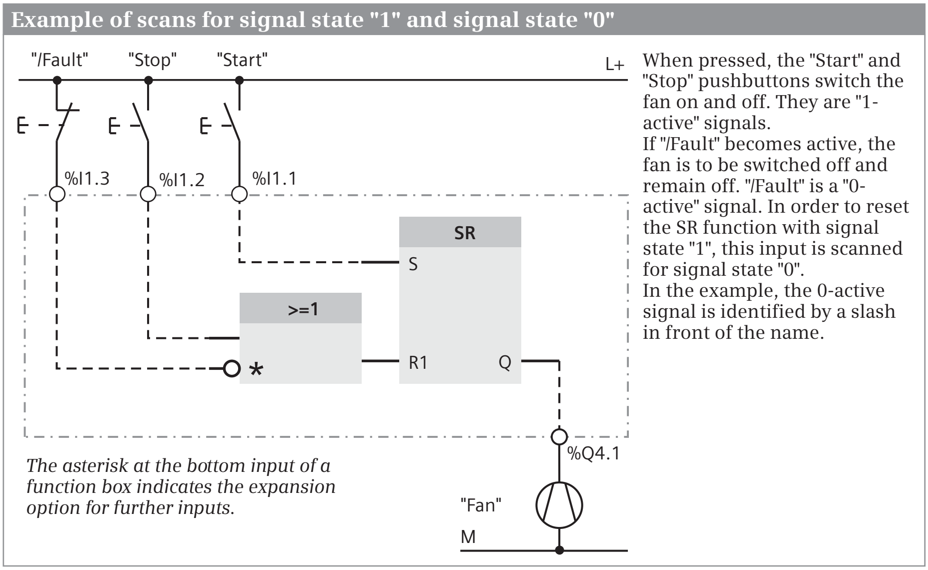


Fig. 8.3 Overview of binary logic operations in the function block diagram



When pressed, the "Start" and "Stop" pushbuttons switch the fan on and off. They are "1-active" signals.
If "/Fault" becomes active, the fan is to be switched off and remain off. "/Fault" is a "0-active" signal. In order to reset the SR function with signal state "1", this input is scanned for signal state "0".
In the example, the 0-active signal is identified by a slash in front of the name.

The asterisk at the bottom input of a function box indicates the expansion option for further inputs.

Fig. 8.4 Scanning for signal states “1” and “0”

operation may be positioned and with a green box where it is positioned when you “let go”.

A binary logic function has two inputs as standard. If you select the function box when programming and then select the *Insert input* command in the shortcut menu with the right mouse button, or more simply: click on the asterisk with the left mouse button, then the program editor expands the function block by a further input.

To program a scan for signal state “0”, drag the negation symbol (*invert RLO*) with the mouse from the program elements catalog under *Basic instructions* > *General* to a box input. In the same manner you can convert a scan for signal state “0” into a scan for signal state “1” or negate the result of logic operation between the boxes.

You connect a binary tag to the input of a binary logic operation. This can be an input, output, bit memory or data bit, a SIMATIC timer/counter function, or the binary output of another function box. Assignment with a constant (TRUE or FALSE) is not permissible.

You can connect further binary function boxes to the output of a binary logic operation. To assign the result of logic operation of a function box to a binary tag, position an assign box at the output which you fetch in the program elements catalog under *Basic instructions* > *Bit logic operations*.

8.2.3 AND function in the function block diagram

An AND function is fulfilled if all inputs have the scan result “1”. A description of the AND function is provided in Chapter 12.1.3 “AND function, series connection” on page 507.

Fig. 8.5 shows an example of AND functions. The first AND function scans the *#Fan1.works* tag for signal state “1” and the *#Fan2.works* tag for signal state “0”. The two results of the scans are linked according to an AND logic operation. The AND function is fulfilled (delivers signal state “1”) if only fan 1 is running. The second AND function is fulfilled if only fan 2 is running.

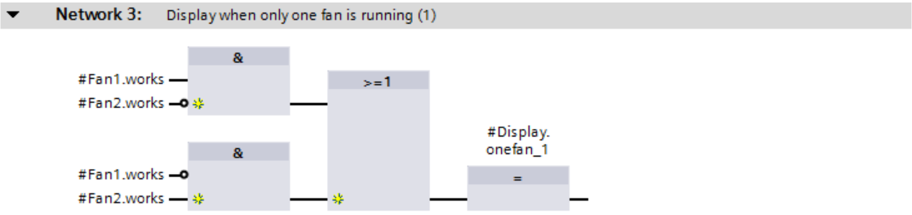

Network 3: Display when only one fan is running (1)

Fig. 8.5 Example of AND-before-OR logic operation

8.2.4 OR function in the function block diagram

An OR function is fulfilled if one or more inputs have the scan result "1". A description of the OR function is provided in Chapter 12.1.4 "OR function, parallel connection" on page 507.

Fig. 8.6 shows an example of OR functions. The first OR function scans the *#Fan1.works* and *#Fan2.works* tags for signal state "1". The two results of the scans are linked according to an OR logic operation. The OR function is fulfilled (delivers signal state "1") if one of the fans is running or if both fans are running. The second OR function is fulfilled if neither of the fans is running.

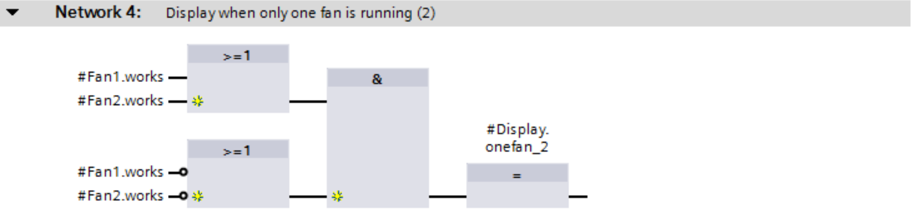

Network 4: Display when only one fan is running (2)

Fig. 8.6 Example of OR-before-AND logic operation

8.2.5 Exclusive OR function in the function block diagram

An exclusive OR function (antivalence function) is fulfilled if an odd number of inputs has the scan result "1". A description of the exclusive OR function is provided in Chapter 12.1.5 "Exclusive OR function, non-equivalence function" on page 508.

Fig. 8.7 shows an example of an exclusive OR function. The *#Fan1.works* and *#Fan2.works* tags are scanned at the inputs of the function box for signal state "1". The exclusive OR function is fulfilled (delivers signal state "1") if only one of the fans is running.

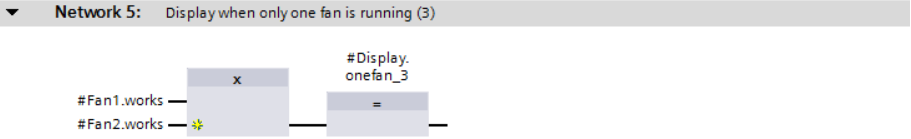

Network 5: Display when only one fan is running (3)

Fig. 8.7 Example of an exclusive OR function

8.2.6 Combined binary logic operations, negating result of logic operation

The function boxes of the AND, OR, and exclusive OR functions can be freely combined with one another. Examples are shown in figures 8.5 and 8.6. Together with Fig. 8.7, the examples – even if the logic operation is different in each case – show the same response: The logic operation is fulfilled if only one of the fans is running.

Negating result of logic operation

The output of a function box can be negated, i.e. the result is signal state “1” if the logic operation is not fulfilled. It is then possible in a simple manner to generate

- a NAND function (negated AND function which is fulfilled if at least one input has the result of scan “0”),
- a NOR function (negated OR function which is fulfilled if all inputs have the result of scan “0”), and
- an inclusive OR function (equivalence function, negated exclusive OR function which is fulfilled if an even number of inputs has the result of scan “1”).

Fig. 8.8 shows a NOR function. The OR function is not fulfilled if none of the fans is running, and then delivers the signal state “0”. This is negated and assigned to the *#Display.nofan* tag.

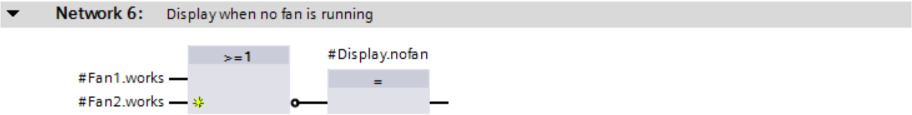


Fig. 8.8 Example of a negated function output

8.2.7 T branch in the function block diagram

You can “divide” a logic operation so that it has two different terminations, the result being a “T branch”. To program a T branch, use the mouse to drag the *Branch* symbol from the program elements catalog under *Basic instructions* > *General* to the position at which the T branch is to commence.

Fig. 8.9 shows a T branch following the lower OR logic operation. The result of logic operation at this position is therefore only “0” if no fan is running. This result of logic operation is negated, linked according to an AND logic operation to “*Clock_2Hz*” and controls the *#Display.nofan* tag.

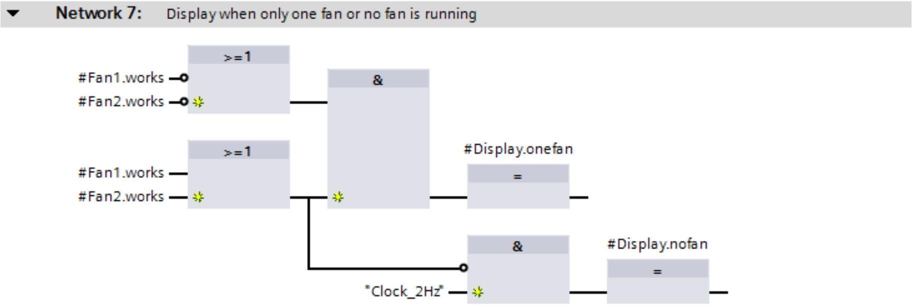


Fig. 8.9 Example of a T branch in the function block diagram

8.2.8 Edge evaluation of binary tags in the function block diagram

An edge evaluation detects the change in a binary signal.

For programming an edge evaluation, drag the symbol for the P or N box with the mouse from the program elements catalog under *Basic instructions > Bit logic operations* to the working area.

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). This "pulse" can be linked further.

The edge trigger flag is named underneath the edge box. This is a memory 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 12.3 "Edge evaluation" on page 515).

Fig. 8.10 shows an application of edge evaluation. Let us assume that an alarm "arrives", i.e. the *#Alarm_bit* signal changes from signal state "0" to signal state "1". The *#Alarm_memory* tag is then set and the *#Alarm_lamp* tag flashes at 0.5 Hz. The alarm memory can be reset using an *#Acknowledge* button. The alarm memory remains reset if *#Acknowledge* has signal state "0" again and *#Alarm_bit* is still present. *#Alarm_memory* is only set again by a further positive edge of *#Alarm_bit* (if *#Acknowledge* then no longer has signal state "1").

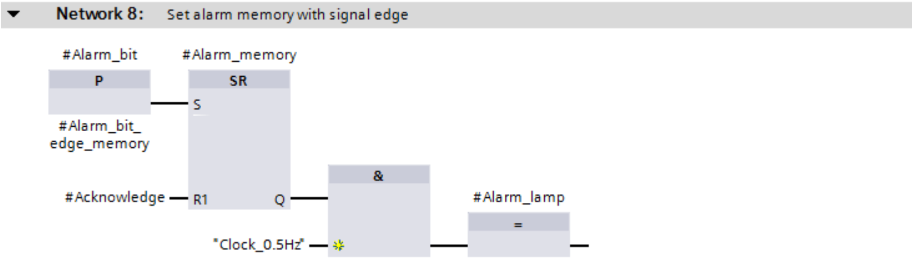


Fig. 8.10 Example of an edge evaluation of a binary tag

8.2.9 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 the data type REAL or LREAL are adhered to. The OK box delivers signal state "1" if the floating-point tag is valid. 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 located at the beginning of a logic operation. You can find the OK/NOT_OK box in the program elements catalog under *Basic instructions > Comparator operations*.

If in Fig. 8.11 the *#Measurement_from_sensor* tag is within the permissible value range for the data type REAL or LREAL and the *#Measurement.Registered* tag has signal state "1", then *#Measured_value_ok* is set.

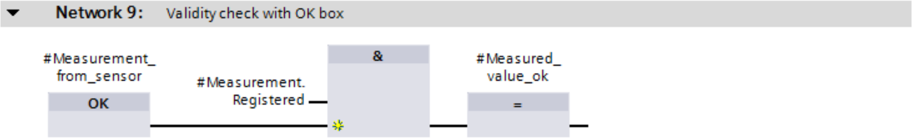


Fig. 8.11 Example of the validity check of floating-point numbers

8.2.10 Comparison functions in the function block diagram

A comparison function compares two digital values and delivers a binary signal as the comparison result. The comparison result has signal state “1” if the comparison is fulfilled, otherwise “0”. The comparison function is described in Chapter 13.3 “Comparison functions” on page 570.

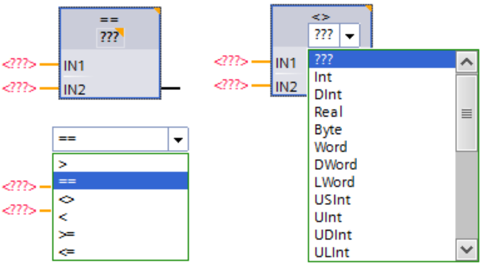


Fig. 8.12 Drop-down lists for setting the comparison mode and data type

To program a comparison function, drag it with the mouse from the program elements catalog under *Basic instructions > Comparator operations* to the working area. You can then use drop-down lists to define the comparison (Fig. 8.12): If you select the comparison function, you can set the comparison type on the triangle in the upper right corner and the data type on the triangle under the comparison type.

Fig. 8.13 displays two comparison boxes whose comparison result is linked according to an AND logic operation. If the *#Measurement_temperature* tag is above a lower limit and below an upper limit, the *#Measured_value_in_range* tag has signal state “1”.

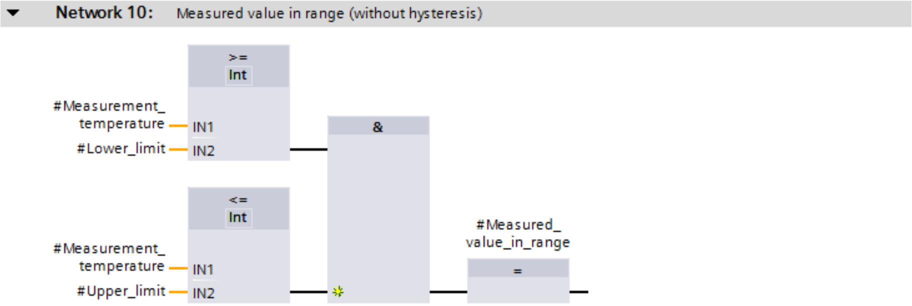


Fig. 8.13 Example of comparison functions

8.3 Programming standard boxes with FBD

Standard boxes control binary tags such as outputs or bit memories. An assign 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 negating assign box. There

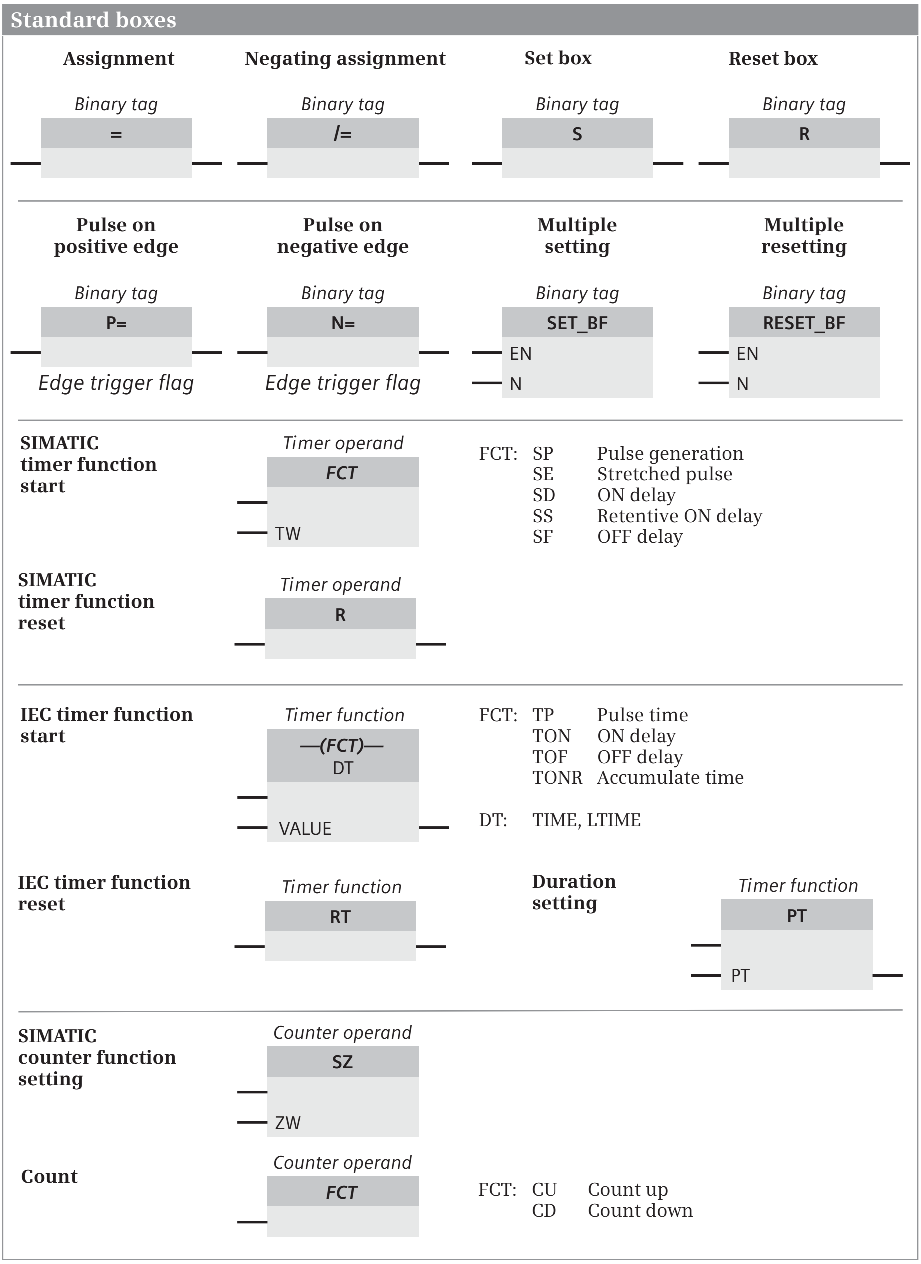


Fig. 8.14 Overview of the standard boxes described in this chapter

are standard boxes for setting and resetting a binary tag, for pulse generation when evaluating signal edges, for controlling a SIMATIC timer/counter function, or for setting and resetting bit arrays (Fig. 8.14).

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.

8.3.1 Assignment and negating assignment

The result of logic operation is directly assigned to the tag above the assign box: With result of logic operation “1”, the tag is set. With result of logic operation “0”, it is reset. With the negating assignment, the tag above the box is set if the result of logic operation is “0”; it is reset if the result of logic operation is “1”.

For programming, drag the symbol for the assignment with the mouse from the program elements catalog under *Basic instructions* > *Bit logic operations* to the working area. Gray boxes indicate the permissible positioning, a green box identifies the position at which the box will be inserted if you release the mouse button.

The assignment and the negating assignment can be used within a logic operation, following a T branch, or as the termination of a logic operation. It can be positioned in series or parallel. The assign box requires a preceding logic operation.

Fig. 8.15 shows the possible arrangements for the assign box. In the logic operation, the assign box is used to control the *#Display.nofan*, *#Display.onefan*, and *#Display.twofans* tags. Two boxes are connected in parallel at the end of the logic operation. They respond identically.

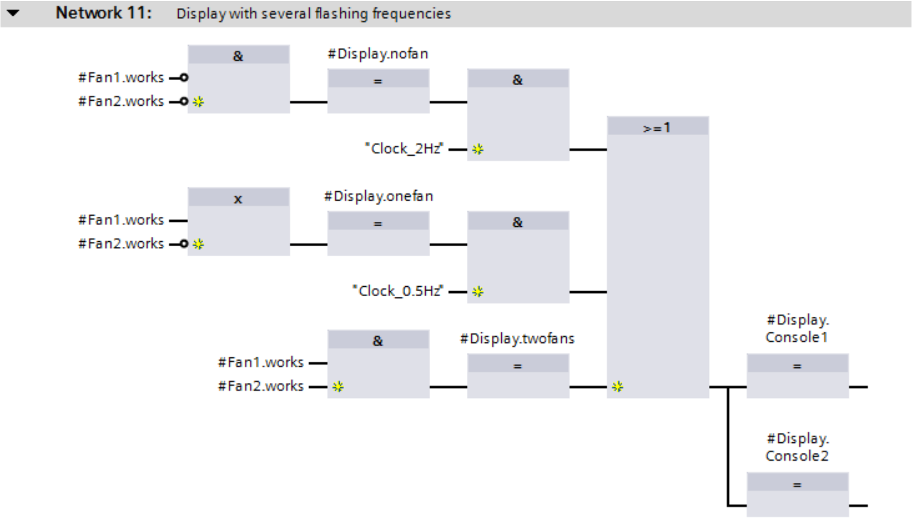


Fig. 8.15 Example of arrangement of the assign box

8.3.2 Set and reset boxes

A set or reset box is used to assign signal state “1” or “0” to a binary tag in the case of a result of logic operation “1”. A result of logic operation “0” has no effect.

For programming, drag the symbol for the set or reset box with the mouse from the program elements catalog under *Basic instructions* > *Bit logic operations* to the working area. Gray boxes indicate the permissible positioning, a green box identifies the position at which the coil will be inserted if you release the mouse button.

Set and reset boxes require a preceding logic operation and terminate a logic operation. The reset box can also be used to reset a SIMATIC timer/counter function.

In Fig. 8.16, *#Fan1.start* with signal state “1” sets the *#Fan1.drive* tag. Signal state “0” at *#Fan1.start* has no effect. With signal state “1” at *#Fan1.stop*, *#Fan1.drive* is reset. Signal state “0” at *#Fan1.stop* has no effect. Because the reset box is arranged after the set box, the memory response is “reset dominant”: If both tags have signal state “1”, *#Fan1.drive* is reset or remains reset.

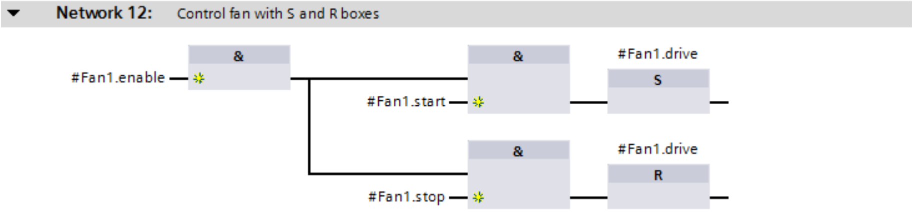


Fig. 8.16 Example of set and reset boxes

8.3.3 Edge evaluation with pulse output in the function block diagram

The P= box and N= box are available for edge evaluation with pulse output. The binary tag located above the P= box is set for the duration of one program cycle if the result of the preceding logic operation changes from signal state “0” to “1” (rising edge).

With the N= box, the binary tag above the box is set for the duration of one program cycle with a falling edge of the preceding logic operation (change in result of logic operation from “1” to “0”).

The binary tag located above the box is referred to as a “pulse flag”. Suitable for pulse flags are, for example, tags from the bit memory or data area. The edge trigger flag is named under the box and must be a different tag for each edge evaluation (see Chapter 12.3 “Edge evaluation” on page 515).

The edge boxes can be arranged within a logic operation or they can terminate a logic operation. Edge boxes can also be programmed following a T branch.

If additional program elements follow an edge box, for example if the edge box has been placed inside a logic operation, then the signal state at the input of the edge box is passed on directly to the output of the box.

In Fig. 8.17, if the AND logic operation of the tags *#Enable* and *#Measurement.Registered* is fulfilled, the *#Measurement.Load* tag has signal state “1” for the duration of one program cycle. The *#Measurement.Load_EM* tag is the edge trigger flag for edge evaluation.

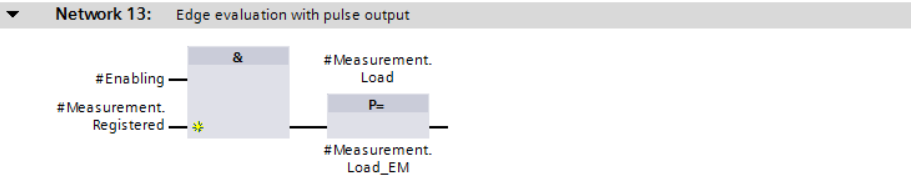


Fig. 8.17 Edge evaluation of result of logic operation (with “pulse flag”)

8.3.4 Multiple setting and resetting (filling the bit array) in the function block diagram

With the result of logic operation “1” at the EN input, the SET_BF box sets the bits of a bit array to signal state “1”. The bit array is defined by the start tag named above the box and the number of bits at the function input N. With result of logic operation “1”, the RESET_BF box resets the bits in the bit array. With result of logic operation “0”, there is no response in both cases.

SET_BF and RESET_BF terminate a logic operation. If the boxes do not have a preceding logic operation, they are always executed.

In Fig. 8.18, with a rising edge from *#Acknowledge*, 16 bits from *“Data.FBD”.Alarm_bit[0]* are set to signal state “1” and 8 bits from *“Output 1”* are reset to signal state “0”. *“Data.FBD”.Alarm_bit* is a tag with the data type ARRAY OF BOOL, *“Output 1”* is a bit in the operand area Outputs.

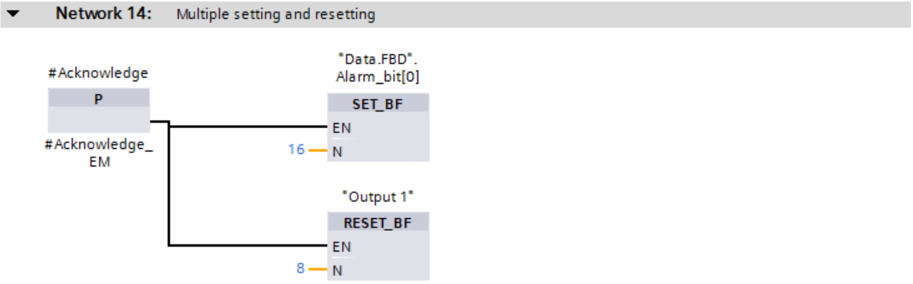


Fig. 8.18 Filling a bit array with SET_BF and RESET_BF

8.3.5 Standard boxes with time response

Starting a SIMATIC timer function

A standard box with a time response starts a SIMATIC timer function with a response as described in Chapter 12.4 “SIMATIC timer functions” on page 524. Available are starting as pulse (SP), as extended pulse (SE), as ON delay (SD), as retentive ON delay (SS), and as OFF delay (SF).

For programming, drag the symbol for the corresponding standard box with the mouse from the program elements catalog under *Basic instructions* > *Timer operations* to the working area.

A standard box with time response requires a preceding logic operation. It can be placed in the middle or at the end of a logic operation and in a T branch. The time operand from the SIMATIC timer functions (T) area is located above the box. The time value is specified in the data format S5TIME at the TV input.

The reset box (R) can be used to reset a SIMATIC timer function. You can find the box in the program elements catalog under *Basic instructions* > *Bit logic operations*. The reset box does not need a preceding logic operation and can be positioned in the middle or at the end of a logic operation and in a T branch.

In Fig. 8.19, the time “*Fan1.on-delay*” is started as an ON delay in the upper current path with the positive edge of *#Fan1.start*. Following expiry of the duration (3 s in the example), the fan is switched on by *#Fan1.drive*. If *#Fan1.start* has signal state “0” prior to expiry of the duration, the fan is not switched on.

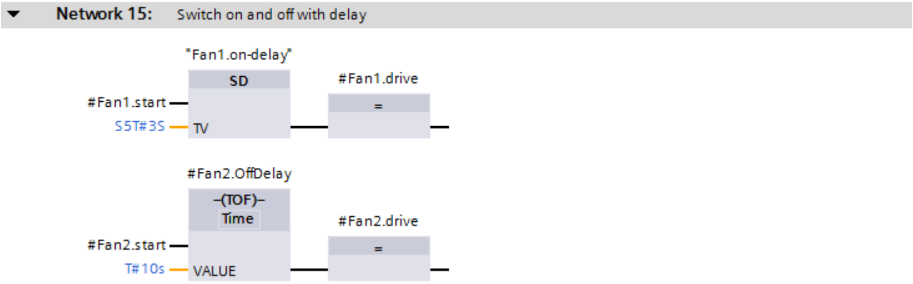


Fig. 8.19 Example of standard boxes with time response

Controlling an IEC timer function

A standard box with time response controls an IEC timer function with a response as described in Chapter 12.4 “SIMATIC timer functions” on page 524. Starting as pulse (TP), as ON delay (TON), as OFF delay (TOF), as accumulating ON delay (TONR), and loading a time function with a duration (PT) and resetting a time function (RT) are available.

For programming, drag the symbol for the corresponding standard box with the mouse from the program elements catalog under *Basic instructions* > *Timer operations* to the working area.

A standard box with time response requires a preceding logic operation. It can be placed in the middle or at the end of a logic operation and in a T branch. The name of the timer function, either the data block for a single instance or the instance name for a local instance, is located above the box. The function for the time response and the data type of the duration, which is specified at the parameter VALUE, are located in the box.

With the PT box, an IEC timer function is loaded with the duration that is indicated at the parameter PT. The RT box resets an IEC timer function. The PT box and the RT box do not need a preceding logic operation and can be positioned in the middle or at the end of a logic operation, and in a T branch.

In Fig. 8.19, the timer function *"Fan2.OffDelay"* is started as an OFF delay in the bottom logic operation with a positive edge of *#Fan2.start*. The box *#Fan2.drive* has signal state "1" if *#Fan2.start* is switched on and for 10 seconds after the switch-off.

8.3.6 Standard boxes with counter response

Controlling a SIMATIC counter function

A standard box with a counter response controls a SIMATIC counter function with a response as described in Chapter 12.6 "SIMATIC counter functions" on page 545. Setting a counter (SC), counting up (CU), and counting down (CD) are available.

For programming, drag the symbol for the corresponding box with the mouse from the program elements catalog under *Basic instructions* > *Counter operations* to the working area.

A standard box with counter response requires a preceding logic operation. It can be placed in the middle or at the end of a logic operation and in a T branch. The counter operand from the SIMATIC counter (C) area is located above the box. The count value in data format WORD is specified at the CV input, where the numerical range extends from W#16#0000 to W#16#0999 or from C#000 to C#999.

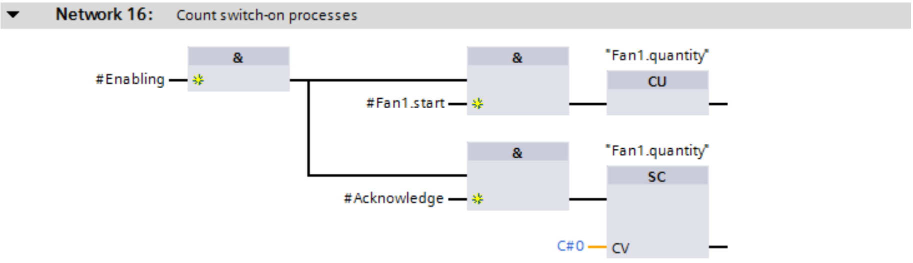


Fig. 8.20 Example of standard boxes with counter response

The reset box (R) can be used to reset a SIMATIC counter function. You can find the box in the program elements catalog under *Basic instructions* > *Bit logic operations*. The reset box does not need a preceding logic operation and can be positioned in the middle or at the end of a logic operation and in a T branch.

In Fig. 8.20, the switch-on processes of *#Fan1.start* are counted with the SIMATIC counter *“Fan1.quantity”*. The *#Acknowledge* signal resets the counter to 0.

8.4 Programming Q boxes with FBD

Q boxes have a binary output named “Q”, which can be linked further. With Q boxes, the memory functions SR and RS, the edge evaluations P_TRIG and N_TRIG, and the timer and counter functions are displayed (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.

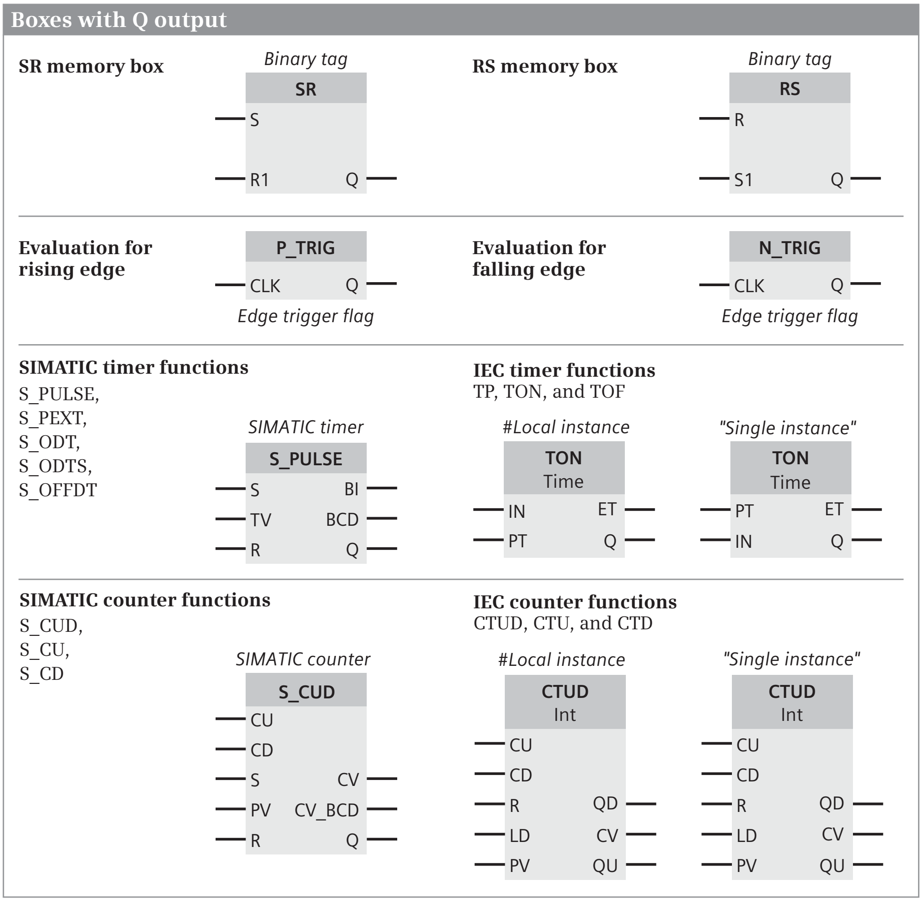


Fig. 8.21 Overview of Q boxes available with FBD

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
- Position boxes following T branches

8.4.1 Memory boxes in the function block diagram

There are two versions of the memory function: as SR box (reset dominant) and as RS box (set dominant). With reset dominant, the memory function is reset or remains reset if both inputs have signal state "1". With set dominant, the memory function is set or remains set in such a case. The response of the memory box is described in Chapter 12.2 "Memory functions" on page 510.

For programming, drag the SR or RS symbol with the mouse from the program elements catalog under *Basic instructions* > *Bit logic operation* to the working area.

Fig. 8.22 shows a binary scaler: Each positive edge of the *#Bin_input* tag changes the signal state of *#Bin_output*. Thus half the input frequency is present at the output.

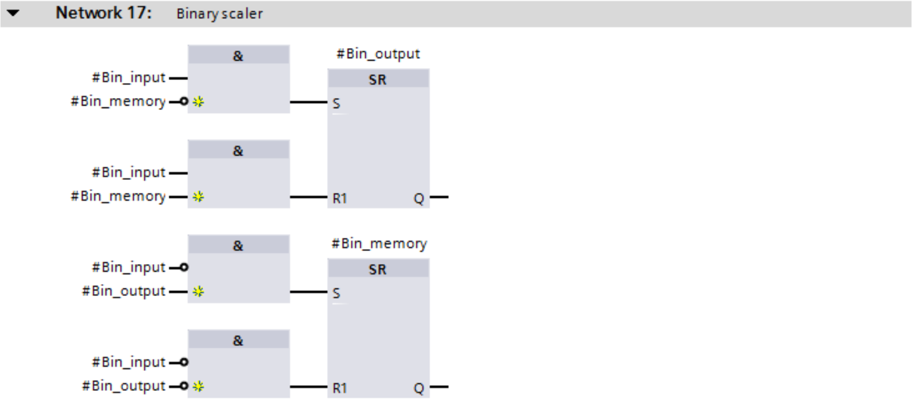


Fig. 8.22 Example of binary scaler

8.4.2 Edge evaluation of the result of logic operation 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 result of the logic operation 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.

For programming, drag the P_TRIG or N_TRIG symbol with the mouse from the program elements catalog under *Basic instructions* > *Bit logic operation* to the working area.

The P_TRIG and N_TRIG boxes require a preceding logic operation and may only be positioned within a logic operation.

In Fig. 8.23, *#Measurement.Memory* is set if *#Measurement_temperature* exceeds an upper limit. In turn, the *#Measurement.Memory* tag sets the *#Measurement.Message* memory. Setting is carried out in both cases by a pulse with positive edge so that acknowledgment is also possible with a set signal present. Acknowledgment is also carried out by a pulse so that, with an acknowledgment signal present, the measured value memory and the alarm memory are set if the upper limit is exceeded again.

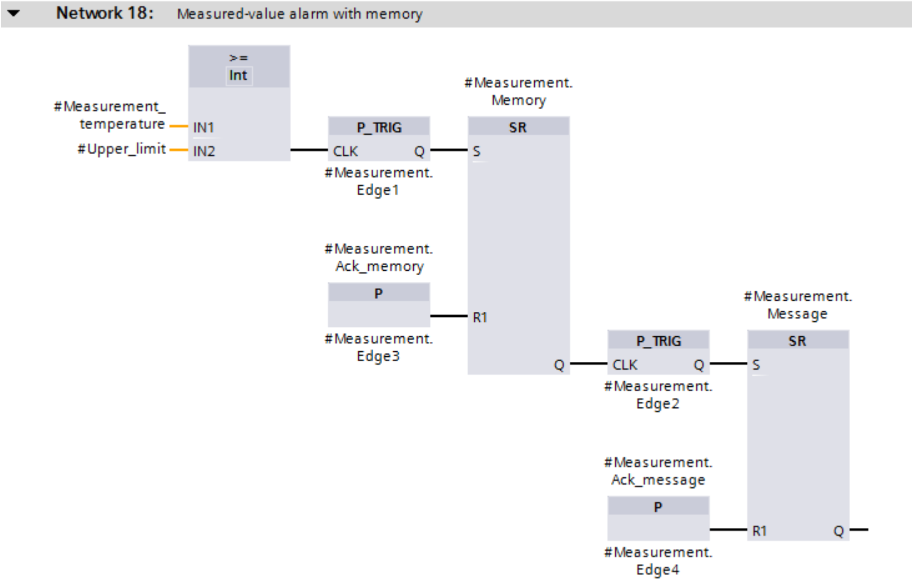


Fig. 8.23 Example of edge evaluation of the result of the logic operation

8.4.3 SIMATIC timer functions in the function block diagram

A SIMATIC timer function can be started as pulse (S_PULSE), as extended pulse (S_PEXT), as ON delay (S_ODT), as retentive ON delay (S_ODTS), or as OFF delay (S_OFFDT). A detailed description of the timer response is provided in Chapter 12.4 “SIMATIC timer functions” on page 524.

For programming, drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions* > *Timer operations* to the working area. You can subsequently change the function using a drop-down list which you can open using the small yellow triangle when the box is selected.

The start input S and the time value TV must be connected; connection of the other box inputs and outputs is optional.

Fig. 8.24 shows a switch-on and switch-off delay. The timer function *“Fan3.on-delay”* is started by *#Fan3.start*. The output Q has signal state “1” after 3 s, which starts the timer function *“Fan3.off-delay”*. At the same time, the *#Fan3.drive* tag is set by the Q output of the box. The Q output continues to have signal state “1” for 10 s after *#Fan3.start* has signal state “0”.

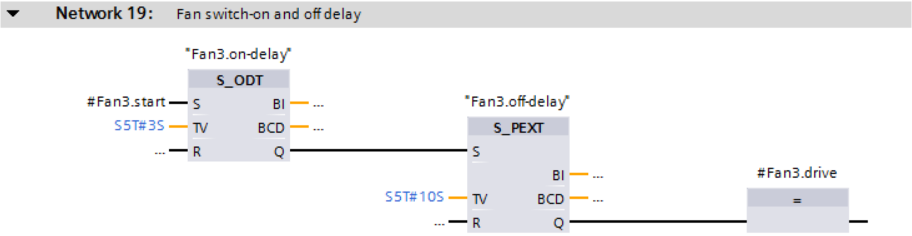


Fig. 8.24 Example of SIMATIC timer functions in the function block diagram

8.4.4 SIMATIC counter functions in the function block diagram

A SIMATIC counter function is available as up counter (S_CU), as down counter (S_CD), or as up/down counter (S_CUD). A detailed description of the counter response is provided in Chapter 12.6 “SIMATIC counter functions” on page 545.

For programming, drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions* > *Counter operations* to the working area. You can subsequently change the function using a drop-down list which you can open using the small yellow triangle when the box is selected.

At least one of the counter inputs (CU or CD) must be connected; connection of the other box inputs and outputs is optional.

Fig. 8.25 shows a down counter. The name of the SIMATIC counter used is positioned above the counter box. *#Quantity_set* sets the counter to the count value W#16#0120. The count value is reduced by 1 with each pulse from *#Workpart_identified*. Once zero has been reached, *#Quantity_reached* is set.

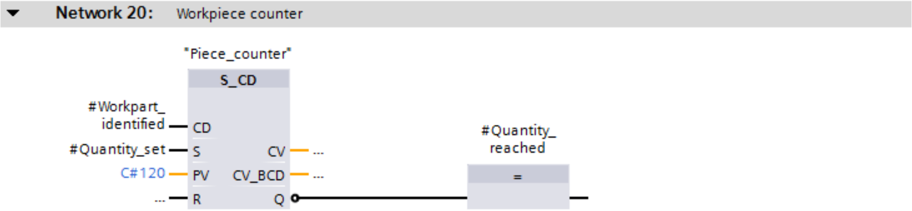


Fig. 8.25 Example of an SIMATIC counter function in the function block diagram

8.4.5 IEC timer functions in the function block diagram

An IEC timer function is available as pulse generation (TP), as ON delay (TON), as OFF delay (TOF), or as accumulating ON delay (TONR). A detailed description of the timer response is provided in Chapter 12.5 “IEC timer functions” on page 539.

For programming, drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions* > *Timer operations* to the working area. When positioning, you select either as single instance or – possible in a function block – as local instance (multi-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*.

You can subsequently change the timer function using a drop-down list which you can open using the small yellow triangle when the box is selected (not with TONR).

With the IEC timer functions, the IN input must have a preceding logic operation and a duration must be connected to the PT input. The Q output can be supplied with an assignment, but cannot be linked further. You can also directly access the output parameters using the instance data, for example with “*<DB_name>”.Q* or “*<DB_name>”.ET* for a single instance.

Fig. 8.26 shows the IEC timer function *#MessageDelay*, which saves its data as local instance in the instance data block of the calling function block. If the *#Measurement_too_high* tag has signal state “1” for longer than 10 s, *#Message_too_high* is set.

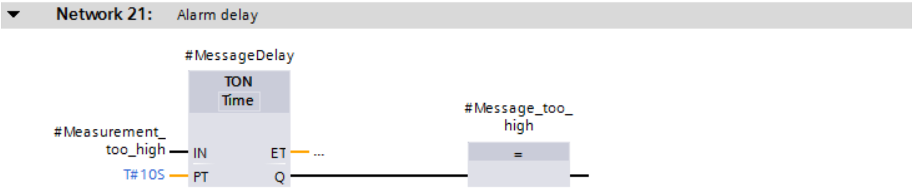


Fig. 8.26 Example of IEC timer functions in the function block diagram

8.4.6 IEC counter functions in the function block diagram

An IEC counter function is available as up counter (CTU), as down counter (CTD), or as up/down counter (CTUD). A detailed description of the counter response is provided in Chapter 12.7 “IEC counter functions” on page 553.

For programming, drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions* > *Counter operations* to the working area. When positioning, you select either as single instance or – possible in a function block – as local instance (multi-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*.

You can subsequently change the timer function using a drop-down list which you can open using the small yellow triangle when the box is selected.

With the IEC counter functions, at least one counter input (CU or CD) must have a preceding logic operation. Connection of the other box inputs and outputs is optional. A standard box can be positioned at the bottom output QU, but not a further logic operation. 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*.

One can also directly access the output parameters using the instance data, for example with *"<DB_name>".QD for a single instance*.

Fig. 8.27 shows the IEC counter function *#LockCounter*, 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.

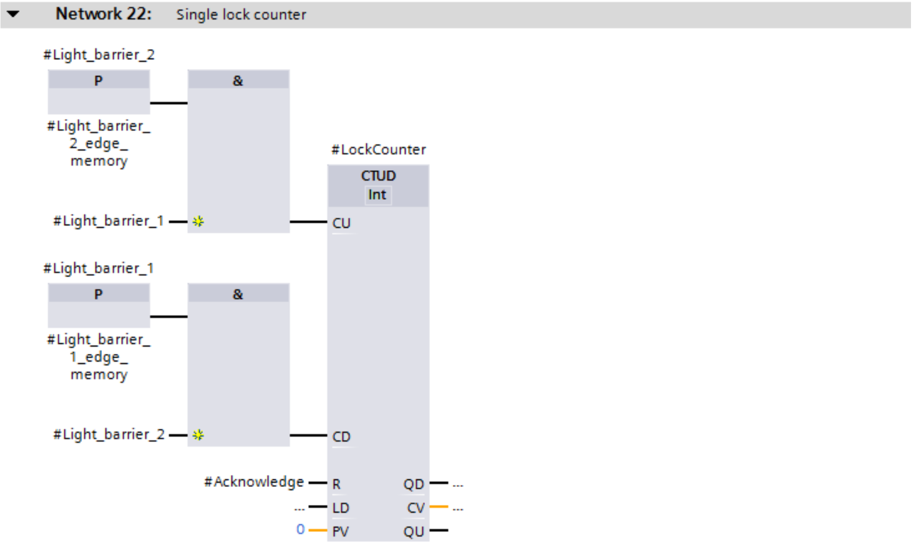
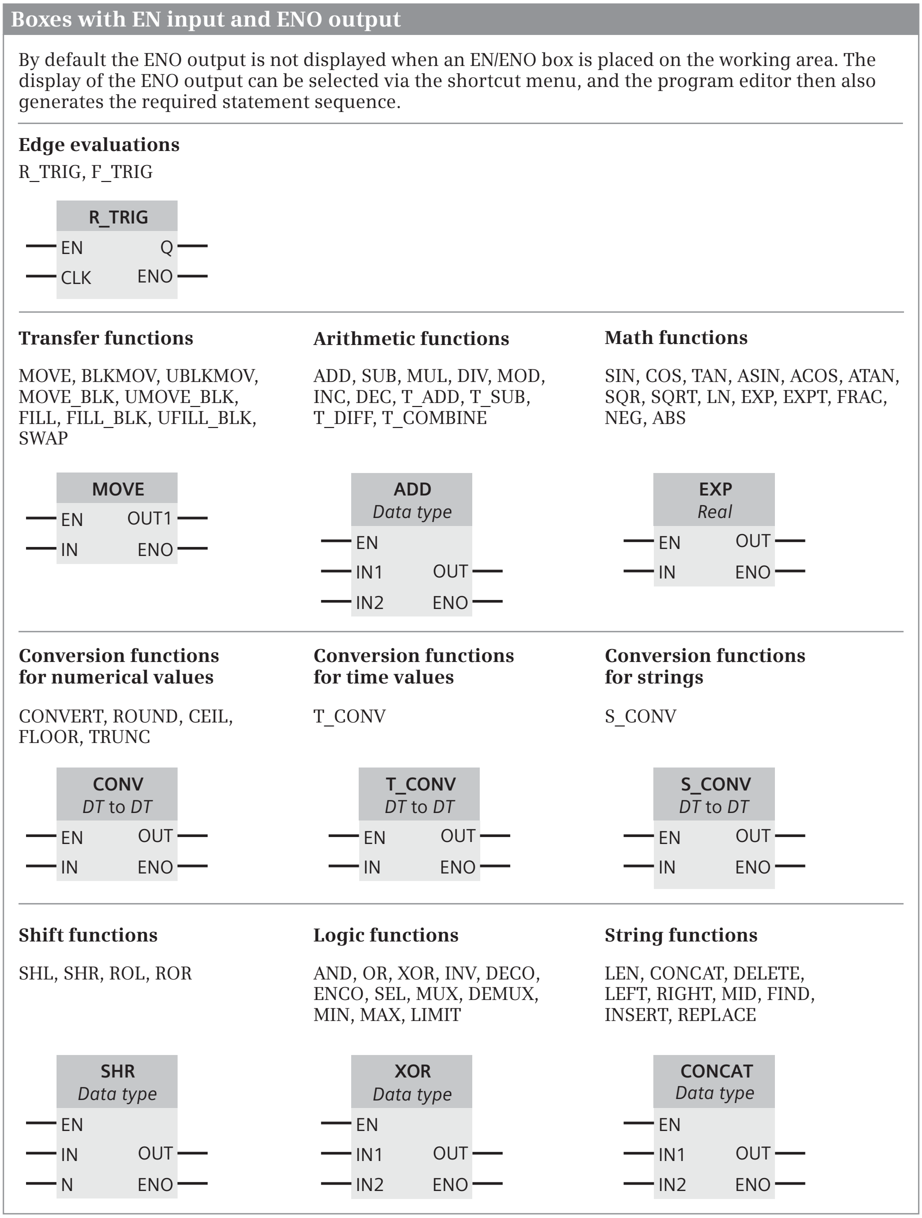


Fig. 8.27 Example of IEC counter functions in the function block diagram

8.5 Programming EN/ENO boxes with FBD

EN/ENO boxes have an enable input EN and an enable output ENO. The enable input can be used to control processing of the box. If an error occurs while the box is being processed, this is displayed at the enable output. Fig. 8.28 provides an overview of the "basic" functions implemented with EN/ENO boxes.

The parameters of the EN/ENO boxes must all be connected. The enable input EN and the enable output ENO are not parameters of the box function. They are used for processing boxes and are added to the box function by the program editor.



Boxes with EN input and ENO output

By default the ENO output is not displayed when an EN/ENO box is placed on the working area. The display of the ENO output can be selected via the shortcut menu, and the program editor then also generates the required statement sequence.

Edge evaluations

R_TRIG, F_TRIG

Transfer functions

MOVE, BLKMOV, UBLKMOV, MOVE_BLK, UMOVE_BLK, FILL, FILL_BLK, UFILL_BLK, SWAP

Arithmetic functions

ADD, SUB, MUL, DIV, MOD, INC, DEC, T_ADD, T_SUB, T_DIFF, T_COMBINE

Math functions

SIN, COS, TAN, ASIN, ACOS, ATAN, SQR, SQRT, LN, EXP, EXPT, FRAC, NEG, ABS

Conversion functions for numerical values

CONVERT, ROUND, CEIL, FLOOR, TRUNC

Conversion functions for time values

T_CONV

Conversion functions for strings

S_CONV

Shift functions

SHL, SHR, ROL, ROR

Logic functions

AND, OR, XOR, INV, DECO, ENCO, SEL, MUX, DEMUX, MIN, MAX, LIMIT

String functions

LEN, CONCAT, DELETE, LEFT, RIGHT, MID, FIND, INSERT, REPLACE

Fig. 8.28 Overview of boxes with enable input EN and enable output ENO

By default, the majority of EN/ENO boxes are displayed without an ENO output when they are moved from the program elements catalog to the working area. Only when you select the command *Generate ENO* from the shortcut menu when the box is selected will the ENO output be displayed and the required statements will be generated during compilation. You can deselect an ENO output using the command *Do not generate ENO* from the shortcut menu.

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 8.6.4 “EN/ENO mechanism in the function block diagram” on page 356. The block calls in the function block diagram which are also shown as EN/ENO boxes are described in Chapter 14.2 “Calling of code blocks” on page 631.

8.5.1 Edge evaluation with an EN/ENO box

The R_TRIG box and the F_TRIG box are available for edge evaluation with an EN/ENO box. A detailed description of the edge evaluation is provided in Chapters 12.3 “Edge evaluation” on page 515 and 12.3.5 “Edge evaluation with an EN/ENO box (LAD, FBD)” on page 520.

For programming, drag one of the edge evaluations with the mouse from the program elements catalog under *Basic instructions* > *Bit logic operations* to the working area. When you release the mouse button, you will be prompted to specify a data area for the instance data. This can be a data block or, if the edge evaluation is programmed in a function block, a local instance (multi-instance) in the instance data block of the function block.

In Fig. 8.29, the *Start* tag is monitored for a rising edge. The instance data is located in the local data of the function block. It consists of the input CLK (in the example: *“Start”* tag), the output Q, and the edge trigger flag. The output Q can also be addressed directly: For a single instance, specify the data block (example: “DB_name”.Q). For a local instance, specify the instance name (example: #Instance_name.Q, in the figure: *#Edge_Start.Q*).

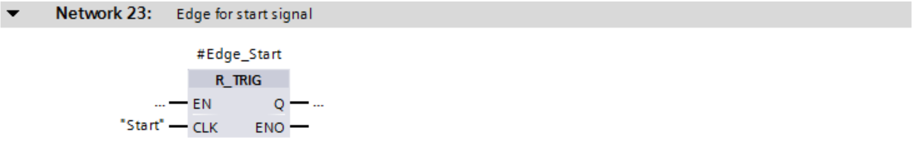


Fig. 8.29 Example of edge evaluation with EN/ENO box in the function block diagram

8.5.2 Transfer functions in the function block diagram

A detailed description of the transfer functions is provided in Chapter 13.2 “Transfer functions” on page 559.

The transfer function MOVE transfers the value of one tag to one or more other tags. MOVE_BLK and UMOVE_BLK transfer individual components from one ARRAY tag to another. BLKMOV and UBLKMOV transfer individual tags or absolutely addressed data areas. FILL_BLK and UFILL_BLK fill components of an ARRAY tag with a value. FILL fills a tag or an absolutely addressed data area with a value. SWAP swaps the order of the bytes in a tag.

For programming, drag the symbol of the transfer function with the mouse from the program elements catalog under *Basic instructions* > *Move operations* to the working area.

In Fig. 8.30, the *#Messages* tag is transferred from the data block *“Data.FBD”* to the *“Alarm bits”* tag in the memory area.

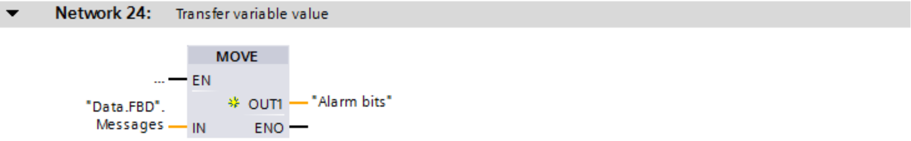


Fig. 8.30 Example of a transfer function in the function block diagram

8.5.3 Arithmetic functions in the function block diagram

A CPU 1500 provides arithmetic functions for numerical values and for time values.

Arithmetic functions for numerical values

An arithmetic function for numerical values implements the basic arithmetical operations with the data formats USINT, UINT, UDINT, ULINT, SINT, INT, DINT, LINT, REAL, and LREAL in the user program. A detailed description of these arithmetic functions is provided in Chapter 13.4 “Arithmetic functions” on page 574.

For programming, drag one of the arithmetic functions (ADD, SUB, MUL, DIV, or MOD) with the mouse from the program elements catalog under *Basic instructions* > *Math functions* to the working area. You can set the function and data types using drop-down lists which you can open using the small yellow triangle when the box is selected. The data type is also automatically set when the first actual value is created.

In Fig. 8.31, the upper limit of a measured value is monitored. A hysteresis is introduced to ensure that the *#Measurement_too_high* alarm does not “pulsate” when the measured value changes slightly in the upper limit range. The alarm *#Measurement_too_high* is only canceled when the measured value has dropped again below the upper limit by the magnitude of the hysteresis.

Arithmetic functions for time values

An arithmetic function for time values adds two durations or one duration to a time (T_ADD), subtracts two durations or one duration from a time (T_SUB), formulates the difference of two times (T_DIFF), or combines a date and a time-of-day into a time (T_COMBINE). A detailed description of these arithmetic functions is provided in Chapter 13.4 “Arithmetic functions” on page 574.

For programming, drag one of the functions (T_ADD, T_SUB, T_DIFF, or T_COMBINE) with the mouse from the program elements catalog under *Extended instruc-*

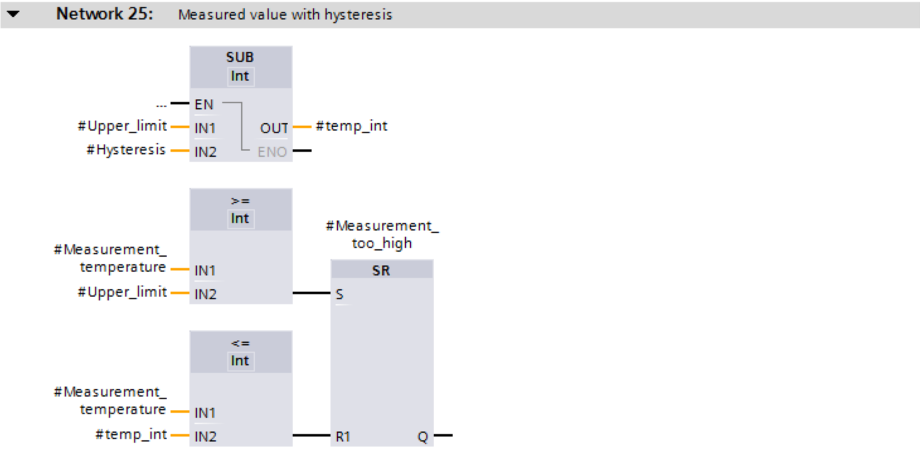


Fig. 8.31 Example of an arithmetic function in the function block diagram

tions > Date and time-of-day to the working area. You can set the function and data types using drop-down lists which you can open using the small yellow triangle when the box is selected.

8.5.4 Math functions in the function block diagram

The math functions comprise, for example, trigonometric functions, exponential functions, and logarithmic functions with tags in the data formats REAL and LREAL. A detailed description of these math functions is provided in Chapter 13.5 “Math functions” on page 578.

For programming, drag one of the math functions (SIN, COS, TAN, ASIN, ACOS, ATAN, SQR, SQRT, LN, EXP, EXPT, FRAC, NEG, ABS) with the mouse from the program elements catalog under *Basic instructions > Math functions* to the working area. You can set the function and data types using drop-down lists which you can open using the small yellow triangle when the box is selected.

Fig. 8.32 shows the calculation of the reactive power according to the equation *#Reactive_power* = *#Voltage* × *#Current* × sin(*#phi*).

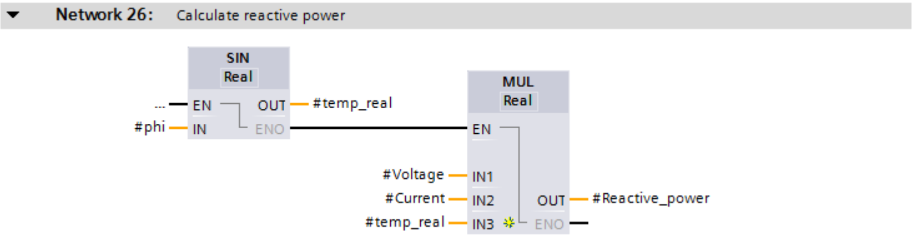


Fig. 8.32 Example of math functions in the function block diagram

8.5.5 Conversion functions in the function block diagram

The conversion functions convert the data formats of tags. A detailed description of the conversion functions is provided in Chapter 13.6 “Conversion functions” on page 586.

For programming, drag one of the conversion functions (CONVERT, ROUND, CEIL, FLOOR, TRUNC, SCALE_X, or NORM_X) with the mouse from the program elements catalog under *Basic instructions* > *Conversion operations* to the working area. You can set the function and data types using drop-down lists which you can open using the small yellow triangle when the box is selected. If the first actual value created has a permissible data type, the data type is also set automatically.

The conversion function T_CONV for data type conversion of date/time can be found in the program elements catalog under *Extended instructions* > *Date and time-of-day*. The conversion function for data type conversions of character strings (S_CONV, STRG_VAL, VAL_STRG, CHARS_TO_STRG, STRG_TO_CHARS, ATH, HTA) can be found in the program elements catalog under *Extended instructions* > *String + Char*.

Fig. 8.33 shows an example of the conversion functions. A measured value present in data format REAL is first converted into data format DINT and then converted into the BCD32 format.

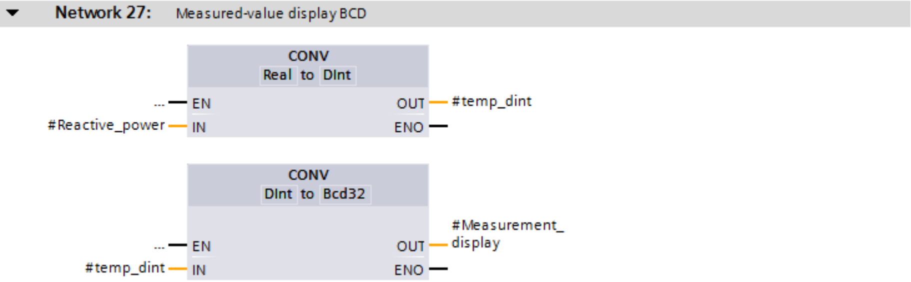


Fig. 8.33 Example of conversion functions in the function block diagram

8.5.6 Shift functions in the function block diagram

The shift functions shift the content of tags bit-by-bit to the left or right. A detailed description of the shift functions is provided in Chapter 13.7 “Shift functions” on page 603.

For programming, drag one of the shift functions (SHL, SHR, ROL, or ROR) with the mouse from the program elements catalog under *Basic instructions* > *Shift and rotate* to the working area. You can set the function and data types using drop-down lists which you can open using the small yellow triangle when the box is selected. The data type is also automatically set when the first actual value is created.

In Fig. 8.34, the respective three decades of two numbers present in BCD16 format of a SIMATIC counter are joined without gaps. Using the shift function SHL – set to data type DWORD! – the *#Quantity_high* tag is shifted to the left by 12 bits, corre-

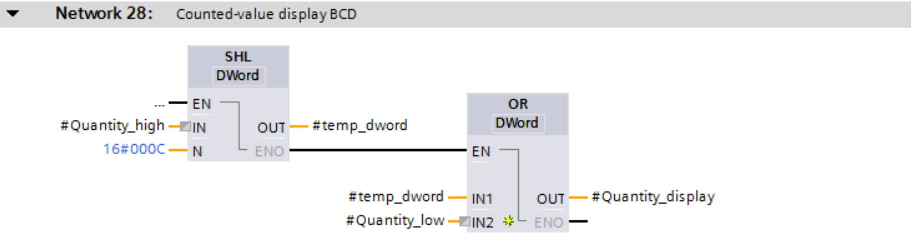


Fig. 8.34 Example of shift functions in the function block diagram

sponding to three decades. A small square on the input parameter IN indicates that the data type of the applied tag (WORD in the example) does not agree with the data type of the function (DWORD in the example) and will be converted implicitly.

The bottom three decades (the *#Quantity_low* tag) are subsequently added by a doubleword logic operation according to OR and output to the *#Quantity_display* tag.

8.5.7 Logic functions in the function block diagram

The logic functions include the word logic operations AND, OR, XOR, the inversion INVERT, the coding functions DECO and ENCO, the selection functions SEL, MUX, DEMUX, MIN, MAX, and the limiting function LIMIT. A detailed description of the logic functions is provided in Chapter 13.8 “Logic functions” on page 607. In the program elements catalog, the logic functions are located under *Basic instructions* > *Word logic operations* (AND, OR, XOR, INVERT, DECO, ENCO, SEL, MUX, and DEMUX) and under *Basic instructions* > *Math functions* (MIN, MAX, and LIMIT).

Word logic operations

The word logic operations link each bit of two tags according to an AND, OR, or exclusive OR function. For programming, drag one of the word logic operations (AND, OR, XOR, INV) with the mouse from the program elements catalog under *Basic instructions* > *Word logic operations* to the working area. You can set the function and data types using drop-down lists which you can open using the small yellow triangle when the box is selected. The data type is also automatically set when the first actual value is created.

Fig. 8.35 shows how you can program 32 edge evaluations simultaneously for rising and falling edges. The alarm bits are collected in a doubleword *Messages*, which is present in data block “*Data.FBD*”. 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 alarm bit has changed. In order to obtain the positive signal edges, the changes are linked to the alarms by an AND logic operation. The bit is set for a rising signal edge wherever the alarm and the change each have a “1”. This corresponds to the pulse flag of the edge evaluation. If you do the same with the negated alarm bits – the alarm 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 alarms.

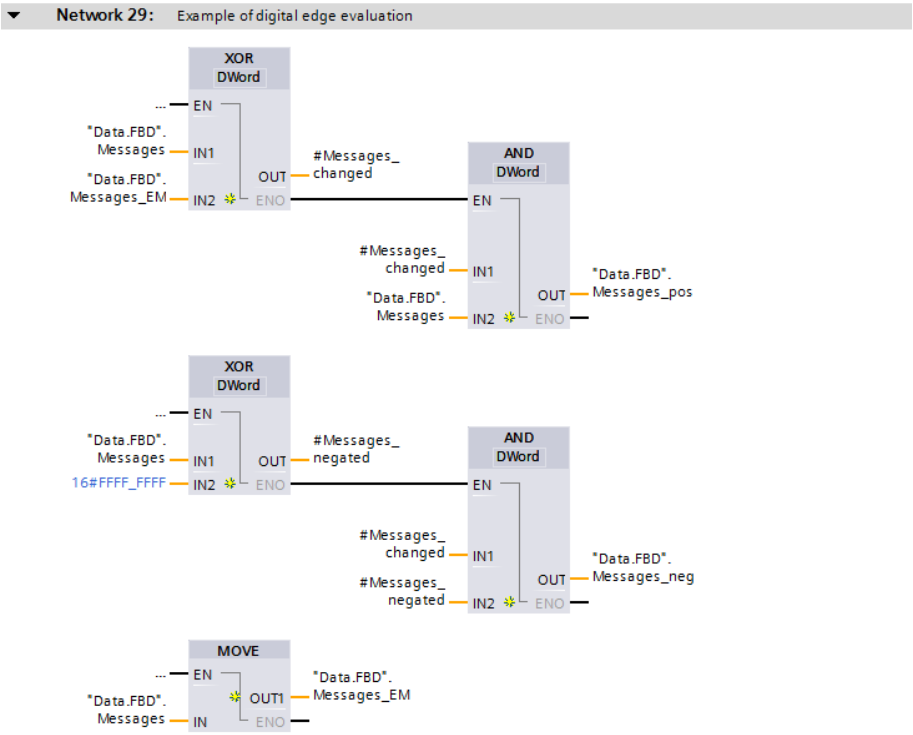


Fig. 8.35 Example of word logic operations in the function block diagram

8.5.8 Functions for character strings in the function block diagram

Character strings are tags with the data type STRING. With the functions for character strings, parts of a character string can be extracted (LEFT, RIGHT, MID), inserted (INSERT), replaced (REPLACE) or deleted (DELETE), two character strings can be combined (CONCAT), and the length of a character string (LEN) or the position of a character in a character string (FIND) can be determined.

A detailed description of these functions is provided in Chapter 13.9 “Processing of strings (data type STRING)” on page 615.

Fig. 8.36 shows the connection of the STRING tags *#Station.Name* and *#Station.Number* to form the tag *#Station.Identification*. In the program elements catalog, the string functions are located under *Extended instructions* > *String + Char*.

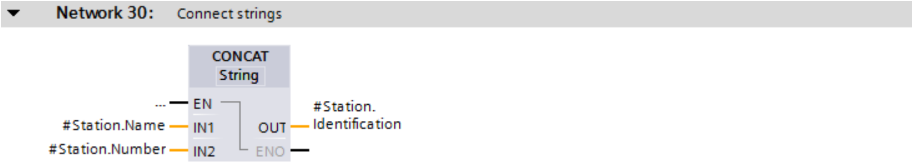


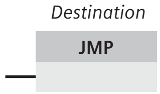
Fig. 8.36 Example of character string functions in the function block diagram

8.6 Program control with FBD

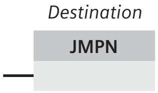
You can influence execution of the user program by means of the program control functions. You use jump functions to exit linear program execution and continue at a different point in the block. Block call functions cause the continuation of program execution in a different block. A block end function terminates the execution

Functions for program control

Jump functions

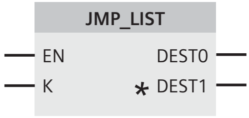


Conditional jump if RLO = "1" or absolute jump if input is not assigned.



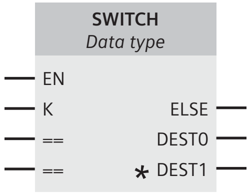
Conditional jump if RLO = "0".

Jump list



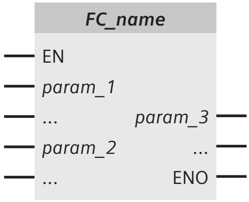
Program branch:
Jump marks are specified at the DESTx parameters to which a branch is made depending on the value at parameter K.

Jump distributor



Program branch:
Jump marks are specified at the DESTx parameters to which a branch is made depending on a comparison with the value at parameter K.

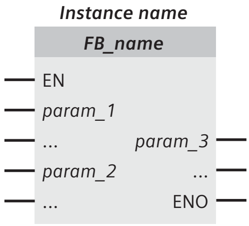
Block calls



Calling a function (FC)

Via EN the call can be controlled depending on the RLO. Via ENO the block can return a group error message.

All parameters param_x must be supplied with values.

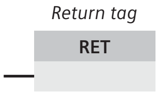


Calling a function block (FB)

Via EN the call can be controlled depending on the RLO. Via ENO the block can return a group error message.

The parameters param_x are supplied with values as required.

Block end function



Conditional block end if RLO = "1" or absolute block end if input is not assigned. The value of the return tag is transferred to the ENO enable output.

Fig. 8.37 Overview of functions for program control in the function block diagram

in the block. The functions that are available in the function block diagram are shown in Fig. 8.37.

8.6.1 Jump functions in the function block diagram

A detailed description of the jump functions is provided in Chapter 14.1 "Jump functions" on page 623.

Jump functions JMP and JMPN

For programming a jump function, drag the symbol of a jump function with the mouse from the program elements catalog under *Basic instructions* > *Program control operations* to the working area. You define the jump label (the jump destination) using the jump box. To program the jump destination, use the mouse to drag the *Label* function to the start of the network with which processing of the program is to be continued from the program elements catalog under *Basic instructions* > *Program control operations* and write the label into the box.

You can subsequently set the jump function (JMP or JMPN) via a drop-down list which you can open using the small yellow triangle when the box is selected. If the box with the jump function JMP does not have a preceding logic operation, the jump is always carried out (absolute jump). The jump function JMPN always requires a preceding logic operation.

It is only possible to jump within a block. A jump function cannot be programmed in association with a T branch. Only one jump function or block end function is permissible per network.

Example of loop jump

Fig. 8.38 shows a jump function using a program loop as an example. In a *#Quantity* array with 16 components from *#Quantity[0]* to *#Quantity[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 *#Quantity[#Index]*. If *#MaxValue* is less than *#Quantity[#Index]*, it is overwritten with the larger value of *#Quantity[#Index]*. *#Index* is then increased by +1. As long as *#Index* is less than or equal to 15, it is jumped to the beginning of the program loop (to the jump destination *MaxSearch*) and the program section is executed again.

Jump list JMP_LIST

The jump list is represented as a box. The box is only processed if the EN input signal state is "1". The value of parameter K (0 to 99) determined 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.

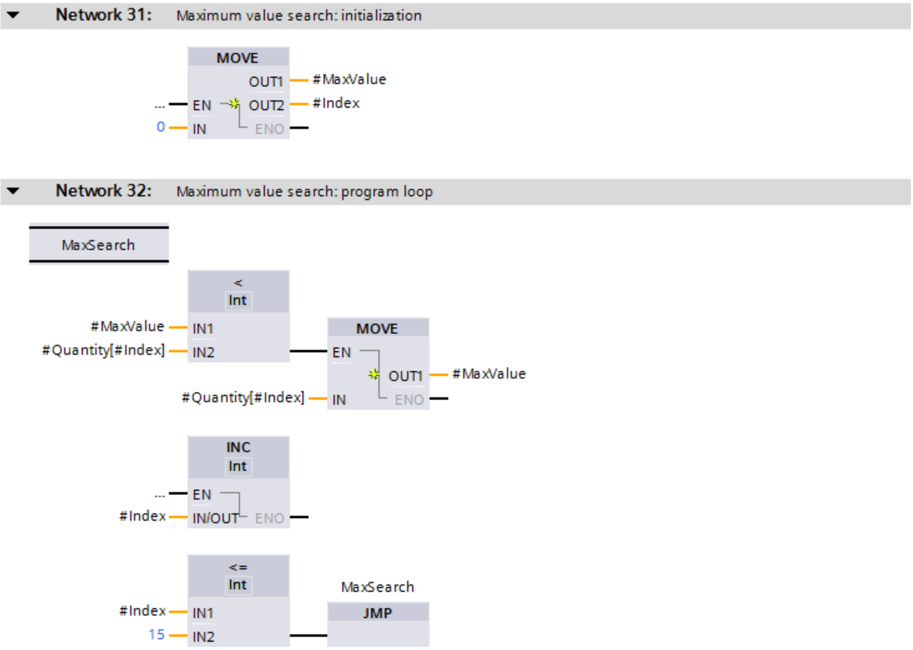


Fig. 8.38 Example of a program loop with conditional jump

Jump distributor SWITCH

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 comparison is fulfilled, program execution 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.

8.6.2 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 the name of the local instance) is additionally present above the box. A detailed description of the block calls is provided in Chapter 14.2 “Calling of code blocks” on page 631.

To call a code block, use the mouse to drag the block which has already been programmed from the project tree under *Program blocks* into the working area. With a logic operation preceding the EN input you can structure the block call depending on conditions.

The top network in Fig. 8.39 shows the call of a function (FC). The function name is present as title in the call box. In the event of an error in the block (ENO is then “0”),

#Adder_error is set to “1” and a jump is made to the network with the *Error* label. In the bottom network, the call of a function block is present as a single instance. The name of the function block is present as the title in the call box, the instance name – in this case the name of the instance data block – is present above the box. If the block reports an error with ENO = “0”, the block is exited with the RET box and the value FALSE.

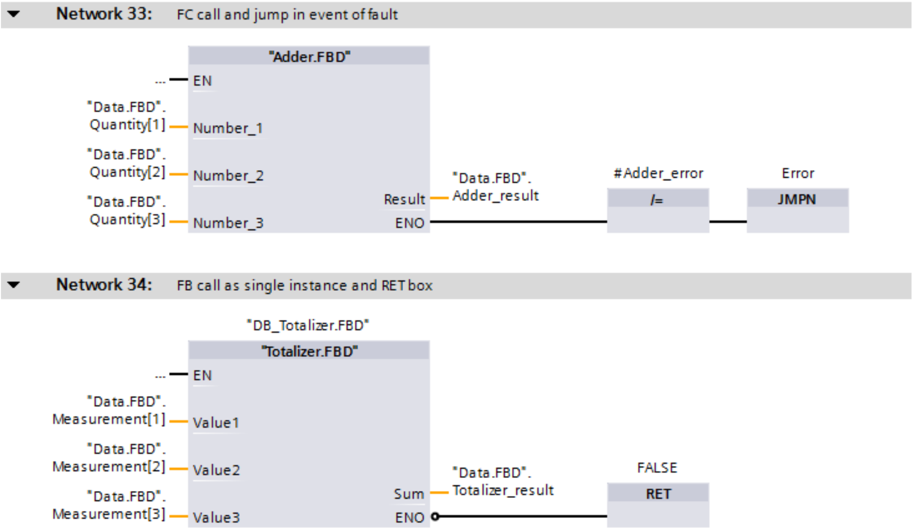

Network 33: FC call and jump in event of fault
"Adder.FBD"
... EN
"Data.FBD". Quantity[1] Number_1
"Data.FBD". Quantity[2] Number_2
"Data.FBD". Quantity[3] Number_3
Result "Data.FBD". Adder_result
ENO
#Adder_error
/=
Error
JMPN
Network 34: FB call as single instance and RET box
"DB_Totalizer.FBD"
"Totalizer.FBD"
... EN
"Data.FBD". Measurement[1] Value1
"Data.FBD". Measurement[2] Value2
"Data.FBD". Measurement[3] Value3
Sum "Data.FBD". Totalizer_result
ENO
FALSE
RET

Fig. 8.39 Examples of functions for program control in the function block diagram

8.6.3 Block end function in the function block diagram

To program the block end function, drag the RET box with the mouse from the program elements catalog under *Basic instructions* > *Program control operations* to the working area. Now define which value the ENO output of the exited block should have. You select the RET box and use the small yellow triangle to select the drop-down menu item

- ▷ *Ret*, then it is the current result of logic operation RLO (i.e. signal state “1”),
- ▷ *Ret True*, then it is the signal state “1” (TRUE),
- ▷ *Ret False*, then it is the signal state “0” (FALSE),
- ▷ *Ret Value*, then it is the signal state of the return tag above the RET box.

A detailed description of the RET box is provided in Chapter 14.3 “Block end functions” on page 636.

In the second network in Fig. 8.39, the block with the RET box is left if the *“Totalizer.FBD”* block signals an error. The ENO output of the exited block is set then to signal state “0” (FALSE).

8.6.4 EN/ENO mechanism in the function block diagram

The EN/ENO mechanism allows the execution of program functions (statements) and blocks depending on the result of logic operation. The enable input EN enables the execution of a program function or a block. The enable output (ENO) reports an error in program execution that occurred during runtime. The enable input EN and the enable output ENO are both of data type BOOL.

EN and ENO are not function or block parameters; they are not declared in the block interface. They are statement sequences which the program editor generates before and after a function or block call.

EN/ENO mechanism with program functions (instructions)

The program editor in FBD displays program functions with the EN/ENO mechanism using EN/ENO boxes. These are the functions described in Chapter 8.5 "Programming EN/ENO boxes with FBD" on page 344, which can be found in the program elements catalog in the *Basic instructions* tab.

If the enable input EN is not occupied or if it has a preceding logic operation which delivers signal state "1" during runtime, then the function is carried out. If the signal state is "0" at the EN input, the function is not carried out and program execution is continued with the next program element.

The enable output ENO provides signal state "1" if the function has been executed without any errors. If an error occurred during execution of the function or if signal state "0" was present at the EN input, resulting in the function not being executed, then the ENO output has signal state "0".

The enable output ENO is not displayed by default in most functions, when you drag them from the program elements catalog to the working area. Then no additional program code is generated for error detection. Using the *Generate ENO* command from the shortcut menu, you can switch on error detection for the selected function, then the ENO output will be displayed and an additional program code will be generated during compilation. The ENO output is "deselected" using the command *Do not generate ENO*.

Controlling a processing sequence

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. In this manner it is possible, for example, to "switch off" the complete sequence, or the rest of the sequence is no longer processed if a box signals an error.

In the example in Fig. 8.40, neither of the boxes is processed if the *#Enabling* tag has signal state "0". If an error occurs during processing of the ADD box, for example a numerical range is exceeded, the subsequent SQRT box is no longer processed.

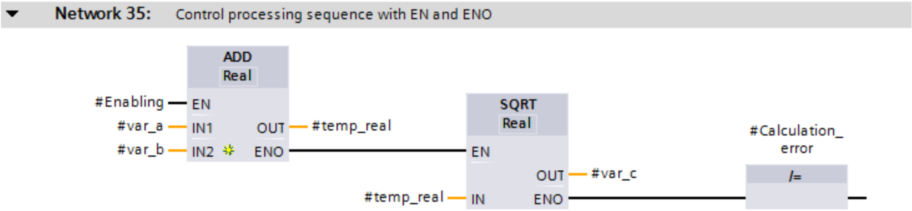


Fig. 8.40 Example of series connection of ENO and EN with FBD

EN/ENO mechanism with blocks

When calling blocks (FC functions and FB function blocks), the program editor always displays the input EN and the output ENO, regardless of which programming language the blocks are programmed in. In this context, blocks also include all of the functions in the program elements catalog that are not listed in the *Basic instructions* tab (called "system blocks" in the following).

If the enable input EN is not occupied or if it has a preceding logic operation which delivers signal state "1" during runtime, then the block is called. If the signal state is "0" at the EN input, a block call is not carried out and the program execution is continued after the block call.

For "system blocks", the enable output ENO provides signal state "1" if the function has been executed without any errors. If an error occurred during execution of the "system block" or if signal state "0" was present at the EN input, resulting in the "system block" not being executed, then the ENO output has signal state "0".

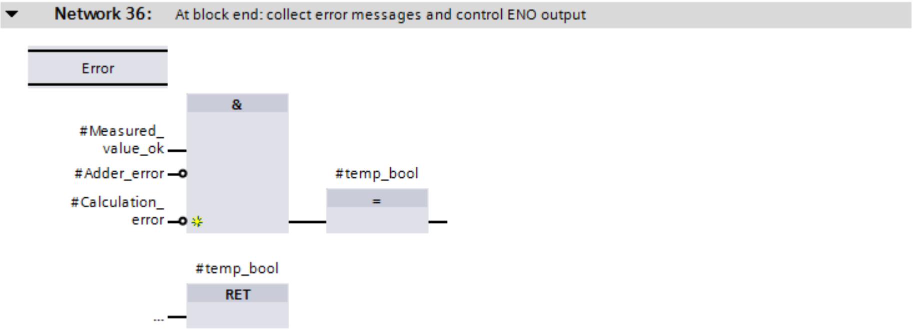
For self-written blocks (FC functions and FB function blocks), it is the responsibility of the user to determine which signal state the enable output ENO provides. By default – without user action – the ENO output signal state is "1". In the event of an error, if you would like to evaluate an error in the calling block, you must set the ENO output to signal state "0".

Controlling the ENO output for self-written blocks

The signal state of the ENO output is controlled in the function block diagram with the RET box (see Chapter 8.6.2 "Block call functions in the function block diagram" on page 354). In principle, you can end the execution in the block in the event of any detected error by using the RET box and the return value FALSE and then no longer execute the remainder of the block program. You can also keep a detected error in "error tags", allow the remaining part of the block program to be executed and then, at the end of the block, terminate the block with FALSE in the event of an error.

An example is shown in Fig. 8.41. In the event of an error, the tag *#Measured_value_ok* has signal state "0" and the tags *#Adder_error* and *#Calculation_error* have signal state "1". In the event of an error, the tag *#temp_bool* is set to signal state "0".

Upon exiting the block with the RET box, the signal state of the tag *#temp_bool* is assigned to the ENO output.



Network 36: At block end: collect error messages and control ENO output

Fig. 8.41 Control of the ENO output

9 Structured Control Language SCL

9.1 Introduction

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 12 “Basic functions” on page 503, 13 “Digital functions” on page 558, and 14 “Program control” on page 622.

Use of the program and symbol editor, which generally applies to all programming languages, is described in Chapter 6 “Program editor” on page 247.

SCL is used to program the contents of blocks (the user program). What blocks are, and how they are created, is described in Chapters 5.3.1 “Block types” on page 155 and 6.3 “Programming a code block” on page 253.

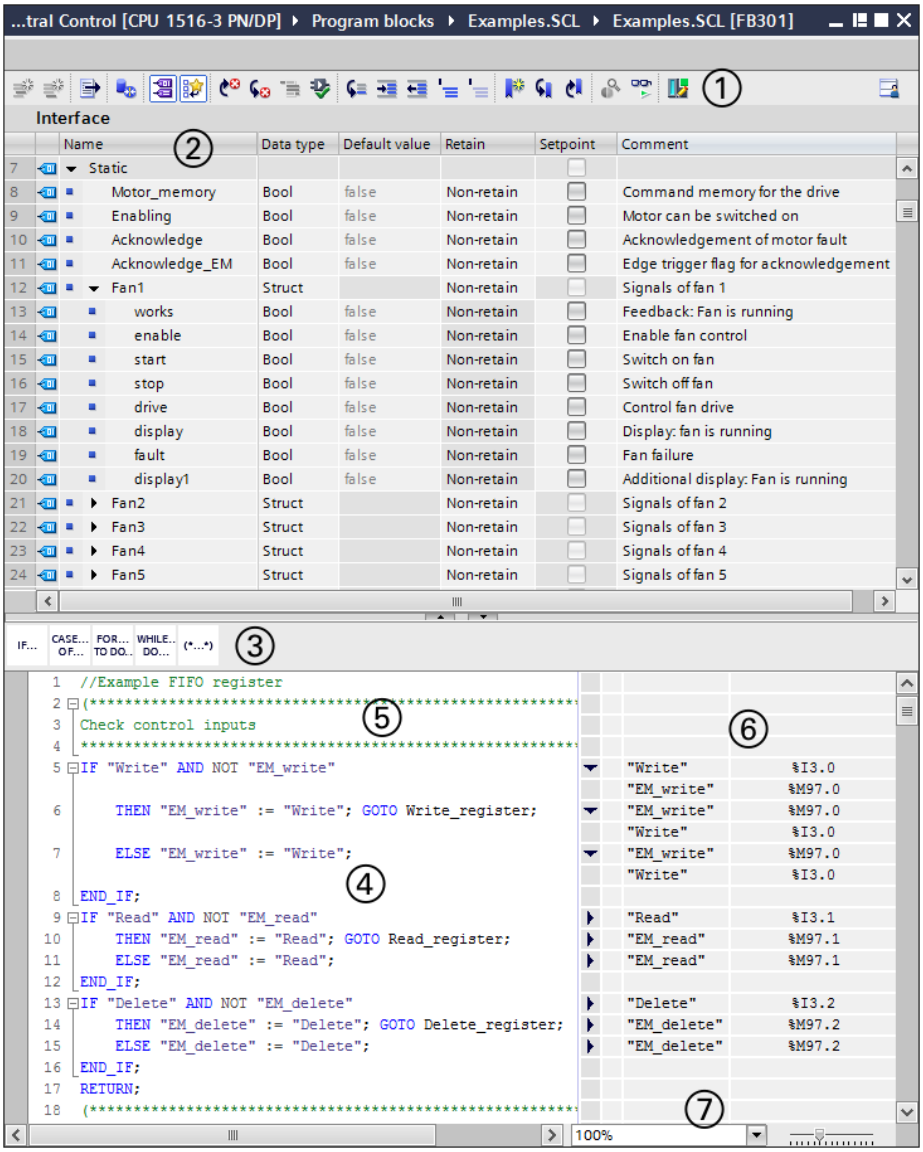
9.1.1 Programming with SCL in general

You use SCL to program the control function of the programmable controller – the user program (control program). The user program is organized in different types of blocks.

Fig. 9.1 shows the program editor's working window. The icons in the toolbar ① can be used to set the display of the working area, e.g. the opening and closing of the parameter list of blocks, and additional functions such as monitoring of the program status. The interface of the block ② in the upper part of the working window lists the block parameters and local data. The favorites bar ③ can be expanded by additional program elements. It can also be hidden. The control function, i.e. the list of SCL statements, is displayed in the working area ④.

You can make the SCL program clearer and easier to read by using comments and empty lines ⑤. Comments and empty lines have no influence on the function and length of the compiled 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. If it is activated, the absolute addressing of the tags is displayed on the right edge of the line ⑥. The size of the font can be adjusted using the zoom setting ⑦.

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.


...tral Control [CPU 1516-3 PN/DP] ▸ Program blocks ▸ Examples.SCL ▸ Examples.SCL [FB301]
1
Interface
2
IF...
CASE... OF...
FOR... TO DO..
WHILE.. DO...
(*...*)
3
5
6
4
7
100%

	Name	Data type	Default value	Retain	Setpoint	Comment
7	Static				□	
8	Motor_memory	Bool	false	Non-retain	□	Command memory for the drive
9	Enabling	Bool	false	Non-retain	□	Motor can be switched on
10	Acknowledge	Bool	false	Non-retain	□	Acknowledgement of motor fault
11	Acknowledge_EM	Bool	false	Non-retain	□	Edge trigger flag for acknowledgement
12	Fan1	Struct		Non-retain	□	Signals of fan 1
13	works	Bool	false	Non-retain	□	Feedback: Fan is running
14	enable	Bool	false	Non-retain	□	Enable fan control
15	start	Bool	false	Non-retain	□	Switch on fan
16	stop	Bool	false	Non-retain	□	Switch off fan
17	drive	Bool	false	Non-retain	□	Control fan drive
18	display	Bool	false	Non-retain	□	Display: fan is running
19	fault	Bool	false	Non-retain	□	Fan failure
20	display1	Bool	false	Non-retain	□	Additional display: Fan is running
21	Fan2	Struct		Non-retain	□	Signals of fan 2
22	Fan3	Struct		Non-retain	□	Signals of fan 3
23	Fan4	Struct		Non-retain	□	Signals of fan 4
24	Fan5	Struct		Non-retain	□	Signals of fan 5

```
//Example FIFO register
(*****************************************************************
Check control inputs
*****************************************************************
IF "Write" AND NOT "EM_write"

    THEN "EM_write" := "Write"; GOTO Write_register;

    ELSE "EM_write" := "Write";

END_IF;
IF "Read" AND NOT "EM_read"
    THEN "EM_read" := "Read"; GOTO Read_register;
    ELSE "EM_read" := "Read";
END_IF;
IF "Delete" AND NOT "EM_delete"
    THEN "EM_delete" := "Delete"; GOTO Delete_register;
    ELSE "EM_delete" := "Delete";
END_IF;
RETURN;
(*****************************************************************
```

```
"Write"       %I3.0
"EM_write"    %M97.0
"EM_write"    %M97.0
"Write"       %I3.0
"EM_write"    %M97.0
"Write"       %I3.0

"Read"        %I3.1
"EM_read"     %M97.1
"EM_read"     %M97.1

"Delete"      %I3.2
"EM_delete"   %M97.2
"EM_delete"   %M97.2
```

Fig. 9.1 Example of a block with SCL program

In order to program an SCL statement, use the keyboard to enter the statements in a line of the input field. The program elements catalog provides you with an overview of the existing functions. Dragging a function 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 them from the *Program blocks* folder into a line.

9.1.2 SCL statements and operators

The SCL program consists of a sequence of individual 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* control 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 operand (a single tag) or several operands (tags) which are linked by operators.

Example: "a + b" is an expression; "a" and "b" are operands, "+" 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 left to right.

Expressions

An expression is a formula for calculating a value and consists of operands (tags) and operators. In the simplest case, an expression is an operand, a tag, or a constant. A sign or a negation can also be included.

An expression can consist of operands which 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 in SCL are 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).

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

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	:	Tag	:=	Expression	;	//	Comment
		#Result	:=	#Tag **AND** #Tag	;		Logical expression
		#Result	:=	#Tag **>=** #Tag	;		Comparison expression
		#Result	:=	#Tag **+** #Tag	;		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 commences 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 instructions* in the program elements catalog are calls of system blocks with return value.

Block call

Label	:	Block name (parameter list)	;	//	Comment
		Tag := block name (parameter list)	;		

Fig. 9.2 Types of SCL statements

Table 9.1 Operators with SCL

Logic operation	Name	Operator	Priority
Parentheses	Left parenthesis, right parenthesis	(,)	1
Arithmetic	Power	**	2
	Unary plus, unary minus (sign)	+, –	3
	Multiplication, division	*, /, 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 operand

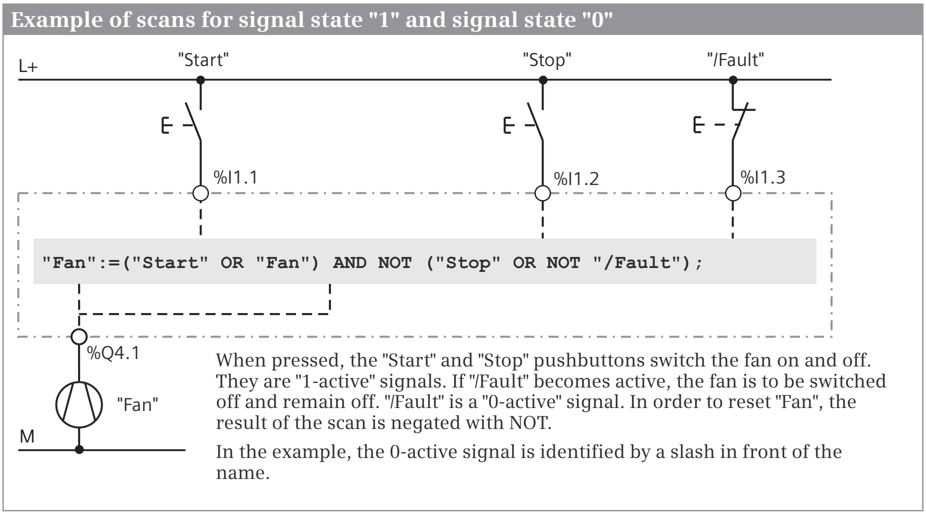
Table 9.2 Binary logic operations with SCL

Operation	Operand	Function
&	Binary operand or binary expression	Scan for signal state "1" and link according to AND logic operation
AND	Binary operand or binary expression	Scan for signal state "1" and link according to AND logic operation
OR	Binary operand or binary expression	Scan for signal state "1" and link according to OR logic operation
XOR	Binary operand or binary expression	Scan for signal state "1" and link according to exclusive OR logic operation
NOT	–	Negation of result of logic operation

in order to negate the result of scan. NOT can also be used to negate the result of binary expressions.

The example in Fig. 9.3 shows the two "Start" and "Stop" pushbuttons. When pressed, they output the signal state "1" in the case of an input module with sinking input. This signal state is used in the logic operation.

The "/Fault" signal is not active in the normal case. Signal state "1" is then present and is negated by means of the NOT operator, and therefore it does not result in resetting of the "Fan" tag. If "/Fault" becomes active, the "Fan" tag is to be reset. The active "/Fault" signal delivers signal state "0" and results in resetting of "Fan".



Example of scans for signal state "1" and signal state "0"
L+
"Start"
"Stop"
"/Fault"
%I1.1
%I1.2
%I1.3
"Fan":=("Start" OR "Fan") AND NOT ("Stop" OR NOT "/Fault");
%Q4.1
"Fan"
M

When pressed, the "Start" and "Stop" pushbuttons switch the fan on and off. They are "1-active" signals. If "/Fault" becomes active, the fan is to be switched off and remain off. "/Fault" is a "0-active" signal. In order to reset "Fan", the result of the scan is negated with NOT.

In the example, the 0-active signal is identified by a slash in front of the name.

Fig. 9.3 Scanning for signal states “1” and “0”

The logic expression in the example uses NOT both for negation of the result of scan of “/Fault” and for negation of the result of the second OR function. You can also formulate the logic operation differently:

```
"Fan":=("Start" OR "Fan") AND NOT "Stop" AND "/Fault";
```

9.2.2 AND function in SCL

An AND function is fulfilled if all function inputs have the result of scan “1”. A description of the AND function is provided in Chapter 12.1.3 “AND function, series connection” on page 507.

SCL implements the AND logic operation using a logic expression with the operators & or AND, which link binary tags or binary expressions.

Fig. 9.4 shows an example of an AND logic operation. The *#Fan1.works* and *#Fan2.works* tags are scanned for signal state “1” and the two scan results are linked according to an AND logic operation. The AND function is fulfilled (delivers signal state “1”) if both fans are running.

```
//AND function
#Display.twoFans := #Fan1.works AND #Fan2.works;
//OR function
#Display.Min_oneFan := #Fan1.works OR #Fan2.works;
//Exclusive OR function
#Display.oneFan := #Fan1.works XOR #Fan2.works;
```

Fig. 9.4 Examples of binary logic operations with SCL

9.2.3 OR function in SCL

An OR function is fulfilled if one or more function inputs have the result of scan “1”. A description of the OR function is provided in Chapter 12.1.4 “OR function, parallel connection” on page 507.

SCL implements the OR logic operation using a logic expression with the operator OR, which links binary tags or binary expressions.

Fig. 9.4 shows an example of an OR logic operation. The *#Fan1.works* and *#Fan2.works* tags are scanned for signal state “1” and the two scan results are linked according to an OR logic operation. The OR function is fulfilled (delivers signal state “1”) if one of the fans is running or if both fans are running.

9.2.4 Exclusive OR function in SCL

An exclusive OR function (antivalence function) is fulfilled if an odd number of function inputs has the result of scan “1”. A description of the exclusive OR function is provided in Chapter 12.1.5 “Exclusive OR function, non-equivalence function” on page 508.

SCL implements the exclusive OR logic operation using a logic expression with the operator XOR, which links binary tags or binary expressions.

Fig. 9.4 shows an example of an exclusive OR logic operation. The *#Fan1.works* and *#Fan2.works* tags are scanned for signal state “1” and the two scan results are linked by an exclusive OR logic operation. The exclusive OR function is fulfilled (delivers signal state “1”) if only one of the fans is running.

9.2.5 Combined binary logic operations in SCL

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. 9.5. The first AND function is fulfilled if fan 1 is running and fan 2 is not running, the second function if fan 1 is not running and fan 2 is running. The *#Display.oneFan_1* tag is set if the first AND function is fulfilled

```
//ORing of AND functions - does not require parentheses
#Display.oneFan_1 := #Fan1.works AND NOT #Fan2.works
   OR NOT #Fan1.works AND #Fan2.works;
```

```
//ANDing of OR functions - parentheses required
#Display.oneFan_2 := (#Fan1.works OR #Fan2.works)
   AND (NOT #Fan1.works OR NOT #Fan2.works);
```

Fig. 9.5 Examples of combined binary logic operations with SCL

or if the second AND function is fulfilled (or if both are fulfilled, but this is not the case in this example).

This logic operation does not require parentheses since the AND function is processed “before” the OR function because of its higher priority. This also applies to ORing of exclusive OR functions or the exclusive ORing of AND functions.

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 bottom example in Fig. 9.5. The first OR function is fulfilled if at least one fan is running or if both fans are running, the second if at least one fan is not running or if neither fan is running. The two OR functions are present in parentheses and the results of the logic operation are linked according to an AND logic operation. The *#Display.oneFan_2* tag is set if only one of the fans is running.

9.2.6 Negate result of logic operation in SCL

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. 9.6 shows the negation of binary functions. The functions are present in parentheses since they have a lower processing priority than NOT. The result of the binary function is generated first and subsequently negated.

```
//NAND function - at least one fan is not running
#Display.nand := NOT (#Fan1.works AND #Fan2.works);
```

```
//NOR function - no fan is running
#Display.nor := NOT (#Fan1.works OR #Fan2.works);
```

```
//Inclusive OR function - neither of the fans or both fans are running
#Display.nxor := NOT (#Fan1.works XOR #Fan2.works);
```

Fig. 9.6 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 the value assignment as memory function. Retentive setting and resetting statements and edge evaluations can be emulated.

9.3.1 Value assignment of a binary tag

The value assignment directly assigns the current result of logic operation to the binary tag named in front of the operator. The response of the assignment is described in Chapter 12.2.2 “Simple and negating coil, assignment” on page 511.

An example of a (binary) value assignment is shown in Fig. 9.7. ① Here the *#Fan1.drive* tag is set to signal state “1” if the logic operation is fulfilled or to signal state “0” if the logic operation is not fulfilled.

9.3.2 Setting and resetting in SCL

The retentive setting and resetting of a binary tag (see Chapter 12.2 “Memory functions” on page 510) can be emulated, for example, with a simple IF branch.

In Fig. 9.7 ②, the *#Fan2.drive* tag is set to signal state “1” if the *#Fan2.start* tag has signal state “1”. If *#Fan2.start* has signal state “0”, *#Fan2.drive* is not influenced. ③ Resetting of *#Fan2.drive* is carried out in a similar manner: If the *#Fan2.stop* OR NOT *#Fan2.fault* expression is fulfilled, *#Fan2.drive* is set to signal state “0”. An expression which is not fulfilled does not influence *#Fan2.drive*. Resetting is programmed following setting and is therefore “dominant”. If both conditions are fulfilled, *#Fan2.drive* is reset or remains reset.

```
//Assignment of value to a binary tag                                  ①
#Fan1.drive := (#Fan1.start OR #Fan1.drive)
   AND NOT #Fan1.stop AND #Fan1.fault;
```

```
//Set tag with RLO = "1"                                               ②
IF #Fan2.start THEN #Fan2.drive := TRUE; END_IF;
```

```
//Reset tag with RLO = "1"                                             ③
IF #Fan2.stop OR NOT #Fan2.fault
   THEN #Fan2.drive := FALSE; END_IF;
```

Fig. 9.7 Assigning, setting, and resetting with SCL

9.3.3 Edge evaluation in SCL

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.

Fig. 9.8 shows one example each with a rising (positive) edge and a falling (negative) edge.

① With the first edge evaluation, a pulse flag (*#Alarm.pulse_pos*) is generated which, with a positive edge, has signal state “1” for the duration of one program cycle. This pulse flag can be used in the user program to carry out actions; in the example the *#Alarm.memory* tag is set to TRUE. Following pulse generation, the edge trigger flag must be updated.

② The second edge evaluation is implemented using an IF statement. If a negative edge is detected, *#Alarm.memory* is reset to FALSE. This is followed by updating of the edge trigger flag.

```
//Set alarm memory with positive signal edge                          ①
#Alarm.pulse_pos := #Alarm.bit AND NOT #Alarm.edge_pos;
#Alarm.edge_pos := #Alarm.bit;
IF #Alarm.pulse_pos THEN #Alarm.memory := TRUE; END_IF;
```

```
//Reset alarm memory with negative signal edge                        ②
IF NOT #Alarm.ack AND #Alarm.edge_neg
   THEN #Alarm.memory := FALSE; END_IF;
#Alarm.edge_neg := #Alarm.ack;
```

Fig. 9.8 Examples of edge evaluation with SCL

9.4 Programming timer and counter functions with SCL

9.4.1 SIMATIC timer functions in SCL

The SIMATIC timer functions are an operand area in the CPU's system memory and their number is limited. SCL handles a SIMATIC timer function like a function with function value. Table 9.3 shows the parameters possible in association with a function call of a SIMATIC timer. The time response of a SIMATIC timer function is described in detail in Chapter 12.4 “SIMATIC timer functions” on page 524.

For programming, enter the tag for the function value and the assignment operator in a line. Drag the function call with the mouse from the program elements catalog under *Basic instructions* > *Timer operations* into the input line. Then replace the dummy values by the actual parameters in the function call. Delete non-required parameters including their name.

In Fig. 9.9, the time *“Fan3.on_delay”* is started as an ON delay by the positive edge of *#Fan3.start*. Following expiry of the duration, the timer function *“Fan3.off_delay”* is started with the duration present as a value in the *#Follow-up_time* tag. The status of the timer function *“Fan.off_delay”* simultaneously has signal state “1” so that fan 3 is switched on following the ON delay. Once the start signal *#Fan3.start* has signal state “0”, fan 3 continues to run for the follow-up time and is then switched off.

Table 9.3 Call of SIMATIC timer functions with SCL

Timer function	Parameter	Data type	Description
	Function value	S5TIME	Current time value
S_PULSE S_PEXT S_ODT S_ODTS S_OFFDT			Start timer as pulse Start timer as extended pulse Start timer as ON delay Start timer as retentive ON delay Start timer as OFF delay
	T_NO S TV R BI Q	TIMER BOOL S5TIME BOOL WORD BOOL	Time operand (T) Start input Default time value Reset input Current integer-coded time value Binary status of timer function

```
//Switch fan on and off with delay
#temp_S5Time := S_ODT(T_NO := "Lüfter3.on_delay",
                      S    := #Fan3.start,
                      TV   := S5T#3s,
                      Q    => #temp_bool);
#temp_S5Time := S_OFFDT(T_NO := "Fan3.off_delay",
                        S    := #temp_bool,
                        TV   := #Follow-up_time,
                        Q    => #Fan3.drive);
```

Fig. 9.9 Example of application of SIMATIC timer functions

9.4.2 SIMATIC counter functions in SCL

The SIMATIC counter functions are an operand area in the CPU's system memory and their number is limited. SCL handles a SIMATIC counter function like a function with function value. Table 9.4 shows the parameters possible in association with a function call of a SIMATIC counter. The counter response is described in detail in Chapter 12.6 “SIMATIC counter functions” on page 545.

For programming, enter the tag for the function value and the assignment operator in a line. Drag the function call with the mouse from the program elements catalog under *Basic instructions* > *Counter operations* into the input line. Then replace the dummy values by the actual parameters in the function call. Delete non-required parameters including their name.

Fig. 9.10 shows the counting of workpieces up to a specific quantity. The counter *#Parts_counter* is set by the *#Quantity_set* tag to a start value of 120. Each positive edge at the *#Workpiece_identified* tag decrements the count value by one unit. If a value of zero is reached – the counter status is then “0” – *#Quantity_reached* is set.

Table 9.4 Call of SIMATIC counter functions with SCL

Counter function	Parameter	Data type	Description
	Function value	WORD	Current count value
S_CU S_CD S_CUD			Up counter Down counter Up/down counter
	C_NO S PV R BI Q	COUNTER BOOL WORD BOOL WORD BOOL	Counter operand (C) Set input Default count value Reset input Current integer-coded count value Binary status of counter function

```
//Simple parts counter
#temp_word := S_CD(C_NO := "Parts_counter",
                   CD   := #Workpiece_identified,
                   S    := #Quantity_set,
                   PV   := 16#0120,
                   Q    => #temp_bool);
#Quantity_reached := NOT #temp_bool;
```

Fig. 9.10 Example of application of a SIMATIC counter function in SCL

9.4.3 IEC timer functions in SCL

An IEC timer function is available as pulse time (TP), as ON delay (TON), as OFF delay (TOF), and as accumulating ON delay (TONR). A detailed description of the timer response is provided in Chapter 12.5 “IEC timer functions” on page 539.

For programming, drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions* > *Timer operations* into a line on the working area. When positioning, you select either as single instance or as local instance (multi-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.

PRESET_TIMER loads an IEC timer function with a duration. RESET_TIMER resets an IEC timer function.

Fig. 9.11 shows the IEC timer function *#Alarm_delay*, which saves its data as local instance in the instance data block of the calling function block. If the *#Measurement_too_high* tag has signal state “1” for longer than 10 s, *#Alarm_too_high* is set.

```
//Alarm delay
#Alarm_delay(IN := #Measurement_too_high,
             PT := T#10s,
             Q  => #Alarm_too_high);
...
PRESET_TIMER(PT := T#5s,
          TIMER := #Alarm_delay);
...
RESET_TIMER(#Alarm_delay);
```

Fig. 9.11 Example of IEC timer function with SCL

9.4.4 IEC counter functions in SCL

An IEC counter function is available as up counter (CTU), down counter (CTD), or up/down counter (CTUD). A detailed description of the counter response is provided in Chapter 12.7 "IEC counter functions" on page 553.

For programming, drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions* > *Counter operations* into a line on the working area. When positioning, you select either as single instance or as local instance (multi-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 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. 9.12 shows the IEC 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 with the name of the instance and the component name, for example *#Lock_counter.CV*. The example shows the passages through a lock, either forward or backward.

```
//Simple lock counter
#temp_bool1 := #Light_barrier1 AND NOT #"Light_barrier1.edge";
#"Light_barrier1.edge" := #Light_barrier1;
#temp_bool2 := #Light_barrier2 AND NOT #"Light_barrier2.edge";
#"Light_barrier2.edge" := #Light_barrier2;
#Lock_counter(CU := #Light_barrier2 AND #temp_bool1,
   CD := #Light_barrier1 AND #temp_bool2,
   R := #Acknowledge, LD := FALSE, PV := 0);
```

Fig. 9.12 Example of IEC counter function with SCL

9.5 Programming digital functions with SCL

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 (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

A detailed description of the transfer functions is provided in Chapter 13.2 "Transfer functions" on page 559.

The "simple" transfer function corresponds with SCL to the value assignment. MOVE_BLK and UMOVE_BLK transfer individual components from one ARRAY tag to another. BLKMOV and UBLKMOV transfer individual tags or absolutely addressed data areas. FILL_BLK and UFILL_BLK fill components of an ARRAY tag with a value. FILL fills a tag or an absolutely addressed data area with a value. SWAP swaps the order of the bytes in a tag.

You can find the transfer functions in the program elements catalog under *Basic instructions* > *Move operations*.

Example of a value assignment: The value of the *#Alarms* tag in data block "*Data.SCL*" is assigned to the *"Alarm_bits"* tag in the memory area.

```
"Alarm_bits" := "Data.SCL".Alarms;
```

9.5.2 Comparison functions in SCL

A comparison function compares the values of two digital tags and delivers a binary comparison result. SCL implements the comparison function using a comparison operator. The comparison result 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 13.3 "Comparison functions" on page 570. Table 9.5 shows the comparison operators available with SCL.

Table 9.5 Comparison functions with SCL

Operator	Description	Approved tags
= <> < <= > >=	Compare for equal Compare for unequal Compare for greater than Compare for greater than-equal Compare for less than Compare for less than-equal	Comparison of fixed-point and floating-point numbers, of durations, of date and time, and of strings
= <>	Compare for equal Compare for unequal	Comparison of bit sequences

Two comparison functions are programmed in Fig. 9.13. In the first comparison, the *#Measurement_temperature* tag is compared with *"Lower_limit",* in the second comparison with *"Upper_limit"*. The result of the two comparisons is linked according to AND and saved in the *#Measurement_in_range* tag. *"Lower_limit"* and *"Upper_limit"* are created as symbolically addressed user constants.

```
#Measured_value_in_range :=
   (#Measurement_temperature >= "Lower_limit") AND
   (#Measurement_temperature <= "Upper_limit");
```

Fig. 9.13 Example of comparison expressions with SCL

9.5.3 Arithmetic functions in SCL

The arithmetic functions for numerical values implement the basic arithmetical operations addition, subtraction, multiplication, and division. SCL uses an arithmetic operator for this. A detailed description of these arithmetic functions is provided in Chapter 13.4 "Arithmetic functions" on page 574. Table 9.6 shows the arithmetic operators available with SCL and the allowed data types.

If an arithmetic function is used for numbers with different data types, the data type of the result is determined as follows:

- If there are two fixed-point numbers with sign, the result receives the larger data type (example: INT + DINT = DINT).
- If there are two fixed-point numbers without sign, the result receives the larger data type (example: USINT + UDINT = UDINT).
- If one fixed-point number has a sign and the other does not, the result receives the next larger data type with a sign, which covers the fixed-point number without sign (example: SINT + USINT = INT).
- If there is a fixed-point number and a floating-point number, the result receives the data type of the floating-point number (example: INT + REAL = REAL).
- When there are two floating-point numbers of different lengths, the result receives the larger data type (example: REAL + LREAL = LREAL).

The permitted data types depend on the block attribute *IEC check*. Wherever possible, implicit data type conversion is used (see also Chapter 4.5.2 "Implicit data type conversion" on page 108).

In Fig. 9.14, the upper limit of a measured value is monitored. A hysteresis is introduced to ensure that the *#Measurement_too_high* and *#Measurement_too_low* alarms do not "pulsate" when the measurement changes rapidly around the upper or lower limit. The alarms 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.

Table 9.6 Arithmetic operators with SCL

		Data type		
Operator	**Description**	**1st operand**	**2nd operand**	**Result**
**	Power	Fixed point, floating point	Fixed point, floating point	Fixed point, floating point
*	Multiplication	Fixed point, floating point TIME, LTIME	Fixed point, floating point Fixed point	Fixed point, floating point TIME, LTIME
/	Division	Fixed point, floating point TIME, LTIME	Fixed point, floating point Fixed point	Fixed point, floating point TIME, LTIME
MOD	Division with remainder as result	Fixed point	Fixed point	Fixed point
+	Addition	Fixed point, floating point TIME LTIME TOD LTOD DATE DT LDT DTL	Fixed point, floating point TIME, DINT TIME, LTIME, LINT TIME, DINT TIME, LTIME, LINT TOD, LTOD TIME TIME, LTIME TIME, LTME	Fixed point, floating point TIME LTIME TOD LTOD DTL DT LDT DTL
–	Subtraction	Fixed point, floating point TIME LTIME TOD LTOD DATE DT LDT DTL DTL	Fixed point, floating point TIME, DINT TIME, LTIME, LINT TIME, DINT TIME, LTIME, LINT DATE TIME TIME, LTIME TIME, LTME DTL	Fixed point, floating point TIME LTIME TOD LTOD LTIME DT LDT DTL LTIME

```
IF #Measurement_temperature >= "Upper_limit"
   THEN #Measurement_too_high := TRUE; END_IF;
IF #Measurement_temperature <= "Upper_limit" - "Hysteresis"
   THEN #Measurement_too_high := FALSE; END_IF;
```

```
IF #Measurement_temperature <= "Lower_limit"
   THEN #Measurement_too_low:= TRUE; END_IF;
IF #Measurement_temperature >= "Lower_limit" + "Hysteresis"
   THEN #Measurement_too_low:= FALSE; END_IF;
```

Fig. 9.14 Example of arithmetic expressions with SCL

9.5.4 Math functions in SCL

The math functions comprise trigonometric functions, exponential functions, and logarithmic functions. The math functions process floating-point numbers. If the input tag has a different data type, it is converted during the implicit data type conversion (see also Chapter 4.5.2 “Implicit data type conversion” on page 108). The program elements catalog contains the math functions under *Basic instructions > Math functions*. A detailed description of these math functions is provided in Chapter 13.5 “Math functions” on page 578. Table 9.7 shows the math 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 natural logarithm (to base e)

Fig. 9.15 shows the calculation of reactive power using the SIN function, calculation of the volume of a sphere, the solution of a quadratic equation, and calculation of an arithmetic mean value.

```
#Reactive_power := #Voltage * #Current * SIN(#phi);
#Volume := 4/3 * "pi" * #Radius**3;
#Solution_1 := -#p/2 + SQRT(SQR(#p/2) - #q);
#Mean_value := (#Motor[1].power + #Motor[2].power)/2;
```

Fig. 9.15 Example of math functions with SCL

9.5.5 Conversion functions in SCL

The conversion functions convert the data formats of tags and expressions. A detailed description of the conversion functions is provided in Chapter 13.6 “Conversion functions” on page 586. The program elements catalog contains CONVERT, ROUND, CEIL, FLOOR, TRUNC under *Basic instructions > Conversion operations*. The conversion function T_CONV can be found under *Extended instructions > Date and* time-of-day, the conversion functions S_CONV, CHARS_TO_STRG, STRG_TO_CHARS, ATH, and HTA under *Extended instructions > String + Char*.

When inserting the CONVERT, T_CONV, or S_CONV functions, you select the data types involved in the conversion in a dialog box (Fig. 9.16). As a result of the dialog,

Table 9.8 Conversion functions with SCL

Function	Description, remark
CONVERT T_CONV S_CONV	Dialog boxes for data type conversion with explicit conversion functions
	Data type conversion of a floating-point number into a fixed-point number or floating-point number
ROUND	With rounding to the next integer
CEIL	With rounding to the next highest integer
FLOOR	With rounding to the next lowest integer
TRUNC	Without rounding
CHARS_TO_STRG STRG_TO_CHARS	Conversion of an ARRAY tag with CHAR or BYTE components into a string and vice versa.
ATH HTA	Conversion of a BYTE sequence with hexadecimal coding into a CHAR sequence with ASCII coding and vice versa.

the program editor inserts the explicit conversion function <target tag> := <conversion function> (<source tag>), e.g.

```
#var_target := INT_TO_REAL(#var_source);
```

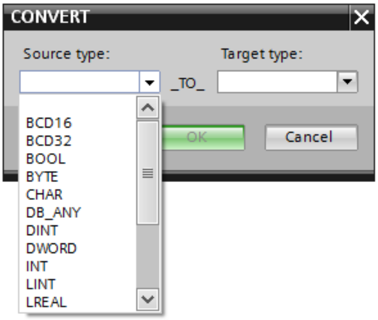


Fig. 9.16 Selection of data types with the CONVERT function

ROUND, CEIL, FLOOR, and TRUNC convert a fractional number in floating-point format into an integer in fixed-point or floating-point format. ROUND converts the fractional number to the next integer. If the result lies between an even number and an uneven number, the even number is output (example: ROUND(0.5) = 0, ROUND(1.5) = 2). CEIL rounds to the next highest integer. FLOOR rounds to the next lowest integer. TRUNC “cuts off” the decimal places and only displays the integer portion.

The function value of ROUND, CEIL, FLOOR and TRUNC has the default data type DINT. If a different data type is to be assigned to the function value, attach the data type to the function with an underscore. Example:

```
#var_sint := TRUNC_SINT(#var_real);
```

If the permissible numerical range is left during a conversion, the ENO tag is set to FALSE and the result of the conversion is invalid.

Fig. 9.17 shows an example of nested conversion functions ①. A value present in data format REAL is first mapped to the data format DINT and then converted into the 7-decade BCD format.

```
#Measurement_display :=                                              ①
   DINT_TO_BCD32(REAL_TO_DINT(#Reactive_power));
#Program1 := DB_ANY_TO_UINT(#Data_block) > 10;                       ②
```

Fig. 9.17 Example of conversion functions with SCL

Conversion of a data block number to UINT

A tag with the data type DB_ANY has the number of the data block as its content. This data type can be converted to the data type UINT, which allows you to continue working with the DB number, for example for indirect addressing.

▷ DB_ANY_TO_UINT provides the DB number as UINT tag.

▷ UINT_TO_DB_ANY converts a UINT tag into a DB number.

Fig. 9.17 shows an example ②. The *#Data_block* tag is typically a block parameter with the data type DB_ANY, which as input parameter allows the transfer of a data block to the called block. In the example, if the number of this data block is greater than 10, the tag *#Program1* is set to “1”; otherwise it is reset.

Conversion of a data block number to VARIANT

A tag with the data type DB_ANY has the number of the data block as its content. This data type can be converted to data type VARIANT. Conversely, a VARIANT pointer to a data block can be converted to data type DB_ANY.

▷ DB_ANY_TO_VARIANT converts the DB number into a VARIANT pointer.

▷ VARIANT_TO_DB_ANY converts a VARIANT pointer into a DB number.

The syntax is as follows:

```
#var_DB_ANY := VARIANT_TO_DB_ANY(IN := #var_VARIANT, ERR => #var_INT);
#var_VARIANT := DB_ANY_TO_VARIANT(IN := #var_DB_ANY, ERR => #var_INT);
```

The conversion using DB_ANY_TO_VARIANT or VARIANT_TO_DB_ANY can only be carried out with type data blocks or ARRAY data blocks.

If an error occurs while the conversion is being carried out, the error is specified on the parameter ERR. In the event of an error, the data block number zero or a zero pointer is returned.

9.5.6 Shift functions in SCL

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 13.7 “Shift functions” on page 603. The program elements catalog contains the shift functions under *Basic instructions > Shift and rotate*. Table 9.9 shows the shift functions available with SCL.

Table 9.9 Shift functions with SCL

Function	Description	Data types IN	Data type N
SHR (IN, N) SHL (IN, N)	Shift to right Shift to left	Bit sequences, fixed point	Fixed point
ROR (IN, N) ROL (IN, N)	Rotate to right Rotate to left	Bit sequences, fixed point	Fixed point

By default the function value of a shift function has the data type that the input tag has. If a different data type is to be assigned to the function value, attach the data type to the function with an underscore.

Example: #var_dword := ROL_DWORD(IN1 := #var_word, N := #var_uint);

In Fig. 9.18, the three decades of two numbers present in BCD format of a SIMATIC counter are joined. The more significant component *#Quantity_high* is shifted to the left by three decades (12 bits) and linked to the less significant component *#Quantity_low* according to an OR logic operation. In the result *#Quantity_display*, the two times three decades are then present as a 6-decade BCD number.

```
#Quantity_display :=
   SHL(IN := #Quantity_high, N := 12) OR #Quantity_low;
```

Fig. 9.18 Example of shift functions with SCL

9.5.7 Word logic operations, logic expression in SCL

The word logic operations apply the binary operations AND, OR, and XOR to each bit of a digital tag. A word logic operation is implemented with a logic expression in SCL. A detailed description is provided in Chapter 13.8.1 “Word logic operations” on page 607. Table 9.10 shows the word logic operations available with SCL.

Table 9.10 Word logic operations with SCL

Operator	Function	Data types of the tags
AND, & OR XOR	AND logic operation OR logic operation Exclusive OR logic operation	Bit sequences, fixed point
NOT	Negation	Bit sequences, fixed point

Fig. 9.19 shows how you can program 32 edge evaluations simultaneously for rising and falling edges. The alarm bits are collected in a doubleword *Alarms*, which is present in data block *"Data.SCL"*. The edge trigger flags *Alarms_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 *Alarms* and *Alarms_EM*, in other words: The associated alarm bit has changed. In order to obtain the positive signal edges, the changes are linked to the alarms by an AND logic operation. The bit for a rising signal edge is set wherever the alarm has a "1" and the change has a "1". This corresponds to the pulse flag of the edge evaluation. If you do the same with the negated alarm bits – the alarm 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 alarms.

```
#Alarm_changes :=
   "Data.SCL".Alarms XOR "Data.SCL".Alarms_EM;
"Data.SCL".Alarms_pos :=
   #Alarm_changes AND "Data.SCL".Alarms;
"Data.SCL".Alarms_neg :=
   #Alarm_changes AND NOT "Data.SCL".Alarms;
"Data.SCL".Alarms_EM := "Data.SCL".Alarms;
```

Fig. 9.19 Example of word logic operations with SCL

9.5.8 Functions for strings in SCL

Strings are tags with the data type STRING. With the functions for strings, parts of a string can be extracted, inserted, replaced or deleted, two strings can be combined, and the length of a string or the position of a character in a string can be determined.

A detailed description of these functions is provided in Chapter 13.9 "Processing of strings (data type STRING)" on page 615. You can find the functions for the processing of strings in the program elements catalog under *Extended instructions > String + Char*. Table 9.11 shows an overview of the available functions.

Table 9.11 Functions for the processing of strings

Function	Meaning, remark	Function	Meaning, remark
LEN	Outputs the length of a string.	DELETE	Deletes part of a string.
FIND	Finds characters in a string.	INSERT	Inserts characters into a string.
LEFT	Outputs the left part of the string.	REPLACE	Replaces characters in a string.
RIGHT	Outputs the right part of the string.	CONCAT	Combines two strings together.
MID	Outputs the middle part of the string.		

The example in Fig. 9.20 shows the replacement of one part of a string by another. In the input tag IN1, REPLACE replaces from position P the number of characters L with the string to be inserted IN2 and outputs the new string as function value.

```
var3_string := REPLACE(IN1 := #var1_string,
                       IN2 := #var2_string,
                       L   := #var1_int,
                       P   := #var2_int);
```

Fig. 9.20 Example of string processing with SCL

9.6 Program control with SCL

You can influence execution of the user program by means of the program control functions. You can recognize errors in program execution by using the ENO tag, the control statements permit you to implement 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 or previous error *);
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 the 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 is permissible for function blocks (FB) and functions (FC). 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 function 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 as with LAD or FBD, in other words the “series connection” of block calls, you can program 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. 9.21 provides a summary of how the enable output ENO and the ENO tag are controlled with a block call.

Fig. 9.21 Schematic for setting of enable output ENO and the ENO tag

<table>
<tr><td colspan="5">Is EN used?</td></tr>
<tr><td colspan="3">YES</td><td colspan="2">NO</td></tr>
<tr><td colspan="3">Is EN = TRUE?</td><td colspan="2" rowspan="3">Block/function being processed</td></tr>
<tr><td colspan="2">YES</td><td>NO</td></tr>
<tr><td colspan="2">Block/function being processed</td><td rowspan="3">Block/function not being processed</td></tr>
<tr><td colspan="2">Has an error occurred?</td><td colspan="2">Has an error occurred?</td></tr>
<tr><td>YES</td><td>NO</td><td>YES</td><td>NO</td></tr>
<tr><td>Tag at the ENO output is set to FALSE</td><td>Tag at the ENO output is set to TRUE</td><td>Tag at the ENO output is set to FALSE</td><td>Tag at the ENO output is set to FALSE</td><td>Tag at the ENO output is set to TRUE</td></tr>
<tr><td>"Block-local" ENO tag is set to FALSE</td><td>"Block-local" ENO tag remains unchanged</td><td>"Block-local" ENO tag remains unchanged</td><td>"Block-local" ENO tag is set to FALSE</td><td>"Block-local" ENO tag remains unchanged</td></tr>
</table>

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.

IF statement

The IF statement processes a statement block depending on a Boolean value (Fig. 9.22).

Example in Fig. 9.23: 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.

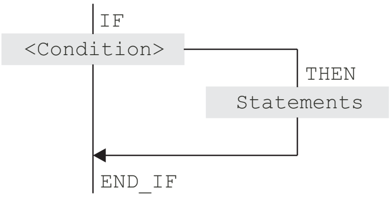
Control statement IF

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;
```

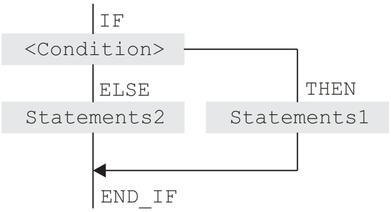
If <Condition> has the value TRUE, the statement block following THEN is processed.
If <Condition> has the value FALSE, processing of the program is continued following END_IF.



IF branch with ELSE

```
IF <Condition>
   THEN <Statements1>;
   ELSE <Statements2>;
END_IF;
```

If <Condition> has the value TRUE, the statement block following THEN is processed.
If <Condition> has the value FALSE, the statement block following ELSE is then 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 statement can be inserted cascaded: An ELSIF scan can again follow ELSIF ... THEN.
ELSE and the subsequent statements can also be omitted.

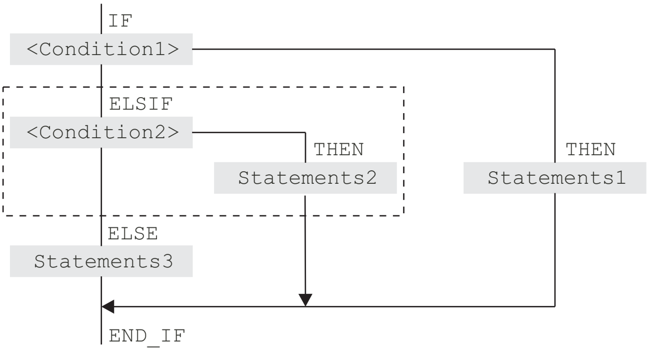


Fig. 9.22 Principle of operation of the IF branch

```
#greater_than := FALSE; #less_than := FALSE; #equal_to := FALSE;
IF #Actual_value > #Setpoint
   THEN #greater_than := TRUE;
      ELSIF #Actual_value < #Setpoint
      THEN #less_than := TRUE;
   ELSE #equal_to := TRUE;
END_IF;
```

Fig. 9.23 Example of the IF statement

CASE statement

You can use the CASE statement to process one or more sequences of statements depending on an INT value (Fig. 9.24).

Control statement CASE

The control statement CASE processes a program section <Statements> depending on whether an integer value <Selection > is within a <Range>. <Range> can be a constant, a fixed range, or a list of constants and fixed ranges.

```
CASE <Selection> OF
   <Range1> : <Statements1>;
   <Range2> : <Statements2>;
   ...
   ELSE       : <Statements0>;
END_CASE;
```

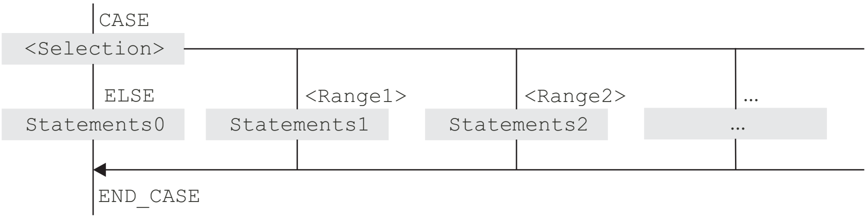


Fig. 9.24 Principle of operation of the CASE branch

Selection is an operand or expression with data type INT. If *Selection* has the value of *Range1*, the *Statements1* are processed and then processing of the program is continued following 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 INT constants.

Various expressions are possible for a component in the list of values:

- A single INT number
- A range of INT numbers (e.g. 15..20)
- A list of INT numbers and INT numerical ranges (e.g. 21,25,30..33)

Each value must only be present once in the list of values.

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. 9.25: A value is assigned to the *#Error_number* tag depending on the assignment of the *#ID* tag.

```
CASE #ID OF
0     : #Error_number := 0;
1,3,5 : #Error_number := #ID + 128;
6..10 : #Error_number := #ID;
ELSE    #Error_number := 16#7F;
END_CASE;
```

Fig. 9.25 Example of the CASE statement

FOR statement

Using the FOR statement, a program loop is repeatedly processed as long as a control tag is within a defined range of values (Fig. 9.26).

Control statement FOR

The control statement FOR processes a program section <Statements> for as long as a #Control tag is within a range of values. The range of values is defined by a <Start_value>, an <Increment>, and an <End_value>.

```
FOR #Control_tag := <Start_value> TO <End_value> BY <Increment>
   DO <Statements>;
END_FOR;
```

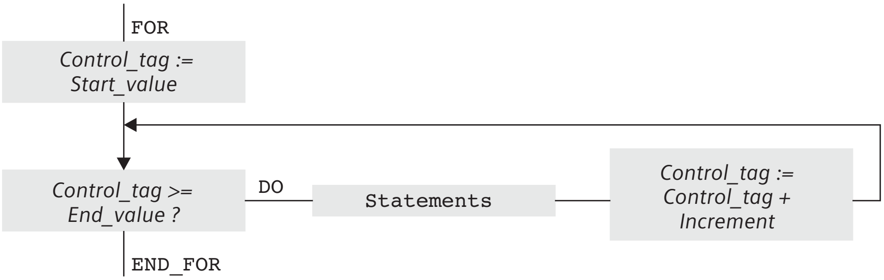

FOR
Control_tag := Start_value
Control_tag >= End_value ?
DO
Statements
Control_tag := Control_tag + Increment
END_FOR

Fig. 9.26 Principle of operation of the FOR loop

A <Start_value> is assigned to the *#Control_tag* in the start assignment. You define the control tag yourself; it must be a tag with data type INT or DINT. <Start_value> is any INT or DINT expression, as are <End_value> and <Increment>.

Control_tag if 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, *#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 *#Control_tag* is outside the range of start value and end value, program execution is continued following END_FOR.

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. 9.27: 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 adopted.

```
#MaxValue := 0;
FOR #Index := 1 TO 16 DO
  IF #MaxValue < #Current[#Index]
    THEN #MaxValue := #Current[#Index];
  END_IF;
END_FOR;
```

Fig. 9.27 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. 9.28).

<Condition> is an operand or expression with data type BOOL. The statements following DO are repeatedly processed for as long as <Condition> is TRUE.

<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

Control statement WHILE

The control statement WHILE processes a program section <Statements> for as long as a <Condition> is fulfilled. <Condition> is a binary tag or an expression with binary result.

```
WHILE <Condition>
   DO <Statements>;
END_WHILE;
```

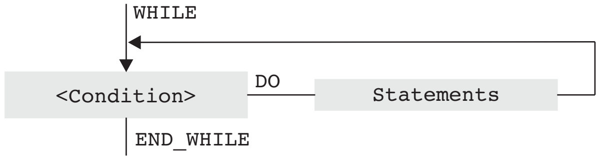


Fig. 9.28 Principle of operation of the WHILE loop

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. 9.29: The data block %DB300 is searched word-by-word from the data word DBW16 for the bit pattern 16#FFFF. For every loop cycle, the 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.

```
#Offset := 0;
WHILE PEEK_WORD(area := 16#84,
          dbNumber    := 300,
          byteOffset := 16 + #Offset) <> 16#FFFF DO
  #Offset := #Offset + 2;
END_WHILE;
#Quantity := #Offset/2 + 1;
```

Fig. 9.29 Example of the WHILE statement

REPEAT statement

The REPEAT statement is used to repeatedly process a program loop for as long as an abort condition is not fulfilled (Fig. 9.30).

<Condition> is an operand or expression with data type BOOL. The statements following REPEAT are repeatedly processed for as long as <Condition> is FALSE.

Control statement REPEAT

The control statement REPEAT processes a program section <Statements> for as long as a <Condition> is not fulfilled. <Condition> is a binary tag or an expression with binary result.

```
REPEAT
   <Statements>;
   UNTIL <Condition>
END_REPEAT;
```

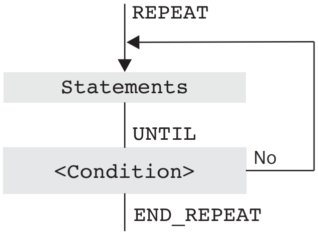


Fig. 9.30 Principle of operation of the REPEAT control statement

<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. 9.31: In the data block *"Data.SCL"*, the *#Measurement* array with INT components is searched. The search ends as soon as a component has a value less than 0. The index of the found array component is then in the control tag *#k*.

```
#k := 0;
REPEAT
  #k := #k + 1;
  UNTIL "Data.SCL".Measurement[#k] < 0
END_REPEAT;
```

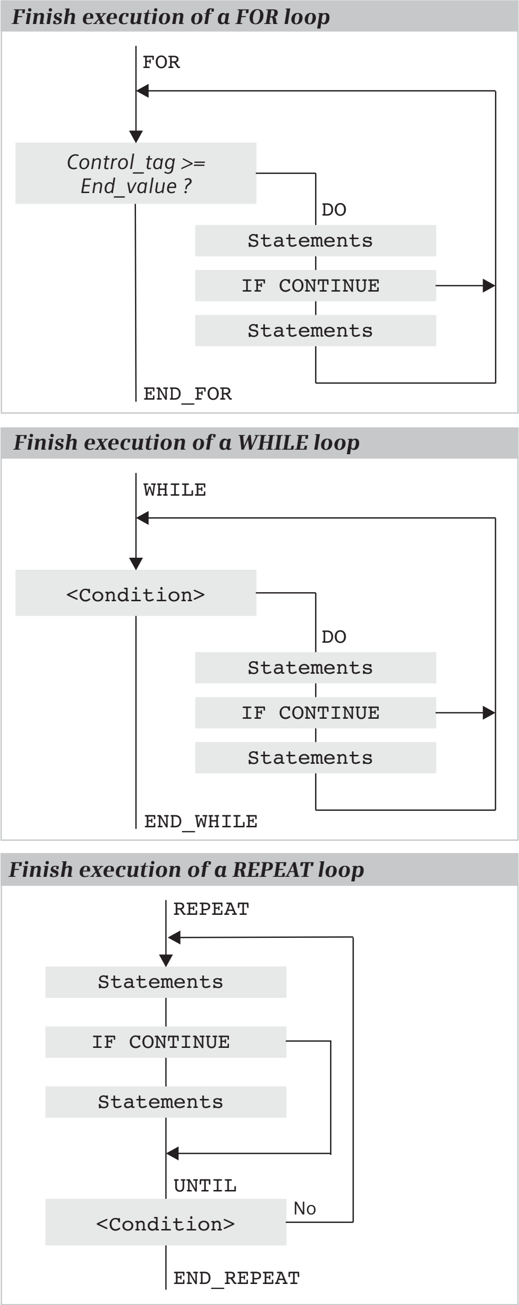
Fig. 9.31 Example of the REPEAT statement

CONTINUE statement

CONTINUE finishes the current program execution in a FOR, WHILE or REPEAT loop (Fig. 9.32).

Control statement CONTINUE

The control statement CONTINUE finishes the current execution of a FOR, WHILE or REPEAT program loop. CONTINUE can be positioned anywhere in the statement part of the loop.



Finish execution of a FOR loop

Finish execution of a WHILE loop

Finish execution of a REPEAT loop

CONTINUE usually depends on a condition. This condition has the data type BOOL and can be a tag or an expression.

For the statement sequence:

```
IF <Condition>
   THEN CONTINUE;
END_IF;
```

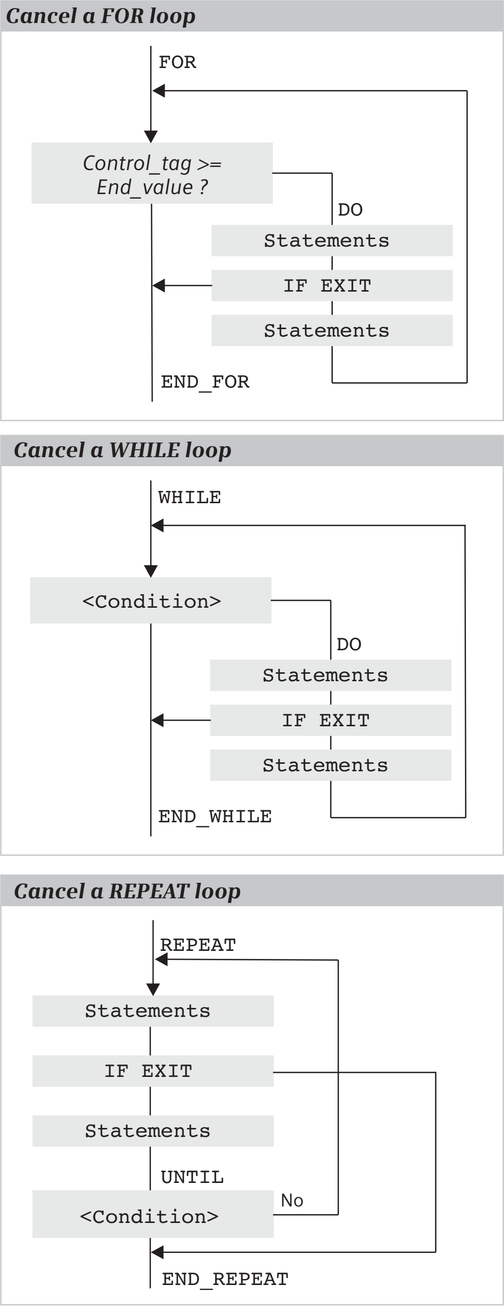
the following is present in the representations:

```
IF CONTINUE
```

Fig. 9.32 Principle of operation of the CONTINUE control statement

Control statement EXIT

The control statement EXIT finishes a FOR, WHILE or REPEAT program loop.
EXIT can be positioned anywhere in the statement part of the loop.



EXIT usually depends on a condition. This condition has the data type BOOL and can be a tag or an expression.

For the statement sequence:

```
IF <Condition>
   THEN EXIT;
END_IF;
```

the following is present in the representations:

```
IF EXIT
```

Fig. 9.33 Principle of operation of the EXIT control 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. 9.34: 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 control tag *#k* is less than *#BitBegin*, the program loop begins again with *#k* increased by +1. If in the last byte (in the last cycle of the external FOR loop), the control tag *#k* is greater than *#BitEnd*, the execution of the internal FOR loop ends.

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. 9.33).

EXIT results in leaving of the loop which directly surrounds the EXIT statement. An example is shown in Fig. 9.34.

```
FOR #i := 0 TO #Quantity - 1 DO
  FOR #k := 0 TO 7 DO
    IF (#i = 0) AND (#k < #BitBegin) THEN CONTINUE; END_IF;
    IF (#i = #Quantity - 1) AND (#k > #BitEnd) THEN EXIT; END_IF;
    POKE_BOOL (area := 16#83,
               dbNumber := 0,
               byteOffset := #ByteBegin + #i,
               bitOffset := #k,
               value := FALSE);
  END_FOR;
END_FOR;
```

Fig. 9.34 Example of the CONTINUE and the EXIT statement

9.6.4 Block functions

The block functions call and terminate blocks. A detailed description of the block functions is provided in Chapter 14.2 "Calling of code blocks" on page 631. Fig. 9.35 shows an example of the block functions with SCL.

Terminate block with RETURN

The RETURN statement terminates processing in the current block.

```
//Block call FC without function value                                     ①
"Adder.SCL"(Number_1 := #Measurements[1],
            Number_2 := #Measurements[2],
            Number_3 := #Measurements[3],
            Total    => #Results[1]);
```

```
//Block call FC with function value                                        ②
#Results[2] := "Adder2.SCL"(Number_1 := #Measurements[1],
                                  Number_2 := #Measurements[2],
                                  Number_3 := #Measurements[3]);
```

```
//Block call FB as single instance                                         ③
"DB_Adder"(Value1 := #Interval[1],
           Value2 := #Interval[2],
           Value3 := #Interval[3],
           Result => #Position[1]);
```

```
//Block call FB as local instance                                          ④
#Result(Value1 := #Interval[1],
        Value2 := #Interval[2],
        Value3 := #Interval[3],
        Result => #Position[1]);
```

```
//Example of supplying parameters with values                              ⑤
#Results[4] := Results[3] +
      LIMIT(MN := #Lower_limit + #Hysteresis,
            IN := REAL_TO_INT(#Result.Result),
            MX := #Upper_limit);
```

Fig. 9.35 Examples of block functions with SCL

The program elements catalog contains RETURN under *Basic instructions* > *Program control operations*.

Example: The block is left if the ENO tag signals an error (is then FALSE).

```
IF NOT ENO THEN RETURN;
END_IF;
```

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 with values (example ① in Fig. 9.35).

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 the example ② in Fig. 9.35, the function value of the *"Adder2.SCL"* function is assigned to the *#Results[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 the example ③ in Fig. 9.35, the *"Adder"* function block is called. The data of the call is present in the instance data block *"DB_Adder"*.

Call FB function block as local instance

When calling a function block as local instance, the instance name of the function block call 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 the example ④ in Fig. 9.35, the *"Adder"* function block is called. The data of the call is present in the instance data block of the calling function block and has the name *#Result*.

Supplying the block parameters

The input parameters on blocks and functions can be constants, tags, and expressions.

In the example ⑤ in Fig. 9.35, the *#Results[4]* tag is assigned a total made up of the *#Results[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 *#Result* from the example above this one. It is addressed by *#Result.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.

10 Statement list STL

10.1 Introduction

This chapter describes programming with a statement list. It provides examples of how the program functions are represented in a statement list. You can find a description of the individual functions, e.g. comparison functions, in Chapters 12 “Basic functions” on page 503, 13 “Digital functions” on page 558, and 14 “Program control” on page 622. Chapter 10.7 “Further STL functions” on page 441 describes functions which only exist in the statement list because they are based on a specific model of the control processor (accumulators, data block registers, address registers, status bits).

Use of the program and symbol editor, which generally applies to all programming languages, is described in Chapter 6 “Program editor” on page 247.

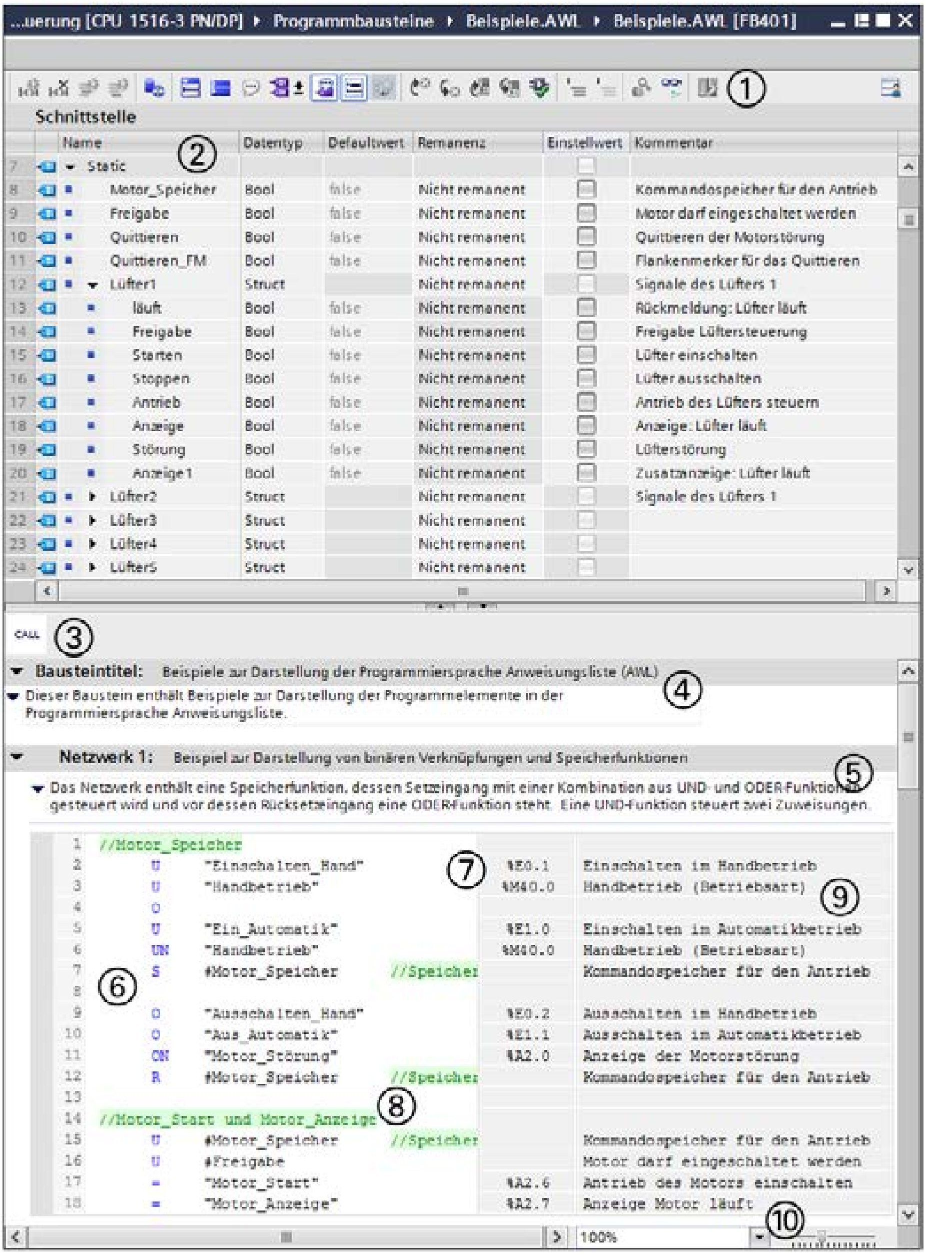
The statement list is used to program the contents of blocks (the user program). What blocks are, and how they are created, is described in Chapters 5.3.1 “Block types” on page 155 and 6.3 “Programming a code block” on page 253.

10.1.1 Programming with STL in general

You use STL to program the control function of the programmable controller – the user program (control program). The user program is organized in different types of blocks. A block can be divided into sections referred to as “networks”. Networks are not required for functioning of the user program, but they do increase the clarity.

Fig. 10.1 shows the program editor's working window. The icons in the toolbar ① can be used to set the display of the working area, e.g. the display of the network comments and additional functions such as monitoring of the program status. The interface of the block ② in the upper part of the working window lists the block parameters and local data. The favorites bar ③ can be expanded by additional program elements. It can also be hidden. Each block has a heading, the block title, and a block comment ④, which can be used to explain the function of the block. These are followed by the first network with its number, heading and comment ⑤.

The control function, i.e. the list of statements, is displayed in the working area ⑥. The program editor constructs an STL program line by line. You write the first statement in the network working area, the second statement underneath this, and so on. The tags can be displayed absolutely, symbolically, or with both addressing types ⑦. A comment ⑧ can be added to each program line. It commences with two slashes, either as line comment or as statement comment. You can insert empty lines to structure



...uerung [CPU 1516-3 PN/DP] ▸ Programmbausteine ▸ Beispiele.AWL ▸ Beispiele.AWL [FB401]

Schnittstelle

	Name	Datentyp	Defaultwert	Remanenz	Einstellwert	Kommentar
7	▾ Static				□	
8	Motor_Speicher	Bool	false	Nicht remanent	□	Kommandospeicher für den Antrieb
9	Freigabe	Bool	false	Nicht remanent	□	Motor darf eingeschaltet werden
10	Quittieren	Bool	false	Nicht remanent	□	Quittieren der Motorstörung
11	Quittieren_FM	Bool	false	Nicht remanent	□	Flankenmerker für das Quittieren
12	▾ Lüfter1	Struct		Nicht remanent	□	Signale des Lüfters 1
13	läuft	Bool	false	Nicht remanent	□	Rückmeldung: Lüfter läuft
14	Freigabe	Bool	false	Nicht remanent	□	Freigabe Lüftersteuerung
15	Starten	Bool	false	Nicht remanent	□	Lüfter einschalten
16	Stoppen	Bool	false	Nicht remanent	□	Lüfter ausschalten
17	Antrieb	Bool	false	Nicht remanent	□	Antrieb des Lüfters steuern
18	Anzeige	Bool	false	Nicht remanent	□	Anzeige: Lüfter läuft
19	Störung	Bool	false	Nicht remanent	□	Lüfterstörung
20	Anzeige1	Bool	false	Nicht remanent	□	Zusatzanzeige: Lüfter läuft
21	▸ Lüfter2	Struct		Nicht remanent	□	Signale des Lüfters 1
22	▸ Lüfter3	Struct		Nicht remanent	□	
23	▸ Lüfter4	Struct		Nicht remanent	□	
24	▸ Lüfter5	Struct		Nicht remanent	□	

Bausteintitel: Beispiele zur Darstellung der Programmiersprache Anweisungsliste (AWL)

Dieser Baustein enthält Beispiele zur Darstellung der Programmelemente in der Programmiersprache Anweisungsliste.

Netzwerk 1: Beispiel zur Darstellung von binären Verknüpfungen und Speicherfunktionen

Das Netzwerk enthält eine Speicherfunktion, dessen Setzeingang mit einer Kombination aus UND- und ODER-Funktionen gesteuert wird und vor dessen Rücksetzeingang eine ODER-Funktion steht. Eine UND-Funktion steuert zwei Zuweisungen.

```
//Motor_Speicher
      U     "Einschalten_Hand"                     %E0.1    Einschalten im Handbetrieb
      U     "Handbetrieb"                          %M40.0   Handbetrieb (Betriebsart)
      O
      U     "Ein_Automatik"                        %E1.0    Einschalten im Automatikbetrieb
      UN    "Handbetrieb"                          %M40.0   Handbetrieb (Betriebsart)
      S     #Motor_Speicher        //Speicher               Kommandospeicher für den Antrieb

      O     "Ausschalten_Hand"                     %E0.2    Ausschalten im Handbetrieb
      O     "Aus_Automatik"                        %E1.1    Ausschalten im Automatikbetrieb
      ON    "Motor_Störung"                        %A2.0    Anzeige der Motorstörung
      R     #Motor_Speicher        //Speicher               Kommandospeicher für den Antrieb

//Motor_Start und Motor_Anzeige
      U     #Motor_Speicher        //Speicher               Kommandospeicher für den Antrieb
      U     #Freigabe                                       Motor darf eingeschaltet werden
      =     "Motor_Start"                          %A2.6    Antrieb des Motors einschalten
      =     "Motor_Anzeige"                        %A2.7    Anzeige Motor läuft
```

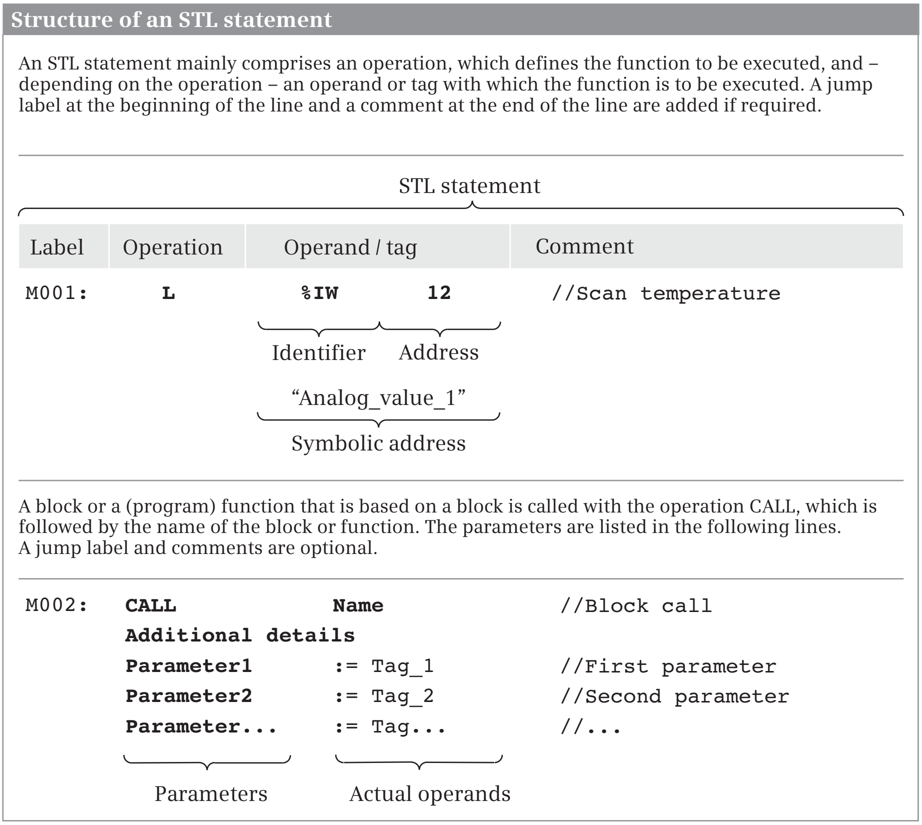
Fig. 10.1 Structure of a block with STL program

the sequence of statements. These and the comments have no effect on the control function or on the length of the compiled program. The tag information ⑨ adds the tag comment to each tag. Like the network comment, it can be hidden. The size of the font can be adjusted using the zoom setting ⑩.

In order to program, use the keyboard to enter the STL statement in a line of the input field. The program elements catalog provides you with an overview of the existing operations and functions. Dragging a statement with the mouse from the program elements catalog is of advantage with STL if you import functions with a parameter list into your program. To call self-created blocks, drag them from the *Program blocks* folder into a line.

10.1.2 Structure of an STL statement

The STL program consists of a sequence of individual STL statements. A statement is the smallest independent unit of the user program. It represents a procedural specification for the CPU. Fig. 10.2 shows the structure of an STL statement.



Structure of an STL statement

An STL statement mainly comprises an operation, which defines the function to be executed, and – depending on the operation – an operand or tag with which the function is to be executed. A jump label at the beginning of the line and a comment at the end of the line are added if required.

Label	Operation	Operand / tag	Comment
M001:	**L**	**%IW 12**	//Scan temperature

A block or a (program) function that is based on a block is called with the operation CALL, which is followed by the name of the block or function. The parameters are listed in the following lines.
A jump label and comments are optional.

```
M002:  CALL              Name              //Block call
       Additional details
       Parameter1        := Tag_1          //First parameter
       Parameter2        := Tag_2          //Second parameter
       Parameter...      := Tag...         //...
```

Fig. 10.2 Components of an STL statement

An STL statement consists of

- ▷ A jump label (optional), which must end with a colon.
- ▷ An operation, which describes what the CPU has to do (e.g. load, scan and link according to AND logic operation, compare, etc.).
- ▷ An operand, which contains the information necessary for executing the operation (e.g. an absolutely addressed operand %IW12, a symbolically addressed tag “Analog_Value_1”, a constant W#16#F001, a jump label, etc.). The operand can also be omitted depending on the operation.
- ▷ A comment (optional), commenced by two slashes and up to the end of the line.

With a block call, the call operation is followed by the parameter list in round brackets.

10.1.3 Entering an STL statement

The program editor creates a two-line input field in an empty network into which you can enter the STL statements.

Following input of the operation in a line, enter a space and then – if necessary – the operand; in the case of a binary logic operation, for example, enter a binary tag from one of the operand areas inputs, outputs, bit memories, and data.

In the same line you can enter a comment, separated by two slashes, up to the end of the line. You can also begin a new line with two slashes and enter a comment – as a sort of caption above the following statements.

Other options for making the statement list manageable are blank lines and networks.

Calling a block or a function with parameters

The operation CALL, followed by the parameter list, calls a block or (program) function with parameters. First the input parameters, then the output parameters, and finally the in/out parameters are listed in the parameter list, in the order of their declaration in each case. A comment can be added to each line. The entry label at the beginning of the statement and the comments are optional. If you use the mouse to drag a block from the *Program blocks* or a function from the program elements catalog into an STL line, the program editor displays the call statement and the parameter list.

Many functions can be provided with additional details, such as the data type or, when comparing time tags, the comparison operation. The additional details are located directly under the call operation. Fig. 10.3 shows two examples: The first function call expects actual parameters with the data type INT, the second function call compares tags with the data type DATE according to the relation “greater”.

The additional details can be selected from a drop-down list when programming the function.

```
    CALL   MIN                       //Call of the function "Determine minimum"
      Int                            // [Data type]
      IN1  := #Value1                //First parameter
      IN2  := #Value2
      IN3  := #Value3
      OUT  := #Result                //Output parameter

    CALL   T_COMP                    //Call "Compare time tags"
      Date GT                        // [Data type, comparison relationship]
      IN1  := #Date1                 //First parameter
      IN2  := D#2013-01-01
      OUT  := #var_greater           //Output parameter
```

Fig. 10.3 Examples of function calls with STL

10.1.4 Addressing of 64-bit tags

The statement list is based on a processor model with 32-bit-wide accumulators. A 64-bit tag, for example with the data type LINT, thus cannot be loaded into an accumulator and linked further. For the “simple” statements, there are functions that can handle 64-bit tags. These functions are supplied with STEP 7 in the global *Long Functions* library. The description is provided at the respective digital functions. In the upper part, Fig. 10.4 shows an example of adding tags with the data type LINT.

The majority of STL statements, which are based on a system or standard block, have been adapted to the tags with “long” data types and can be directly supplied with these tags. An example is shown in the lower part of Fig. 10.4.

```
    CALL   "ADD_LINT"                //Add tags with the data type LINT
      IN1  := #var1_lint             //Input tag 1
      IN2  := #var2_lint             //Input tag 2
      OUT  := #var3_lint             //Total

    CALL   T_CONV                    //Call "Convert time tags"
      LTime_Of_Day TO LTime
      IN   := #var_Date              //From data type LTIME_OF_DAY
      OUT  := #var_Time              //To data type LTIME
```

Fig. 10.4 Examples for addressing 64-bit tags with STL

10.1.5 STL networks in LAD and FBD blocks

Networks with STL program can also be used in a block with LAD or FBD program. To insert, select the network behind which the STL network is to be inserted, and select the command *Insert STL network* from the shortcut menu.

The processor of a CPU 1500 does not have the accumulators, address registers, data block registers, and status bits (status word) familiar from the CPU 300/400. These registers are emulated and are only available in the STL programming language.

If STL networks are used in a block with LAD or FBD program, no data can therefore be transferred between an STL network and an LAD/FBD network via these registers. In an LAD/FBD network which follows an STL network, the contents of these register are no longer available. If an STL network follows this, the register contents from a preceding STL network (in the same block) are available again.

One exception to this is the status bit RLO (result of logic operation): It is set to "undefined" during a language change and is no longer available in a subsequent network with changed programming language.

10.2 Programming binary logic operations with STL

The binary logic operations are carried out in the statement list using the AND, OR, and exclusive OR statements. The binary tags for the logic operation can be scanned for signal state "1" or "0". The binary operations can be "nested" using parenthesized expressions and thus influence the processing sequence (Table 10.1).

Table 10.1 Binary logic operations with STL

Operation	Operand	Function
A AN O ON X XN	Binary operand Binary operand Binary operand Binary operand Binary operand Binary operand	Scan for signal state "1" and link according to AND logic operation Scan for signal state "0" and link according to AND logic operation Scan for signal state "1" and link according to OR logic operation Scan for signal state "0" and link according to OR logic operation Scan for signal state "1" and link according to exclusive OR logic operation Scan for signal state "0" and link according to exclusive OR logic operation
A(AN(O(ON(X(XN( )	–	Left parenthesis with AND logic operation Left parenthesis with negation and AND logic operation Left parenthesis with OR logic operation Left parenthesis with negation and OR logic operation Left parenthesis with exclusive OR logic operation Left parenthesis with negation and exclusive OR logic operation Right parenthesis
O	–	ORing of AND functions
NOT SET CLR	–	Negation of result of logic operation Set result of logic operation to "1" Set result of logic operation to "0"

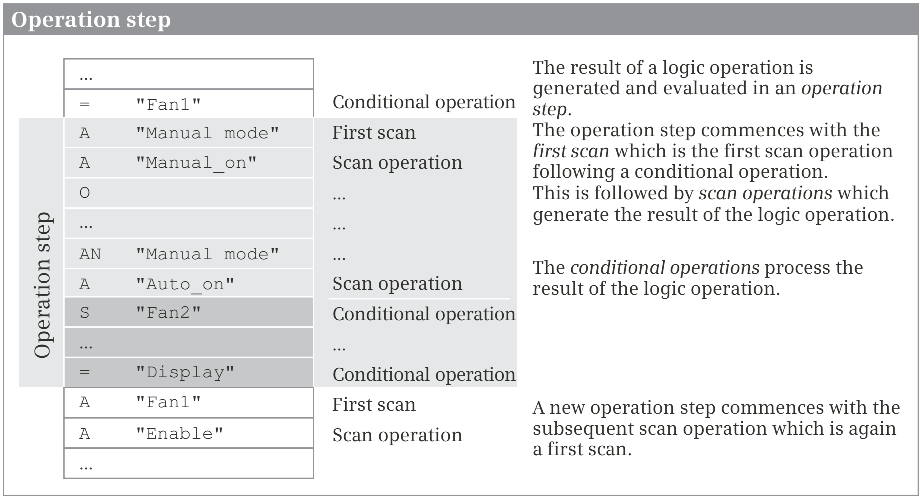
10.2.1 Processing of a binary logic operation, operation step

A binary logic operation consists of scan operations and conditional operations. The sequence of scan operations and subsequent conditional operations is referred to as an operation step (Fig. 10.5).

The first scan operation processed following a conditional operation is the *first input bit scan*. This is of special significance because the control processor directly imports the scan result of this statement as the result of logic operation. The "old" result of logic operation is thus lost. The first input bit scan always represents the beginning of a logic operation. The logic operation (AND, OR, Exclusive OR) specified in the first input bit scan does not play any role here. To make the programming understandable, however, it should correspond to the logic operation to be executed. For example, an OR function should begin with a scan for OR. For an individual scan statement without a link to other scans, the AND function is used.

The result of logic operation is generated by the *scan operations*. You scan the signal state of a binary operand for "1" or "0" and link it according to AND, OR or exclusive OR. The result of this logic operation is saved by the control processor as the new result of logic operation.

Conditional operations are operations whose execution depends on the result of logic operation. These are operations for assigning, setting and resetting binary operands, for starting timers and counters, etc. The conditional operations (apart from a few exceptions) are executed if the result of logic operation (RLO) is "1" and not executed if RLO is "0". They do not change the RLO (apart from a few exceptions), and therefore the RLO is the same for several successive conditional operations.



...		
=	"Fan1"	Conditional operation
A	"Manual mode"	First scan
A	"Manual_on"	Scan operation
O		...
...		...
AN	"Manual mode"	...
A	"Auto_on"	Scan operation
S	"Fan2"	Conditional operation
...		...
=	"Display"	Conditional operation
A	"Fan1"	First scan
A	"Enable"	Scan operation
...		

The result of a logic operation is generated and evaluated in an *operation step*.
The operation step commences with the *first scan* which is the first scan operation following a conditional operation.
This is followed by *scan operations* which generate the result of the logic operation.

The *conditional operations* process the result of the logic operation.

A new operation step commences with the subsequent scan operation which is again a first scan.

Fig. 10.5 Binary logic operation with STL, definition of operation step

10.2.2 Scanning for signal states “1” and “0”

Before the scan operations link the signal states together, they scan the status of the associated binary tags.

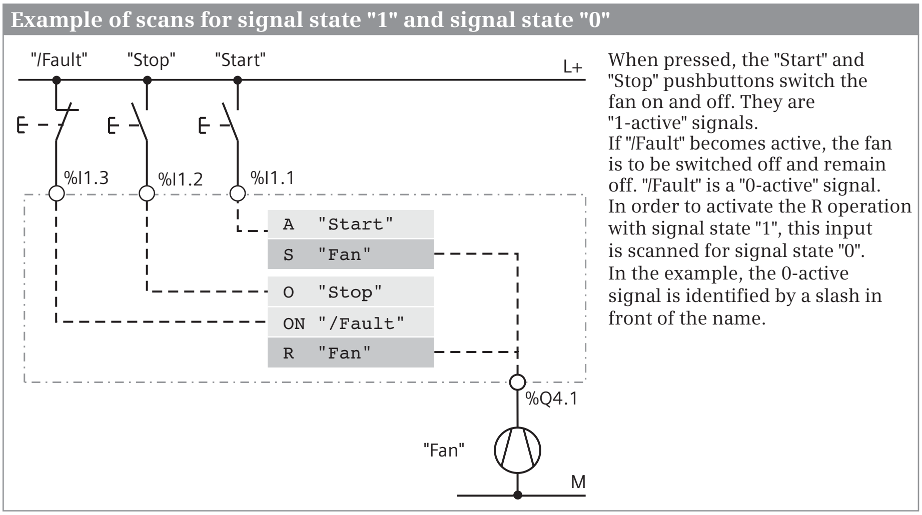
The *status* of a binary tag is identical to the signal state of the binary tag. This can be “0” or “1”. The physical variable at the module terminal for which an input has signal state “1” or “0” depends on the type of input module (see Chapter 12.1.2 “Working with binary signals” on page 504).

Strictly speaking, the control processor does not link the signal state of the binary tag scanned, it initially generates a *scan result*. When scanning for signal state “1”, the scan result is identical to the signal state of the binary tag scanned. When scanning for signal state “0”, the scan result is the negated signal state of the binary tag scanned. Scans for signal state “0” have an “N” following the specified logic operation (AN, ON, XN). The control processor generates the result of logic operation from the logic operation of the scan results.

The *result of logic operation* (RLO) is the signal state used by the control processor for further binary signal processing. The RLO contains the state of the binary logic operation: “1” means that the operation is fulfilled; “0” means that the operation is not fulfilled. The result of logic operation is used to set or reset binary tags.

The example in Fig. 10.6 shows the two *“Start”* and *“Stop”* pushbuttons. When pressed, they output the signal state “1” in the case of an input module with sinking input. The SR function is set or reset with this signal state.

The *“/Fault”* signal is not active in the normal case. Signal state “1” is then present and is negated by scanning for signal state “0”, and the reset operation therefore remains uninfluenced. If *“/Fault”* becomes active, the *“Fan”* tag is to be reset. The active signal *“/Fault”* delivers signal state “0”, which by scanning for signal state “0” activates the reset operation as signal state “1”.



Example of scans for signal state "1" and signal state "0"

```
A   "Start"
S   "Fan"
O   "Stop"
ON  "/Fault"
R   "Fan"
```

When pressed, the "Start" and "Stop" pushbuttons switch the fan on and off. They are "1-active" signals.
If "/Fault" becomes active, the fan is to be switched off and remain off. "/Fault" is a "0-active" signal.
In order to activate the R operation with signal state "1", this input is scanned for signal state "0".
In the example, the 0-active signal is identified by a slash in front of the name.

Fig. 10.6 Scanning for signal states “1” and “0”

10.2.3 AND function in the statement list

An AND function is fulfilled if all binary tags have the scan result “1”. A description of the AND function is provided in Chapter 12.1.3 “AND function, series connection” on page 507.

Fig. 10.7 shows an example of an AND function. The *#Fan1.works* and *#Fan2.works* tags are scanned for signal state “1” and the two scan results are linked according to an AND logic operation. The AND function is fulfilled (delivers signal state “1”) if both fans are running.

```
//AND function
A     #Fan1.works
A     #Fan2.works
=     #Display.twoFans                //Two fans are running

//OR function
O     #Fan1.works
O     #Fan2.works
=     #Display.MinOneFan              //At least one fan is running

//Exclusive OR function
X     #Fan1.works
X     #Fan2.works
=     #Display.oneFan                 //Only one fan is running
```

Fig. 10.7 Example of binary logic operations with STL

10.2.4 OR function in the statement list

An OR function is fulfilled if one or more inputs have the scan result “1”. A description of the OR function is provided in Chapter 12.1.4 “OR function, parallel connection” on page 507.

Fig. 10.7 shows an example of an OR function. The *#Fan1.works* and *#Fan2.works* tags are scanned for signal state “1” and the two scan results are linked according to an OR logic operation. The OR function is fulfilled (delivers signal state “1”) if one of the fans is running or if both fans are running.

10.2.5 Exclusive OR function in the statement list

An exclusive OR function (antivalence function) is fulfilled if an odd number of inputs has the scan result “1”. A description of the exclusive OR function is provided in Chapter 12.1.5 “Exclusive OR function, non-equivalence function” on page 508.

Fig. 10.7 shows an example of an exclusive OR function. The *#Fan1.works* and *#Fan2.works* tags are scanned for signal state “1” and the two scan results are linked by an exclusive OR logic operation. The exclusive OR function is fulfilled (delivers signal state “1”) if only one of the fans is running.

10.2.6 Combined binary logic operations in the statement list

The AND, OR, and exclusive OR functions can be freely combined with one another. The control processor processes an AND function with higher priority than an OR function (ANDing before ORing, like in the notation of Boolean algebra). The exclusive OR function has the same priority as an OR function.

The parentheses operations and the individual OR logic operation are available to bypass this processing priority.

ORing of AND functions

The individual OR logic operation O links the results of two AND functions.

Fig. 10.8 shows two AND functions with two inputs each, and the results of the logic operation are linked according to an OR logic operation. The first AND function is fulfilled if fan 1 is running and fan 2 is not running, the second function if fan 1 is not running and fan 2 is running. The *#Display.oneFan_1* tag is set if the first AND function is fulfilled or if the second AND function is fulfilled (or if both are fulfilled, but this is not the case in this example).

```
//Combined logic operation
//Without parentheses
A     #Fan1.works
AN    #Fan2.works
O                                 //ORing of AND functions
AN    #Fan1.works
A     #Fan2.works
=     #Display.oneFan_1           //Only one fan is running
```

Fig. 10.8 Example of ORing of AND functions

ANDing of OR functions

A parenthesized expression is required for the ANDing of OR functions. The OR functions are written in parentheses, and their results of the logic operation are linked to the operation present next to the parentheses (the AND function in this case).

Fig. 10.9 shows two OR functions: The first one is fulfilled if at least one fan is running or if both fans are running, the second one if at least one fan is not running or if neither fan is running. Each OR function itself stands in a parenthesized expression. The logic operation results of the OR functions are – due to the operation

“A(” – connected according to AND. The *#Display.oneFan_2* tag is set if only one of the fans is running.

```
//Combined logic operation
//With parentheses
A(
O     #Fan1.works
O     #Fan2.works
)
A(                               //ANDing of OR functions
ON    #Fan1.works
ON    #Fan2.works
)
=     #Display.oneFan_2          //Only one fan is running
```

Fig. 10.9 Example of ANDing of OR functions

Parenthesized expressions in binary logic operations

The example in Fig. 10.9 clearly indicates the generally applicable schema for binary parenthesized expressions. The function to be processed “first” is present in a parenthesis. How the result of the logic operation of the parenthesis is to be processed further is shown by the logic operation specified in front of the left parenthesis operation. Fig. 10.10 is a general representation of this schema.

A parenthesized expression can be linked by the operation “A(” according to AND, by the operation “O(” according to OR, and by the operation “X(” according to exclusive OR. Just like with scanning for signal state “0”, implicit negation of the

Processing a binary parenthesized expression

`...`		The logic operation prior to the parenthesized expression delivers a result of logic operation RLO 1, which is saved during processing of the left-parenthesized operation. The logic operation in the parenthesized expression delivers a result RLO 2. This result is gated with the saved RLO 1 in accordance with the specification present in the left-parenthesized operation. The result of the logic operation following the parenthesis is therefore: RLO 3 = RLO 1 (Parenthesis function) RLO 2
`Logic operation 1`		
`...`	Delivers RLO 1	
`Parenthesis function (`		
`...`		
`Logic operation 2`		
`...`	Delivers RLO 2	
`)`	Following the parenthesis: RLO 3	
`...`		
`Further logic operation`		
`...`		

Fig. 10.10 Generally applicable schema for the processing of binary parenthesized expressions

signal state is also possible here: The operation “AN(” negates the signal state of the parenthesized expression prior to linking, as do the operations “ON(” and “XN(”.

Any logic operations can be present within the parenthesized expression, including operations with parenthesized expressions. The nesting depth has a value of seven, i.e. a parenthesized expression can be commenced up to seven times without it being necessary to first terminate a parenthesized expression. Any number of parenthesized expressions can be programmed “in succession” (on one level).

Conditional operations in parenthesized expressions

All STL operations can be programmed within a parenthesized expression. The use of conditional operations such as assignment or setting/resetting is of interest in association with binary logic operations. Note that only the result of logic operation may be linked further which is valid with the right parenthesis operation.

In Fig. 10.11, a memory function is programmed with set and reset operations within a parenthesized expression. The signal state of the memory function must be scanned in order to link it further; this is carried out using the scan operation in front of the right parenthesis operation. The resulting AND function has three inputs: the OR function in front of the parenthesis, the signal state of the memory function in the parenthesis, and the last scan operation with the flashing frequency.

```
//Parenthesized
//expressions with
//conditional operations
A(                           //OR function is first AND input
O     #Enable_manual
O     #Enable_auto
)

A(
A     #Fan1.start
S     #Fan1.drive            //Set memory
O     #Fan1.stop
ON    #Fan1.fault
R     #Fan1.drive            //Reset memory
A     #Fan1.drive            //Scan memory!
)                            //Memory state is second AND input

A     "Clock 0.5 Hz"         //Flashing pulse is third AND input
=     #Fan1.display1
```

Fig. 10.11 Example with conditional operations in a parenthesized expression

10.2.7 Control of result of logic operation

Negate RLO

The NOT operation negates the result of logic operation at any position in an operation. An operation step is not ended by NOT. Using the NOT operation it is possible in a simple manner to obtain:

- ▷ a NAND function, i.e. a negated AND function, which is fulfilled if at least one input has the scan result “0”,
- ▷ a NOR function, i.e. a negated OR function, which is fulfilled if all inputs have the scan result “0”, and
- ▷ an inclusive OR function (equivalence function), i.e. a negated exclusive OR function which is fulfilled if an even number of inputs has the scan result “1”.

Fig. 10.12 shows a NOR function in the top example. The OR function is not fulfilled if none of the fans is running, and then delivers the signal state “0”. This is negated and assigned to the *#Display.noFan* tag.

Set and reset RLO

The SET operation sets the result of logic operation to “1”. The CLR operation sets the result of logic operation to “0”. SET and CLR terminate an operation step (Fig. 10.12).

```
//Negate result of
//logic operation
O     #Fan1.works
O     #Fan2.works                 //Negate RLO
NOT                               //No fan is running
=     #Display.noFan

//Set RLO
SET                               //Set RLO to "1"
S     #Fan1.drive                 //Fan 1 is switched on
R     #Fan2.drive                 //Fan 2 is switched off

//Reset RLO
CLR                               //Set RLO to "0"
CD    "Parts_counter"             //The internal edge trigger flag for
                                  //counting down is reset
```

Fig. 10.12 Example of controlling the result of logic operation

10.3 Programming memory functions with STL

The memory functions control binary tags such as outputs or bit memories. Memory functions exist for assigning, setting, and resetting a binary tag or for evaluating a change in signal state (Table 10.2).

Table 10.2 Memory functions with STL

Operation	Operand	Function
=	Binary tag	Assignment of result of logic operation
S R	Binary tag Binary tag	Set to signal state "1" with result of logic operation "1" Reset to signal state "0" with result of logic operation "1"
FP FN	Edge trigger flag Edge trigger flag	Evaluation of a positive edge of result of logic operation Evaluation of a negative edge of result of logic operation

10.3.1 Assignment in the statement list

The assignment directly assigns the result of logic operation to the binary tag named with the operation. The response of the assignment is described in Chapter 12.2.2 "Simple and negating coil, assignment" on page 511.

In Fig. 10.13, the *#Display.MinOneFan* tag is set to signal state "1" if the operation is fulfilled and to signal state "0" if it is not fulfilled ①. The result of logic operation is negated by NOT and, together with a further statement, controls the *#Display.noFan* tag.

```
//Assignment                      ①
O     #Fan1.works
O     #Fan2.works
=     #Display.MinOneFan          //At least one fan is running
NOT                               //Negate RLO
=     #Display.noFan              //No fan is running

//Set                             ②
A     #Fan1.enable
A     #Fan1.start
S     #Fan1.drive                 //Switch on fan 1

//Reset                           ③
A     #Fan1.enable
A     #Fan1.stop
ON    #Fan1.fault
R     #Fan1.drive                 //Switch off fan 1
```

Fig. 10.13 Example of assignment, setting and resetting with STL

10.3.2 Setting and resetting in the statement list

The set or reset operation is used to assign signal state “1” or “0” to a binary tag in the case of a result of logic operation “1”. A result of logic operation “0” has no effect.

The response of these operations is described in Chapter 12.2.3 “Single setting and resetting” on page 511.

In Fig. 10.13, an AND function comprising *#Fan1.enable* and *#Fan1.start* controls the set operation ②. *#Fan1.drive* is set to signal state “1” if the AND function is fulfilled, or there is no reaction if the AND function is not fulfilled. The reset operation is controlled by an OR function where an AND function with two inputs is connected to its first input ③. *#Fan1.drive* is reset to signal state “0” if the operation is fulfilled, or there is no reaction if the operation is not fulfilled. As a result of positioning of the reset operation after the set operation, the memory response is “reset dominant”: If the logic operations in front of the two operations have signal state “1”, *#Fan1.drive* is reset or remains reset.

10.3.3 Edge evaluation in the statement list

Edge evaluation detects a change in the result of logic operation.

The edge evaluation has the result of logic operation “1” for one processing cycle if the result of logic operation prior to the operation changes from “0” to “1” (FP operation, rising edge) or from “1” to “0” (FN operation, falling edge). This “pulse” can be linked further or control a conditional operation.

The edge trigger flag is present next to the edge operation. This is a flag or data bit which saves the “old” signal state of the result of logic operation. The change in signal is recognized by comparing the signal states of the “new” (current) result of logic operation and the edge trigger flag (see also Chapter 12.3 “Edge evaluation” on page 515).

```
//Edge evaluation
A     #Alarm_bit
FP    #Alarm_bit_Edge_flag       //Evaluation for positive edge
S     #Alarm_memory              //#Set alarm memory

A     #Acknowledge
R     #Alarm_memory              //#Reset alarm memory

A     #Alarm_memory
A     "Clock 0.5 Hz"
=     #Indicator_light           //Display alarm
```

Fig. 10.14 Example of edge evaluation with STL

Fig. 10.14 shows an application of edge evaluation. Let us assume that a alarm has “arrived”, i.e. the *#Alarm_bit* signal changes to “1”. The *#Alarm_memory* tag is then set. The alarm memory can be reset using an *#Acknowledge* button. The alarm memory remains reset if *#Acknowledge* has signal state “0” again and *#Alarm_bit* is still present. *#Alarm_memory* is only set again by a further positive edge of *#Alarm_bit* (if *#Acknowledge* then no longer has signal state “1”).

10.4 Programming timer and counter functions with STL

10.4.1 SIMATIC timer functions in the statement list

The SIMATIC timer functions are an operand area in the CPU's system memory and their number is limited. Table 10.3 shows the operations possible in conjunction with a timer operand. The time response of the SIMATIC timer functions is described in detail in Chapter 12.4 “SIMATIC timer functions” on page 524.

For programming, enter the timer operation in a line or drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions* > *Basic instructions* > *Timer operations* to the working area. The operation is followed by a space and then the timer operand (T) to which you can assign a symbolic name in the PLC tag table.

Table 10.3 Operations for SIMATIC timer operands

Operation	Operand	Function
SP SE SD SS SF	Timer operand	Start a SIMATIC timer function as pulse Start a SIMATIC timer function as extended pulse Start a SIMATIC timer function as ON delay Start a SIMATIC timer function as retentive ON delay Start a SIMATIC timer function as OFF delay
FR R	Timer operand	Enabling a SIMATIC timer function Resetting a SIMATIC timer function
L LC	Timer operand	Direct loading of a time value Coded loading of a time value
A, AN O, ON X, XN	Timer operand	Status scan of a SIMATIC timer function and linking according to AND Status scan of a SIMATIC timer function and linking according to OR Status scan of a SIMATIC timer function and linking according to exclusive OR

When programming a SIMATIC timer function you must make sure that the operations are in the correct order: first enable, then start and reset, and finally load time value and scan status. In so doing, you only program the operations required for the function to be executed.

When starting a SIMATIC timer function, the control processor obtains the defined duration from accumulator 1. When and how this value enters the accumulator is unimportant. In order to make your program easier to read, you should preferably load the duration into the accumulator directly prior to the start operation, either as a constant with direct specification of the duration in data format S5TIME or as a tag with the duration as content. Loading of a value into the accumulator is described in Chapter 13.2.5 “Loading and transferring with STL” on page 562.

Note that a valid duration must also be present in accumulator 1 even if the timer function is not started when processing the start operation.

In Fig. 10.15, the time *“Fan3.on_delay”* is started as an ON delay by the positive edge of *#Fan3.start*. The duration of 3 seconds was previously loaded into the accumulator as the constant S5T#3S. Following expiry of the duration, the timer function *“Fan3.off_delay”* is started with the duration present as a value in the *#Follow-up_time* tag. The status of the timer function *“Fan.off_delay”* simultaneously has signal state “1” so that fan 3 is switched on following the ON delay. Once the start signal *#Fan3.start* has signal state “0”, fan 3 continues to run for the follow-up time and is then switched off.

```
//SIMATIC timer function
A     #Fan3.start
L     S5T#3S
SD    "Fan3.on_delay"             //Start as ON delay

A     "Fan3.on_delay"
L     #Follow-up_time
SF    "Fan3.off_delay"            //Start as OFF delay

U     "Fan3.off_delay"            //Scan status
=     #Fan3.drive
```

Fig. 10.15 Example of application of SIMATIC timer functions with STL

Example of clock generator

The somewhat more complex example in Fig. 10.16 shows a clock generator with a different pulse-to-pause ratio implemented by means of a single timer function. The JC statement *Conditional jump* is executed if the result of logic operation is “1”.

10.4.2 SIMATIC counter functions in the statement list

The SIMATIC counter functions are an operand area in the CPU's system memory and their number is limited. Table 10.4 shows the counter operations possible in conjunction with a counter operand. The counter response is described in detail in Chapter 12.6 “SIMATIC counter functions” on page 545.

For programming, enter the counter operation in a line or drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions > Basic instructions > Counter operations* in a line. The operation is followed by a space and then the counter operand (C) to which you can assign a symbolic name in the PLC tag table.

```
      AN     #Start_input
      R      "Timer function"
      R      #Output
      JC     M1
      A      "Timer function"
      JC     M2
      AN     #Output
      =      #Output
      L      #Pulse_duration
      JC     M2
      L      #Pause_duration
M2:   AN     "Timer function"
      SE     "Timer function"
M1:   NOP 0 //Further program
```

#Start_input starts the clock generator.

If the time *“Timer function”* is not running or has expired, it is started as an extended pulse.

The binary scaler *#Output* changes its signal state with each (new) start of the time and thus also determines the duration – *#Pulse_duration* or *#Pause_duration* – with which the time is started.

Fig. 10.16 Example of clock generator with different pulse-to-pause ratio

When programming a SIMATIC counter function you must make sure that the operations are in the correct order: first enable, then count, set and reset, and finally load count value and scan status. In so doing, you only program the operations required for the function to be executed.

When setting a SIMATIC counter function, the control processor obtains the initial count value from accumulator 1. When and how this value enters the accumulator is unimportant. In order to make your program easier to read, you should preferably load the initial count value into the accumulator directly prior to the set operation, either as a constant with direct specification of the count value in data format W#16# or C# or as a tag with the count value as content. Loading of a value into the accumulator is described in Chapter 13.2.5 “Loading and transferring with STL” on page 562.

Table 10.4 Operations for SIMATIC counter operands

Operation	Operand	Function
CU CD S	Counter operand	Increment a SIMATIC counter function by one unit Decrement a SIMATIC counter function by one unit Set a SIMATIC counter function to a start value
FR R	Counter operand	Enabling a SIMATIC counter function Resetting a SIMATIC counter function
L LC	Counter operand	Direct loading of a count value Coded loading of a count value
A, AN O, ON X, XN	Counter operand	Status scan of a SIMATIC counter function and linking according to an AND logic operation Status scan of a SIMATIC counter function and linking according to an OR logic operation Status scan of a SIMATIC counter function and linking according to an exclusive OR logic operation

Note that a valid count value must also be present in accumulator 1 even if the counter function is not set when processing the set operation.

Fig. 10.17 shows the counting of workpieces up to a specific quantity. The counter *#Parts_counter* is set by the *#Quantity_set* tag to a start value of 120. Each positive edge at the *#Workpiece_identified* tag decrements the count value by one unit. If a value of zero is reached – the counter status is then “0” – *#Quantity_reached* is set.

```
A      #Workpiece_identified
CD     "Parts_counter"                //Count down

A      #Quantity_set
L      C#120                          //Load default value
S      "Parts_counter"                //Set counter to default value

AN     "Parts_counter"                //Scan status of counter
=      #Quantity_reached
```

Fig. 10.17 Example of application of a SIMATIC counter function with STL

10.4.3 IEC timer functions in the statement list

An IEC timer function is available as pulse time (TP), as ON delay (TON), as OFF delay (TOF), and as accumulating ON delay (TONR). A detailed description of the timer response is provided in Chapter 12.5 “IEC timer functions” on page 539.

```
//IEC timer function
CALL #AlarmDelay                      //Start as ON delay
  Time                                //Data type of time value
  IN := #Measurement_too_high
  PT := T#10S                         //10 s duration
  Q  := #Alarm_too_high
  ET :=                               //ET is not required

//Load IEC timer function
CALL PRESET_TIMER
  Time IEC_Timer                      //Data types
  PT    := T#2s                       //Time value
  TIMER := #AlarmDelay                //Timer function

//Reset IEC timer function
CALL RESET_TIMER
  IEC_Timer                           //Data type of timer function
  TIMER := #AlarmDelay                //Timer function
```

Fig. 10.18 Example of IEC timer function with STL

For programming, drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions* > *Timer operations* into a line on the working area. When positioning, you select either as single instance or as local instance (multi-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* or *"DB_name".ET* for a single instance.

PRESET_TIMER loads an IEC timer function with a time value. RESET_TIMER resets an IEC timer function.

Fig. 10.18 shows the IEC timer function *#AlarmDelay*. It has been inserted as an ON delay TON which saves its data as a local instance (multi-instance) in the instance data block of the function block. If the *#Measurement_too_high* tag has signal state "1" for longer than 10 s, *#Alarm_too_high* is set.

10.4.4 IEC counter functions in the statement list

An IEC counter function is available as up counter (CTU), as down counter (CTD), or as up/down counter (CTUD). A detailed description of the counter response is provided in Chapter 12.7 "IEC counter functions" on page 553.

```
A     "Light_barrier2"
FP    "Light_barrier2_Edge_flag"
A     "Light_barrier1"
=     #temp_bool1                          //Count up

A     "Light_barrier1"
FP    "Light_barrier1_Edge_flag"
A     "Light_barrier2"
=     #temp_bool2                          //Count down

      CALL #Lock_counter                   //Start as up/down counter
        Int                                //Data type
        CU   := #temp_bool1
        CD   := #temp_bool2
        R    := #Acknowledge
        LOAD :=
        PV   := 0
        QU   :=
        QD   :=
        CV   :=
```

Fig. 10.19 Example of IEC counter function with STL

For programming, drag the corresponding symbol with the mouse from the program elements catalog under *Basic instructions* > *Counter operations* into a line on the working area. When positioning, you select either as single instance or as local instance (multi-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 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* or *“DB_name”.CV* for a single instance.

Fig. 10.19 shows the IEC 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 with the name of the instance and the component name, for example *#Lock_counter.CV*. The example shows the passages through a lock, either forward or backward.

10.5 Programming digital functions with STL

The digital functions process digital values up to length of 32 bits in the accumulators 1 and 2. For tags that are 64 bits long, there are the functions in the global library *Long Functions* of STEP 7.

The processor of a CPU 1500 does not have the accumulators, address registers, data block registers, and status bits that are typical for a CPU 300/400. If STL statements are used in connection with these registers and status bits, they are emulated. This additional machine code not only requires more memory space, but also more processing time.

Accumulators 1 and 2 are sufficient to execute a digital function. They are supplied with numerical values with the load function (L); the digital function manipulates these values and the transfer function (T) transmits the result back to the user or system memory. Fig. 10.20 shows the allocation of the accumulators for the execution of digital functions.

10.5.1 Transfer functions in the statement list

A transfer functions copies the value of a tag. A detailed description of the transfer functions is provided in Chapter 13.2.5 “Loading and transferring with STL” on page 562. Table 10.5 shows the transfer functions available with STL.

Transfer with loading and transferring

The *Load* operation transfers a digital value from the CPU's memory area into accumulator 1. The *Transfer* operation transfers a digital value from accumulator 1 to the memory area. The program elements catalog contains the transfer functions under *Basic instructions* > *Basic instructions* > *Load and transfer*.

Accumulator assignment with digital functions

Digital function with one input value
The input value is loaded and occupies accumulator 1. The result of the manipulation is again present in accumulator 1, from where it can be transferred to a tag. The contents of accumulator 2 are not changed.

Program — *Accumulator assignment **following** execution of instruction*

Program	Accumulator 1	Accumulator 2
L #Input_value	Input_value	<Accumulator 1>
Function	Result	<Accumulator 1>
T #Result	Result	<Accumulator 1>

Digital function with two input values
The input values are loaded in succession. The first value loaded initially occupies accumulator 1, and this is shifted into accumulator 2 when the second input value is loaded. Accumulator 1 is then occupied by the second input value loaded. The logic operation is carried out according to the following scheme:
Result = <Accu 2> Function <Accu 1> = <Input_value 1> Function <Input_value 2>
The result of the logic operation is present in accumulator 1, from where it can be transferred to a tag. The first input value loaded is present in accumulator 2.

Program — *Accumulator assignment **following** execution of instruction*

Program	Accumulator 1	Accumulator 2
L #Input_value_1	Input_value_1	<Accumulator 1>
L #Input_value_2	Input_value_2	Input_value_1
Function	Result	Input_value_1
T #Result	Result	Input_value_1

Digital function with several input values
The input values are loaded in succession. The first value loaded initially occupies accumulator 1, and this is shifted into accumulator 2 when the second input value is loaded. Accumulator 1 is then occupied by the second input value loaded. Following the logic operation, the next input value is loaded and gated with the previous result, etc. Intermediate results can be saved using the transfer function without changing the accumulator contents.

Program — *Accumulator assignment **following** execution of instruction*

Program	Accumulator 1	Accumulator 2
L #Input_value_1	Input_value_1	<Accumulator 1>
L #Input_value_2	Input_value_2	Input_value_1
Function	Intermediate_result	Input_value_1
L #Input_value_3	Input_value_3	Intermediate_result
Function	End_result	Intermediate_result
T #End_result	End_result	Intermediate_result

Fig. 10.20 Accumulator assignment with digital operations

Table 10.5 Transfer functions with STL

Operation	Operand	Meaning, remark
L	Digital tag from the operand areas: peripheral inputs, peripheral outputs, inputs, outputs, bit memories, data, and temporary local data	Transfer from an operand area to accumulator 1
L LC	SIMATIC timer function, SIMATIC counter function	Direct loading into accumulator 1 Coded loading into accumulator 1
T	Digital tag from the operand areas: peripheral inputs, peripheral outputs, inputs, outputs, bit memories, data, and temporary local data	Transfer from accumulator 1 to an operand area
Function	**Parameter**	**Meaning, remark**
MOVE_LWORD MOVE_LINT MOVE_ULINT MOVE_LREAL	Tags with the data types LWORD, LINT, ULINT, and LREAL	Transfer between tags with "long" data types
MOVE_BLK_VARIANT	Tags with any data types (except BOOL) or components of an ARRAY tag	Transfer of values to tags of any length or transfer of ARRAY components
MOVE_BLK UMOVE_BLK	Components of an ARRAY tag	Transfer between two ARRAY tags
FILL_BLK UFILL_BLK	Components of an ARRAY tag	Filling in the area of an ARRAY tag
BLKMOV UBLKMOV	Tags and constants with any data types (except BOOL) and data areas addressed with a pointer	Transfer of tags and data areas of any length
FILL	Tags and constants with any data types (except BOOL) and data areas addressed with a pointer	Transfer of values (bit patterns) to tags and data areas of any length
SWAP	Tags with the data types WORD, DWORD, and LWORD	Change the byte sequence

For SIMATIC timer/counter functions, the *coded loading* (LC) transfers the current time value or the current counter value (BCD-coded) to accumulator 1. Observe that a BCD-coded time value does not contain the time scale.

Transfer of tags with "long" data types

The transfer functions MOVE_LWORD, MOVE_LINT, MOVE_ULINT, and MOVE_LREAL copy tags with the data types LWORD, LINT, ULINT, and LREAL. Conversion functions handle the transfer between tags with "short" and "long" data types.

The transfer functions for tags with "long" data types are available in the global *Long Functions* library.

Transfer of components of an ARRAY tag

The transfer functions MOVE_BLK and UMOVE_BLK transfer components of an ARRAY tag to a different ARRAY tag with the same structure and the same data type of the components. The transfer functions FILL_BLK and UFILL_BLK fill components of an ARRAY tag. These transfer functions can be found in the program elements catalog under *Basic instructions* > *Move operations*.

Transfer of tags with any data types

The transfer functions MOVE_BLK_VARIANT, BLKMOV and UBLKMOV transfer tags with any data types (except for BOOL) and data areas addressed with an ANY pointer. The transfer function FILL fills a tag or a data area with a value. These transfer functions can be found in the program elements catalog under *Basic instructions* > *Move operations*.

Fig. 10.21 shows examples of the transfer functions in STL: The *#Alarms* tag is transferred from the data block "*Data.STL*" to the "*Alarm_bits*" tag in the bit memory address area. The tag *#var_ulint* with data type ULINT is loaded with the value 123. BLKMOV transfers the value of the *#var_ulint* tag into the bit memory address area beginning at memory byte %MB 32.

```
L     "Data.STL".Alarms                //Load value into accumulator 1
T     "Alarm_bits"                     //Fetch value from accumulator 1

CALL  "MOVE_ULINT"                     //Transfer of "long" tags
  IN  := 123
  OUT := #var_ulint

CALL  "BLKMOV"                         //Transfer with BLKMOV
  VARIANT
  SRCBLK  := #var_ulint                //Transfer of tags and data areas
  RET_VAL := #var_int                  //of any length
  DSTBLK  := P#M32.0 BYTE 8
```

Fig. 10.21 Example of transfer functions in STL

10.5.2 Comparison functions in the statement list

A comparison function compares two digital values and delivers a binary result TRUE (signal state "1") for a fulfilled comparison or FALSE (signal state "0") for an unfulfilled comparison. The comparison functions are described in Chapter 13.3 "Comparison functions" on page 570. Table 10.6 shows the comparison functions available with STL.

The "simple" comparison functions make comparisons according to the data types INT, DINT, and REAL. Tags with the data types USINT or UINT can be compared after loading according to INT or DINT. If a tag is extended with the data type SINT and the correct sign, it can be compared according to INT. For two tags with the data type UDINT to be compared, they must be converted to the data type ULINT before the comparison according to ULINT.

Table 10.6 Comparison functions with STL

Operation	Operand	Meaning, remark
==		Compare for equal
<>		Compare for unequal
>		Compare for greater than
>=		Compare for greater than-equal
<		Compare for less than
<=		Compare for less than-equal
I	–	according to INT characteristic
D	–	according to DINT characteristic
R	–	according to REAL characteristic
Function	**Parameter**	**Meaning, remark**
EQ_LWORD	LWORD tags	Compare for equal
NE_LWORD		Compare for unequal
EQ_		Compare for equal
NE_		Compare for unequal
GT_		Compare for greater than
GE_		Compare for greater than-equal
LT_		Compare for less than
LE_		Compare for less than-equal
LINT	LINT tags	
ULINT	ULINT tags	
LREAL	LREAL tags	
Function	**Parameter**	**Meaning, remark**
T_COMP	Date and time values	Comparison of tags with time data types
S_COMP	Strings	Comparison of tags with string data types

Comparison of tags with data types INT, DINT, and REAL

The comparison functions compare the contents of accumulators 1 and 2, and the result of the comparison is assigned to the result of logic operation. The result of logic operation has signal state “1” if the comparison is fulfilled, otherwise “0”. The program elements catalog contains the comparison functions under *Basic instructions* > *Basic instructions* > *Comparator operations*.

Fig. 10.22 shows the general scheme which is used to carry out a comparison function ①. A comparison function does not change the accumulator contents. It is always carried out independent of conditions. A comparison function sets the status bits.

The comparison function delivers a binary result of logic operation and can therefore be used together with other binary functions. An operation step commences with the comparison function. Fig. 10.22 shows some examples of how you can integrate a comparison function into a binary logic operation.

② At the beginning of a logic operation, a comparison function is always a first input bit scan. The RLO delivered by the comparison function can be directly further linked with binary scans.

```
L     Tag1
L     Tag2
Comparison function
=     Result of comparison
```

① The tags are compared according to the schema *Tag1* (compare) *Tag2*.

```
L     #var1
L     #var2
<Comparison function>
A     "Input1"
=     "Output1"
```

② The tag *“Output1”* is set if the comparison is fulfilled and *“Input1”* has signal state “1”.

```
O     "Input2"
O(
L     #var1
L     #var2
<Comparison function>
)
O     "Input3"
=     "Output2"
```

③ The tag *“Output2”* is set if *“Input2”* or *“Input3”* has signal state “1” or if the comparison is fulfilled.

```
L     #var1
L     #var2
>I
JC    Greater than
==I
JC    Equal to
```

④ In the example, two comparison functions are applied to the same accumulator contents. The first comparison generates RLO = “1” if *#var1* is greater than *#var2* so that the jump to the *Greater than* label is carried out. The second comparison for equal to is then carried out without reloading the accumulators and generates a new RLO.

```
L     #var1
L     #var2
>I
JP    Greater than
JZ    Equal to
```

⑤ In this example, evaluation of the comparison is carried out using the status bits CC0 and CC1. The comparison relationship – “Greater than” in this case – is of no importance when setting the status bits, one could also have used a different relationship, e.g. “Less than”. JP scans whether the first comparison value is greater than the second one, JZ scans whether they are equal.

Fig. 10.22 Examples for the comparison function in binary logic operations

③ A comparison function within a binary logic operation must be set within parentheses since a new operation step is started with the comparison function (first input bit scan).

④ Since a comparison function does not change the accumulator contents, it is possible in STL to repeatedly carry out successive comparisons.

⑤ The comparison function sets the status bits depending on the relationship between the compared values, i.e. independent of the comparison operation specified. You can utilize this fact by scanning the status bits with the corresponding jump functions (see also Chapter 10.6.1 “Jump functions in the statement list” on page 436).

In Fig. 10.23, two comparison functions are programmed in the upper part ❶. For the first comparison, the *#Measurement_temperature* and *#Lower_limit* tags are loaded into the accumulators. The comparison function then compares the first value *#Measurement_temperature* (in accumulator 2) with the second value *#Lower_limit* (in accumulator 1) for “greater than or equal to” in data format INT. The result of the comparison is saved during processing of the operation “A(”. The second comparison is carried out with the *#Measurement_temperature* and *#Upper_limit* tags. Its comparison result is linked with the saved comparison result according to an AND logic operation. If both comparisons are fulfilled, i.e. if the *#Measurement_temperature* tag is between *#Lower_limit* and *#Upper_limit*, then *#Measurement_in_range* is set.

```
L     #Measurement_temperature    //"Simple" comparison function ❶
L     #Lower_limit
>=I                               //Comparison with lower limit
A(                                //Save comparison result 1
L     #Measurement_temperature
L     #Upper_limit
<=I                               //Comparison with upper limit
)                                 //Comparison results 1 and 2
=     #Measurement_in_range       //Link according to AND logic operation

CALL  "EQ_LINT"                   //Comparison of "long" tags ❷
  IN1 := #var_lint1
  IN2 := #var_lint2
  OUT := #var_bool                //Result of comparison

CALL  T_COMP                      //Comparison of time tags ❸
  LTime GT
  IN1 := #var_ltime1
  IN2 := #var_ltime2
  OUT := #var_bool                Result of comparison

CALL  S_COMP                      //Comparison of strings ❹
  String NE
  IN1 := #var_string1
  IN2 := #var_string2
  OUT := #var_bool                //Result of comparison
```

Fig. 10.23 Examples of comparison function with STL

Comparison of tags with “long” data types

The comparison functions EQ_xxx, NE_xxx, GT_xxx, GE_xxx, LE_xxx and LT_xxx compare tags with the data types LINT, ULINT and LREAL for equal to, not equal to, greater than, greater than-equal to, less than, and less than-equal to. EQ_LWORD and NE_LWORD compare two tags with the data type LWORD for equal to and not equal to. The comparison functions for tags with “long” data types are available in the global *Long Functions* library. Fig. 10.23 shows in the second part ❷ the comparison of two tags with the data type LINT for greater than.

Comparison of tags with a time data type

T_COMP compares tags with the data types DATE, TIME, DATE_AND_TIME, TIME_OF_DAY, LTIME, LTIME_OF_DAY, LDT, DTL and S5TIME for the relations equal to (EQ), not equal to (NE), greater than (GT), greater than-equal to (GE), less than (LT), and less than-equal to (LE). T_COMP can be found in the program elements catalog under *Extended instructions* > *Date and time-of-day*. The data types and the comparison relations are selected from the drop-down lists after being dragged to the working area. Fig. 10.23 shows in the third section ❸ the comparison of two tags with the data type LTIME for greater than.

Comparison of tags with a string data type

S_COMP compares tags with the data type STRING for the relations equal to (EQ), not equal to (NE), greater than (GT), greater than-equal to (GE), less than (LT), and less than-equal to (LE). T_COMP can be found in the program elements catalog under *Extended instructions* > *String + Char*. The comparison relations are selected from a drop-down list after being dragged to the working area. Fig. 10.23 shows the comparison of two strings for unequal in the fourth section ❹.

10.5.3 Arithmetic functions in the statement list

The arithmetic functions for numerical values realize the basic arithmetical operations addition, subtraction, multiplication, and division. A detailed description is provided in Chapter 13.4 “Arithmetic functions” on page 574. Table 10.7 shows the arithmetic functions available with STL.

“Simple” arithmetic functions

With the “simple” arithmetic functions, the contents of the accumulators 1 and 2 are added, subtracted, multiplied, and divided according to the data formats INT, DINT and REAL. The result is stored in accumulator 1. The program elements catalog contains the arithmetic functions under *Basic instructions* > *Basic instructions* > *Math functions*. Fig. 10.24 shows in the top part ① the general scheme that is used to program a “simple” arithmetic function. The first operand to be linked is initially loaded into accumulator 1. When loading the second operand, the content of accumulator 1 is shifted into accumulator 2. The contents of the accumulators 2 and 1 can then be linked using the arithmetic function. The result is stored in accumulator 1.

Table 10.7 Arithmetic functions with STL

Operation	Operand	Meaning, remark
+ – * /		Addition Subtraction Multiplication Division
I D R	– – –	according to INT characteristic according to DINT characteristic according to REAL characteristic
MOD	–	Division with remainder as result
DEC INC	Decrement Increment	Decremental reduction of the accumulator contents Incremental increase of the accumulator contents
Function	**Parameter**	**Meaning, remark**
ADD_ SUB_ MUL_ DIV_		Addition Subtraction Multiplication Division
LINT ULINT LREAL	LINT tags ULINT tags LREAL tags	of LINT tags of ULINT tags of LREAL tags
Function	**Parameter**	**Meaning, remark**
T_ADD T_SUB T_DIFF T_COMBINE	Tags with date and time data types	Addition of two tags Subtraction of two tags Calculation of difference between two points in time Combination of time and duration

```
L      #Tag1
L      #Tag2
<Arithmetic function>
T      #Result_of_calculation
```

① General representation of a "simple" arithmetic function

```
L      #Value1
L      #Value2
+I
L      #Value3
-I
T      #Result1
```

② #Result1 := #Value1 + #Value2 – #Value3

```
L      #Value6
L      #Value5
+R
+R
T      #Result2
```

③ #Result2 := #Value5 + 2 × #Value6

```
L      #Value8
L      #Value7
*D
*D
T      #Result3
```

④ #Result3 := #Value7 × (#Value8)^2

Fig. 10.24 Examples of "simple" arithmetic functions

An arithmetic function carries out the calculation according to the specified characteristic independent of the contents of the accumulators and independent of conditions.

You can permit an arithmetic function to directly follow a previous arithmetic function (chain calculation, Fig. 10.24). ②: The result of the first function is then linked further by means of the next function and the accumulators serve as intermediate memories. ③ and ④: The first loaded value remains unchanged in accumulator 2 during execution of the arithmetic function. You can reuse it without having to load it again.

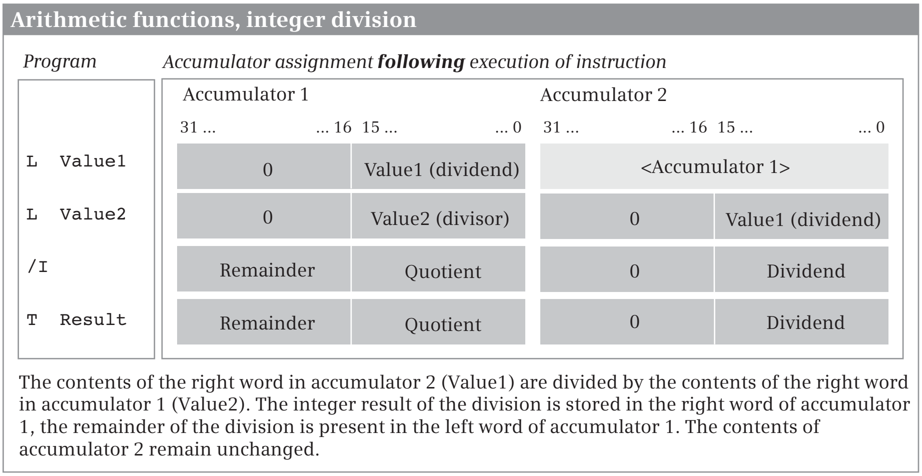
In Fig. 10.26 on page 426, the upper limit of a measured value is monitored ①. A hysteresis is introduced to ensure that the *#Measurement_too_high* alarm does not "pulsate" when the measured value changes rapidly around the upper limit. The alarm *#Measurement_too_high* is only canceled when the measured value has dropped again below the upper limit by the magnitude of the hysteresis.

Special features when calculating with data type INT

The left words of the accumulators are not taken into consideration when adding and subtracting. The result leaves the last word of accumulator 1 unchanged.

The left words of the accumulators are not taken into consideration when multiplying (*I). Following execution of the *I function, the product is present as a DINT number in accumulator 1.

The /I function interprets the values present in the right words of accumulators 1 and 2 as numbers with data type INT. It divides the value in accumulator 2 (dividend) by the value in accumulator 1 (divisor) and delivers two results: the quotient and the remainder, both numbers with data type INT (Fig. 10.25).



Arithmetic functions, integer division

Program — *Accumulator assignment* ***following*** *execution of instruction*

Program	Accumulator 1 31 16	 15 0	Accumulator 2 31 16	 15 0
`L  Value1`	0	Value1 (dividend)	<Accumulator 1>	
`L  Value2`	0	Value2 (divisor)	0	Value1 (dividend)
`/I`	Remainder	Quotient	0	Dividend
`T  Result`	Remainder	Quotient	0	Dividend

The contents of the right word in accumulator 2 (Value1) are divided by the contents of the right word in accumulator 1 (Value2). The integer result of the division is stored in the right word of accumulator 1, the remainder of the division is present in the left word of accumulator 1. The contents of accumulator 2 remain unchanged.

Fig. 10.25 Result of arithmetic function /I

Following execution of the function, the quotient is present in the right word of accumulator 1. It is the integer result of the division. The quotient is zero if the dividend is equal to zero and the divisor not equal to zero, or if the magnitude of the dividend is smaller than the magnitude of the divisor. The quotient is negative if the divisor was negative. Following /I, the leftover remainder of the division (not the decimal places!) is present in the left word. With a negative dividend, the remainder is also negative.

Following execution of the calculation, the status bits CC0 and CC1 indicate whether the quotient is negative, zero, or positive. The status bits OV and OS signal that the permissible numerical range has been left. A division by zero delivers a value of zero in each case as quotient and remainder and sets the status bits CC0, CC1, OV, and OS to “1”.

Division DIV with fixed-point numbers

DIV divides the input tag IN1 (dividend) by the input tag IN2 (divisor) and delivers the quotient in the result OUT. The quotient is the integer result of the division. The quotient is zero if the dividend is equal to zero and the divisor not equal to zero, or if the magnitude of the dividend is smaller than the magnitude of the divisor. The quotient is negative if the divisor is negative. A division by zero delivers a value of zero as quotient and sets the status bits CC0, CC1, OV, and OS to “1”.

Division DIV with floating-point numbers

DIV divides the input tag IN1 (dividend) by the input tag IN2 (divisor) and delivers the quotient in the result OUT. An error occurs if one of the input tags is an invalid floating-point number or if an attempt is made to divide the floating-point numbers ∞ by ∞ or 0 by 0.

Division MOD with remainder as result

MOD divides the input tag IN1 (dividend) by the input tag IN2 (divisor) and delivers the remainder of the division in the result OUT. The remainder is the leftover part of the division; this is not the decimal places. With a negative dividend, the remainder is also negative. A division by zero delivers a value of zero as remainder and sets the status bits CC0, CC1, OV, and OS to “1”.

Arithmetic functions with “long” data types

The functions for the addition, subtraction, multiplication, and division of tags with the “long” data types LINT, ULINT, and LREAL are located in the global library *Long Functions*. Fig. 10.26 shows an example of the multiplication of two tags with the data type ULINT ②.

Arithmetic functions for time tags

T_ADD, T_SUB and T_DIFF link points in time (date values or time of day) and durations. T_ADD can add two durations in the formats TIME and LTIME or add a duration (TIME, LTIME) to a point in time (TIME_OF_DAY, LTIME_OF_DAY, DATE_AND_TIME, DTL, LDT). T_SUB can subtract two durations in the formats TIME and LTIME or subtract a duration (TIME, LTIME) from a point in time (TIME_OF_DAY,

```
L     #Measurement_temperature      ① //Comparison of INT values
L     #Upper_limit
>=I
S     #Measurement_too_high
L     #Upper_limit
L     #Hysteresis
-I                                  //Subtraction according to INT
L     #Measurement_temperature
>I                                  //Comparison according to INT
R     #Measurement_too_high
CALL  "MUL_ULINT"                   ② //Multiplication according to ULINT
  IN1 := #var1_ulint
  IN2 := #var2_ulint
  OUT := #var3_ulint                //Result
CALL  T_DIFF                        ③ //Calculation of difference
  Date_And_Time TO Time
  IN1 := #var1_dt                   //Data type DATE_AND_TIME
  IN2 := #var2_dt
  OUT := #var_time                  //Result in data type TIME
```

Fig. 10.26 Example of arithmetic function with STL

LTIME_OF_DAY, DATE_AND_TIME, DTL, LDT). T_DIFF calculates the difference between two points in time (TIME_OF_DAY, LTIME_OF_DAY, DATE_AND_TIME, DATE, DTL, LDT) and outputs it as duration (TIME, LTIME, INT). These arithmetic functions can be found in the program elements catalog under *Extended instructions > Date and time-of-day*. You can select the data types from the drop-down lists after dragging them to the working area. Fig. 10.26 shows the calculation of the difference of two points in time in the DATE_AND_TIME format and the output as duration in the TIME format ③.

10.5.4 Math functions in the statement list

The math functions comprise, for example, trigonometric functions, exponential functions, and logarithmic functions. A detailed description is provided in Chapter 13.5 “Math functions” on page 578. Table 10.8 shows the math functions available with STL.

“Simple” math functions with the data type REAL

A math function calculates a result from the value present in accumulator 1 and saves it in accumulator 1. The program elements catalog contains the arithmetic functions under *Basic instructions > Basic instructions > Math functions*.

You program a math function in accordance with the scheme ① shown in Fig. 10.27. A math function is carried out independently of conditions and influences the status bits.

Table 10.8 Math functions with STL

Operation *)	Operand	Meaning, remark
SIN COS TAN	–	Calculate sine Calculate cosine Calculate tangent
ASIN ACOS ATAN	–	Calculate arcsine Calculate arccosine Calculate arctangent
EXP LN	–	Generate exponential function to base e Generate natural logarithm (to base e)
SQR SQRT	–	Generate square Extract square root
Function	**Parameter**	**Meaning, remark**
SIN_LREAL COS_LREAL TAN_LREAL	Tags with the data type LREAL	Calculate sine Calculate cosine Calculate tangent
ASIN_LREAL ACOS_LREAL ATAN_LREAL	Tags with the data type LREAL	Calculate arcsine Calculate arccosine Calculate arctangent
EXP_LREAL LN_LREAL	Tags with the data type LREAL	Generate exponential function to base e Generate natural logarithm (to base e)
SQR_LREAL SQRT_LREAL	Tags with the data type LREAL	Generate square Extract square root

*) If these operations are called via CALL, they can process REAL and LREAL tags.

```
L     Input tag                  ①
Math function                    //General schema
T     Result
-----------------------------------------------------------------
L     #Voltage                   ②
L     #Current
*R                               //Multiplication according to REAL
L     #phi
SIN                              //Calculate sine
*R                               //Multiplication according to REAL
T     #Reactive power            //Save result
-----------------------------------------------------------------
CALL  SQRT                       ③
  LREAL                          //Square root with data type LREAL
  IN  := #var1_lreal
  OUT := #var2_lreal
-----------------------------------------------------------------
CALL  SIN                        ④
  LREAL                          //Sine with data type LREAL
  IN  := #var1_lreal
  OUT := #var2_lreal
```

Fig. 10.27 Example of math functions with STL

Fig. 10.27 shows the calculation of the reactive power ② according to the equation *#Reactive power* = *#Voltage* × *#Current* × sin(*#phi*). The *#Voltage* and *#Current* tags are initially loaded and multiplied according to REAL. The *#phi* value is then loaded; the product of *#Voltage* and *#Current* is now present in accumulator 2. The sine of generated from the value *#phi*. The product of *#Voltage* and *#Current* present in accumulator 2 is multiplied by the sine of *#phi* present in accumulator 1 by means of the subsequent operation *R, the result saved in accumulator 1, and then transferred to the *#Reactive power* tag.

Math functions with the data type LREAL

The math functions for tags with the data type LREAL are available in the global *Long Functions* library. Fig. 10.27 shows the calculation of the square root from the tag *#var1_lreal* ③.

The "simple" math functions can also be called in STL as a function with the CALL operation. Then the data types REAL or LREAL can be selected from a drop-down list. Fig. 10.27 shows an example of a sine calculation of an LREAL tag under ④.

10.5.5 Conversion functions in the statement list

The conversion functions convert the data formats of tags. The "simple" conversion functions for STL convert the contents of accumulator 1. The data types of tags can be converted using "extended" conversion functions. A detailed description of the conversion functions is provided in Chapter 13.6 "Conversion functions" on page 586.

The program elements catalog contains the "simple" conversion functions under *Basic instructions* > *Basic instructions* > *Conversion operations* and the "extended" conversion functions under *Extended instructions* > *Date and time-of-day* (T_CONV) and *Extended instructions* > *String + Char*. Table 10.9 shows the conversion functions available with STL.

"Simple" conversion functions

A "simple" conversion function converts the value present in accumulator 1 and saves the result in accumulator 1. The conversion function is only effective on accumulator 1. Depending on the function, either only the right word (bits 0 to 15) or the complete contents are affected by this. Conversion functions do not change the contents of the remaining accumulators.

You program a conversion function in accordance with the general schema ① shown in Fig. 10.28. A conversion function is carried out according to the defined characteristic even if no data types have been declared when using absolutely addressed operands. A conversion function is carried out independent of conditions.

You can subject the content of accumulator 1 to several successive conversions and thus carry out conversions in several steps without having to save the converted values in intermediate memory.

Table 10.9 Conversion functions with STL

Operation	Operand	Function
ITD ITB DTB DTR	–	Data type conversion from INT to DINT Data type conversion from INT to BCD Data type conversion from DINT to BCD Data type conversion from DINT to REAL
BTI BTD	–	Data type conversion from BCD to INT Data type conversion from BCD to DINT
 RND+ RND– RND TRUNC	–	Data type conversion from REAL to DINT With rounding to the next higher number With rounding to the next lower number With rounding to the next integer Without rounding
INVI INVD NEGI NEGD NEGR ABS	–	One's complement for INT One's complement for DINT Two's complement (negation) of an INT number Two's complement (negation) of a DINT number Negation of a REAL number Generation of magnitude of a REAL number
Function	**Parameter**	**Meaning, remark**
CONVERT	Tags with elementary data type	Conversion of elementary data types (except time data types)
T_CONV S_CONV	Tags with time data type Tags with STRING data type	Conversion of time data types Conversion of string data types
VAL_STRG STRG_VAL	Tags with number and string data type	Conversion of a fixed-point or floating-point number into a STRING tag and vice versa
Chars_TO_Strg Strg_TO_Chars	Tags with CHAR/BYTE and string data type	Conversion of an ARRAY tag with CHAR or BYTE elements into a STRING tag and vice versa
ATH HTA	Tags with CHAR/BYTE and string data type	Conversion of an ARRAY tag with BYTE elements in hexadecimal representation into a STRING tag and vice versa

```
L     Input tag                    ①
Conversion function                //General schema
T     Result

L     #Measurement_temperature     //Successive conversions  ②
ITD                                //Conversion INT to DINT
DTB                                //Conversion DINT to BCD32
T     #Measurement_display

CALL CONVERT                       //Conversion with CONVERT  ③
  LINT TO REAL
  IN   :   := #var_lint
  RET_VAL := #var_real

CALL S_CONV                        //Conversion with S_CONV  ④
  LREAL TO STRING
  IN  := #var_lreal
  OUT := #var_string
```

Fig. 10.28 Example of conversion functions with STL

Fig. 10.28 shows an example of the “simple” conversion functions under ②. A measured value that is present in data format INT is loaded into accumulator 1, then it is first expanded to the data format DINT and then converted into the BCD format.

“Extended” conversion functions

The “extended” conversion functions are based on system blocks. You convert the data types of tags with elementary data types, time and string data types.

Fig. 10.28 shows two examples of the “extended” conversion functions. ③ A tag that is present in the LINT data format is converted via CONVERT into a tag with the data type REAL. ④ A tag that is present in the LREAL data format is converted via S_CONV into a string.

10.5.6 Shift functions in the statement list

A shift function shifts a tag value bit by bit to the right or left. The “simple” shift functions for STL shift the contents of accumulator 1. The values of tags can be shifted using “extended” shift functions. A detailed description of the shift functions is provided in Chapter 13.7 “Shift functions” on page 603.

Table 10.10 Shift functions with STL

Operation	Operand	Function
		Shift word-by-word
SLW		to left
SRW		to right
SSI		with sign to right
	n	with the shift number as parameter
	–	with the shift number in accumulator 2
		Shift doubleword-by-doubleword
SLD		to left
SRD		to right
SSD		with sign to right
	n	with the shift number as parameter
	–	with the shift number in accumulator 2
		Doubleword rotation
RLD		to left
RRD		to right
	n	with the shift number as parameter
	–	with the shift number in accumulator 2
		Doubleword rotation by one position
RLDA	–	to left by the condition code bit CC1
RRDA	–	to right by the condition code bit CC1
Function	**Parameter**	**Meaning, remark**
SHR	Tags with bit-serial and fixed-point data type	Shift to right
SHL		Shift to left
ROR	Tags with bit-serial and fixed-point data type	Rotate to right
ROL		Rotate to left

The program elements catalog contains the “simple” shift functions under *Basic instructions* > *Basic instructions* > *Shift and rotate*. Table 10.10 shows the shift functions available with STL.

“Simple” shift functions

A “simple” shift function shifts the value present in accumulator 1 by so many bit positions to the left or right as specified in accumulator 2 or as a parameter. The shift functions are carried out independent of conditions. They only change the content of accumulator 1. The result of logic operation (RLO) is not influenced. The shift functions set status bit CC0 to “0” and status bit CC1 to the signal state of the last bit shifted out.

L #Shift number	① General schema for the representation of a "simple" shift function
L #Input tag	
<Shift function>	▷ With the shift number in accumulator 2
T #Result	
L #Input tag	▷ With the shift number as parameter
<Shift function + shift number>	
T #Result	
L #Value	② Successive shift functions
SSD 4	
SLD 2	
T #Result	
L 16	③
L #Quantity_high	
SLD	//Shift with shift number in accumulator 2
L #Quantity_low	
SLW 4	//Shift word-by-word to left
OD	
SRD 4	//Shift doubleword-by-doubleword
T #Quantity_display	
L #Quantity_high	//Shorter program
SLD 12	//Shift doubleword-by-doubleword to left
L #Quantity_low	
OD	
T #Quantity_display	
CALL SHL	④
LWORD USINT	//Tag with data type LWORD
IN := #var_lword	//Shift to left
N := #var_usint	
RET_VAL := #var_lword	

Fig. 10.29 Examples of shift functions with STL

You can program a shift function in two different ways: with the shift number in accumulator 2 or with the shift number as parameter (Fig. 10.29, example ①). Shift functions can be applied as often as desired to the content of the accumulator. In the example ② in Fig. 10.29, shifting is carried out with the correct sign by (in the end) 2 positions to the right, where the two right bit positions are reset to signal state “0”.

In the example ③ in Fig. 10.29, the decades of two numbers present in BCD format of a SIMATIC counter are joined. In the top program the shift number 16 is loaded first and then with *#Quantity_high* the tag to be shifted. SLD shifts the content of the complete accumulator 1 by 16 (the shift number in accumulator 2). The subsequently loaded *#Quantity_low* tag is shifted by 4 bits to the left and linked according to an OR logic operation to the result of the previous shift. The six decades which are now present without gaps are shifted by a further 4 bits to the right and saved. The solution in the bottom program is somewhat shorter: The *#Quantity_high* tag is shifted to the left by three decades. The space which becomes vacant is occupied by the *#Quantity_low* tag.

“Extended” shift functions

The “extended” shift functions are based on system blocks. You shift the contents of tags with elementary data types. In Fig. 10.29, in the example ④, the value of a tag with the data type LWORD is shifted as many places to the left as specified by the value of the shift number with the data type USINT.

10.5.7 Word logic operations in the statement list

A word logic operation links the individual bits of two tags according to AND, OR, or exclusive OR. The “simple” word logic operations link the contents of accumulator 1 to a parameter or to the contents of accumulator 2. Tags with the data type LWORD can be processed using “extended” word logic operations. A detailed description of the word logic operations is provided in Chapter 13.8.1 “Word logic operations” on page 607.

The program elements catalog contains the “simple” word logic operations under *Basic instructions* > *Basic instructions* > *Word logic operations*. The “extended” word logic operations can be found in the global library *Long Functions*. Table 10.11 shows the word logic operations available with STL.

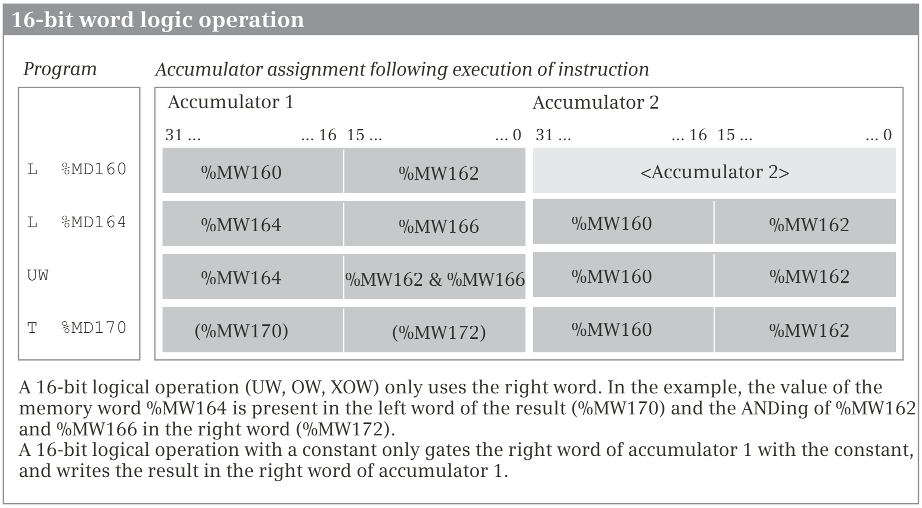
“Simple” word logic operation

A “simple” word logic operation links the contents of accumulator 1 either with the contents of accumulator 2 or with the parameter (Fig. 10.31, example ①). The word logic operations are carried out independent of conditions. The result of logic operation (RLO) is not affected.

The 16-bit word logic operations only act on the right word (bits 0 to 15) of the accumulators. The left word (bits 16 to 31) remains unaffected (Fig. 10.30).

Table 10.11 Word logic operations with STL

Operation	Operand	Function
AW AW AD AD	W#16#xxxx – DW#16#xxxx_xxxx –	Word-by-word AND logic operation with the parameter Word-by-word AND logic operation with the content of accumulator 2 Doubleword-by-doubleword AND logic operation with the parameter Doubleword-by-doubleword AND logic operation with the content of accumulator 2
OW OW OD OD	W#16#xxxx – DW#16#xxxx_xxxx –	Word-by-word OR logic operation with the parameter Word-by-word OR logic operation with the content of accumulator 2 Doubleword-by-doubleword OR logic operation with the parameter Doubleword-by-doubleword OR logic operation with the content of accumulator 2
XOW XOW XOD XOD	W#16#xxxx – DW#16#xxxx_xxxx –	Word-by-word exclusive OR logic operation with the parameter Word-by-word exclusive OR logic operation with the content of accumulator 2 Doubleword-by-doubleword exclusive OR logic operation with the parameter Doubleword-by-doubleword exclusive OR logic operation with the content of accumulator 2
Function	**Parameter**	**Meaning, remark**
AND_LWORD OR_LWORD XOR_LWORD	LWORD tags	Bit-by-bit link according to AND logic operation Bit-by-bit link according to OR logic operation Bit-by-bit link according to exclusive OR logic operation



Program	Accumulator 1 (31 ... 16)	Accumulator 1 (15 ... 0)	Accumulator 2 (31 ... 16)	Accumulator 2 (15 ... 0)
L %MD160	%MW160	%MW162	<Accumulator 2>	
L %MD164	%MW164	%MW166	%MW160	%MW162
UW	%MW164	%MW162 & %MW166	%MW160	%MW162
T %MD170	(%MW170)	(%MW172)	%MW160	%MW162

A 16-bit logical operation (UW, OW, XOW) only uses the right word. In the example, the value of the memory word %MW164 is present in the left word of the result (%MW170) and the ANDing of %MW162 and %MW166 in the right word (%MW172).
A 16-bit logical operation with a constant only gates the right word of accumulator 1 with the constant, and writes the result in the right word of accumulator 1.

Fig. 10.30 Execution of a 16-bit word logic operation

Following execution of a word logic operation you can directly connect the next word logic operation (load operand and execute word logic operation or execute word logic operation with constant) without having to save the intermediate result in an operand (e.g. local data). The accumulators serve here as intermediate memories. Examples are shown in Fig. 10.31 under ②.

Fig. 10.31 shows in the example ③ how you can program 32 edge evaluations simultaneously for rising and falling edges. The alarm bits are collected in a doubleword *Alarms*, which is present in the data block *“Data.STL”*. The edge trigger flags *Alarms_EM* are also present in this data block. If the two doublewords are

L #Tag1 L #Tag2 <Word logic operation> T #Result	① General schema of a "simple" word logic operation ▷ Linking with a value in accumulator 2
L #Tag <Word logic operation + constant> T #Result	▷ Linking with the parameter (constant)
L #Value1 L #Value2 AW L #Value3 OW T #Result1	② Successive word logic operations: The result of the AW operation is present in accumulator 1 and is shifted into accumulator 2 upon loading of #Value3. The two values can then be linked according to OW.
L #Value4 L #Value5 XOW AW 16#FFF0 T #Result2	The result of the XOW operation is present in accumulator 1. Its bits 0 to 3 are set to "0" by the AW statement.

L "Data.STL".Alarms
L "Data.STL".Alarms_EM
XOD
T #Alarms_change
	③
	//Edge evaluation example
	//What bits have changed?

L #Alarms_change L "Data.STL".Alarms AD T "Data.STL".Alarms_pos	//What change was a positive edge?
L #Alarms_change L "Data.STL".Alarms INVD AD T "Data.STL".Alarms_neg	//Invert alarm bits //What change was a negative edge?
L "Data.STL".Alarms T "Data.STL".Alarms_EM	//Update edge trigger flag

CALL "AND_LWORD"
IN1 := #var1_lword
IN2 := #var2_lword
OUT := #var3_lword
	④ “Extended" word logic operation

Fig. 10.31 Example of word logic operations with STL

linked by an XOR logic operation, the result is a doubleword in which each set bit represents a different assignment of *Alarms* and *Alarms_EM*, in other words: the associated alarm bit has changed.

In order to obtain the positive signal edges, the changes are linked to the alarms by an AND logic operation. The bit is set for a rising signal edge wherever the alarm and the change each have a “1”. This corresponds to the pulse flag of the edge evaluation. If you do the same with the negated alarm bits – the alarm 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 alarms.

“Extended” word logic operations

The “extended” word logic operations are based on system blocks. They link two tags with the data type LWORD according to AND, OR, or exclusive OR. An example is shown in Fig. 10.31 under ④.

10.5.8 Functions for strings in the statement list

Strings are tags with the data type STRING. With the functions for strings, parts of a string can be extracted, inserted, replaced or deleted, two strings can be combined, and the length of a string or the position of a character in a string can be determined.

A detailed description of these functions is provided in Chapter 13.9 “Processing of strings (data type STRING)” on page 615. You can find the functions for the processing of strings in the program elements catalog under *Extended instructions > String + Char*. Table 10.12 shows an overview of the available functions.

Table 10.12 Functions for the processing of strings

Function	Meaning, remark	Function	Meaning, remark
LEN	Outputs the current length of a string.	FIND	Finds characters in a string.
MAX_LEN	Outputs the configured maximum length of a string.	DELETE	Deletes part of a string.
LEFT	Outputs the left part of the string.	INSERT	Inserts characters into a string.
RIGHT	Outputs the right part of the string.	REPLACE	Replaces characters in a string.
MID	Outputs the middle part of the string.	CONCAT	Combines two strings together.

The example in Fig. 10.32 shows the replacement of one part of a string by another. In the input tag IN1, starting at position P, REPLACE replaces the number of characters L with the string to be inserted IN2 and outputs the new string at the parameter OUT.

```
CALL REPLACE                       //Replace characters
  STRING
  IN1 := #var1_string              //Input tag
  IN2 := #var2_string              //String to be inserted
  L   := #var1_int                 //Number of characters replaced
  P   := #var2_int                 //Character position
  OUT := #var3_string              //Output tag
```

Fig. 10.32 Example of string processing with STL

10.6 Program control with STL

You can influence execution of the user program by means of the program control functions. These are essentially:

- the jump functions (absolute jump, jumps depending on the result of logic operation or the condition code bits, jump list, loop jump),
- the block call functions (calling of FC and FB blocks and calling of functions which are based on blocks) and
- the block end functions (ending the block processing).

The familiar EN/ENO mechanism used with LAD, FBD and STL can be emulated in an STL program. If blocks written with STL are called in a block written with LAD, FBD or SCL, the ENO output must be specifically controlled.

10.6.1 Jump functions in the statement list

With jump functions you can exit linear program execution and continue at a different point in the block.

Table 10.13 shows the jump functions available with STL. The program elements catalog contains the jump functions under *Basic instructions* > *Basic instructions* > *Program control operations*. The absolute jump functions, jump functions depending on the result of logic operation, jump list, and the loop jump are described in detail in Chapter 14.1 “Jump functions” on page 623. A description of the jump functions that depend on the status bits is provided in Chapter 10.7.1 “Working with status bits” on page 442.

You can set the jump label prior to each statement in the block. A jump statement must always be followed by an operation; it can also be a nil operation. It is possible to jump within the block beyond network limits.

The example in Fig. 10.33 shows a program branch in part ①, which processes its own program section depending on the size of the tag *#Number*. After this, the execution of the shared program is continued.

Table 10.13 Jump functions with STL

Operation	Operand	Function
JU	Label	Jump absolute
JC	Label	Jump if RLO = “1”
JCN	Label	Jump if RLO = “0”
JCB	Label	Jump if RLO = “1” and save RLO in BR
JNB	Label	Jump if RLO = “0” and save RLO in BR
JBI	Label	Jump if BR = “1” 1)
JNBI	Label	Jump if BR = “0”
JZ	Label	Jump if result equal to zero 1)
JN	Label	Jump if result not equal to zero
JP	Label	Jump if result greater than zero
JPZ	Label	Jump if result greater than or equal to zero
JM	Label	Jump if result less than zero
JMZ	Label	Jump if result less than or equal to zero
JUO	Label	Jump if result invalid 1)
JO	Label	Jump if overflow
JOS	Label	Jump if stored overflow
JL	Label	Jump distributor
LOOP	Label	Loop jump

1) A description of these jump functions is provided in Chapter 10.7.1 “Working with status bits” on page 442

```
      L     #Number              //Comparison of a numerical value①
      L     10_000
      >I
      JC    Greater              //Jump if greater than 10_000
      ==I
      JC    Equal                //Jump if equal to 10_000
      ...                        //Program section for "less than"
      ...
      JU    next
Greater :...                     //Program section for "greater than"
      ...
      JU    next
Equal   :...                     //Program section for "equal to"
      ...
next    :...                     //Further program
      L     #Input_value         //Example of loop jump           ②
      T     #temp_dword
      L     32
next: T     #BitNumber           //#BitNumber is loop counter
      L     #temp_dword
      SLD   1                    //Shift to left by one position
      T     #temp_dword          //CC0 = 0 and CC1 = 1 if bit = "1"
      L     #BitNumber
      JN    ok                   //Jump if CC0 <> CC1
      LOOP  next                 //Back if bit was = "0"
      L     256                  //If no bit was = "1"
ok:   DEC   1
      T     #BitNumber
```

Fig. 10.33 Examples of jump functions

The lower part ② shows an example of the loop jump LOOP. The position of the highest set bit in the *#BitNumber* tag (data type USINT) is saved in the *#Input_value* tag (data type DWORD). If no bit is set, 16#FF is located in *#BitNumber*.

10.6.2 Block call function in the statement list

A block call function continues program execution in the called code block. A detailed description is provided in Chapter 14.2 “Calling of code blocks” on page 631. The program elements catalog contains the block call functions under *Basic instructions* > *Basic instructions* > *Program control operations*. Table 10.14 shows the available block call functions.

Table 10.14 Block call functions

Operation	Operand	Function
CALL CALL CALL	Code block #Instance Code block, data block	Calling a function (FC) Calling a function block (FB) as local instance Calling a function block (FB) as single instance
UC CC	Code block Code block	Absolute change to a block without parameter Conditional change to a block without parameter

Calling a block with CALL

CALL calls a block independently of the conditions. The called block can receive or return data via block parameters. CALL is also used to call (program) functions which are located in the program elements catalog under *Extended instructions*.

Directly after a block call, the signal state of the enable output ENO is saved in the binary result BR and it can be evaluated (see also Chapter 10.7.2 “EN/ENO mechanism in the statement list” on page 447).

Fig. 10.34 shows examples for calling a function (FC) and a function block (FB) with the operation CALL. ① Directly after the function call (FC), the binary result is scanned and the *#Adder_Error* tag is set to “1” in the event of an error. After this, a jump is made to an *Error* program section. ② The binary result is scanned directly after the call of the function block. If there is an error (BR is then “0”), the result of logic operation is negated and the block is left. BR remains “0” and transfers the signal state to the ENO output of the exited block.

With CALL, no data can be transferred to the called block via the accumulators, data block registers, and address registers.

Change to a block with UC or CC

With the operations UC (absolute block change) and CC (conditional block change), program execution is continued in the block which is present as an operand for UC or CC. Execution of CC depends on the result of logic operation: If the RLO = “1”, a switch is made to the specified block; if it is “0”, the statement that follows CC is processed.

```
//FC call and jump in event of error         ①
CALL  "Adder.STL"
      Number1 := "Data.STL".Number[1]
      Number2 := "Data.STL".Number[2]
      Number3 := "Data.STL".Number[3]
      Total := "Data.STL".Result[1]
A     BR                                     //BR = "0" means error
NOT                                          //Negate RLO
=     #Adder_error                           //Set error tag
JC    Error                                  //and jump to "Error"
```

```
//FC call as single instance                 ②
CALL  "Adder.STL", "DB_Adder.STL"
      Value1    := "Data.STL".Number[4]
      Value2    := "Data.STL".Number[5]
      Value3    := "Data.STL".Number[6]
      Result := "Data.STL".Result[2]
A     BR                                     //BR = "0" means error
NOT                                          //Negate RLO
BEC                                          //and block end
```

Fig. 10.34 Examples of block calls in STL

The block to which the switch is made can be a function (FC) or a function block (FB). The block must not have any block parameters and, in the case of a function block, it must not have any instance data blocks. In the called block, the block attribute *Parameter passing via register* must be activated and the block attribute *Optimized block access* must be deactivated. Then the program editor generates an additional code, which enables a data transfer via the accumulators, the data block registers, and the address registers. UC and CC reset the status bit OS (Exception Bit Overflow Stored). The status bit RLO (result of logic operation) is set to "undefined" and is no longer available in the called block. The remaining status bits remain unchanged.

The block to which the switch is made can be programmed as usual in the STL programming language, but without access to block parameters and, for a function block, without access to static local data.

You can also change to a block present as block parameter of type BLOCK_FC or BLOCK_FB. Memory-indirect addressing of the block is possible when changing using the UC and CC operations.

Fig. 10.35 shows examples of the switch to a different block with the operations UC and CC. "Block" is a function (FC) or a function block (FB) without block parameters.

Example ①: The change is carried out independent of conditions.

Example ②: The switch is carried out if the tag *#var_change* has signal state “1”.

Example ③ shows a data transfer via the data block register: The data block registers are written before the switch. In the block to which the switch is made, a data word of one data block is then transferred to a data word of the other data block. The executed program is as follows:

```
L     "DataBlock2".%DBW4
T     "DataBlock1".%DBW12
```

In the same way, data can be transferred via the accumulators and via the address registers.

```
//Unconditional change to a block                      ①
UC    "Block"
```

```
//Conditional change to a block                        ②
U     #var_change
CC    "Block"
```

```
//Example: Data transfer via register                  ③
//Program in calling block
OPN   "DataBlock1"
OPNDI "DataBlock2"
UC    "Block"

      //Program in "Block"
      //Register contents were retained
      L  %DIW4
      T  %DBW12
```

Fig. 10.35 Examples of block changes with UC and CC

10.6.3 Block end functions in the statement list

A block end function terminates program execution in a code block. A detailed description is provided in Chapter 14.3 “Block end functions” on page 636. The program elements catalog contains the block end functions under *Basic instructions* > *Basic instructions* > *Program control operations*. Table 10.15 shows the available block end functions.

Table 10.15 Block end functions

Operation	Operand	Function
BEC BEU BE	–	Conditional block end Unconditional block end Block end

The program execution in a code block can be ended with the statements BEC (conditional block end), BEU (unconditional block end), and BE (block end). Program execution is then continued in the calling block after the block call. BEC is only carried out if the result of logic operation = “1”. BEU and BE are carried out independently of conditions. BEU can be repeatedly programmed in the block, whereas BE can only be programmed once at the end of the block. BE can also be omitted.

The example ② in Fig. 10.34 shows the exiting of the block processing in the event of an error with BEC. In the example in Fig. 10.36, the block is exited in the event of an error with BR = “0”.

```
         ...
         SET                        //RLO = "1"
         SAVE                       //Set BR
         BEU                        //Block with BR = "1" exited
//Report error via RLO
Error:   CLR                        //Set RLO = "0" for error
         SAVE                       //Save RLO in BR
         BE                         //and block with BR = "0" exited
```

Fig. 10.36 Example of block end functions

10.7 Further STL functions

This chapter describes the operations that are specific to STL. These are operations which directly influence the contents of the status word, the accumulators, the address registers and the data block registers, as well as the nil operations.

The processor of a CPU 1500 does not have the accumulators, address registers, data block registers, and status bits (status word) familiar from the CPU 300/400. These registers are emulated and are only available in the STL programming language. This emulation increases the extent of the machine code that is generated during the compilation for STL, and thus increases the program execution time as well. The increased demand for code affects blocks and networks with the STL program which use the operations in connection with these tabs.

The following functions are described in this chapter:

▷ The status bits CC0, CC1, OV and OS are set during the processing of digital values. With the aid of the status bit BR, the EN/ENO mechanism can be emulated as it is used for LAD, FBD and SCL. The status bits can be scanned with binary logic operations and with jump functions.

▷ The accumulator functions manipulate the contents of the accumulators.

▷ The direct access to data block registers allows the indirect addressing of data blocks and a partial addressing of data operands.

- ▷ The use of address registers allows indirect addressing, i.e. the address of operands is only calculated during runtime and can be changed during runtime.
- ▷ The null instructions cause no response and are used as placeholders or for decompilation.

10.7.1 Working with status bits

Description of the status bits

The status bits are emulated in a CPU 1500 with additional program code. They are compiled in a 16-bit word, the status word. Table 10.16 shows the status bits available for a CPU 1500.

Table 10.16 Status bits, assignment of the status word

Bit No.	Status bit	Description	
0 to 3	–	Insignificant, filled with “0”.	–
4	OS	Exception Bit Overflow Stored The status bit OS saves a setting of status bit OV: Whenever the CPU sets the status bit OV, it also sets the status bit OS. However, whereas OV is reset by the next correctly executed operation, OS remains set.	You can evaluate OS with scan statements and with the jump statement JOS. JOS, UC and CC reset the status bit OS.
5	OV	Overflow The status bit OV indicates a numerical range overflow or the use of invalid floating-point numbers (REAL).	You can evaluate OV with scan statements and with the jump statement JO.
6 7	CC0 CC1	Condition code bit 0 Condition code bit 1 The status bits CC0 and CC1 provide information on the result of a comparison function, an arithmetic or math function, a word logic operation, or the shifted-out bit of a shift function.	You can evaluate all combinations of the status bits CC0 and CC1 using scan statements and jump functions.
8	BR	Binary result The binary result BR is an additional, block-independent memory for the result of logic operation. You can use BR also for the EN/ENO mechanism.	You can evaluate BR with scan statements and with jump functions.
9 to 15	–	Insignificant, filled with “0”.	–

Status word STW

The status word contains the status bits. You can load it into accumulator 1 or write it with a value from accumulator 1.

```
L   STW      //Load status word
T   #var     //and store in the tag #var
...
L   #var     //Load value of tag #var
T   STW      //and transfer to status word
```

You can use the status word to scan the status bits or to set them as required. You can thus save a current status word or commence a program section with a specific assignment of the status bits.

Setting the status bits by means of digital functions

Table 10.17 shows how the status bits CC0, CC1, OV and OS are set by the "simple" digital functions.

For arithmetic functions with the data formats INT, DINT and REAL, note the different meaning of CC0 and CC1 for an overflow.

Setting the displays for the comparison functions is independent of the comparison function that is carried out; it only depends on the relation of the two values being used in the comparison function. A REAL comparison checks for valid REAL numbers. The comparison functions also control the result of logic operation RLO: if the comparison is fulfilled, the RLO is set to signal state "1"; if the comparison is not fulfilled, it is set to "0".

Save binary result

The SAVE statement saves the result of logic operation in the binary result. The program elements catalog contains SAVE under *Basic instructions > Basic instructions > Bit logic operations*. Saving with SAVE does not affect the sequence of a logic operation.

A block change does not change BR, which means that you can pass on a binary state, e.g. a group error message, to the calling block.

Example:

```
A     #var_error  //In the event of an error, #var_error = "0"
SAVE              //In the event of an error, reset BR to "0"
NOT               //Negate RLO
BEC               //In the event of an error, exit the block
```

In the event of an error, the tag *#var_error* is set to signal state "0". With the scan of *#var_error*, the signal state is mapped in the result of logic operation and transferred into the binary result with SAVE. NOT negates the result of logic operation, which means that in the event of an error the block is exited with BR = "0". When the block is exited, the signal state of BR is transferred to the ENO output.

The jump functions JCB and JNB also influence the binary result BR (see Chapter 14.1.3 "Conditional jump functions" on page 625).

Evaluating status bits with scan statements

The status of the status bits can be mapped on the result of logic operation using scan statements and thus be integrated into a binary logic operation. Table 10.18 shows the available scan statements.

Table 10.17 Setting the status bits CC0, CC1, OS, and OV by means of digital functions

INT calculation

The result is:	CC0	CC1	OV	OS
< –32 768 (+I, –I)	0	1	1	1
< –32 768 (*I)	1	0	1	1
–32 768 to –1	1	0	0	–
0	0	0	0	–
+1 to +32 767	0	1	0	–
> +32 767 (+I, –I)	1	0	1	1
> +32 767 (*I)	0	1	1	1
32 768 (/I)	0	1	1	1
(–) 65 536	0	0	1	1
Division by zero	1	1	1	1

DINT calculation

The result is:	CC0	CC1	OV	OS
< –2 147 483 648 (+D, –D)	0	1	1	1
< -2 147 483 648 (*D)	1	0	1	1
–2 147 483 648 to –1	1	0	0	–
0	0	0	0	–
+1 to +2 147 483 647	0	1	0	–
> +2 147 483 647 (+D, –D)	1	0	1	1
> +2 147 483 647 (*D)	0	1	1	1
2 147 483 648 (/D)	0	1	1	1
(–) 4 294 967 296	0	0	1	1
Division by zero (/D, MOD)	1	1	1	1

REAL calculation

The result is:	CC0	CC1	OV	OS
+ normalized	0	1	0	–
± denormalized	0	0	1	1
± zero	0	0	0	–
– normalized	1	0	0	–
+ infinite (division by zero)	0	1	1	1
– infinite (division by zero)	1	0	1	1
± invalid REAL number	1	1	1	1

Comparison

The result is:	CC0	CC1	OV	OS
equal to	0	0	0	–
greater than	0	1	0	–
less than	1	0	0	–
Invalid REAL number	1	1	1	1

Conversion NEG_I

The result is:	CC0	CC1	OV	OS
+1 to +32 767	0	1	0	–
0	0	0	0	–
–1 to –32 767	1	0	0	–
(–) 32 768	1	0	1	1

Conversion NEG_D

The result is:	CC0	CC1	OV	OS
+1 to +2 147 483 647	0	1	0	–
0	0	0	0	–
–1 to –2 147 483 647	1	0	0	–
(–) 2 147 483 648	1	0	1	1

Conversion ITB, DTB

The result is:	CC0	CC1	OV	OS
Numerical range over-	–	–	1	1

Conversion RND, RND+, RND–, TRUNC

Conversion of an invalid REAL number	CC0	CC1	OV	OS
	–	–	1	1

Shift function

The shifted-out bit is:	CC0	CC1	OV	OS
"0"	0	0	0	–
"1"	0	1	0	–
with shift number 0	–	–	–	–

Word logic operation

The result is:	CC0	CC1	OV	OS
zero	0	0	0	–
not zero	0	1	0	–

Table 10.18 Scan statements for status bits

Operation	Operand	Function	
A	–	Scan for fulfilled condition and link according to AND logic operation	
O	–	Scan for fulfilled condition and link according to OR logic operation	
X	–	Scan for fulfilled condition and link according to OR logic operation	
AN	–	Scan for non-fulfilled condition and link according to AND logic operation	
ON	–	Scan for non-fulfilled condition and link according to OR logic operation	
XN	–	Scan for non-fulfilled condition and link according to OR logic operation	
			Condition:
	>0	Result greater than zero	[CC0 = 0] & [CC1 = 1]
	>=0	Result greater than or equal to zero	[CC0 = 0]
	<0	Result less than zero	[CC0 = 1] & [CC1 = 0]
	<=0	Result less than or equal to zero	[CC1 = 0]
	<>0	Result not equal to zero	[CC0 = 0] & [CC1 = 1] v [CC0 = 1] & [CC1 = 0]
	==0	Result equal to zero	[CC0 = 0] & [CC1 = 0]
	UO	Result is invalid (unordered)	[CC0 = 1] & [CC1 = 1] & [OV = 1] & [OS = 1]
	OV	Overflow	[OV = 1]
	OS	Exception Bit Overflow Stored	[OS = 1]
	BR	Binary result	[BR = 1]

The program elements catalog contains the scan operations under *Basic instructions* > *Basic instructions* > *Bit logic operations*.

Example: A floating-point number is checked for validity. To do this, the tag is compared with any floating-point constant. The type of comparison does not play a role here.

```
L     #var_real         //Floating-point number to be checked
L     1.0               //Any constant
==R                     //Any REAL comparison
A     UO                //Scan for "Result invalid"
=     #var_invalid      //Set error bit
```

The status bits can also be scanned with the loading of the status word.

Evaluating status bits with jump functions

If you evaluate the status bits with jump functions, you can jump directly to a different program section within the block depending on the status of the status bits. Table 10.19 shows the available jump functions.

The jump functions that depend on the status bits CC0, CC1, OV and OS do not change the result of logic operation and have no influence on a binary logic operation. After the jump function or at the jump destination, a binary logic operation can be continued.

The jump function JO is carried out if the result of an arithmetic operation is no longer located in the valid numerical range. In the case of a chain calculation with sev-

Table 10.19 Evaluating the status bits with jump functions

Operation	Operand	Description	
		Jump if:	Condition:
JP	*Label*	Result greater than zero	[CC0 = 0] & [CC1 = 1]
JPZ	*Label*	Result greater than or equal to zero	[CC0 = 0]
JM	*Label*	Result less than zero	[CC0 = 1] & [CC1 = 0]
JMZ	*Label*	Result less than or equal to zero	[CC1 = 0]
JN	*Label*	Result not equal to zero	[CC0 = 0] & [CC1 = 1] v [CC0 = 1] & [CC1 = 0]
JZ	*Label*	Result equal to zero	[CC0 = 0] & [CC1 = 0]
JUO	*Label*	Result is invalid (unordered)	[CC0 = 1] & [CC1 = 1] & [OV = 1] & [OS = 1]
JO	*Label*	Overflow	[OV = 1]
JOS	*Label*	Exception Bit Overflow Stored	[OS = 1]
JBI	*Label*	Binary result = "1"	[BR = "1"]
JBIN	*Label*	Binary result = "0"	[BR = "0"]

eral operations executed in succession, the status bit OV must be executed following each calculation function since the next calculation whose result is in the permissible numerical range resets OV again. Scan the status bit OS with the jump function JOS in order to evaluate a possible numerical range overflow at the end of the chain calculation. The jump function JOS resets the status bit OS. The example in Fig. 10.37 shows how a numerical range overflow in a chain calculation can be evaluated using scan statements or jump functions.

The jump functions JBI and JNBI terminate a binary logic operation; a new logic operation starts following the jump function or at the jump destination. The result of logic operation is retained and can be assigned to a binary tag, for example, after the jump function or at the jump destination.

```
//Scan statements             //Jump functions
   L     #Value1                 L     #Value1
   L     #Value2                 L     #Value2
   +I                            +I
   A     OV                      JO    Label_ST1
   =     #Status1
   L     #Value3                 L     #Value3
   +I                            +I
   A     OV                      JO    Label_ST2
   =     #Status2
   L     Value4                  L     Value4
   +I                            +I
   A     OS                      JOS   Label_ST
   =     #Group_status
   T     #Result                 T     #Result
```

Following the first and second additions, an evaluation is carried out for overflow. This evaluation of the overflow status bit only comprises the immediately preceding arithmetic function.

The overflow status bit OS saves a numerical range overflow for the complete calculation.

Fig. 10.37 Example of overflow evaluation

10.7.2 EN/ENO mechanism in the statement list

The EN/ENO mechanism with LAD, FBD, and SCL uses the enable input EN and enable output ENO. A conditional block call is implemented using EN. The called block signals an error to the program in the calling block via ENO. EN and ENO are statement sequences which the program editor adds to a block call or a function

```
//Conditional call of block             ①
...
       U       #var_call                //Call block with RLO = "1"
       JCN     M1                       /if RLO = "0", then jump to M1
       CALL "Block name"
         <Parameter list>
M1:    ...                              //Further program
```

```
//BR as group error                     ②
       SET
       SAVE                             //Set BR to "1" (= no error)
...
       L       #Number1
       L       #Number2
       +I
       T       #Total
       AN      OV                       //in the event of error: Scan = "0"
       A       BR                       //include BR
       SAVE                             //and save in BR again
...
//Block left                            //BR = "1": No error
       BE                               //BR = "0": Error
```

```
//in the event of error,                ③
//processing aborted
...
       L       #Number1
       L       #Number2
       +I
       T       #Total
       A       OV                       //If error occurred
       JC      Error                    //then jump to Error
...
//Block end
       SET
       SAVE                             //No error: BR = "1"
       BEU                              //Block left
Error: CLR
       SAVE                             //Error occurred: BR = "0"
       BE                               //Block left
```

```
//Evaluate ENO output                   ④
       CALL "Block name"
         <Parameter list>
       A       BR                       //Scan ENO output
       =       #var_error
...
                                        //Alternative: Scan with
                                        //Jump function JNB
```

Fig. 10.38 Emulating the EN/ENO mechanism in STL

and represents as block parameters. These statement sequences are not available for STL. If you also want to use the EN/ENO mechanism with STL, you must emulate it using corresponding statement sequences.

If you program a block using STL and you want to call it in LAD, FBD or SCL, you should ensure the “correct” supply of the status bit BR (binary result) because the enable output ENO assumes the signal state which the status bit BR had when the block was exited. If no error occurred, the block should be exited with BR = “1” and if the processing was erroneous, with BR = “0”. Some examples for this are shown in Fig. 10.38.

You can emulate the enable input EN in its function as conditional block call using a jump function. If the condition is not fulfilled, a jump is carried out beyond the block call and the block is not processed (see example ① in Fig. 10.38).

In the example ②, the status bit BR is used as a group error message. BR is set at the start of the block to “1”. If an error now occurs during block processing, e.g. if a numerical range overflow occurs (in the event of an error, the condition must deliver signal state “0” here), set the binary result to “0”. The error condition is linked to BR and the result is saved again in BR. This ensures that BR remains at signal state “0” if it has been set to “0” once. This error evaluation can be programmed multiple times in the block. When the block is exited, BR then has the value “0” when an error occurs and “1” if no error has occurred.

In the example ③, the program is aborted when an error is detected and a jump is made to the end of the block, to the jump destination *Error* in the example. BR is then set to “0” and the block is exited. This error evaluation can be programmed multiple times in the block. If no error occurs, the block program is processed to the end, BR is set to “1”, and then the block is exited.

Example ④ shows the evaluation of the ENO output after a block call. The signal state of the ENO output is saved in the status bit BR. It can be evaluated with a scan statement or with a jump statement. In the example, BR is scanned with an AND operation and the result is saved in the tag *#var_error*. Alternatively, the BR bit can also be evaluated using a jump function JCB or JNB. If you have used BR as a group error message in the calling block, observe that it can be overwritten in the called block.

10.7.3 Accumulator functions

When a block or network is programmed in the STL programming language, a CPU 1500 emulates two 32-bit arithmetic registers, the so-called accumulators. The accumulator functions transfer values between the two accumulators, swap bytes in accumulator 1, or change the content of accumulator 1. Execution of the accumulator functions is independent of the result of logic operation and of the status bits. Neither the result of logic operation nor the status bits are influenced. Table 10.20 shows the available accumulator functions of a CPU 1500.

In the program elements catalog, the accumulator functions can be found under *Basic instructions* > *Basic instructions* > *Additional instructions* (POP, PUSH, TAK), under *Basic instructions* > *Basic instructions* > *Math functions* (+), and under *Basic instructions* > *Basic instructions* > *Conversion operations* (CAW, CAD).

Table 10.20 Accumulator functions

Operation	Operand	Function
+	Constant	Add a constant value to the content of accumulator 1
PUSH POP TAK	–	Shift the content of accumulator 1 to accumulator 2 Shift the content of accumulator 2 to accumulator 1 Swap the contents of accumulators 1 and 2
CAW CAD	–	Swap the bytes in the right word of accumulator 1 Swap the bytes in accumulator 1

Adding a constant to accumulator 1

The addition of constants changes the contents of accumulator 1. You program the addition of constants according to the following general scheme:

```
L     Tag
+     Data_type#±k      //Adding a constant
T     Result
```

Example:

```
L     #index            //The value of the tag #index
+     INT#-256          //is reduced by -256
T     #index
```

The addition of constants is preferably used for calculating addresses since – unlike an arithmetic function with the load statement – it influences neither the contents of accumulator 2 nor the status bits.

The "Add constant" statement adds the constant present in the operation to the content of accumulator 1. The data type of the constant (SINT, INT, DINT, USINT, UINT, UDINT) comes before the constant. The constant can then be located within the numerical range of the data type.

With SINT, INT, USINT, UINT only the right word of accumulator 1 is influenced; no transfer to the left word takes place. A SINT constant is extended to 16 bits with the correct sign.

The addition of constants is independent of conditions.

Direct transfer between the accumulators

The statements TAK, PUSH and POP transfer the contents of accumulators 1 and 2. The operating principle of these statements is shown in Fig. 10.39.

The **TAK** statement swaps the contents of accumulators 1 and 2.

The **PUSH** statement shifts the content of accumulator 1 into accumulator 2. The content of accumulator 1 is not changed in the process. The previous content of accumulator 2 is lost.

The **POP** statement shifts the content of accumulator 2 into accumulator 1. The content of accumulator 2 is not changed in the process. The previous content of accumulator 1 is lost.

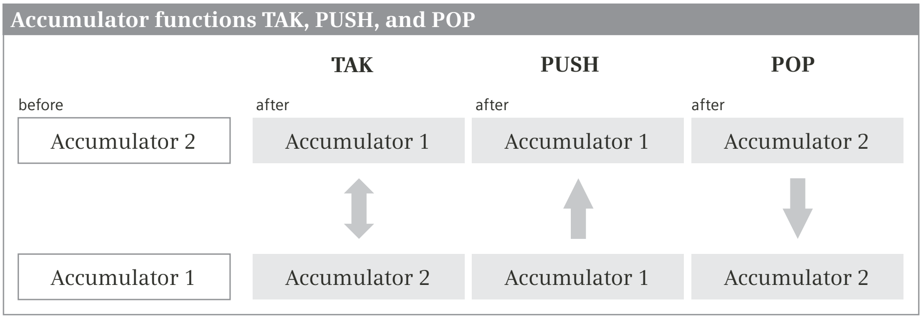


Fig. 10.39 Direct transfer between the accumulators

Swap bytes in accumulator 1

The **CAW** statement swaps the two right bytes in accumulator 1 (Fig. 10.40). The left bytes remained unchanged.

The **CAD** statement swaps all bytes in accumulator 1. The byte present on the far left is present on the far right following CAD; the two bytes in the middle swap locations.

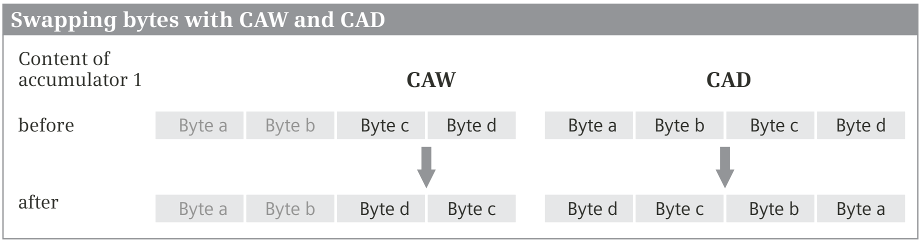


Fig. 10.40 Swapping bytes with CAW and CAD

The SWAP function is available for swapping bytes in a tag. Example:

```
CALL SWAP
  LWORD
  IN      := #var1_lword
  RET_VAL := #var2_lword
```

SWAP is described in Chapter 13.2.10 “Swap bytes (SWAP)” on page 570.

10.7.4 Working with the data block registers

The data operands are saved in the data blocks. For the “complete addressing”, specify the data block in which the data operand can be found in the address. If you use the “partial addressing” option, you must open (select) the data block before you can address a data operand (see Chapter 10.7.5 “Partial addressing of data operands” on page 453).

A CPU 1500 emulates two data block registers. These registers contain the numbers of the current data blocks. These registers are called “global data block registers” (abbreviated to: DB registers) and “Instance data block registers” (abbreviated to: DI registers). Table 10.21 shows the operations for data block registers available for STL.

Table 10.21 Functions for data block registers in the statement list

Operation	Operand	Function
OPN OPNDI CDB	Data block Data block –	Opening a data block using the DB register Opening a data block using the DI register Swapping data block registers
L L L L	DBNO DBLG DINO DILG	Loading the number of the data block opened via the DB register Loading the length of the data block opened via the DB register Loading the number of the data block opened via the DI register Loading the length of the data block opened via the DI register

Handling of the registers by the CPU is absolutely equivalent. Each data block can be opened by one of the two registers (also by both simultaneously). The opened data block must be present in the work memory. An opened data block remains “valid” until another data block is opened. Observe that a “complete addressing” sets the DB register to zero.

Open data block (OPN and OPNDI)

The OPN statement opens the specified data block via the DB register, the OPNDI statement via the DI register. The number of the data block is written to the respective data block register. The data block can be addressed absolutely or symbolically or be a tag with parameter type DB_ANY. Examples:

```
OPN   "Motor1_DB"       //Symbolic addressing
OPNDI "Motor2_DB"
OPN   %DB101            //Absolute addressing
OPNDI %DB102
OPN   #Motor1           //Addressing via a block parameter
OPNDI #Motor2
```

Opening a data block via a block parameter with the parameter type DB_ANY allows the transfer of a data block to the called block. You can use this for indirect addressing of a data block (see Chapter 4.3.4 “Indirect addressing of a data block” on page 102).

Absolute or symbolic complete addressing of data operands sets the DB register to zero. The writing of the DI register in a function block has no influence on the symbolic addressing of block parameters and static local data and vice versa.

Opening of a data block is carried out independent of any conditions. It does not influence the result of logic operation and the accumulator contents.

Swapping data block registers (CDB)

The CDB statement exchanges the contents of the data block registers. It is executed independent of conditions and influences neither the status bits nor the other registers.

```
CDB   //Exchange contents of data block registers
```

Loading data block length (L DBLG and L DILG)

The L DBLG statement loads the length of the data block which has been opened via the DB register into accumulator 1. The L DILG statement loads the length of the data block which has been opened via the DI register into accumulator 1. The length is equivalent to the number of data bytes.

```
L     DBLG        //Load length of data block in DB register
L     DILG        //Load length of data block in DI register
```

These load statements transfer the previous contents of accumulator 1 into accumulator 2 in accordance with a “normal” load function. If a data block has not been opened via the associated register, zero is loaded as the length.

Loading data block number (L DBNO and L DINO)

The L DBNO statement loads the number of the data block which has been opened via the DB register into accumulator 1. The L DINO statement loads the number of the data block which has been opened via the DI register into accumulator 1.

```
L     DBNO        //Load number of data block in DB register
L     DINO        //Load number of data block in DI register
```

These load statements transfer the previous contents of accumulator 1 into accumulator 2 in accordance with a “normal” load function. If a data block has not been opened via the associated register, zero is loaded as the number. Example:

```
L     DBNO        //Load data block number
L     10          //and compare,
==I               //if comparison is fulfilled,
JC    Data10      //then jump to the label Data10
```

Direct writing back of the number into a data block register is not possible; you can only influence the data block register using OPN or OPNDI (open data block) and CDB (exchange data block register).

10.7.5 Partial addressing of data operands

Partial addressing of data operands is only possible in the programming language STL. Data operands can only be addressed in absolute mode if the *Optimized block access* block attribute is deactivated in the data block.

“Partial addressing” means that only the data operand is specified in the statement. To do this, it is important that the “correct” data block has been opened in advance. A data block is opened using a data block register, of which there are two types: The global data block register (abbreviated to DB register) and the instance data block register (DI register). Accordingly, there are also two statements: Opening via the DB register and opening via the DI register (see Chapter 10.7.4 “Working with the data block registers” on page 451).

Therefore two data blocks may be open simultaneously. The addressed data block is defined by various data operands: A data operand with the operand ID “DB” is present in a data block opened via the DB register, a data operand with the operand ID “DI” in a data block opened via the DI register (Table 10.22).

Table 10.22 Operand IDs with partially addressed data operands

Operand area	Identifier	Bit (1 bit)	Byte (8 bits)	Word (16 bits)	Doubleword (32 bits)
Data partially addressed via DB register	DB	DBXy.x	DBBy	DBWy	DBDy
Data partially addressed via DI register	DI	DIXy.x	DIBy	DIWy	DIDy

x = bit address, y = byte address

Example: Adding two partially addressed data operands from different data blocks.

```
OPN   %DB12       //Assign default values to DB register
OPNDI "Data13"    //Assign default values to DI register
...
L     %DBW14      //Load data word DW 14 from %DB12
L     %DIW18      //Load data word DW 18 from "Data13"
+I                //add
T     %DBW16      //Store in data word DW 16 in data block %DB12
```

The absolute address of a data operand is shown in the *Offset* column of the block interface once the data block has been compiled.

DB and DI registers cannot be zero for the partial addressing! Observe that the complete addressing of a data operand, for example %DB20.%DBW20 or “Data”.tag, sets the DB register to zero. A subsequent partial addressing, for example L %DBW10, leads to an error.

It is not possible to specify a partially addressed data operand as an actual parameter at a block parameter.

10.7.6 Absolute addressing of temporary local data

The temporary local data are local tags in a code block. Local tags are usually addressed symbolically.

With the programming language STL, absolute addressing is possible for blocks in which the *Optimized block access* attribute is deactivated. The operand ID is “L” (Table 10.23). Example:

```
L     #var_int
L     %LW10            //Temporary local data
+I
T     %LW12
```

If you wish to access local data in absolute mode or if it is essential to do so, you can declare an array at the first position of the temporary local data declaration which reserves the required number of bytes. You can then access this array area in absolute mode. With organization blocks, you define the array following the 20 bytes for the start information.

The absolute address of a temporary local data operand is shown in the *Offset* column of the block interface once the code block has been compiled. After the compilation of the block, the memory requirement for the temporary local data is also displayed in the call structure (double-click on *Program information* in the project tree and select the *Call structure* tab).

Table 10.23 Absolute addressing of temporary local data

Operand area	Operand ID	Bit (1 bit)	Byte (8 bits)	Word (16 bits)	Doubleword (32 bits)
Temporary local data	L	%Ly.x	%LBy	%LWy	%LDy

y = byte address; x = bit address

10.7.7 Working with the address registers

For the indirect addressing of operands, a CPU 1500 emulates two address registers. The address registers are 32-bit wide and are called AR1 and AR2. Table 10.24 shows the statements that are connected to an address register. A graphic representation of the data flow between the operand areas, the address registers AR1 and AR2, and the accumulator 1 is shown in Fig. 10.41. All statements are executed independent of conditions, and do not influence the status bits.

An area pointer is used for the indirect addressing via address registers. Chapter 4.9.2 “Area pointer” on page 135 shows how it is structured.

Note: The writing of the address register AR2 in a function block has no influence on the symbolic addressing of block parameters and static local data and vice versa.

Table 10.24 Overview of address register functions

Operation	Operand	Function
LAR1 LAR1 LAR1 LAR1	 Operand Pointer AR2	Load address register AR1 with the content of accumulator 1 Load address register AR1 with the content of the operand Load address register AR1 with the pointer Load address register AR1 with the content of address register AR2
LAR2 LAR2 LAR2	 Operand Pointer	Load address register AR2 with the content of accumulator 1 Load address register AR2 with the content of the operand Load address register AR2 with the pointer
TAR1 TAR1 TAR1	 Operand AR2	Transfer the content of address register AR1 to accumulator 1 Transfer the content of address register AR1 to the operand Transfer the content of address register AR1 to address register AR2
TAR2 TAR2	 Operand	Transfer the content of address register AR2 to accumulator 1 Transfer the content of address register AR2 to the operand
CAR		Swap the contents of the address registers
+AR1 +AR1	 Pointer	Add the content of accumulator 1 to address register AR1 Add the pointer to address register AR1
+AR2 +AR2	 Pointer	Add the content of accumulator 1 to address register AR2 Add the pointer to address register AR2

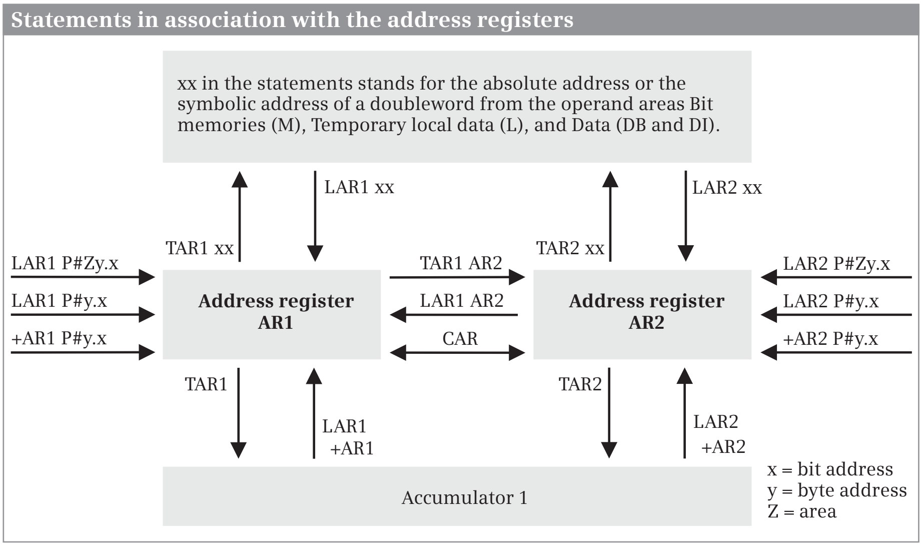

Statements in association with the address registers
xx in the statements stands for the absolute address or the symbolic address of a doubleword from the operand areas Bit memories (M), Temporary local data (L), and Data (DB and DI).
LAR1 xx
TAR1 xx
LAR2 xx
TAR2 xx
LAR1 P#Zy.x
LAR1 P#y.x
+AR1 P#y.x
Address register AR1
TAR1 AR2
LAR1 AR2
CAR
Address register AR2
LAR2 P#Zy.x
LAR2 P#y.x
+AR2 P#y.x
TAR1
LAR1
+AR1
TAR2
LAR2
+AR2
Accumulator 1
x = bit address
y = byte address
Z = area

Fig. 10.41 Statements for working with address registers

Loading into an address register

The statement LAR1 loads an area pointer into the address register AR1. The statement LAR2 loads an area pointer into the address register AR2. You can select an area-internal or cross-area pointer or a tag of doubleword width from the operand areas “Bit memories” (M), “Temporary local data” (LD) or “Data” (DB and DI) as the source. The content of the tags must correspond to the format of an area pointer.

If you do not specify an operand, LAR1 or LAR2 loads the content of accumulator 1 into address register AR1 or AR2.

Using the LAR1 AR2 statement, you copy the content of address register AR2 into address register AR1.

Transferring from an address register

The statement TAR1 transfers the complete area pointer from the address register AR1, the statement TAR2 transfers the complete area pointer from the address register AR2. You can select a tag of doubleword width from the operand areas “Bit memories” (M), “Temporary local data” (LD) or “Data” (DB and DI) as the target.

If you do not specify a tag, TAR1 or TAR2 transfers the content of address register AR1 or AR2 into accumulator 1. The previous content of accumulator 1 is shifted into accumulator 2 during this procedure; the previous content of accumulator 2 is lost.

Using the TAR1 AR2 statement, you copy the content of address register AR1 into address register AR2.

Swap address register contents

The CAR statement swaps the contents of address registers AR1 and AR2. Fig. 10.42 (top) shows an example of application of the statement.

Addition to address register

A value can be added to the address registers, e.g. to increment the address of an operand in program loops each time the loop is executed. You can either enter the value as a constant (as area-internal pointer) for the statement, or it is located in the right word of accumulator 1. The type of pointer present in the address register (area-internal or cross-area) and the operand area are retained.

The +AR1 P#y.x and +AR2 P#y.x statements add a pointer to the specified address register. Note that the maximum size of the area pointer is P#4095.7 with these statements. If a value larger than P#4095.7 is present in the accumulator, the number is interpreted as a fixed-point number in two's complement and subtracted. Fig. 10.42 (middle) shows an example of application of the statements.

The +AR1 and +AR2 statements interpret the value present in accumulator 1 as a number in integer format, expand it with the correct sign to 24 bits, and add it to the content of the address register. A pointer can also be reduced in this manner.

Example of swapping address register contents

```
      LAR1  P#M100.0
      LAR2  P#DBX200.0
      OPN   %DB20
      A     "Swap"
      JC   TAU
      CAR
TAU:  L     D[AR1,P#0.0]
      T     D[AR2,P#0.0]
      L     D[AR1,P#4.0]
      T     D[AR2,P#4.0]
```

8 bytes of data are transferred between the memory area starting at %MB100 and the data area starting at %DB20.DBB200.
The transfer direction defines the tag *“Swap”*. If *“Swap”* has signal state “0”, the contents of the address register are swapped.

If you wish to transfer data between two data blocks in this manner, also load the two data block registers together with the address registers (using OPN and OPNDI) and swap the contents where appropriate using the TDB statement.

Example of adding a pointer to the address register

```
      OPN   "Data14"
      LAR1  P#DBX20.0
      LAR2  P#M10.0
      L     #Number_data
Loop: T     #Loop_counter
      L     #Reference_value
      L     W[AR1,P#0.0]
      >I
      =     [AR2,P#0.0]
      +AR1  P#2.0
      +AR2  P#0.1
      L     #Loop_counter
      LOOP  Loop
```

A data area of length *#Number_ data* in data block *“Data14”* is compared with the *#Reference_value* tag word-by-word starting at data word *%DBW20*. If the reference value is larger than the value in the array, a memory bit is to be set to “1” starting at bit memory %M10.0, otherwise to “0”.

Example of adding the accumulator content to the address register

```
OPN   %DB14
LAR1  %MD220
L     %MB18
SLW   3
+AR1
L     0
T     DBD[AR1,P#0.0]
T     DBD[AR1,P#4.0]
T     DBD[AR1,P#8.0]
T     DBD[AR1,P#12.0]
```

In data block %DB14, the 16 bytes are to be deleted whose addresses are calculated from the pointer in bit memory doubleword %MD220 and a (byte) offset in memory byte %MB18. Prior to addition to AR1, the content of %MB18 must be aligned (SLW 3).

Fig. 10.42 Examples of register-indirect addressing with STL

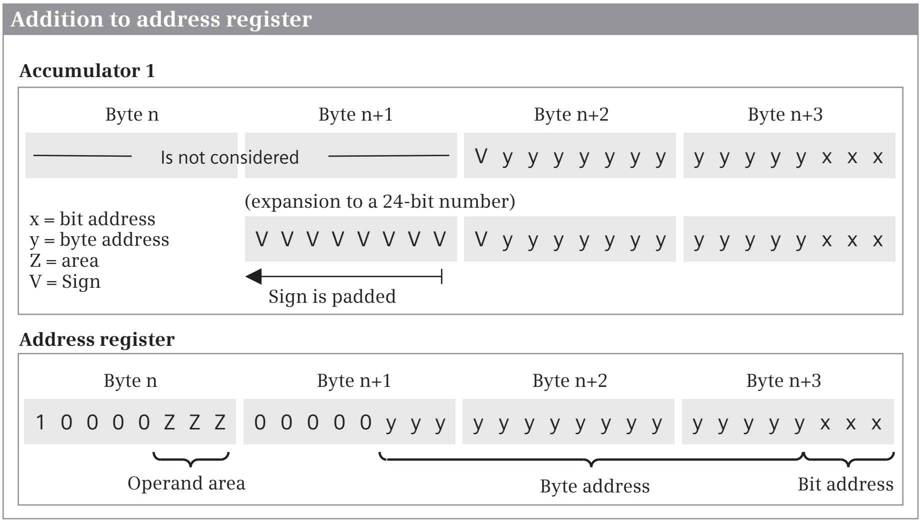


Fig. 10.43 Addition to address register

Upward or downward violation of the maximum range of the byte address (0 to 65 535) has no further effects: The highest bits are “truncated” (Fig. 10.43).

Note that the bit address is present in bits 0 to 2. If you wish to already increment the byte address in accumulator 1, you must add starting with bit 3 (shift the value to the left by 3 digits). Fig. 10.42 (bottom) shows an example of application of the statement.

10.7.8 Memory-indirect addressing

“Address register” for memory-indirect addressing

Indirect memory addressing uses an operand from the operand areas “Bit memories” (M), “Temporary local data” (LD) or “Data” (DB and DI) as “address registers”. For a function block, block parameters and static local data can also be used as “address registers”. If the “address register” is located in a data block – this is the case for data blocks, function block parameters, and static local data – the attribute *Optimized block access* must be deactivated. A word or doubleword is required depending on the operands to be addressed (see below).

A bit memory word or a bit memory doubleword can be used generously as an “address register” in the user program since the bit memories are global tags.

You can use a word or doubleword from the temporary local data if the content of the word or doubleword is not used beyond execution in the block.

Use of a data operand as address register is partial addressing. The data operand is only “valid” for as long as the associated (“correct”) data block is open. Refer to Chapter 10.7.5 “Partial addressing of data operands” on page 453 for information on what you must observe.

Indirectly addressable operands

The memory-indirect addressable operands can be divided into two categories: Operands which can have a bit address, and operands which never have a bit address.

Operands which can have a bit address are located in the following operand areas: inputs (I), outputs (Q), peripheral inputs and outputs (I:P and Q:P), data (DB and DI), and temporary local data (L). These operands require an area pointer as address which contains the bit and byte address – even if the operand to be addressed is of word width, for example, and has no bit address. The structure of this area-internal pointer is described in Chapter 4.9.2 "Area pointer" on page 135. Refer to Chapter 10.7.5 "Partial addressing of data operands" on page 453 for information on what you must observe when addressing data operands.

Memory-indirect addressable operands which never have a bit address are SIMATIC timer functions (T), SIMATIC counter functions (C), data blocks (DB), functions (FC), and function blocks (FB). With indirect addressing of these operands, an operand of word width containing a number as the address is sufficient as the "address register".

A data tag can only be indirectly accessed if the *Optimized block access* attribute is deactivated in the data block. Data in the block interface of function blocks with the *Optimized block access* attribute activated can only be accessed if the retentivity of the tag is set to *Set in IDB*. It is not possible to access the temporary local data of a block in which the *Optimized block access* attribute is activated.

Memory-indirect addressing with an area pointer

The area pointer required for memory-indirect addressing is always an area-internal pointer, i.e. it always consists of byte and bit address. You must specify 0 as the bit address when addressing a digital operand.

You can use the memory-indirect addressing with area pointer for all binary operands in conjunction with the binary logic operations and memory functions, and for all digital operands in conjunction with the load and transfer functions. The upper example in Fig. 10.44 uses the "*Pointer*" tag as address register. The tag could be, for example, the bit memory doubleword %MD200 with the data type DINT.

Memory-indirect addressing with a number

The number required for memory-indirect addressing is an unsigned 16-bit fixed-point number. Memory-indirect addressing with a number can be applied in conjunction with SIMATIC timer and counter functions and with the block types DB, FC, and FB.

You can open a data block via the DB register (OPN [..]) or via the DI register (OPNDI [..]). If there is zero in the address word, the CPU performs an NOP operation and the current data block is no longer opened. A subsequent partially addressed access – e.g. with L %DBB0 – generates an addressing error.

Examples of memory-indirect addressing with an area pointer	
L P#128.0 T "Pointer"	The address register “Pointer” is loaded with byte address 128. The bit address is 0.
L IW["Pointer"]	The statement L %IW128 is executed.
L "Pointer" L 2 SLD 3 +D T "Pointer"	The byte address is incremented by 2 in the address register. Since the bit address is in the bottom 3 bits, the value is shifted by 3 to the left. One can also immediately add a value multiplied by 8 – in this case 16 – to the address register.
T QW["Pointer"]	The statement T %QW130 is executed.
L P#54.2 T "Pointer"	The address register “Pointer” is loaded with byte address 54 and bit address 2.
A I["Pointer"]	The statement A %I54.2 is executed.
L "Pointer" L 1 +D T "Pointer"	The bit address is incremented by 1 in the address register.
= M["Pointer"]	The statement = %M54.3 is executed.
Examples of memory-indirect addressing with a number	
L 108 T "Number"	The address register “Number” is loaded with the value 108.
CU C["Number"]	The statement CU %C108 is executed.
L "Number" L 10 +D T "Number"	The value is incremented by 10 in the address register.
R T["Number"]	The statement R %T118 is executed.
OPN DB["Number"] OPN DI["Number"]	The statements OPN %DB118 and OPN %DI118 are executed.
UC FC["Number"] CC FB["Number"]	The statements UC %FC118 and CC %FB118 are executed.

Fig. 10.44 Examples of memory-indirect addressing with STL

You can indirectly address the call of code blocks with UC FC [..] and CC FC [..] or UC FB [..] and CC FB [..]. The call with UC or CC is simply a change to another block; a transfer of block parameters or the opening of an instance data block does not take place.

The lower example in Fig. 10.44 uses the “*Number*” tag as address register. The tag could be, for example, the bit memory word %MW204 with the data type INT.

10.7.9 Register-indirect addressing

Register-indirect addressing uses one of the address registers AR1 or AR2 in order to determine the address of the operand.

Register-indirect addressing is possible in two versions: With *area-internal* register-indirect addressing, the address in the address register varies within an operand area. With *cross-area* register-indirect addressing, the variable address also comprises the operand area.

With register-indirect addressing, an offset is specified in addition to the address register, and is added to the content of the address register during execution of the operation without changing the content of the register. This offset has the format of an area-internal pointer. You must always specify it, and only as a constant. With indirectly addressed digital operands, this offset must have bit address 0. The maximum value is P#8191.7.

Refer to Chapter 4.9.2 “Area pointer” on page 135 for information on the structure of the area pointers used for register-indirect addressing. The statements required for working with the address registers are described in Chapter 10.7.7 “Working with the address registers” on page 454.

Indirectly addressable operands

With register-indirect addressing, operands located in the following operand areas can be accessed: Inputs (I), Outputs (Q), Peripheral inputs and outputs (I:P and Q:P), Bit memories (M), Temporary local data (L), and Data (DB and DI, function block parameters and static local data).

A data tag can only be indirectly accessed if the *Optimized block access* attribute is deactivated in the data block. Data in the block interface of function blocks with the *Optimized block access* attribute activated can only be accessed if the retentivity of the tag is set to *Set in IDB*. It is not possible to access the temporary local data of a block in which the *Optimized block access* attribute is activated.

Refer to Chapter 10.7.5 “Partial addressing of data operands” on page 453 for information on what you must observe when addressing data operands.

Area-internal, register-indirect addressing

With area-internal, register-indirect addressing, the operand area is assigned when addressing and cannot be changed. The pointers located in address registers AR1 and AR2 can be area-internal or cross-area. The operand area which is present in the address specification is always used.

The examples in Fig. 10.45 (top) show area-internal, register-indirect addressing.

Examples of area-internal register-indirect addressing		
LAR1	P#10.0	The address register AR1 is loaded with byte address 10. The bit address is 0.
L L	MW[AR1,P#0.0] MW[AR1,P#2.0]	The statement L %MW10 is executed. The statement L %MW12 is executed.
+AR1	P#20.0	The byte address is incremented by 20 in address register AR1.
L L	MW[AR1,P#0.0] MW[AR1,P#4.0]	The statement L %MW30 is executed. The statement L %MW34 is executed.
LAR2	P#A16.3	The address register AR2 is loaded with the pointer to output bit %Q16.3.
A	I[AR2,P#0.0]	The statement A %I16.3 is executed.
+AR2	P#0.1	The bit address is incremented by 1 in address register AR2.
=	M[AR2,P#4.0]	The statement = %M20.4 is executed.
Examples of cross-area register-indirect addressing		
LAR1	P#M64.0	The address register AR1 is loaded with the pointer to memory bit %M64.0.
L L	W[AR1,P#0.0] W[AR1,P#2.0]	The statement L %MW64 is executed. The statement L %MW66 is executed.
+AR1	P#12.0	The byte address is incremented by 12 in address register AR1.
L L	B[AR1,P#0.0] B[AR1,P#4.0]	The statement L %MB76 is executed. The statement L %MB80 is executed.
LAR2	P#DB32.0	The address register AR2 is loaded with the pointer to data bit %DB32.0.
T A L	D[AR2,P#0.0] [AR2,P#0.1] MW[AR2,P#4.0]	The statement T %DBD32 is executed. The statement A %DBX32.1 is executed. The statement L %MW36 is executed.

Fig. 10.45 Examples of register-indirect addressing with STL

Cross-area register-indirect addressing

With cross-area register-indirect addressing, the operand area is located together with the byte and bit address in the address register. Only the operand width is present in the addressing statement, no information for a bit, “B” for a byte, “W” for a word, and “D” for a doubleword. The pointer present in address register AR1 or AR2 must be cross-area.

The examples in Fig. 10.45 (bottom) show cross-area, register-indirect addressing.

10.7.10 Direct access to complex local tags

You can access local tags with elementary data types using “normal” STL statements. Local tags with structured data types or block parameters of type POINTER or ANY cannot be handled as an entity. To process such tags, one initially determines the start address at which the tag is saved, and then processes parts of the tag using indirect addressing. In this way you can also process block parameters with structured data types.

For the blocks that are involved in the direct access, the attribute *Optimized block access* must be deactivated.

Loading of the start address of tags is not possible for tags from the following operand areas: Inputs (I), Outputs (Q), Peripheral inputs and outputs (I:P and Q:P), Bit memories (M), Data operands (DB and DI) as well as of block parameters with the parameter type VARIANT.

Loading tag address

You obtain the start address of a local tag using the statements

```
L      P##name
LAR1   P##name
LAR2   P##name
```

where *#name* is the name of the local tag. These statements load a cross-area pointer into accumulator 1 or into address register AR1 or AR2. The area pointer contains the address of the first byte of the tag. Depending on the code block used, the tag areas specified in Table 10.25 are approved for *#name*.

In the case of functions (FC), the address of a block parameter cannot be directly loaded into an address register. In this case you can use the path via accumulator 1, for example with L P##name and LAR1.

In the case of function blocks without multi-instance capability, the absolute address of the local tag is loaded. Function blocks without multi-instance capability can only be called as a single instance with its own data block. They can only be created via a source file with the keyword CODE_VERSION1.

In the case of function blocks with multi-instance capability, the absolute address for the static local data and the block parameters is loaded relative to the start of the local instance data. If you wish to determine the absolute address of the tag in

Table 10.25 Permissible operands for loading the tag start address

Operation	*#name* is a	OB	FC	FB with multi-instance capability	FB without multi-instance capability[2]
L P##*name*	Temporary local data	x	x	x	x
	Static local data	–	–	x[1]	x
	Block parameter	–	x	x[1]	x
LAR*n* P##*name*	Temporary local data	x	x	x	x
	Static local data	–	–	x[1]	x
	Block parameter	–	–	x[1]	x

[1] Tag address relative to address register AR2
[2] Only possible when generating from a source file

the data block with multi-instance capability, you must add the *area-internal* pointer of address register AR2 to the loaded tag address.

You can also apply the loading of a tag address to block parameters. Chapters 10.7.11 “Data storage of the block parameters of a function (FC)” on page 465 and 10.7.12 “Data storage of the block parameters of a function block (FB)” on page 467 describes how the block parameters are saved in the memory and their respective contents.

```
TAR2
LAR1    P##name
+AR1
```

Load the start address of the *#name* tag into address register AR1.

```
TAR2
AD         16#00FF_FFFF
L           P##name
+D
```

Load the start address of the *#name* tag into accumulator 1.

```
Static
   First_name : 'Elisabeth'
...

LAR1     P##First_name
TAR2
+AR1
L     ’Mari’
T     D[AR1,P#2.0]
L     ’on’
T     W[AR1,P#6.0]
L     6
T     B[AR1,P#1.0]
```

Processing of a tag with structured data type:
Load the start address of *#First_name* tag into address register AR1, fetch the offset address from AR2 into accumulator 1, and add to the content of address register AR1. The start address of *#First_name* is now present in address register AR1.
Write ’Marion’ into the *#First_name* tag starting at byte 2 and update the current length of the STRING tag in byte 1 to a value of 6.

Fig. 10.46 Examples of loading a tag address

The two examples at the top in Fig. 10.46 show the program in a function block with multi-instance capability for loading the tag start address into address register AR1 or accumulator 1. Digital operation AD is only required if the operand area is to be hidden in the address. In the bottom example, the tag *#First_name* is occupied by a different value.

10.7.11 Data storage of the block parameters of a function (FC)

The program editor stores a block parameter of a function as a cross-area pointer in the block code following the actual call statement and therefore every block parameter requires a doubleword in the memory. The pointer points to the actual parameter itself depending on the type of data and declaration, to a copy of the actual parameter in the temporary local data of the calling block (the program editor creates this), or to a pointer in the temporary local data of the calling block which in turn points to the actual parameter (Table 10.26). Exception: With the parameter types TIMER, COUNTER, BLOCK_FC, BLOCK_FB and DB_ANY, the pointer is a 16-bit number located in the left word of the block parameter.

Table 10.26 Parameter storage for functions

Data type	INPUT	IN_OUT	OUTPUT
	The parameter is an area pointer to a		
Elementary	Value	Value	Value
Structured	DB pointer	DB pointer	DB pointer
TIMER, COUNTER, BLOCK_FC, BLOCK_FB, DB_ANY	Number	–	–
POINTER	DB pointer	DB pointer	DB pointer
ANY	ANY pointer	ANY pointer	ANY pointer

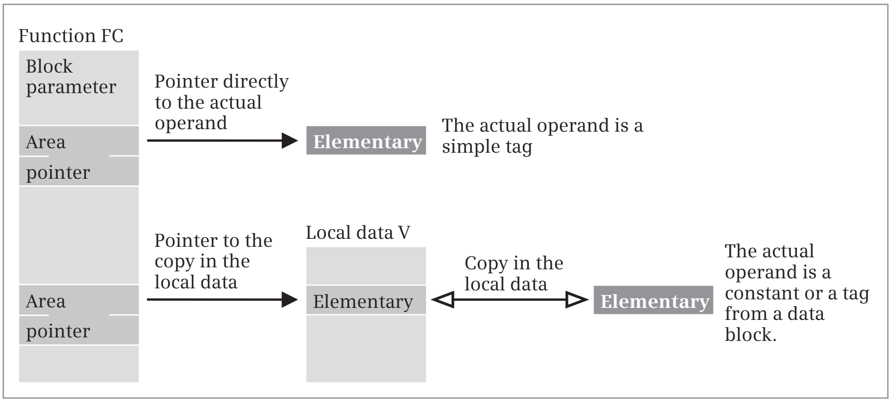
With elementary data types, the block parameter points directly to the actual operand (Fig. 10.47). With the area pointer as block parameter, however, it is not possible to access any constants or operands located in data blocks. Therefore, when compiling the block, the program editor copies the value of a constant or an actual operand present in a data block (and completely addressed) into the temporary local data of the calling block and points the area pointer to this. This operand area is named V (temporary local data of preceding block, V area).

Copying into the V area is carried out prior to the actual FC call in the case of input and in/out parameters, but following the call in the case of in/out and output parameters and thus also with the function value. The principle therefore also applies that you can only scan input parameters and only write output parameters. For example, if you transfer a value to an input parameter with a completely addressed data operand, the value is stored in the temporary local data of the preceding block and forgotten, since copying into the “actual” tag in the data block no longer takes place.

Parameter transfer for functions (FC)

Pointer to the actual operand or its value

A block parameter of a function (FC) is a 32-bit area pointer. If the block parameter has an elementary data type and if the actual operand is a simple operand, then the pointer points directly to the actual operand. A constant as an actual operand or a completely addressed operand cannot be accessed with a 32-bit pointer. In such cases the program editor copies the value of the constant or data operand into the temporary local data of the preceding block, and positions the pointer of the block parameter to this value.


Function FC
Block parameter
Pointer directly to the actual operand
Area pointer
Elementary
The actual operand is a simple tag
Local data V
Pointer to the copy in the local data
Copy in the local data
Area pointer
Elementary
Elementary
The actual operand is a constant or a tag from a data block.

Pointer to a further pointer

If the block parameter has a structured data type or the parameter type POINTER, the program editor creates a 48-bit pointer to the actual parameter in the temporary local data of the preceding block. The block parameter of the FC then points to this DB pointer.
If the block parameter has the parameter type ANY, the program editor creates an 80-bit pointer to the actual parameter in the temporary local data of the preceding block.
Exception: The actual parameter already has the data type ANY and is in the temporary local data of the preceding block. A further pointer is not created in this case.

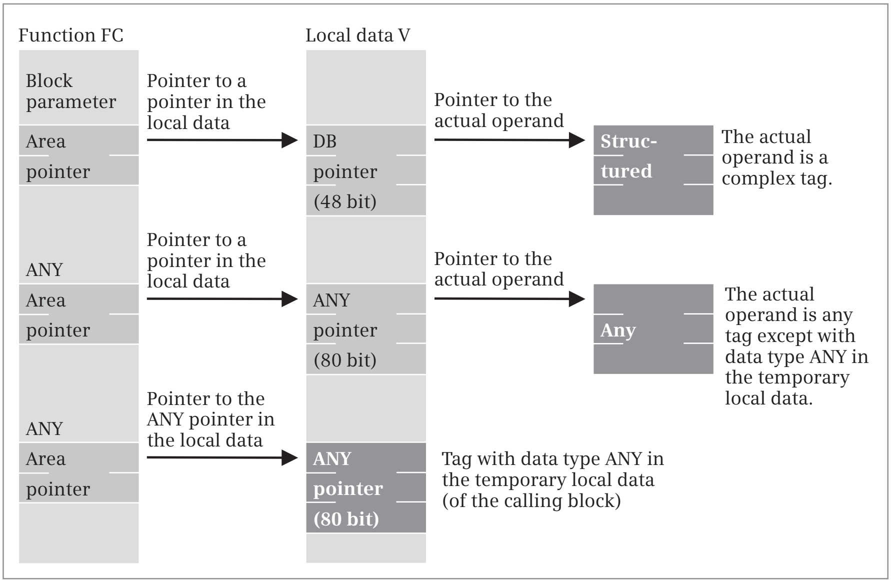

Function FC
Local data V
Block parameter
Pointer to a pointer in the local data
Area pointer
DB pointer (48 bit)
Pointer to the actual operand
Struc-tured
The actual operand is a complex tag.
ANY
Pointer to a pointer in the local data
Area pointer
ANY pointer (80 bit)
Pointer to the actual operand
Any
The actual operand is any tag except with data type ANY in the temporary local data.
ANY
Pointer to the ANY pointer in the local data
Area pointer
ANY pointer (80 bit)
Tag with data type ANY in the temporary local data (of the calling block)

Fig. 10.47 Parameter transfer for functions (FC)

The same applies to loading a corresponding output parameter: Since copying from the “actual” tag from the data block into the V area does not take place, you load an (indefinite) value from the V area in this case.

As a result of the copying process, you **must** write an output parameter with a value and thus also a function value defined with an elementary data type in the block if a completely addressed data operand is envisaged or could be used as the actual parameter. If you do not assign a value to the output parameter, e.g. because you leave the block beforehand or jump beyond the program position, the local data is not supplied either. It then has the value which it had “by chance” prior to the block call. The output parameter is then written with this “undefined” value. Note in this context that certain operations, for example retentive setting, do not write a value to the operand if they are processed with the result of logic operation “0”.

With structured data types, the actual parameters are located either in a data block or in the V area. Since an area pointer cannot access an actual operand in a data block, the program editor creates a DB pointer in the V area when compiling which then points to the actual operand in the data block (DB No. <> 0) or to the V area (DB No. = 0). The DB pointers for all declaration types in the V area are created before the “actual” FC call.

With the parameter types TIMER, COUNTER, BLOCK_FC, BLOCK_FB and DB_ANY, a number is present instead of the area pointer in the block parameter (16 bits left-justified in the 32-bit parameter).

The parameter type POINTER is handled just like a structured data type.

With the parameter type ANY, the program editor creates a 10-byte long ANY pointer in the V area which can then point to any tag. The principle is the same as with the structured data types.

An exception is made by the program editor if you apply an actual parameter to a block parameter of type ANY where the actual parameter is in the temporary local data and is of type ANY. In this case the program editor does not create any further ANY pointers but applies the area pointer (the block parameter) directly to the actual parameter. In this case, the ANY pointer can be changed during runtime, see Chapter 4.3.5 “Indirect addressing with an ANY pointer” on page 103.

10.7.12 Data storage of the block parameters of a function block (FB)

The program editor stores the block parameters of a function block in the instance data of the call. With a function block call, the program editor generates statement sequences which copy the values of the actual parameters prior to the actual call into the instance data and back again from the instance data to the actual parameters following the call. You do not see these statement sequences when viewing the compiled block, you only notice this indirectly because memory space is occupied.

The block parameters are present in the instance data either as a value, a 16-bit number, or a pointer to the actual parameter (Table 10.27). When storing as a value, the memory space required depends on the data type of the block parame-

Table 10.27 Parameter storage for function blocks

Data type	INPUT	IN_OUT	OUTPUT
Elementary	Value	Value	Value
Structured	Value	DB pointer	Value
TIMER, COUNTER, BLOCK_FC, BLOCK_FB, DB_ANY	Number	–	–
POINTER	DB pointer	DB pointer	–
ANY	ANY pointer	ANY pointer	–

ter; the number occupies 2 bytes, a DB pointer occupies 6 bytes, and an ANY pointer occupies 10 bytes.

The relationships between block parameters, instance data assignment, and actual parameters are shown in Fig. 10.48.

Copying of block parameters saved as values in the instance data is carried out prior to the “actual” FB call by means of statement sequences for input and in/out parameters, but following the call in the case of in/out and output parameters. The principle therefore also applies that you can only scan input parameters and only write output parameters. For example, if you transfer a (new) value to an input parameter, the current value of the actual parameter is lost. If you load an output parameter, you load the (old) value in the instance data block and not that of the actual parameter.

Because the block parameters are saved in the instance data, they need not be supplied each time the function block is called. If no values are supplied, the program uses the “old” value of the input or in/out parameter, or you fetch the value of the output parameter at a different position later in the program. You can address the tags in the instance data outside the function block just like the tags in a global data block (with an instance data block) or like a STRUCT tag (with a local instance).

If you apply a temporary local tag with data type ANY to an ANY parameter, the program editor copies the content of this tag into the ANY pointer (into the block parameter) in the instance data.

10.7.13 Data storage of a local instance in a multi-instance

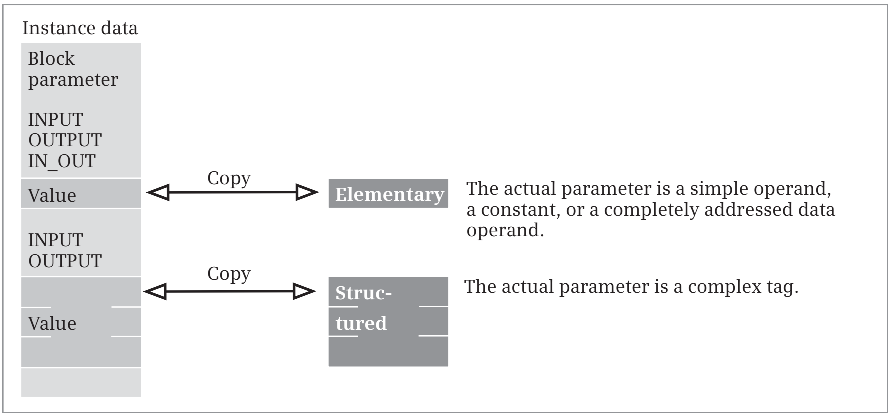
Function blocks require a data block – the instance data block – in order to save the block parameters and the static local data. This can be a separate data block or – if the call of the function block is within a function block – the instance data block of the calling function block. You define the data block in which the instance data is saved when calling the function block:

- If you select *Single instance*, a separate data block is generated for the call of the function block.
- If you select *Multi instance*, the data of the called function block is inserted as a “local instance” in the instance data block of the calling function block.

Parameter transfer with function blocks

Value in the instance data

A block parameter of a function block is located in the instance data of the call. If the block parameter has an elementary data type, the value of the actual parameter is copied into the instance data or from the instance data to the actual parameter. The same applies to an input or output parameter with structured data type.
If the block parameter has a data type TIMER, COUNTER, BLOCK_xx or DB_ANY, the number of the timer or counter function or of the block is present in the instance data.

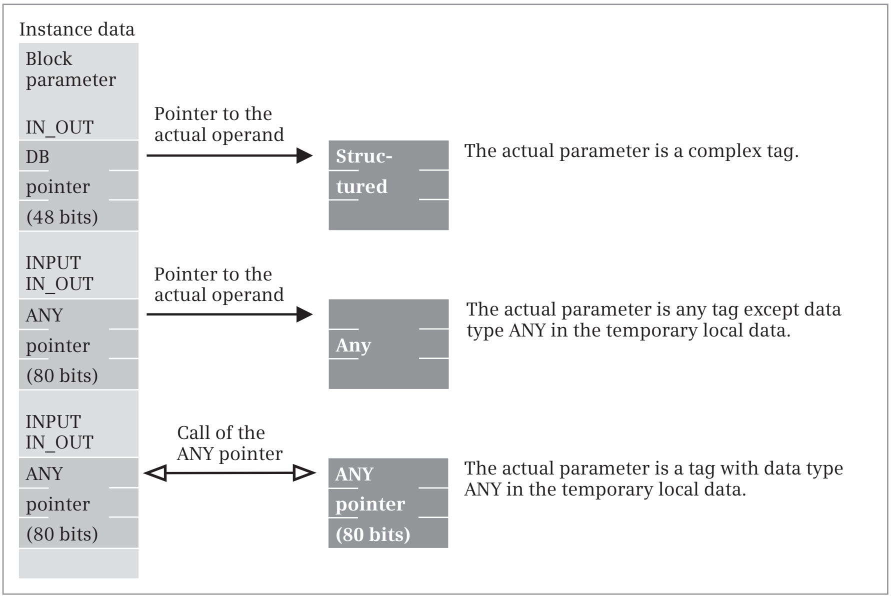


The actual parameter is a simple operand, a constant, or a completely addressed data operand.

The actual parameter is a complex tag.

Pointer in the instance data

If an in/out parameter has a structured data type, a 48-bit pointer to the actual parameter is created in the instance data.
If a block parameter has the data type ANY, an ANY pointer to the actual parameter is created in the instance data. Exception: if the actual parameter also has the data type ANY and is located in the temporary local data, it is copied into the instance data.



The actual parameter is a complex tag.

The actual parameter is any tag except data type ANY in the temporary local data.

The actual parameter is a tag with data type ANY in the temporary local data.

Fig. 10.48 Parameter transfer with function blocks (FB)

The data of a local instance is a subset of the static local data of the calling function block (Fig. 10.49). The local instance has a name which you define during programming of the statement. In a function block you can program several local instances of the same function block by defining different instance names for each of them.

The individual components of a local instance are shown in the instance data block in *Expanded mode*. You can address the components of a local instance from the calling function block as a static local tag using *#Instance_name.Component_name* or from any block as a global data tag using *"Data_block_name". Instance_name.Component_name*.

Function blocks with local instances can again be a local instance. In this manner you can "nest" up to eight instances.

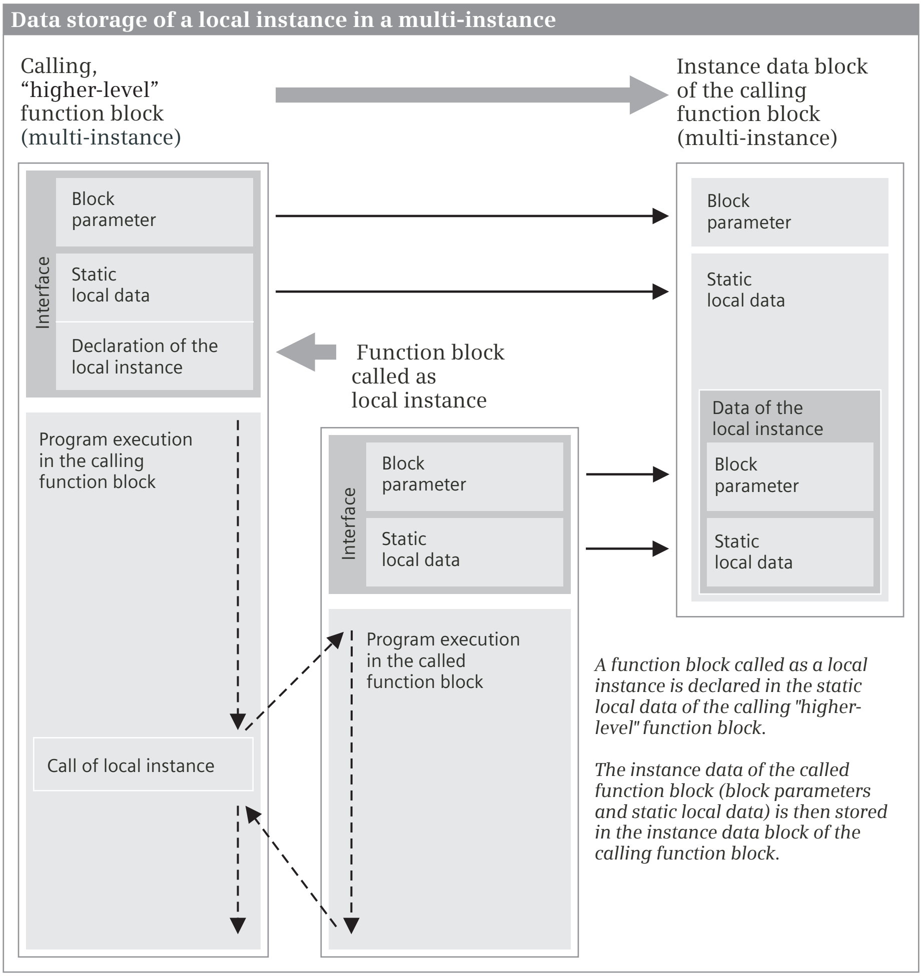

Data storage of a local instance in a multi-instance
Calling, "higher-level" function block (multi-instance)
Instance data block of the calling function block (multi-instance)
Interface
Block parameter
Static local data
Declaration of the local instance
Function block called as local instance
Program execution in the calling function block
Call of local instance
Interface
Block parameter
Static local data
Program execution in the called function block
Block parameter
Static local data
Data of the local instance
Block parameter
Static local data
A function block called as a local instance is declared in the static local data of the calling "higher-level" function block.
The instance data of the called function block (block parameters and static local data) is then stored in the instance data block of the calling function block.

Fig. 10.49 Data storage of a local instance in a multi-instance

10.7.14 Null instructions

Null instructions result in no response whatsoever by the control processor during execution. Table 10.28 shows the null instructions available with STL.

Table 10.28 Null instructions with STL

Operation	Operand	Function
BLD	Number	Controls the construction of an LAD or FBD representation
NOP NOP	0 1	Statement with memory content W#16#0000 Statement with memory content W#16#FFFF

The program elements catalog contains the null instructions under *Basic instructions > Basic instructions > Further instructions*.

NOP instructions

You can use the NOP 0 (bit pattern 16-times “0”) and NOP 1 (bit pattern 16-times “1”) statements to enter a statement which has no effect. Note that the nil operations occupy memory space (2 bytes) and have a command runtime.

Example: A statement must always be present for a jump label. Use NOP 0 if you wish to have nothing executed at an entry in your program.

```
      A   %I 1.0
      JC  MXX1
      ...
MXX1: NOP 0
      ...
```

An empty line for clearer commenting of programs can be entered more effectively using an (empty) line comment (no memory requirements in user memory and no runtime losses since no code is sent).

Program display (BLD instruction)

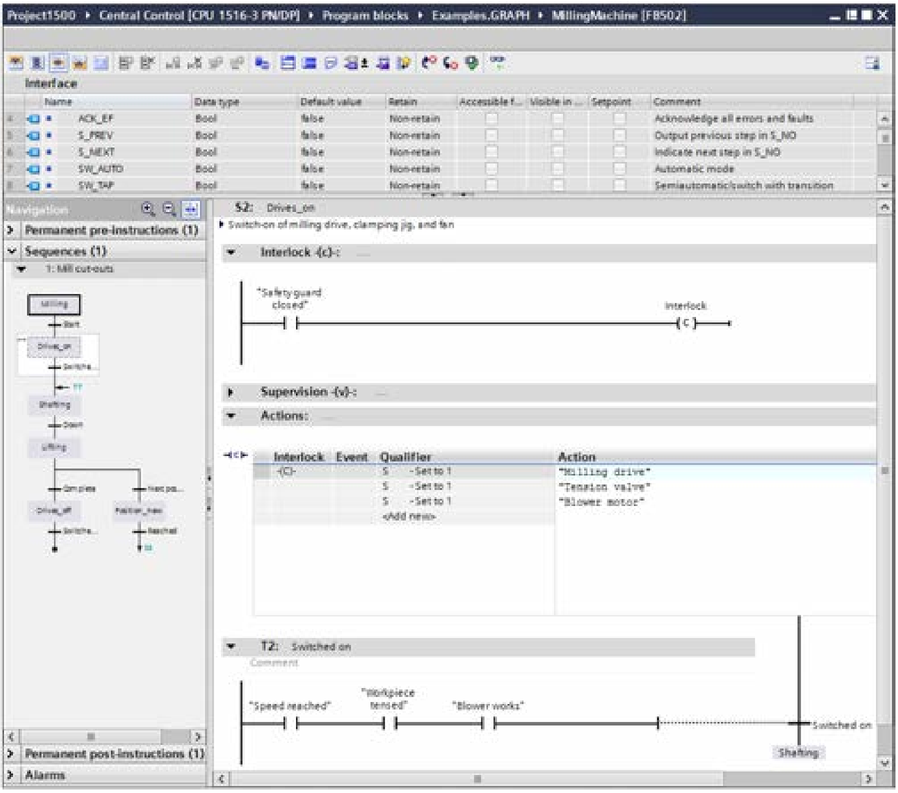
The program editor uses BLD *nnn* display construction statements to integrate information for decompilation into the program.

11 S7-GRAPH sequential control

11.1 Introduction

11.1.1 What is a sequential control?

Static assignment of the input signals to the outputs (as with logic controls) does not dominate in the case of sequential controls, but their time sequence. The control procedures executed in succession are divided into sequence steps, or steps for short. A step contains one or more actions such as switch motor on or off. Only the actions of an active (processed) step are carried out. Progression to the next step is carried out by means of transitions (step enabling conditions). The transition can be process-dependent, e.g. as a result of signals from the controlled machine or plant, or time-dependent, e.g. following expiry of a delay time.



Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ Program blocks ▸ Examples.GRAPH ▸ MillingMachine [FB502]

Interface

	Name	Data type	Default value	Retain	Accessible f...	Visible in ...	Setpoint	Comment
4	ACK_EF	Bool	false	Non-retain	□	□	□	Acknowledge all errors and faults
5	S_PREV	Bool	false	Non-retain	□	□	□	Output previous step in S_NO
6	S_NEXT	Bool	false	Non-retain	□	□	□	Indicate next step in S_NO
7	SW_AUTO	Bool	false	Non-retain	□	□	□	Automatic mode
8	SW_TAP	Bool	false	Non-retain	□	□	□	Semiautomatic/switch with transition

S2: Drives_on

Switch-on of milling drive, clamping jig, and fan

Interlock -(c)-:

Supervision -(v)-:

Actions:

Interlock	Event	Qualifier	Action
-(C)-		S - Set to 1	"Milling drive"
		S - Set to 1	"Tension valve"
		S - Set to 1	"Blower motor"
		<Add new>	

T2: Switched on

Comment

Fig. 11.1 Example of representation of a step in a sequencer

A sequential control is started at an initialization step; several can be present in a sequential control. These are followed by alternate transitions and steps in a linear sequencer. Branches are also possible in addition to the linear progression where a step is followed by a single step: With alternative branching (OR branch), only one of the following partial sequences is processed, with simultaneous branching (AND branch), all following partial sequences are processed.

A sequential control can contain several independent sequencers.

Fig. 11.1 shows an example of the working window of the GRAPH editor. The sequencer is shown in the GRAPH navigation on the left side and the working area shows the step S2 with the transition T2 as selected in the navigation.

11.1.2 Properties of a sequential control

A sequential control consists of a function block GRAPH and a data block GRAPH_DB. The function block controls the sequencer; the data block is the instance data block of the sequencer FB and contains the structure of the sequencer and the associated data (Fig. 11.2).

The function block first processes the permanent instructions which precede the sequencers and are processed in each cycle. The active sequence steps (of which

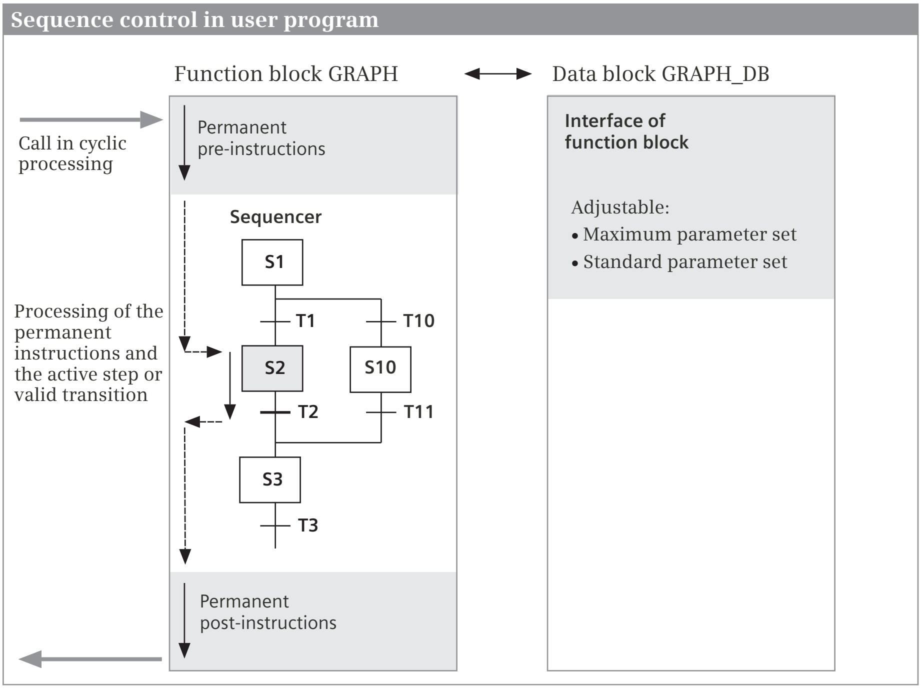

Sequence control in user program
Function block GRAPH
Data block GRAPH_DB
Call in cyclic processing
Permanent pre-instructions
Interface of function block
Adjustable:
• Maximum parameter set
• Standard parameter set
Sequencer
S1
T1
T10
S2
S10
T2
T11
S3
T3
Processing of the permanent instructions and the active step or valid transition
Permanent post-instructions

Fig. 11.2 Components and properties of a sequential control

there can be several) and the following transitions are subsequently processed. The series-connected permanent instructions are processed following the sequencers.

The instance data block contains the interface of the function block and the status data for the sequential control. In the main menu you can select between two parameter sets under *Options* > *Settings* and *PLC programming* > *GRAPH*: If the *Maximum interface parameters* checkbox is activated, all parameters are displayed; if the option is deactivated, only the standard parameter set is displayed. You can manually add individual input or output parameters to the interface at any time.

11.1.3 Program for a sequential control, quantity framework

The GRAPH function block contains the program for a sequential control. You can program any function block using GRAPH, and also several ones – each with their own sequential control – in a user program.

The instance data block of a GRAPH function block contains the structure data of the sequential control.

- ▷ It can accommodate up to 250 steps and transitions. (One step and one transition are only handled as a pair here.)
- ▷ Up to 249 simultaneous branches are possible.
- ▷ Up to 125 alternative branches are possible.

The structure of a sequential control is defined during configuration. A sequential control can contain several sequencers within the scope of the quantity framework specified above. Limitation of the quantity framework, especially for the simultaneous branches, may be meaningful for runtime reasons.

11.1.4 Operating modes

The GRAPH sequential control allows the following operating modes:

- ▷ In **automatic mode**, a switch is made from one sequence step to the next when the transition between them is fulfilled.
- ▷ In **semiautomatic mode** (“jogging”), a switch is only made to the next step if the transition is fulfilled *and* a manual transition signal is present.
- ▷ In **automatic or semiautomatic mode** (“step enabling”), a switch is made to the next step if either the transition is fulfilled (as in automatic mode) *or* a manual transition signal is present.
- ▷ In **manual mode**, the steps are selected by means of the step number and activated and deactivated individually.

The operating modes are set in the parameters of the GRAPH function block.

11.1.5 Procedure for configuration

Define control sequences within your user program which can be solved using a sequential control. Itemize the task until you have gained an overview of the number and sequence of steps and the required signals.

First enter the signals which are already specified into the PLC tag table. If the sequential control is a complete block within the user program, it is appropriate to create a separate PLC tag table for the sequential control. Signals which are added later can be entered in the PLC tag table at any time.

Create a new function block in the GRAPH language. Enter the structure of the sequencer in the opened block with the steps, transitions, branches, and jumps. At least one initialization step must be present at which the sequential control starts, and also a sequence end at which processing of the sequence ends.

If necessary, program the permanent instructions which have to be executed prior to and after processing of the sequential control.

Then enter the actions for each step and the step enabling conditions for each transition. You can configure interlocks and supervision times for each step. The actions are programmed per step in a list. Each action contains the triggering event, the operation, and the tag.

Ladder logic (LAD) and function block diagram (FBD) can be used as languages for programming of interlock, supervision, and transition. The conditions contain the binary logic operations in the form of series and parallel connection of contacts (LAD) or are in the form of linked boxes with the AND and OR functions (FBD). Comparison functions can be used in addition.

Following programming of the sequential control, it is recommendable to compile the function block and to eliminate any errors. You then insert it at the desired position in the user program, for example in the main program, and create the instance data block for the call.

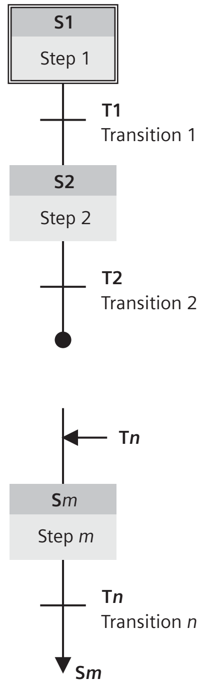
11.2 Elements of a sequential control

11.2.1 Steps and transitions

A **sequence step** contains the actions (instructions, commands) to be executed when activating a step, e.g. switch on a drive or call a block. You program the actions as a table. A step is identified by a number and/or name (Fig. 11.3).

You can configure an interlock condition for each sequence step with which the actions in the step are controlled. If the step is activated and the interlock condition fulfilled, the envisaged action is executed. The action is not executed if the interlock condition is not fulfilled. The transition to the next step is independent of the interlock function.

Elements of the programming language GRAPH – Steps and transitions



A **sequential control** contains one or more sequencers. A **sequencer** consists of steps (instructions) and transitions (conditions).

A **step** (*Sn*) contains the actions to be executed in the form of instructions. The actions of a step are only executed if the step is "activated". A deactivated step does not execute any actions. A step is activated if

> the previous transition is fulfilled or

> when starting the sequencer if it is an initialization step.

A **transition** (*Tn*) contains the step enabling conditions in the form of scans and logic operations. The conditions of a transition are only scanned if the transition is "valid", i.e. if the previous step is activated. If a valid transition is fulfilled, the previous step is deactivated and the following program element is activated.

A **sequence end** terminates a sequencer. A sequence end always follows a transition. If the transition is fulfilled, processing of this sequencer is terminated.

A **jump** is the transfer from a transition to any step in a sequencer which is present in the same sequential control as the jump.

A jump is always positioned after a transition. If this is fulfilled, the jump is made to the specified step, which is then activated.

The entry point is always in front of a step.

Fig. 11.3 Steps and transitions

The actions in a step can be monitored. If the step supervision is fulfilled, i.e. the supervision function is triggered, a fault is present which prevents transition to the next step and which can be output as an alarm.

A **transition** contains the conditions required for progression of the sequencer, e.g. the expiry of a timer function or the linking of sensor scans. Ladder logic (LAD) or function block diagram (FBD) can be used as the programming languages for the step enabling conditions. A transition is identified by a number or name.

Processing of a sequencer

You start the processing of a sequential control by calling the GRAPH function block. The first step processed when starting a sequential control is the initial step. You identify this step when configuring. It need not be the first step in a sequencer. It can also be in the middle of a sequencer.

If the sequential control consists of several independent sequencers, you can define initial steps in each sequencer which are activated when starting the sequential control.

The actions in a sequence step are carried out if the step is "active". The following transition is then also "valid" and is processed. If a valid transition is fulfilled, i.e. the step enabling condition has signal state “1”, the previous step is deactivated and the step following the transition is activated.

If the last transition is followed by a sequence end, processing of the sequencer is terminated. To restart a sequential control, for example if all sequencers have been finished, apply a rising signal edge to the input parameter INIT_SQ of the GRAPH function block.

11.2.2 Jumps in a sequential control

You can leave linear processing of the steps within a sequencer or between sequencers in a sequential control. For example, you can repeatedly process the sequencer in a program loop by jumping to a previous step shortly before the end of the sequencer.

You configure a jump following a transition and the associated jump destination prior to a step. The jump is executed if the transition is fulfilled and the step which is jumped to is activated.

A jump is defined by the number of the transition after which it is configured. The jump destination is defined by the number of the step which follows the jump destination. It is permissible to jump to a destination from more than one position.

11.2.3 Branching of a sequencer

A sequencer can contain simultaneous and alternative branches (Fig. 11.4). Simultaneous and alternative branches can be used together in a sequencer.

With **simultaneous branching**, all branches are processed in parallel (quasi at the same time). A simultaneous branch commences following a transition. Each simultaneous branch commences with a step. If the transition is fulfilled, the first steps of each simultaneous branch are processed simultaneously.

The simultaneous branches are combined following their respective last step. The subsequent transition is valid, i.e. it will be processed if the last steps of all combined simultaneous branches are active (AND condition).

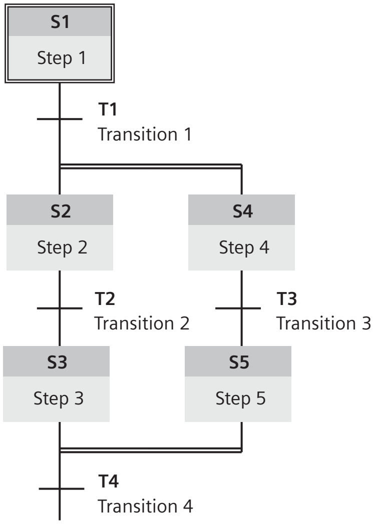
If a simultaneous branch is finished by a sequence end, this has no influence on the rest of the sequencer.

The division into simultaneous branches and their combination are not mutually dependent. You can insert a simultaneous branch at any position and also combine a simultaneous branch with the branch on its left at any position.

With **alternative branching**, only one of the branches is processed. An alternative branch commences following a step. Each alternative branch commences with a transition. If the transition is fulfilled, the first step of the respective branch becomes active and the transitions of the other branches become invalid.

If two or more transitions are fulfilled simultaneously, the step becomes active which follows the transition which is on the furthest left.

Elements of the programming language GRAPH – branching



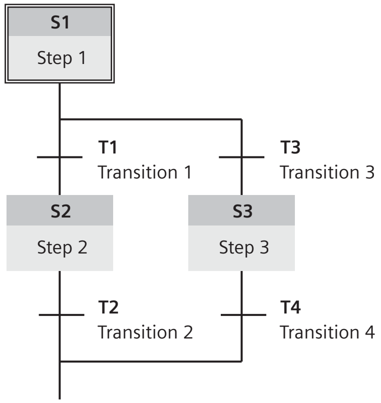
A **sequencer** can include branching; it is then divided into two or more branches. The branches can merge again or finish separately.

With **simultaneous branching**, the further branches are processed simultaneously. An simultaneous branch starts and finishes with a step.

If the previous transition is fulfilled, the first steps of all simultaneous branches are activated simultaneously.

If the simultaneous branches are merged following their respective last step, the last steps of all simultaneous branches must have been activated so that the further transition becomes valid (AND condition).

A simultaneous branch can also be finished by a sequence end. The processing in the other simultaneous branches is not influenced by this.



With **alternative branching**, only one of the further branches (alternative) is processed. An alternative branch starts and finishes with a transition.

If the previous step is activated, the first transitions of the alternative branches are valid in each case. Only the alternative branch whose start transition is fulfilled first is processed. If two or more start transitions are fulfilled, only the branch which is on the furthest left of the associated branches is processed.

If alternative branches are combined following their respective last transition, the following program element is activated if the transition of the processed branch is fulfilled (OR condition).

An alternative branch can also be finished by a sequence end or a jump.

Fig. 11.4 Simultaneous and alternative branching

The alternative branches are combined following their respective last step. If the transition of the processed branch is fulfilled, the next (common) step is processed. The combination of alternative branches corresponds to an OR condition.

An alternative branch can be finished by a sequence end or a jump.

11.2.4 GRAPH-specific tags

A data structure is created in the static local data for each configured step, for each configured transition, and for the sequential control. You can use the components of the data structure in the user program. You can read these values at any time. To guarantee trouble-free control of the sequencer, a write operation is not advisable. Some of these components are described below as examples.

Step activation time

The step activation time is started when activating a step. This delivers two values. The value *#Step_name.T* corresponds to the total duration which has passed since activation of the step. The value *#Step_name.U* contains the uninterrupted duration, in other words the duration from the start of step activation minus the duration for a fault triggered by the supervision function. *Step_name* is the symbolic name of the respective step. For example, if the step is named *Lower_drill*, the step activation time is scanned within the GRAPH function block by *#Lower_drill.T* (total step activation time) or *#Lower_drill.U* (uninterrupted step activation time). The values exist in the data type TIME.

The limits for the step activation time – the step supervision times – are set when configuring the sequential control under *Options* > *Settings* and *PLC programming* > *GRAPH*. They apply to all steps.

GRAPH provides two functions for comparing the step activation time with the step supervision times: The *CMP>T* function has signal state “1” if the current activation duration is greater than the configured supervision time. The *CMP>U* function has signal state “1” if the current, uninterrupted activation duration is greater than the configured uninterrupted supervision time. The two functions are represented as comparison functions.

Step status

The binary tag *#Step_name.S1* has signal state “1” for one processing cycle if the step named *Step_name* is activated. The *#Step_name.X* tag indicates with signal state “1” that the step is activated. The *#Step_name.S0* tag has signal state “1” for one processing cycle if the step is deactivated.

Transition status

The binary tag *#Transition_name.TV* indicates with signal state “1” that the transition named *Transition_name* is valid, i.e. it is being processed. The *#Transition_name.TT* tag indicates with signal state “1” that the transition is fulfilled. The *#Transition_name.TS* tag indicates with signal state “1” that the transition is switched.

11.2.5 Permanent instructions

Permanent instructions are program components which are processed in every cycle independent of the status of the sequential control. Permanent pre-instructions are processed prior to the sequential control, post-instructions after the sequential control.

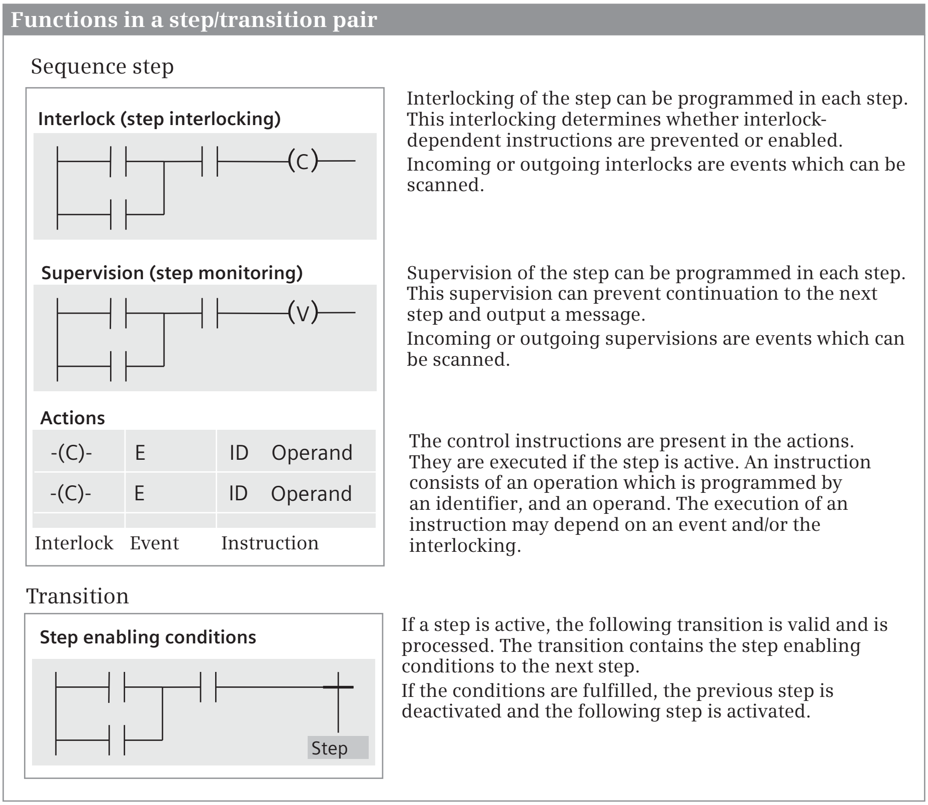
Permanent instructions can be programmed in LAD or FBD. Any number of permanent instructions can be used.

For programming, double-click on the permanent instructions in the GRAPH navigation or click on the *Permanent pre-instructions* or *Permanent post-instructions* symbol in the toolbar of the working window.

11.2.6 Step and transition functions

You program a step together with the subsequent transition. These are always handled in pairs. The step can remain empty if no actions are envisaged at the current position in the sequencer. The subsequent transition is then valid immediately. It is also possible to program an “empty transition” without step enabling conditions. This is then fulfilled immediately when processing.

Fig. 11.5 shows the components of a step/transition pair. With an alternative branch, the first transitions of the branches are counted to the previous step. When combining a simultaneous branch, the common transition is displayed in each case with the last step of a branch.


Functions in a step/transition pair
Sequence step
Interlock (step interlocking)
C
Supervision (step monitoring)
V
Actions
Transition
Step enabling conditions
Step

Interlocking of the step can be programmed in each step. This interlocking determines whether interlock-dependent instructions are prevented or enabled.

Incoming or outgoing interlocks are events which can be scanned.

Supervision of the step can be programmed in each step. This supervision can prevent continuation to the next step and output a message.

Incoming or outgoing supervisions are events which can be scanned.

Interlock	Event	Instruction
-(C)-	E	ID Operand
-(C)-	E	ID Operand

The control instructions are present in the actions. They are executed if the step is active. An instruction consists of an operation which is programmed by an identifier, and an operand. The execution of an instruction may depend on an event and/or the interlocking.

If a step is active, the following transition is valid and is processed. The transition contains the step enabling conditions to the next step.

If the conditions are fulfilled, the previous step is deactivated and the following step is activated.

Fig. 11.5 Components of a sequence step/transition pair

Interlock

An interlock condition is specific to a step. If the interlock condition is fulfilled (this is the “good case”), the actions depending on the interlock are carried out for the active step. If the interlock condition is not fulfilled, the actions depending on the interlock are not carried out. The change in status of the interlock condition can be scanned with events.

The event L1 means that the interlock condition changes from the “fulfilled” state to the “not fulfilled” state (fault coming). Actions dependent on the interlock condition are then no longer executed.

The event L0 means that the interlock condition changes from the “not fulfilled” state to the “fulfilled” state (fault going). Actions dependent on the interlock condition are then executed again.

The transition to the next step is independent of the state of the interlock condition. When deactivating a step, a fulfilled interlock condition is automatically canceled.

You program an interlock condition as a logic operation in LAD or FBD. You can use a maximum of 32 program elements per interlock condition.

An interlock error is signaled if a non-fulfilled interlock condition occurs. You can activate or deactivate the acknowledgment requirement for signaling of the interlock error. If the acknowledgment requirement is activated, processing of the sequencer is only continued following acknowledgment.

Supervision

A monitoring condition specific to a step is referred to as supervision. If the supervision condition is fulfilled, a fault is present which results in a fault signal. A non-fulfilled supervision condition is the “good case”. The change in status of the supervision condition can be scanned with events.

The event V1 means that the supervision condition changes from the “not fulfilled” state to the “fulfilled” state. A fault is then present.

The event V0 means that the supervision condition changes from the “fulfilled” state to the “not fulfilled” state. The fault then goes again.

In the case of a fulfilled supervision condition, the transition to the next step is omitted even if the following transition is fulfilled. The uninterrupted step activation time #*Step_name*.U is stopped and the complete step activation time #*Step_name*.T continues.

A fulfilled supervision condition is automatically reset when a step is deactivated (a deactivated step cannot be faulty, only activated steps are monitored). Therefore, monitoring can only be carried out for actions which are programmed in the associated step.

You program a supervision condition as a logic operation in LAD or FBD. You can use a maximum of 32 program elements per supervision condition.

A supervision error is signaled when a non-fulfilled supervision condition occurs. You can activate or deactivate the acknowledgment requirement for signaling of the supervision error. If the acknowledgment requirement is activated, processing of the sequencer is only continued following acknowledgment.

Actions in general

An active step uses actions to control operands or tags, to call blocks, or to carry out arithmetic operations. An action can consist of the interlock condition, an event, and an instruction.

An instruction is, for example, the setting of a tag using the S operation. The instruction is S “Start drive” and means: As long as the step is active, the tag “Start drive” is set to signal state “1” in each processing cycle.

An instruction can be linked to an interlock condition. For this purpose, the instruction is preceded by the symbol -(C)- (condition). The action is then -(C)- S “Start drive” and means: As long as the step is active and as long as the condition is fulfilled, the tag “Start drive” is set to signal state “1” in each processing cycle.

An instruction can be linked to an event. An event is a change in status, for example the activation of the sequence step with the start information S1. The event is specified in front of the instruction. The action is then S1 S “Start drive” and means: If the step is active and if the event – in this case the step activation – occurs, the tag “Start drive” is set once to signal state “1”.

Events, interlock conditions, and instructions can be combined together. For the example, the action is then -(C)- S1 S “Start drive” and means: The tag “Start drive” is set once to signal state “1” if the step is activated and the interlock condition is fulfilled at the same time.

If the step does not contain any actions, it is an “empty step” which reacts like an active step. The following transition is then processed immediately.

Events

An event controls an action. The change in status of a step, a supervision or an interlock condition is used to execute an instruction once (Table 11.1).

Table 11.1 Events for actions

Identifier	Event	The action is carried out once if ...
S1 S0	Step is activated Step is deactivated	the step is processed for the first time the step is processed for the last time
V1 V0	Supervision comes Supervision goes	the supervision error (fault) occurs the supervision error is removed
L1 L0	Interlock goes Interlock comes	the interlock condition changes to ”not fulfilled” (fault) the interlock condition changes to “fulfilled”
A1	Message is acknowledged	a message is acknowledged
R1	Registration comes	a registration comes (rising edge at parameter REG_EF or REG_S)

The instructions provided by the GRAPH programming language in actions, and how they can be combined with events, are described in the next Chapter 11.2.7 “Processing of actions”.

Transitions

A transition contains the step enabling conditions to the next step. A transition is processed (the transition is “valid”) if the previous step is processed (the step is “active”). The transition is “fulfilled” if the step enabling condition has signal state “1”. The previous active step is then processed for a last time with the event S0 (the step is “deactivated”) and the following step is processed. The first processing takes place with the event S1 (the step is “activated”).

In the case of simultaneous branching, a transition is followed by two or more steps which are all activated in the case of a fulfilled transition. During the combination of the simultaneous branching, all last steps of the branches must be active before the common transition becomes valid.

In the case of alternative branching, the first transitions are all valid if the step prior to the branch is active. If one of the transitions is fulfilled, the following step is activated. The transitions of the other branches are then no longer valid and therefore only one branch is processed. If two or more transitions are fulfilled simultaneously, the branch which is on the furthest left is processed. During the combination of alternative branching, all last transitions must be fulfilled before the following common step is activated.

A jump following a transition leads to a step which does not directly follow the transition in the graphic representation. This step is activated if the transition is fulfilled.

If a transition is followed by a sequence end, processing of the sequencer is terminated if the transition is fulfilled.

If a transition does not contain a step enabling condition, it is an “empty transition”. A valid empty transition is fulfilled immediately and activates the following step.

11.2.7 Processing of actions

Controlling binary tags

Tags from the operand areas Inputs I, Outputs Q, Bit memories M, and Data DB.DBX can be controlled in a step. The tags have the data type BOOL and can be addressed absolutely or symbolically. If no symbolic address is available for absolute addressing, the GRAPH editor will generate a symbolic address according to the “Tag_n” pattern.

Table 11.2 shows the possible operations and the permissible combinations with interlock condition and events. The first column (ID) contains the qualifier of the operation. In the case of an instruction identified by “-(C)-”, the interlock condition is optional and can also be omitted.

Table 11.2 Actions for binary tags

ID	Interlock	Events	Execution
N			Set a tag to signal state "1" for as long as the step is active (non-retentive assign function)
	-(C)-	–	In each program cycle
	-(C)-	S1, V1, A1, R1	Single execution in the next program cycle
	–	S0, V0, L1, L0	Single execution in the next program cycle
S			Set a tag to signal state "1" (set function)
	-(C)-	–	In each program cycle
	-(C)-	S1, V1, A1, R1	Single execution in the next program cycle
	–	S0, V0, L1, L0	Single execution in the next program cycle
R			Set a tag to signal state "0" (reset function)
	-(C)-	–	In each program cycle
	-(C)-	S1, V1, A1, R1	Single execution in the next program cycle
	–	S0, V0, L1, L0	Single execution in the next program cycle
D	-(C)-	–	Set a tag with delay (ON delay)
L	-(C)-	–	Set a tag to signal state "1" for a specific period (pulse)

The N operation is used to set a binary tag to signal state "1" for as long as the step is active and an optional interlock condition is fulfilled. The binary tag is reset to signal state "0" when the step is deactivated or with a non-fulfilled interlock condition. In association with an event, the operation is executed once in the program cycle which follows the event.

The S operation is used to set a binary tag (latching) to signal state "1" for as long as the step is active and an optional interlock condition is fulfilled. In association with an event, the operation is executed once in the program cycle which follows the event.

The R operation is used to reset a binary tag (latching) to signal state "0" for as long as the step is active and an optional interlock condition is fulfilled. In association with an event, the operation is executed once in the program cycle which follows the event.

The D operation is used to set a binary tag to signal state "1" delayed by a specific duration. The duration of the delay is specified in seconds as a constant or PLC tag with data type TIME or DWORD. The duration starts when the step is activated and the optional interlock condition fulfilled. The binary tag is reset to signal state "0" when the step is deactivated or when the optional interlock condition is no longer fulfilled. Combination of the D operation with an event is not permissible. The binary tag shows the response of an ON delay.

The L operation is used to set a binary tag to signal state "1" for a specific duration. The duration is specified in seconds as a constant or PLC tag with data type TIME or DWORD. The duration starts when the step is activated and the optional interlock condition fulfilled. The binary tag is reset to signal state "0" when the duration has expired, when the step is deactivated, or when the optional interlock condition is no longer fulfilled. Combination of the L operation with an event is not permissible. The binary tag exhibits a pulse response.

Controlling timer functions

Tags from the operand area SIMATIC timer functions T can be controlled in a step. The tags have the data type TIMER and can be addressed absolutely or symbolically.

Table 11.3 shows the possible operations and the permissible combinations with interlock condition and events. The first column (ID) contains the qualifier of the operation. In the case of an instruction identified by "-(C)-", the interlock condition is optional and can also be omitted.

Table 11.3 Actions for SIMATIC timer functions

ID	Interlock	Events	Execution
TL	-(C)- –	S1, V1, A1, R1 S0, V0, L1, L0	Start a timer function once as extended pulse
TD	-(C)- –	S1, V1, A1, R1 S0, V0, L1, L0	Start a timer function once as retentive ON delay
TF	–	–	Start a timer function as OFF delay
TR	-(C)- –	S1, V1, A1, R1 S0, V0, L1, L0	Stop and reset a timer function once

The TL operation starts a SIMATIC timer function as extended pulse. The duration is specified as a constant or PLC tag with data type S5TIME or WORD. The operation always depends on an event. The timer function is started if the step is activated and the event occurs and when – depending on the type of event – the optional interlock condition is fulfilled simultaneously. The timer has signal state "1" once the timer function has been started and signal state "0" again when the timer function has expired. The response of the timer function, which is independent of the further response of the interlock condition and the step activation, is described in Chapter 12.4.4 "Timer response as extended pulse" on page 531.

The TD operation starts a SIMATIC timer function as retentive ON delay. The duration is specified as a constant or PLC tag with data type S5TIME or WORD. The operation always depends on an event. The timer function is started if the step is activated and the event occurs and when – depending on the type of event – the optional interlock condition is fulfilled simultaneously. The timer status has signal state "1" once the timer function has expired and signal state "0" again when the step is deactivated or the interlock condition is no longer fulfilled. The response of the timer function is described in Chapter 12.4.6 "Timer response as retentive ON delay" on page 535.

The TF operation starts a SIMATIC timer function as OFF delay. The duration is specified as a constant or PLC tag with data type S5TIME or WORD. The operation is not linked to an event or the interlock condition. The timer function is started when the step is deactivated, i.e. left. The timer status is set to signal state “1” by activation of the step and reset to signal state “0” when the timer function has expired. The response of the timer function is described in Chapter 12.4.7 “Timer response as OFF delay” on page 537.

The TR operation resets a SIMATIC timer function. The operation always depends on an event. The timer function is reset if the step is activated and the event occurs and when – depending on the type of event – the optional interlock condition is fulfilled simultaneously. The further response of the timer function (the timer status) depends on the operating mode in which the timer was started. The response is described in Chapters 12.4.4 “Timer response as extended pulse” on page 531, 12.4.6 “Timer response as retentive ON delay” on page 535, and 12.4.7 “Timer response as OFF delay” on page 537.

Controlling counter functions

Tags from the operand area “SIMATIC counter functions C” can be controlled in a step. The tags have the data type COUNTER and can be addressed absolutely or symbolically.

Table 11.4 shows the possible operations and the permissible combinations with interlock condition and events. The first column (ID) contains the qualifier of the operation. In the case of an instruction identified by “-(C)-”, the interlock condition is optional and can also be omitted.

Table 11.4 Actions for SIMATIC counter functions

ID	Interlock	Events	Execution
CU	-(C)- –	S1, V1, A1, R1 S0, V0, L1, L0	Increment a counter function once by one unit
CD	-(C)- –	S1, V1, A1, R1 S0, V0, L1, L0	Decrement a counter function once by one unit
CR	-(C)- –	S1, V1, A1, R1 S0, V0, L1, L0	Reset a counter function once
CS	-(C)- –	S1, V1, A1, R1 S0, V0, L1, L0	Set a counter function once with a count value

The principle of operation of the SIMATIC counter functions used by GRAPH is described in detail in Chapter 12.6 “SIMATIC counter functions” on page 545.

The CU operation increments the count value of a SIMATIC counter function by one unit. The operation always depends on an event. The counter function counts up if the step is activated and the event occurs and when – depending on the type of event – the optional interlock condition is fulfilled simultaneously.

The CD operation decrements the count value of a SIMATIC counter function by one unit. The operation always depends on an event. The counter function counts down if the step is activated and the event occurs and when – depending on the type of event – the optional interlock condition is fulfilled simultaneously.

The CS operation sets a SIMATIC counter function to a specified count value. The count value is specified as a constant or PLC tag with data type WORD. The operation always depends on an event. The counter function is set if the step is activated and the event occurs and when – depending on the type of event – the optional interlock condition is fulfilled simultaneously.

The CR operation resets a SIMATIC counter function to zero. The operation always depends on an event. The counter function is reset if the step is activated and the event occurs and when – depending on the type of event – the optional interlock condition is fulfilled simultaneously.

Executing program instructions

Program instructions, for example block calls, math functions, or conversion functions, can be executed in a step.

Table 11.5 shows the permissible combinations of a program instruction with interlock condition and events. The first column (ID) contains the qualifier of the operation. In the case of an instruction identified by "-(C)-", the interlock condition is optional and can also be omitted.

Table 11.5 Actions for a program instruction

ID	Interlock	Events	Execution
N			Execute program instruction
	-(C)-	–	In each program cycle
	-(C)-	S1, V1, A1, R1	Single call in the next program cycle
	–	S0, V0, L1, L0	Single call in the next program cycle

The N operation executes a program instruction for as long as the step is active and an optional interlock condition is fulfilled. In association with an event, the program instruction is executed once in the program cycle which follows the event.

All instructions which are listed in the task window in the *Instructions* pallet (with the exception of the logic operations and comparators in the *LAD* or *FBD* folder under *Basic instructions*) are permissible as program instructions. Assignments and simple arithmetic operations on digital values are permissible in addition. Examples of program instructions in an action:

```
var1 := var2                   Assignment
var1 := var2 + var3            Simple arithmetic function
var1 := SIN(var2)              Math function
var1 := SHL_WORD(var2,var3)    Shift function
CALL WAIT                      Block call, here: system function
(WT := var1
)
CALL TP TIME, "DB_name"        Block call, here: IEC timer function
(IN := var1
 PT := var2
 Q  => var3
 ET => var4
)
```

You can also call self-created blocks: The syntax of a function call is *CALL "FC_name" (parameter list)* and the syntax of a function block call is *CALL "FB_name", "DB_name" (parameter list)*.

Activating and deactivating steps

Further steps can be activated or deactivated in a step. Individual steps are addressed symbolically. If all steps are addressed, the operand is named S_ALL.

Table 11.6 shows the permissible combinations of step activation or -deactivation with interlock condition and events. The first column (ID) contains the qualifier of the operation. In the case of an instruction identified by "-(C)-", the interlock condition is optional and can also be omitted.

Table 11.6 Actions for block calls

ID	Interlock	Events	Execution
ON			Activate a different step
	-(C)-	S1, V1, A1, R1	Single activation on the occurrence of event
	–	S0, V0, L1, L0	Single activation on the occurrence of event
OFF			Deactivate a different step
	-(C)-	S1, V1, A1, R1	Single deactivation on the occurrence of event
	–	S0, V0, L1, L0	Single deactivation on the occurrence of event
OFF			Deactivate all other steps (with S_ALL operand)
	-(C)-	S1, V1	Single deactivation on the occurrence of event
	–	L1	Single deactivation on the occurrence of event

The ON operation in conjunction with a single step activates a step which is not the current step. The operation always depends on an event. The other step is activated if the current step is activated and the event occurs and – depending on the type of event – the optional interlock condition is fulfilled at the same time.

The OFF operation in conjunction with a single step deactivates a step which is not the current step. The operation always depends on an event. The other step is deactivated if the current step is activated and the event occurs and – depending on the type of event – the optional interlock condition is fulfilled at the same time.

The OFF operation in conjunction with the S_ALL operand deactivates all other steps. The operation always depends on an event. The other steps are deactivated if the current step is activated and the event occurs and – depending on the type of event – the optional interlock condition is fulfilled at the same time.

If a step is both activated and deactivated in a processing cycle, deactivation has priority.

11.3 Configuring a sequential control

You program a sequential control in the following steps:

- Insert a function block which is to accommodate the sequential control into your program.
- Configure the sequencer(s) in the function block.
- Program the actions in the steps and the step enabling conditions in the transitions.
- Supplement the sequencer by permanent instructions if applicable.
- Compile the function block and generate the associated instance data block.
- Call the function block in the program and test the sequential control.

The user program can contain several function blocks with different sequential controls.

Basic settings for the sequential control

Select the *Options > Settings* command in the main menu and click on *GRAPH* in the *PLC programming* group. You can then adapt the properties of the sequential control. For example, you can set the time monitoring functions for the sequence steps here and the properties which are assigned to a new GRAPH function block when added, such as selecting the LAD or FBD programming language for the conditions, using maximum or standard interface parameters.

11.3.1 Programming the GRAPH function block

A prerequisite for programming a sequential control is that a project has been created with a PLC station. In the project tree, open the *Program blocks* folder under the PLC station and double-click on *Add new block*.

In the *Add new block* window, select *Function block* as block type and *GRAPH* as language. You can select the block number as desired if you activate the *Manual* option. Select a meaningful name for the block which has not already been assigned to another block, a PLC tag, a symbolically addressed constant, or a PLC data type. If the *Add new and open* checkbox is activated, the new block is incorporated into data management by clicking on the *OK* button and opened for processing with the GRAPH editor (Fig. 11.6).

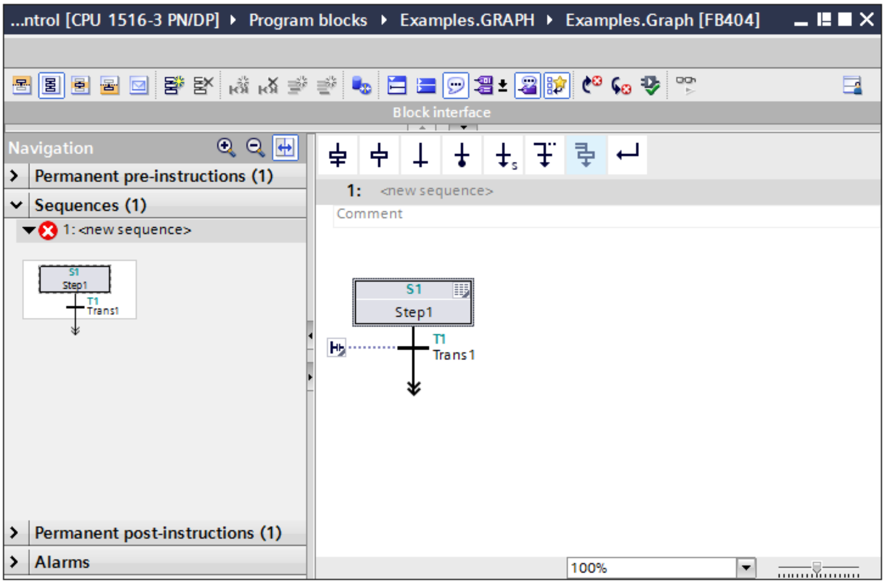

...ntrol [CPU 1516-3 PN/DP] ▸ Program blocks ▸ Examples.GRAPH ▸ Examples.Graph [FB404]
Block interface
Navigation
Permanent pre-instructions (1)
Sequences (1)
1: <new sequence>
S1
Step1
T1
Trans1
Permanent post-instructions (1)
Alarms
1: <new sequence>
Comment
S1
Step1
T1
Trans1
100%

Fig. 11.6 Working window of the GRAPH editor

The working window of the GRAPH editor is divided in two. The left side contains the GRAPH navigation with which you can navigate within the sequential control (not to be confused with the project tree with which you navigate within the project). The right side contains the working area in which you program the sequential control with steps, transitions, and branches. Use of the working window is described in Chapter 6 "Program editor" on page 247.

11.3.2 Configuring the sequencer structure

You configure the structure of a sequencer in the sequence view. The sequencer structure is displayed when you click on the *Sequence view* symbol in the toolbar of the working window. You can save an incompletely entered sequencer with the project at any time and continue processing later by opening the function block. An incomplete sequencer is identified in the navigation by a white cross on a red circle.

You create a further sequencer within the sequential control using the *Insert sequence* symbol from the toolbar of the working window. In addition to permanent instructions, you can also select one of the previously entered sequencers in the navigation for processing.

Creating a sequencer

With a newly created sequencer, a step and a transition are already positioned in the working area. The double arrow underneath the transition indicates that the sequencer is still "open" and has to be completed. Then use the mouse to drag further program elements into the working area from the favorites bar or the program elements catalog under *Basic instructions* and the *GRAPH structure* folder in order to extend the sequencer. Small gray squares in the working area indicate where the selected program element can be positioned and a green square indicates where it is positioned when you "let go".

You can remove a selected program element from the working area using the *Delete* command from the shortcut menu. With the mouse you can drag a program element in the sequencer to another (approved) position in the sequencer. All missing program elements which are required for the sequencer structure – for example a transition between two steps – are indicated by the GRAPH editor in red lettering.

In a linear sequencer, these are followed by alternate transitions and steps. You can extend the sequencer using the *Step and transition* command until the desired number of steps has been reached. Then insert a *sequence end* after the last transition.

You can program a jump in that you position the *Jump to step* instruction following a transition. The GRAPH editor then displays a table with the already programmed steps. Then select a step and the jump destination will be automatically inserted into the sequencer.

You program an OR branch using the *Alternative branch* instruction. You can position this instruction following each step and an alternative branch is then inserted with the first transition. Several alternative branches can be opened in parallel. You can terminate an alternative branch with a *Sequence end* or with another alternative branch or with the "main branch" on the far left. To do this, use the mouse to drag the double arrow at the end of the alternative branch to another partial sequencer after a transition or use the instruction *Close branch*.

You program an AND branch using the *Simultaneous branch* instruction. You can position this instruction following each transition and a simultaneous branch is then inserted with the first step. Several simultaneous branches can be opened in parallel. The *Step and transition* instruction first inserts the transition in a simultaneous branch (sequentially) and then the next step. You can terminate a simultaneous branch with a sequence end (first insert a single transition following the last step) or with another simultaneous branch or with the "main branch" on the far left. To do this, use the mouse to drag the double arrow at the end of the simultaneous branch to another partial sequencer following a step or use the instruction *Close branch*.

Processing of the sequencer commences with an initial step. You define a step as the initial step if you select it in the working window and activate the *Initial step* option in the shortcut menu. This can be any step. Several steps are permissible as initial steps within a sequence controller.

Naming of sequencer, steps, and transitions

The title of a sequencer is present in the title bar above the working area behind the sequencer number. In the case of a newly created sequencer, *<new sequence>* is present here. You can change this name by clicking in the title bar.

The number of a step (e.g. S1) or of a transition (e.g. T1) can be changed by selecting the number and choosing the *Rename* command from the shortcut menu. You can change the designation of a step (e.g. Step1) or of a transition (e.g. Trans1) in the single step view by clicking in the title bar of the step or of the transition and entering a different designation.

The GRAPH editor provides support in renumbering steps and transitions. Select a step or a transition and select the command *Renumber...* from the shortcut menu. In the displayed window you can select whether you wish to renumber steps and/or transitions, the number starting at which this is to be carried out, and whether the renumbering is to take place in the complete sequential control (the complete block), in the current sequencer, or in the current branch.

11.3.3 Programming steps and transitions

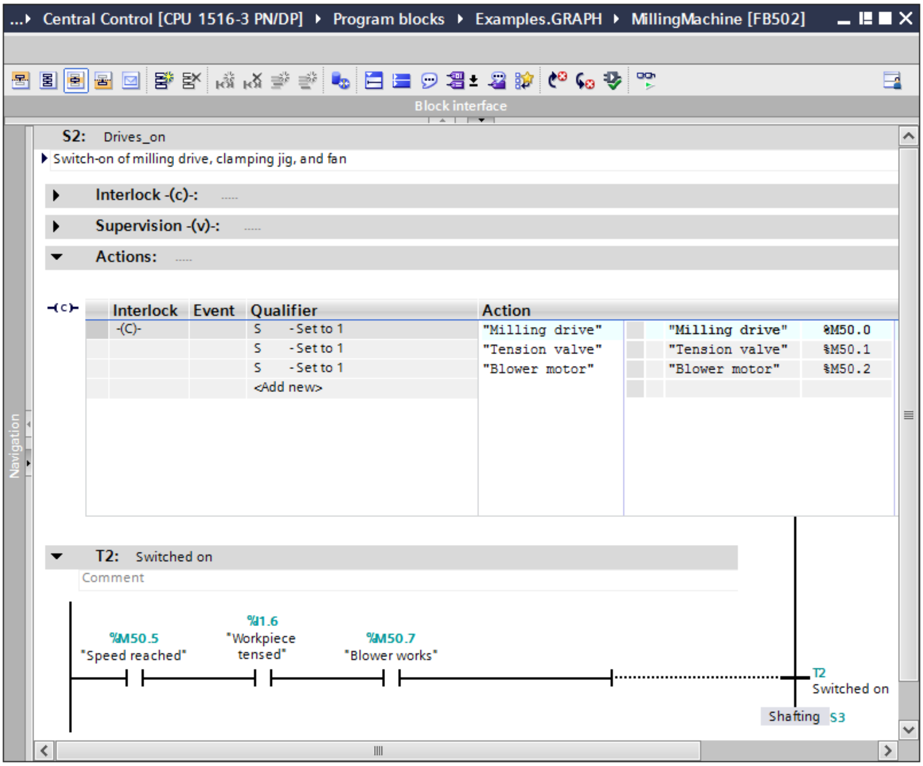
In order to program the actions and step enabling conditions, select a step or a transition and then *Single step view* in the toolbar of the working window, or double-click on a step or transition. The step and the associated, following transition are displayed (Fig. 11.7). In the case of alternative branching, all following transitions of the alternative branches are displayed for the step prior to the branch.

The single step display consists of four “networks” which you can open and close using the *Open all networks* and *Close all networks* symbols. Clicking on the small triangle on the left of the “Network title” results in the same response. You can change the programming language in the networks (with the exception of the *Actions* networks) in the block properties: Select the function block with the sequential control in the project tree and then the *Properties* command in the shortcut menu. Set the language in the networks (LAD, FBD) under *Block* in the *General* tab. You can provide each network with a heading.

You program the step interlocks in the *Interlock* network. Open the network and program the logic operation as usual with LAD or FBD. You can find the permissible instructions (mainly bit logic operations and comparators) in the favorites or in the program elements catalog under *LAD* or *FBD*.

You program the step supervision in the *Supervision* network and the step enabling conditions in the *Transition* network in the same manner as you program the step interlocks in the *Interlock* network.

The *Actions* network consists of a table in which you enter the instructions to be executed. You enter the instructions in the *Qualifier* column. Click on *<Add new>* and then select the desired instruction from the drop-down list. Specify the associated tag or operand in the *Action* column. You can control the display using the *Abso-*



...▸ Central Control [CPU 1516-3 PN/DP] ▸ Program blocks ▸ Examples.GRAPH ▸ MillingMachine [FB502]
Block interface
Navigation

S2: Drives_on

Switch-on of milling drive, clamping jig, and fan

Interlock -(c)-:

Supervision -(v)-:

Actions:

Interlock	Event	Qualifier	Action		
-(C)-		S - Set to 1	"Milling drive"	"Milling drive"	%M50.0
		S - Set to 1	"Tension valve"	"Tension valve"	%M50.1
		S - Set to 1	"Blower motor"	"Blower motor"	%M50.2
		<Add new>			

T2: Switched on

Comment

Fig. 11.7 Example of single step view

lute/symbolic operands symbol. In the *Event* column, select the event from a dropdown list for which the instruction is to be executed. There is a mandatory entry for an edge-controlled instruction; this is indicated by the <???> string highlighted in red. You specify in the *Interlock* column whether the instruction specified in the *Qualifier* column depends on the step interlock.

Some instructions in actions, for example block calls with parameter list, require several lines. You add further lines to an action by positioning the cursor in the line and activating the *Allow multi-line mode* option in the shortcut menu. A further line is added each time you press the RETURN button in the *Action* column.

In addition to the code letter, you can also display the significance of the events and qualifiers in the table. Click with the mouse in the table, and activate the *Show event descriptions* and/or *Show qualifier descriptions* commands in the shortcut menu.

11.3.4 Programming permanent instructions

To program permanent instructions, select the *Permanent pre-instructions* or *Permanent post-instructions* section in the GRAPH navigation. In the navigation, click on a network in these instructions and the logic operations of the permanent instructions will be displayed in the working area. You set the programming language (LAD, FBD) as with single-step programming in the properties of the function block.

You can use the complete LAD/FBD set of instructions in the program elements catalog for the permanent instructions. You add an additional network by selecting the previous network and then the *Insert network* command from the shortcut menu. Each network can be provided with a heading.

11.3.5 Configuring block-independent alarms

In order to configure block-independent alarms, open the *Alarms* section in the GRAPH navigation, for example using the *Alarm view* symbol in the toolbar of the working window (Fig. 11.8).

You can activate or deactivate the alarms. With the interlock and supervision alarms, you can activate or deactivate the acknowledgment requirement. You make the default settings for this in the main menu using the *Options* > *Settings* command under *PLC programming* > *GRAPH* in the *Alarm properties* section. You can change the standard text “GRAPH7_INTERLOCK_ERROR” or “GRAPH7_SUPERVISION_ FAULT”.

11.3.6 Attributes of the GRAPH function block

You set the block attributes in the block properties. Select the GRAPH function block in the project tree and then the *Properties* command from the shortcut menu (Fig. 11.9).

The *IEC check* attribute indicates how strict the data type test is to be in the code block. Further details are described in Chapter 4.5.2 “Implicit data type conversion” on page 108.

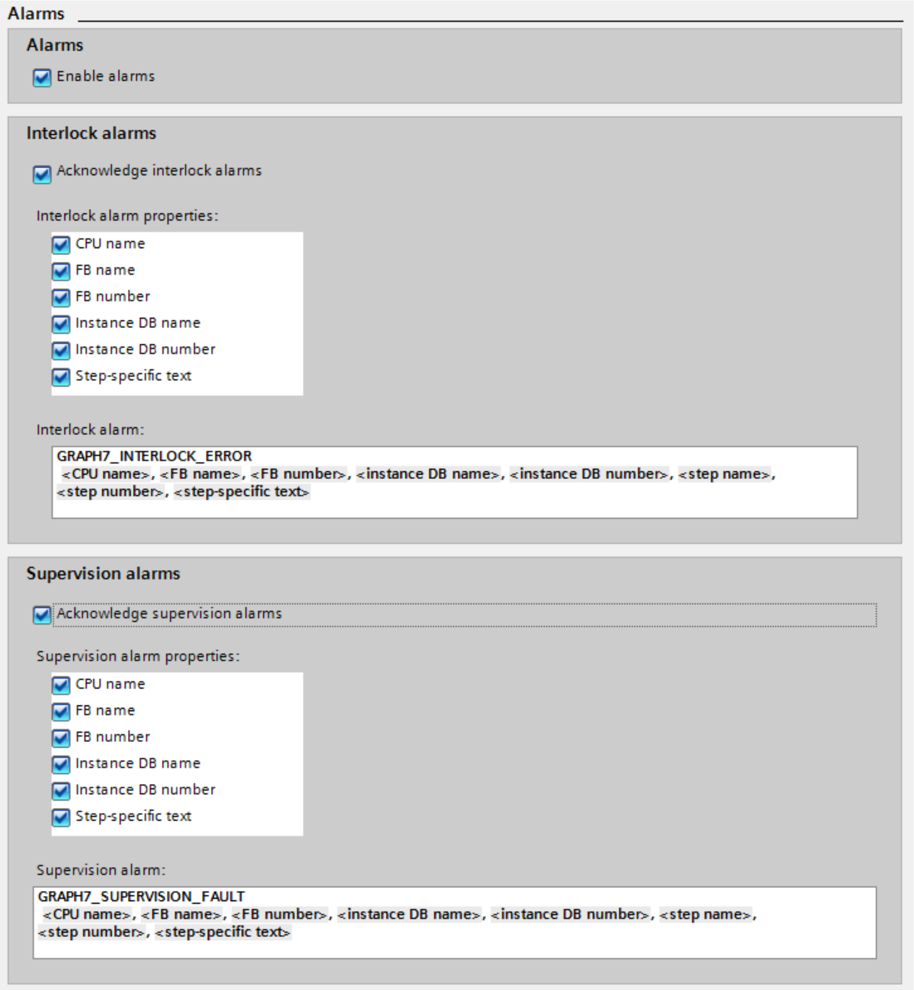
The attribute *Handle errors within block* is activated if one of the functions GET_ERROR or GET_ERROR_ID is programmed in the GRAPH function block, for example in the downstream permanent instructions (see Chapter 5.8.2 “Local error handling” on page 213).

The attribute *Block can be used as know-how protected library element* indicates whether the GRAPH function block with know-how protection can be used in a library.

The *Optimized block access* attribute is always activated and means that a memory-optimized instance data block is created when the GRAPH function block is called.

Using the *Skip steps* attribute, you allow steps in a sequencer to remain deactivated if both transitions before and after the step are fulfilled. The step between these transitions is not processed. A switch is made immediately to the next step.

If the *Acknowledgment required for supervision errors* attribute is activated, processing of the sequencer is only continued in the event of a supervision error when an acknowledgment has been made.



Alarms

Alarms

☑ Enable alarms

Interlock alarms

☑ Acknowledge interlock alarms

Interlock alarm properties:

☑ CPU name
☑ FB name
☑ FB number
☑ Instance DB name
☑ Instance DB number
☑ Step-specific text

Interlock alarm:

GRAPH7_INTERLOCK_ERROR
<CPU name>, <FB name>, <FB number>, <instance DB name>, <instance DB number>, <step name>, <step number>, <step-specific text>

Supervision alarms

☑ Acknowledge supervision alarms

Supervision alarm properties:

☑ CPU name
☑ FB name
☑ FB number
☑ Instance DB name
☑ Instance DB number
☑ Step-specific text

Supervision alarm:

GRAPH7_SUPERVISION_FAULT
<CPU name>, <FB name>, <FB number>, <instance DB name>, <instance DB number>, <step name>, <step number>, <step-specific text>

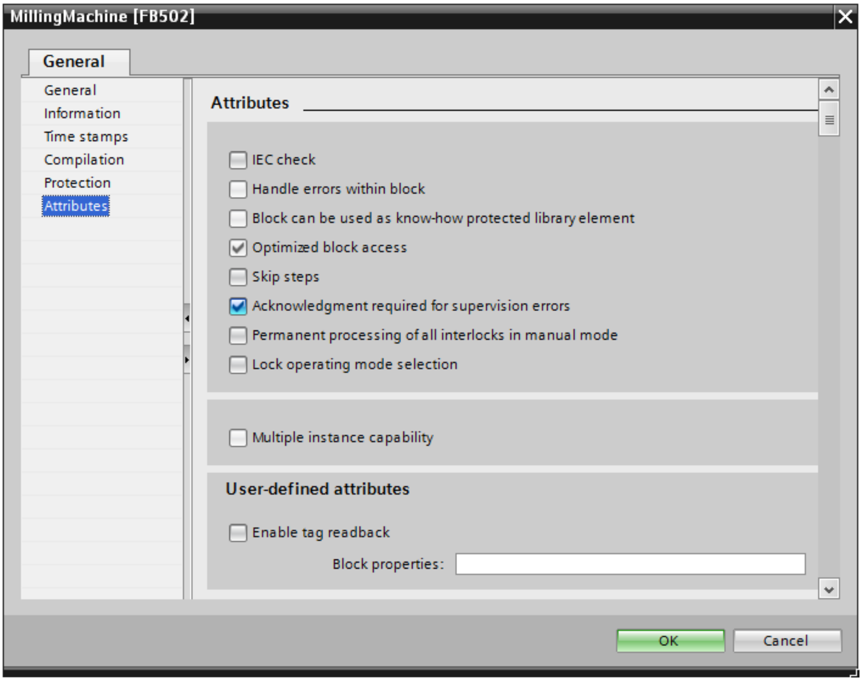
Fig. 11.8 Configuration of GRAPH alarms

If the *Permanent processing of all interlocks in manual mode* attribute is activated, the interlock conditions of all steps, including the non-activated ones, are processed in manual mode.

The activated *Lock operating mode selection* attribute prevents a manual changeover of the operating mode via the *Testing* task card.

11.3.7 Using the GRAPH function block

You define the number of block parameters prior to calling. You make the default setting in the main menu using the *Options* > *Settings* under *PLC programming* > *GRAPH* in the *Interface* section. If the *Maximum interface parameters* option is activated, the maximum parameter set is shown, otherwise the standard parameter



Attributes

□ IEC check
□ Handle errors within block
□ Block can be used as know-how protected library element
☑ Optimized block access
□ Skip steps
☑ Acknowledgment required for supervision errors
□ Permanent processing of all interlocks in manual mode
□ Lock operating mode selection

□ Multiple instance capability

User-defined attributes

□ Enable tag readback

Block properties:

Fig. 11.9 Block attributes for the sequential control

set. The standard parameter set allows operation of the sequential control in the various operating modes with the possibility for acknowledging messages. The maximum parameter set contains additional parameters for diagnostics. You can manually add or remove individual parameters in both parameter sets.

For the currently opened block, you define the parameter set using the *Edit* > *Maximum interface parameters* or *Edit* > *Default interface parameters* command.

Calling the GRAPH function block

The function block for the sequential control is always called as a single instance. Open the called block, drag the GRAPH function block from the project tree into the working area, and define the instance data block.

When the instance data block is generated, the GRAPH editor creates write-protected PLC data types (G7_...), including data types for a step (*G7_StepPlus)* and for a transition (*G7_TransitionPlus*). *G7_StepPlus* and *G7_TransitionPlus* contain the GRAPH-specific tags for each step or transition (see Chapter 11.2.4 “GRAPH-specific tags” on page 478). You can address these tags in the GRAPH function block using *#Step_name.var* or *#Transition_name.var*, or outside the function block using *“DB_name”.Step_name.var* or *“DB_name”.Transition_name.var*.

Example: If a step is named *Drives_on*, you can scan the step activation using *#Drives_on.S1* and the uninterrupted step activation time using *#Drives_on.U* in the GRAPH function block.

Operating modes

The operating modes of the sequential control are controlled using positive signal edges at the input parameters:

▷ SW_AUTO Switch on *Automatic mode*
The sequential control automatically switches to the next step if the transition is fulfilled.

▷ SW_TAP Switch on *Semiautomatic mode* (“jogging”)
The sequential control switches to the next step if the transition is fulfilled and if a positive edge occurs at the T_PUSH parameter.

▷ SW_TOP Switch on *Automatic mode* or *Semiautomatic mode* (“stepping”)
The sequential control switches to the next step if the transition is fulfilled or if a positive edge occurs at the T_PUSH parameter.

▷ SW_MAN Switch on *Manual mode*
The sequential control activates the step displayed at the S_NO output parameter if a positive edge occurs at the S_ON parameter. The step displayed at the S_NO parameter is deactivated by a positive edge at the S_OFF parameter. You can define a special step using the S_SEL parameter. You can switch to the previous step (in the direction of smaller step numbers) using the S_PREV parameter and to the subsequent step using S_NEXT.

The parameters described are available with both the maximized and standard parameter sets.

Compiling the GRAPH function block

You compile a GRAPH function block just like any other function block (see Chapter 6.5 “Compiling blocks” on page 276).

For optimal operation, the GRAPH function block requires additional system blocks (*G7_RT_Plus_...*), which are saved during the compilation in the project tree under *Program blocks > System blocks > Program resources*.

11.4 Testing the sequential control

A prerequisite for testing the user program is an online connection between the programming device and the machine or plant to be controlled. The user program has been compiled without errors and downloaded to the CPU. The general procedure is described in Chapter 15.5 “Testing the user program” on page 677.

Additional information must be observed when loading the GRAPH function block. To enable testing of a GRAPH sequential control, you can use the following test functions: program status for sequencers and individual steps, control sequencer, and synchronize sequencer.

11.4.1 Loading the GRAPH function block

If you change the program in the GRAPH function block, you must create the instance data block again so that the changes – for example new steps and transitions – are imported into the data block.

Reloading of a (modified) GRAPH function block with the associated instance data block in RUN mode can lead to problems in execution of the sequential control. Therefore you must deactivate the sequential control prior to reloading. You can always do this in the general settings or when loading.

Standard deactivation of the sequential control when loading can be set in the main menu using the *Options* > *Settings* under *PLC programming* > *GRAPH* in the *Load* section: Activate the *Turn off sequence before downloading DB* option. If the option is deactivated, you can set the *Turn off sequence before downloading DB* action when downloading the GRAPH function block with the instance data block in the *Download preview* window.

11.4.2 Settings for program testing

The settings for program testing are made in the task window on the *Testing* task card (see Fig. 11.10 on the right). Open the GRAPH function block online and click on the *Test settings* pallet in the *Testing* task card. The attributes referred to below are described in Chapter 11.3.6 “Attributes of the GRAPH function block” on page 494.

When activated, the settings have the following meaning:

- Track active step
  The respective current step is displayed in the single step view or sequence view.
- Skip steps
  A step whose preceding and following transitions are active is skipped, i.e. not activated (only available with non-activated attribute *Generate minimized DB*).
- Mandatory acknowledgement at supervision errors
  Only available if the *Acknowledge supervision alarms* attribute is activated. This setting is then switched on by default.
- Stop sequence
  If the following transition is fulfilled, processing of the sequencer is stopped.
- Stop timers
  All step activation timers are stopped. If the setting is canceled, the step activation timers continue to run.
- Process all interlocks
  Only available if the *Permanent processing of all interlocks in manual mode* attribute is activated and *Manual mode* is switched on. When using the setting, the sequencer must be synchronized.

- ▷ Process all transitions
  All transitions are processed. It is indicated whether the respective transition is fulfilled.
- ▷ Activate actions
  This setting is switched on by default. If it is switched off, no further actions are executed in the active step.
- ▷ Activate supervisions
  This setting is switched on by default. If it is switched off, the supervision is ignored in the active step.
- ▷ Activate interlocks
  This setting is switched on by default. If it is switched off, the interlock condition is ignored in the active step.

Which test settings are selectable depends on the operating mode of the sequential controller.

11.4.3 Using operating modes

When testing, you can set the operating modes of the sequential control in the *Sequence control* pallet on the *Testing* task card (Fig. 11.10 on the left).

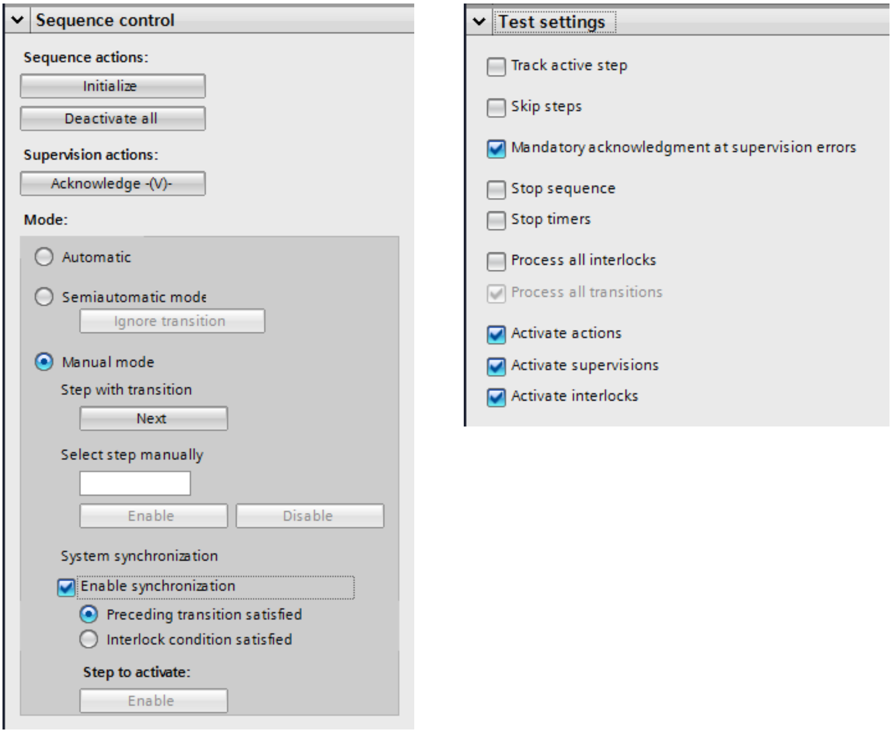
Select the operating mode prior to actual program testing:

- ▷ Automatic
  The next step is activated as soon as the transition is fulfilled.
- ▷ Semiautomatic mode
  The next step is activated as soon as
  - the valid transition is fulfilled or
  - a rising edge occurs at the block parameter T_PUSH or
  - the *Ignore transition* button is clicked.
- ▷ Manual mode
  The step to be activated is selected manually.

You commence testing of the sequential control by clicking on the *Initialize* button. You can deactivate all steps using the *Deactivate all* button. The *Acknowledge -(V)-* button is used to acknowledge a supervision error.

In manual mode you switch to the next step by clicking on the *Next* button when the transition is fulfilled.

In manual mode you can activate any step by entering its number in the *Select step manually* box and clicking on the *Enable* button. Proceed accordingly to deactivate a step. As an alternative to entering the step number, you can also select the step to be activated or deactivated in the sequencer which is displayed in the GRAPH navigation or in the working window.



Sequence control

Sequence actions:

Initialize

Deactivate all

Supervision actions:

Acknowledge -(V)-

Mode:

○ Automatic

○ Semiautomatic mode

Ignore transition

● Manual mode

Step with transition

Next

Select step manually

Enable Disable

System synchronization

☑ Enable synchronization

● Preceding transition satisfied

○ Interlock condition satisfied

Step to activate:

Enable

Test settings

□ Track active step

□ Skip steps

☑ Mandatory acknowledgment at supervision errors

□ Stop sequence

□ Stop timers

□ Process all interlocks

☑ Process all transitions

☑ Activate actions

☑ Activate supervisions

☑ Activate interlocks

Fig. 11.10 Test aids for GRAPH in the *Testing* task card

11.4.4 Synchronization a sequencer

A sequential control only works correctly if the statuses of the sequencer and the process to be controlled are matched to each other. If individual steps are activated and deactivated when testing in manual mode, it is possible that the sequential control and the controlled process are no longer synchronous. You should synchronize the sequencer before leaving manual mode and switching to automatic or semiautomatic mode.

To synchronize the sequencer, activate the *Enable synchronization* checkbox in the *Sequence control* pallet on the *Testing* task card. There are two manners in which the synchronization point – the step to be activated – can be found:

- If you select the *Preceding transition satisfied* option, all steps are selected whose previous transition is fulfilled and whose following transition is not fulfilled.
- If you select the *Interlock condition satisfied* option, all steps are selected whose interlock condition is fulfilled and whose following transition is not fulfilled.

To carry out synchronization, select the desired step in the GRAPH navigation or in the sequencer view in the working window and click on the *Enable* button.

11.4.5 Testing with program status

The general procedure for testing with program status is described in Chapter 15.5.2 “Testing with program status” on page 679.

For testing with program status, open the GRAPH function block and switch the program status on using the *Monitoring on/off* symbol in the toolbar of the working window. If an online connection to the CPU does not yet exist, the connection dialog is opened. After the online connection is added, the program status is displayed.

Program status in the sequencer view

You switch on the sequencer view using the *Sequence view* symbol in the toolbar of the working window. The program status in the sequencer view indicates the status of a step or transition using different colors:

- Step shown in green: No fault present.
- Step shown in red: A supervision error is present.
- Step shown in yellow: An interlock error is present.
- Transition shown in green: The transition is fulfilled.
- Transition shown in black: The transition is not fulfilled.

On the *Testing* task card, you can define on the *Test settings* pallet that supervision errors must be acknowledged during testing in order to continue program execution. You can acknowledge the supervision error by clicking on the *Acknowledge -(V)-* button on the *Sequence control* pallet.

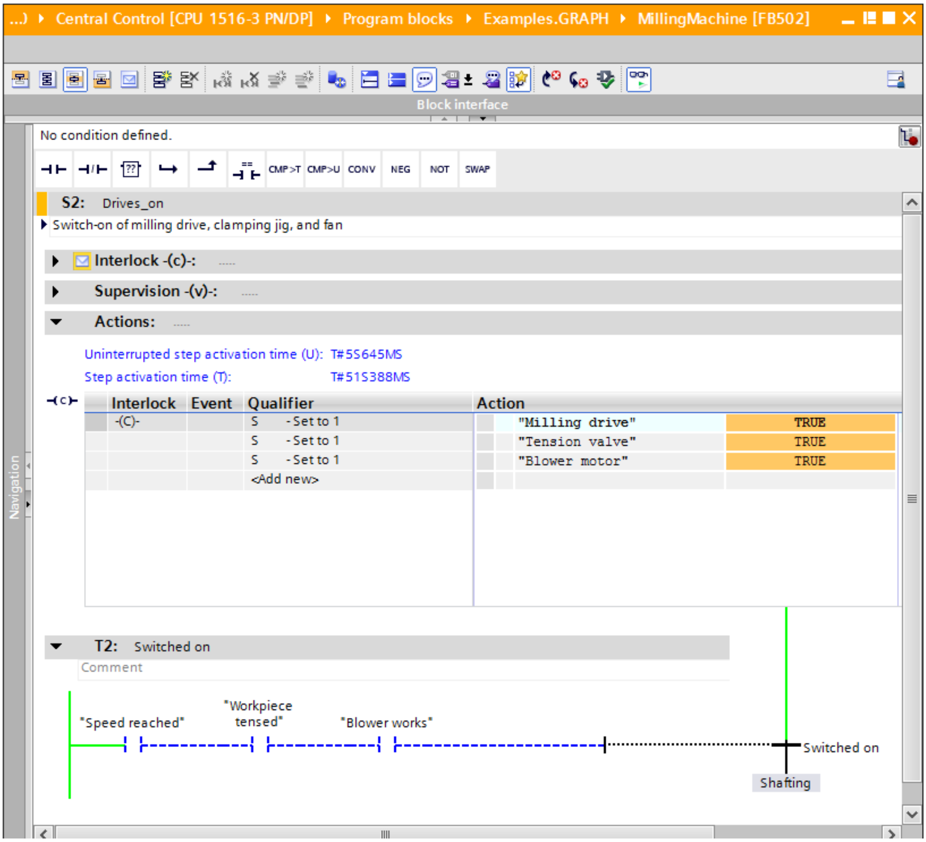
Program status in the single step view

In the GRAPH navigation, select the step to be monitored and switch on monitoring. The action list is extended by the monitored value (Fig. 11.11). Depending on the representation in LAD or FBD, the logic operations are displayed with a green continuous line (for signal state “1”) or a blue dashed line (for signal state “0”) in the conditions for interlock, supervision, and transition.

To select the display format, click in the action list on a monitored value and select the *Display format for network* > ... or *Display format* > ... command from the shortcut menu. You can then select between automatic, decimal, hexadecimal, and floating-point.

To control a tag in the action list or in one of the conditions, click on the tag and select the *Modify* > ... command from the shortcut menu. You can then select between *Modify to 0*, *Modify to 1*, and *Modify operand* in order to define a digital value.

In order to reduce the cycle processing time when testing, you can switch on the program status in a condition (e.g. a transition) starting at a particular point in the program or only monitor a specific tag. Click on the tag starting at which the program status is to be displayed or which is to be monitored and select the *Modify > Monitor from here* or *Modify > Monitor selection* command from the shortcut menu.



...) ▸ Central Control [CPU 1516-3 PN/DP] ▸ Program blocks ▸ Examples.GRAPH ▸ MillingMachine [FB502]

No condition defined.

S2: Drives_on

▸ Switch-on of milling drive, clamping jig, and fan

▸ Interlock -(c)-: .....

▸ Supervision -(v)-: .....

▾ Actions: .....

Uninterrupted step activation time (U): T#5S645MS

Step activation time (T): T#51S388MS

Interlock	Event	Qualifier	Action	
-(C)-		S - Set to 1	"Milling drive"	TRUE
		S - Set to 1	"Tension valve"	TRUE
		S - Set to 1	"Blower motor"	TRUE
		<Add new>		

▾ **T2:** Switched on

Comment

Fig. 11.11 Example of the program status in the single step view in LAD

12 Basic functions

This chapter describes the basic functions largely independent of the program language selected. Binary logic operations are an exception, for the differences between the programming languages are greatest with these functions.

The Chapters 7 "Ladder logic LAD" on page 287, 8 "Function block diagram FBD" on page 323, 9 "Structured Control Language SCL" on page 359 and 10 "Statement list STL" on page 395 describe how you can program the functions using the individual programming languages and what special features exist.

12.1 Binary logic operations

12.1.1 Introduction

Binary logic operations process the signal states of binary tags in accordance with AND, OR, and exclusive OR. The implementation of binary logic operations varies significantly in the various programming languages:

- Ladder logic (LAD) uses NO and NC symbols in series and parallel connections, with a coil as termination.
- In the function block diagram (FBD), function boxes handle the linking of binary signals.
- With the Structured Control Language (SCL), expressions with binary tags form the binary logic operations.
- In the statement list (STL), the binary logic operations are scans positioned underneath each other.

Binary logic operations can be used together with all binary tags. 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, STL).
- Control of program execution using 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 7.2.2 "Series and parallel connection of contacts" on page 292, for FBD in

Chapter 8.2.6 “Combined binary logic operations, negating result of logic operation” on page 329, for SCL in Chapter 9.2.5 “Combined binary logic operations in SCL” on page 365, and for STL in Chapter 10.2.6 “Combined binary logic operations in the statement list” on page 404.

12.1.2 Working with binary signals

Signal states “1” and “0”

The term “signal state” refers to a logic status in a binary logic operation, without considering the physical implementation. Two (contrary) statuses are simply identified as signal state “1” and signal state “0”. It could be understood, for example, that signal state “1” can always be equated to a higher electrical potential than signal state “0”, but this is incorrect. The same applies to the graphic metaphor in the ladder logic, i.e. a “current” flows with signal state “1” and does not with signal state “0”.

Signal state “1” and signal state “0” are two terms which allow the description of logic operations in a user program. It is insignificant how these statuses are implemented physically within the CPU. These terms have a physical correlation at the interface to the controlled machine or process. The type of digital module defines how the logic term “Signal state” is converted into a physical variable and vice versa, and how a physical variable is converted into a signal state “1” or “0”.

The terms “TRUE” and “FALSE” are also commonly used for signal states “1” and “0”. As a side note: In this book, the signal states “1” and “0” are set in quotation marks to distinguish them from the digits 1 and 0.

Types of digital modules

The type of digital input module determines how a physical variable is converted into a signal state (Fig. 12.1):

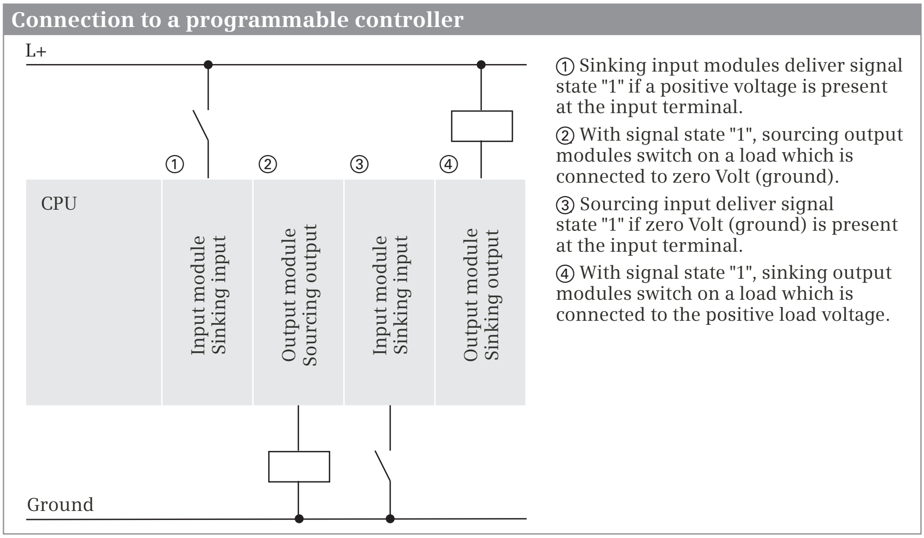
- If the module is designed for connection of an AC voltage transmitter, “Voltage present” at the module terminal means signal state “1” and “No voltage present” or an open connection means signal state “0”.

- If a module has the property “Sinking input”, a positive (DC) voltage at the module terminal is converted into signal state “1”. Zero voltage (ground) or an open connection means signal state “0”.

- If a module has the property “Sourcing input”, zero voltage (ground) at the module terminal means signal state “1” and a positive voltage or an open connection means signal state “0”.

The type of digital output module determines how a signal state is converted into a physical variable in an assignment to an output:

- ▷ If the module is designed for connection of an AC voltage load, signal state “1” means that voltage is present at the module terminal. The terminal is deenergized if the signal state is “0”.
- ▷ If the module possesses output relays, the relay contact is closed with signal state “1” and open with signal state “0”.
- ▷ A module with the property “Sourcing output” delivers a positive (DC) voltage in the case of signal state “1” at the module terminal and no voltage in the case of signal state “0” (high-impedance to power supply with electronic output amplifiers).
- ▷ If a module has the property “Sinking output”, it delivers zero volt (ground) at the terminal in the case of signal state “1” and no voltage in the case of signal state “0” (high-impedance to ground with electronic output amplifiers).


Connection to a programmable controller
L+
①
②
③
④
CPU
Input module Sinking input
Output module Sourcing output
Input module Sinking input
Output module Sinking output
Ground

① Sinking input modules deliver signal state "1" if a positive voltage is present at the input terminal.

② With signal state "1", sourcing output modules switch on a load which is connected to zero Volt (ground).

③ Sourcing input deliver signal state "1" if zero Volt (ground) is present at the input terminal.

④ With signal state "1", sinking output modules switch on a load which is connected to the positive load voltage.

Fig. 12.1 Connection of binary signals to a programmable controller

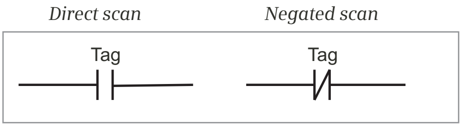
Types of sensors, NO contacts, and NC contacts

The CPU obtains control signals and feedbacks from the machine or process via sensors (signal transmitters, limit switches, pushbuttons, etc.). There are two types of binary sensors: normally open contacts and normally closed contacts. A normally open (NO) contact is a sensor which closes a circuit when activated. A normally closed (NC) contact interrupts a circuit when activated. It is used, for example, in closed circuit connections in order to switch off a control function when activated or to control the machine to a safe state in the case of an open-circuit.

To allow better understanding of the control function, it is usually defined in the user program that an action is triggered by signal state “1”. In addition, certain con-

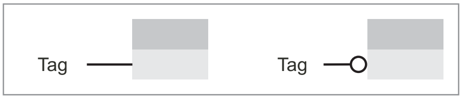
Scan signal state of binary tags

The signal state of a binary tag, for example an input, has to be scanned before it can be gated with other signal states. In association with a scan, the signal state can be negated if this is necessary for the logic operation. Different symbols are used in the various programming languages for scanning of the signal state.



Ladder logic LAD

LAD uses an NO symbol for the direct scan (scan for signal state "1") and an NC symbol for the negated scan (scan for signal state "0").



Function block diagram FBD

In FBD, the scan for signal state "1" is represented as a direct input of a function box, and the negated scan (scan for signal state "0") by the negation symbol (the negation circle) on the function box.

```
AND tag                  AND NOT tag
OR tag                   OR NOT tag
XOR tag                  XOR NOT tag
```

Structured Control Language SCL

In association with a logical operator AND, OR and XOR, a direct scan is always present with SCL (scan for signal state "1"). The negated scan (scan for signal state "0") is emulated by the NOT function.

```
A    tag                 AN   tag
O    tag                 ON   tag
X    tag                 XN   tag
```

Statement list STL

STL uses the binary functions A, O, and X to scan for signal state "1". With the negated scan (scan for signal state "0"), an "N" is appended to the binary function (AN, ON, XN).

Fig. 12.2 Direct and negated scanning of a binary operand

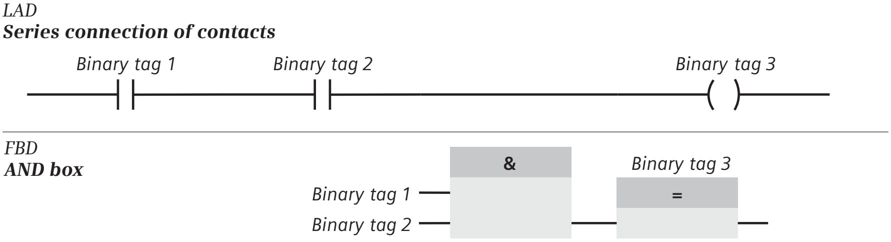
trol functions only result in actions with signal state “1”. Therefore it may be necessary to convert an zero-active signal to signal state “1” (i.e. negate it) before it is used in the user program. LAD has the NC contact for this purpose, FBD and STL have the scan for signal state “0”, and SCL has the negation NOT (Fig. 12.2).

Result of logic operation, assignment

The user program contains the statements for the control processor on how the signal states are to be linked together. The signal state resulting from the linked signal states is referred to as *Result of logic operation*. The result of logic operation assumes the value “0” or “1” just like a signal state. The result of logic operation can be linked to further binary functions.

The result of logic operation is output in order to control an actuator (relay, contactor, lamp, etc.). The memory functions, primarily the assignment, are available for this purpose. An assignment takes over the original signal state of the result of logic operation. If it is necessary to output the negated signal state (the opposite result of logic operation), a negation is programmed prior to the assignment or you use the negating assignment (LAD, FBD).

AND function, series connection



SCL
AND function

```
Binary tag 3 := Binary tag 1 & Binary tag 2;
Binary tag 3 := Binary tag 1 AND Binary tag 2;
```

STL
AND function

```
A  Binary tag 1
A  Binary tag 2
=  Binary tag 3
```

Tags	*Signal state*			
Binary tag 1 Binary tag 2	"0" "0"	"1" "0"	"0" "1"	**"1"** **"1"**
Binary tag 3	"0"	"0"	"0"	**"1"**

The AND function is fulfilled if all function inputs have the signal state "1". The AND function then delivers a result of logic operation "1" at its function output, shown in the examples by the assignment (coil).

Each AND function in the examples is shown with two inputs; the number of inputs of an AND function is theoretically unlimited.

Note: In the examples, the binary tags are scanned directly (for signal state "1"). With a negated scan (for signal state "0") it is necessary to consider the negated signal state of the binary operands for the AND operation.

Fig. 12.3 Representation and principle of operation of the AND function

12.1.3 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” (Fig. 12.3).

LAD implements the AND function using a series connection of contacts. FBD uses the AND box with two or more inputs. SCL uses the logic operator AND or &. In STL, the AND function is represented by the AND (A) operation.

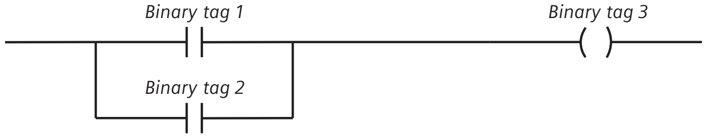
12.1.4 OR function, parallel connection

The OR function links two or more binary states together and delivers a result of logic operation “0” if all states (all results of the scans) are simultaneously “0”. In all other cases, the result of logic operation is “1” (Fig. 12.4).

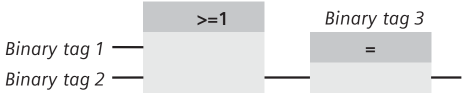
LAD implements the OR function using a parallel connection of contacts. FBD uses the OR box with two or more inputs. In STL, the OR function is represented by the OR (O) operation. SCL uses the logic operator OR.

OR function, parallel connection

LAD
Parallel connection of contacts



FBD
OR box



SCL
OR function

```
Binary tag 3 := Binary tag 1 OR Binary tag 2;
```

STL
OR function

```
O  Binary tag 1
O  Binary tag 2
=  Binary tag 3
```

Tags	*Signal state*			
Binary tag 1 Binary tag 2	**"0"** **"0"**	"1" "0"	"0" "1"	"1" "1"
Binary tag 3	**"0"**	"1"	"1"	"1"

The OR function is fulfilled if at least one of the function inputs has the signal state "1".
The OR function then delivers a result of logic operation "1" at its function output, shown in the examples by the assignment (coil).

Each OR function in the examples is shown with two inputs; the number of inputs of an OR function is theoretically unlimited.

Note: In the examples, the binary tags are scanned directly (for signal state "1"). With a negated scan (for signal state "0") it is necessary to consider the negated signal state of the binary operands for the OR operation.

Fig. 12.4 Representation and principle of operation of the OR function

12.1.5 Exclusive OR function, non-equivalence function

The exclusive OR function links two or more binary states together and delivers a result of logic operation “1” if an odd number of states (of the scan results) are simultaneously “1”. In all other cases, the result of logic operation is “0”. In the special case where the exclusive OR function has two inputs, it delivers the result of logic operation “1” if the two inputs have different signal states (Fig. 12.5).

The exclusive OR function does not exist with LAD. The function can be emulated using series and parallel connections. FBD uses the exclusive OR box with two or more inputs. In STL, the exclusive OR function is represented by the exclusive OR (X) operation. SCL uses the logic operator XOR.

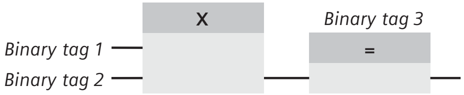
12.1.6 Negate result of logic operation, NOT contact

The result of logic operation can be negated at any position within a logic operation. Signal state “1” then becomes “0” and vice versa (Fig. 12.6).

Exclusive OR function, non-equivalence function

LAD
The exclusive OR function must be emulated by a combination of series and parallel connections.

FBD
Exclusive OR box



SCL
XOR function

```
Binary tag 3 := Binary tag 1 XOR Binary tag 2;
```

STL
Exclusive OR function

```
X  Binary tag 1
X  Binary tag 2
=  Binary tag 3
```

Tags	*Signal state*			
Binary tag 1	"0"	**"1"**	**"0"**	"1"
Binary tag 2	"0"	**"0"**	**"1"**	"1"
Binary tag 3	"0"	**"1"**	**"1"**	"0"

The exclusive OR function is fulfilled if an odd number of the function inputs has the signal state "1". The exclusive OR function then delivers a result of logic operation "1" at its function output, shown in the examples by the assignment.

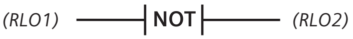
Each exclusive OR function in the examples is shown with two inputs; the number of inputs of an exclusive OR function is theoretically unlimited.

Note: In the examples, the binary tags are scanned directly (for signal state "1"). With a negated scan (for signal state "0") it is necessary to consider the negated signal state of the binary operands for the exclusive OR operation.

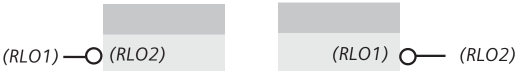
Fig. 12.5 Representation and principle of operation of the exclusive OR function

Negation of result of logic operation

LAD
NOT contact



FBD
Negation symbol



SCL
NOT operator

```
(RLO1) NOT (RLO2)
```

STL
NOT statement

```
   (RLO1)
NOT
   (RLO2)
```

Tags	*Signal state*	
RLO1	"0"	"1"
RLO2	"1"	"0"

The negation reverses the result of logic operation (RLO): Signal state "1" become signal state "0", and "0" becomes "1".

Fig. 12.6 Representation and principle of operation of the negation

Ladder logic

The NOT contact negates the “current flow”. If the current path has “current” prior to the NOT contact, no more “current” flows following the NOT contact and vice versa.

Function block diagram

The negation circle at the input or output of a function symbol negates the result of logic operation. You can

- apply the negation to the scan of a binary operand; this then corresponds to scanning for signal state “0”,
- 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 operand and the logic operation is not fulfilled, i.e. RLO = “0”).

Structured Control Language

The NOT operator negates the result of logic operation. You can use NOT at any position within an expression. If NOT is positioned directly before a tag, the signal state of the tag is negated (corresponds to scanning for signal state “0”). NOT positioned before an expression negates the result of logic operation of the expression.

Statement list

The NOT operation negates the result of logic operation. You can use NOT at any position, also within a logic operation. You can use this operation, for example, to apply a negated AND function to an output.

12.2 Memory functions

12.2.1 Introduction

The memory functions are used together with the binary logic operations in order to influence the signal states of binary tags using the result of logic operation generated by the control processor.

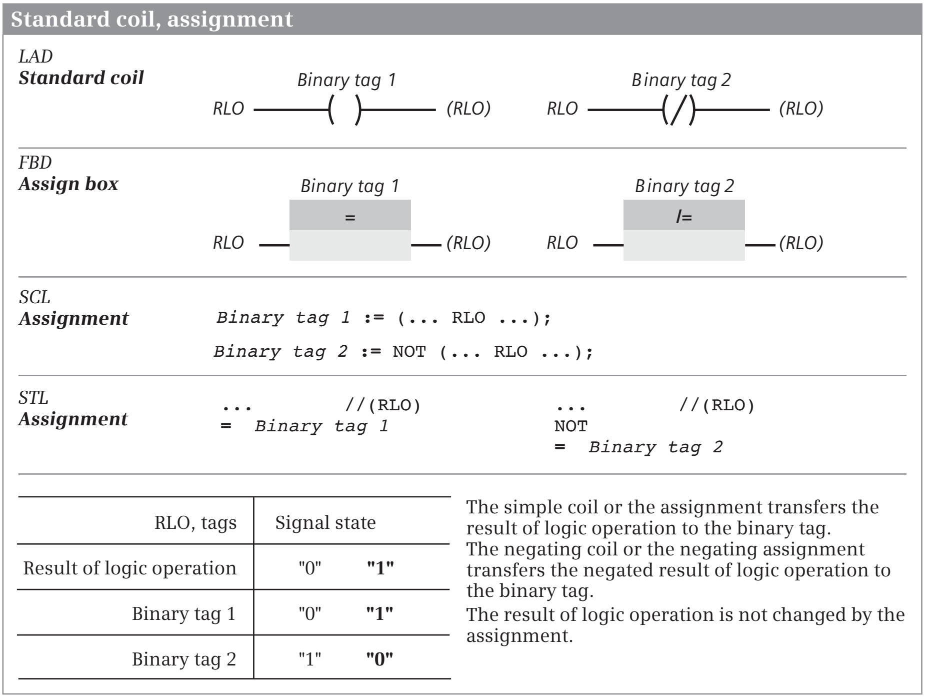
The following memory functions are available:

- Assignment of the result of logic operation
- Single setting and resetting
- Multiple setting and resetting
- Dominant setting and resetting

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.

12.2.2 Simple and negating 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”. The assignment is represented in LAD by the simple coil, and in FBD by the assign box. With SCL, the “:=” assignment operator stands for the assignment, and with STL the operation “=” (Fig. 12.7).



```
Binary tag 1 := (... RLO ...);
Binary tag 2 := NOT (... RLO ...);
```

```
...          //(RLO)
=  Binary tag 1
```

```
...          //(RLO)
NOT
=  Binary tag 2
```

RLO, tags	Signal state	
Result of logic operation	"0"	**"1"**
Binary tag 1	"0"	**"1"**
Binary tag 2	"1"	**"0"**

The simple coil or the assignment transfers the result of logic operation to the binary tag.
The negating coil or the negating assignment transfers the negated result of logic operation to the binary tag.
The result of logic operation is not changed by the assignment.

Fig. 12.7 Simple and negating coil, simple and negating assignment

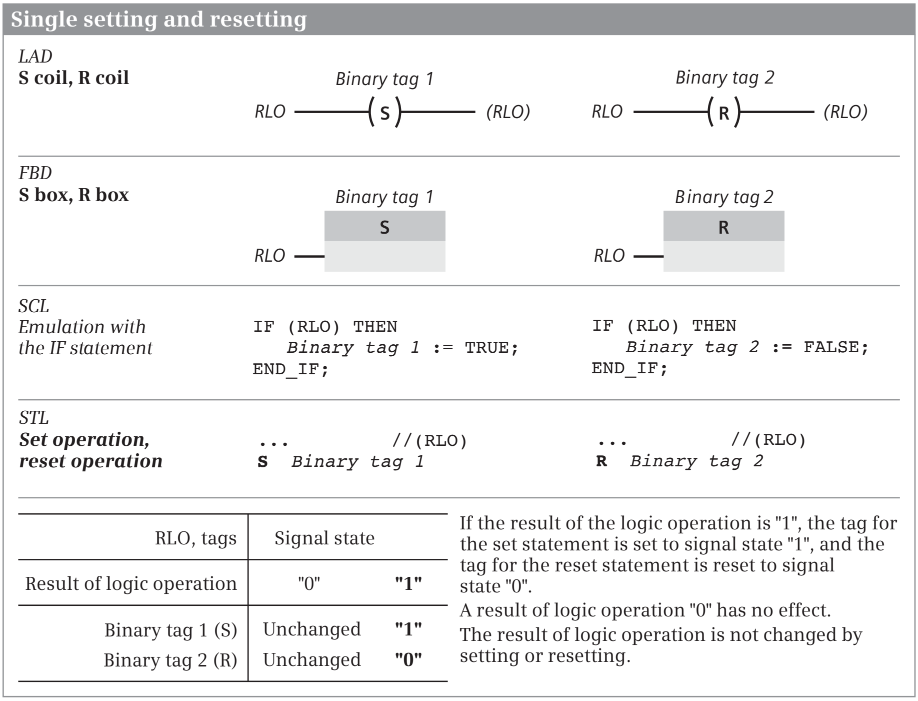
The negating assignment negates the result of logic operation before it transfers it to a binary tag. If the result of logic operation is “1”, the binary tag is reset to signal state “0”; if it is “0”, the binary tag is set to signal state “1”. The negating assignment is represented in LAD by the negating coil, and in FBD by the negating assign box. SCL and STL use the negation NOT to negate a result of logic operation.

The simple and negating coil or the simple and negating assignment do not influence the result of logic operation.

12.2.3 Single setting and resetting

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.

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 (Fig. 12.8).



Single setting and resetting

LAD
S coil, R coil

FBD
S box, R box

SCL
Emulation with the IF statement

```
IF (RLO) THEN
   Binary tag 1 := TRUE;
END_IF;
```

```
IF (RLO) THEN
   Binary tag 2 := FALSE;
END_IF;
```

STL
Set operation, reset operation

```
...        //(RLO)
S  Binary tag 1
```

```
...        //(RLO)
R  Binary tag 2
```

RLO, tags	Signal state	
Result of logic operation	"0"	**"1"**
Binary tag 1 (S) Binary tag 2 (R)	Unchanged Unchanged	**"1"** **"0"**

If the result of the logic operation is "1", the tag for the set statement is set to signal state "1", and the tag for the reset statement is reset to signal state "0".
A result of logic operation "0" has no effect.
The result of logic operation is not changed by setting or resetting.

Fig. 12.8 Representation and principle of operation of the set and reset functions

Single setting and resetting do not influence the result of logic operation.

With LAD, single setting is represented by the set coil, and single resetting by the reset coil. With FBD, single setting is represented by the set box, and single resetting by the reset box. SCL only has the assignment operator for controlling a binary operand. Setting or resetting a binary operand with RLO = “1” can be emulated together with other SCL statements, for example by the IF statement. With STL, an “S” stands for setting a binary operand, and an “R” for resetting.

To make the programming clearer, you should always use the single set and reset statements in pairs for a specific binary tag, and only once each. You should also avoid controlling this binary tag in addition by an assignment.

When using the individual memory functions on the same binary tag, the positioning sequence is important since with simultaneous activation of the set and reset statements, the statement processed last is dominant. For example, if the reset statement is processed following the set statement, resetting is dominant.

Note that the binary tag used for a single memory function may be reset during startup by the CPU's operating system. In certain cases, the signal state is retained:

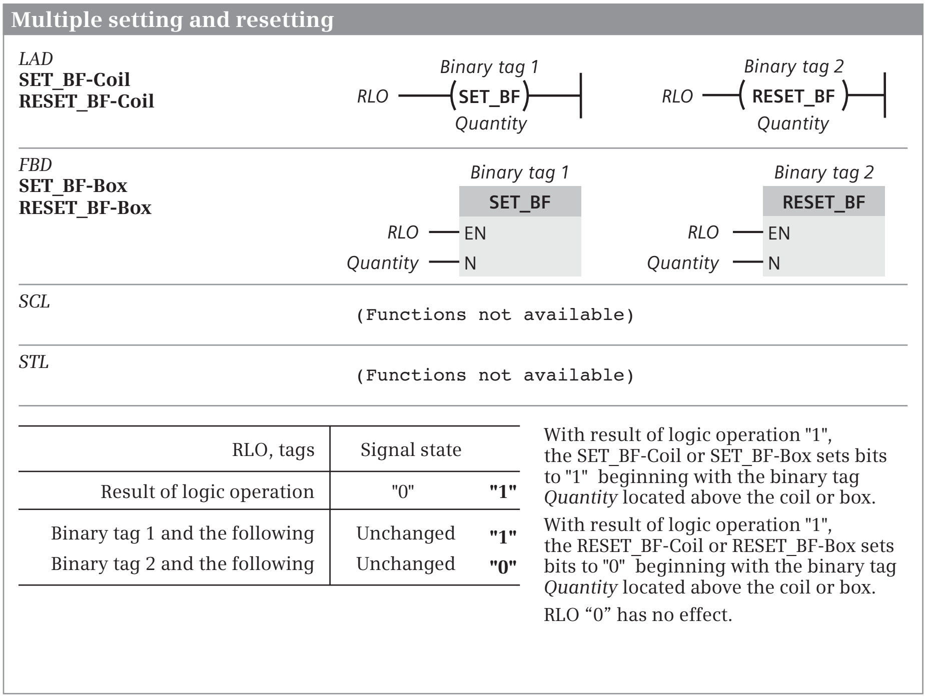
This depends on the operand area used (e.g. static local data) and on the settings in the CPU (e.g. retentive behavior).

12.2.4 Multiple setting and resetting

With multiple setting and resetting, the bits are set in the specified destination area to signal state “1” (SET_BF) or to signal state “0” (RESET_BF).

Multiple setting and resetting is represented 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. 12.9).

Multiple setting and resetting is represented 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 enable 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.



RLO, tags	Signal state	
Result of logic operation	"0"	**"1"**
Binary tag 1 and the following	Unchanged	**"1"**
Binary tag 2 and the following	Unchanged	**"0"**

With result of logic operation "1", the SET_BF-Coil or SET_BF-Box sets bits to "1" beginning with the binary tag *Quantity* located above the coil or box.

With result of logic operation "1", the RESET_BF-Coil or RESET_BF-Box sets bits to "0" beginning with the binary tag *Quantity* located above the coil or box.

RLO “0” has no effect.

Fig. 12.9 Multiple setting and resetting

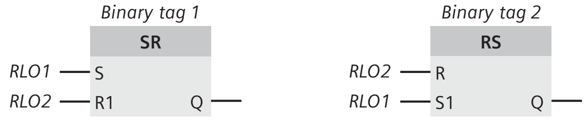
The functions SET_BF and RESET_BF are not present in SCL and STL. In SCL, SET_BF and RESET_BF can be emulated, for example, using the FOR statement. An example is given in the description of the control statements in section “CONTINUE statement” on page 389. In STL they can be emulated with a program loop. An example is shown in Chapter 10.6.1 “Jump functions in the statement list” on page 436.

12.2.5 Dominant setting and resetting, memory function

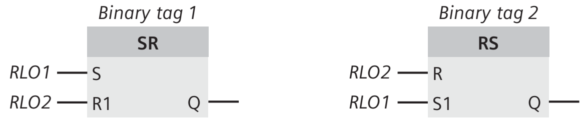
The functions of single setting and resetting are combined in a memory box. The common binary tag is named above the box. The S or S1 input corresponds to single setting, the R or R1 input to single resetting. The signal state possessed by the binary tag named above the memory function is present at the Q output of the memory function.

Dominant setting and resetting

LAD
SR box, RS box



FBD
SR box, RS box



SCL
Emulation with the IF statement

```
IF (RLO2)
   THEN Binary tag 1:= FALSE; //Reset dominant
   ELSIF (RLO1)
      THEN Binary tag 1 := TRUE;
END_IF;

IF (RLO1)
  THEN Binary tag 2:= TRUE;   //Set dominant
  ELSIF (RLO2)
     THEN Binary tag 2 := FALSE;
END_IF;
```

STL
Emulation by the sequence of set and reset statements

```
...       //(RLO1)          ...       //(RLO2)
S  Binary tag 1             R  Binary tag 2

...       //(RLO2)          ...       //(RLO1)
R  Binary tag 1             S  Binary tag 2
```

RLO, tags	Signal state			
RLO 1 (set)	"0"	"1"	"0"	**"1"**
RLO 2 (reset)	"0"	"0"	"1"	**"1"**
Binary tag 1 (SR box)	-	"1"	"0"	**"0"**
Binary tag 2 (RS box)	-	"1"	"0"	**"1"**

Result of logic operation "1" at the set input sets the binary tag, result of logic operation "1" at the reset input resets it.
If both inputs are "1" simultaneously, the binary tag at the SR box is reset and the binary tag at the RS box is set.

Fig. 12.10 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 the label, the two boxes also differ in the positioning of the set input and reset input (Fig. 12.10).

A memory function is set (or more precisely: the binary tag named above the memory box) when the set input has signal state “1” and the reset input has 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 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.

With SCL, the memory function can be emulated, for example, by an assignment together with an IF statement. With STL, the memory function is implemented by single setting and resetting. The sequence of statements defines whether setting or resetting is dominant.

Note that the binary tag used for a memory function may be reset during startup 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 the settings in the CPU (e.g. retentive behavior).

12.3 Edge evaluation

12.3.1 Principle of operation of an edge evaluation

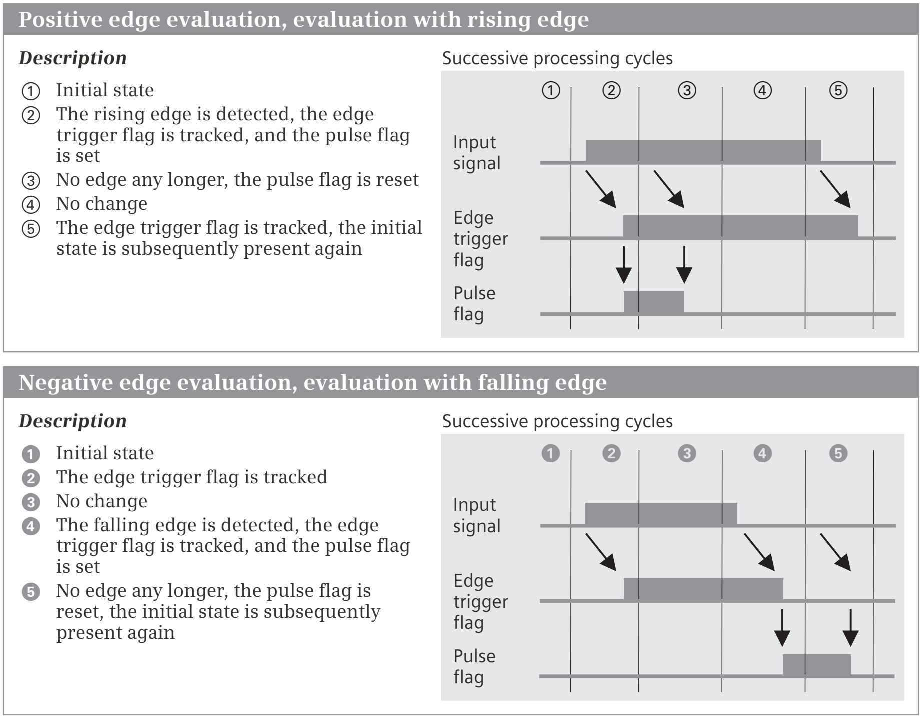
Edge evaluation detects the change in a signal state, a signal edge. A positive (rising) edge is present if the signal changes from state “0” to state “1”. In the reverse case one speaks of a negative (falling) edge.

In the circuit diagram, the equivalent of an edge evaluation is a passing contact. If this passing contact outputs a pulse when the relay is switched on, this corresponds to the rising edge. A pulse of the passing 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 processor 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 (Fig. 12.11).

The saved result of logic operation (RLO) is present in a so-called “edge trigger flag” (this need not necessarily be a bit memory). It must be a binary operand whose signal state must be available again during the next processing of the edge evaluation (in the next program cycle) and which you do not otherwise use in the program. Memory bits, data bits in global data blocks, and static local data bits in function blocks are suitable as operands.

This edge trigger flag saves the “old” RLO, namely the result of logic operation with which the processor processed the edge evaluation last. If a signal edge is now pres-



Positive edge evaluation, evaluation with rising edge

Description

① Initial state
② The rising edge is detected, the edge trigger flag is tracked, and the pulse flag is set
③ No edge any longer, the pulse flag is reset
④ No change
⑤ The edge trigger flag is tracked, the initial state is subsequently present again

Negative edge evaluation, evaluation with falling edge

Description

❶ Initial state
❷ The edge trigger flag is tracked
❸ No change
❹ The falling edge is detected, the edge trigger flag is tracked, and the pulse flag is set
❺ No edge any longer, the pulse flag is reset, the initial state is subsequently present again

Fig. 12.11 Principle of operation of an edge evaluation in successive cycles

ent, i.e. if the current RLO is different from the signal state of the edge trigger flag, the processor updates the signal state of the edge trigger flag in that it assigns the current RLO to it. The signal state of the edge trigger flag is equal to the current RLO (if this has not changed again in the meantime) during the next processing of the edge evaluation (usually in the next program cycle), and the processor does not detect an edge any more.

A detected edge is indicated by the RLO following edge evaluation. If the processor detects a signal edge, it sets the RLO following edge evaluation to “1”. The RLO is equal to “0” if a signal edge is not present.

Signal state “1” following an edge evaluation therefore means “Edge detected”. Signal state “1” is only present for a brief time, usually only for one processing cycle. Since the processor 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 resets the RLO to “0” again following edge evaluation.

You can directly process the RLO following edge evaluation, e.g. link it further to binary logic operations, save it in a memory function, or assign it to a binary tag (a so-called “pulse flag”). A pulse flag is used if the RLO of the edge evaluation is also to be processed at another position in the program; it is quasi the intermediate memory for a detected edge (the passing contact in the circuit diagram). Memory bits, data bits in global data blocks, and temporary and static local data bits are suitable as pulse flags.

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 positive edge: 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 input 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 input signal is reset to state “0” again, the edge trigger flag is also updated and the initial state is reached. The pulse flag was set to “1” for one processing cycle only.

Principle of operation of negative edge: ❶ In the initial state, the input signal, the edge trigger flag, and the pulse flag have signal state “0”. ❷ The input signal then changes its state from “0” to “1”. This change is saved in the edge trigger flag, which is also set to “1”. The pulse flag remains “0” since a falling edge is not present. ❸ No changes take place in the next processing cycles. ❹ The input signal then changes from “1” to “0”. The edge trigger flag still has signal state “1” initially and a falling edge is therefore detected. The pulse flag is set to “1” and the edge trigger flag updated to “0”. ❺ The pulse flag is reset to “0” again. The pulse flag was set to “1” for one processing cycle only.

12.3.2 Edge evaluation of a binary tag (LAD, FBD)

This edge evaluation monitors the signal state of a binary tag for a change. Fig. 12.12 shows the display and the signal states of the edge evaluation. The principle of operation of edge evaluation is described in detail in Fig. 12.11.

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 trigger flag underneath. The Q output corresponds to the pulse flag. A positive, rising edge is detected by the P box and a negative, falling edge is detected by the N box.

Edge evaluation of a binary tag (LAD, FBD)

The **P contact** or the **P box** detects a positive edge of the binary tag above it and then sets the RLO to signal state "1" for the duration of one cycle (in processing cycle ② in the table).

LAD: The P contact links the RLO in front of it to the pulse flag according to AND (series connection).

Positive (rising) edge

Processing cycles	①	②	③	④	⑤
Binary tag 1	**"0"**	**"1"**	**"1"**	"1"	"0"
Edge trigger flag 1	**"0"**	**"1"**	**"1"**	"1"	"0"
RLO1	**"0"**	**"1"**	**"0"**	"0"	"0"

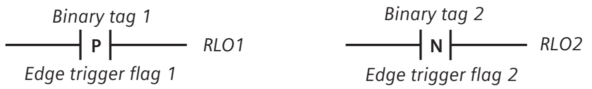
The **N contact** or the **N box** detects a negative edge of the binary tag above it and then sets the RLO to signal state "1" for the duration of one cycle (in processing cycle ❹ in the table).

LAD: The N contact links the RLO in front of it to the pulse flag according to AND (series connection).

Negative (falling) edge

Processing cycles	❶	❷	❸	❹	❺
Binary tag 2	"0"	"1"	**"1"**	**"0"**	**"0"**
Edge trigger flag 2	"0"	"1"	**"1"**	**"0"**	**"0"**
RLO2	"0"	"0"	**"0"**	**"1"**	**"0"**

LAD
P contact,
N contact



FBD
P box,
N box

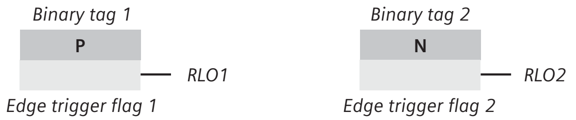


Fig. 12.12 Edge evaluation of a binary tag (LAD, FBD)

12.3.3 Edge evaluation with pulse output (LAD, FBD)

This edge evaluation generates a pulse on a binary tag from the change of the result of logic operation (of the "current flow"). The function of the edge evaluation is shown in Fig. 12.13; the description corresponds to that in Fig. 12.11. Here, the input signal corresponds to the result of preceding logic operation (the "current flow").

The edge evaluation with pulse output with a P coil (positive, rising edge) or an N coil (negative, falling edge) is represented in the ladder logic. The pulse flag, which has signal state "1" for the duration of one processing cycle when an edge is detected, is located above the coil. The edge trigger flag is underneath the coil. The result of logic operation after the coil corresponds to the result of logic operation before the coil; it is simply "passed on".

The edge evaluation with pulse output with a P= box (positive, rising edge) or an N= box (negative, falling edge) is represented in the ladder logic. The pulse flag, which has signal state "1" for the duration of one processing cycle when an edge is detected, is located above the box. The edge trigger flag is underneath the box. The result of logic operation after the box corresponds to the result of logic operation before the box; it is simply "passed on".

Edge evaluation with pulse output

P coil, P= box: If there is a *positive* signal edge before the P coil or at the input of the P= box (RLO1), 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 (RLO1).

Positive (rising) edge

Processing cycles	①	②	③	④	⑤
RLO1	**"0"**	**"1"**	**"1"**	"0"	"0"
Edge trigger flag 1	**"0"**	**"1"**	**"1"**	"0"	"0"
Pulse flag 1	**"0"**	**"1"**	**"0"**	"0"	"0"

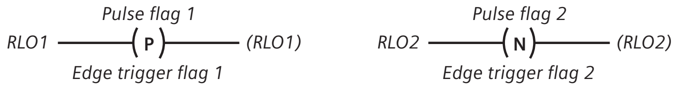
N coil, N= box: If there is a *negative* signal edge before the N coil or at the input of the N= box (RLO2), pulse flag 1 has signal state "1" for the duration of one cycle (in processing cycle d in the table).

After the N coil or N= box, there is the same RLO as before the coil or box (RLO2).

Negative (falling) edge

Processing cycles	❶	❷	❸	❹	❺
RLO2	"0"	**"1"**	**"0"**	**"0"**	"0"
Edge trigger flag 2	"0"	**"1"**	**"0"**	**"0"**	"0"
Pulse flag 2	"0"	**"0"**	**"1"**	**"0"**	"0"

LAD
P coil
N coil



FBD
P= box
N= box

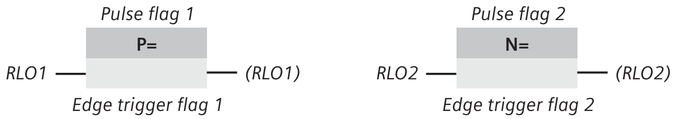


Fig. 12.13 Edge evaluation with pulse output (LAD, FBD)

12.3.4 Edge evaluation with a Q box (LAD, FBD)

This edge evaluation generates a pulse when the result of logic operation changes (change in “current flow”). Fig. 12.14 shows the representation and signal states of edge evaluation. The principle of operation of edge evaluation is described in detail in Fig. 12.11; here, the input signal corresponds to the result of logic operation.

The edge evaluation with a Q box is displayed in the ladder logic with a P_TRIG box (positive, rising edge) or an N_TRIG box (negative, falling edge). The result of the preceding logic operation is monitored at the input CLK for a change. For an edge, the output Q is set to signal state “1” for the duration of one processing cycle. The edge trigger flag is underneath the box.

The edge evaluation with a Q box is displayed in the function block diagram with a P_TRIG box (positive, rising edge) or an N_TRIG box (negative, falling edge). The result of the preceding logic operation is monitored at the input CLK for a change. For an edge, the output Q is set to signal state “1” for the duration of one processing cycle. The edge trigger flag is underneath the box.

Edge evaluation with a Q box (LAD, FBD)

The **P_TRIG box** detects a positive edge at the CLK input and then sets the Q output to signal state "1" for the duration of one processing cycle (in processing cycle ② in the table).

Positive (rising) edge

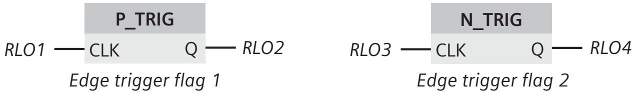
Processing cycles	①	②	③	④	⑤
RLO1	**"0"**	**"1"**	**"1"**	"1"	"0"
Edge trigger flag 1	**"0"**	**"1"**	**"1"**	"1"	"0"
RLO2	**"0"**	**"1"**	**"0"**	"0"	"0"

The **N_TRIG box** detects a negative edge at the CLK input and then sets the Q output to signal state "1" for the duration of one processing cycle (in processing cycle ❹ in the table).

Negative (falling) edge

Processing cycles	❶	❷	❸	❹	❺
RLO3	"0"	"1"	**"1"**	**"0"**	**"0"**
Edge trigger flag 2	"0"	"1"	**"1"**	**"0"**	**"0"**
RLO4	"0"	"0"	**"0"**	**"1"**	**"0"**

LAD
P_TRIG box,
N_TRIG box



FBD
P_TRIG box,
N_TRIG box

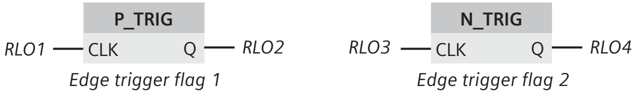


Fig. 12.14 Edge evaluation with a Q box (LAD, FBD)

12.3.5 Edge evaluation with an EN/ENO box (LAD, FBD)

The edge evaluation with an EN/ENO box can be called as a single instance with its own data block or, in a function block, as a local instance (multi-instance). The instance data contains the edge trigger flag needed for edge detection, the input signal CLK and the output signal Q.

This edge evaluation generates a pulse when the result of logic operation changes (change in “current flow”). Fig. 12.15 shows the representation and signal states of edge evaluation. The principle of operation of edge evaluation is described in detail in Fig. 12.11; here, the input signal corresponds to the result of logic operation. The input signal, edge trigger flag, and output signal are located in the instance data.

The edge evaluation with an EN/ENO box is displayed in the ladder logic with an R_TRIG box (positive, rising edge) or an F_TRIG box (negative, falling edge). The result of the preceding logic operation at the input CLK is monitored for a change. For an edge, the output Q is set to signal state “1” for the duration of one processing cycle. The edge trigger flag is underneath the box.

Edge evaluation with an EN/ENO box (LAD, FBD)

The **R_TRIG box** detects a positive edge at the CLK input and then sets the Q output to signal state "1" for the duration of one processing cycle (in processing cycle ② in the table).

The edge trigger flag is located in the instance data.

Positive (rising) edge

Processing cycles	①	②	③	④	⑤
RLO1	**"0"**	**"1"**	**"1"**	"1"	"0"
Edge trigger flag	**"0"**	**"1"**	**"1"**	"1"	"0"
RLO2	**"0"**	**"1"**	**"0"**	"0"	"0"

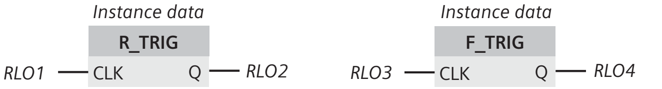
The **F_TRIG box** detects a negative edge at the CLK input and then sets the Q output to signal state "1" for the duration of one processing cycle (in processing cycle ❹ in the table).

The edge trigger flag is located in the instance data.

Negative (falling) edge

Processing cycles	❶	❷	❸	❹	❺
RLO3	"0"	"1"	**"1"**	**"0"**	**"0"**
Edge trigger flag	"0"	"1"	**"1"**	**"0"**	**"0"**
RLO4	"0"	"0"	**"0"**	**"1"**	**"0"**

LAD
R_TRIG box,
F_TRIG box



FBD
R_TRIG box,
F_TRIG box

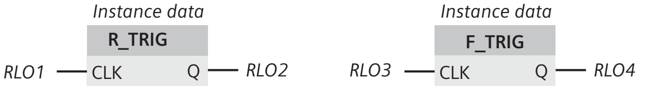


Fig. 12.15 Edge evaluation with an EN/ENO box (LAD, FBD)

The edge evaluation with an EN/ENO box is displayed in the function block diagram with an R_TRIG box (positive, rising edge) or an F_TRIG box (negative, falling edge). The result of the preceding logic operation at the input CLK is monitored for a change. For an edge, the output Q is set to signal state “1” for the duration of one processing cycle. The edge trigger flag is underneath the box.

12.3.6 Edge evaluation with SCL

An edge evaluation can be implemented for SCL, for example, with a pulse flag or with an IF branch. Fig. 12.16 shows the display and the signal states of the edge evaluation. The principle of operation of edge evaluation is described in detail in Fig. 12.11 on Page 516. The *Input signal* tag in Fig. 12.16 can also be replaced with a binary expression.

Edge evaluation with a pulse flag is suitable if the result of the edge evaluation is to be processed in a different program section. For this edge evaluation, the pulse flag is set to signal state “1” if the signal states of the input signal and 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”.

Edge evaluation with SCL

In SCL, the evaluation of a signal state change can be implemented, for example, by means of a pulse flag or IF branch.

A positive, rising edge is present if the input signal has signal state "1" and the edge trigger flag has signal state "0". In the table this corresponds to processing cycle ②, after the edge trigger flag has been updated.
A negative, falling edge is present if the input signal has signal state "0" and the edge trigger flag has signal state "1". In the table this corresponds to processing cycle ❹, after the edge trigger flag has been updated.
For an edge, the pulse flag is set to signal state "1" for the duration of one cycle or – in the case of an IF branch – the statements are processed once after THEN.

Signal states in successive processing cycles

with a positive (rising) edge

Processing cycles	①	②	③	④	⑤
Input signal 1	**"0"**	**"1"**	**"1"**	"1"	"0"
Edge trigger flag 1	**"0"**	**"1"**	**"1"**	"1"	"0"
Pulse flag 1	**"0"**	**"1"**	**"0"**	"0"	"0"

with a negative (falling) edge

Processing cycles	❶	❷	❸	❹	❺
Input signal 2	"0"	"1"	**"1"**	**"0"**	**"0"**
Edge trigger flag 2	"0"	"1"	**"1"**	**"0"**	**"0"**
Pulse flag 2	"0"	"0"	**"0"**	**"1"**	**"0"**

SCL

Edge evaluation with a pulse flag

```
//Evaluation for positive edge

//Modify pulse flag
Pulse flag 1 := input signal 1 AND NOT edge trigger flag 1;

//Update edge trigger flag
Edge trigger flag 1 := Input signal 1;
```

```
//Evaluation for negative edge

//Modify pulse flag
Pulse flag 2 := NOT input signal 2 AND edge trigger flag 2;

//Update edge trigger flag
Edge trigger flag 2 := Input signal 2;
```

SCL

Edge evaluation with an IF statement

```
//Evaluation for positive edge

IF input signal 1 AND NOT edge trigger flag 1
   THEN (* statements *);
       // Program section is executed if there is a pos. edge
       // corresponds to pulse flag 1 = "1"
END_IF;

//Update of edge trigger flag
Edge trigger flag 1 := Input signal 1;
```

```
//Evaluation for negative edge

IF NOT input signal 2 AND edge trigger flag 2
   THEN (* statements *);
       // Program section is executed if there is a neg. edge
       // corresponds to pulse flag 2 = "1"
END_IF;

//Update of edge trigger flag
Edge trigger flag 2 := Input signal 2;
```

Fig. 12.16 Edge evaluation with SCL

Edge evaluation with an IF branch is suitable if a program section which, for example, encompasses several statements is to be processed in the event of an edge. An edge is present if the signal states of the input signal and of the edge trigger flag are different. Then the statements after THEN are processed. After this, the edge trigger flag is updated so that no signal edge will be detected during the next processing and the statements after THEN will no longer be processed.

12.3.7 Edge evaluation with STL

For an edge evaluation with STL, the operations FP and FN are available (Fig. 12.17). Chapter 12.3.1 “Principle of operation of an edge evaluation” on page 515 describes in detail how an edge evaluation functions. The result of logic operation before the edge evaluation corresponds to the input signal. The result of logic operation after the edge evaluation corresponds to the pulse flag.

FP detects a positive, rising edge of the result of logic operation which is present during the processing of the operation. FN detects a negative, falling edge. The operand for the operation FP or FN is the edge trigger flag in which the “old” result of logic operation is saved.

Edge evaluation with STL

The **FP statement** detects a positive edge of the result of the logic operation prior to the statement and then sets the result of the logic operation following the statement to the signal state "1" for the duration of one cycle (in processing cycle ② in the table).
The operand at operation FP is the edge trigger flag.

Positive (rising) edge

Processing cycles	①	②	③	④	⑤
RLO1	**"0"**	**"1"**	**"1"**	"1"	"0"
Edge trigger flag 1	**"0"**	**"1"**	**"1"**	"1"	"0"
RLO2	**"0"**	**"1"**	**"0"**	"0"	"0"

The **FN statement** detects a negative edge of the result of the logic operation prior to the statement and then sets the result of the logic operation following the statement to the signal state "1" for the duration of one cycle (in processing cycle ❹ in the table).
The operand at operation FN is the edge trigger flag.

Negative (falling) edge

Processing cycles	❶	❷	❸	❹	❺
RLO3	"0"	"1"	**"1"**	**"0"**	**"0"**
Edge trigger flag 2	"0"	"1"	**"1"**	**"0"**	**"0"**
RLO4	"0"	"0"	**"0"**	**"1"**	**"0"**

STL
FP operation,
FN operation

```
...                    //RLO1     ...                    //RLO3
FP   Edge trigger flag 1          FN   Edge trigger flag 2
...                    //RLO2     ...                    //RLO4
```

Fig. 12.17 Edge evaluation with STL

12.4 SIMATIC timer functions

12.4.1 Overview

The SIMATIC timer functions implement timing processes in the program such as waiting and monitoring times, measurement of a time interval, or the generation of pulses. The progress of the output signal of a started SIMATIC timer function depends on the selected response. The following are available:

- Pulse generation
  The output signal is at least as long as the start signal, but its maximum length is equal to the set duration.
- Extended pulse
  Independent of the duration of the start signal, the output signal is as long as the set duration.
- ON delay
  If the start signal is longer than the set duration, the output signal begins after the set duration and lasts until the end of the start signal.
- Retentive ON delay
  Independent of the duration of the start signal, the output signal begins after the set duration and ends when the timer function is reset.
- OFF delay
  The output signal begins with the start signal and ends after the set duration has elapsed after the end of the start signal.

A data record which is present in the system data is permanently assigned to each SIMATIC timer function; this limits the number of SIMATIC timer functions. SIMATIC timer functions are global tags; the symbols are declared in the PLC tag table.

The SIMATIC timer functions run in STARTUP and RUN modes.

SIMATIC timers as overall function

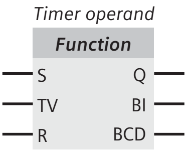
The overall function is represented in the programming languages LAD and FBD as a box (Fig. 12.18). The box of a timer function contains the related representation of all individual timer operations in the form of function inputs and outputs. The address of the timer function is named above the box in absolute or symbolic form. The timer response is quasi the heading in the box. Assignment of the S and TV inputs is mandatory, assignment of the other inputs and outputs is optional. With SCL, the complete function call corresponds to the overall function. With STL, the individual statements must be programmed in the indicated sequence.

SIMATIC timers as single elements

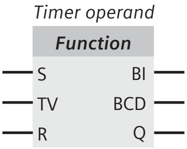
In the representation as single elements, attention must be paid to the programming sequence so that the timer function responds as described further below: First program the start statement, then the reset statement, and finally scan the

SIMATIC timer functions, representation as total function

LAD



FBD



Parameters for LAD and FBD:

Name	Declaration	Data type	Description
S	INPUT	BOOL	Start input
TV	INPUT	TIME	Defined duration
R	INPUT	BOOL	Reset input
Q	OUTPUT	BOOL	Timer status
BI	OUTPUT	TIME	Time value integer
BCD	OUTPUT	S5TIME	Time value BCD

STL

```
A     Enable input
FR    Timer operand
A     Start input
L     Duration
FCT   Timer operand
A     Reset input
R     Timer operand
L     Timer operand
T     Time value integer
LC    Timer operand
T     Time value BCD
A     Timer operand
=     Timer status
```

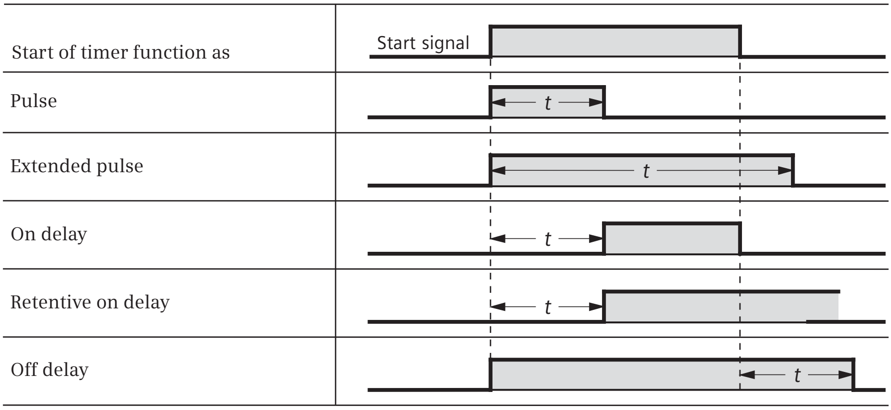
SCL

```
Time value BCD   := Function
       T_NO      := Timer operand,
       S         := Start input,
       TV        := Duration,
       R         := Reset input,
       Q         := Timer status,
       BI        := Time value integer);
```

Function identifier

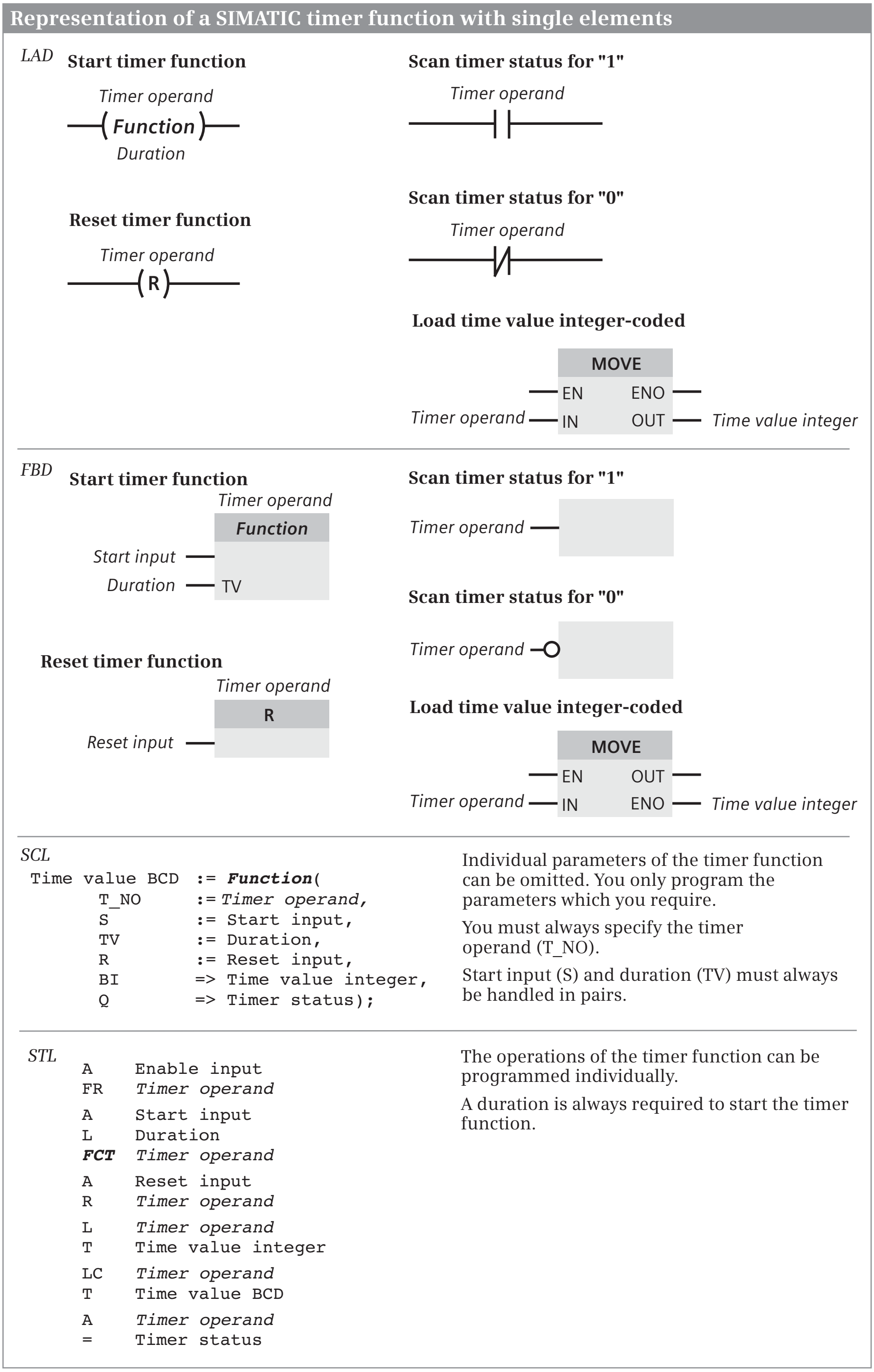
Start of timer function as	With LAD and FBD as Box	Single element	With SCL with	With STL with
Pulse	S_PULSE	SP	S_PULSE	SP
Extended pulse	S_PEXT	SE	S_PEXT	SE
On delay	S_ODT	SD	S_ODT	SD
Retentive on delay	S_ODTS	SS	S_ODTS	SS
Off delay	S_OFFDT	SF	S_OFFDT	SF

Pulse diagram



t = set duration

Fig. 12.18 SIMATIC timer functions as overall function



Representation of a SIMATIC timer function with single elements

```
Time value BCD  := Function(
       T_NO     := Timer operand,
       S        := Start input,
       TV       := Duration,
       R        := Reset input,
       BI       => Time value integer,
       Q        => Timer status);
```

Individual parameters of the timer function can be omitted. You only program the parameters which you require.

You must always specify the timer operand (T_NO).

Start input (S) and duration (TV) must always be handled in pairs.

```
A     Enable input
FR    Timer operand

A     Start input
L     Duration
FCT   Timer operand

A     Reset input
R     Timer operand

L     Timer operand
T     Time value integer

LC    Timer operand
T     Time value BCD

A     Timer operand
=     Timer status
```

The operations of the timer function can be programmed individually.

A duration is always required to start the timer function.

Fig. 12.19 SIMATIC timer functions, representation as single elements

timer function. If the enabling statement is used with STL, it must be programmed prior to the start statement (Fig. 12.19).

When programming a timer function, you need not use all statements available for the timer function. It is sufficient to use the statements required for the desired function. In the normal case these are the starting of the timer function with specification of the duration and the binary scanning of the timer function.

12.4.2 Programming a timer function

Starting a timer function

A timer function is started, for example, using a binary tag. In the figures this tag is named *Start input*.

A timer function starts (the time starts running) when the signal state of the start input changes. Such a change in signal state is always required to start a timer function. In the case of an OFF delay, the signal state must change from “1” to “0”, in all other cases the time starts when changing from “0” to “1”.

You can start every timer function using one of five possible responses. However, it is not meaningful to assign several responses to one timer function.

Specification of duration

When starting, the timer function is loaded with a default value of data type S5TIME. In the figures this default value is named *Duration*. The duration can be specified as a constant or as a tag.

The duration is calculated internally from the time value and time scale: Duration = Time value × Time scale. The duration is the time during which a timer function is active (“time running”). The time value represents the number of time periods for which the timer function runs. The time scale specifies the interval at which the CPU's operating system changes the time value (Fig. 12.20).

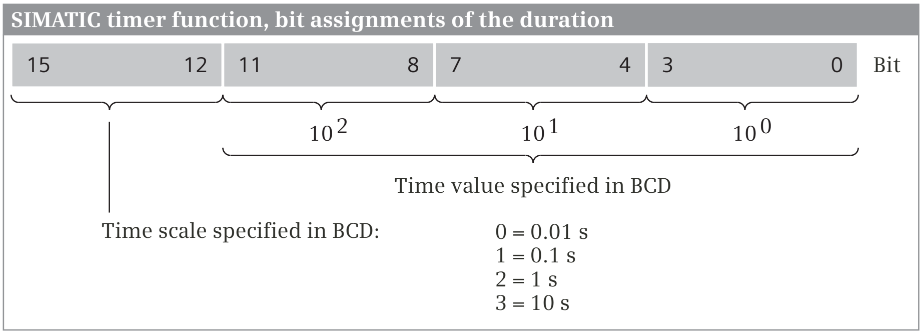

SIMATIC timer function, bit assignments of the duration
15
12
11
8
7
4
3
0
Bit
10^2
10^1
10^0
Time value specified in BCD
Time scale specified in BCD:
0 = 0.01 s
1 = 0.1 s
2 = 1 s
3 = 10 s

Fig. 12.20 Bit assignment of the duration with a SIMATIC timer function

You can also directly establish the duration in a word operand. The smaller you select the time scale, the more exact is the actually processed duration. For example, if you wish to implement a duration of one second, three possibilities exist:

Duration = W#16#2001	Time scale 1 s
Duration = W#16#1010	Time scale 100 ms
Duration = W#16#0100	Time scale 10 ms

The last possibility should be preferred in this example.

When starting the timer function, the CPU applies the programmed time value. The operating system updates the timer functions at a fixed interval and independent of processing of the user program, i.e. with active timers it counts down the count value at the interval of the time scale. The time is considered to be expired when a value of zero is reached. The CPU then sets the timer status (signal state "0" or "1" depending on time response) and omits all further activities until the timer function is started again. If you enter a value of zero (0 ms or W#16#0000) for the duration when starting a timer function, the timer function remains active until the CPU has processed the timer function and established that the time has expired.

The timer functions are updated asynchronous to program execution. It may therefore occur that the timer status at the beginning of the cycle has a different value from that at the end. If you only use the timer operations at one position in the program, no malfunctions can occur due to asynchronous time updating.

Resetting a timer function

A timer function is reset, for example, using a binary tag. In the figures, this tag is named *Reset input*.

A timer function is reset as long as the reset input has signal state "1". Resetting of the timer function sets the time value and the time scale to zero and the timer status to "0". Starting of the timer function is not possible for as long as the reset is present.

Note with STL: Resetting a timer function does not reset the internal edge trigger flag for starting. To start again, the start operation must first be processed with RLO "0" before the timer function can be started with a signal edge. You can also use enabling of the timer function for this.

Scanning the timer status

The timer status is as it were the "result" of the timer function. Its time response is based on the response of the timer function. For example, the timer status can be assigned to a binary tag. In the figures this tag is named *Timer status*.

The time response of the timer status is shown in general in Fig. 12.18 and described in detail further below in the descriptions of the responses of a timer function.

Scanning the current time value BCD-coded

The time value is the current value of the “remaining time” at the time of scanning. With a timer function running, the time value is counted down from the defined duration to zero. In the BCD-coded form, the remaining time contains the time scale and duration in data type WORD or S5TIME. In the figures, this tag is named *Time value BCD*.

Scanning the current time value integer-coded

The time value is the current value of the “remaining time” at the time of scanning. With a timer function running, the time value is counted down from the defined duration to zero. In the integer-coded form, the remaining time only delivers the current magnitude of the time value, the time scale is not included. In the figures, this tag is named *Time value integer* with data type INT.

Enabling a timer function

A running time is “re-triggered” by enabling, i.e. a restart is triggered. Enabling is only available in the programming language STL; it is not required to start or reset a timer function, i.e. for normal execution.

Enabling is triggered by a positive edge at the enabling operation, for example by a binary tag. In the figures, this tag is named *Enable input*.

Enabling resets the internal edge trigger flag for starting the timer function. If the result of logic operation is “1” during the next processing of the start operation, the timer function is started again.

Example: A timer function is started as a pulse time by a positive edge at the start input. The start input remains permanently at signal state “1”. The time can then be restarted by a positive edge at the enable input without resulting in a change in the signal state at the start input. It is irrelevant whether the time is still running or has already expired.

12.4.3 Timer response as pulse

Starting a pulse time

The diagram in Fig. 12.21 describes the response of the timer function following starting as pulse and when resetting.

① The timer function starts if the signal state at its start input changes from “0” to “1” (positive edge). It runs for the programmed duration as long as the signal state at the start input remains “1”. The scans for signal state “1” (the timer status) deliver the result of scan “1” for as long as the time is running. The time value is counted down from the start value according to the set scale.

② The timer function stops if the signal state at its start input changes to “0” before the time has expired. The scan of the timer function for signal state “1” (the timer status) then delivers the result of scan “0”. The time value indicates the remaining duration by which the time was interrupted too early.

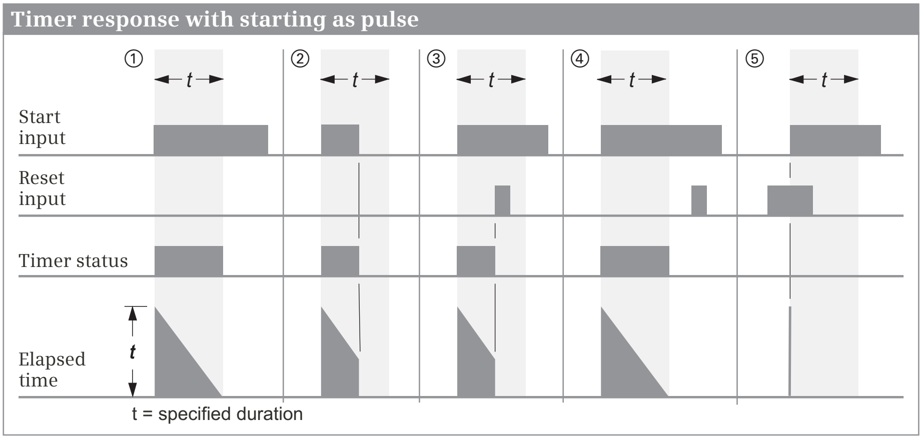


Fig. 12.21 Timer response with starting and resetting as pulse

Resetting a pulse time

Resetting a pulse time has a static effect and has priority over starting of the timer function (Fig. 12.21).

③ Signal state “1” at the reset input of the timer function with the time running resets the timer function. A scan for signal state “1” (the timer status) then delivers the result of scan “0”. The time value and the time scale are also set to zero. If the signal state at the reset input changes from “1” to “0” while signal state “1” is still present at the start input, the timer function remains unaffected.

④ If the time is not running, signal state “1” at the reset input has no effect.

⑤ If a reset signal is present and the signal state at the start input changes from “0” to “1” (positive edge), the timer function is started but the subsequent reset immediately resets it again (indicated by a line in the diagram). If the scan of the timer status is programmed following the reset, the brief starting does not influence the scan of the timer function.

Enabling a pulse time

Enabling is only possible in the programming language STL. The diagram in Fig. 12.22 shows enabling of a timer function started as a pulse.

❶ If the signal state at the enable input changes from “0” to “1” (positive edge) while the time is running, the time for processing of the start operation restarts as long as signal state “1” is still present at the start input. With this restart, the programmed duration is applied as the current time value. A change in the signal state at the enable input from “1” to “0” has no effect.

❷ If the signal state at the enable input changes from “0” to “1” (positive edge) while the time is not running and signal state “1” is still present at the start input, the timer function also starts with the programmed duration as pulse.

❸ With signal state “0” at the start input, a positive signal edge at the enable input has no effect.

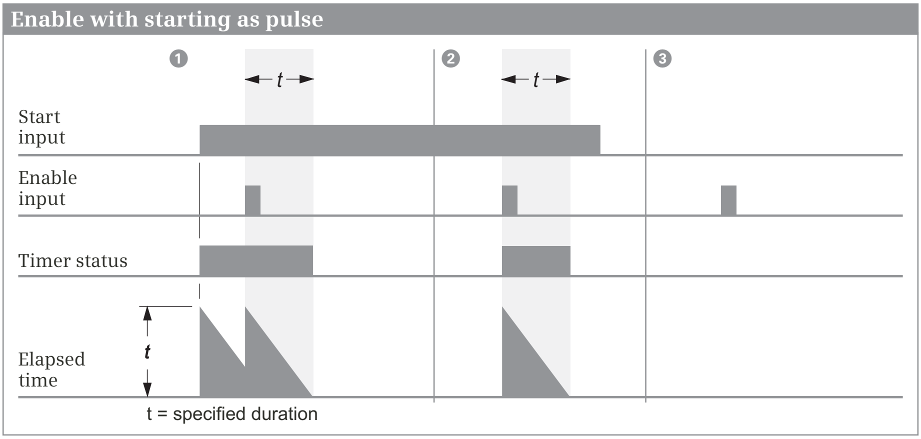


Fig. 12.22 Enabling with a pulse time

12.4.4 Timer response as extended pulse

Starting as extended pulse

The diagram in Fig. 12.23 describes the response of the timer function following starting as extended pulse and when resetting.

①② The timer function starts if the signal state at its start input changes from “0” to “1” (positive edge). It runs for the programmed duration even if the signal state at the start input returns to “0”. The scans for signal state “1” (the timer status) deliver the result of scan “1” for as long as the time is running. The time value is counted down from the start value according to the set scale.

③ The timer function starts again with the programmed time value (the timer function is “retriggered”) if the signal state at the start input changes from “0” to “1” (positive edge) while the time is running. It can be restarted any number of times without expiring.

Resetting with extended pulse

Resetting a time started as a extended pulse has a static effect and has priority over starting of the timer function (Fig. 12.23).

④⑤ Signal state “1” at the reset input of the timer function with the time running resets the timer function. A scan for signal state “1” (timer status) delivers the result of scan “0” if the timer function is reset. The time value and the time scale are also set to zero.

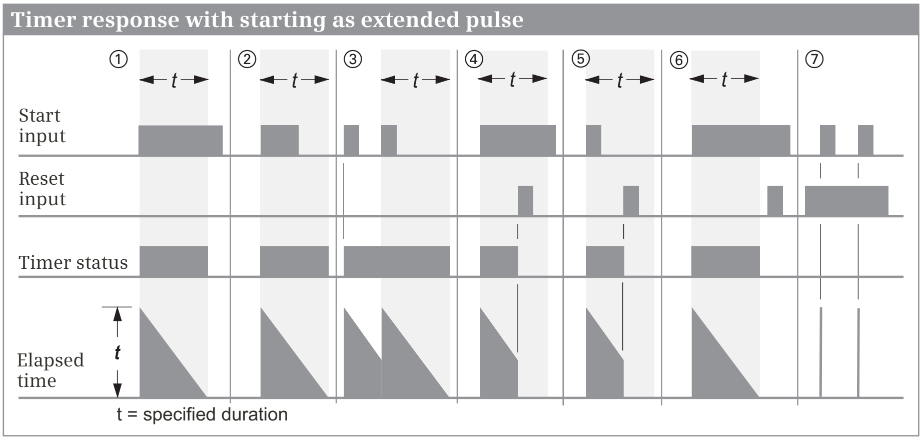


Fig. 12.23 Timer response as extended pulse

⑥ Processing of the reset input with signal state "1" has no effect when the time is not running.

⑦ If a reset signal is present and the signal state at the start input changes from "0" to "1" (positive edge), the timer function is started but the subsequent reset immediately resets it again (indicated by a line in the diagram). If the scan of the timer status is programmed following the reset, the brief starting does not influence the scan of the timer function.

Enabling with extended pulse

Enabling is only possible in the programming language STL. The diagram in Fig. 12.24 shows enabling of a timer function started as a extended pulse.

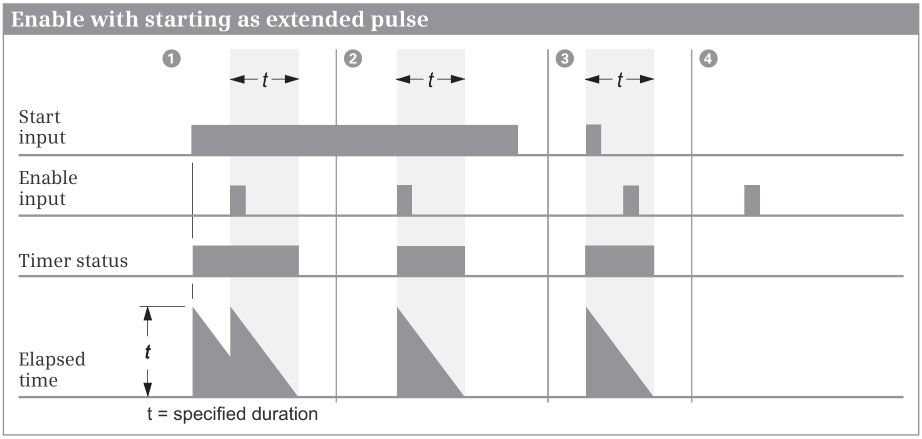


Fig. 12.24 Enabling with extended pulse

❶ If the signal state at the enable input changes from “0” to “1” (positive edge) while the time is running, the time for processing of the start operation restarts as long as signal state “1” is still present at the start input. With this restart, the programmed duration is applied as the current time value. A change in the signal state at the enable input from “1” to “0” has no effect.

❷ If the signal state at the enable input changes from “0” to “1” (positive edge) while the time is not running and signal state “1” is still present at the start input, the timer function also starts with the programmed duration as extended pulse.

❸❹ With signal state “0” at the start input, a positive signal edge at the enable input has no effect.

12.4.5 Timer response as ON delay

Starting as ON delay

The diagram in Fig. 12.25 describes the response of the timer function following starting as ON delay and when resetting.

① The timer function starts if the signal state at its start input changes from “0” to “1” (positive edge). It expires with the programmed duration. The scans for signal state “1” (timer status) deliver the result of scan “1” if the time has expired correctly and the start input is still controlled by signal state “1” (delayed switch-on). The time value is counted down from the start value according to the set scale.

② The timer function stops if the signal state at the start input changes from “1” to “0” while the time is running. A scan of the timer function for signal state “1” (timer status) always delivers the result of scan “0” in such cases. The time value indicates the remaining duration by which the time was interrupted too early.

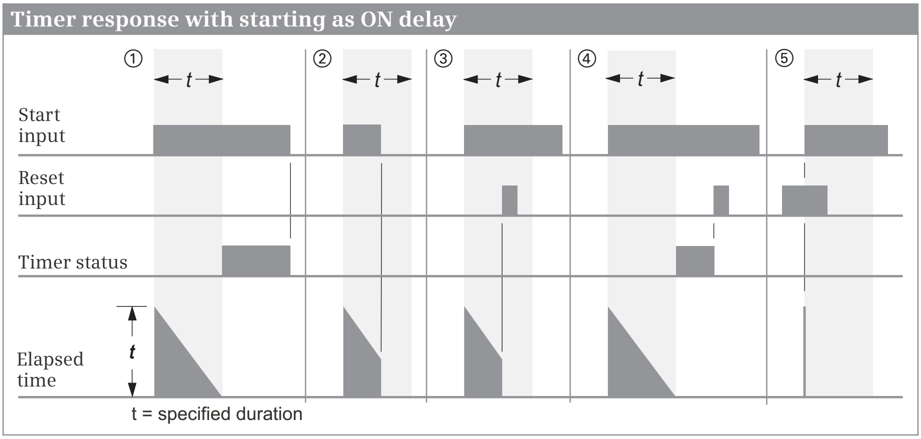


Fig. 12.25 Timer response as ON delay

Resetting as ON delay

Resetting an ON delay has a static effect and has priority over starting of the timer function (Fig. 12.25).

③④ Signal state “1” at the reset input resets the timer function irrespective of whether the time is running or not. A scan for signal state “1” (timer status) then delivers the result of scan “0” even if the time is not running and the signal state “1” is still present at the start input. The time value and the time scale are also set to zero. If the signal state at the reset input changes from “1” to “0” while signal state “1” is still present at the start input, the timer function remains unaffected.

⑤ If a reset signal is present and the signal state at the start input changes from “0” to “1” (positive edge), the timer function is started but the subsequent reset immediately resets it again (indicated by a line in the diagram). If the scan of the timer status is programmed following the reset, the brief starting does not influence the scan of the timer function.

Enabling as ON delay

Enabling is only possible in the programming language STL. The diagram in Fig. 12.26 shows enabling of a timer function as ON delay.

❶ If the signal state at the enable input changes from “0” to “1” (positive edge) while the time is running, the time for processing of the start operation restarts as long as signal state “1” is still present at the start input. With this restart, the programmed duration is applied as the current time value. A change in the signal state at the enable input from “1” to “0” has no effect.

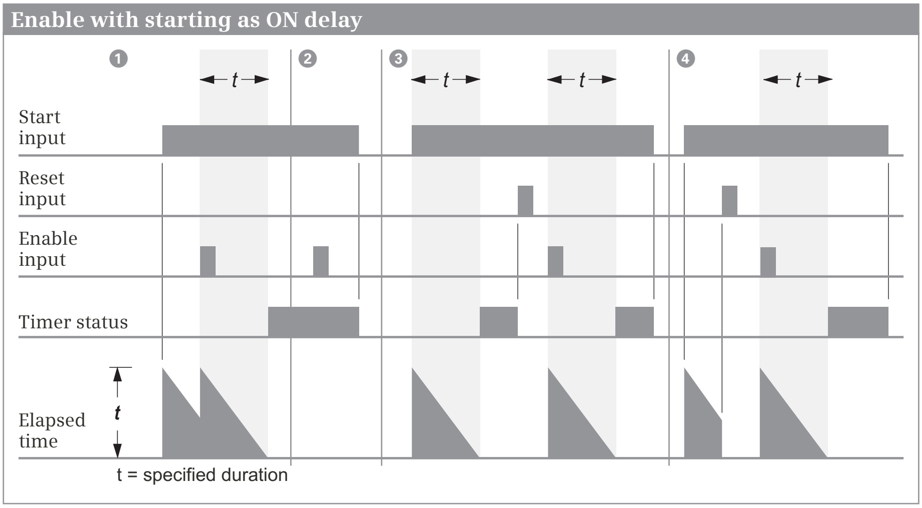


Fig. 12.26 Enabling with ON delay

❷ If the signal state at the enable input changes from “0” to “1” (positive edge) when the time has expired correctly, the timer function remains uninfluenced when the start operation is processed.

❸❹ With the timer function reset, a positive signal edge at the enable input restarts the timer function if signal state “1” is still present at the start input. This restart takes over the programmed duration as current time value.

With signal state “0” at the start input, a positive edge at the enable input has no effect.

12.4.6 Timer response as retentive ON delay

Starting as retentive ON delay

The diagram in Fig. 12.27 describes the response of the timer function following starting and when resetting.

①② The timer function starts if the signal state at its start input changes from “0” to “1” (positive edge). It runs for the programmed duration even if the signal state at the start input returns to “0”. If the time has expired, a scan of the timer function for signal state “1” (timer status) delivers the result of scan “1” independent of the signal state at the start input. The result of scan only becomes “0” again if the timer function has been reset, independent of the signal state at the start input. The time value is counted down from the start value according to the set scale.

③ The timer function starts again with the programmed time value (the timer function is “retriggered”) if the signal state at the start input changes from “0” to “1” (positive edge) while the time is running. It can be restarted any number of times without expiring.

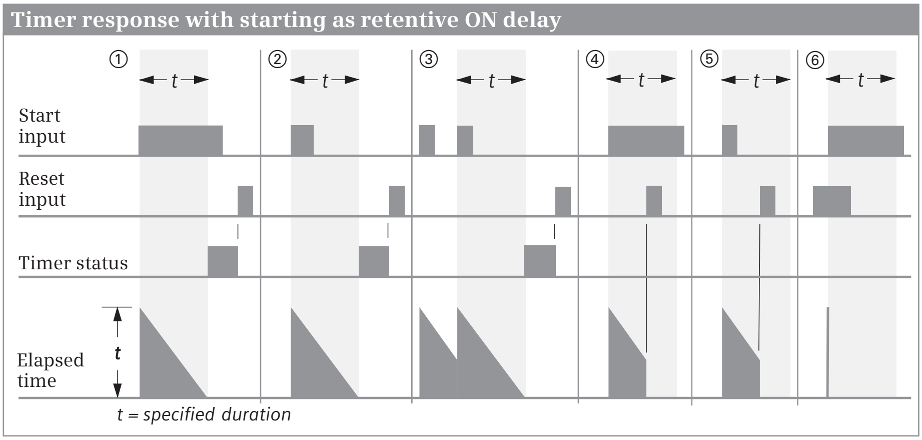


Fig. 12.27 Timer response as retentive ON delay

Resetting as retentive ON delay

Resetting a retentive ON delay has a static effect and has priority over starting of the timer function (Fig. 12.27).

④⑤ Signal state “1” at the reset input resets the timer function independent of the signal state at the start input. The scans of the timer function for signal state “1” (timer status) then deliver the result of scan “0”. The time value and the time scale are set to zero.

⑥ If a reset signal is present and the signal state at the start input changes from “0” to “1” (positive edge), the timer function is started but the subsequent reset immediately resets it again (indicated by a line in the diagram). If the scan of the timer status is programmed following the reset, the brief starting does not influence the scan of the timer function.

Enabling as retentive ON delay

Enabling is only possible in the programming language STL. The diagram in Fig. 12.28 shows enabling of a timer function started as a retentive ON delay.

❶ If the signal state at the enable input changes from “0” to “1” (positive edge) while the time is running, the timer function for processing of the start operation restarts as long as signal state “1” is still present at the start input. With this restart, the timer function takes over the programmed duration as the current time value. A change in the signal state at the enable input from “1” to “0” has no effect.

❷ If the signal state at the enable input changes from “0” to “1” (positive edge) when the time has expired correctly, the timer function remains uninfluenced when the start operation is processed.

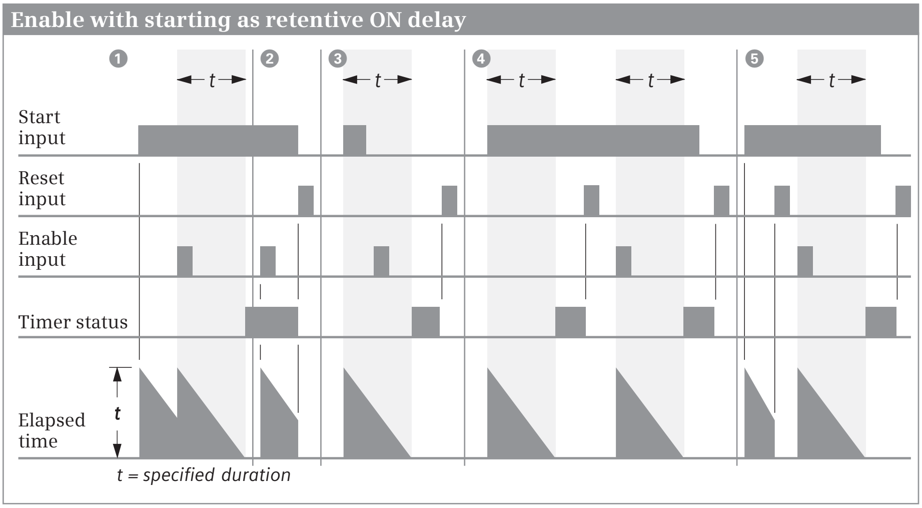


Fig. 12.28 Enabling with retentive ON delay

❸ With signal state “0” at the start input, a positive signal edge at the enable input has no effect.

❹❺ With the timer function reset and signal state “1” at the start input, a positive edge at the enable input restarts the timer function. This restart takes over the programmed duration as current time value.

12.4.7 Timer response as OFF delay

Starting as OFF delay

The diagram in Fig. 12.29 describes the response of the timer function following starting as OFF delay and when resetting.

①③ The timer function starts if the signal state at its start input changes from “1” to “0” (negative edge). It expires with the programmed duration. The scans of the timer function for signal state “1” (timer status) deliver the result of scan “1” if the signal state at the start input is “1” or if the time is running (delayed switch-off). The time value is counted down from the start value according to the set scale.

② The timer function is reset if the signal state at its start input changes from “0” to “1” (positive edge) while the time is running. Only a negative edge at the start input restarts the time.

Resetting as OFF delay

Resetting an OFF delay has a static effect and has priority over starting of the timer function (Fig. 12.29).

④ Signal state “1” at the reset input of the timer function with the time running resets the timer function. The result of scans for signal state “1” (timer status) is then “0”. The time value and the time scale are also set to zero.

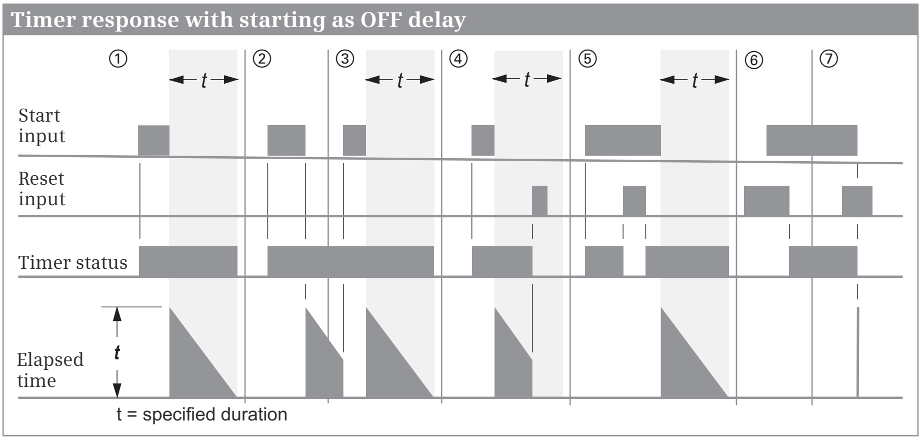


Fig. 12.29 Timer response as OFF delay

⑤⑥ Signal state “1” at the start input and at the reset input resets the binary output of the timer function (a scan of the timer function for signal state “1”, the timer status, then delivers the result of scan “0”). If the signal state at the reset input then changes to “0” again, the output of the timer function has signal state “1” again.

⑦ If a reset signal is present and the signal state at the start input changes from “1” to “0” (negative edge), the timer function is started but the subsequent reset immediately resets it again (indicated by a line in the diagram). The scan for signal state “1” (the timer status) then immediately delivers the result of scan “0”.

Enabling as OFF delay

Enabling is only possible in the programming language STL. The diagram in Fig. 12.30 shows enabling of a timer function started as an OFF delay.

❶ If the signal state at the enable input changes from “0” to “1” (positive edge) when the time is not running, the timer function remains uninfluenced when the start operation is processed. A change in the signal state at the enable input from “1” to “0” has no effect either.

❷ If the signal state at the enable input changes from “0” to “1” (positive edge) when the time is running, the timer function restarts when the start operation is processed. This restart takes over the programmed duration as current time value.

❸ A change in the signal state at the enable input from “0” to “1” (positive edge) or a change in the signal state from “1” to “0” (negative edge) with the time not running has no effect.

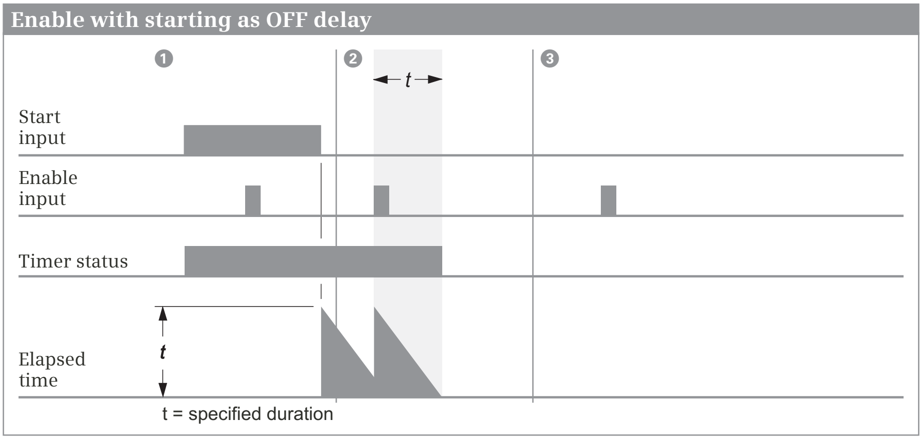


Fig. 12.30 Enabling with OFF delay

12.5 IEC timer functions

12.5.1 Introduction

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. The following IEC timer functions are available:

▷ TP	Pulse generation
▷ TON	ON delay
▷ TOF	OFF delay
▷ TONR	Accumulating ON delay

An IEC timer function is a statement with its own data. When programming a timer 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 timer function in a function block, you can also select *Multi-instance*. In this case the data of the timer function is saved as local instance in the instance data block of the function block.

The duration of an IEC timer function can have the data type TIME or LTIME. Accordingly, the data structure of an IEC timer function is mapped in the system data type (SDT) IEC_TIMER or IEC_LTIMER. The individual components of the data structure are shown in Chapter 4.11.1 "System data types for IEC timer functions" on page 139.

The data of several timer functions can be stored in an instance data block under different names. This reduces the number of required data blocks. With LAD and FBD, 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.

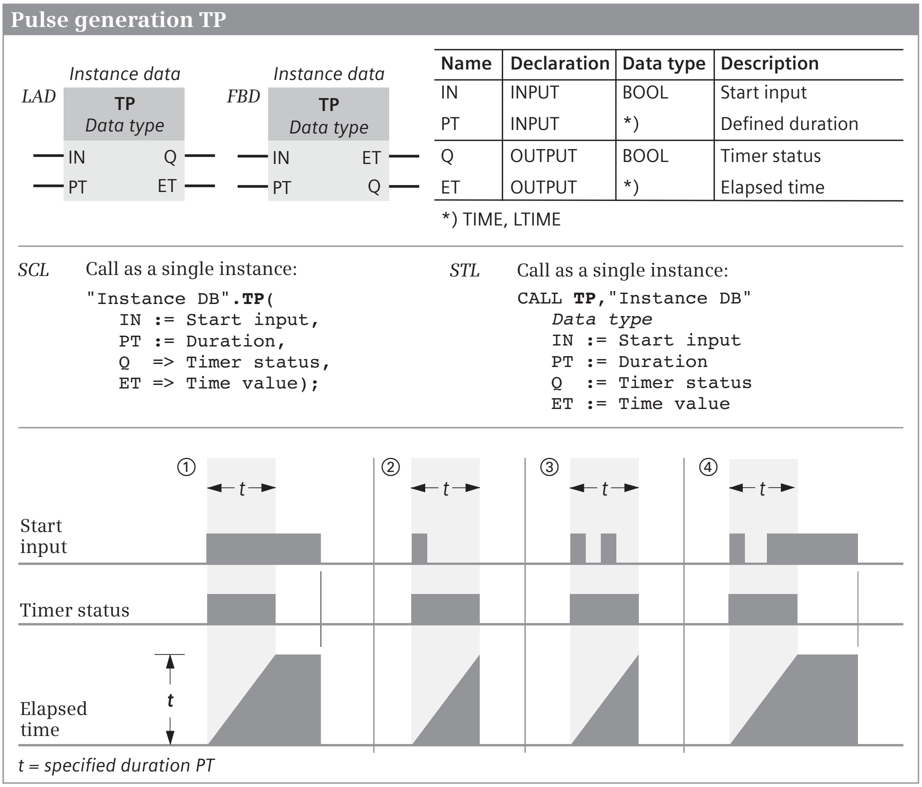
When calling an IEC timer function you must supply the start input IN and the defined duration PT (preset time) with tags. Supplying of the timer status Q and the elapsed time ET is optional. You can scan the time status like a binary tag at any point in the user program with *Instance_Name.Q*.

The timer functions run in 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 you assign the time status and/or the current time value to a tag and then scan only the tag.

12.5.2 Pulse generation TP

The pulse generation shortens or extends an input signal to the programmed duration (Fig. 12.31).



Name	Declaration	Data type	Description
IN	INPUT	BOOL	Start input
PT	INPUT	*)	Defined duration
Q	OUTPUT	BOOL	Timer status
ET	OUTPUT	*)	Elapsed time

*) TIME, LTIME

SCL Call as a single instance:

```
"Instance DB".TP(
   IN := Start input,
   PT := Duration,
   Q  => Timer status,
   ET => Time value);
```

STL Call as a single instance:

```
CALL TP,"Instance DB"
   Data type
   IN := Start input
   PT := Duration
   Q  := Timer status
   ET := Time value
```

Fig. 12.31 Pulse generation TP, representation and function

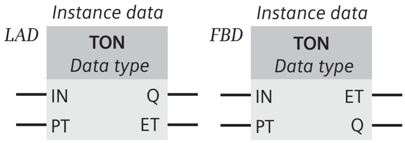
The timer function starts if the signal state at its start input IN changes from “0” to “1”. It runs for the duration programmed at the PT input, independent of the further response of the signal state at the start input. The Q output delivers signal state “1” for as long as the time is running ① ② ③ ④.

The ET output delivers the expired time. This duration commences at T#0s and ends at the preset time PT. If the time has expired, ET remains at the expired value until the signal state at the IN input changes again to “0” ① ④. If the IN input has signal state “0” prior to expiry of the preset time PT, the ET output immediately changes to T#0s following expiry of PT ② ③.

12.5.3 ON delay TON

The ON delay delays an input signal by the programmed duration (Fig. 12.32).

ON delay TON



Name	Declaration	Data type	Description
IN	INPUT	BOOL	Start input
PT	INPUT	*)	Defined duration
Q	OUTPUT	BOOL	Timer status
ET	OUTPUT	*)	Elapsed time

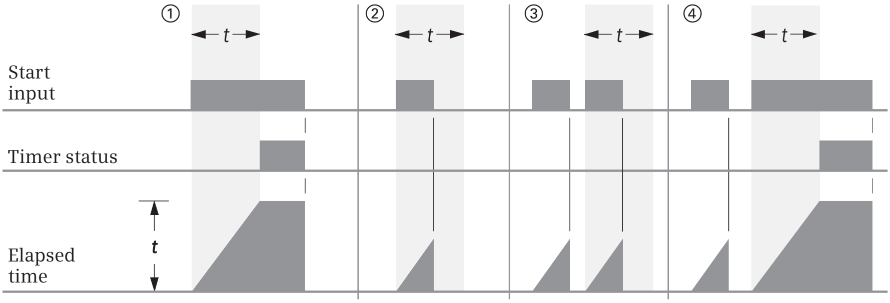
*) TIME, LTIME

SCL Call as a single instance:

```
"Instance DB".TON(
   IN := Start input,
   PT := Duration,
   Q  => Timer status,
   ET => Time value);
```

STL Call as a single instance:

```
CALL TON,"Instance DB"
   Data type
   IN := Start input
   PT := Duration
   Q  := Timer status
   ET := Time value
```



t = specified duration PT

Fig. 12.32 ON delay TON, representation and function

① ④ 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. The Q output delivers signal state "1" if the time has expired and for as long as the start input is still "1".

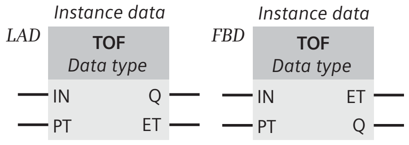
② ③ The time is reset if the signal state at the start input IN changes from "1" to "0" before the time has expired. It starts again with the next positive edge at the IN input.

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" ① ④. If the IN input has signal state "0" prior to expiry of PT, the ET output immediately changes to T#0s ② ③ ④.

12.5.4 OFF delay TOF

The OFF delay delays the switching off of an input signal by the programmed duration (Fig. 12.33).

OFF delay TOF



Name	Declaration	Data type	Description
IN	INPUT	BOOL	Start input
PT	INPUT	*)	Defined duration
Q	OUTPUT	BOOL	Timer status
ET	OUTPUT	*)	Elapsed time

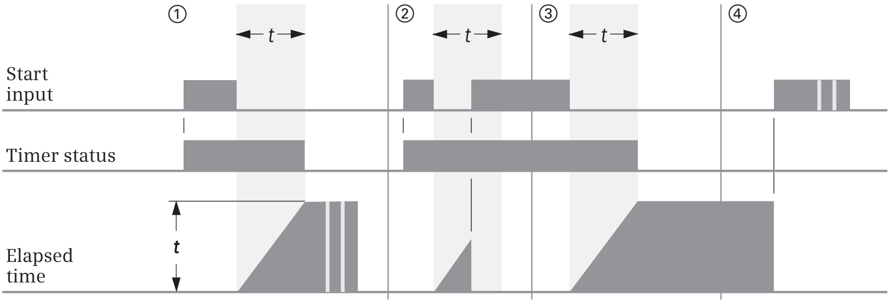
*) TIME, LTIME

SCL Call as a single instance:

```
"Instance DB".TOF(
   IN := Start input,
   PT := Duration,
   Q  => Timer status,
   ET => Time value);
```

STL Call as a single instance:

```
CALL TOF,"Instance DB"
   Data type
   IN := Start input
   PT := Duration
   Q  := Timer status
   ET := Time value
```



t = specified duration PT

Fig. 12.33 OFF delay TOF, representation and function

①③ The Q output has signal state “1” if the signal state at the 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. The Q output remains at signal state “1” for as long as the time is running. The Q output is reset if the time has expired.

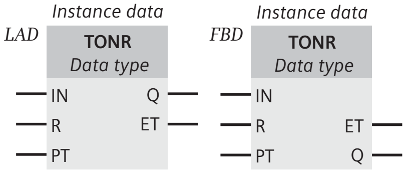
② The duration is reset and the Q output remains “1” if the signal state at the start input changes to “1” again before the time has expired.

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 at the expired value until the IN input has signal state “1” ④. If the IN input has signal state “1” prior to expiry of PT, the ET output immediately changes to T#0s ②.

12.5.5 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. 12.34).

Accumulating ON delay TONR



Name	Declaration	Data type	Description
IN	INPUT	BOOL	Start input
R	INPUT	BOOL	Reset input
PT	INPUT	*)	Defined duration
Q	OUTPUT	BOOL	Timer status
ET	OUTPUT	*)	Elapsed time

*) TIME, LTIME

SCL Call as a single instance:

```
"Instance DB".TONR(
   IN := Start input,
   R  := Reset input,
   PT := Duration,
   Q  => Timer status,
   ET => Time value);
```

STL Call as a single instance:

```
CALL TONR,"Instance DB"
   Data type
   IN := Start input
   R  := Reset input
   PT := Duration
   Q  := Timer status
   ET := Time value
```

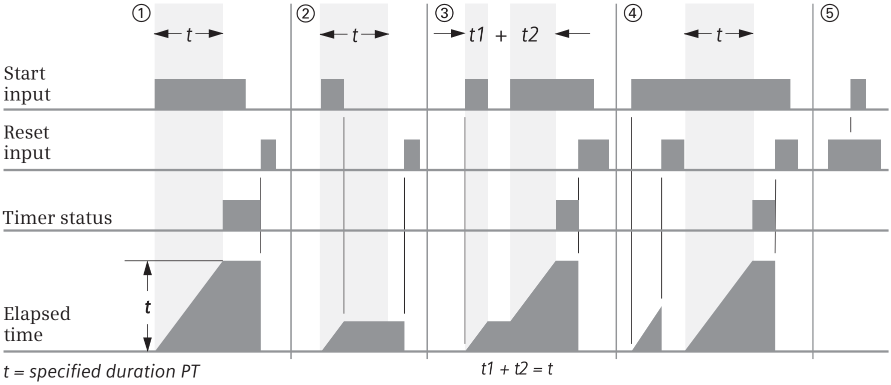


Fig. 12.34 Accumulating ON delay TONR, representation and function

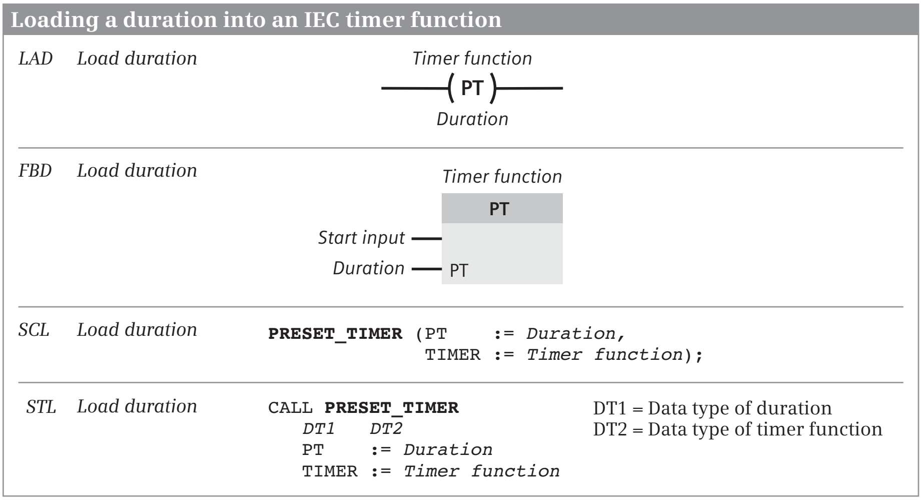
① The timer function starts if the signal state at its start input IN changes from “0” to “1”. It expires with the duration programmed at the PT input. Output Q delivers signal state “1” if the time has expired, regardless of the further course of the signal state at the start input. ② If the signal 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.

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.

12.5.6 Loading an IEC timer function with a duration

The program elements shown in Fig. 12.35 are available for loading an IEC timer function with a duration. For a single instance, the data block is specified as the timer function. For a local instance, the instance name is specified.



```
PRESET_TIMER (PT    := Duration,
              TIMER := Timer function);
```

```
CALL PRESET_TIMER
   DT1   DT2
   PT    := Duration
   TIMER := Timer function
```

Fig. 12.35 Loading a duration into an IEC timer function

LAD: The PT coil is only executed if the signal state before the coil is “1”. Signal state “0” has no effect.

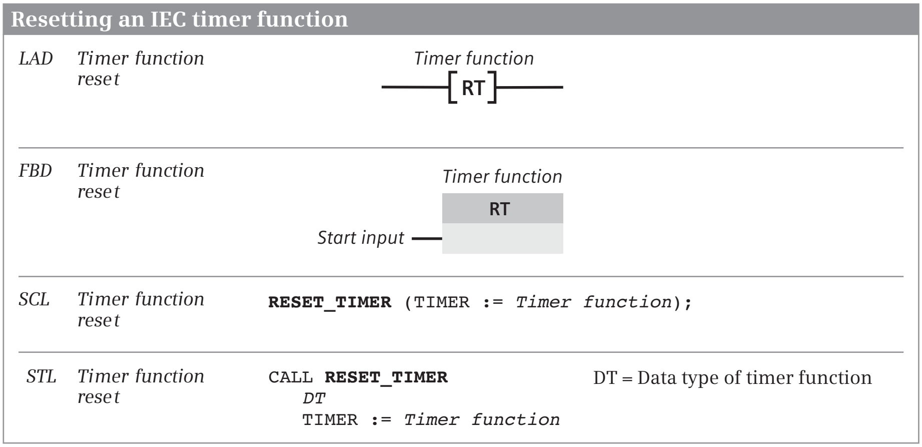
FBD: The PT box is only executed if the signal state at the start input is “1”. Signal state “0” has no effect.

SCL: The PRESET_TIMER function is always executed during processing.

STL: PRESET_TIMER is called together with the CALL operation. After the call comes the data type of the duration, which is specified at the parameter PT, and the data type of the timer function, which is specified at the parameter TIMER.

12.5.7 Resetting an IEC timer function

The program elements shown in Fig. 12.36 are available for resetting an IEC timer function. For a single instance, the data block is specified as the timer function. For a local instance, the instance name is specified.



```
RESET_TIMER (TIMER := Timer function);
```

```
CALL RESET_TIMER
   DT
   TIMER := Timer function
```

Fig. 12.36 Resetting a timer function

LAD: The RT coil is only executed if the signal state before the coil is “1”. Signal state “0” has no effect.

FBD: The RT box is only executed if the signal state at the start input is “1”. Signal state “0” has no effect.

SCL: The RESET_TIMER function is always executed during processing.

STL: RESET_TIMER is called together with the CALL operation. After the call comes the data type of the timer function, which is specified at the parameter TIMER.

12.6 SIMATIC counter functions

12.6.1 Overview

You can use the SIMATIC counter functions to execute counting tasks directly using the CPU. The counter functions can count up and down; the numerical range extends over three decades (000 to 999).

The counting frequency of these counter functions depends on the execution time of your program. In order to count, the CPU must recognize a change in the signal state of the input pulse, i.e. an input pulse (or a pause) must be present for at least one program cycle. The longer the program execution time, the lower the counting frequency.

The following responses are available for a SIMATIC counter function:

- Up counter
  Each pulse at the counter input increments the count value by one unit.
- Down counter
  Each pulse at the counter input decrements the count value by one unit.
- Up/down counter
  A pulse at the up counter increments the count value by one unit; a pulse at the down counter decrements the count value by one unit.

A data record which is present in the system data is permanently assigned to each SIMATIC counter function; this limits the number of SIMATIC counter functions. SIMATIC counter functions are global tags; the symbols are declared in the PLC tag table.

The SIMATIC counter functions run in STARTUP and RUN modes.

SIMATIC counters as overall function

The overall function is represented in the programming languages LAD and FBD as a box (Fig. 12.37). The box of a counter function contains the related representation of all individual counter operations in the form of function inputs and outputs. The address of the counter function is named above the box in absolute or symbolic form. The counter response is quasi the heading in the box. Assignment of the first box input is mandatory, assignment of the other inputs and outputs is optional. With SCL, the complete function call corresponds to the overall function. With STL, the individual statements must be programmed in the indicated sequence.

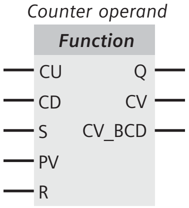
SIMATIC counters as single elements

In the representation as single elements, attention must be paid to the programming sequence so that the counter function responds as described later in this book: First program the count up and count down statement, then the set statement, followed by the reset statement, and finally scan the counter function. If the enabling statement is used with STL, it must be programmed prior to the counter statement (Fig. 12.38).

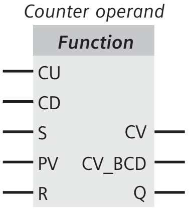
When programming a counter function, you need not use all statements available for the counter function. It is sufficient to use the statements required for the desired function. In the normal case these are the count up or count down function, setting of the counter function with specification of the count value, and the binary scanning of the counter status.

SIMATIC counter functions, representation as total function

LAD



FBD



Parameters for LAD and FBD:

Name	Declaration	Data type	Description
CU	INPUT	BOOL	Count up input
CD	INPUT	BOOL	Count down input
S	INPUT	BOOL	Set input
PV	INPUT	WORD	Specified count value
R	INPUT	BOOL	Reset input
Q	OUTPUT	BOOL	Counter status
CV	OUTPUT	WORD	Count value integer
CV_BCD	OUTPUT	WORD	Count value BCD

LAD: The representation shows an up/down counter S_CUD. The CD parameter is omitted with the up counter S_CU, and the CU parameter with the down counter S_CD.

FBD: The representation shows an up/down counter S_CUD. The CD parameter is omitted with the up counter S_CU, and the CU parameter with the down counter S_CD.

Function identifier:

	With LAD and FBD as		With SCL	With STL
	Box	Single element	with	with
Up counter	S_CU	CU	S_CU	CU
Down counter	S_CD	CD	S_CD	CD
Up/down counter	S_CUD	CU + CD	S_CUD	CU + CD

SCL

```
Count value BCD:= Function(
        C_NO     := Counter operand,
        CU       := Count up,
        CD       := Count down,
        S        := Set input,
        PV       := Count value,
        R        := Reset input,
        Q        => Counter status,
        CV       => Count value integer);
```

SCL: The representation shows an up/down counter S_CUD. The CD parameter is omitted with the up counter S_CU, and the CU parameter with the down counter S_CD.

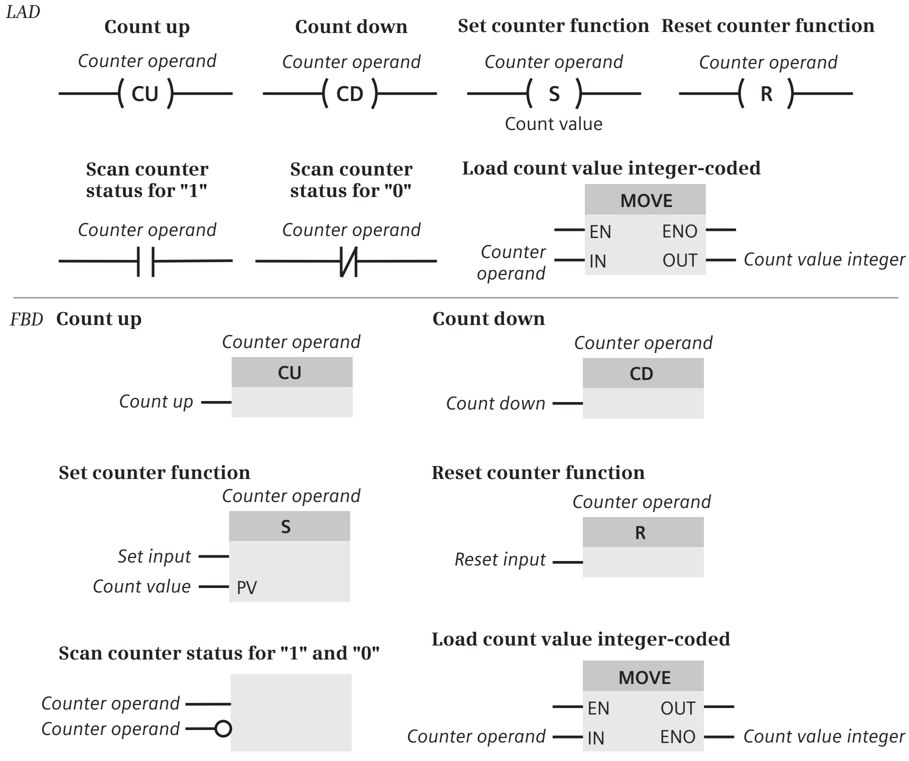
STL

```
A     Enable input;
FR    Counter operand;
A     Count up;
CU    Counter operand;
A     Count down;
CD    Counter operand;
A     Set input;
L     Count value;
S     Counter operand;
A     Reset input;
R     Counter operand;
L     Counter operand;
T     Count value integer;
LC    Counter operand;
T     Count value BCD;
A     Counter operand;
=     Counter status;
```

STL: The representation shows an up/down counter. With the up counter you omit the counting down, with the down counter you omit the counting up.

Fig. 12.37 SIMATIC counter functions as overall function

Representation of a SIMATIC counter function with single elements



SCL

```
Count value BCD := S_CUD(
        C_NO     := Counter operand,
        CU       := Count up,
        CD       := Count down,
        S        := Set input,
        PV       := Count value,
        R        := Reset input,
        Q        => Counter status,
        CV       => Count value integer);
```

Individual parameters of the counter function can be omitted. You only program the parameters that you require.
You must always specify the counter operand (C_NO).
Set input (S) and count value (PV) must always be handled in pairs.

STL

```
A     Enable input;
FR    Counter operand;
A     Count up;
CU    Counter operand;
A     Count down;
CD    Counter operand;
A     Set input;
L     Count value;
S     Counter operand;
A     Reset input;
R     Counter operand;
L     Counter operand;
T     Count value integer;
LC    Counter operand;
T     Count value BCD;
A     Counter operand;
=     Counter status;
```

The operations of the counter function can be programmed individually.
A count value is always required to set the counter function.

Fig. 12.38 SIMATIC counter functions, representation as single elements

12.6.2 Programming a counter function

Count up

A counter function is counted up, for example, using a binary tag. In the figures, this tag is named *Count up*.

Each positive edge when counting up increments the count value by one unit until the upper limit of 999 is reached. Any further positive edges for counting up then have no effect. Carrying forward does not take place.

Count down

A counter function is counted down, for example, using a binary tag. In the figures, this tag is named *Count down*.

Each positive edge when counting down decrements the count value by one unit until the lower limit of 0 is reached. Any further positive edges for counting down then have no effect. Counting with a negative count value does not take place.

Set counter function

A counter function is set to a default value, for example, using a binary tag. In the figures, this tag is named *Set input*.

With a positive edge at the set input, the default value is transferred to the counter.

Specification of count value

When setting, the counter function is loaded with a default value of data type WORD (BCD16 in the range from W#16#0000 to W#16#0999). In the figures, this tag is named *Count value*.

Fig. 12.39 shows the bit assignment of the count value.

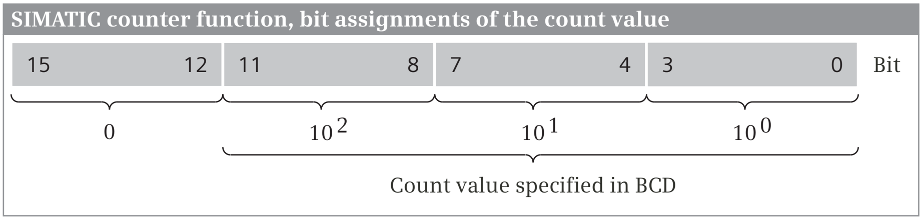

SIMATIC counter function, bit assignments of the count value
15
12
11
8
7
4
3
0
Bit
0
10^2
10^1
10^0
Count value specified in BCD

Fig. 12.39 Bit assignment of the count value of a SIMATIC counter

Reset counter function

A counter function is reset, for example, using a binary tag. In the figures, this tag is named *Reset input*.

A counter function is reset as long as the reset input has signal state "1". Resetting of the counter function sets the count value to zero and the counter status to "0".

Setting, counting up, and counting down of the counter function is not possible for as long as the reset is present.

Note with STL: Resetting a counter function does not reset the internal edge trigger flags for setting, counting up, and counting down. To set, count up or count down again, the corresponding operation must first be processed with RLO "0" before the counter function detects a signal edge. You can also use enabling of the counter function for this.

Scan counter status

The counter status indicates with signal state "1" that the current count value is greater than zero. With a count value of zero, the counter status has signal state "0". For example, the counter status can be assigned to a binary tag. In the figures, this tag is named *Counter status*.

Scanning the current count value BCD-coded

The count value is the current counter value at the time of scanning. It can be assigned, for example, to a tag with data type WORD. In the figures, this tag is named *Count value BCD*. The range of values is from W#16#0000 to W#16#0999.

Scanning the current count value integer-coded

The count value is the current counter value at the time of scanning. It can be assigned, for example, to a tag with data type INT. In the figures, this tag is named *Count value integer*. The range of values is from 0 to +999.

12.6.3 Principle of operation of a counter function

Fig. 12.40 shows the principle of operation of the SIMATIC counter function.

❶ The counter is at a count value of zero. The counter status has signal state "0".

❷ A positive edge at the count up input increments the count value by one unit to 1.

❸ A positive edge at the count up input increments the count value by one unit to 2.

❹ The positive edge at the set input sets the counter to the specified count value of 4.

❺ A positive edge at the count up input increments the count value by one unit to 5.

❻ One positive edge each at the count up and count down inputs result in the end that the count value does not change.

❼ The positive edge at the count down input decrements the count value by one unit to 4.

❽ The positive edge at the count down input decrements the count value by one unit to 3.

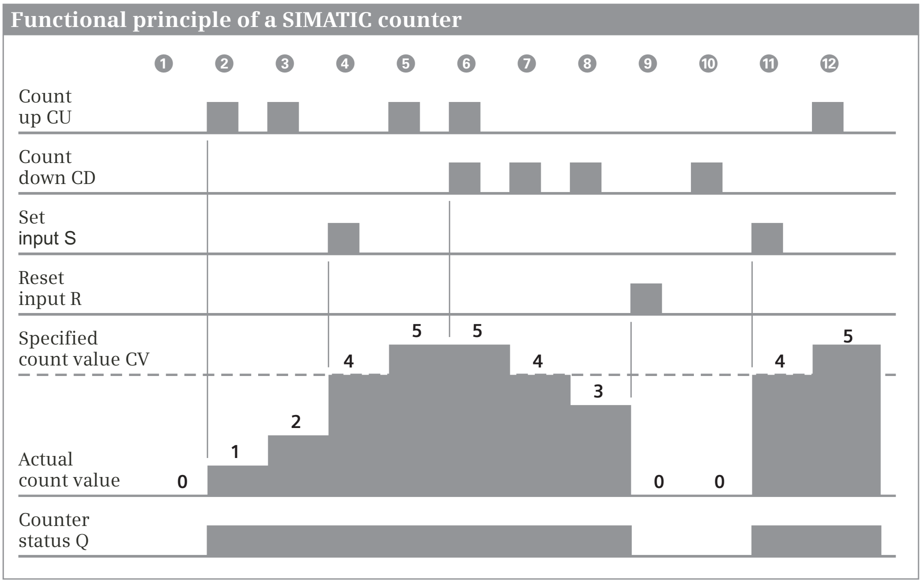


Fig. 12.40 Principle of operation of a SIMATIC counter function

⑨ Signal state “1” at the reset input resets the counter function. The count value is set to 0 and the counter status has signal state “0”.

⑩ Counting down with a count value of 0 has no effect.

⑪ With a positive edge at the set input, the count value is set to 4. The counter status has signal state “1”.

⑫ The positive edge at the count up input increments the count value by one unit to 5.

The sequence of counter statements upon which the example is based can be obtained from Fig. 12.38 on Page 548.

12.6.4 Enabling a counter function with STL

As a result of enabling, setting as well as counting up and down are executed even without a positive signal edge at the corresponding inputs. This is only possible if the corresponding operation continues to be processed with RLO “1”. Enabling is only present in the programming language STL; it is not required for setting, resetting or counting, i.e. for normal execution.

Enabling is triggered by a positive edge at the enabling operation, for example by a binary tag. In the figure, this tag is named *Enable input*.

Enabling resets the internal edge trigger flags for setting and counting. If the result of logic operation is “1” with the next processing at the set, count up or count down input, the corresponding function is executed again.

Note: Enabling has a quasi-simultaneous effect on setting, counting up, and counting down! Attention must therefore be paid to the sequence of set and count operations.

The following example explains the principle of operation of enabling at the inputs of the counter function (Fig. 12.41):

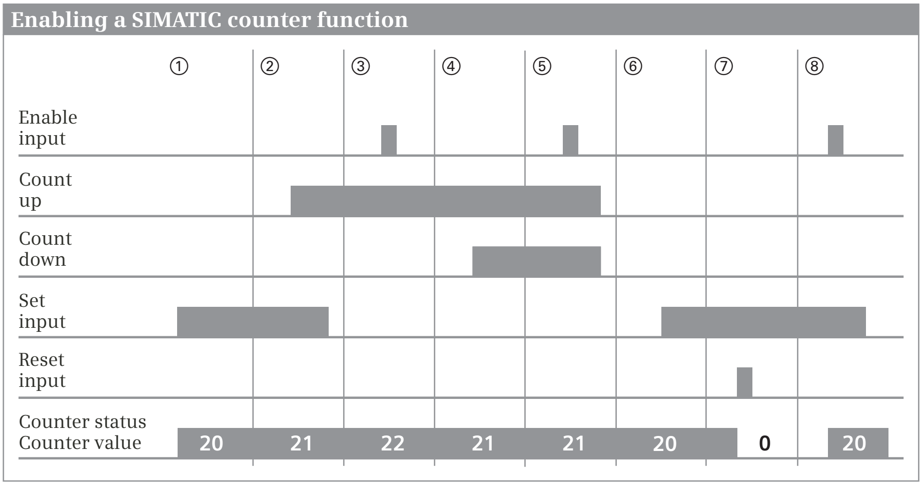


Fig. 12.41 Enabling a SIMATIC counter function

① The positive edge at the set input sets the counter to the start value 20.

② A positive edge at the count up input increments the count value by one unit.

③ Since the signal state at the count up input is “1”, the count value is incremented by one unit when enabled.

④ The positive edge at the count down input decrements the count value by one unit.

⑤ Counting up and down are executed as a result of enabling since signal state “1” is present at both inputs.

⑥ The positive edge at the set input sets the counter function to the start value 20.

⑦ Signal state “1” at the reset input resets the counter function. The scan of the counter function for signal state “1” delivers the result of scan “0”.

⑧ Since signal state “1” is still present at the set input, enabling results in the counter function being set to 20 again. The scan for signal state “1” now delivers the result of scan “1”.

The sequence of counter statements upon which the example is based can be obtained from Fig. 12.38 on Page 548.

12.7 IEC counter functions

12.7.1 Introduction

You can use the counter functions to execute counting tasks directly using 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 your 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

An IEC 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*. In this case the data of the counter function is saved as 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, LINT, USINT, UINT, UDINT, and ULINT. The data structure of a counter function is dependent on this setting. The setup of the data structure is shown in Section 4.8.4 “Parameter types for IEC counter functions” on page 132.

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*.

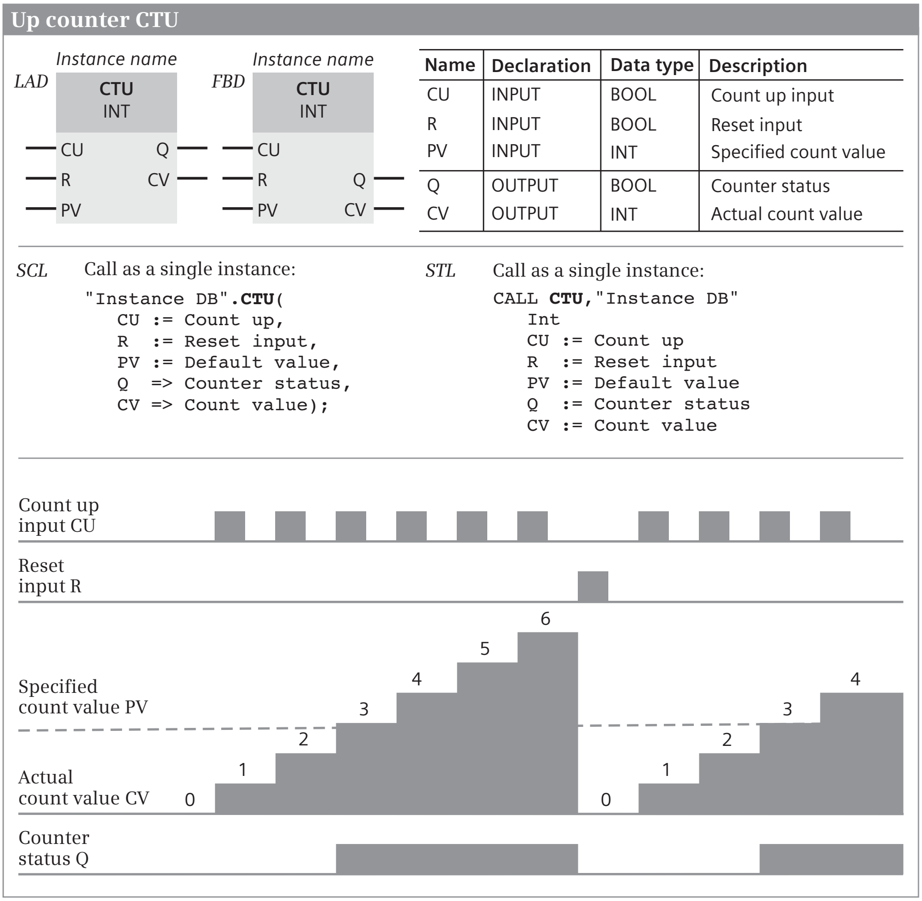
The counter functions run in STARTUP and RUN modes.

12.7.2 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. 12.42).

The count value is reset to zero if the reset input R has signal state “1”. A positive edge at the CU input has no effect for as long as the R input has signal state “1”.

The Q output has signal state “1” if the current count value is greater than or equal to the specified count value (CV ≥ PV).



Name	Declaration	Data type	Description
CU	INPUT	BOOL	Count up input
R	INPUT	BOOL	Reset input
PV	INPUT	INT	Specified count value
Q	OUTPUT	BOOL	Counter status
CV	OUTPUT	INT	Actual count value

SCL Call as a single instance:

```
"Instance DB".CTU(
   CU := Count up,
   R  := Reset input,
   PV := Default value,
   Q  => Counter status,
   CV => Count value);
```

STL Call as a single instance:

```
CALL CTU,"Instance DB"
   Int
   CU := Count up
   R  := Reset input
   PV := Default value
   Q  := Counter status
   CV := Count value
```

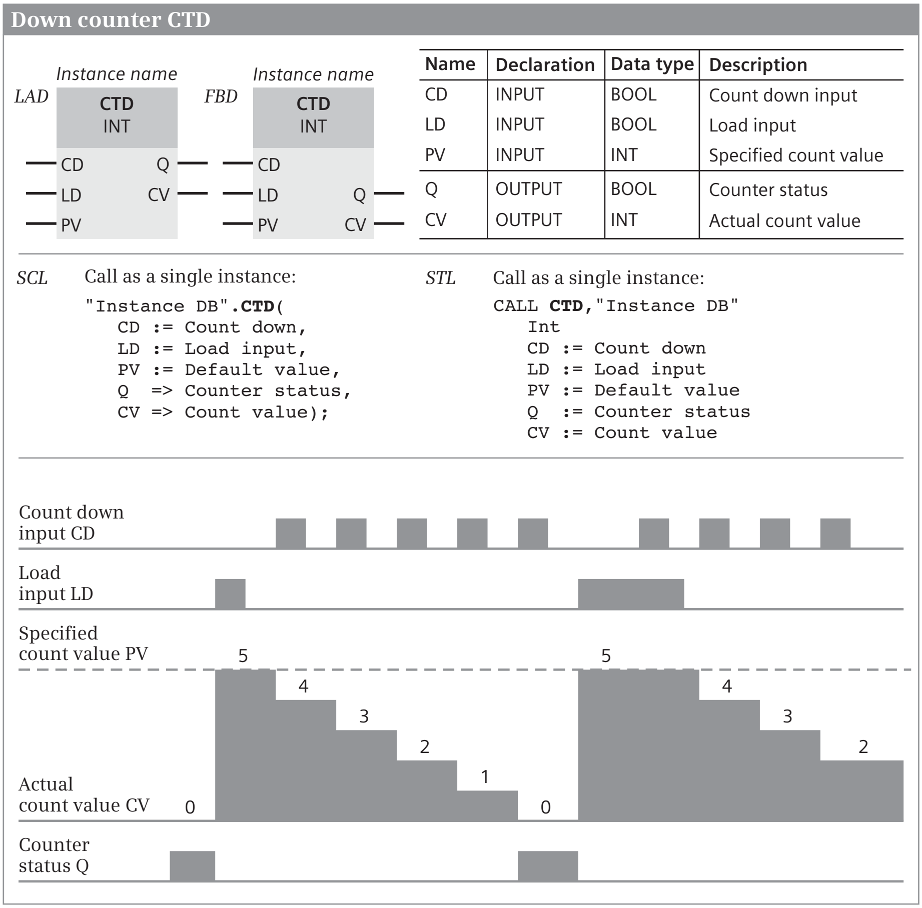
Fig. 12.42 Up counter CTU, representation and function

12.7.3 Down counter CTD

If the signal state at the count down 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. 12.43).

The count value CV is set to the specified count value PV if the LD input has signal state “1”. A positive edge at the CD input has no effect for as long as the LD input has signal state “1”.

The Q output has signal state “1” if the current count value is less than or equal to zero (CV ≤ 0).



Name	Declaration	Data type	Description
CD	INPUT	BOOL	Count down input
LD	INPUT	BOOL	Load input
PV	INPUT	INT	Specified count value
Q	OUTPUT	BOOL	Counter status
CV	OUTPUT	INT	Actual count value

SCL Call as a single instance:

```
"Instance DB".CTD(
   CD := Count down,
   LD := Load input,
   PV := Default value,
   Q  => Counter status,
   CV => Count value);
```

STL Call as a single instance:

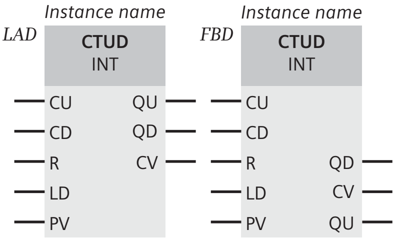
```
CALL CTD,"Instance DB"
   Int
   CD := Count down
   LD := Load input
   PV := Default value
   Q  := Counter status
   CV := Count value
```

Fig. 12.43 Down counter CTD, representation and function

12.7.4 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 (Fig. 12.44).

Up/down counter CTUD



Name	Declaration	Data type	Description
CU	INPUT	BOOL	Count up input
CD	INPUT	BOOL	Count down input
R	INPUT	BOOL	Reset input
LD	INPUT	BOOL	Load input
PV	INPUT	INT	Specified count value
QU	OUTPUT	BOOL	Count up status
QD	OUTPUT	BOOL	Count down status
CV	OUTPUT	INT	Actual count value

SCL Call as a single instance:

```
"Instance DB".CTUD(
   CU := Count up,
   CD := Count down,
   R  := Reset input,
   LD := Load input,
   PV := Default value,
   QU => Status_up,
   QD => Status_down,
   CV => Count value);
```

STL Call as a single instance:

```
CALL CTUD,"Instance DB"
   Int
   CU := Count up
   CD := Count down
   R  := Reset input
   LD := Load input
   PV := Default value
   QU := Status_up
   QD := Status_down
   CV := Count value
```

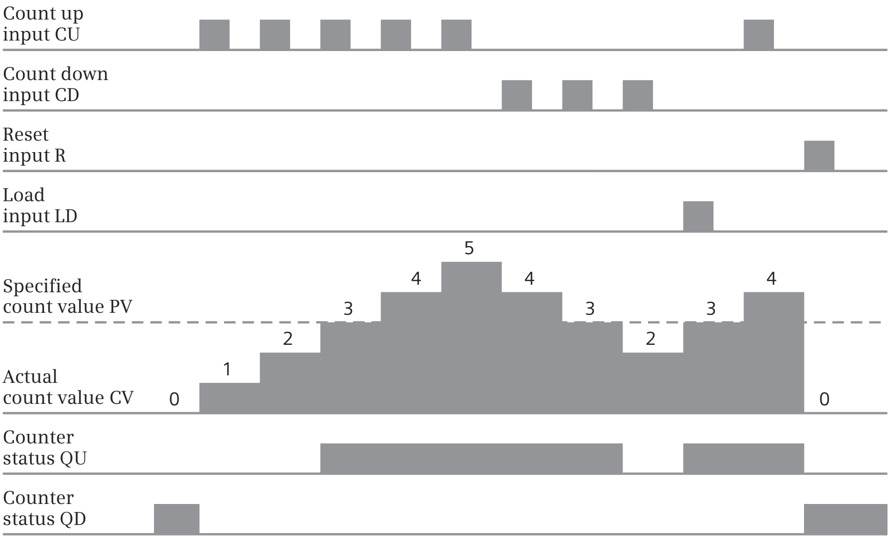


Fig. 12.44 Up/down counter CTUD, representation and function

If the current count 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 current count 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 current count value CV is set to the specified count value PV if the LD input has signal state “1”. Positive signal edges at the CU and CD counter inputs have no effect for as long as the LD input has signal state “1”.

The count value is reset to zero if the reset input R has signal state “1”. Positive signal edges at the CU and CD counter inputs and signal state “1” at the LD input have no effect for as long as the R input has signal state “1”.

The QU output has signal state “1” if the current count value is greater than or equal to the specified count value (CV ≥ PV).

The QD output has signal state “1” if the current count value is less than or equal to zero (CV ≤ 0).

13 Digital functions

13.1 General information

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 digital functions are implemented internally – not visible to you as the user – either by means of simple statement sequences or by calling a system or standard block. Therefore you can find the digital functions in the statements catalog under *Basic instructions* and *Extended instructions*.

The Chapters 7 “Ladder logic LAD” on page 287, 8 “Function block diagram FBD” on page 323, 9 “Structured Control Language SCL” on page 359, and 10 “Statement list STL” on page 395 describe how you can program the functions using the individual programming languages and what special features exist.

A CPU 1500 provides the following digital functions:

- 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 link two tags with fixed-point and floating-point data types in accordance with the basic arithmetic operations.
- The arithmetic functions for time values link two tags with a duration data type or time data type, for example the calculation of the difference of two times of day.
- The math functions convert the value of a tag with a floating-point data type 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), arithmetic, logic and comparison expressions in the case of SCL, and operations for linking the contents of accumulators in the case of STL.

13.2 Transfer functions

The following are available for transfer of tag contents between memory areas:

- “Simple” statements for copying a digital tag to another tag – with the MOVE and S_MOVE box in the case of LAD and FBD, with the value assignment function in the case of SCL, and with load and transfer functions in the case of STL.
- Transfer from and to tags with PEEK and POKE (SCL),
- system blocks for the transfer or filling of a memory area in the work memory (MOVE_BLK_VARIANT, MOVE_BLK, UMOVE_BLK, FILL_BLK, UFILL_BLK, BLKMOV, UBLKMOV, FILL) and
- swapping bytes with SWAP (LAD, FBD) or CAW and CAD (STL).

The FieldRead and FieldWrite functions (not described in the book) emulate the indirect addressing of field components for LAD and FBD. Indirect addressing with a variable index is more user-friendly and is described in Chapter 4.3.2 “Indirect addressing of ARRAY components” on page 100.

13.2.1 General information on the “simple” transfer function

The transfer function executed using “simple” statements copies the content of a digital tag to another tag or transfers a fixed value to a digital tag.

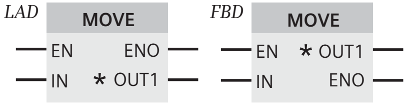
As a result of the different language elements, the transfer function is represented differently in the various programming languages: by the MOVE box in the case of LAD and FBD, by the value assignment function in the case of SCL, and by load and transfer functions in the case of STL.

13.2.2 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. 13.1). 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 outputs. A tag with elementary data type can also be a component of a structured data type.

A tag at the IN parameter can have all of the data types, except for STRING. The S_MOVE function is available for the transfer of STRING tags. The data type of the tags at the parameters OUT1, OUT2, etc. must be compatible with the data type of the tags at the parameter IN and is influenced by the block attribute *IEC check* (see Chapter 4.5.2 “Implicit data type conversion” on page 108).

MOVE box



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	*)	Source tag
OUT1	OUTPUT	*)	First destination tag

*) See text

Function:
The value present at the IN parameter is transferred to the OUT1 parameter. If the tag at the IN parameter has an elementary data type, the number of output parameters can be increased.

Fig. 13.1 Representation and function of the MOVE box with LAD and FBD

The MOVE box can also transfer a tag with structured data type, with hardware data type, with PLC data type, with system data type, or entire data blocks that are derived from a data type (type data blocks). In these cases, the data types at IN and OUT1 must coincide. An extension of the box outputs (with OUT2, OUT3, etc.) is then not possible.

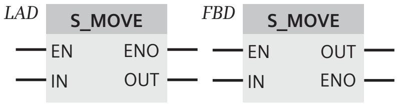
You can use EN to control execution of the MOVE box depending on the result of logic operation. If EN = “1” or not connected, the transfer function is executed and ENO has signal state “1”. If EN = “0”, ENO is also = “0”. The MOVE box does not report any errors.

13.2.3 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. 13.2). 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.

S_MOVE box



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	STRING	Source tag
OUT	OUTPUT	STRING	Destination tag

Function:
The value present at the IN parameter is transferred to the OUT parameter.

Fig. 13.2 S_MOVE box, representation and function

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".

13.2.4 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 compatible and depend on the block attribute *IEC check* (see Chapter 4.5.2 "Implicit data type conversion" on page 108).

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 LDT and 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
```

Every LDT tag can be assigned another LDT tag or a LDT constant.

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. With multi-dimensional arrays, you can handle the partial arrays like correspondingly dimensioned tags.

Example: #Array1 : ARRAY [1..8,1..16] OF INT represents a two-dimensional array; you can now address the complete array using *#Array1*, a partial array using *#Array1[#i]* (corresponds to the lines of the matrix), and an array component using *#Array1[#i,#k]*. The partial array *#Array1[#i]* can be assigned to a correspondingly dimensioned array, e.g. #Array2 := #Array1[i], where i = 1 to 8, and #Array2 : ARRAY [1..16] OF INT.

`Array1   ARRAY [1..8, 1..16] OF INT`
`Array2   ARRAY [1..16] OF INT`
`var_int  INT`
`i        INT`
`k        INT`
	Declaration

`#var_int := #Array1[#i,#k];`	Assignment of a component
`#Array2  := #Array1[#i];`	Assignment of a partial array

13.2.5 Loading and transferring with STL

The loading and transferring is applied for tags that are up to 32 bits wide. The transferring of tags with 64 bits is described in Chapter 10.5.1 “Transfer functions in the statement list” on page 415.

There are two statements with STL for the transfer of tag values that identify the transfer direction: The load statement is used to load a tag value from a memory area into accumulator 1. The transfer statement is used to transfer the value of accumulator 1 to a tag in a memory area:

```
L   #Input_tag          //Load into accumulator 1

T   #Output_tag         //Transfer from accumulator 1
```

The data types of the input and output tags are unimportant. The load statement can contain a constant or a digital tag with a width of up to 32 bits. The input tag is loaded right-justified into accumulator 1 and vacant bit positions are set to zero (Fig. 13.3).

The output tag for the transfer statement can be 8, 16, or 32 bits wide. The tag value is obtained right-justified from accumulator 1. The contents of accumulator 1 are not changed by this.

The load and transfer statements are executed independent of the result of logic operation and the status bits. Neither the result of logic operation nor the status bits are influenced.

Influencing of accumulator 2 when loading

The load function additionally changes the contents of accumulator 2. While the value of the input tag is being loaded into accumulator 1, accumulator 2 is simultaneously assigned the old value of accumulator 1. The load function transfers the complete contents – independent of the size of the input tag – from accumulator 1 to accumulator 2. The previous contents of accumulator 2 are lost in the process.

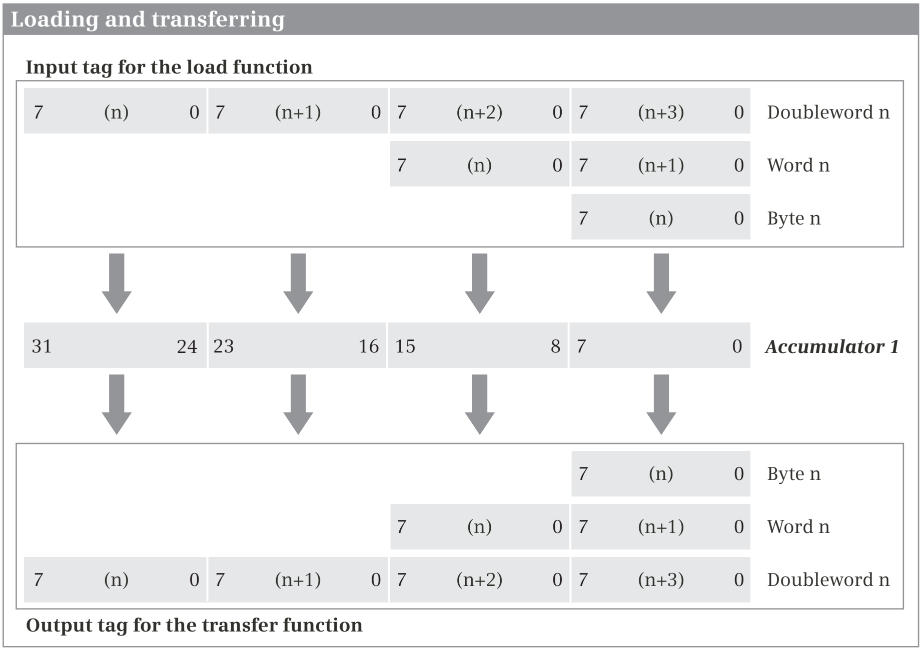


Fig. 13.3 Transfer of tags with a width of 8, 16, or 32 bits

Transferring only from accumulator 1

You can use the transfer function just for accumulator 1. If you wish to transfer a value from accumulator 2, use the corresponding accumulator functions (TAK or POP) to transfer the contents of accumulator 2 into accumulator 1 and then transfer the value (see also Chapter 10.7.3 “Accumulator functions” on page 448).

13.2.6 Copy data area (MOVE_BLK_VARIANT)

MOVE_BLK_VARIANT copies

- a tag, an operand area or a type data block, or
- transfers array components from one array (ARRAY tag) to another array or the same array.

Transfer of a tag or an operand area

The actual parameter at the input parameters SRC (source) and DEST (destination) can be an absolutely or symbolically addressed tag, an absolutely addressed operand area, or a type data block. The parameter COUNT is assigned the value one and the parameters SRC_INDEX and DEST_INDEX are assigned the value zero (Fig. 13.4).

The specified area is copied in the case of inputs and outputs independent of the actual assignment with input and output modules. The SIMATIC timer/counter operand areas cannot be accessed.

You can use the ANY pointer for specifying an absolutely addressed operand area; its structure is described in Chapter 4.9.4 “ANY pointer” on page 135. With an ANY pointer of type BOOL, e.g. for an array with binary components, the number (the repetition factor) must be divisible by 8. With an ANY pointer of type STRING, the number must be 1. How to use an ANY pointer during runtime is described in Chapter 4.3.5 “Indirect addressing with an ANY pointer” on page 103.

Transfer of ARRAY components

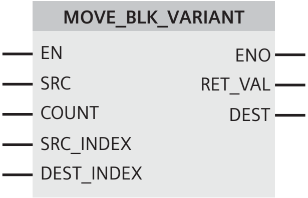
MOVE_BLK_VARIANT transfers one or more array components from one array (ARRAY data type) to another array or the same array. The source and destination components must be of the same data type.

Specify the first array components at the parameter SRC as the data source and the index of these array components at the parameter SRC_INDEX. The value of SRC_INDEX is relative to the lower array limit (the value zero corresponds to the lower array limit of the source array). This allows you to address components of an array which did not exist at the time of programming.

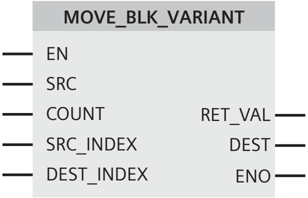
Specify the first array components at the parameter DEST as the data destination and the index of these array components at the parameter DEST_INDEX. The value of DEST_INDEX is relative to the lower array limit (the value zero corresponds to the lower array limit of the destination array). The parameter COUNT is assigned the number of components to be transferred.

MOVE_BLK_VARIANT

LAD



FBD



SCL

```
var_int := MOVE_BLK_VARIANT (
  SRC        := var_variant ,
  COUNT      := var_udint ,
  SRC_INDEX  := var_dint ,
  DEST_INDEX := var_dint ,
  DEST       => var_variant );
```

STL

```
CALL MOVE_BLK_VARIANT
  SRC        := var_variant
  COUNT      := var_udint
  SRC_INDEX  := var_dint
  DEST_INDEX := var_dint
  RET_VAL    := var_int
  DEST       := var_variant
```

Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
SRC	INPUT	VARIANT	Tag or ARRAY component (source)
COUNT	INPUT	UDINT	One or number of ARRAY components
SRC_INDEX	INPUT	DINT	Zero or relative index of the ARRAY component (source)
DEST_INDEX	INPUT	DINT	Zero or relative index of the ARRAY component (destination)
DEST	OUTPUT	VARIANT	Tag or ARRAY component (destination)
RET_VAL	RETURN	INT	Error information

Transfer of a tag or an operand area:

MOVE_BLK_VARIANT transfers an operand, a tag, an operand area, or a type data block from the parameter SRC (source) to the parameter DEST (destination). COUNT is assigned 1, SRC_INDEX and DEST_INDEX are assigned 0. RET_VAL contains the error information.

Transfer of ARRAY components:

MOVE_BLK_VARIANT transfers components between arrays of the data type ARRAY. The first array component of the source array is present at the SRC parameter; the first component of the destination array is present at the DEST parameter. COUNT specifies the number of components to be transferred.

The index of the source component is present at SRC_INDEX and the index of the destination component is present at DEST_INDEX. Both specifications are relative to the lower array limit (0 = lower array limit).

RET_VAL contains the error information.

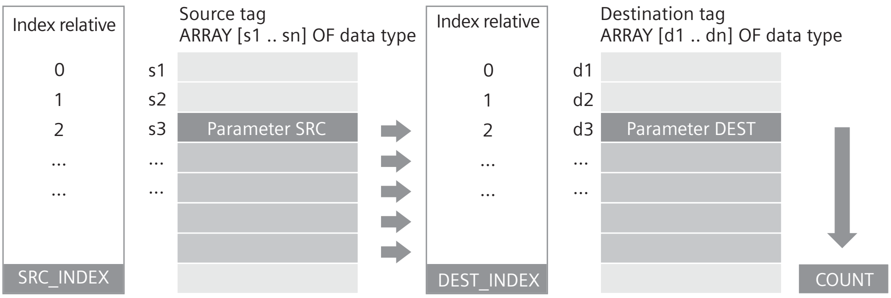


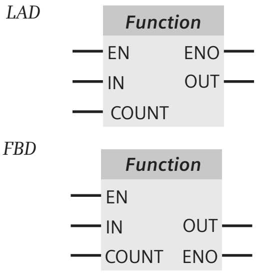
Fig. 13.4 Transfer function MOVE_BLK_VARIANT

The programmed source and destination areas must be located in the source and destination tags in their entirety. In the event of an error, e.g. when the range limits are exceeded, an error notification is displayed at the parameter RET_VAL.

13.2.7 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

MOVE_BLK, UMOVE_BLK, FILL_BLK, UFILL_BLK



SCL

```
Function (IN    := ... ,
          COUNT := ... ,
          OUT   => ... );
```

STL

```
CALL Function
     DT1    DT2
     IN     := ...
     COUNT  := ...
     OUT    := ...
```

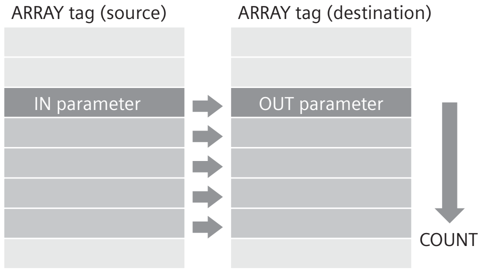
DT1: Data type for COUNT
DT2: Data type for IN and OUT

Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	A component of an ARRAY tag	Source tag (MOVE_BLK, UMOVE_BLK)
IN	INPUT	Elementary data type	Source tag (FILL_BLK, UFILL_BLK)
COUNT	INPUT	USINT, UINT, UDINT, ULINT	Quantity
OUT	OUTPUT	A component of an ARRAY tag	Destination tag

Function:

MOVE_BLK Transfer data area
UMOVE_BLK Fill data area uninterruptible

From the ARRAY tags at the IN parameter, the number of components specified at the COUNT parameter are transferred to the ARRAY tag at the OUT parameter. The components specified at the parameters IN and OUT are the respective start components for source and destination.



Function:

FILL_BLK Fill data area
UFILL_BLK Fill data area uninterruptible

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 destination.

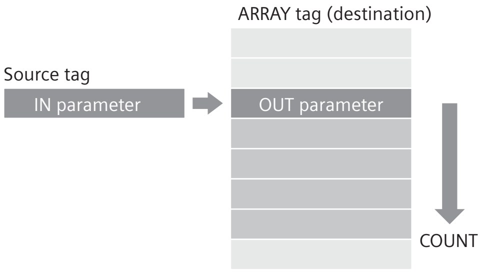


Fig. 13.5 Transfer functions MOVE_BLK, UMOVE_BLK, FILL_BLK and UFILL_BLK

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. 13.5).

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. A maximum of 16 KB can be transferred using UMOVE_BLK.

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.

13.2.8 Fill data area (FILL, FILL_BLK, UFILL_BLK)

FILL_BLK and UFILL_BLK transfer the content of a tag or a constant value to sequential components of an ARRAY tag. The data source 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. 13.5).

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. UFILL-BLK copies a maximum of 16 KB.

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.

13.2.9 Copy and fill data area (BLKMOV, UBLKMOV, FILL)

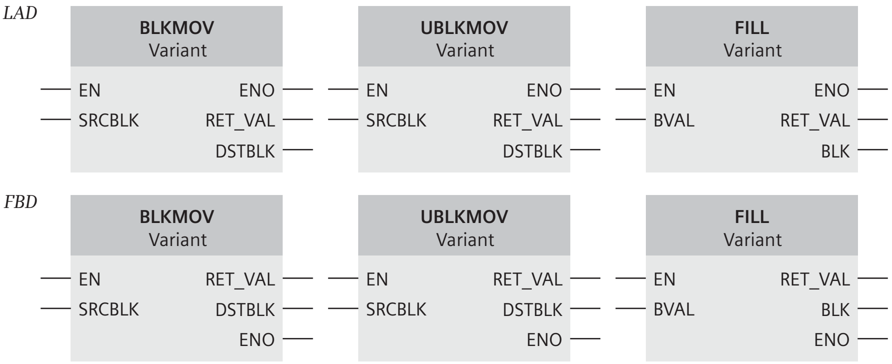
BLKMOV and UBLKMOV transfer the contents of one memory area to another memory area. The source area is defined by parameter SRCBLK and the destination area by parameter DSTBLK. BLKMOV copies the values so that the process can be interrupted by a higher-priority program (advantage: quick response time to alarms). UBLKMOV copies without interruption (advantage: transfer of consistent data areas). During transfer by UBLKMOV, alarm events that occur are stored and processed after the transfer ends.

FILL transfers the contents of a tag to a memory area multiple times. The data source is defined by parameter BVAL and the destination area by parameter BLK (Fig. 13.6).

Block parameters with data type VARIANT

The system blocks BLKMOV, UBLKMOV, and FILL each have two parameters with data type VARIANT. You can create an operand, a tag, or an absolutely addressed area from the operand areas Inputs, Outputs, Bit memories, and Data blocks at these parameters. The data block that is used must be present in the work memory; the attribute *Optimized block access* is deactivated.

BLKMOV, UBLKMOV, FILL



SCL

```
<Error info> :=        <Error info> :=        <Error info> :=
BLKMOV (               UBLKMOV (              FILL (
  SRCBLK := ... ,        SRCBLK := ... ,        BVAL := ... ,
  DSTBLK => ... );       DSTBLK => ... );       BLK  => ... );
```

STL

```
CALL BLKMOV            CALL UBLKMOV           CALL FILL
   Variant                Variant                Variant
   SRCBLK  := ...         SRCBLK  := ...         BVAL     := ...
   RET_VAL := ...         RET_VAL := ...         RET_VAL  := ...
   DSTBLK  := ...         DSTBLK  := ...         BLK      := ...
```

Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
SRCBLK	INPUT	VARIANT	Source tag (BLKMOV, UBLKMOV)
BVAL	INPUT	VARIANT	Source tag (FILL)
RET_VAL	RETURN	INT	Error information
DSTBLK	OUTPUT	VARIANT	Destination tag (BLKMOV, UBLKMOV)
BLK	OUTPUT	VARIANT	Destination tag (FILL)

Fig. 13.6 Transfer functions BLKMOV, UBLKMOV, and FILL

The specified area is copied in the case of inputs and outputs independent of the actual assignment with input and output modules. The SIMATIC timer/counter operand areas cannot be accessed.

You can use the ANY pointer for specifying an absolutely addressed operand area; its structure is described in Chapter 4.9 “Pointer” on page 134. With an ANY pointer of type BOOL, e.g. for an array with binary components, the number (the repetition factor) must be divisible by 8. With an ANY pointer of type STRING, the number must be 1. How to use an ANY pointer during runtime is described in Chapter 4.3.5 “Indirect addressing with an ANY pointer” on page 103.

Transfer in general

The transfer is carried out in the direction of increasing addresses (incrementing). The source and destination areas must not overlap. No data transfer takes place if the limits of operand areas are violated and an error message is output.

BLKMOV, UBLKMOV: If the source and destination areas are of different length, transfer is only performed up to the length of the smaller area.

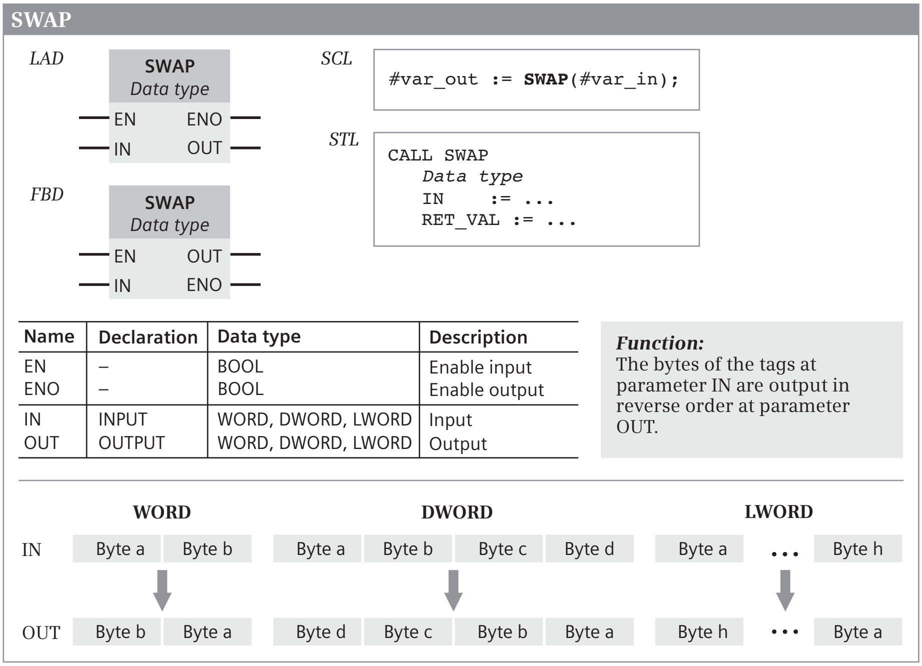
FILL: The destination area is always written completely, even if the source area is larger than the destination area or if the length of the destination area is not an integral multiple of the length of the source area.

Transfer of a STRING tag

If a STRING tag is only present at the SRCBLK/BVAL parameter, the current characters of the tag are copied. The two bytes with the length data are not written into the destination area.

If tags with data type STRING are present at both the SRCBLK/BVAL and DSTBLK/BLK parameters, the two length bytes are also transferred to the destination tag.

If a STRING tag is only present at the parameter DSTBLK/BLK, then each byte of the source area is transferred to a current character. The current length is updated.



```
#var_out := SWAP(#var_in);
```

```
CALL SWAP
   Data type
   IN     := ...
   RET_VAL := ...
```

Name	Declaration	Data type	Description
EN ENO	– –	BOOL BOOL	Enable input Enable output
IN OUT	INPUT OUTPUT	WORD, DWORD, LWORD WORD, DWORD, LWORD	Input Output

Function:
The bytes of the tags at parameter IN are output in reverse order at parameter OUT.

Fig. 13.7 Swap bytes with SWAP

13.2.10 Swap bytes (SWAP)

SWAP reads the tag at the IN parameter, exchanges the sequence of the bytes, and makes the result available at the OUT parameter (Fig. 13.7). WORD, DWORD, and LWORD can be set as data types.

For word and doubleword-wide operands, the statements CAW and CAD are available for STL (see section "Swap bytes in accumulator 1" on page 450).

13.3 Comparison functions

A comparison function compares the values of two tags to one another or it checks whether a tag value is within or outside of a value range. A comparison function provides a binary comparison result. Depending on the data type, the "simple" comparison functions or special system blocks are available for a comparison.

The "simple" comparison functions are the comparison contact in the case of LAD, the comparison box in the case of FBD, the comparison expression in the case of SCL, and the comparison operation in the case of STL. For tags with "long" data types there are system blocks for STL in the *Long Functions* global library.

13.3.1 Execution of "simple" comparison function

The "simple" comparison function compares the contents of two input tags and sets the binary comparison result to "1" (TRUE) if the comparison is fulfilled or to "0" (FALSE) if the comparison is not fulfilled. Fig. 13.8 shows the available comparison functions and the permissible data types in the various programming languages.

The data types of the tag to be compared must be compatible. The block attribute *IEC check* controls the degree of compatibility (see Chapter 4.5.2 "Implicit data type conversion" on page 108).

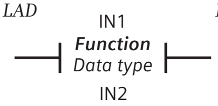
A prerequisite for a fulfilled comparison with floating-point numbers is that they are valid. If an invalid floating-point number is compared, the comparison is never fulfilled.

The comparison of character values CHAR or STRING is done within the scope of the ASCII coding. Two strings are the same if the relevant (occupied) characters are the same and the current length is the same. A string is considered as "greater than" if it is longer when the first characters are identical. The maximum lengths of the strings are not included in the comparison.

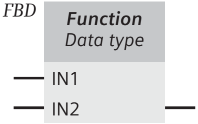
The comparison of time values TIME or DTL is done within the scope of the specified data type. A time (date, time of day) is regarded as smaller if the numerical value is smaller, i.e. if the time is older.

"Simple" comparison function

LAD



FBD



Name	Declaration	Data type	Description
IN1	INPUT	*Data type*	Input tag 1
IN2	INPUT	*Data type*	Input tag 2
–	OUTPUT	BOOL	Comparison result

The comparison function compares the contents of two tags IN1 and IN2 according to the following scheme: Result := IN1 <Comparison> IN2.

Function:		***Data types:***
== <>	equal to not equal to	Elementary data types (except BOOL), DT, DTL and STRING
> < >= <=	greater than less than greater than or equal to less than or equal to	Elementary data types (except bit-serial data types), DT, DTL and STRING

SCL

```
#Comparison_result := #var_IN1 Comparison function #var_IN2;
```

Comparison function:		***Data types:***
= <>	equal to not equal to	Elementary data types, DT, DTL and STRING
> < >= <=	greater than less than greater than or equal to less than or equal to	Elementary data types (except BOOL), DT, DTL and STRING

STL

```
L   #var_IN1
L   #var_IN2
Comparison operation
=   #Comparison_result
```

The contents of accumulator 2 (IN1) are compared with the contents of accumulator 1 (IN2) and the result of the comparison assigned to the RLO:
Result of comparison = IN1 *Comparison operation* IN2

The comparison operation does not change the contents of the accumulators.
The comparison operation sets the status bits.

Comparison operation:

==I,	==D,	==R	equal to
<>I,	<>D,	<>R	not equal to
>I,	>D,	> R	greater than
<I,	<D,	<R	less than
>=I,	>=D,	>=R	greater than or equal to
<=I,	<=D,	<=R	less than or equal to

The comparison operation interprets the contents of the accumulators in accordance with the data type during the operation:
I = INT, D = DINT, R = REAL.

Comparison of tags with "long" data types

```
CALL "EQ_LINT"
   IN1  := #var_IN1
   IN2  := #var_IN2
   OUT  := #var_result
```

Comparison function:		***Data types:***
EQ_... NE_...	equal to not equal to	LWORD, LINT, ULINT, LREAL
GT_... LT_... GE_... LE_...	greater than less than greater than or equal to less than or equal to	LINT, ULINT, LREAL

Fig. 13.8 Function and representation of “simple” comparison functions

Simple comparison function with LAD and FBD

For LAD and FBD, the “simple” comparison is done using the comparison contact or the comparison box. You select the comparison relationship and the data type from a drop-down list. If the data types are different, select the data type with the greater data width.

Simple comparison function with SCL

With SCL, the comparison is implemented by a comparison expression.

Simple comparison function with STL

With STL, the comparison is carried out according to the data type defined by the comparison operation. The user must make sure that the “correct” data type is present in the accumulators. Comparisons according to INT only compare the right word of the accumulators, comparisons according to DINT and REAL compare the complete contents.

A comparison function is executed independent of conditions and influences the status bits.

System blocks from the global *Long Functions* library are available for the comparison of tags with “long” data types.

13.3.2 Comparison function T_COMP

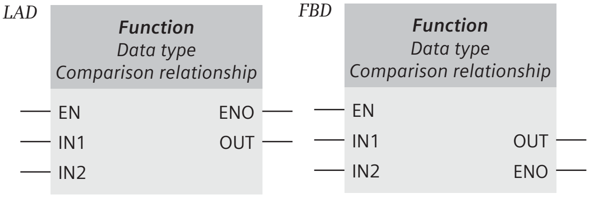
T_COMP compares two tags with a time data type (Fig. 13.9). For LAD and FBD, the comparison function is displayed with an EN/ENO box. For STL, it is a block call. To compare time data types for SCL tags, use the comparison expression (see Chapter 13.3.1 “Execution of “simple” comparison function” on page 570). The tags to be compared are present at parameters IN1 and IN2. The parameter OUT provides the comparison result: Signal state “1” (TRUE) if the comparison is fulfilled and signal state “0” (FALSE) if it is not fulfilled.

A time is considered as “greater than” if it is later, in other words closer to the present time or further in the future than the comparison value. T_COMP does not signal an error.

13.3.3 Comparison function S_COMP

S_COMP compares two tags with a string data type (Fig. 13.9). For LAD and FBD, the comparison function is displayed with an EN/ENO box. For STL, it is a block call. To compare tags with character data types for SCL, use the comparison expression (see Chapter 13.3.1 “Execution of “simple” comparison function” on page 570). The tags to be compared are present at parameters IN1 and IN2. The parameter OUT provides the comparison result: Signal state “1” (TRUE) if the comparison is fulfilled and signal state “0” (FALSE) if it is not fulfilled.

Comparison functions T_COMP and S_COMP



STL

```
CALL Function
   Data type  Function
   IN1 := ...
   IN2 := ...
   OUT := ...
```

Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN1	INPUT	*Data type*	Input tag
IN2	INPUT	*Data type*	Input tag
OUT	OUTPUT	BOOL	Comparison result

With SCL, the comparison of tags with time or string types is implemented with a comparison expression.

Function:	Comparison of two tags	***Data type:***
T_COMP	with time data types	DATE, TIME, LTIME, TOD, LTOD, DT, DTL, LDT
S_COMP	with string data types	STRING

Comparison relationship:

EQ	=	equal to	LT	>	greater than	LT	<	less than
NE	<>	not equal to	GE	>=	greater than or equal to	LE	<=	less than or equal to

Fig. 13.9 Comparison of tags with time and string data type

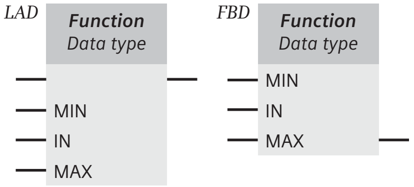
Starting from the left, the characters of the tags are compared by their ASCII code (for example, 'a' is greater than 'A'). The first character to be different decides the result of the comparison. Two strings are the same if the relevant (occupied) characters are the same and the current length is the same. A string is considered as “greater than” if it is longer when the first characters are identical. The maximum lengths of the strings are not included in the comparison. S_COMP does not signal an error.

13.3.4 Range comparison

The range comparison checks whether the value of a digital tag is within or outside of a value range marked by limit values (Fig. 13.10). The range comparison is available for LAD and FBD.

The comparison is fulfilled (comparison result = “1” or TRUE) if the input value for the function IN_RANGE is within the range or if the input value for OUT_RANGE is outside the range. The limits (MIN, MAX) and the input 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.

Range comparison



Name	Declaration	Data type	Description
MIN	INPUT	*Data type*	Lower limit
IN	INPUT	*Data type*	Input tag
MAX	INPUT	*Data type*	Upper limit
–	OUTPUT	BOOL	Comparison result

Function:

IN_RANGE	:= MIN ≤ IN ≤ MAX	within the range
OUT_RANGE	:= MIN > IN > MAX	outside of the range

Data type:
all fixed-point and floating-point data types

Fig. 13.10 Range comparison with IN_RANGE and OUT_RANGE

All fixed-point and floating-point numbers are permitted as data types of the input parameters. If the attribute *IEC check* is activated, the data types to be compared must be the same. If the attribute is deactivated, the data types are converted within the scope of the implicit data type conversion. You then select the data type in the box with the greatest data width. The lower limit and upper limit may also be assigned constants. If invalid floating-point numbers are specified, the comparison is not fulfilled.

13.4 Arithmetic functions

An arithmetic function adds, subtracts, multiplies and divides two numerical values. In addition to the basic arithmetic operations for numbers, there are also arithmetic functions for date and time.

13.4.1 Arithmetic functions for numerical values

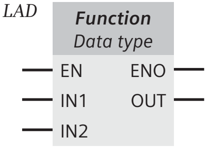
For LAD and FBD, the basic arithmetic operations are implemented by means of EN/ENO boxes. For SCL, they are implemented by means of arithmetic expressions and for STL by means of linking the accumulator contents or, for the "long" data types, by means of system blocks. Fig. 13.11 shows the general representation of an arithmetic function in the various programming languages.

A division (DIV) with fixed-point numbers provides the result in the form of the integer portion of the quotient IN1/IN2. An MOD division provides the remainder of the division as result. The remainder is the leftover part of the division; this is not the decimal places.

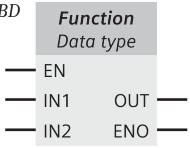
Fixed-point and floating-point numbers are permitted as data types for the basic arithmetic operations. A floating-point number can be in a range with full accuracy ("normalized" floating-point number) or in a range with limited accuracy ("denormalized" floating-point number), see also Chapter 4.6.6 "Floating-point data types

Arithmetic functions (basic arithmetic operations)

LAD



FBD



Name	Declaration	Data type	Description
EN	-	BOOL	Enable input
ENO	-	BOOL	Enable output
IN1	INPUT	*Data type*	Input tag 1
IN2	INPUT	*Data type*	Input tag 2
OUT	OUTPUT	*Data type*	Result

An arithmetic function links the contents of two tags IN1 and IN2 according to the following scheme: Result := IN1 <Link> IN2.
The ADD and the MUL box can be expanded with further input parameters.

Function:

ADD	Addition
SUB	Subtraction
MUL	Multiplication
DIV	Division
MOD	Division with remainder as result

Data types:
Fixed-point numbers
Floating-point numbers (not with MOD)

SCL

```
OUT := IN1 Function IN2;
```

Function:

+	Addition
-	Subtraction
*	Multiplication
/	Division
MOD	Division with remainder as result

Data types:
Fixed-point numbers
Floating-point numbers (not with MOD)

STL

```
L  IN1
L  IN2
Arithmetic operation
T  OUT
```

The contents of accumulator 2 (IN1) are linked with the contents of accumulator 1 (IN2) and the result is stored in accumulator 1.

Arithmetic operation:

+I,	+D,	+R	Addition
-I,	-D,	-R	Subtraction
*I,	*D,	*R	Multiplication
/I,	/D,	/R	Division
MOD			Division with remainder as result

The arithmetic operation interprets the contents of the accumulators in accordance with the data type during the operation:
I = INT, D = DINT, R = REAL.

With a calculation with data type INT, only the right words of the accumulators are influenced; the contents of the left words remain unchanged.
An arithmetic function influences the status bits.

Arithmetic functions with "long" data types

For the basic arithmetic operations with "long" data types, there are the system blocks in the *Long Functions* global library.

```
CALL "ADD_LINT"
   IN1 := #var_LINT1
   IN2 := #var_LINT2
   OUT := #var_result
```

Arithmetic function:		*Data types:*
ADD_...	Add	LINT, ULINT, LREAL
SUB_...	Subtract	
MUL_...	Multiply	
DIV_...	Divide	

Fig. 13.11 Description of arithmetic functions for numerical values

REAL and LREAL" on page 118). If the result of a calculation with a CPU 1500 falls in the range with limited accuracy, zero is output as result and a downward violation of the numerical range is signaled.

The arithmetic function reports an error if the permitted numerical range is left or if an invalid floating-point number is specified. In the event of an error, the ENO output is set to signal state "0" in the case of LAD and FBD, the ENO tag and ENO output are set to FALSE in the case of SCL, and the overflow bits OV and OS are set to "1" in the case of STL.

13.4.2 Arithmetic functions for date and time

The arithmetic functions for date and time link tags with time data types. The functions are displayed in LAD and FBD as EN/ENO boxes. They are displayed as functions with function value in SCL and as block calls in STL (Fig. 13.12).

If the result of the arithmetic function is not within the permissible range, the result is limited to the corresponding value. In the event of an error, the ENO output is set to "0" in the case of LAD and FBD, the ENO tag and ENO output are set to FALSE in the case of SCL, and the overflow bits OV and OS are set to "1" in the case of STL.

13.4.3 Decrementing and incrementing

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. 13.13).

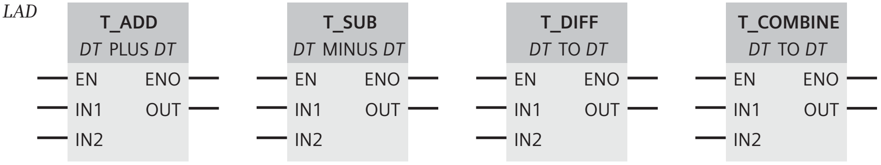
LAD, FBD: You can find the functions in the program elements catalog under *Basic instructions* > *Math functions*. When reaching the lowest and highest numerical value for the respective data type, the enable output ENO is set to signal state "0".

SCL: Decrementing and incrementing can be emulated with subtraction and addition.

STL: The decrement (DEC) and increment (INC) statements change the value present in accumulator 1. It is reduced (decremented) or increased (incremented) by the number of units specified in the parameter of this statement. The parameter can have values from 0 to 255. The change is only effective on the right byte in accumulator 1. The calculation is executed "modulo", i.e. if the value is reduced below 0 or increased above 255, counting commences again at 255 or 0. Decrementing and incrementing are preferably used for calculating addresses since – unlike an arithmetic function – they influence neither the contents of accumulator 2 nor the status bits.

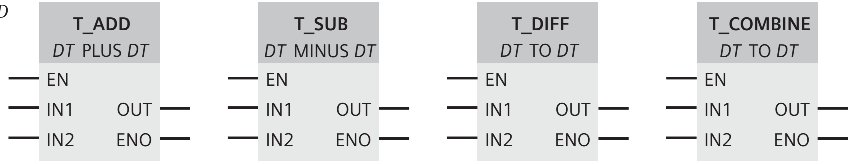
Arithmetic functions for date/time and duration

LAD



DT = data type

FBD



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN1	INPUT	*Data type (see below)*	Input tag with time data type
IN2	INPUT	*Data type (see below)*	Input tag with time data type
OUT	OUTPUT	*Data type (see below)*	Result

SCL

```
#var_OUT :=          #var_OUT :=
  T_ADD(               T_DIFF(
  IN1 := ... ,         IN1 := ...,
  IN2 := ... );        IN2 := ... );

#var_OUT :=          #var_OUT :=
  T_SUB(               T_COMBINE(
  IN1 := ... ,         IN1 := ... ,
  IN2 := ... );        IN2 := ... );
```

STL

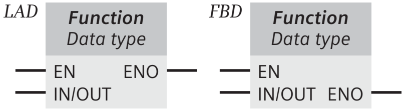
```
CALL T_ADD           CALL T_SUB
  DT PLUS DT           DT MINUS DT
  IN1 := ...           IN1 := ...
  IN2 := ...           IN2 := ...
  OUT := ...           OUT := ...

CALL T_DIFF          CALL T_COMBINE
  DT TO DT             DT TO DT
  IN1 := ...           IN1 := ...
  IN2 := ...           IN2 := ...
  OUT := ...           OUT := ...
```

DT = data type

Function:		***Data types:***
T_ADD	Addition	
	Duration + Duration = Duration Time + Duration = Time	Duration: TIME, LTIME Time: TOD, LTOD, DT, LDT, DTL
T_SUB	Subtraction	
	Duration - Duration = Duration Time - Duration = Time	
T_DIFF	Differentiation	
	Time - Time = Duration	
T_COMBINE	Combination	
	Date + Time of day = Time	Date: DATE, WORD, UINT, INT Time of day: TOD, LTOD Time: DT, DTL, LDT

Fig. 13.12 Arithmetic functions for tags with time data types

Decrementing, incrementing



Name	Declaration	Data type	Description
EN ENO	– –	BOOL BOOL	Enable input Enable output
IN/OUT	INOUT	*Data type*	Digital tag

Function:
DEC Decrement by —
INC Decrement by +1

Data type:
Fixed-point data type

SCL

```
#var_fixed-point := #var_fixed-point + 1;

#var_fixed-point := #var_fixed-point - 1;
```

For SCL, decrementing and incrementing can be emulated by a subtraction or an addition.

STL

```
L     #var_fixed-point
DEC n                //Decrement
T     #var_fixed-point

L     #var_fixed-point
INC n                //Increment
T     #var_fixed-point
```

DEC reduces the value in accumulator 1 by the parameter of the operation.
INC increases the value in accumulator 1 by the parameter of the operation.

Fig. 13.13 Decrementing and incrementing

13.5 Math functions

13.5.1 General function description

A math function converts the value of a tag present at the input in accordance with the function, and writes it into the tag present at the output. Fig. 13.14 shows the math functions in the various programming languages.

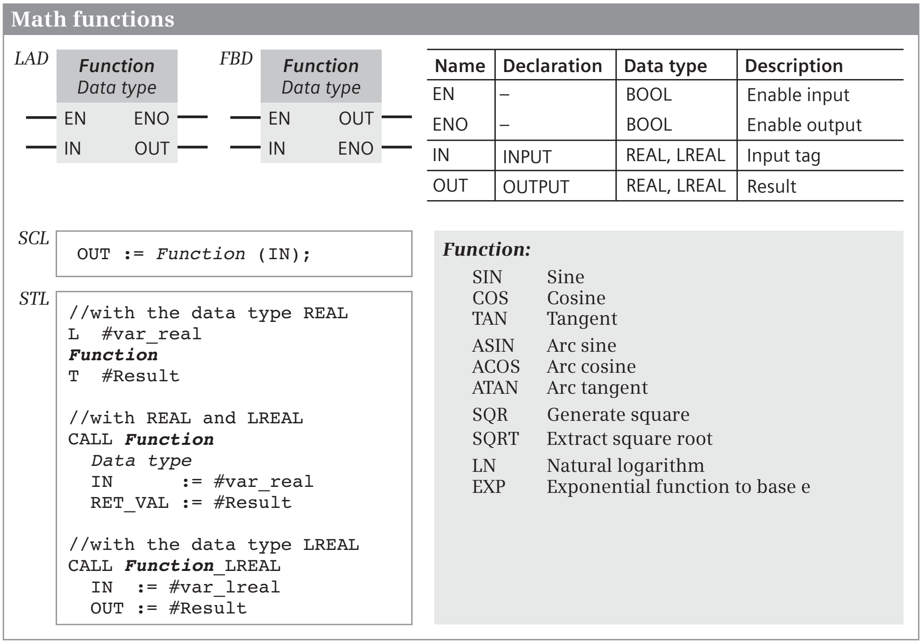
“Math functions” include the following:

- Sine (SIN), cosine (COS), tangent (TAN)
- Arcsine (ASIN), arccosine (ACOS), arctangent (ATAN)
- Generate square (SQR), extract square root (SQRT)
- Natural logarithm (LN) and exponential function to base e (EXP)

Further math functions are:

- Exponential function to any base (EXPT)
- Extract decimal points (FRAC)
- Generate absolute value (ABS) and generate two's complement (NEG)

The math functions process the numbers in the data formats REAL and LREAL. Tags with a fixed-point data type can also be created at the input and output parameters within the scope of implicit data type conversion (not for STL).



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	REAL, LREAL	Input tag
OUT	OUTPUT	REAL, LREAL	Result

```
OUT := Function (IN);
```

```
//with the data type REAL
L  #var_real
Function
T  #Result

//with REAL and LREAL
CALL Function
  Data type
  IN      := #var_real
  RET_VAL := #Result

//with the data type LREAL
CALL Function_LREAL
  IN  := #var_lreal
  OUT := #Result
```

Function:

SIN	Sine
COS	Cosine
TAN	Tangent
ASIN	Arc sine
ACOS	Arc cosine
ATAN	Arc tangent
SQR	Generate square
SQRT	Extract square root
LN	Natural logarithm
EXP	Exponential function to base e

Fig. 13.14 Overview of math functions

The functions are displayed in LAD and FBD as EN/ENO boxes. They are displayed as functions with a function value in SCL. STL provides the corresponding statements for the data type REAL. With block calls, tags with the data type LREAL can also be processed in STL.

An error occurs if the permissible numerical range is left or if the input tag is an invalid floating-point number. The ENO output is then set to “0” in the case of LAD and FBD, the ENO tag and ENO output are set to FALSE in the case of SCL, and the overflow bits OV and OS are set to “1” in the case of STL.

Mathematical operations with STL

STL provides mathematical operations, which require the contents of accumulator 1 as tag with the data type REAL and then calculate the corresponding function. In addition, a mathematical function can be called with CALL and its data type REAL or LREAL can then be set via a drop-down list. In the *Long Functions* global library, there are system blocks which process tags with the data type LREAL (see also Chapter 10.5.4 “Math functions in the statement list” on page 426).

13.5.2 Trigonometric functions SIN, COS, TAN

The trigonometric functions generate the sine (SIN), cosine (COS), or tangent (TAN) of the input tag IN and deliver this in the result OUT. An angle in radians is expected at the input tag IN (Fig. 13.14).

Two units are commonly used for the magnitude of an angle, degrees from 0° to 360° (360th part of the circumference of a circle) and radians from 0 to 2π (with π = +3.141593e+00). Both units can be converted proportionately. For example, the value in radians for a 90° angle is π/2, in other words +1.570796e+00. With values larger than 2π (+6.283185e+00), 2π or a multiple thereof is subtracted until the input value for the trigonometric function is less than 2π.

An error occurs if the input tag IN is an invalid floating-point number, +∞ or −∞. The value of IN is then output in the result OUT.

13.5.3 Arc functions ASIN, ACOS, ATAN

The arc functions (inverse trigonometric functions) generate the arcsine (ASIN), arccosine (ACOS), or arctangent (ATAN) of the input tag IN and output this in the result OUT (Fig. 13.14). The arc functions are the inverse functions of the respective trigonometric function. They expect a number within a specific value range at the input tag IN and output an angle in radians (Table 13.1).

Table 13.1 Range of values of the arc functions

Function	Permissible range of values	Returned value
Arcsine ASIN	−1 to +1	−π/2 to +π/2
Arccosine ACOS	−1 to +1	0 to π
Arctangent ATAN	Complete range	−π/2 to +π/2

An error occurs if the input tag IN is not in the range ±1 (with ASIN or ACOS) or is an invalid floating-point number. An invalid floating-point number is then output in the result OUT.

13.5.4 Generate square and extract square root

The functions SQR (generate square) and SQRT (extract square root) are represented in Fig. 13.14 on page 579.

Generate square SQR

SQR generates the square of the input tag IN and outputs it in the result OUT.

$$OUT = IN^2$$

An error occurs if the input tag IN or the result is an invalid floating-point number. An invalid floating-point number is output in the result OUT in the first case and +∞ in the second case.

Extract square root SQRT

SQRT generates the square root of the input tag IN and outputs it in the result OUT.

$$OUT = \sqrt{IN}$$

An error occurs if the input tag IN is negative, an invalid floating-point number, or ±∞. An invalid floating-point number or ±∞ is then output in the result OUT.

13.5.5 Logarithm and power

The functions LN (natural logarithm) and EXP (exponentiation to base e) are shown in Fig. 13.14 on page 579. The power for any base can be calculated using EXPT (not for STL, Fig. 13.15 on page 582).

Calculate natural logarithm LN

LN calculates the natural logarithm to base e (= 2.718282e+00) from the input tag IN and outputs it in the result OUT.

$$OUT = \ln(IN)$$

An error occurs if:

- ▷ The input tag IN1 is zero, negative, −∞, or a negative invalid floating-point number. ∞ is then output in the result OUT.
- ▷ The input tag IN1 is +∞ or a positive invalid floating-point number. The value of IN1 is then output in the result OUT.

The natural logarithm is the inverse function of the exponential function: if $y = e^x$, then $x = \ln y$.

If you wish to calculate any logarithm, use the equation

$$\log_b a = \frac{\log_n a}{\log_n b}$$

where b or n is any base. If you set n = e, you can use the natural 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

$$\lg a = \frac{\ln a}{\ln 10} = 0.4342945 \cdot \ln a$$

Exponentiation to base e EXP

EXP generates the exponential from base e (= 2.718282e+00) and the input tag IN and outputs it in the result OUT.

$$OUT = e^{IN}$$

An error occurs if the input tag IN or the result is an invalid floating-point number. An invalid floating-point number is output in the result OUT in the first case and +∞ in the second case.

Exponentiation to any base EXPT

EXPT calculates the power from the base at parameter IN1 and the exponent at parameter IN2 and stores the result at parameter OUT (Fig. 13.15).

$$OUT = IN1^{IN2}$$

The enable output ENO or the tag ENO is set to signal state “0”

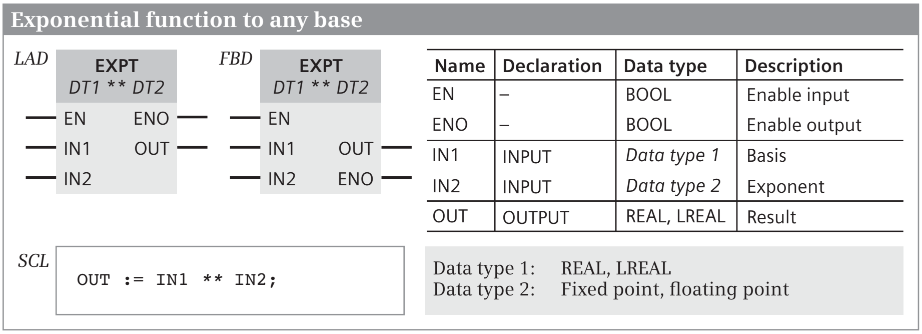
- ▷ if the value at parameter IN1 is +∞ and at parameter IN2 is not –∞. Then +∞ is output at the OUT parameter.
- ▷ if the value at parameter IN1 is –∞ or negative. Then an invalid floating-point number is output at the OUT parameter (if IN2 is a floating-point number), otherwise –∞.
- ▷ if the value 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.

13.5.6 Extract decimal points, generate absolute value and negation

Further math functions are:

- ▷ FRAC Extract decimal points
- ▷ ABS Generate absolute value
- ▷ NEG Generate negation (two's complement)

You can find the math functions in the program elements catalog under *Basic instructions* > *Math functions* (LAD, FBD, SCL) or under *Basic instructions* > *Basic instructions* > *Math functions* (STL).



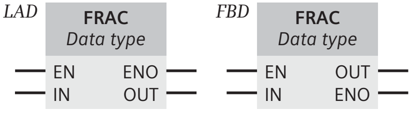
```
OUT := IN1 ** IN2;
```

Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN1	INPUT	*Data type 1*	Basis
IN2	INPUT	*Data type 2*	Exponent
OUT	OUTPUT	REAL, LREAL	Result

Data type 1: REAL, LREAL
Data type 2: Fixed point, floating point

Fig. 13.15 Math function EXPT

Extract decimal points FRAC



Name	Declaration	Data type	Description
EN	-	BOOL	Enable input
ENO	-	BOOL	Enable output
IN	INPUT	REAL, LREAL	Input tag
OUT	OUTPUT	REAL, LREAL	Result

SCL

```
#var_OUT :=
  FRAC_Data_type (#var_IN);
```

STL

```
CALL FRAC
  Data type
  IN      := ...
  RET_VAL := ...
```

Fig. 13.16 Extracting decimal points

Extract decimal points FRAC

FRAC extracts the decimal places from the number present at parameter IN and outputs them at parameter OUT (Fig. 13.16).

The enable output ENO is set to signal state “0” if the value at parameter IN is an invalid floating-point number or ±∞. Then a positive invalid floating-point number is output at parameter OUT.

Generate absolute value ABS

ABS generates the amount from the number at parameter IN and outputs the result at parameter OUT. For a floating-point number, the sign of the mantissa is set to “0” (Fig. 13.17).

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”.

Generate negation, two's complement NEG

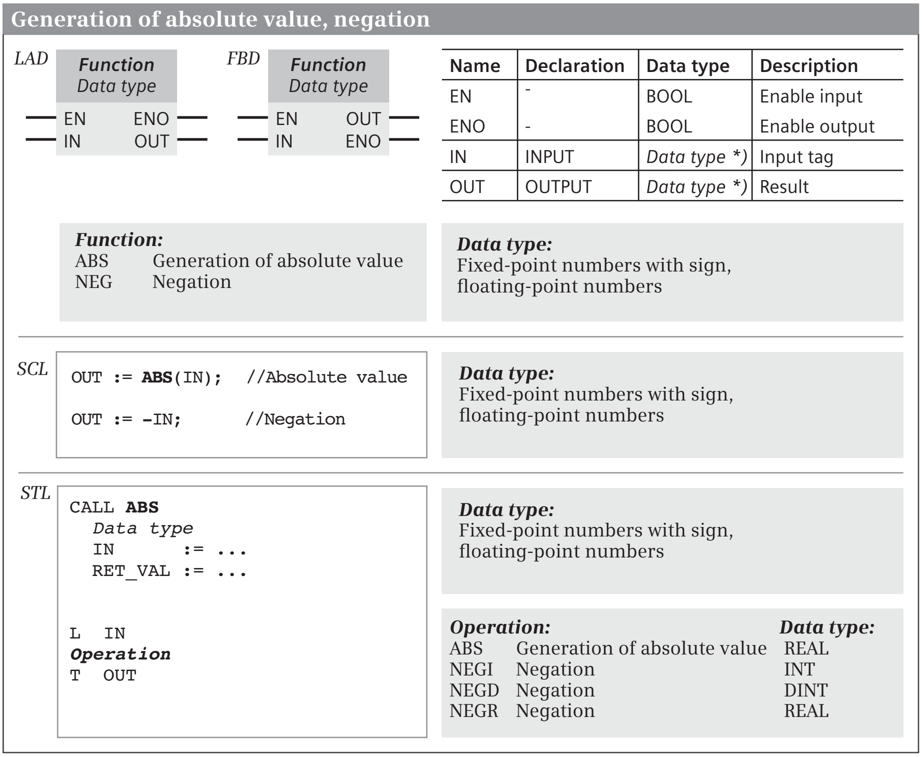
NEG changes the sign of the number at parameter IN and outputs the result at parameter OUT. The negation is equivalent to multiplying by −1. For a floating-point number, the sign of the mantissa is changed, even for an invalid floating-point number (Fig. 13.17).

If the result is outside of the valid number range, e.g. NEG(−128) for the data type SINT, the output ENO is set to signal state “0”.

For STL, the operations NEGI, NEGD, and NEGR form the two's complement of the number in accumulator 1, which must have the “suitable” data type.

13.5.7 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



Generation of absolute value, negation

Name	Declaration	Data type	Description
EN	-	BOOL	Enable input
ENO	-	BOOL	Enable output
IN	INPUT	*Data type *)*	Input tag
OUT	OUTPUT	*Data type *)*	Result

Function:

ABS	Generation of absolute value
NEG	Negation

Data type:
Fixed-point numbers with sign,
floating-point numbers

SCL

```
OUT := ABS(IN);   //Absolute value

OUT := -IN;       //Negation
```

Data type:
Fixed-point numbers with sign,
floating-point numbers

STL

```
CALL ABS
  Data type
  IN      := ...
  RET_VAL := ...


L  IN
Operation
T  OUT
```

Data type:
Fixed-point numbers with sign,
floating-point numbers

Operation:		***Data type:***
ABS	Generation of absolute value	REAL
NEGI	Negation	INT
NEGD	Negation	DINT
NEGR	Negation	REAL

Fig. 13.17 Generate absolute value and negation

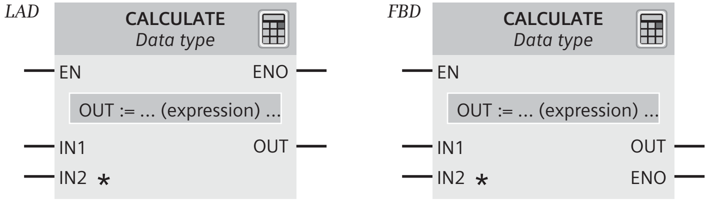
linked as input parameters of the box and specify the data type of the expression (the output parameter). The logic operation function is specified in a dialog (Fig. 13.18).

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 also 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.

CALCULATE box



Function:
The input tags are linked to each other according to a freely definable expression and the result is output at the parameter OUT.
The number of input parameters is expandable.

Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN1	INPUT	*Data type*	Input tag 1
IN2	INPUT	*Data type*	Input tag 2
OUT	OUTPUT	*Data type*	Result

Data types:	***Usable functions:***
BYTE, WORD, DWORD, LWORD	AND, OR, XOR, NOT
WORD, DWORD, LWORD	AND, OR, XOR, NOT, SWAP()
USINT, UINT, UDINT, ULINT, SINT, INT, DINT, LINT	+, –, *, /, MOD, NOT, –(), ABS()
REAL, LREAL	+, –, *, /, MOD, NOT, –(), ABS(), **, SQR(), SQRT(), LN(), EXP(), SIN(), COS(), TAN(), ASIN(), ACOS(), ATAN(), TRUNC(), ROUND(), CEIL(), FLOOR(), FRAC()

Fig. 13.18 CALCULATE box, representation and function

Linking of bit sequences

Input parameters with the data types BYTE, WORD, DWORD, and LWORD 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 sequence can be inverted with the NOT operator (one's complement formation INV). With SWAP the bytes of a bit sequence can be replaced (not with data type BYTE).

Linking of fixed-point numbers

Input parameters with data types USINT, UINT, UDINT, ULINT, SINT, INT, DINT, and LINT 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 exponentiation (**), 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.

13.6 Conversion functions

If you link tags together, they must have a compatible data type. This also applies if you assign values or supply function and block parameters. If a tag is not available in the required data type and no implicit data type conversion is possible, the data type must be converted. The conversion functions are available for this.

The following conversion functions are available:

- Conversion with CONVERT
- Conversion with ROUND, CEIL, FLOOR, TRUNC
- Conversion of time data types (T_CONV)
- Conversion of string data types (S_CONV, STRG_TO_CHARS, CHARS_TO_STRG, STRG_VAL, VAL_STRG)
- Conversion of hexadecimal numbers (ATH, HTA)
- Scaling and normalizing (SCALE_X, NORM_X, SCALE, UNSCALE).

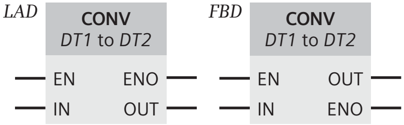
These conversion functions are "explicit" conversion functions where the bit assignments of the tags change or where conversion errors can occur, for example a conversion from DINT to REAL. These conversions must be programmed. Then there are also the "implicit" conversion functions, which "automatically" convert a data type into a compatible data type. Further details can be found in Chapter 4.5.2 "Implicit data type conversion" on page 108.

13.6.1 Data type conversion with the conversion function CONVERT

Fig. 13.19 shows the conversion function CONVERT for the various programming languages. You can find CONVERT in the program elements catalog under *Basic instructions > Conversion operations*.

For LAD and FBD, CONVERT is displayed as an EN/ENO box. For SCL, there are functions with the notation *Source data type_TO_destination data type*, and for STL, you enter the statement CALL CONVERT. In addition to CONVERT, STL provides opera-

Conversion function CONVERT



SCL

```
OUT := DT1_TO_DT2(IN);
```

STL

```
CALL CONVERT
  DT1_TO_DT2
  IN      := ...
  RET_VAL := ...
```

Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	*Data type 1*	Input tag
OUT	OUTPUT	*Data type 2*	Output tag

CONVERT transfers the value of the tag with data type DT1 at parameter IN to the tag with data type DT2 at parameter OUT. The permissible data types and the transfer method are described in the text.

Fig. 13.19 Conversion function CONVERT

tions which directly convert the contents of accumulator 1. Table 13.2 shows the data type conversions possible with CONVERT.

In the event of a conversion error, the ENO output is then set to “0” in the case of LAD and FBD, the ENO tag and ENO output are set to FALSE in the case of SCL, and the overflow bits OV and OS are set to “1” in the case of STL.

Data type conversion of bit-serial data types with CONVERT

If the source tag has the data type BOOL, then its value is transferred into bit 0 of the destination tag. If the destination tag has the data type BOOL, then the value of bit 0 from the source tag is transferred to the destination tag. If this bit has the value “1” and there are still more bits with the value “1” in the source tag, the ENO output is also set to “0” (FALSE).

The bit pattern of a source tag with the data type BYTE, WORD, DWORD or LWORD is transferred to the destination tag starting from the right (from bit 0). If the source tag has less bits than the destination tag, the unassigned bits are filled with “0”. If the source tag has more bits than the destination tag, the remaining bits are ignored.

Data type conversion of fixed-point numbers with CONVERT

If the destination tag has the data type BOOL, then the value of bit 0 from the source tag is transferred to the destination tag. If this bit has the value “1” and there are still more bits with the value “1” in the source tag, the ENO output is also set to “0” (FALSE).

If the destination tag has the data type CHAR, BYTE, WORD, DWORD or LWORD, the bit pattern of the source tag is entered in the destination tag starting from the right (from bit 0). If the source tag has a negative sign or if bits are lost, the ENO output is set to “0” (FALSE).

Table 13.2 Explicit data type conversion with CONVERT

to from	BOOL	BYTE	WORD	DWORD	LWORD	USINT	UINT	UDINT	ULINT	SINT	INT	DINT	LINT	REAL	LREAL	S5TIME	TIME	LTIME	DATE	TOD	LTOD	DT	LDT	DTL	CHAR	STRING
BOOL		X	X	X	X	X	X	X	X	X	X	X	X													
BYTE	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X		X	
WORD	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X		X	
DWORD	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X		X	
LWORD	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X		X	
USINT	X	X	X	X	X		X	X	X	X	X	X	X	X	X		X	X	X	X	X		X		X	X
UINT	X	X	X	X	X	X		X	X	X	X	X	X	X	X		X	X	X	X	X		X		X	X
UDINT	X	X	X	X	X	X	X		X	X	X	X	X	X	X		X	X	X	X	X		X		X	X
ULINT	X	X	X	X	X	X	X	X		X	X	X	X	X	X		X	X	X	X	X		X		X	X
SINT	X	X	X	X	X	X	X	X	X		X	X	X	X	X		X	X	X	X	X		X		X	X
INT	X	X	X	X	X	X	X	X	X	X		X	X	X	X		X	X	X	X	X		X		X	X
DINT	X	X	X	X	X	X	X	X	X	X	X		X	X	X		X	X	X	X	X		X		X	X
LINT	X	X	X	X	X	X	X	X	X	X	X	X		X	X		X	X	X	X	X		X		X	X
REAL		X	X	X	X	X	X	X	X	X	X	X	X		X											X
LREAL		X	X	X	X	X	X	X	X	X	X	X	X	X												X
S5TIME			X														X	X								
TIME		X	X	X	X	X	X	X	X	X	X	X	X			X		X		X						
LTIME		X	X	X	X	X	X	X	X	X	X	X	X			X	X				X		X			
DATE		X	X	X	X	X	X	X	X	X	X	X	X									X	X	X		
TOD		X	X	X	X	X	X	X	X	X	X	X	X				X				X	X	X	X		
LTOD		X	X	X	X	X	X	X	X	X	X	X	X					X		X		X	X	X		
DT																			X	X	X		X	X		
LDT		X	X	X	X	X	X	X	X	X	X	X	X					X	X	X	X	X		X		
DTL																			X	X	X	X	X			
CHAR		X	X	X	X	X	X	X	X	X	X	X	X													X
STRING						X	X	X	X	X	X	X	X	X	X										X	

Additionally: BCD16 > INT
INT > BCD16 BCD32 > DINT
DINT > BCD32

If the destination tag has a fixed-point data type, the value of the source tags is expanded with the correct sign and transferred into the destination tag. If the value of the source tags is negative and the destination data type is an unsigned fixed-point data type or if bits are lost, the ENO output is set to “0” (FALSE).

If the destination tag has a floating-point data type, the value of the source tag is converted to a floating-point number in the proper format.

If the destination tag has a time data type, the bit pattern of the source tag is transferred into the destination tag with no changes.

If the destination tag has the STRING data type, the value of the source tag is converted to a string in the proper format. The destination tag must be at least 21 characters long. If it is shorter, the ENO output is set to “0” (FALSE).

Data type conversion of BCD numbers with CONVERT

A source tag with the data type BCD16 (3 decades + sign) can be converted to a destination tag with the data type INT in the proper format. A source tag with the data type BCD32 (7 decades + sign) can be converted to a destination tag with the data type DINT in the proper format. If the bit pattern contains an invalid tetrad, no conversion takes place and the ENO output is set to “0” (FALSE).

A source tag with the data type INT can be converted to a destination tag with the data type BCD16 (3 decades + sign). A source tag with the data type DINT can be converted to a destination tag with the data type BCD32 (7 decades + sign). If the result of the conversion is outside of the BCDS numerical range, no conversion takes place and the ENO output is set to “0” (FALSE).

Data type conversion of floating-point numbers with CONVERT

If the destination tag has the data type BYTE, WORD, DWORD or LWORD, the bit pattern of the source tag is entered in the destination tag starting from the right (from bit 0). If the source tag has less bits than the destination tag, the unassigned bits are filled with “0”. If the source tag has more bits than the destination tag, the remaining bits are ignored.

If the destination tag has a fixed-point data type, the value of the source tag is converted and transferred into the destination tag in the proper format. If the source tag is an invalid floating-point number or the value of the source tag departs from the numerical range of the destination data type, the ENO output is set to “0” (FALSE).

If the destination tag has the STRING data type, the value of the source tag is converted to a string in the proper format. The destination tag must be at least 14 characters long. If it is shorter or if the source tag is an invalid floating-point number, the ENO output is set to “0” (FALSE).

Data type conversion of date and time with CONVERT

If the destination tag has the data type BYTE, WORD, DWORD or LWORD, the bit pattern of the source tag is entered in the destination tag starting from the right (from bit 0). If the source tag has less bits than the destination tag, the unassigned bits are filled with "0". If the source tag has more bits than the destination tag, the remaining bits are ignored.

If the destination tag has a fixed-point data type, the value of the source tag is converted and transferred into the destination tag in the proper format. If signs are switched, the ENO output is set to "0" (FALSE).

The value of a source tag with data type S5TIME is transferred in the proper format to a destination tag with the data type TIME or LTIME. The value of a source tag with data type TIME is transferred in the proper format to a destination tag with the data type S5TIME, LTIME or TOD. If the value range of the destination tag is exceeded, no conversion takes place. The value of a source tag with the data type LTIME is transferred in the proper format to a destination tag with the data type S5TIME, TIME, TOD or LDT. If the value range of the destination tag is exceeded, no conversion takes place.

The value of a source tag with the data type DATE is transferred in the proper format to a destination tag with the data type DT, LDT or DTL.

The value of a source tag with the data type TOD is transferred in the proper format to a destination tag with the data type TIME (time since midnight), LTOD, DT, DTL or LDT. For DT, DTL and LDT, the time of day is replaced.

The value of a source tag with the data type LTOD is transferred in the proper format to a destination tag with the data type LTIME (time since midnight), TOD, DT, DTL or LDT. For DT, DTL and LDT, the time of day is replaced.

The value of a source tag with the data type DT is transferred in the proper format to a destination tag with the data type TOD, LTOD, LDT or DTL. If the destination tag has the data type DATE, the bit pattern is transferred right-aligned and unchanged.

The value of a source tag with the data type LDT is transferred in the proper format to a destination tag with the data type TOD, LTOD, DT or DTL. If the destination tag has the data type LTIME or DATE, the bit pattern is transferred right-aligned and unchanged.

The value of a source tag with the data type DTL is transferred in the proper format to a destination tag with the data type DT or LDT. If the destination tag has the data type TOD or LTOD, the time of day is transferred from the source tag to the destination tag in the proper format.

If the destination tag has the data type DATE, the date is transferred in the proper format from the source tag to the destination tag. If there is an area overflow, the ENO output is set to "0" (FALSE).

Data type conversion of string formats with CONVERT

The bit pattern of a source tag with the data type CHAR is transferred, starting from the right (bit 0), to the destination tag with a bit serial or fixed-point data type. If the source tag has less bits than the destination tag, the unassigned bits are filled with “0”. If the destination tag has the STRING data type, the source tag is entered in the first character. If the length of the destination tag is undefined, “1” is entered for the length.

If the source tag has the data type STRING and the destination tag has a fixed-point or floating-point data type, the sign and/or digits are permitted as characters in the character string. Leading spaces are ignored. The period serves as the decimal point. The comma is permitted as the thousands separator. The conversion takes place to the first invalid character or to the end of the string. If the structure of the string is invalid or if the numerical range is exceeded, the ENO output is set to “0” (FALSE).

If the source tag has the data type STRING and the destination tag has the data type CHAR, the first character of the string is transferred to the destination tag. If the string is empty, 16#00 is entered.

13.6.2 Data type conversion with ROUND, CEIL, FLOOR, and TRUNC

The conversion functions ROUND, CEIL, FLOOR and TRUNC convert floating-point numbers into fixed-point numbers or into whole floating-point numbers (without decimal places). You can find these conversion functions in the program elements catalog under *Basic instructions* > *Conversion operations*. Fig. 13.20 shows the conversion functions in the various programming languages. For LAD and FBD, these are represented as EN/ENO box. For SCL, they are functions with an input value and for STL, they are block calls.

In addition, STL provides operations, which directly convert the contents of accumulator 1 (see Chapter 10.5.5 “Conversion functions in the statement list” on page 428).

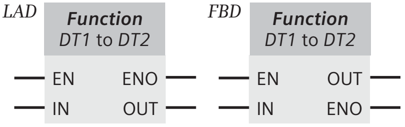
For SCL, the data type of the output tag is set to DINT by default and can be omitted in the notation. If you want to set a different data type, specify it after the conversion function.

In the event of a conversion error or an invalid floating-point number as input tag, the ENO output is set to “0” in the case of LAD and FBD, the ENO tag and ENO output are set to FALSE in the case of SCL, and the overflow bits OV and OS are set to “1” in the case of STL.

ROUND converts a fractional number into an integer and returns 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.

CEIL converts a fractional number into an integer and returns an integer which is greater than or equal to the input value.

Conversion functions ROUND, CEIL, FLOOR, and TRUNC



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	*Data type 1*	Input tag
OUT	OUTPUT	*Data type 2*	Output tag

SCL

```
OUT := Function_DT2(IN);
```

STL

```
CALL Function
  DT1  DT2
  IN      := ...
  RET_VAL := ...
```

Function:
Conversion of a floating-point number into a fixed-point number

ROUND	With rounding to the next integer
CEIL	With rounding to the next higher integer
FLOOR	With rounding to the next lower integer
TRUNC	Without rounding

Data types:

Data type 1:	Floating-point data type
Data type 2:	Fixed-point or floating-point data type

Fig. 13.20 Conversion functions ROUND, CEIL, FLOOR, and TRUNC

FLOOR converts a fractional number into an integer and returns an integer which is less than or equal to the input value.

TRUNC converts a fractional number to an integer and returns the integer portion of the input value; the fractional part is "truncated".

Table 13.3 shows the different effects of the conversion functions. The range between –1 and +1 has been selected as an example.

13.6.3 Data type conversion with T_CONV

The conversion function T_CONV transfers a value from a tag with a time data type to another tag and vice versa. Fig. 13.21 shows the conversion function in the various programming languages. You find T_CONV in the program elements catalog under *Extended instructions* > *Date and time-of-day*.

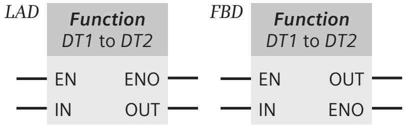
The conversion is carried out in the same way as with the conversion function CONVERT and the data types S5TIME, TIME, LTIME, DATE, TOD, LTOD, DT, LDT, and DTL. The description is provided in Chapter 13.6.1 "Data type conversion with the conversion function CONVERT" on page 586.

In the event of a conversion error, the ENO output is set to "0" in the case of LAD and FBD, the ENO tag and ENO output are set to FALSE in the case of SCL, and the overflow bits OV and OS are set to "1" in the case of STL.

Table 13.3 Rounding modes when converting fractional numbers

Input value		Result			
REAL	DW#16#	ROUND	CEIL	FLOOR	TRUNC
1.0000001	3F80 0001	1	2	1	1
1.00000000	3F80 0000	1	1	1	1
0.99999995	3F7F FFFF	1	1	0	0
0.50000005	3F00 0001	1	1	0	0
0.50000000	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
–0.50000000	BF00 0000	0	0	–1	0
–0.50000005	BF00 0001	–1	0	–1	0
–0.99999995	BF7F FFFF	–1	0	–1	0
–1.00000000	BF80 0000	–1	–1	–1	–1
–1.0000001	BF80 0001	–1	–1	–2	–1

Conversion functions T_CONV and S_CONV



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	*Data type 1*	Input tag
OUT	OUTPUT	*Data type 2*	Output tag

SCL

```
OUT := DT1_TO_DT2(IN);
```

STL

```
CALL Function
  DT1_TO_DT2
  IN  := ...
  OUT := ...
```

Function:
T_CONV Conversion of time data types
S_CONV Conversion of string data types

The function transfers the value of the tag with data type DT1 at parameter IN to the tag with data type DT2 at parameter OUT. The permissible data types and the transfer method are described in the text.

Fig. 13.21 Conversion functions T_CONV and S_CONV

13.6.4 Data type conversion with S_CONV

The conversion function S_CONV transfers a value from a tag with a string data type to another tag and vice versa. Fig. 13.21 shows the conversion function in the various programming languages. S_CONV can be found in the program elements catalog under *Extended instructions > String + Char*.

In the event of a conversion error, the ENO output is set to "0" in the case of LAD and FBD, the ENO tag and ENO output are set to FALSE in the case of SCL, and the overflow bits OV and OS are set to "1" in the case of STL.

The conversion is carried out in the same way as with the conversion function CONVERT and the data types STRING and CHAR. The description is provided in Chapter 13.6.1 "Data type conversion with the conversion function CONVERT" on page 586.

13.6.5 Conversion functions STRG_TO_CHARS and CHARS_TO_STRG

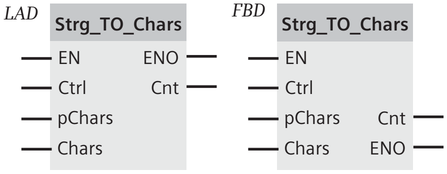
The function STRG_TO_CHARS converts a string with data type STRING into an array with data type ARRAY OF CHAR or ARRAY OF BYTE. The function CHARS_TO_STRG converts an array with data type ARRAY OF CHAR or ARRAY OF BYTE into a string with data type STRING. For LAD and FBD, the conversion functions are EN/ENO boxes. For SCL, they are function calls with a function value and for STL, they are block calls. You can find the conversion functions in the program elements catalog under *Extended instructions > String + Char*. Fig. 13.22 shows the conversion functions in the various programming languages.

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 the conversion, e.g. if the destination area is too small to accommodate the copied characters, the ENO output is set to signal state "0" for LAD and FBD and the ENO tag and the ENO output are set to FALSE for SCL.

Conversion function STRG_TO_CHARS



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
Ctrl	INPUT	STRING	Character string
pChars	INPUT	DINT	Position
Chars	INOUT	VARIANT	Character array
Cnt	OUTPUT	UINT	Quantity

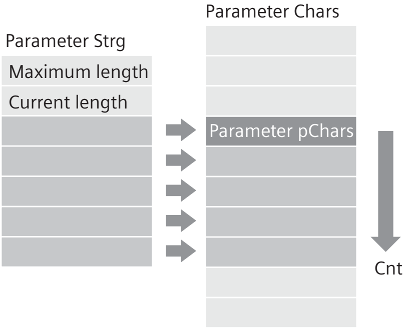
SCL

```
Strg_TO_Chars(
  Strg   := ... ,
  pChars := ... ,
  Cnt    => ... ,
  Chars  := ... );
```

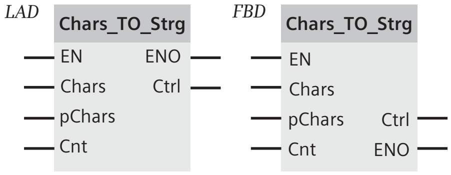
STL

```
CALL Strg_TO_Chars
  Strg   := ...
  pChars := ...
  Cnt    := ...
  Chars  := ...
```

Function:
The string at the STRG parameter is inserted beginning at the position PCHARS in the character array at the CHARS parameter with the data type ARRAY OF CHAR or ARRAY OF BYTE. The parameter CNT indicates the number of inserted characters.



Conversion function CHARS_TO_STRG



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
Chars	INPUT	VARIANT	Character array
pChars	INPUT	DINT	Position
Cnt	INPUT	UINT	Quantity
Ctrl	OUTPUT	STRING	Character string

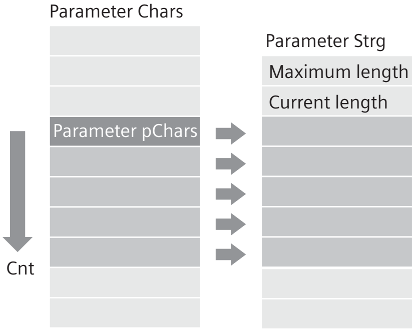
SCL

```
Chars_TO_Strg(
  Chars  := ... ,
  pChars := ... ,
  Cnt    => ... ,
  Strg   := ... );
```

STL

```
CALL 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 beginning at the position PCHARS are taken, converted to a string, and output at the STRG parameter.



Valid characters:
all characters with ASCII coding

Fig. 13.22 Conversion functions STRG_TO_CHARS and CHARS_TO_STRG

13.6.6 Conversion functions STRG_VAL and VAL_STRG

The function STRG_VAL converts a string into a numerical value. The function VAL_STRG converts a numerical value into a string. For LAD and FBD, the conversion functions are EN/ENO boxes and for STL, they are block calls. These conversion functions are not available in SCL. You can find the conversion functions in the program elements catalog under *Extended instructions* > *String + Char*.

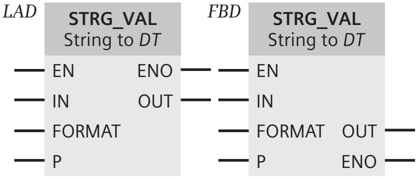
Conversion of a string into a number (STRG_VAL)

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. 13.23).

The first character to be converted must be a number, a sign, or a space. Leading spaces are ignored. If the decimal point is a period (bit 0 in FORMAT is “0”), then the comma is allowed as thousands separator to the left of the decimal point and it is ignored. If the comma is used as the decimal point, then the period is allowed as the thousands separator and it is ignored.

In the event of an error, zero is output at the parameter OUT and the ENO output is set to “0”.

Conversion function STRG_VAL



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	STRING	String
FORMAT	INPUT	WORD	Format specifications
P	IN_OUT	UINT	Character position
OUT	OUTPUT	*Data type*	Result

SCL (not available)

STL

```
CALL STRG_VAL
  String TO data type
  IN     := ...
  FORMAT := ...
  P      := ...
  OUT    := ...
```

Function:
The STRING tag present 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:
Fixed-point and floating-point data types

Structure of the FORMAT parameter

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	N	D

D = Decimal separator	"0" = Point "1" = Comma
N = Notation	"0" = Decimal fraction "1" = Exponential

Fig. 13.23 Conversion function STRG_VAL

Conversion of a number into a string (VAL_STRG)

The conversion function VAL_STRG converts a numerical value present at parameter IN into a string and outputs it at parameter OUT. OUT must be assigned a valid STRING tag which is long enough to accommodate the converted value (Fig. 13.24).

The first converted character is written in the STRING tag at the position specified by parameter P. If P is longer than the current length of the string, spaces are appended up to position P. After the conversion, the position of the next, unreplaced character in the string is present in P.

The SIZE parameter specifies the number of integer places. If the converted value occupies fewer spaces, leading spaces are inserted.

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 zeroes. The digits before the "E" are assigned as in fixed-point notation.

If an error occurs when processing the conversion function, the ENO output is set to "0". The OUT parameter then contains a value with leading spaces and a "C" as the last character.

13.6.7 Data type conversion of hexadecimal numbers

The ATH function converts a string of ASCII-coded characters into a string of hexadecimal numbers. The HTA function converts a string of hexadecimal numbers into a string of ASCII-coded characters. For LAD and FBD, the conversion functions are EN/ENO boxes. For SCL, they are function calls with a function value and for STL, they are block calls. You can find the conversion functions in the program elements catalog under *Extended instructions > String + Char*. Fig. 13.25 shows the conversion functions in the various programming languages.

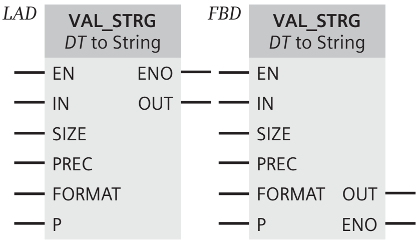
The number of characters to be converted is specified at parameter N. Up to 32 767 characters are permissible.

ATH Conversion from ASCII to hexadecimal

ATH converts a string present in ASCII code into a string in hexadecimal code. Only the digits 0 to 9, the uppercase letters A to F, and the lowercase letters a to f are permissible. An illegal character is converted into zeroes and an error message is output at the RET_VAL parameter.

At parameter IN, a tag with a data type STRING, ARRAY_OF_CHAR or ARRAY_OF_BYTE can be specified. At the parameter OUT, a tag with a data type bit sequence, fixed-point number, ARRAY_OF_CHAR or ARRAY_OF_BYTE can be specified.

Conversion function VAL_STRG



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	*Data type*	Numerical values
SIZE	INPUT	USINT	Number of characters
PREC	INPUT	USINT	Decimal places
FORMAT	INPUT	WORD	Format specifications
P	IN_OUT	UINT	Character position
OUT	OUTPUT	STRING	Result

SCL (not available)

STL

```
CALL VAL_STRG
  Data type TO String
  IN     := ...
  SIZE   := ...
  PREC   := ...
  FORMAT := ...
  P      := ...
  OUT    := ...
```

Function:
The numerical value at the IN parameter is converted into a STRING tag and output at the OUT parameter.
The conversion format is defined by the FORMAT parameter.

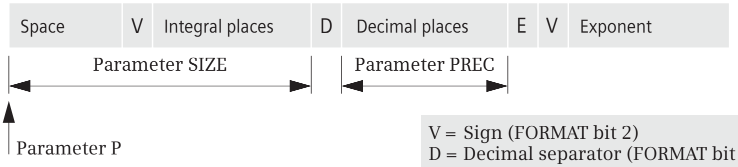
Data type:
Fixed-point and floating-point data types

Structure of the FORMAT parameter

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	V	N	D

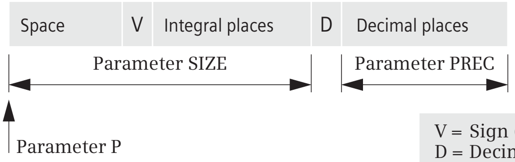
D = Decimal separator	"0" = Point "1" = Comma
N = Notation	"0" = Decimal fraction "1" = Exponential
V = Sign	"0" = with "+" and "–" "1" = only with "–"

Output format for exponential representation (FORMAT bit 1 = "1")



V = Sign (FORMAT bit 2)
D = Decimal separator (FORMAT bit 0)
E = Exponential identifier "E"

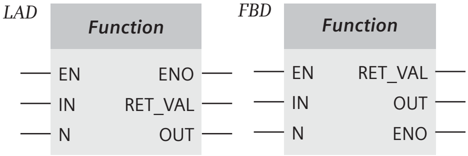
Output format for decimal fraction representation (FORMAT bit 1 = "0")



V = Sign (FORMAT bit 2)
D = Decimal separator (FORMAT bit 0)

Fig. 13.24 Conversion function VAL_STRG

Conversion functions ATH and HTA



Function:
ATH Conversion of a string into a hexadecimal number
HTA Conversion of a hexadecimal number into a string

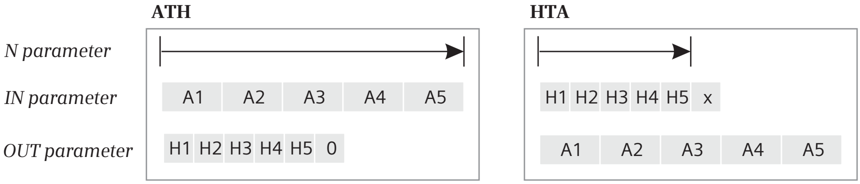
SCL

```
#var_RET_VAL := Function (
            IN  := ...
            N   := ...
            OUT => ... );
```

STL

```
CALL Function
  IN      := ...
  N       := ...
  RET_VAL := ...
  OUT     := ...
```

Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	*Data type 1*	Input tag
N	INPUT	INT	Quantity
RET_VAL	RETURN	WORD	Error information
OUT	OUTPUT	*Data type 2*	Output tag



In the case of an odd quantity, a zero is written into the last half-byte.

Valid characters:
Digits 0 to 9
Uppercase letters A to F
Lowercase letters a to f

Fig. 13.25 Conversion functions ATH and HTA

HTA Conversion from hexadecimal to ASCII

HTA converts a string present in hexadecimal code into an ASCII-coded string. The result is output with the digits 0 to 9 and with the uppercase letters A to F.

At parameter IN, a tag with a data type bit sequence, fixed-point number, ARRAY_OF_CHAR or ARRAY_OF_BYTE can be specified. At the parameter OUT, a tag with a data type STRING, ARRAY_OF_CHAR or ARRAY_OF_BYTE can be specified.

13.6.8 Scaling and normalizing

For converting a value from one value range to another, you have the following functions at your disposal:

▷ SCALE_X
A value in the numerical range from 0 to 1 is converted (scaled) to a value range that is defined by a lower and upper limit value.

▷ NORM_X
A value in a value range which is defined by a lower and upper limit value is converted (standardized) to the numerical range from 0 to 1.

- ▷ SCALE
  A value in the range from 0 to 27 648 (unipolar) or from –27 468 and 27 648 (bipolar) is converted to a value range which is defined by a lower and upper limit value. Application: Conversion of an analog value read from a module into a value with physical units.

- ▷ UNSCALE
  A value in a range which is defined by a lower and upper limit value is converted to a value range from 0 to 27 648 (unipolar) or from –27 468 and 27 648 (bipolar). Application: Conversion of a value with physical units into an analog value written to a module.

For LAD and FBD, these functions are EN/ENO boxes. For SCL, they are function calls with a function value and for STL, they are block calls. You can find these functions in the program elements catalog under *Basic instructions* > *Conversion operations*.

Conversion with SCALE_X

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. 13.26).

Please note: The value created at the parameter VALUE must be within the limits of 0 and 1 (inclusive)! If this is not the case, the value at the output parameter OUT can be less than MIN or greater than MAX. If it is still within the permissible value range for the data type, SCALE_X does not report 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 parameter 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).

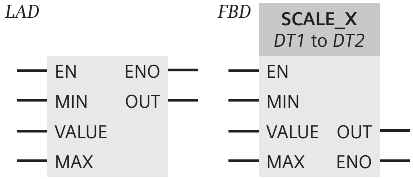
Conversion with NORM_X

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. 13.26).

Please note: The value created at the parameter VALUE must be within the limits of MIN and MAX (inclusive)! If this is not the case, the value at the output parameter OUT can be less than 0 and greater than 1. NORM_X does not report any errors (ENO = “1”).

The function NORM_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 OUT data type 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).

Scaling SCALE_X



SCL

```
#var_OUT := SCALE_X_DT2(
  MIN    := ... ,
  VALUE  := ... ,
  MAX    := ... );
```

STL

```
CALL SCALE_X
  DT1  DT2
  MIN     := ...
  VALUE   := ...
  MAX     := ...
  RET_VAL := ...
```

Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
MIN	INPUT	*Data type 2*	Lower limit
VALUE	INPUT	*Data type 1*	Input tag
MAX	INPUT	*Data type 2*	Upper limit
OUT	OUTPUT	*Data type 2*	Result

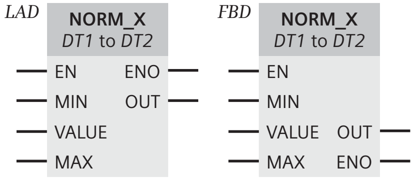
Function:
The tag value created at the parameter VALUE is converted linearly between the limits MIN and MAX and output at the parameter OUT.

OUT := VALUE × (MAX – MIN) + MIN

$0 < \text{VALUE} \leq 1$

Data type:
Data type 1: Floating-point number
Data type 2: Fixed-point and floating-point number

Normalizing NORM_X



SCL

```
#var_OUT := NORM_X_DT2(
  MIN    := ... ,
  VALUE  := ... ,
  MAX    := ... );
```

STL

```
CALL SCALE_X
  DT1  DT2
  MIN     := ...
  VALUE   := ...
  MAX     := ...
  RET_VAL := ...
```

Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
MIN	INPUT	*Data type 1*	Lower limit
VALUE	INPUT	*Data type 1*	Input tag
MAX	INPUT	*Data type 1*	Upper limit
OUT	OUTPUT	*Data type 2*	Result

Function:
The tag value created at the parameter VALUE is converted linearly between the limits MIN and MAX and output at the parameter OUT.

OUT := (VALUE – MIN) / (MAX – MIN)

$0 < \text{OUT} \leq 1$

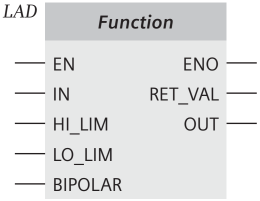
Data type:
Data type 1: Floating-point number
Data type 2: Fixed-point and floating-point number

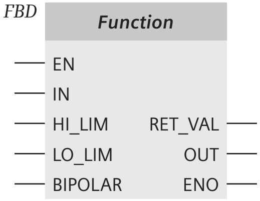
Fig. 13.26 Conversion functions SCALE_X and NORM_X

Conversion with SCALE

SCALE converts a fixed-point number between the limits 0 and +27 648 (unipolar) or between the limits –27 648 and +27 648 (bipolar) into a floating-point number and scales it between a lower limit and upper limit specified by you (Fig. 13.27). Example of the application: Conversion of an analog value from an analog input module into a value with physical units.

Conversion functions for scaling and unscaling





SCL

```
#var_RET_VAL := Function(
     IN       := ... ,
     HI_LIM   := ... ,
     LO_LIM   := ... ,
     BIPOLAR  := ... ,
     OUT      => ... );
```

STL

```
CALL Function
  IN       := ...
  HI_LIM   := ...
  LO_LIM   := ...
  BIPOLAR  := ...
  RET_VAL  := ...
  OUT      := ...
```

Name	Declaration	Data type		Description
		SCALE	UNSCALE	
EN	–	BOOL	BOOL	Enable input
ENO	–	BOOL	BOOL	Enable output
IN	INPUT	INT	REAL	Lower limit
HI_LIM	INPUT	REAL	REAL	Input tag
LO_LIM	INPUT	REAL	REAL	Input tag
BIPOLAR	INPUT	BOOL	BOOL	Upper limit
RET_VAL	RETURN	WORD	WORD	Error information
OUT	OUTPUT	REAL	INT	Result

Function:

SCALE	Scale
UNSCALE	Unscale

SCALE

Conversion formula:

$$OUT = [\ \frac{IN — K1}{K2 — K1} \times (HI\_LIM - LO\_LIM)\ ] + LO\_LIM \qquad K1 < IN < K2$$

There are two different cases:

1) The value of IN is unipolar in the range from 0 to +27 468; in this case the BIPOLAR parameter is assigned "0" and the constants have the values K1 = 0 and K2 = +27 468.

2) The value of IN is bipolar in the range from -468 to +27 468; in this case the BIPOLAR parameter is assigned "1" and the constants have the values K1 = -468 and K2 = +27 468.

UNSCALE

Conversion formula:

$$OUT = [\ \frac{IN - LO\_LIM}{HI\_LIM - LO\_LIM} \times (K2 - K1) + K1 \qquad K1 < OUT < K2$$

There are two different cases:

1) The value of OUT is unipolar in the range from 0 to +27 468; in this case the BIPOLAR parameter is assigned "0" and the constants have the values K1 = 0 and K2 = +27 468.

2) The value of OUT is bipolar in the range from -468 to +27 468; in this case the BIPOLAR parameter is assigned "1" and the constants have the values K1 = -468 and K2 = +27 468.

Fig. 13.27 Conversion functions with SCALE and UNSCALE

If the value at IN is greater than 27 648, the upper limit is output and an error signaled. If the value at IN is less than K1 (see Fig. 13.27), the lower limit is output and an error signaled. If the lower limit is greater than the upper limit, the result is scaled inversely proportional to the input value.

Conversion with UNSCALE

UNSCALE unscales a floating-point number between lower and upper limits and converts it into a fixed-point number between 0 and +27 648 (unipolar) or between −27 648 and +27 648 (Fig. 13.27). Example of the application: Conversion of a value with physical units into an analog value for an analog output module.

If the value at IN is greater than the value of the upper limit HI_LIM, the value of the constant K2 is output and an error signaled. If the value at IN is less than the lower limit LO_LIM, the value K1 is output and an error signaled.

13.7 Shift functions

13.7.1 General function description

A shift function shifts the content of a tag bit by bit to the left or right. The shifted out bits are lost in the case of shifting, or are applied again at the other side of the tag in the case of rotating. Fig. 13.28 shows the general representation of a shift function in the various programming languages.

The shift functions for 64-bit wide tags for STL are described in Chapter 10.5.6 “Shift functions in the statement list” on page 430.

13.7.2 Shift to right

Shift to right with LAD and FBD

The SHR shift function shifts the contents of the input tags present at the IN parameter to the right by the number of bit positions specified by the shift number at the N input. If the shift number has the value zero, the input value is copied to the output value. If the shift number is greater than the number of available bit positions, the input value is shifted by the number of the available bit positions.

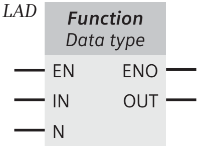
If the input tag has a data type with sign, the bit positions that become free during the shifting are filled with the sign. In all other cases, the bit positions that become free when shifting are padded with zeroes.

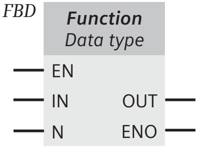
Shift to right with SCL

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. If the shift number has the value zero, the input value is copied to the output value. If the shift number is greater than the number of available bit positions, the input value is shifted by the number of the available bit positions.

The bit positions that become free when shifting are padded with zeroes.

Shift functions





Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	*Data type*	Input tag
N	INPUT	UINT	Quantity
OUT	OUTPUT	*Data type*	Result

SCL

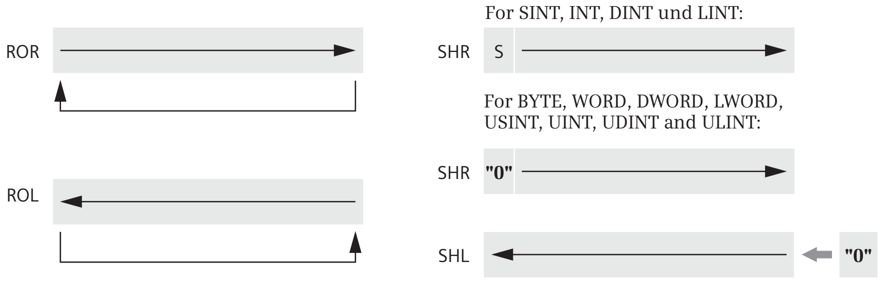
```
OUT := Function_Data type (IN := ... , N := ... );
```

Function: ROR Rotate to right
ROL Rotate to left

Data type: BYTE, WORD, DWORD, LWORD

Function: SHR Shift to right
SHL Shift to left

Data type: BYTE, WORD, DWORD, LWORD,
SINT, INT, DINT, LINT,
USINT, UINT, UDINT, ULINT


ROR
ROL
For SINT, INT, DINT und LINT:
SHR
S
For BYTE, WORD, DWORD, LWORD,
USINT, UINT, UDINT and ULINT:
SHR
"0"
SHL
"0"

STL

```
L   IN                              L   N
Operation with N parameter          L   IN
T   OUT                             Operation without parameter
                                    T   OUT
```

A shift function changes the contents of accumulator 1: depending on the operation, the bits 0 to 15 (word by word) or 0 to 31 (doubleword by doubleword). The parameter for the operation specifies by how many bit positions shifting is carried out. If the operation has no parameters, the number of bit positions to be shifted is taken as a positive fixed-point number from accumulator 2.

Function:	Doubleword by doubleword (bits 0 to 31):	Word by word (bits 0 to 15):
Shift to right	SRD	SRW
Shift with sign	SSD	SSI
Shift to left	SLD	SLW
Rotate to right	RRD	—
Rotate to left	RLD	—
Rotate to right through A1	RRDA	—
Rotate to left through A1	RLDA	—

Fig. 13.28 Shift functions, representation, and principle of operation

Shift to right with STL

The shift number is either specified as a parameter in the shift function or is present as a positive fixed-point number in accumulator 2.

The SRW shift function shifts the contents of bits 0 to 15 of accumulator 1 bit by bit to the right. The bit positions that become free when shifting are padded with zeroes. The left word of accumulator 1 remains unaffected.

The SSI shift function shifts the contents of bits 0 to 15 of accumulator 1 bit by bit to the right. The bit positions that become free when shifting are filled with the sign of the fixed-point number. The left word of accumulator 1 remains unaffected.

The SRD shift function shifts the entire contents of accumulator 1 bit by bit to the right. The bit positions that become free when shifting are padded with zeroes.

The SSD shift function shifts the entire contents of accumulator 1 bit by bit to the right. The bit positions that become free when shifting are filled with the sign of the fixed-point number.

13.7.3 Shift to left

Shift to left with LAD, FBD, and SCL

The SHL shift function shifts the contents of the tag present at the IN input bit by bit to the left by the number of positions specified by the shift number at the N input. If the shift number has the value zero, the input value is copied to the output value. If the shift number is greater than the number of available bit positions, the input value is shifted by the number of the available bit positions.

The bit positions that become free when shifting are padded with zeroes.

Shift to left with STL

The shift number is either specified as a parameter in the shift function or is present as a positive fixed-point number in accumulator 2.

The SLW shift function shifts the contents of bits 0 to 15 of accumulator 1 bit by bit to the left. The bit positions that become free when shifting are padded with zeroes. The left word of accumulator 1 remains unaffected; carrying-over to bit 16 is not carried out.

The SLD shift function shifts the entire contents of accumulator 1 bit by bit to the left. The bit positions that become free when shifting are padded with zeroes.

13.7.4 Rotate to right

Rotate to right with LAD, FBD, and SCL

The ROR 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. If the shift number has the value zero, the input value is copied to the output value.

The bit positions that become free when shifting are filled with the signal state of the shifted-out positions.

Rotate to right with STL

The shift number is either specified as a parameter in the shift function or is present as a positive fixed-point number in accumulator 2.

The RRD function shifts the entire contents of accumulator 1 bit by bit to the right. The bit positions that become free when shifting are filled by the shifted-out bit positions.

If the shift number = 0, the operation is not executed (nil operation NOP); if it is 32, the content of accumulator 1 is retained and status bit CC1 has the signal state of the last shifted-out bit (bit 0). If the shift number = 33, shifting is carried out by one position; with 34, shifting is by two positions, etc.

The RLDA shift function shifts the entire contents of accumulator 1 by one bit to the left. The bit position that becomes free when shifting (bit 0) is filled with the signal state of status bit CC1. Status bit CC1 contains the signal state of the shifted-out bit (bit 31); status bit CC0 is set to “0”.

13.7.5 Rotate to left

Rotate to left with LAD, FBD, and SCL

The ROL function shifts the contents of the tag present at the IN input bit by bit to the left by the number of positions specified by the shift number at the N input. If the shift number has the value zero, the input value is copied to the output value.

The bit positions that become free when shifting are filled with the signal state of the shifted-out positions.

Rotate to left with STL

The shift number is either specified as a parameter in the shift function or is present as a positive fixed-point number in accumulator 2.

The RLD function shifts the entire contents of accumulator 1 bit by bit to the left. The bit positions that become free when shifting are filled by the shifted-out bit positions.

If the shift number = 0, the operation is not executed (nil operation NOP); if it is 32, the content of accumulator 1 is retained and status bit CC1 has the signal state of the last shifted-out bit (bit 0). If the shift number = 33, shifting is carried out by one position; with 34, shifting is by two positions, etc.

The RRDA shift function shifts the entire contents of accumulator 1 by one bit to the right. The bit position that becomes free when shifting (bit 31) is filled with the signal state of status bit CC1. Status bit CC1 contains the signal state of the shifted-out bit (bit 0); status bit CC0 is set to “0”.

13.8 Logic functions

The (digital) logic functions comprise the following functions:

▷ Word logic operations according to AND, OR, and exclusive OR

▷ Invert

▷ Code bit and set bit number (DECO, ENCO)

▷ Selection and limiting functions (SEL, MUX, DEMUX, MIN, MAX, LIMIT).

You can find the logic functions in the program elements catalog under *Basic instructions* > *Word logic operations* and *Basic instructions* > *Math functions* (MIN, MAX and LIMIT) if they are not implemented by a logical expression (SCL) or basic instructions (STL).

13.8.1 Word logic operations

General processing of a word logic operation

A word logic operation links the values of two digital tags bit by bit according to AND, OR, or exclusive OR. FBD and LAD use boxes with EN/ENO for the word logic operation. With SCL, the word logic operation is a logic expression. STL links the contents of accumulators 1 and 2 or the content of accumulator 1 to a constant. There are functions in the *Long Functions* global library for the data type LWORD. Fig. 13.29 shows the general representation of a digital logic operation in the various programming languages.

AND logic operation

The AND logic operation links the individual bits of the input tags according to an AND logic operation. The individual bits only have signal state “1” in the result if the corresponding bits of the two values to be linked have signal state “1”.

A word by word AND logic operation with STL (AW) only uses the right words (bits 0 to 15) of the accumulators. The contents in the left words are not changed.

Since the bits with signal state “0” in the second input tag (“mask”) also set these bits in the result to “0” independent of the assignment of these bits in the first input tag, one also says that these bits are “masked”. This masking is the main application of the (digital) AND logic operation.

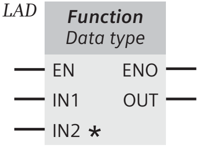
OR logic operation

The OR logic operation links the individual bits of the input tags according to an OR logic operation. The individual bits only have signal state “0” in the result if the corresponding bits of the two values to be linked have signal state “0”.

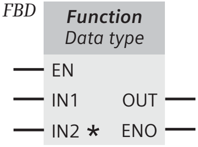
A word by word OR logic operation with STL (OW) only uses the right words (bits 0 to 15) of the accumulators. The contents in the left words are not changed.

Word logic operations AND, OR, and XOR

LAD



FBD



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN1	INPUT	*Data type*	Input tag 1
IN2	INPUT	*Data type*	Input tag 2
OUT	OUTPUT	*Data type*	Result

SCL

```
#var_OUT := #var_IN1 Function #var_IN2;
```

Function:
AND Bit by bit AND operation
OR Bit by bit OR operation
XOR Bit by bit exclusive OR operation

Data type:
BYTE, WORD, DWORD, LWORD

Linking of the individual bits:

Bit of the IN1 parameter	"0"	"0"	**"1"**	**"1"**
Bit of the IN2 parameter	"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"

STL

```
L    #var_IN                     L    #var_Mask
Operation with mask              L    #var_IN
T    #var_OUT                    Operation without parameter
                                 T    #var_OUT
```

A word logic operation changes the contents of accumulator 1: depending on the operation, the bits 0 to 15 (word by word) or 0 to 31 (doubleword by doubleword). The parameter for the operation specifies the mask used for linking the contents of accumulator 1. If the operation has no parameters, the mask is taken from accumulator 2.

Function:	**Doubleword by doubleword (bits 0 to 31):**	**Word by word (bits 0 to 15):**
AND logic operation	AD	AW
OR logic operation	OD	OW
Exclusive OR logic operation	XOD	XOW

```
CALL Function_LWORD
  IN1  := ...
  IN2  := ...
  OUT  := ...
```

Function:
AND Bit by bit AND operation
OR Bit by bit OR operation
XOR Bit by bit exclusive OR operation

Fig. 13.29 Representation and function of word logic operations

Since the bits with signal state “1” in the second input tag (“mask”) also set these bits in the result to “1” independent of the assignment of these bits in the first input tag, one also says that these bits are “unmasked”. This unmasking is the main application of the (digital) OR logic operation.

Exclusive OR logic operation

The exclusive OR logic operation links the individual bits of the input tags according to an exclusive OR logic operation. The individual bits only have signal state “1” in the result if only one of the corresponding bits of the two values to be linked has signal state “1”. If a bit in the second input tag has signal state “1”, the inverted signal state of the bit of the first input tag is present at this position in the result.

A word by word exclusive OR logic operation with STL (XOW) only uses the right words (bits 0 to 15) of the accumulators. The contents in the left words are not changed.

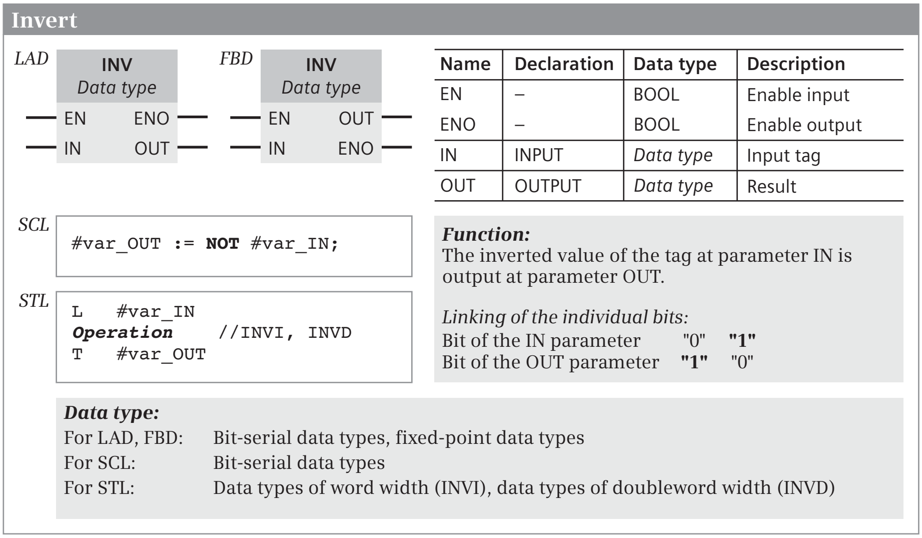
Only those bits have signal state “1” in the result which have different signal states in both tags prior to the digital exclusive OR logic operation. Detection of the bits with different signal states or the “negating” of the signal states of individual bits are the main applications of the (digital) exclusive OR logic operation.

13.8.2 Invert, generate one's complement

Inverting negates the value of a tag bit by bit; signal state “1” becomes signal state “0” and vice versa. Fig. 13.30 shows the representation of the function in the various programming languages.

For SCL, the one's complement is formed with the operator NOT (Boolean negation), which inverts the bits of the following operands or the result of the following expression.

For STL, the INVI operation inverts the content of the right word in accumulator 1 (bits 0 to 15). The left word is not affected. The INVD operation inverts the content of the complete accumulator 1 (bits 0 to 31).



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	*Data type*	Input tag
OUT	OUTPUT	*Data type*	Result

```
#var_OUT := NOT #var_IN;
```

```
L   #var_IN
Operation    //INVI, INVD
T   #var_OUT
```

Function:
The inverted value of the tag at parameter IN is output at parameter OUT.

Linking of the individual bits:

Bit of the IN parameter	"0"	**"1"**
Bit of the OUT parameter	**"1"**	"0"

Data type:

For LAD, FBD:	Bit-serial data types, fixed-point data types
For SCL:	Bit-serial data types
For STL:	Data types of word width (INVI), data types of doubleword width (INVD)

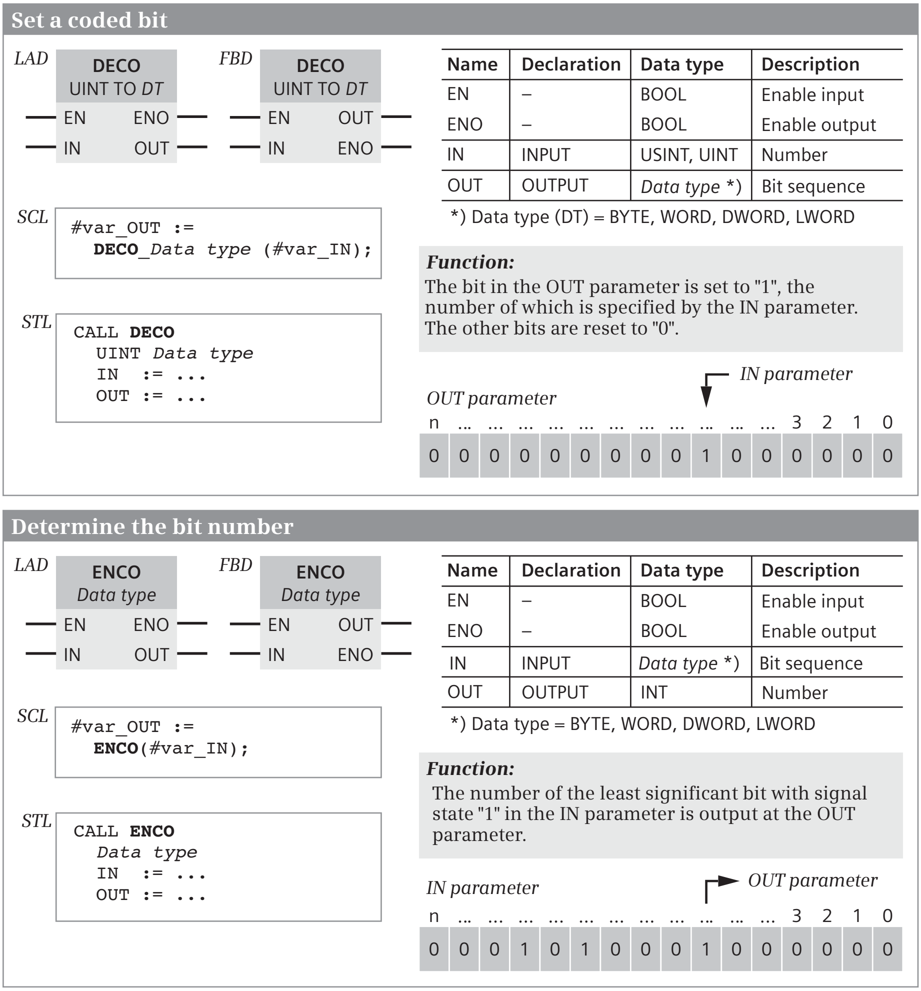
Fig. 13.30 Invert, generate one's complement

13.8.3 Coding functions DECO and ENCO

DECO converts a binary number to a bit pattern. ENCO converts a bit pattern to a binary number. For LAD and FBD, these functions are represented as EN/ENO box. For SCL, they are functions with a function value and for STL, they are block calls. Fig. 13.31 shows the representation of the function in the various programming languages.

DECO Code bit

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



```
#var_OUT :=
  DECO_Data type (#var_IN);
```

```
CALL DECO
  UINT Data type
  IN  := ...
  OUT := ...
```

Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	USINT, UINT	Number
OUT	OUTPUT	*Data type* *)	Bit sequence

*) Data type (DT) = BYTE, WORD, DWORD, LWORD

Function:
The bit in the OUT parameter is set to "1", the number of which is specified by the IN parameter. The other bits are reset to "0".

n	...	...	...	...	...	...	...	...	...	...	...	3	2	1	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

```
#var_OUT :=
  ENCO(#var_IN);
```

```
CALL ENCO
  Data type
  IN  := ...
  OUT := ...
```

Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	*Data type* *)	Bit sequence
OUT	OUTPUT	INT	Number

*) Data type = BYTE, WORD, DWORD, LWORD

Function:
The number of the least significant bit with signal state "1" in the IN parameter is output at the OUT parameter.

n	...	...	...	...	...	...	...	...	...	...	...	3	2	1	0
0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0

Fig. 13.31 Representation and description of the DECO and ENCO functions

type at parameter OUT, only a subset of the bits is selected in parameter IN: 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);
```

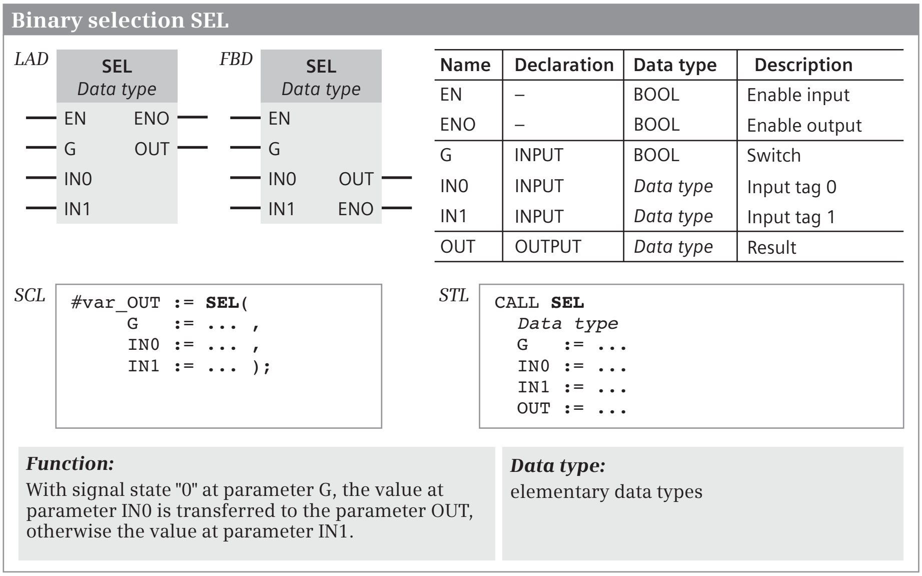
ENCO Set bit number

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.

13.8.4 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. 13.32).



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
G	INPUT	BOOL	Switch
IN0	INPUT	*Data type*	Input tag 0
IN1	INPUT	*Data type*	Input tag 1
OUT	OUTPUT	*Data type*	Result

SCL

```
#var_OUT := SEL(
      G   := ... ,
      IN0 := ... ,
      IN1 := ... );
```

STL

```
CALL SEL
  Data type
  G   := ...
  IN0 := ...
  IN1 := ...
  OUT := ...
```

Function:
With signal state "0" at parameter G, the value at parameter IN0 is transferred to the parameter OUT, otherwise the value at parameter IN1.

Data type:
elementary data types

Fig. 13.32 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 IN*n* 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 IN*n*) 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.

If the value of K is outside the range of input parameters IN0 to IN*n*, 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. 13.33).

Dependent on the value of the parameter K, DEMUX issues the tag at the input (parameter IN) to a parameter OUT0 to OUT*n*, 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 OUT*n*) 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 of output parameters OUT0 to OUT*n*, 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. 13.33).

13.8.5 Minimum selection MIN, maximum selection MAX

The minimum selection **MIN** transfers the smallest of the values present at the parameters to parameter OUT. For LAD and FBD, up to 100 inputs can be configured. For SCL, up to 32 inputs can be configured. STL provides 3 inputs. 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. 13.34).

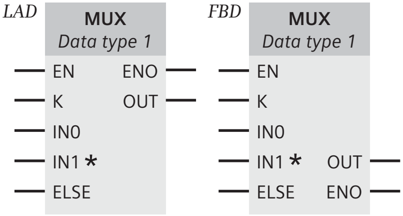
The maximum selection **MAX** transfers the largest of the values present at the parameters to parameter OUT. For LAD and FBD, up to 100 inputs can be configured. For SCL, up to 32 inputs can be configured. STL provides 3 inputs. If there is an invalid REAL number at the input parameters, the function is not executed and the enable output ENO is set to signal state “0”.

13.8.6 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. 13.35).

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.

Multiplexing MUX



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
K	INPUT	UINT	Selection
IN0	INPUT	*Data type 1*	Input tag 0
IN1	INPUT	*Data type 1*	Input tag 1
ELSE	INPUT	*Data type 1*	Substitute value
OUT	OUTPUT	*Data type 1*	Result

SCL

```
#var_OUT := MUX_Data type1(
  K      := ... ,
  IN1    := ... ,
  IN2    := ... ,
  ...
  INELSE := ... );
```

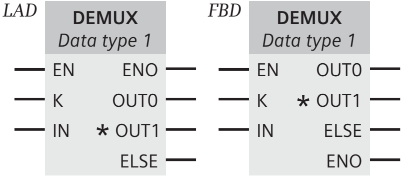
STL

```
CALL MUX
  DT1  DT2
  K        := ...
  IN1      := ...
  IN2      := ...
  ...
  INELSE  := ...
  RET_VAL := ...
```

Function:
Depending on the assignment of parameter K, MUX transfers the value of an input parameter IN*n* to the output parameter OUT. If the value at K is greater than the number of inputs, the value at the ELSE or INELSE parameter is transferred to the output.

Data type:
Data type 1 (DT1): elementary data types
Data type 2 (DT2): fixed-point data types

Demultiplexing DEMUX



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
K	INPUT	UINT	Selection
IN	INPUT	*Data type 1*	Input tag
OUT0	OUTPUT	*Data type 1*	Output tag 0
OUT1	OUTPUT	*Data type 1*	Output tag 1
ELSE	OUTPUT	*Data type 1*	Substitute output

SCL

```
DEMUX(K       := ... ,
      IN      := ... ,
      OUT0    => ... ,
      OUT1    => ... ,
      ...
      OUTELSE => ... );
```

STL

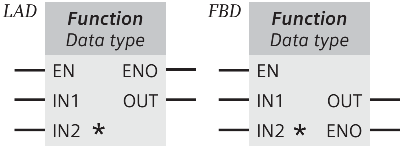
```
CALL DEMUX
  DT1  DT2
  K        := ...
  IN       := ...
  OUT0     := ...
  OUT1     := ...
  ...
  OUTELSE  := ...
```

Function:
Depending on the assignment of parameter K, DEMUX transfers the value of the input parameter IN to an output parameter OUT*n*. If the value at K is greater than the number of outputs, the value at the IN parameter is transferred to the ELSE or OUTELSE output.

Data type:
Data type 1 (DT1): elementary data types
Data type 2 (DT2): fixed-point data types

Fig. 13.33 Multiplexing MUX, representation and function

Minimum selection MIN, maximum selection MAX



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN1	INPUT	*Data type*	Input tag 1
IN2	INPUT	*Data type*	Input tag 2
OUT	OUTPUT	*Data type*	Result

SCL

```
#var_OUT := Function_Data type(
     IN1 := ... ,
     IN2 := ... ,
     ...        );
```

STL

```
CALL Function
  Data type
  IN1 := ...
  IN2 := ...
  IN3 := ...
  OUT := ...
```

Function:

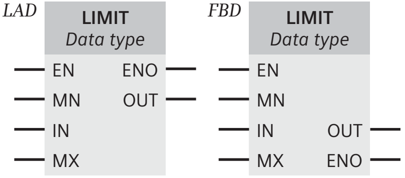
MIN Select minimum
The lowest input value is copied to the OUT output.

MAX Select maximum
The highest input value is copied to the OUT output.

Data type:
Fixed-point and floating-point data types

Fig. 13.34 Minimum and maximum selection, representation and function

Limiter LIMIT



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
MN	INPUT	*Data type*	Lower limit
IN	INPUT	*Data type*	Input tag
MX	INPUT	*Data type*	Upper limit
OUT	OUTPUT	*Data type*	Result

SCL

```
#var_OUT := LIMIT_Data type(
      MN  := ... ,
      IN  := ... ,
      MX  := ... );
```

STL

```
CALL LIMIT
  Data type
  MN  := ...
  IN  := ...
  MX  := ...
  OUT := ...
```

Function:
The value of the tag present at the IN parameter is restricted to the limits MN and MX and output at the OUT parameter.

Data type:
Fixed-point and floating-point data types

Fig. 13.35 Limiter LIMIT, representation and function

13.9 Processing of strings (data type STRING)

A string can be processed with the following functions:

▷ LEN	Outputs the current length of a string
▷ MAX_LEN	Outputs the set maximum 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

All functions for processing strings expect a valid string with plausible values in the length bytes (maximum length ≤ 254, current length ≤ maximum length) at the parameters with data type STRING. If you do not assign default values to strings when declaring, they are automatically assigned as empty strings (current length = 0) with the maximum length (= 254).

Please note that strings which you declare in the temporary local data cannot be assigned default values. For blocks with standard access (the attribute *Optimized block access* is deactivated), the individual bytes of a string have a quasi-random value. In this case you must assign a defined value (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. For blocks with the attribute *Optimized block access* activated, for STRING tags which are declared in the temporary local data, the lengths are assigned plausible values and the characters are assigned '$00'.

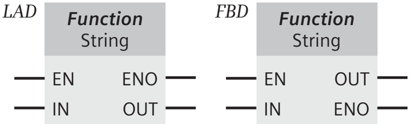
13.9.1 Output current length of a string LEN

The LEN function outputs the current length of a string (set in the first byte) present at the IN parameter at the OUT parameter. For an “empty” string, the current length is zero. The maximum value of the current length is the maximum length set in the second byte. LEN returns an error on incorrect parameter assignment (Fig. 13.36).

13.9.2 Output maximum length of a string MAX_LEN

The MAX_LEN function outputs the maximum length of a string (set in the second byte) present at the IN parameter at the OUT parameter. The maximum value of the maximum length of a string is 254 characters. MAX_LEN returns an error on incorrect parameter assignment (Fig. 13.36).

Output length of a string LEN, MAX_LEN



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	STRING	String
OUT	OUTPUT	INT	Length

SCL

```
#var_OUT := Function(
       IN := ...);
```

STL

```
CALL Function
  String
  IN   := ...
  OUT  := ...
```

Function:

LEN outputs the current length (number of characters) of the string present at the IN parameter at the OUT parameter.

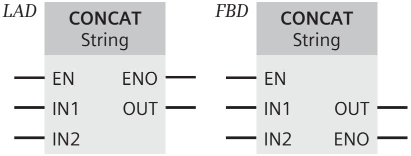
MAX_LEN outputs the maximum length (number of characters) of the string present at the IN parameter at the OUT parameter.

Fig. 13.36 Output length of a string, function and representation

13.9.3 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. 13.37).

Combination of two strings CONCAT



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN1	INPUT	STRING	String 1
IN2	INPUT	STRING	String 2
OUT	OUTPUT	STRING	Result

SCL

```
var_OUT := CONCAT(
    IN1 := ... ,
    IN2 := ... );
```

STL

```
CALL CONCAT
  String
  IN1 := ...
  IN2 := ...
  OUT := ...
```

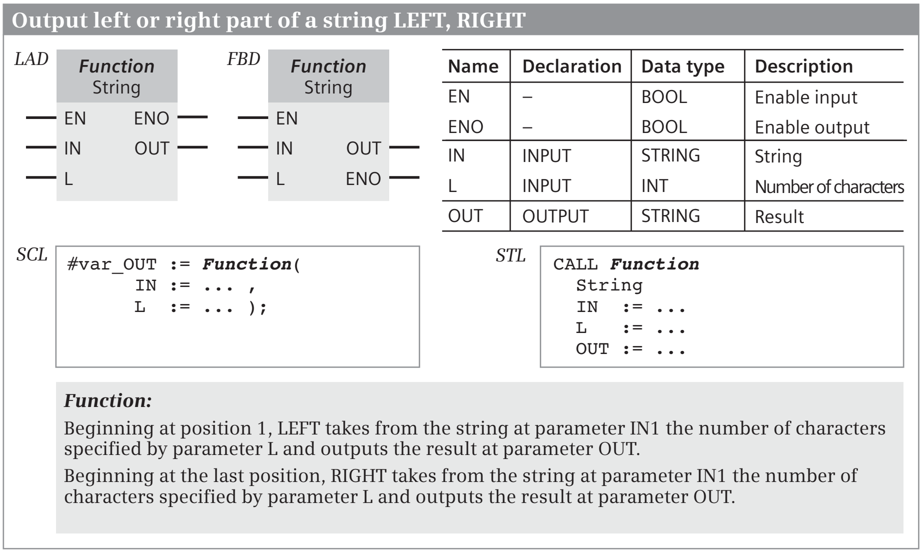
Function:

CONCAT adds the string at parameter IN2 to the string at parameter IN1 and outputs the result at the OUT parameter.

Fig. 13.37 Combining two strings CONCAT, function and representation

13.9.4 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. 13.38).



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	STRING	String
L	INPUT	INT	Number of characters
OUT	OUTPUT	STRING	Result

```
#var_OUT := Function(
      IN := ... ,
      L  := ... );
```

```
CALL Function
  String
  IN  := ...
  L   := ...
  OUT := ...
```

Function:

Beginning at position 1, LEFT takes from the string at parameter IN1 the number of characters specified by parameter L and outputs the result at parameter OUT.

Beginning at the last position, RIGHT takes from the string at parameter IN1 the number of characters specified by parameter L and outputs the result at parameter OUT.

Fig. 13.38 Output left or right part of a string LEFT and RIGHT

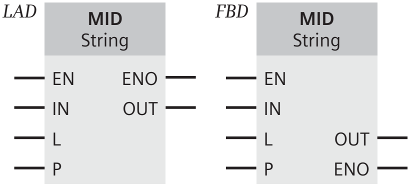
13.9.5 Output right part of string RIGHT

The function RIGHT returns the last 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. 13.38).

13.9.6 Output middle part of string MID

The MID function takes the middle part of the string present at the parameter IN and outputs it at the parameter OUT. The middle part begins at the position specified by parameter P and is as many characters long as the parameter L specifies (Fig. 13.39).

Output the middle part of a string MID



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	STRING	Input tag
L	INPUT	INT	Number of characters
P	INPUT	INT	Character position
OUT	OUTPUT	STRING	Output tag

SCL

```
#var_OUT := MID(
      IN := ... ,
      L  := ... ,
      P  := ... );
```

STL

```
CALL MID
  String
  IN  := ...
  L   := ...
  P   := ...
  OUT := ...
```

Function:
Beginning at position P, MID takes from the string at parameter IN1 the number of characters specified by parameter L and outputs the result at parameter OUT.

Fig. 13.39 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”.

13.9.7 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. 13.40).

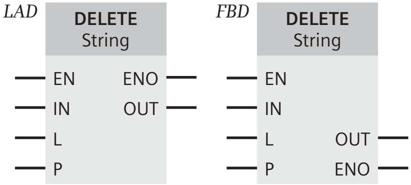
If L is equal to zero, the input string is output. If P is greater than the current length of the input tag, the input tag is output and ENO is set to “0”. If the sum of P and L is greater than the current length of the input tag, the string is deleted up to the end. If L is negative or P is zero or negative, a blank string is output and ENO is set to “0”.

13.9.8 Insert string INSERT

The INSERT function inserts the string at parameter IN2 into the string at parameter IN1 and outputs the result at the OUT parameter. Parameter P specifies the position from which the insertion is to take place (Fig. 13.41).

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 to

Delete part of a string DELETE



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN	INPUT	STRING	Input tag
L	INPUT	INT	Number of characters
P	INPUT	INT	Character position
OUT	OUTPUT	STRING	Output tag

SCL

```
#var_OUT := DELETE(
      IN  := ... ,
      L   := ... ,
      P   := ... );
```

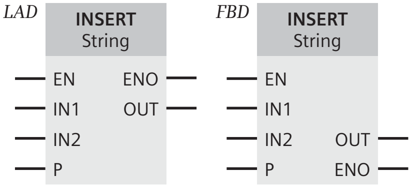
STL

```
CALL DELETE
  String
  IN   := ...
  L    := ...
  P    := ...
  OUT  := ...
```

Function:
Beginning at position P, DELETE deletes from the string at parameter IN the number of characters specified by parameter L and outputs the result at parameter OUT.

Fig. 13.40 Delete part of string DELETE, function and representation

Inserting a string into another string INSERT



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN1	INPUT	STRING	Input tag
IN2	INPUT	STRING	String to be inserted
P	INPUT	INT	Character position
OUT	OUTPUT	STRING	Output tag

SCL

```
#var_OUT := INSERT(
     IN1 := ... ,
     IN2 := ... ,
     P   := ... );
```

STL

```
CALL INSERT
  String
  IN1 := ...
  IN2 := ...
  P   := ...
  OUT := ...
```

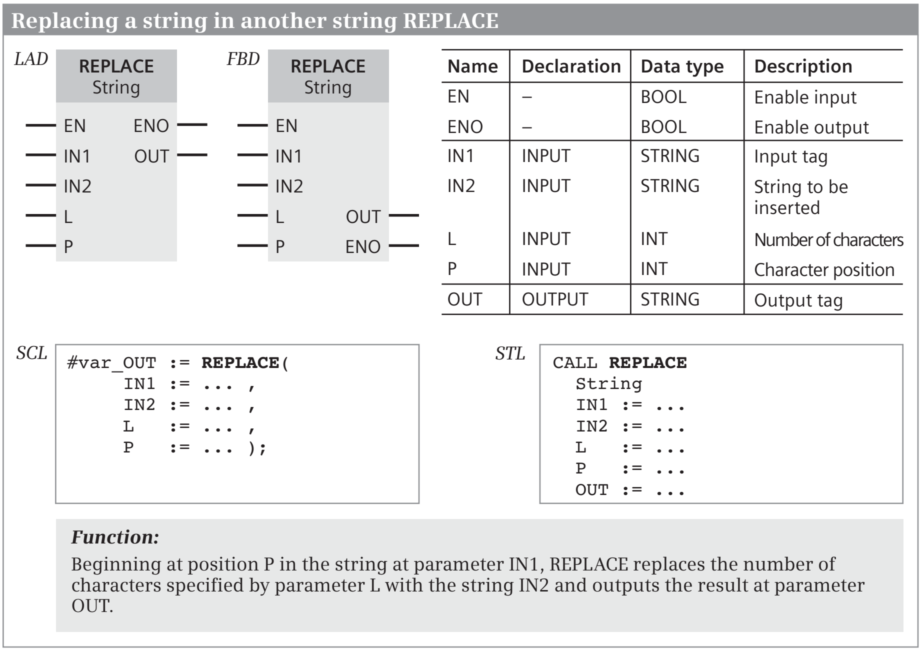
Function:
INSERT inserts the string IN2 in the string at parameter IN1 beginning at position P and outputs the result at the OUT parameter.

Fig. 13.41 Insert string INSERT, function and representation

“0”. If the new string is longer than permitted by the maximum length of the output string, the characters are entered up to the permitted length and ENO is set to “0”.

13.9.9 Replace part of string REPLACE

The REPLACE function replaces characters present in the string at parameter IN1 by the string at parameter IN2 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. 13.42).



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN1	INPUT	STRING	Input tag
IN2	INPUT	STRING	String to be inserted
L	INPUT	INT	Number of characters
P	INPUT	INT	Character position
OUT	OUTPUT	STRING	Output tag

SCL

```
#var_OUT := REPLACE(
     IN1 := ... ,
     IN2 := ... ,
     L   := ... ,
     P   := ... );
```

STL

```
CALL REPLACE
  String
  IN1 := ...
  IN2 := ...
  L   := ...
  P   := ...
  OUT := ...
```

Function:

Beginning at position P in the string at parameter IN1, REPLACE replaces the number of characters specified by parameter L with the string IN2 and outputs the result at parameter OUT.

Fig. 13.42 Replace part of string REPLACE, function and representation

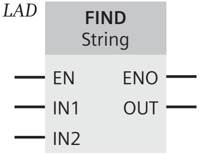
If L = 0, then the string IN2 is inserted into the string IN1 from position P, without deleting characters in IN1. If P = 1, the first L characters of IN1 are replaced by IN2. If P is greater than the current length of IN1, IN2 is appended to IN1 and ENO is set to “0”. If L is negative or P is zero or negative, a blank 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”.

13.9.10 Find part of string FIND

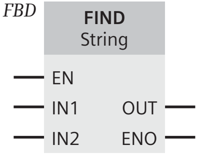
The FIND function determines the position of the string at parameter IN2 in the string at parameter IN1 and outputs it at the OUT parameter. The position of the first character is output if a match has been found. If IN2 is not contained in IN1, zero is returned (Fig. 13.43).

Finding a string in another string FIND

LAD



FBD



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
ENO	–	BOOL	Enable output
IN1	INPUT	STRING	Input tag
IN2	INPUT	STRING	String to be found
OUT	OUTPUT	INT	Output tag

SCL

```
#var_OUT := FIND(
     IN1 := ... ,
     IN2 := ... );
```

STL

```
CALL FIND
  String
  IN1 := ...
  IN2 := ...
  OUT := ...
```

Function:

FIND searches for the string IN2 in the string at parameter IN1 and outputs its position at the OUT parameter.

Fig. 13.43 Find part of string FIND, function and representation

14 Program 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 287, 8 “Function block diagram FBD” on page 323, 10 “Statement list STL” on page 395, and 9 “Structured Control Language SCL” on page 359 describe how you can program the functions using the individual programming languages and what special features exist.

You can use jump functions to exit linear user program execution and continue at a different point in the block. Jump functions can be executed absolutely or dependent upon the result of logic operation. The control statements which are only present with SCL for controlling program execution are described in Chapter 9.6.3 “Control statements” on page 383. Chapter 10.7.1 “Working with status bits” on page 442 describes how to scan the status bits using jump functions in the STL programming language.

If a function block (FB) or a function (FC) is to be processed, the block must be “called”. Depending on the program in the block, a list of block parameters can also be provided when the block is called. The program in the block will then work with this list. The graphical programming languages LAD and FBD represent the call function as a box with inputs and outputs. In the text-based programming languages, the call function mostly consists of the block name or instance name, followed by the list of block parameters.

The program in the block does not have to be concluded with a specific statement. Depending on the result of the logic operation, however, it is possible to prematurely terminate program execution in the block. The block end function can also receive an error message, which can be evaluated in the calling block at the enable output ENO. A block call that is dependent on the result of logic operation can be formed using an enable input EN.

This chapter also describes the access to data blocks which is possible in addition to “normal” addressing of data tags in the work memory: How the attributes of a data block are read and how data blocks in the load memory can be accessed. ARRAY data blocks have the structure of the ARRAY data type. Data tags in these data blocks are addressed with special statements. This chapter also describes what CPU data blocks are and how they are created and deleted during runtime.

14.1 Jump functions

14.1.1 Introduction

Jump functions can interrupt linear execution of the program and continue at a different position in the block. You identify this position by means of a jump label which you specify in the jump statement as the jump destination. For a CPU 1500, you can use up to 256 jump labels in one block.

The jump function and jump destination must be in the same block. The jump destination or jump label must be unique within a block. It is permissible to jump to a jump destination from more than one position. Both forward and backward jumps are possible with regard to the direction of program execution.

Table 14.1 shows an overview of the types of jump functions.

Table 14.1 Types of jump functions

Jump function	Present with			
	LAD	FBD	SCL	STL
Absolute jump	X	X	X	X
Jump depending on RLO	X	X	–	X
Jump depending on status bits	–	–	–	1)
Jump list	X	X	2)	X
Jump distributor	X	X	2)	–
Loop jump	–	–	2)	X

1) See Chapter 10.7.1 "Working with status bits" on page 442
2) See Chapter 9.6.3 "Control statements" on page 383

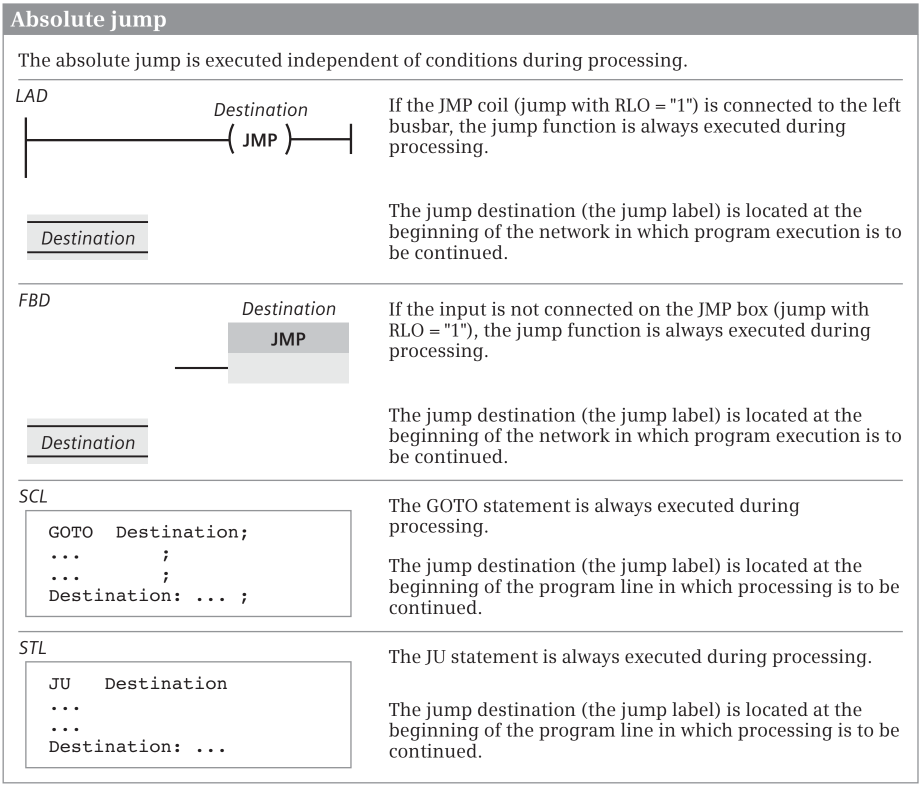
14.1.2 Absolute jump

An absolute jump is carried out independent of conditions. When processing the jump function, program execution is continued at the specified jump label. Fig. 14.1 shows the implementation of the jump function in the various programming languages.

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 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.



The absolute jump is executed independent of conditions during processing.

If the JMP coil (jump with RLO = "1") is connected to the left busbar, the jump function is always executed during processing.

The jump destination (the jump label) is located at the beginning of the network in which program execution is to be continued.

If the input is not connected on the JMP box (jump with RLO = "1"), the jump function is always executed during processing.

The jump destination (the jump label) is located at the beginning of the network in which program execution is to be continued.

```
GOTO  Destination;
...        ;
...        ;
Destination: ... ;
```

The GOTO statement is always executed during processing.

The jump destination (the jump label) is located at the beginning of the program line in which processing is to be continued.

```
JU   Destination
...
...
Destination: ...
```

The JU statement is always executed during processing.

The jump destination (the jump label) is located at the beginning of the program line in which processing is to be continued.

Fig. 14.1 Absolute jump independent of conditions

Absolute jump GOTO (SCL)

The jump function GOTO exits linear program execution and continues it at a different position in the block. If statements form a defined block, e.g. a program call within a program loop,

- ▷ the jump destination must be within this statement block if the GOTO statement is also within the statement block,
- ▷ one 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"
```

Absolute jump JU (STL)

The jump function JU is always carried out, i.e. independent of any conditions. JU interrupts linear execution of the program and continues it at the position identified by the jump label. The jump function JU does not influence the status bits.

If scan statements are present directly in front of the jump function and also at the jump destination, these are handled like a single logic operation.

A jump label must always be followed by a statement. This can also be a null operation, e.g. NOP 0:

```
Label: NOP 0     //Entry with null operation
```

14.1.3 Conditional jump functions

A conditional jump is executed depending on the result of logic operation. Depending on the jump function, program execution is continued at the specified jump label with RLO = “1” or with RLO = “0”. Fig. 14.2 shows the implementation of the conditional jump function in the various programming languages.

With SCL, the dependency of the jump statement GOTO on the RLO can be emulated, for example, by an IF statement:

```
IF (* Condition *) THEN GOTO Destination; END_IF;
```

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 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 JC and JCN (STL)

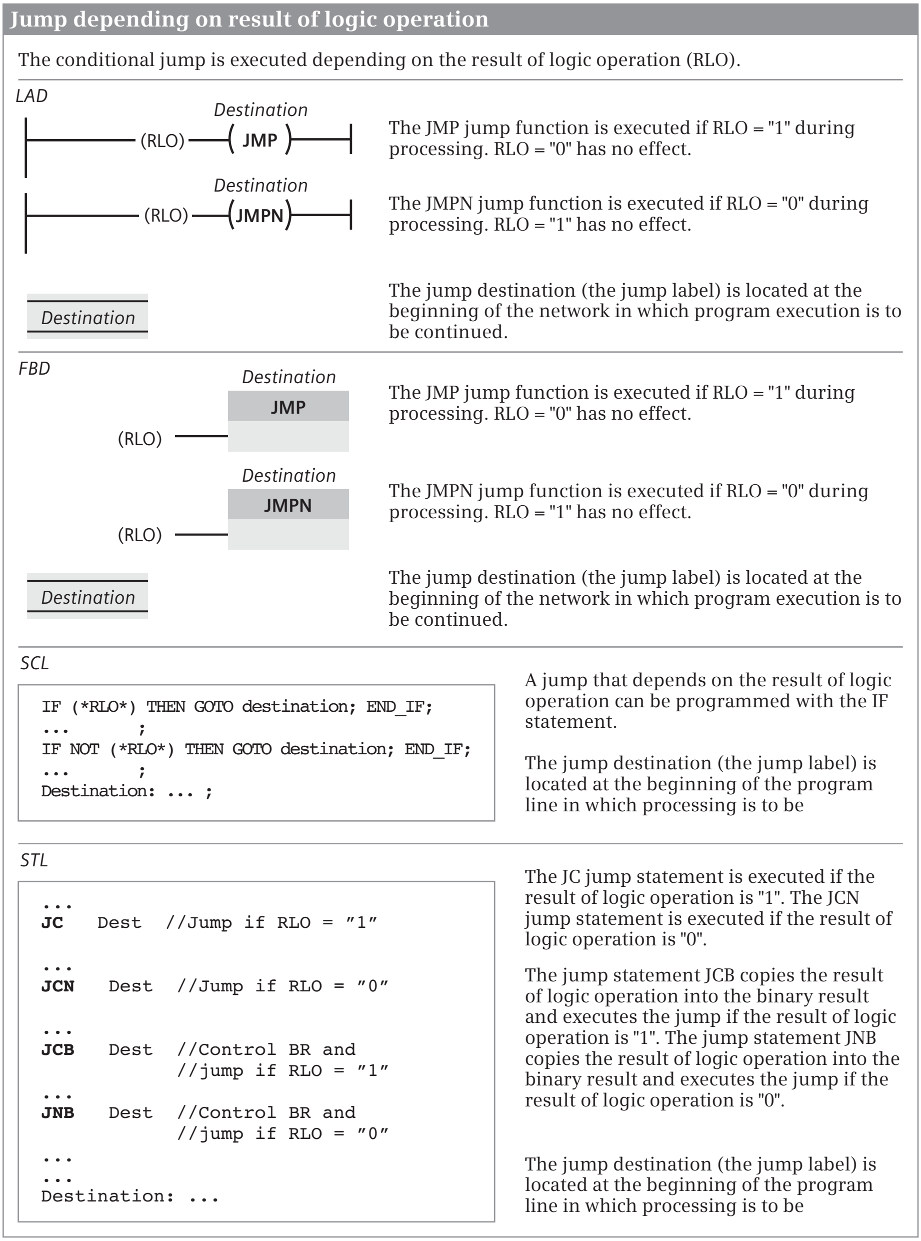
The jump function JC is only executed if the result of logic operation is “1” when this function is processed. The jump is not performed if it is “0” and execution of the program is continued with the following statement.

The jump function JCN is only executed if the result of logic operation is “0” when this function is processed. The jump is not performed if it is “1” and execution of the program is continued with the following statement.

JC and JCN always set the result of logic operation to “1” – even if the condition is not fulfilled. If the statements directly following these jump functions contain operations dependent on the result of logic operation, they are executed if the jump is not carried out. If this jump function is directly followed by scan statements, these scans are handled as first input bit scans, i.e. a new logic operation then starts.

Conditional jump functions JCB and JNB (STL)

The jump function JCB is only executed if the result of logic operation is “1” when this function is processed. The jump is not performed if it is “0” and execution of the program is continued with the following statement.



Jump depending on result of logic operation

The conditional jump is executed depending on the result of logic operation (RLO).

LAD

The JMP jump function is executed if RLO = "1" during processing. RLO = "0" has no effect.

The JMPN jump function is executed if RLO = "0" during processing. RLO = "1" has no effect.

The jump destination (the jump label) is located at the beginning of the network in which program execution is to be continued.

FBD

The JMP jump function is executed if RLO = "1" during processing. RLO = "0" has no effect.

The JMPN jump function is executed if RLO = "0" during processing. RLO = "1" has no effect.

The jump destination (the jump label) is located at the beginning of the network in which program execution is to be continued.

SCL

```
IF (*RLO*) THEN GOTO destination; END_IF;
...      ;
IF NOT (*RLO*) THEN GOTO destination; END_IF;
...      ;
Destination: ... ;
```

A jump that depends on the result of logic operation can be programmed with the IF statement.

The jump destination (the jump label) is located at the beginning of the program line in which processing is to be

STL

```
...
JC   Dest  //Jump if RLO = "1"

...
JCN   Dest  //Jump if RLO = "0"

...
JCB   Dest  //Control BR and
            //jump if RLO = "1"
...
JNB   Dest  //Control BR and
            //jump if RLO = "0"
...
...
Destination: ...
```

The JC jump statement is executed if the result of logic operation is "1". The JCN jump statement is executed if the result of logic operation is "0".

The jump statement JCB copies the result of logic operation into the binary result and executes the jump if the result of logic operation is "1". The jump statement JNB copies the result of logic operation into the binary result and executes the jump if the result of logic operation is "0".

The jump destination (the jump label) is located at the beginning of the program line in which processing is to be

Fig. 14.2 Conditional jump depending on result of logic operation

The jump function JNB is only executed if the result of logic operation is “0” when this function is processed. The jump is not performed if it is “1” and execution of the program is continued with the following statement.

At the same time, JCB and JNB transfer the result of logic operation to the binary result – even if the condition is not fulfilled. JCB and JNB then always set the result of logic operation to “1” – even if the condition is not fulfilled. If the statements directly following this jump function contain operations dependent on the result of logic operation, they are executed if the jump is not carried out. If this jump function is directly followed by scan statements, these scans are handled as first input bit scans, i.e. a new logic operation then starts.

14.1.4 Jump list

The jump list JMP_LIST (LAD and FBD) or JL (STL) allows jumping to a program section in the block depending on a numerical value (Fig. 14.3). In SCL, the CASE statement can be used for this functionality. For more information on the CASE statement, refer to Chapter 9.6.3 “Control statements” on page 383.

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, execution 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, program execution 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 STL

The JL operation works together with a list of JU jump functions. This list directly follows JL and can have a maximum of 255 entries. With JL there is a jump label which points to the end of the list (to the first statement following the list). You program a jump list in accordance with the general schema shown in Fig. 14.3.

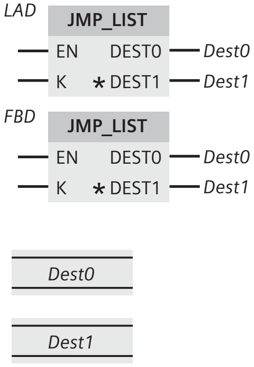
The number of the jump to be executed is present in the right byte of accumulator 1. If 0 is present in accumulator 1, the first jump statement is executed; if 1 is present, the second jump statement is executed, etc. If the number is greater than the length of the list, JL branches to the end of the list (to the statement located after the last jump).

JL is independent of conditions and does not change the status bits.

Only JU statements may be present in the list without gaps. You can assign any names to the jump labels within the context of the general specifications.

Jump depending on a numerical value

The jump list is used to define jump functions which are executed depending on a numerical value.



Name	Declaration	Data type	Description
EN	–	BOOL	Enable input
K	INPUT	UINT	Selection
DEST0	–	–	Jump label 0
DEST1	–	–	Jump label 1

Only jump labels can be present at the output parameters.

The jump destinations (jump labels) are located in each case at the beginning of the network in which the processing of the program is to be continued.

SCL

```
CASE K OF
0    : GOTO Dest0;
1    : GOTO Dest1;
...              ;
ELSE       : GOTO ...  ;
END_CASE;
...        ;
Dest0: ... ;
...        ;
Dest1: ... ;
```

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.

STL

```
L   #Jump_number
JL  End
JU  Dest0
JU  Dest1
JU ...
...
End: ...    //Further program
...
Dest0: ...
...
Dest1: ...
...
```

The value in the tag #Jump_number determines the JU statement to be performed.

Only absolute jump functions JU may be present in the jump list according to JL.

The jump destinations (the jump labels) are located at the beginning of the program line in which processing is to be continued.

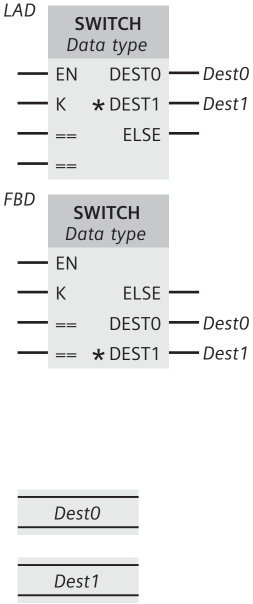
Fig. 14.3 Jump depending on a numerical value

14.1.5 Jump distributor

In LAD and FBD, the jump distributor SWITCH allows jumping to a program section in the block depending on a comparison with a numerical value (Fig. 14.4). In SCL, the IF statement can be used for this functionality. For more information on the IF statement, refer to Chapter 9.6.3 “Control statements” on page 383. STL can map these jump distributors with comparison and jump functions.

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.



Name	Declaration	Data type	Description
EN	–	BOOL	Enable 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 list

Only jump labels can be present at the output parameters.

Data type:
BYTE, WORD, DWORD, LWORD,
USINT, UINT, UDINT, ULINT,
SINT, INT, DINT, LINT,
REAL, LREAL, TIME, LTIME, DATE, LDT
TIME_OF_DAY, LTIME_OF_DAY

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

```
IF K <Comparison> Value0 THEN GOTO Dest0; END_IF;
IF K <Comparison> Value1 THEN GOTO Dest1; END_IF;
...        ;
GOTO ...   ;   //ELSE branch
...        ;
...        ;
Dest0: ... ;
...        ;
Dest1: ... ;
```

A jump that depends on a comparison with a numerical value can be programmed with IF statements.

The jump destinations (the jump labels) are located at the beginning of the program line in which processing is to be continued.

STL

```
L      K
L      #Value0
<Comparison>
JC    Dest0
TAK              //Statement sequence
L      #Value1   //for each
<Comparison>     //additional
JC    Dest1      //Comparison
...
JU ...           //ELSE branch
...
Dest0: ...
...
Dest1: ...
```

A jump that depends on a comparison with a numerical value can be programmed with comparison functions.

The jump destinations (the jump labels) are located at the beginning of the program line in which processing is to be continued.

Fig. 14.4 Jump depending on a comparison result

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 list 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”.

If the first comparison is fulfilled, execution 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.

14.1.6 Loop jump

The loop jump LOOP in the programming language STL permits simplified programming of loops (Fig. 14.5). In LAD and FBD, this functionality can be programmed with simple statements (see example in Chapter 7.6.1 “Jump functions in the ladder logic” on page 316 and 8.6.1 “Jump functions in the function block diagram” on page 353). For program loops, SCL uses the statements FOR, WHILE and REPEAT, which are described in Chapter 9.6.3 “Control statements” on page 383.

Loop jump LOOP

The loop jump LOOP simplifies the programming of program loops with STL.

STL

```
      L      #Number
Next: T      #Counter
...                     //Statement sequence
...                     //in the
...                     //program loop
      L      #Counter
      LOOP   Next
...                     //Further program
```

The #Number tag contains the total number of executed loops. The #Number tag contains the number of loops still to be executed.

LOOP reduces the content of accumulator 1 by one unit and carries out the jump if a value of zero has not yet been reached.

Fig. 14.5 Loop jump LOOP

Loop jump LOOP with STL

LOOP interprets the right word of accumulator 1 as an unsigned 16-bit fixed-point number in the range of 0 to 65535. During processing, LOOP initially decrements the contents of accumulator 1 by 1. If the value is then not zero, the jump to the specified jump label is executed.

If the value is equal to zero following decrementing, no jump is carried out, and the directly following statement is processed.

The value in accumulator 1 thus corresponds to the number of program loops to be executed. You must save this number in a loop counter. You can use any digital tag as a loop counter.

The loop jump does not change the status bits.

14.2 Calling of code blocks

14.2.1 General information on block calls

If a function block (FB) or a function (FC) is to be processed, it must be “called”. You can give data to the called block for processing and import data from the called block. This data transfer is done using block parameters. The call function contains the block name (the name of the call instance for function blocks) and the list with the block parameters.

You call a block by using the mouse to “drag” it from the project tree under the *Program blocks* folder to the working area, i.e. the block that is to be called must already exist. The programming language in which the block is written plays no role in this case. For LAD and FBD, the program editor also adds the enable input EN and the enable output ENO to the call box. With SCL, you can add the enable input EN to a function block as the first entry of the parameter list. You can add the enable output ENO as the last entry of the parameter list for a function block and for a function. With STL, you must map the functionality of the enable input and the enable output as needed. Further details can be found in Chapters 7.6.4 “EN/ENO mechanism in the ladder logic” on page 320, 8.6.4 “EN/ENO mechanism in the function block diagram” on page 356, 9.6.2 “EN/ENO mechanism with SCL” on page 381, and 10.7.2 “EN/ENO mechanism in the statement list” on page 447.

Following processing of the call function, program execution is continued in the called block. The block is processed up to a block end function or up to its end. Program execution then returns to the calling block and continues processing of this block after the call function.

An organization block (OB) 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.

Chapter 5.2 “Creating a user program” on page 149 describes the available blocks and block parameters, what has to be observed with a call (for example the nesting depth), and how the blocks and block parameters are programmed.

14.2.2 Calling a function FC

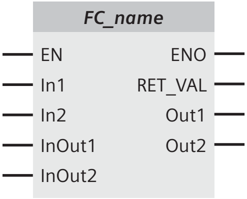
A function FC is a block which does not save any of its own data. All block parameters must be supplied with actual parameters when called. Fig. 14.6 shows the block call for a function.

Calling a function FC

A **function FC** is a block which does not save any of its own data. It can have block parameters, all of which must be supplied with actual parameters when called.

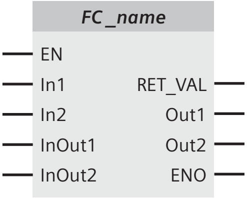
A function FC has a function value (return value) with the preset name RET_VAL. This name can be changed, however. This function value can be "deactivated" for the interface declaration if it is occupied with the data type VOID. Any other data type "activates" the function value.

LAD



A function FC 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.

FBD



A function FC 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.

SCL

```
"FC_name"(
     In1    := … ,
     In2    := … ,
     Out1   => … ,
     Out2   => … ,
     InOut1 := … ,
     InOut2 := … )
```

A function FC 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.

```
#Tag := "FC_name"(
     In1    := … ,
     In2    := … ,
     Out1   => … ,
     Out2   => … ,
     InOut1 := … ,
     InOut2 := … );
```

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.

STL

```
Call   "FC_name"
    In1     := …
    In2     := …
    RET_VAL := …
    Out1    := …
    Out2    := …
    InOut1  := …
    InOut2  := …
```

A function FC is called in STL by the CALL operation and by its name. The next lines contain the parameter list. The parameters are specified in the order of their declaration.

If the function value is "activated", it is represented as the first output parameter.

Fig. 14.6 Call functions for a function FC

FC call with LAD and FBD

You use the call box with LAD and FBD to call a function FC.

You can use the enable input EN to structure the block call depending on the result of logic operation. 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 the EN input has a preceding logic operation, the block call is only executed if the latter is fulfilled.

With the enable output ENO, you can transfer a group error message to the program of the calling block. You can influence the signal state of the enable output for LAD and FBD using the block end function.

FC call with SCL

SCL distinguishes between functions FC with and without a function value.

You call a function FC without a function value with its name. This is followed by the parameter list in round brackets. You must assign values to all existing parameters; the parameter 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. The parameter sequence is defined by the declaration. All parameters must be supplied with values.

You cannot use the implicitly defined enable input EN for a function. Instead, use the IF statement for a conditional call.

With the enable output ENO, you can transfer a group error message to the program of the calling block. You can influence the signal state of the enable output in SCL using the block-internal ENO tag. If you wish to use the implicitly defined enable output ENO, add it to the parameter list as the last entry.

FC call with STL

With STL, you call a function FC using the operation CALL and the name of the function. CALL is an absolute call, i.e. the specified block is always called and processed independent of conditions.

If you want to call a function FC conditionally, you can, for example, set a conditional jump function before the function call, which skips the call statement if the block is not to be called.

With the enable output ENO, you can transfer a group error message to the program of the calling block. You can influence the signal state of the enable output for STL using the binary result BR. The group error message (the quasi enable output ENO) is scanned directly after the call statement via the binary result BR.

14.2.3 Calling a function block FB

A function block FB is a block which can save its own data for each call. The data is saved in an instance data block. When called as a single instance, this is a separate data block for each call; for a call in a function block, the called function block can also save its data as local instance in the instance data block of the calling function block ("multi-instance"). Fig. 14.7 shows the call of a function block in the various programming languages.

The block parameters of a function block are saved in the instance data. Block parameters whose values are saved do not have to be provided with actual parameters when called. In this case, they retain the value of the last call. Block parameters which are saved as pointers must be supplied when called. These are in/out parameters with a structured data type and all block parameters with a parameter type.

FB call with LAD and FBD

You use the call box with LAD and FBD to call a function block. You can use the enable input EN to structure the block call depending on the result of logic operation. 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 the EN input has a preceding logic operation, the block call is only executed if the latter is fulfilled.

With the enable output ENO, you can transfer a group error message to the program of the calling block. You can influence the signal state of the enable output for LAD and FBD using the block end function.

FB call with SCL

With SCL, you call a function block with the name of the instance data, i.e. the name of the data block for a single instance or the tag name for a local instance. The parameter list contains all parameters in the declared order.

If you wish to use the implicitly defined enable input EN, add it to the parameter list as the first entry.

With the enable output ENO, you can transfer a group error message to the program of the calling block. You can influence the signal state of the enable output in SCL using the block-internal ENO tag. If you wish to use the implicitly defined enable output ENO, add it to the parameter list as the last entry.

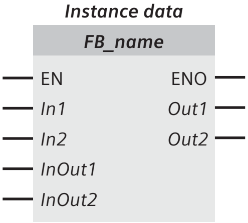
FB call with STL

With STL, you call a function block FB with the operation CALL and the name of the instance data, i.e. the name of the data block for a single instance or the tag name for a local instance. CALL is an absolute call, i.e. the specified block is always called and processed independent of conditions. The parameter list contains all parameters in the declared order.

Call of a function block (FB) and a system function block (SFB)

A **function block FB** is 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 complex data type and block parameters with parameter type must always be supplied. Supplying of the other block parameters is optional.

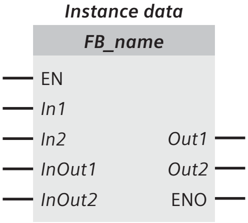
LAD



A function block FB is called in LAD by the EN/ENO box. The input and in/out parameters are present on the left 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



A function block FB is called in FBD by the EN/ENO box. The input and in/out parameters are present on the left 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   => … ,
  InOut1 := … ,
  InOut2 := … );

#Instance_name(
  In1    := … ,
  In2    := … ,
  Out1   => … ,
  Out2   => … ,
  InOut1 := … ,
  InOut2 := … );
```

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.

This is followed by the list of block parameters in parentheses, in each case in the order of their declaration and separated by a comma. Only the block parameters need be listed which are supplied.

STL

```
CALL "FB_name", "DB_name"
  In1    := …
  In2    := …
  Out1   := …
  Out2   := …
  InOut1 := …
  InOut2 := …

CALL #Instance_name
  In1    := …
  In2    := …
  Out1   := …
  Out2   := …
  InOut1 := …
  InOut2 := …
```

The block name is specified when calling as a single instance, followed by a comma and the name of the instance data block. When called as a local instance, the instance name is specified.

The next lines contain the list of block parameters, in each case in the order of their declaration. Only the block parameters need be listed which are supplied.

Fig. 14.7 Call functions for a function block FB

If you want to call a function block FB conditionally, you can, for example, set a conditional jump function before the function block call, which skips the call statement if the block is not to be called.

With the enable output ENO, you can transfer a group error message to the program of the calling block. You can influence the signal state of the enable output for STL using the binary result BR. The group error message (the quasi enable output ENO) is scanned directly after the call statement via the binary result BR.

14.3 Block end functions

A block end function prematurely terminates the processing in a block. A return is made to the previously processed block in which the call of the block just terminated is present. After an organization block is terminated, processing is continued in the CPU's operating system. Programming of a block end function at the end of the block program is optional. Fig. 14.8 shows the representation of the block end functions in the various programming languages.

14.3.1 Block end function RET (LAD and FBD)

The block end function is programmed as RET coil or RET box at the end of a network. If the preceding logic operation is fulfilled, the block is exited (conditional block end). If the preceding logic operation is not fulfilled, the next network in the block is processed.

If the RET coil is connected to the left power rail, or RLO = “1” is fixed at the box input or it is not allocated, the block will always be exited (absolute block end). Any network that follows this can then only be processed if it has a jump label and is jumped to using a jump function.

A signal state can be saved in the return value and is mapped to the enable output ENO. To set the return value, double-click on the RET function and select *Ret* (RLO, corresponds to the result of logic operation), *Ret True* or *Ret False* for a constant value, or *Ret Value* for a binary tag from the drop-down list. If an organization block is ended 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 RET. The block end function can be programmed multiple times in a block.

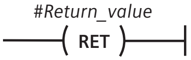
14.3.2 RETURN statement (SCL)

With RETURN the currently processed block is exited without conditions. A conditional block end can be programmed using the IF statement. RETURN transfers the signal state of the block-internal ENO tag to the enable output of the exited block.

Block end functions

Processing in the current block is terminated by the block end functions, and continued in the calling block. If an organization block is terminated, processing is continued in the CPU's operating system.

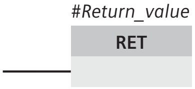
LAD



The RET coil is present alone in a network as termination of a preceding logic operation. The block is exited if the preceding logic operation is fulfilled.

When the block is exited, the signal state of the return value is mapped in the ENO enable output of the call box.

FBD



The RET box is present alone in a network as termination of a preceding logic operation. The block is exited if the preceding logic operation is fulfilled.

When the block is exited, the signal state of the return value is mapped in the ENO enable output of the call box.

SCL

```
ENO := #Return_value;
RETURN;         //Block end
...
...
IF <Condition> THEN
   ENO := #Return_value;
   RETURN;
END_IF;
...
```

RETURN terminates block processing independent of conditions. The value of the block-internal tag ENO is transferred to the enable output ENO when the block is exited.

Exiting the block dependent upon the result of logic operation can be programmed with an IF statement.

STL

```
//Conditional block end
A   #Return_value
SAVE
A   #Exit_block
BEC

//Absolute block end
A   #Return_value
SAVE
BEU

//Block end
A   #Return_value
SAVE
BE
```

BEC terminates block processing if RLO = "1".

BEU and BE terminate block processing independent of conditions.

BE is the last statement in a block and can also be omitted.

The status bit BR is controlled with SAVE. The signal state of BR is assigned to the enable output ENO of the block.

Fig. 14.8 Block end functions

If an organization block is ended via RETURN, the signal state of the ENO tag has no meaning. RETURN can be programmed multiple times in a block. Programming of RETURN at the end of a block is optional.

14.3.3 Block end functions BEC, BEU, and BE (STL)

Execution of BEC depends on the result of logic operation (RLO). If the RLO is “1” upon execution of BEC, the statement is executed and the block is terminated. A return is made to the previously executed block in which the block call was present. If the RLO is “0” upon execution of the BEC statement, the statement is not executed. The RLO is set to “1” and the statement that follows BEC is processed. A subsequently programmed scan statement is always a first input bit scan.

The block is exited upon processing of BEU. A return is made to the previously executed block in which the block call was present. In contrast to the BE statement, you can program BEU repeatedly within a block. The program section following BEU is only processed if it is jumped to by means of a jump function.

The block is terminated upon processing of BE. A return is made to the previously executed block in which the block call was present. BE is the last statement of a block. Programming of BE is optional.

When a block is ended, the signal state of the status bit BR (binary result) is assigned to the enable output ENO. Chapters 10.7.1 “Working with status bits” on page 442 and 10.7.2 “EN/ENO mechanism in the statement list” on page 447 describe how BR can be controlled and evaluated.

14.4 Data block functions

The data tags are saved in the data blocks. Data blocks can be located at two locations in the user memory: in the work memory and/or in the load memory. “Normally” a data block is first saved in the load memory when it is loaded into the CPU and is later transferred to the work memory. This is because the data tags in the work memory are accessed in a time-optimized way and this can be carried out as often as required.

It is additionally possible to save data blocks only in the load memory, which can be designed much larger than the work memory. This is preferentially carried out for data which is used very infrequently in the program, for example for recipes or archives, since access operations to the load memory require a very long time and the number of write operations is physically limited. In the case of data blocks that are only located in the load memory, the *Only store in load memory* attribute is activated.

Chapter 6.4 “Programming a data block” on page 270 describes how to add a data block to the user program.

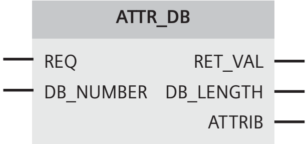
The program elements catalog also contains functions which write the contents of data blocks to the memory cards in CSV format so that data protocols and recipes, for example, can be further processed with a spreadsheet routine. Recipes in CSV format can also be imported from the memory card into a data block in the user memory. These functions are described in the Chapter 18.5 “Data logging and transferring recipes” on page 813.

14.4.1 Read data block attributes

ATTR_DB provides information about a data block in the user memory. You find ATTR_DB in the program elements catalog under *Extended instructions > Data block control* (Fig. 14.9).

Read data block attributes

ATTR_DB



ATTR_DB reads the attributes of a data blocks in the user memory.

Assignment of the ATTRIB parameter

Bit	Meaning	
0	"0" "1"	Data block is only in the work memory Data block is only in the load memory
1	"0" "1"	Data block is not write-protected Data block is write-protected
2	"0" "1"	Data block is retentive Data block is not retentive
3	"0" "1"	Data block is either in the load memory or in the work memory Data block is in both the load memory and in the work memory
4–7	"0"	= unassigned =

The bits 0 and 3 of the parameter ATTRIB indicate in the memory that contains the data block:

Bit 0	Bit 3	The data block is located
"0" "1"	"0" "0"	only in the work memory only in the load memory
"0" "1"	"1" "1"	in the load and work memory in the load and work memory

Fig. 14.9 Reading data block attributes with ATTR_DB

The data block is specified on the parameter DB_NUMBER, either with its name, its number (constant), or with a UINT tag with the number of the data block as value.

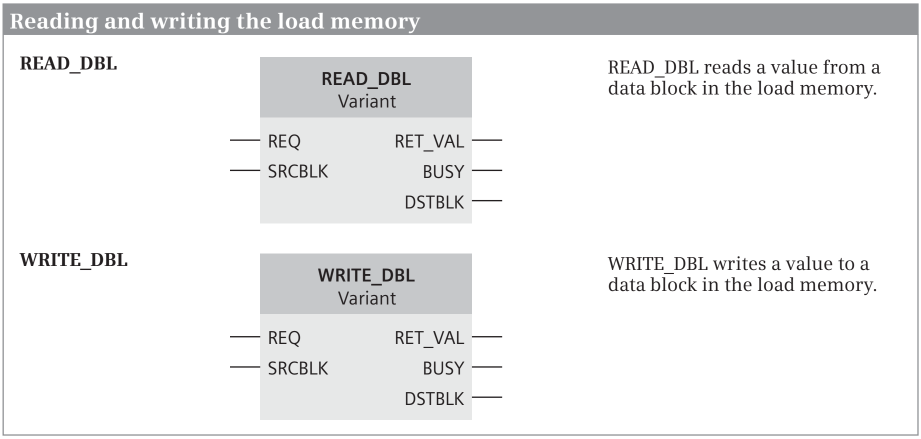
The attributes are output at the ATTRIB parameter with signal state “1” on the REQ parameter. Section “Block attributes for data blocks” on page 273 describes the attributes used.

The parameter DB_LENGTH contains the number of existing bytes. For data blocks with the attribute *Optimized block access* activated, the length cannot be read out. The length zero is then present at the parameter DB_LENGTH.

If an error occurs while the job is being processed, the parameter RET_VAL gives out error information.

14.4.2 Reading and writing the load memory

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. Using the “normal” addressing, data tags are addressed in the work memory. READ_DBL and WRITE_DBL access data blocks in the load memory. You find the system blocks in the program elements catalog under *Extended instructions* > *Data block control* Fig. 14.10 shows the graphic representation of the system blocks.



READ_DBL reads a value from a data block in the load memory.

WRITE_DBL writes a value to a data block in the load memory.

Fig. 14.10 Transfer data areas from and to the load memory

READ_DBL

READ_DBL transfers a data block or a data area – for example, recipe data – from the load memory to the work memory. Both the source data block and the target data block must have the same access type: The block attribute *Optimized block access* must be either enabled or disabled in both data blocks.

The read job is started with a signal state "1" at the REQ parameter. READ_DBL is an asynchronously working system block which needs several processing cycles to execute the job. As long as the parameter BUSY has signal state "1", the read procedure is not completed. If an error occurs while the job is being processed, the parameter RET_VAL gives out error information.

The data area (source) which is located in the load memory is specified on the parameter SRCBLK and the data area (target) which is located in the work memory is specified on the parameter DSTBLK. The actual parameters for these block parameters are described below.

WRIT_DBL

WRIT_DBL transfers a data block or a data area – for example, archive data – from the work memory to the load memory. Both the source data block and the target data block must have the same access type: The block attribute *Optimized block access* must be either enabled or disabled in both data blocks.

The write job is started with a signal state at the REQ parameter. WRIT_DBL is an asynchronously working system block which needs several processing cycles to execute the job. As long as the parameter BUSY has signal state "1", the write procedure is not completed. If an error occurs while the job is being processed, the parameter RET_VAL gives out error information.

The data area (source) which is located in the work memory is specified on the parameter SRCBLK and the data area (target) which is located in the load memory is specified on the parameter DSTBLK.

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.

Parameters SRCBLK and DSTBLK

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 pointers to an absolutely addressed data area, e.g. P#DB100.DBX16.0 BYTE 64 (see Chapter 4.9.4 “ANY pointer” on page 135 for description).

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.

14.4.3 ARRAY data blocks

An ARRAY data block has the structure of the ARRAY data type: The data tags are the components of an array which all have the same data type. The array has an index which starts with zero and ends at an adjustable upper limit.

You can create a new ARRAY 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 icon for *Data block* and the entry *Array DB* from the drop-down list as the type. Give the data block a meaningful name and, if desired, a different number using the *manual* option.

Then select the data type of the data tag from a drop-down list in the *Array data type* field. If the components of the ARRAY data block are to have a PLC data type, you must have previously created the PLC data type. Then define the upper limit for the index. The lower limit is always zero and cannot be changed. The new data block is created by clicking on the *OK* button.

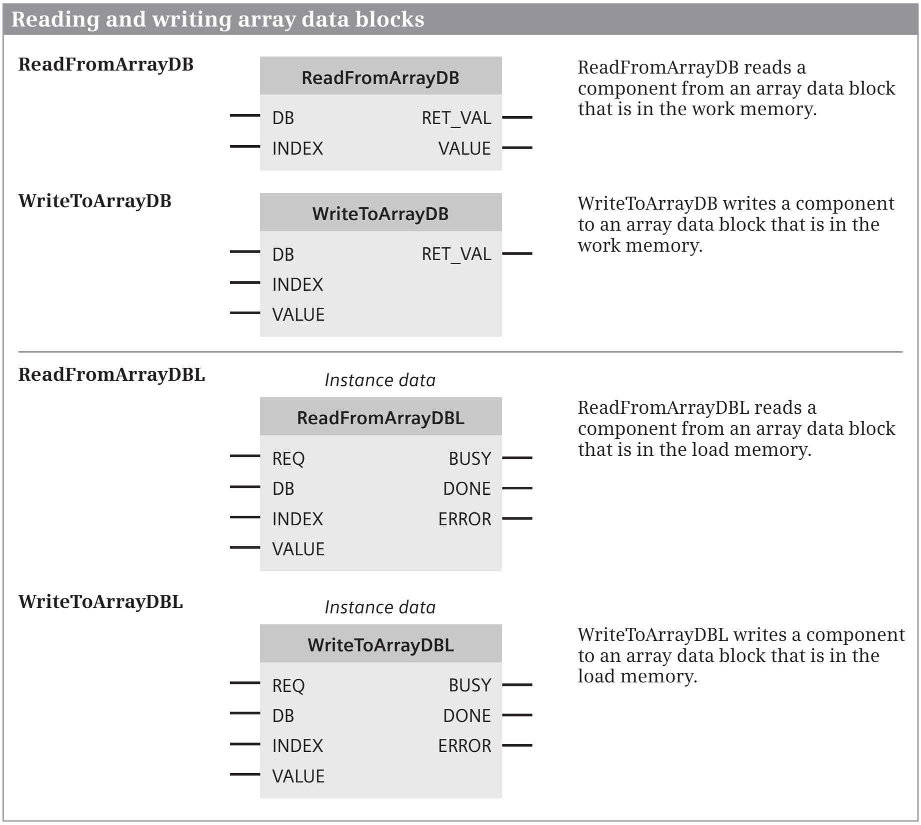
An ARRAY data block has the attributes *Only store in load memory*, *Data block write-protected in the device*, and *Optimized block access*. These attributes are described in Chapter 5.3.2 “Block properties” on page 157. Chapter 4.3.3 “Indirect addressing of a tag in an ARRAY DB” on page 102 shows how to address the data tags in an ARRAY data block. System blocks provide another access option. These also permit indirect addressing of the data block and they can access the load memory (see next chapter).

14.4.4 System blocks for access to ARRAY data blocks

For reading and writing components of an ARRAY data block, there are the following system blocks:

- ReadFromArrayDB
  Read from an ARRAY data block in the work memory
- WriteToArrayDB
  Write to an ARRAY data block in the work memory
- ReadFromArrayDBL
  Read from an ARRAY data block in the load memory
- WriteToArrayDBL
  Write to an ARRAY data block in the load memory

You can find these system blocks in the program elements catalog under *Basic instructions* > *Move operations*. Fig. 14.11 shows the graphic representation of the system blocks.



ReadFromArrayDB reads a component from an array data block that is in the work memory.

WriteToArrayDB writes a component to an array data block that is in the work memory.

ReadFromArrayDBL reads a component from an array data block that is in the load memory.

WriteToArrayDBL writes a component to an array data block that is in the load memory.

Fig. 14.11 Reading and writing array data blocks

ReadFromArrayDB

ReadFromArrayDB reads a component from an ARRAY data block which is located in the main memory (the block attribute *Only store in load memory* is deactivated). Read access takes place synchronously, i.e. the read value is immediately available after the function call.

The ARRAY data block is specified on the parameter DB, either with its name, its number (constant), or with a tag with the data type UINT and the number of the data block as value. The component that has its index present at the parameter INDEX, either as a constant or as a tag, is output at the parameter VALUE.

WriteToArrayDB

WriteToArrayDB writes a component to an ARRAY data block which is located in the main memory (the block attribute *Only store in load memory* is deactivated). Read access takes place synchronously, i.e. the value is immediately available in the ARRAY data block after the function call.

The ARRAY data block is specified on the parameter DB, either with its name, its number (constant), or with a tag with the data type UINT and the number of the data block as value. The component that is to be written and has its index present at the parameter INDEX, either as a constant or as a tag, is specified at the parameter VALUE.

ReadFromArrayDBL

ReadFromArrayDBL reads a component from an ARRAY data block which is located in the load memory (the block attribute *Only store in load memory* is activated). Read access takes place asynchronously, i.e. several processing cycles can elapse before the read value is available.

The ARRAY data block is specified on the parameter DB, either with its name, its number (constant), or with a tag with the data type UINT and the number of the data block as value. The index of the component to be read is located at the parameter INDEX, either as a constant or as a tag.

Processing of the job is initiated with a rising edge on the parameter REQ. As long as the job is being executed, the BUSY parameter has signal state “1”. When the DONE parameter has signal state “1”, the job has been processed without errors. Otherwise, error information is present at the ERROR parameter.

The read value is output at the VALUE parameter.

WriteToArrayDBL

WriteToArrayDBL writes a component to an ARRAY data block which is located in the load memory (the block attribute *Only store in load memory* is activated). Write access takes place asynchronously, i.e. several processing cycles can elapse before the value is available in the ARRAY data block.

The ARRAY data block is specified on the parameter DB, either with its name, its number (constant), or with a tag with the data type UINT and the number of the data block as value. The index of the component to be written is located at the parameter INDEX, either as a constant or as a tag. The value to be written is present at the parameter VALUE.

Processing of the job is initiated with a rising edge on the parameter REQ. As long as the job is being executed, the BUSY parameter has signal state “1”. When the DONE parameter has signal state “1”, the job has been processed without errors. Otherwise, error information is present at the ERROR parameter.

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.

14.4.5 CPU data blocks

A data block that is created during runtime using CREATE_DB is called a “CPU data block”. In the user program, you can treat a CPU data block like a data block that was created by the programming device. You can write data to this block and read data.

A CPU data block is initially only present in the user memory of the CPU. In online mode, a CPU data block is marked with a small CPU symbol in the user memory. Its tag values can be monitored using a watch table.

You can load a CPU data block with a complete loading procedure (*Online > Load from device*) from the user memory into the offline project. In the offline project, a CPU data block is marked with a small CPU symbol. A CPU data block cannot be loaded back from the offline project into the CPU.

In the offline project, you can open a CPU data block and view its contents or compare it to the online version, but you cannot synchronize it. You also cannot change a CPU data block, provide it with know-how protection, compile it, or convert it into a different type of data block. You can delete a CPU data block using the programming device, both in the user memory and in the offline project.

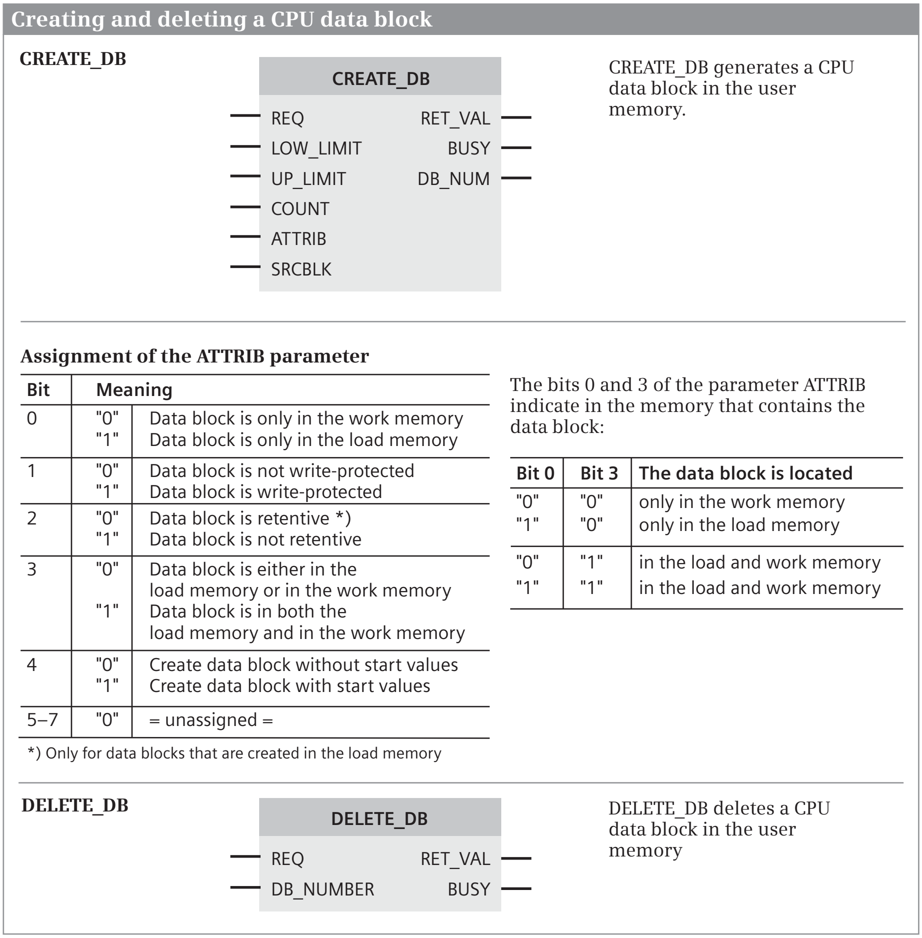
The following system blocks exist for creating and deleting a CPU data block:

- ▷ CREATE_DB Create a CPU data block in the user memory
- ▷ DELETE_DB Delete a CPU data block in the user memory

You find these system blocks in the program elements catalog under *Extended instructions > Data block control*. Fig. 14.12 shows the graphic representation of the system blocks.

CREATE_DB

CREATE_DB generates a data block in the user memory. For the number of the data block, the system block uses the lowest free number in the number range which is specified by the input parameters LOW_LIMIT and UP_LIMIT. The numbers speci-



Creating and deleting a CPU data block

CREATE_DB

CREATE_DB generates a CPU data block in the user memory.

Assignment of the ATTRIB parameter

Bit	Meaning	
0	"0" "1"	Data block is only in the work memory Data block is only in the load memory
1	"0" "1"	Data block is not write-protected Data block is write-protected
2	"0" "1"	Data block is retentive *) Data block is not retentive
3	"0" "1"	Data block is either in the load memory or in the work memory Data block is in both the load memory and in the work memory
4	"0" "1"	Create data block without start values Create data block with start values
5–7	"0"	= unassigned =

*) Only for data blocks that are created in the load memory

The bits 0 and 3 of the parameter ATTRIB indicate in the memory that contains the data block:

Bit 0	Bit 3	The data block is located
"0"	"0"	only in the work memory
"1"	"0"	only in the load memory
"0"	"1"	in the load and work memory
"1"	"1"	in the load and work memory

DELETE_DB

DELETE_DB deletes a CPU data block in the user memory

Fig. 14.12 Creating and deleting a CPU data block

fied at these parameters are included in the number range. If the two values are the same, the data block is created with exactly this number. The number of a data block already included in the user program cannot be assigned again.

The output parameter DB_NUM delivers the number of the actually created data block. The input parameter COUNT is used to specify the length of the data block to be created. The length corresponds to the number of data bytes and must be an even number.

At the ATTRIB parameter, define additional properties of the data block to be created. Using the bits 0 and 3, you specify the part of the user memory in which the data block is to be created: only in the work memory, only in the load memory, or in both. Using bit 1, you set the write protection of the data block. If the data block is created in the load memory, you can activate the retentivity with bit 2.

Using bit 4, you can set whether the data block is to be created with or without start values. The start values are taken from the data block or the data area which is specified at the parameter SRCBLK. The *Optimized block access* attribute must be deactivated for the data block addressed with SRCBLK. The following are permitted as actual parameters:

- ▷ an entire data block that is derived from a PLC data type or system data type,
- ▷ a tag from a data block, and
- ▷ a pointer to an absolutely addressed data area, e.g. P#DB100.DBX16.0 BYTE 64 (see Chapter 4.9.4 "ANY pointer" on page 135 for description).

If the area specified at SRCBLK is larger than the created data block, the start values are entered up to the length of the data block. If the area at SRCBLK is smaller, the remaining bytes in the data block are filled with 16#00.

Job processing is started with signal state "1" at the REQ parameter. As long as the job is being executed, the BUSY parameter has signal state "1". Then the start values in the source area must not be changed. A data block is not created in the event of an error. The DB_NUM parameter is then occupied by zero and an error number is output via RET_VAL.

DELETE_DB

DELETE_DB deletes a data block that was created with CREATE_DB in the user memory. The number of the data block to be deleted can be specified by you at the parameter DB_NUMBER.

The deletion process is started with signal state "1" at the REQ parameter. As long as the job is being executed, the BUSY parameter has signal state "1". If an error occurred, the parameter RET_VAL gives out error information.

If the data block is currently called or if it is still called in the call hierarchy further "up" (in the calling blocks), the organization block OB 121 is called during an attempted deletion. If this is not present, the CPU switches to the STOP operating state.

15 Online mode and program test

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 established via Industrial Ethernet. The mechanical connection (networking) and the logical connection (the definition of the transmission protocols) are not configured. Only the network addresses – the addresses of the PROFINET interface of the two devices – must be harmonized with one another.

In online mode, STEP 7 changes the representation of the user interface: The title bars of the windows are displayed in orange. In the project tree, the objects of the station which is switched 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 state of the CPU, to set the time on the CPU, and to 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 state of the CPU. If certain prerequisites are met, it can also take place in the RUN operating state.

You can compare the online versions and the offline version of a block. Changes to a block are always made in the offline version, which is then transferred to the CPU.

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 CPU user program 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.

15.1 Connection of 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 82.

If the programming device already has an address other than the PLC station, STEP 7 sets up a “temporary” IP address on the programming device. This temporary IP address is deleted when Windows is shut down.

15.1.1 IP addresses of the programming device

Determining and setting network addresses with 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.

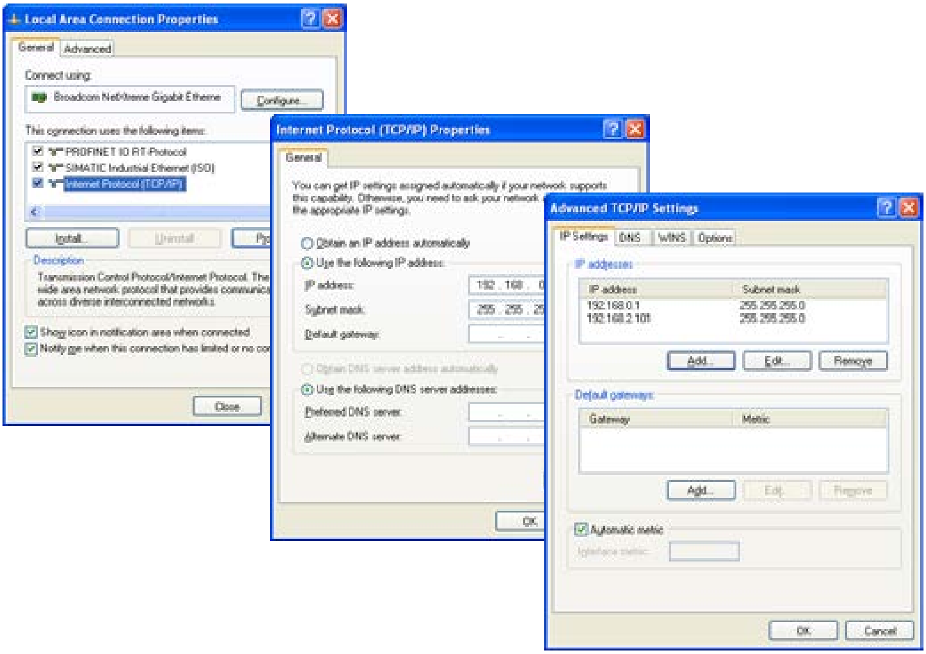
The connection status is displayed in the *General* tab. Click here on the *Properties* button. In the *Properties of ...* window, select the entry *Internet Protocol (TCP/IP)* or *Internet Protocol Version 4 (TCP/IPv4)* in the *This connection uses the following items* field, and then click on the *Properties* button (Fig. 15.1).

The dialog window *Internet Protocol Properties...* offers in the *General* tab the options *Obtain an IP address automatically* (via a DHCP server) and *Use the following IP address* (manual settings). If you want to enter an additional IP address, for example for the SIMATIC project, select the option *User defined* in the *Alternate Configuration* tab and enter the IP address and the subnet mask.

Set access point

When installing STEP 7, 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



Local Area Connection Properties

Connect using: Broadcom NetXtreme Gigabit Etherne

This connection uses the following items:

☑ PROFINET IO RT-Protocol

☑ SIMATIC Industrial Ethernet (ISO)

☑ Internet Protocol (TCP/IP)

Description

Transmission Control Protocol/Internet Protocol. The
wide area network protocol that provides communica
across diverse interconnected networks.

☑ Show icon in notification area when connected

☑ Notify me when this connection has limited or no co

Internet Protocol (TCP/IP) Properties

You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network
the appropriate IP settings.

○ Obtain an IP address automatically

● Use the following IP address:

IP address: 192 . 168 . 0

Subnet mask: 255 . 255 . 25

Default gateway: . .

○ Obtain DNS server address automatically

● Use the following DNS server addresses:

Preferred DNS server: . .

Alternate DNS server: . .

Advanced TCP/IP Settings

IP addresses

IP address	Subnet mask
192.168.0.1	255.255.255.0
192.168.2.101	255.255.255.0

Default gateways

Gateway	Metric

☑ Automatic metric

Interface metric:

Fig. 15.1 Setting IP addresses with the Windows Control Panel

Access Point of the Application box. Select the LAN or WLAN interface module used under *Interface Parameter Assignment Used* and close the tool.

Interface (adapter) in the programming device

STEP 7 lists all active interface adapters of the programming device in the project tree under *Online access*. In order to check and set the interface properties, click with the right mouse button on the interface used and select the *Properties* command from the shortcut menu. In the properties window, select the previously configured subnet with which the programming device is to be connected under *General* from the *Assignment* drop-down list. The current settings, such as the IP address, can be found under *Configuration* > *Industrial Ethernet*.

15.1.2 Connecting the programming device to the PLC station

Connect the terminal of the programming device to the PROFINET interface of the CPU. You can use a standard or “cross-over” cable for this. The CPU can handle both cable types. Ensure that a memory card is inserted in the CPU and switch on the power supply for the CPU.

After the startup, the CPU is in the STOP state (if the memory card is blank). The RUN/STOP LED illuminates yellow and the mode is indicated in the display with a yellow background.

If you set the mode switch to RUN, the CPU ramps up and the RUN LED flashes. If the CPU does not detect any errors during ramping up, it changes to RUN mode – even without user program. The RUN operating state is indicated with a green RUN/STOP LED and has a green background in the display.

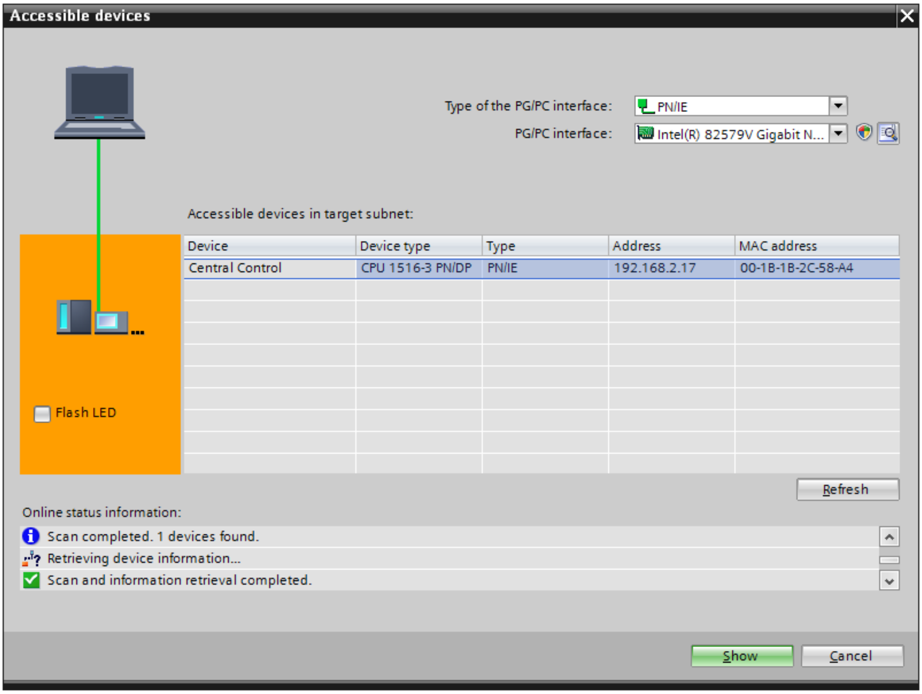
The CPU is ready for communication in both the RUN and STOP modes.

Search for accessible devices

Start STEP 7, select the *Online & diagnostics* portal in the Portal view, and then select *Accessible devices*. If necessary, set the type of PG/PC interface in the *Accessible devices* window and the adapter used under *PG/PC interface*.

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. 15.2).

Select the line with the station. You can then click the *Flash LED* checkbox in order to briefly flash the LEDs on the front panel of the CPU. To further edit the selected station in the project view, deactivate the *Flash LED* checkbox and click on the *Show* button.



Type of the PG/PC interface: PN/IE

PG/PC interface: Intel(R) 82579V Gigabit N...

Accessible devices in target subnet:

Device	Device type	Type	Address	MAC address
Central Control	CPU 1516-3 PN/DP	PN/IE	192.168.2.17	00-1B-1B-2C-58-A4

Online status information:

- Scan completed. 1 devices found.
- Retrieving device information...
- Scan and information retrieval completed.

Fig. 15.2 Dialog window *Accessible devices*

Configuring a temporary IP address on the programming device

If the network settings of the programming device do not agree with those of the CPU, STEP 7 suggests the setting of a matching project-specific IP address on the programming device. This IP address is present temporarily until the programming device is switched off or until you delete the address. Answer the corresponding dialogs for assigning an IP address with *Yes* or *OK*. The assigned IP address is displayed in the confirmation dialog.

STEP 7 then shows the found CPU in the project view. The CPU is located with its IP or MAC address in the *Online access* group under the used interface module as a new group in the project tree.

15.1.3 Assigning an IP address to the CPU

If a CPU does not have an IP address, you can assign an IP address to the CPU: Highlight the PLC station and select the *Online & diagnostics* command from the shortcut menu. Select the entry *Assign IP* address in the Functions section of the diagnostics window. Enter the desired IP address and subnet mask and click *Assign IP address*. The result of the action is reported in the inspector window in the *Info* tab.

15.1.4 Switching on online mode

Under *Online access*, select the PLC station and then *Online & diagnostics* from the shortcut menu. If the CPU does not yet have an IP address, enter the IP address and subnet mask in the diagnostics window under *Functions > Assign IP address* and click on *Assign IP address*. Then repeat the command *Online & diagnostics*.

The diagnostics window displays the diagnostic data read from the PLC station and the *Online tools* task card with the CPU control panel. Further details can be found in Chapter 15.4 “Hardware diagnostics” on page 672.

If a project matching the online PLC station is present, open the project and select the PLC station in the project tree. Select *Go online* from the shortcut menu or activate the *Go online* icon in the main menu. If necessary, add the access data in the *Go online* window and click on *Go online*.

Further procedure

- ▷ Chapter 15.4 “Hardware diagnostics” on page 672 describes how you can use the diagnostics and online tools, for example to start and stop the CPU or to reset to the default settings.
- ▷ The following Chapter 15.2 “Transferring project data” describes how you can upload a user program to the PLC station and edit the user program online.
- ▷ Chapter 15.5 “Testing the user program” on page 677 describes how you can test a user program.
- ▷ Chapter 15.2.4 “Working with online project data” on page 660 describes how you can access the online project data of the CPU without the user program.

15.1.5 Resetting the CPU memory

A memory reset returns the CPU to its “initial state”. It can only be carried out in the STOP state. The complete user program present in the work memory and all operands are deleted independent of the retentivity setting. The hardware configuration with the IP address, the diagnostics buffer, the real-time clock, the runtime meter, and the running force jobs are retained during the memory reset.

If a memory card is inserted, the execution-relevant parts of the user program are copied from the load memory into the work memory.

There are three options for a memory reset of the CPU: using a connected programming device (see Chapter 15.4.5 “Online tools” on page 676), by means of the mode switch on the CPU, or via the CPU display.

Memory reset using the mode switch

Note: If a memory card is inserted, a memory reset is carried out using the following switch operation. If no memory card is inserted, the CPU is reset to the factory settings.

To perform a memory reset using the mode switch, move the switch to the STOP position. The STOP LED lights up. Then hold the switch in the MRES position for at least three seconds. During this procedure, the STOP LED goes out for a second, then illuminates for a second, goes out again for a second, and then illuminates continuously. Now move the switch to the STOP position, then to the MRES position within three seconds, and then back to the STOP position again. While the memory reset is being performed, the STOP LED will flash for at least three seconds at 2 Hz and then remain lit.

If the CPU requests a memory reset by slowly flashing the STOP LED, move the mode switch to the MRES position and then to the STOP position. While the memory reset is being performed, the STOP LED will flash for at least three seconds at 2 Hz and then remain lit.

Memory reset via the CPU display

Using the *Left* or *Right* keys, select the menu *Settings* and confirm with *OK*. Now use the *Down* key to select the line *Reset* and confirm with *OK*. In the *Reset* submenu, select the line *Memory reset* and confirm with *OK*. Confirm the *Memory reset* submenu with *OK*. The memory reset is then executed.

15.1.6 Reset to the factory settings

“Reset to factory settings” restores the factory default settings in the CPU. The CPU must be in the STOP operating state. When resetting to the factory settings, the complete user program present in the work memory and all operands are deleted independent of the retentivity setting. All of the parameters, the I&M data (with the exception of the I&M0 data) and the runtime meters are deleted and the real-time clock is reset to the value DTL#1990-01-01-0:0:0.000.

There are three options for resetting the CPU to the factory settings: using a connected programming device (see Chapter 15.4.4 “Diagnostic functions” on page 675), by means of the mode switch on the CPU, or via the CPU display.

When the reset is carried out using the mode switch or the CPU display, the IP address is deleted; when the reset is carried out using the programming device, the IP address can be retained or deleted.

Resetting to the factory settings using the mode switch

Note: If a memory card is inserted, a memory reset is carried out using the following switch operation. If no memory card is inserted, the CPU is reset to the factory settings.

To reset to the factory settings, switch the CPU to the STOP operating state and remove the memory card. Then perform a memory reset:

Switch the mode switch to STOP. The STOP LED lights up. Then hold the switch in the MRES position for at least three seconds. During this procedure, the STOP LED goes out for a second, then illuminates for a second, goes out again for a second, and then illuminates continuously. Now move the switch to the STOP position, then to the MRES position within three seconds, and then back to the STOP position again. While the memory reset is being performed, the STOP LED will flash for at least three seconds at 2 Hz and then remain lit.

Finally, the CPU enters the “Reset to factory settings” event into the diagnostics buffer, and goes to the STOP operating state.

Reset to factory settings via the CPU display

Using the *Left* or *Right* keys, select the menu *Settings* and confirm with *OK*. Now use the *Down* key to select the line *Reset* and confirm with *OK*. In the *Reset* submenu, select the line *Factory defaults* and confirm with *OK*. Confirm the *Factory defaults* submenu with *OK*. The rest is then executed.

Finally, the CPU enters the “Reset to factory settings” event into the diagnostics buffer, and goes to the STOP operating state.

15.2 Transferring project data

You have configured the hardware and completed and compiled the user program. You can now carry out the transfer to the PLC station via an online connection or using a memory card as data medium.

If you transfer the user program to the CPU via an online connection, it is written into the load memory. In the case of a CPU 1500, the load memory is on the memory card.

You can also write to a memory card in the programming device and use it as data medium. Transfer the project data from the offline data management system to the memory card that is inserted in the programming adapter. Then insert the memory card into the CPU when the system is de-energized or in the STOP operating state. When switching on, the modules are initialized and the execution-relevant data is imported from the load memory into the work memory of the CPU.

15.2.1 Loading project data for the first time

To load the project data, connect the programming device to the CPU, switch the CPU on, and open the project on the programming device.

Select the PLC station in the project tree and then the *Download to the device > Hardware and software (only changes)* command from the shortcut menu. When loading for the first time, the dialog window *Extended download to device* shows the address of the configured PLC station in the *Configured access nodes of ...* table. If applicable, select the subnet and adapter to which the PLC station is connected from the drop-down lists *Type of the PG/PC interface* and *PG/PC interface*. The *Online status information* table signals the status and the end of scanning for stations.

Select the desired station in the *Compatible devices in target subnet* table and click on the *Load* button.

The PLC station does not have the configured address

If the configured address does not agree with the address set in the CPU, STEP 7 cannot find the device matching the configuration. Activate the *Show all compatible devices* checkbox in this case. The search then starts again.

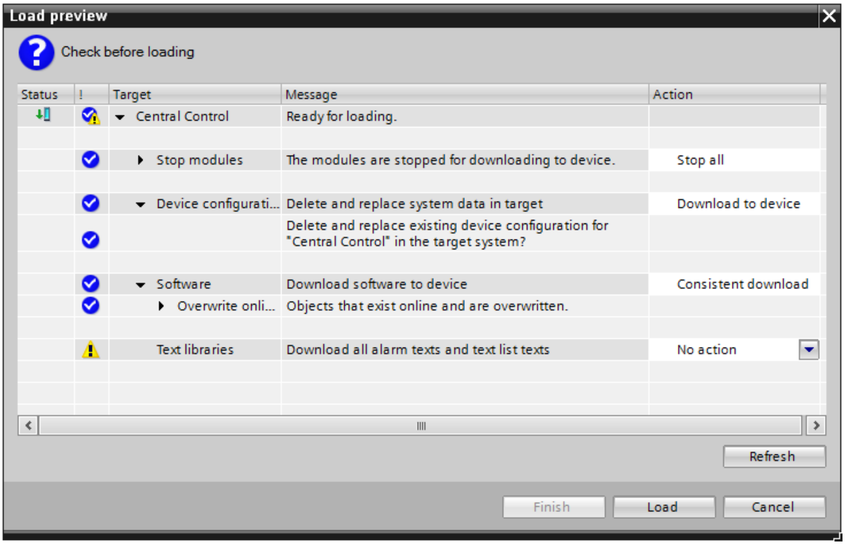
The devices that have been found are displayed together with their addresses in the table *Compatible devices in target subnet*. Select the required PLC station in this table and click on the *Load* button.

If the network settings of the programming device do not match the configured IP address when connecting via the PN interface, the dialog window *Assign IP address* is displayed. Following confirmation, STEP 7 then adds a further temporary, project-specific IP address.

The project data is compiled prior to loading

If necessary, the project data is compiled prior to loading. Only consistent project data which has been compiled without errors can be loaded. After the compilation, check the messages in the *Load preview* dialog window and change the suggested actions, if applicable. You can continue with loading by clicking on the *Load* button (Fig. 15.3).

The complete project data can only be loaded when the CPU is at STOP. If the RUN operating state is activated, you will be prompted to activate the stop action in the *Action* column. As long as no loading is possible, the *Load* button is grayed out.



Load preview

Check before loading

Status	!	Target	Message	Action
	✓	Central Control	Ready for loading.	
	✓	Stop modules	The modules are stopped for downloading to device.	Stop all
	✓	Device configurati...	Delete and replace system data in target	Download to device
	✓		Delete and replace existing device configuration for "Central Control" in the target system?	
	✓	Software	Download software to device	Consistent download
	✓	Overwrite onli...	Objects that exist online and are overwritten.	
	⚠	Text libraries	Download all alarm texts and text list texts	No action

Fig. 15.3 Dialog window *Load preview*

Start CPU following loading

The results of loading are displayed in the dialog window *Load results*. After the loading is completed you can control the operating state of the CPU after the loading is completed using the *Start all* checkbox in the *Action* column.

Caution: Make sure when starting the CPU – possibly with a faulty program – that the controlled machine cannot cause damage to property or injury to persons and that no dangerous states can occur!

To finish loading, click on the *Finish* button. If the *Start all* checkbox has been activated and no error occurs when the CPU is started up, the CPU is then in the RUN operating state. The RUN/STOP LED lights up green.

The result of loading is also shown in the inspector window under *Info* in the *General* tab.

Activating online mode

In order to activate online mode, select the PLC station or the *Program blocks* folder and then select the *Go online* command from the shortcut menu or activate the *Go online* icon in the main menu.

The title bar of the active window has an orange background. The project tree uses icons to indicate the agreement and existence of offline and online versions for each block. You can now

- open the diagnostics window
  It shows such things as the module status, the diagnostics buffer, the memory utilization and the current cycle times. It also allows the execution of online

functions such as setting the CPU clock or assigning an IP address, see Chapter 15.4.2 “Diagnostic information” on page 673.

- ▷ use the online tools
  These show the modes, the current cycle times, and the memory utilization. They also allow the CPU modes to be controlled using the programming device, see Chapter 15.4.5 “Online tools” on page 676.
- ▷ edit and compare the online version and offline version of a block
  (see Chapter 15.3 “Working with blocks in online mode” on page 662)
- ▷ test the user program
  (see Chapter 15.5 “Testing the user program” on page 677)

You can use the *Go offline* icon to switch online mode off again.

Loading an incorrectly compiled, inconsistent program

An error which occurs when compiling prior to loading is indicated by a white cross on a red background in the dialog window *Load preview*. The *Target* column indicates under *Software* the component where the error has occurred (click triangle on the left). Continuation or restart of loading is only possible when the error has been eliminated.

Error message following loading

If the CPU does not start following loading – the RUN/STOP LED lights up yellow – or if the red ERROR LED lights up or flashes, the diagnostics buffer can provide information on the cause. Remaining in the STOP state or returning to it could be the result of, for example, a faulty I/O access in the user program. Chapter 15.4.3 “Diagnostics buffer” on page 674 describes how the diagnostics buffer supports you during troubleshooting.

15.2.2 Reloading the project data

The project data can only be changed in the offline version. For example, if you want to change the CPU properties or the online version of a block, switch to offline mode, make the changes offline, and then start a load process.

When reloading project data, only the changes compared to the online project data are loaded. Select the object to be loaded in the project tree and then the *Download to device* >... command from the shortcut menu. You can

- ▷ with the PLC station selected, choose the commands *Hardware and Software (only changes)*, *Hardware configuration* or *Software (only changes)* or
- ▷ with the *Program blocks* folder selected or with one or more blocks selected, choose the command *Software (only changes)*.

The project data is compiled and the result is displayed in the *Load preview* dialog window. Set the actions if applicable and start the loading process by clicking on the *Load* button. You can set additional actions at the conclusion of the loading process in the *Load results* window. Clicking on the *Finish* button finishes the load process.

The result of loading is shown in the inspector window under *Info* > *General*. Further information on downloading individual blocks is provided in Chapter 15.3 “Working with blocks in online mode” on page 662.

15.2.3 Protecting the user program

The user program can be protected against unauthorized access by means of the following measures:

- Access protection restricts access to the CPU
- Know-how protection protects a block against unauthorized access
- Copy protection prevents a copied block from being executed by tying it to a specific memory card or CPU

Access protection restricts the access rights to online project data in the CPU. Blocks in the offline data management system and on the memory card are not protected by this. The method for protecting a block with know-how protection and/or copy protection is described in Chapter 6.3.4 “Protecting blocks” on page 259.

You configure access protection in the properties of the CPU with hardware configuration. The configured access rights can be limited via entries on the CPU display and, at the same time, via the ENDIS_PW function in the user program. The CPU saves the respective last setting.

Additional measures for protecting the user program can include deactivating the Web server, deactivating time synchronization via an NTP server, and deactivating PUT/GET communication. The latter is described in Chapter 17.3 “S7 communication” on page 761. Data exchange between CPUs via communication functions, for example with open user communication, is not limited by access protection (exception: the PUT/GET communication with complete protection).

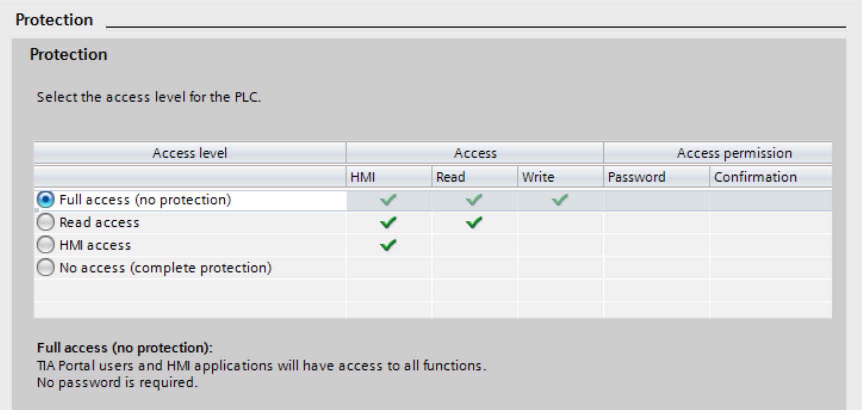
Access protection of the user program with protection levels

With access protection for a CPU 1500, you can protect the access to specific functions using passwords. You set the access protection under *General* > *Protection* with the hardware configuration when parameterizing the CPU properties (Fig. 15.4).

Full access (no protection) does not provide any protection. Any person can read and modify the configuration data and the user program.

If *Read access* is allowed, anyone can read the configuration data, the user program, and the diagnostic data (and load it into the programming device, for example). Without entering the password, the user cannot change the configuration data and the user program (cannot load it into the CPU, for example). The user also cannot perform a writing test function, cannot change the operating state from the programming device, and cannot perform a firmware update via an online connection.

If *HMI access* is permitted, it is possible to access diagnostic data and to access data from an HMI device. Without entering the password, the user cannot read or change the configuration data and the user program (cannot load it into the programming



Protection

Protection

Select the access level for the PLC.

Access level	Access			Access permission	
	HMI	Read	Write	Password	Confirmation
● Full access (no protection)	✓	✓	✓		
○ Read access	✓	✓			
○ HMI access	✓				
○ No access (complete protection)					

Full access (no protection):
TIA Portal users and HMI applications will have access to all functions.
No password is required.

Fig. 15.4 Setting the protection levels

device or CPU, for example). The user also cannot perform a writing test function, cannot change the operating state from the programming device, and cannot perform a firmware update via an online connection.

If *No access (complete protection)* is set, neither read access nor write access to the configuration data and the user program is possible without knowing the corresponding password. HMI access is also not possible. In addition, the server function for the PUT/GET communication is deactivated (cannot be changed).

You can perform accesses that are marked with a green checkmark in the configuration input screen without knowing the password. For all other actions, you need the password that corresponds to the access type. To set the passwords, select the option *No access (complete protection)* and then enter the password in the line with the access type.

The protection is effective once the settings have been loaded to the CPU. For the first access that is password-protected, you will be prompted to enter the password. Access protection by the password applies to the duration of the online connection or until the access privilege has been canceled again using *Online > Delete access rights*. The password-protected access can only be carried out at a specific point in time from a single programming device.

Blocking access with the CPU display

You can block the access to a password-protected CPU on the display of a CPU 1500. The block only functions if the mode switch is set to RUN. The configured access protection takes effect in the STOP position.

To block the access rights, select the menu *Settings > Protection* on the CPU display. Now you can set the access for each protection level separately (full access, read access, HMI access):

- ▷ *Allow* means that access is allowed for this level of protection if the correct password is known.
- ▷ *Disallow* means that no access is possible with the programing device while the mode switch is in the RUN position, even if the correct password is known.

You can also password-protect the operation of the CPU display in the CPU properties: Under *General* > *Display* > *Display protection*, select the checkbox *Enable display protection* and enter a password.

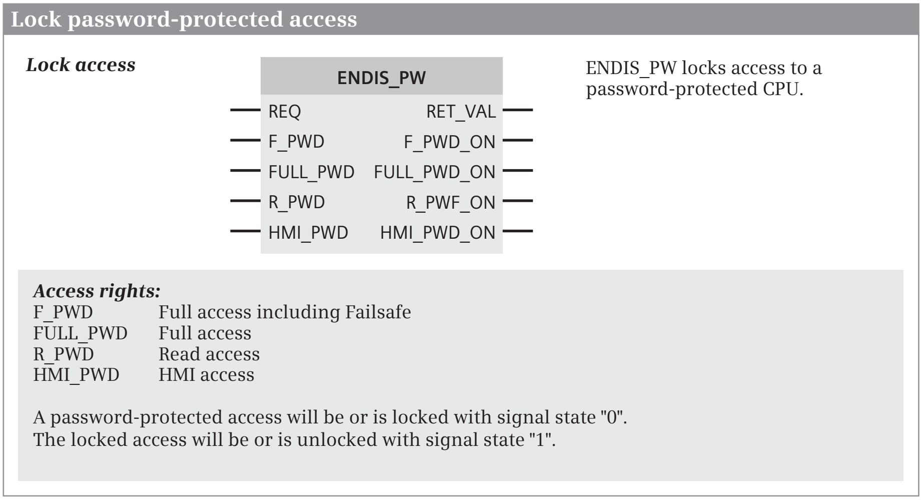
The blocks that are activated with the CPU display are in effect even after power is restored (Power ON) and with a change in the operating state. The access block via the CPU display competes with the access block via the user program with ENDIS_PW. The most recent setting is the effective one.

Blocking access with the user program

The ENDIS_PW function blocks access to a password-protected CPU during runtime. The block can be activated and deactivated separately for each level of protection (full access with and without Failsafe, read access, HMI access). Currently existing, password-protected access cannot be blocked using ENDIS_PW.

The blocks that are activated using ENDIS_PW remain in effect even after power is restored (Power ON) and when the operating state is changed. The blocks are deactivated if the mode switch is at STOP, or if they have been deactivated using the CPU display (the most recent setting is the effective one).

You can find ENDIS_PW in the program elements catalog under *Basic instructions* > *Program control operations*. Fig. 15.5 shows the graphical representation.



Lock password-protected access

Lock access

ENDIS_PW locks access to a password-protected CPU.

Access rights:

F_PWD	Full access including Failsafe
FULL_PWD	Full access
R_PWD	Read access
HMI_PWD	HMI access

A password-protected access will be or is locked with signal state "0".
The locked access will be or is unlocked with signal state "1".

Fig. 15.5 Locking access with ENDIS_PW

Signal state “1” at parameter REQ blocks a password-protected access to the CPU if the corresponding input parameter has signal state “0”. With signal state “1” at the corresponding input parameter, a block that was set using ENDIS_PW or the CPU display is canceled. The output parameters have signal state “0” when a block is set. Signal state “1” at an output parameter shows that a corresponding password-protected access is possible. Error information is output at parameter RET_VAL.

15.2.4 Working with online project data

Working without the offline project in online mode

You can also open the program in a CPU without the associated project.

Select the *Online & diagnostics* portal in the Portal view and then select *Accessible devices*. Set the LAN adapter (the PG/PC interface module) if applicable. Select the PLC station in the *Accessible devices* table and click on the *Show* button. If the programming device does not possess the matching network parameters, STEP 7 opens a dialog window to allow you to set these temporarily.

In the project view, the PLC station is displayed in the project tree under *Online access* and the used interface (module). Alternatively you can double-click under the used interface on *Update accessible devices*. The accessible PLC stations are then displayed as folders under the interface.

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 control panel, for example, or read out the diagnostics buffer in the diagnostic functions.

The *Program blocks* folder contains the online blocks. If you open it, STEP 7 loads the blocks into the folder. A block is opened by double-clicking it and the program in the block is displayed.

If you wish to edit, delete, or test an online block, you must create an offline project and transfer the online blocks to the project. New blocks can only be created, modified, deleted, or tested in the offline project data.

Uploading the online project data from the CPU

Uploading of online project data requires an offline project in the programming device. If the offline project matching the online project is not available, create an “empty” offline project and then copy the online project data into the project.

Use the *Project > New* command in the main menu to create a new project. Use the command *Add new device* to add a PLC station with a suitable CPU to the project. Set the right access data and activate the online mode.

To upload the online project data, select the PLC station in the project tree and then select the *Online > Upload from device* command from the main menu. In the *Preview for loading from device* dialog window, activate the checkbox *Continue* and start the upload by clicking on the *Upload from device* button. The *Program blocks*, *PLC* *tags*, and *PLC data types* folders present in the offline data management are deleted and replaced by the objects existing online.

15.2.5 Working with the memory card

A SIMATIC Memory Card for a CPU 1500 is an SD memory card (secure digital memory card) preformatted by Siemens.

The memory card is essential for operation of a CPU.

The memory card can be inserted or pulled at any time in the de-energized state. In STOP mode, the memory card can only be removed if no data traffic is taking place. Therefore end the communication between the programming device and the CPU first.

If you insert or remove a memory card while the CPU is activated, the CPU executes a memory reset and then goes into STOP mode.

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.

Formatting a memory card

You format a memory card in the CPU. Establish an online connection and double-click on *Online & diagnostics* in the project tree. In the dialog window, select *Functions* > *Format memory card* and then the *Format* button. All project data except for the IP address are deleted.

Formatting using Windows Explorer makes the memory card unusable in a CPU 1500. With the exception of the files "__LOG__" and "crdinfo.bin", deleting files and folders from the memory card is allowed.

Accessing a memory card in the card reader

Insert a memory card in the card reader on the programming device. Open the project tree and select the command *Project > Card Reader/USB memory* in the main menu. The *Card reader/USB memory* folder is opened in the project tree.

Setting the card type

You can use a memory card as a program card or as an update card. The load memory is located on the program card; using it, you can also transfer a project to the CPU, as a replacement for the online connection. You can transfer a firmware update to the CPU using the update card.

To set the type of card, insert the memory card into the programming device's card reader. In the project tree, open the *Card Reader/USB memory* 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 appears, select *Program* or *Update firmware* from the drop-down list in the *PLC card mode* field.

Transferring project data to the memory card

After the memory card has been set as the program card, copy the project data of the PLC station to the memory card, e.g. from the shortcut menu using *Copy* when a PLC station is selected and then *Paste* with the SD card that is then selected or by "dragging" the PLC station to the memory card by pressing and holding the mouse button. The project is compiled. After an error-free compilation, the *Load preview* window is displayed; continue the loading process with the checkbox *Continue* activated by clicking on the *Load* button. Clicking on the *Finish* button finishes the load process.

You can also load individual blocks or block groups to the memory card. Select the objects in the *Program blocks* folder of the PLC station and drag them to the *Program blocks* folder of the memory card. The prerequisites for loading to the memory card are checked and displayed in the *Load preview* window. If all of the prerequisites are fulfilled, continue loading by clicking on the *Load* button.

Blocks that only exist on the memory card are deleted during this.

Transferring project data from the memory card

It is only possible to transfer all of the blocks from a memory card to the project. Open the memory card in the project tree, select the *Program blocks* folder of the memory card and use the mouse to drag it to the PLC station. The prerequisites for loading from the memory card are checked and displayed in the *Preview for uploading from device* window. If all of the prerequisites are fulfilled, continue loading by clicking on the *Upload from device* button.

15.3 Working with blocks in online mode

15.3.1 Introduction

Once the project data has been transferred to the CPU, there are two versions of a block: the offline version in the project on the programming device and the online version in the user memory of the CPU. The online version of a code block is saved in two locations: in the load memory and in the work memory. The online version of a data block can either be located only in the load memory, only in the work memory, or in both.

The offline and online versions of a data block can have different contents, i.e. different values for the data tags, for these can be modified by the user program during runtime. If you program a data block, the data tags are assigned a start value depending on their type. As standard, the default value is the start value. With data type INT, for example, it is the value zero, with data type DATE it is the value D#1990-01-01. You can modify the start value according to your requirements.

The start values are present in the offline version of a data block. If the data block is transferred to the CPU, it is present with the start values in the load memory. The first time the data block is loaded, it is transferred with the start values to the work memory. The start values can be changed there via the user program. The values of the data tags in the work memory are referred to as actual values.

In the online mode, the project tree uses symbols on the blocks to show whether the offline and online versions differ: A green, filled circle indicates that both versions are the same, two blue/orange semicircles indicate that the two versions are different, and if one semicircle is not filled, the corresponding block version is missing (blue stands for offline, orange for online). The offline and online versions of a data block are the same if the data tags are the same. The start values in the offline version and the actual values in the online version may differ in this situation.

You can now change the offline version of a block and load the modified block into the CPU, where it replaces the online version of the original block. For a data block, you have various options for influencing the values of the data tags in the offline and online versions. When a block is deleted, the offline version is deleted. During the next loading process, the associated online version is deleted. Finally, the offline version of a block can be compared with the online version in the CPU or offline version from a different PLC station.

Caution! Reloading or deleting blocks during operation of the plant can cause serious damage to property or injury to persons if there are functional disturbances or program errors! Make sure that no dangerous situations can arise before you start the actions!

15.3.2 Changing and loading a block

Editing the online version of a block

The program of a block can only be changed in the offline version. If you wish to modify the online version, you must carry out the change in the offline version and subsequently transfer the block to the CPU.

If you change the block program in online mode, for example by adding a new scan to the logic operation during program testing, the program editor automatically switches to the offline version. After the change, transfer the changed offline version to the CPU.

Adding the online version of a block

Using the *Add new block* tool in the project tree, you can generate the offline version of a new block, even if online mode is switched on. Program the offline version of the block and the associated block call – if the new block is not an organization block – and then transfer the calling and called blocks to the CPU.

Deleting the online version of a block

If you select a block in the project tree and select the command *Delete* from the shortcut menu, the offline version of the block is deleted. The online version of the block is first retained in the user memory. The online version is then also deleted during the next loading process: During a loading process, all of the blocks in the user memory which are only available online are deleted.

In connection with the deleting of a code block, you should also delete its call, i.e. delete the call of the deleted block in the calling block, because otherwise an error will be reported during compilation. When deleting an organization block, you must also delete any assigned events.

Downloading a block into the user memory

To download into the CPU, select one or more blocks in the project tree in the *Program blocks* folder and then the command *Download to the device > Software (only changes)* from the shortcut menu. Alternatively, you can select *Online > Download to device* from the main menu. With the command *Online > Extended download to device...*, you can select the PLC station before downloading.

You can also download the code block you are processing at the moment from the program editor into the CPU. In the working window, click on a free spot in the workspace and select the command *Download to device* from the shortcut menu.

The block(s) are compiled. Downloading is aborted if errors occur during compilation. Only blocks which have been compiled without errors can be downloaded.

The envisaged actions are listed in the *Load preview* dialog. *Consistent download* means that all blocks affected by the change are downloaded. Set the desired actions in the *Action* column and click on the *Load* button.

Downloading in the STOP operating state

Any download process can be implemented if the CPU is in the STOP operating state. The blocks are then written to the load memory and the execution-relevant parts are transferred to the work memory. The configuration data, the entire user program, or more than the (CPU-specific) maximum number of blocks can only be loaded in the STOP operating state.

If the mode switch is in the RUN position, the checkbox *Start all* is provided in the *Load results* dialog window after loading. If the checkbox is activated, the RUN operating state is activated when the loading process is finished.

Only the changed blocks are written to the load memory using the command *Download to device > Software (only changes)* and the parts of the execution-relevant code blocks are transferred to the work memory.

If the block interface for a data block has not been changed or falls under memory reserve, the actual values of non-retentive data tags are overwritten with the start values from the load memory during the transition to the RUN operating state. The actual values of the retentive tags are retained, even if the start values have been changed.

If the change of the block interface of a data block exceeds the memory reserve, the actual values of all of the tags are overwritten (“re-initialized”) with the start values from the load memory.

Downloading in the RUN operating state

Individual blocks can be reloaded in the RUN operating state without having to put the CPU in the STOP operating state.

The *Download to device > Software (only changes)* command writes the changed blocks to the load memory. The execution-relevant parts of the code block are then transferred to the work memory and processed.

If the block interface for a data block has not been changed or falls under memory reserve, the actual values of data tags are not changed in the work memory.

If the change of the block interface of a data block exceeds the memory reserve, the actual values of all of the tags are overwritten ("re-initialized") with the start values from the load memory.

Overwriting actual values of data tags in the work memory

In the work memory of the CPU, either the actual values of non-retentive data tags, the actual values of all of the data tags, or the data tags marked as set values can be overwritten with the start values. The transferring of set values is described in Chapter 15.3.5 "Working with setpoints" on page 668.

To overwrite the actual values of non-retentive data tags, load the relevant data blocks (after a change to the start values, for example) using the command *Download to device > Software (only changes)*. During a transition from the STOP to the RUN operating state, the actual values of non-retentive data tags in the work memory are overwritten with the start values from the load memory. The actual values of retentive data tags are retained during this.

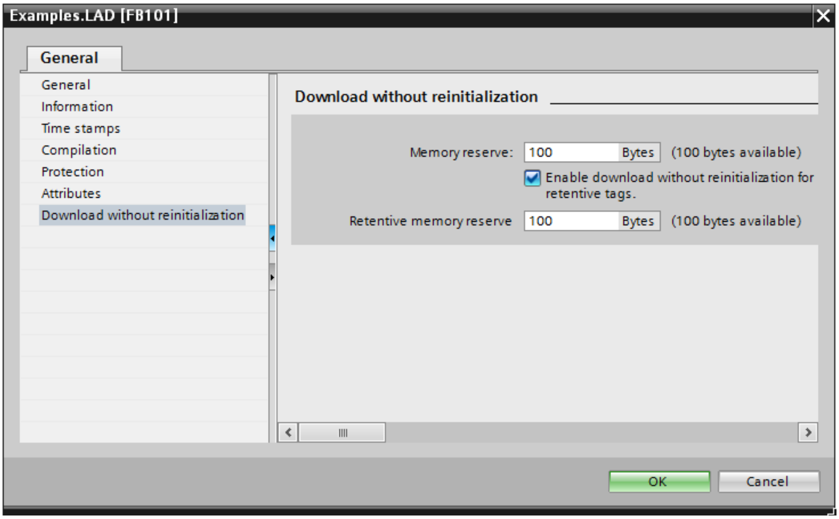
If the actual values of all of the data tags in the work memory are to be overwritten with the start values from the load memory, you must load the program using the command *Online > Download and reset PLC program*. Loading takes place in the STOP operating state. During the transition to the RUN operating state, the actual values of all of the data tags are overwritten.

15.3.3 Download without reinitialization

If the interface was changed on function blocks and data blocks, these blocks are "re-initialized" when they are loaded into the CPU, i.e. provided with the original data (the start values). This response can be disruptive, especially for data blocks with actual values which were collected during operation.

A CPU 1500 gives you the capability of expanding the interface of function blocks and data blocks and "reloading" the changed blocks in the RUN operating state without influencing existing data. You can apply "Download without reinitialization" for blocks with the *Optimized block access* attribute activated.

A CPU 1500 provides a so-called "memory reserve" for each function block and data block in which the later interface expansions are entered. By default, this memory reserve is 100 bytes in size and is not used at first. You can change the default setting in the main menu under *Options > Settings* and *PLC programming > General*.



Download without reinitialization

Memory reserve: 100 Bytes (100 bytes available)

☑ Enable download without reinitialization for retentive tags.

Retentive memory reserve 100 Bytes (100 bytes available)

Fig. 15.6 Setting the memory reserve for the download without reinitialization

You change the memory reserve of a block in its properties: In the *Download without reinitialization* section, enter the number of bytes. You can also define whether and how much memory reserve is to be provided in the retentive memory area (Fig. 15.6).

If you have compiled and loaded a block and then want to expand the interface without changing the loaded values, activate the memory reserve. To do this, open the block and select the *Download without reinitialization* icon in the toolbar of the working window. From this point on, each interface expansion is placed in the memory reserve and you can use the download without reinitialization.

The retentivity setting *Set in IDB* is not available for expanding the interface. You cannot expand an existing interface tag with a structured data type by adding components.

At a later point in time, for example, if the system is not operating, you can revise the user program and dissolve the memory reserve so that it will be available again with full capacity for future interface expansions.

You can enable the memory reserve for an individual block if you click again on the *Download without reinitialization* icon in the working window. After a confirmation prompt, the tags are moved from the memory reserve to the regular area. The block must be re-compiled and reloaded, this time with reinitialization again.

You enable the memory reserve for all of the blocks if you select the *Program blocks* folder in the project tree and select the command *Compile > Software (reset memory reserve)* from the shortcut menu. The tags from the memory reserve are moved to the regular area and the blocks are compiled. The interface tags are reinitialized during a subsequent loading action. The set memory reserve of the blocks is retained and continues to be active.

15.3.4 Uploading a block from the CPU

You can upload an individual block or all of the blocks from the user memory of the CPU into the offline project. As a prerequisite, open the project that belongs to the user program and start online mode.

To upload all of the blocks, select the *Program blocks* folder in the project tree. To upload an individual block, select the block in the *Program blocks* folder. A block is only uploaded if the online version differs from the offline version or if only the online version exists.

Then select the command *Upload from device* in the shortcut menu or the command *Online > Upload from device* in the main menu. In the *Upload preview* dialog window, messages are displayed, which you can answer with actions, for example by specifying overwriting existing objects or inserting them under other names (but with the same number). As soon as uploading is possible, the *Upload from device* button is activated. Click the *Upload from device* button to start uploading.

When a data block is uploaded, the start values that are in the load memory are transferred into the offline version as start values.

Uploading actual values

You can overwrite the start values of data tags in the offline version of a data block with the actual values from the work memory. To do so, you must establish an online connection and open the data block. Switch to monitoring mode by clicking on the *Monitor all* icon. The *Monitor value* column appears, showing the actual values of the data tags. Clicking on the *Snapshot of the monitored values* icon imports the current monitor values into the *Snapshot* column. Note that the monitor values can come from different program cycles.

To import the "frozen" actual values from the snapshot into the offline version of the data block as start values, you have the option of importing all of the values, only the set values, or only the retentivity values.

- ▷ To import an individual value, select the value in the *Snapshot* column and select the command *Copy* from the shortcut menu. Select the start value and choose the command *Insert*. The "frozen" actual value is imported as a start value. Repeat the procedure for other values as needed. In this way, you can also transfer successive values in one copy process.
- ▷ To import all of the values, click on the *Copy all values from the "Snapshot" column to the "Start value" column* icon. All of the "frozen" actual values are imported from the *Snapshot* column as start values.
- ▷ To import the setpoint values, click on the *Copy all setpoints from the "Snapshot" Column to the "Start value" Column* icon. The actual values marked as set values from the *Snapshot* column are imported as start values (see Chapter 15.3.5 "Working with setpoints" on page 668).
- ▷ To import the actual values of retentive data tags as start values, select the data block in the project tree and select the commands *Snapshot of the monitor values*

and *Apply snapshot values as start values > Only retain values* in the shortcut menu.

Start values in a write-protected data block are not changed.

Uploading actual values for several data blocks

To import the actual values of several data blocks as start values, select the data blocks in the project tree and select the command *Snapshot of the monitor values* in the shortcut menu. Then, select *Apply snapshot values as start values > Only setpoints values* or *Apply snapshot values as start values > Only retain values* in the shortcut menu.

15.3.5 Working with setpoints

Individual data tags can be marked as "setpoints". For tags marked in this way, the actual values can be overwritten with the start values while the program is running and the actual values can be imported from the user program into the offline version of the data block as start values.

Marking setpoints

To mark a tag as a setpoint, activate the checkbox in the *setpoint* column. Marking is possible

▷ in a PLC data type, if it is used

  – as a data type of a tag in the static local data of a function block,
  – as a data type of a tag in a global data block or
  – as a template for a type data block,

▷ in the static local data of a function block and

▷ in a global data block.

For a PLC data type, you can mark individual components as set values. The derived tag or the derived type data block adopt the marking.

For a tag with the data type STRUCT, you can only mark components as set values, not the entire tag. For a tag with the data type ARRAY, you can only mark the entire tag as a set value. If an array tag is comprised of structures (data type ARRAY OF STRUCT), you can mark the individual components of the first structure as set values, the components of the other structures adopt this marking.

Initializing setpoints

For data tags that are marked as setpoints, the actual values in the user memory can be overwritten with the start values from the offline block in the RUN operating state without influencing the actual values of the other data tags.

Establish an online connection and open the data block to initialize the setpoints. Change individual start values as required. Click on the *Initialize setpoint* icon. The start values in the offline version of the data block are transferred once to the work memory. This applies to both the retentive tags and the non-retentive tags.

Caution! Changing the data values during operation of the plant can cause serious damage to property or injury to persons if there are functional disturbances or program errors! Make sure that no dangerous situations can arise before you start the actions!

Importing setpoints as start values

For data tags that are marked as setpoints, the start values in the offline block can be overwritten with the actual values from the user memory

Establish an online connection and open the data block to overwrite the start values. Switch to monitoring mode by clicking on the *Monitor all* icon. The *Monitor value* column appears, showing the actual values of the data tags. Clicking on the *Snapshot of the monitored values* icon imports the current monitor values into the *Snapshot* column. Note that the monitor values can come from different program cycles.

To import the setpoints, click on the *Copy all setpoints from the "Snapshot" Column to the "Start value" Column* icon. The actual values marked as set values from the *Snapshot* column are imported as start values into the offline version of the data block.

To overwrite the start values for several data blocks, select the data block(s) in the project tree, select the command *Snapshot of the monitor values* from the shortcut menu, and then the command *Apply snapshot values as start values > Only setpoints*.

15.3.6 Comparing blocks

Compare editor and detailed comparison

The compare editor compares blocks, PLC tag tables, and PLC data types of a PLC station

- with the corresponding objects in the CPU (offline/online comparison), or
- with the corresponding objects in another PLC station (offline/offline comparison). The stations to be compared can come from the same project, from different projects or from a library.

If the time stamps of the two blocks agree, the compare editor assumes that the blocks are the same. Comments and block attributes are not considered in the offline/online comparison. Know-how protected blocks cannot be compared.

The compare editor gives you an overview of the compared objects. The detailed comparison shows the differences of an object.

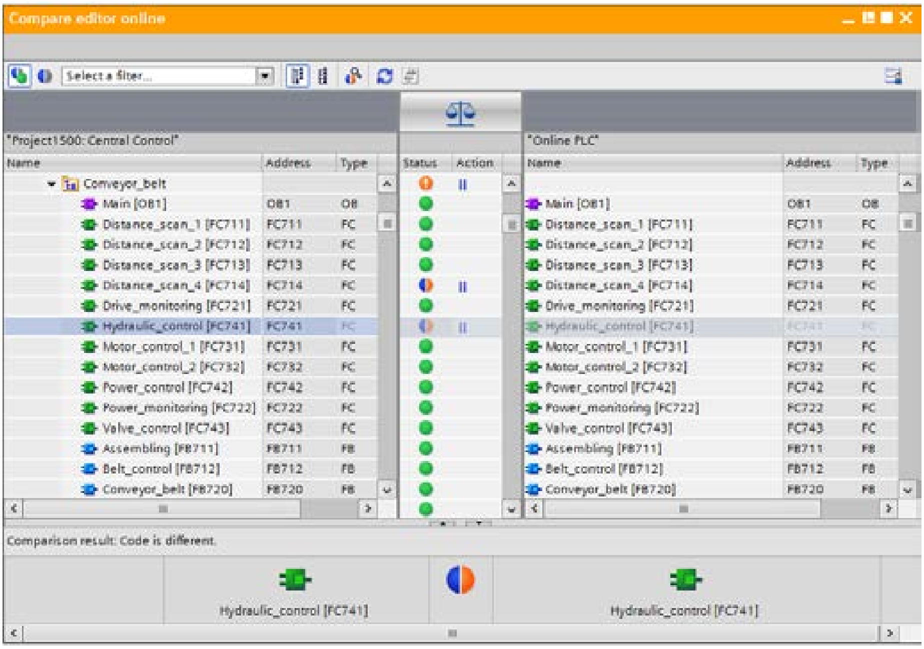
Offline/online comparison of blocks

An online connection to the CPU is required for the offline/online comparison. The comparison can be carried out in the STOP state or RUN mode.

To start the compare editor, select the PLC station in the project tree or select the *Compare* > *Offline/online* command from the shortcut menu or the *Tools* > *Compare* > *Offline/online* command in the main menu.

With an offline/online comparison, the blocks are assigned by means of the absolute address (block type and number). The compare editor displays all blocks and the comparison status in the working window.

As standard, blocks which have different offline and online versions are displayed (Fig. 15.7). You can control the display using the *Show only objects with differences* and *Show identical and different objects* icons. The icons *Display in hierarchical view* and *Display in flat view* switch between a display with the call structure and a list display.


Compare editor online

"Project1500: Central Control"					"Online PLC"		
Name	Address	Type	Status	Action	Name	Address	Type
Conveyor_belt			!	‖			
Main [OB1]	OB1	OB	●		Main [OB1]	OB1	OB
Distance_scan_1 [FC711]	FC711	FC	●		Distance_scan_1 [FC711]	FC711	FC
Distance_scan_2 [FC712]	FC712	FC	●		Distance_scan_2 [FC712]	FC712	FC
Distance_scan_3 [FC713]	FC713	FC	●		Distance_scan_3 [FC713]	FC713	FC
Distance_scan_4 [FC714]	FC714	FC	◐	‖	Distance_scan_4 [FC714]	FC714	FC
Drive_monitoring [FC721]	FC721	FC	●		Drive_monitoring [FC721]	FC721	FC
Hydraulic_control [FC741]	FC741	FC	◐	‖	Hydraulic_control [FC741]	FC741	FC
Motor_control_1 [FC731]	FC731	FC	●		Motor_control_1 [FC731]	FC731	FC
Motor_control_2 [FC732]	FC732	FC	●		Motor_control_2 [FC732]	FC732	FC
Power_control [FC742]	FC742	FC	●		Power_control [FC742]	FC742	FC
Power_monitoring [FC722]	FC722	FC	●		Power_monitoring [FC722]	FC722	FC
Valve_control [FC743]	FC743	FC	●		Valve_control [FC743]	FC743	FC
Assembling [FB711]	FB711	FB	●		Assembling [FB711]	FB711	FB
Belt_control [FB712]	FB712	FB	●		Belt_control [FB712]	FB712	FB
Conveyor_belt [FB720]	FB720	FB	●		Conveyor_belt [FB720]	FB720	FB

Comparison result: Code is different.

Hydraulic_control [FC741] ◐ Hydraulic_control [FC741]

Fig. 15.7 Example of offline/online comparison of blocks

A green, filled circle indicates that the offline and online versions are identical. Blue-orange semicircles indicate that the object's offline and online versions differ. If one semicircle is not filled, the corresponding version is missing (left side or blue stands for offline, right side or orange for online). An exclamation mark in an orange circle indicates an object with differences in the identified folder.

In the *Action* column, you can select an action from a drop-down list for different objects, for example *Download to device* or *Upload from device*. Clicking on the *Execute actions* icon in the toolbar starts the set actions. The comparison is carried out again by using the *Refresh the view* icon. You can only carry out one offline/online comparison at a time.

Offline/offline comparison of blocks

For an offline/offline comparison, the user can compare PLC stations which are located in the same project, in different projects, or in a library. Edit the opened project in the project tree. Open additional projects as reference projects (see Chapter 1.3.4 “Working with reference projects” on page 45). Note that objects in reference projects cannot be added, changed or deleted.

To start the compare editor, select a PLC station in the project tree and select the *Compare > Offline/offline* command from the shortcut menu or the *Tools > Compare > Offline/offline* command in the main menu. The objects of the selected PLC station are displayed in the left half of the working window. Using the mouse, drag the PLC station that is to be compared into the title bar on the right side (labeled “Insert here to add a new object or replace an existing one”). You can move other PLC stations into the title bar on one of the two sides at any time in order to carry out further comparisons.

For the automatic offline/offline comparison, the blocks are assigned based on the symbolic address. For the manual offline/offline comparison, select the blocks to be compared. To switch between automatic and manual comparisons, click on the button with the scales in the title bar. For the automatic comparison (the “scales” button is white), all of the blocks are compared. For the manual comparison (the “scales” button is gray), only the two blocks that are selected are compared.

The further procedure is as with the offline/online comparison. The comparison icons are now blue for objects of the current project and gray for objects of the selected project. You can only carry out one offline/offline comparison at a time.

Detailed comparison

You can start a detailed comparison for a block. The compared versions of the block are then displayed next to each other and the differences highlighted.

To start the detailed comparison, select a block in the compare editor and activate the *Start detailed comparison* icon or select the *Start detailed comparison* command from the shortcut menu.

For code blocks, you can use icons in the compare editor's toolbar to navigate to the first, preceding, subsequent, or last difference. If the *Synchronize scrolling between editors* icon is activated, the corresponding networks remain visible in parallel when scrolling vertically. If networks are missing or if the sequence is interchanged, the compare editor inserts “pseudo networks” with the heading *No corresponding network found*. These networks cannot be edited.

You can modify the offline version of the open block in the current project. A new comparison is carried out using the *Update comparison results* icon.

15.4 Hardware diagnostics

The hardware diagnostics detects and signals module faults, e.g. failure of the load voltage or an open-circuit on signal modules.

The modules with diagnostic capability distinguish between parameterizable and non-parameterizable diagnosis events. In the case of the parameterizable diagnosis events, the message is only output if you have enabled the diagnostic function in the parameter settings. The non-parameterizable diagnosis events are always signaled irrespective of a diagnostics enable.

This chapter describes the diagnostics options offered by the programming device in online mode. Chapter 5.9 “Diagnostics in the user program” on page 225 describes how you can react to a diagnosis event in the program.

When a diagnosis event occurs:

- ▷ An error LED lights up on the CPU
- ▷ The diagnosis event is passed on to the CPU's operating system
- ▷ A diagnostic interrupt is triggered if you have enabled this in the parameter settings (the diagnostic interrupts are disabled by default)

All diagnostic events signaled to the CPU's operating system are entered into a diagnostics buffer in the sequence of their occurrence with date and time. 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.

15.4.1 Status displays on the modules

The status displays on the modules signal the operating state and can help to localize a fault. For this purpose, each module has LEDs with colors that correspond to their functions.

Status displays on the CPU

The CPU has three status LEDs on the upper edge of the front panel, which signal the operating state with various light configurations (Table 15.1).

The CPU display shows the operating state in the main menu. The *Diagnostics* menu signals when there are new diagnostics alarms and entries in the diagnostics buffer. The *Modules* menu contains such things as order data and the module status.

Status displays on the signal modules

Every input/output channel of a digital and analog module has a green channel LED CHx to indicate whether voltage is present on the input or output channel. It is thus possible to check the wiring from the sensor to the digital input channel or from the digital output channel to the actuator. A green RUN LED indicates the module is

Table 15.1 Light configurations of the status LEDs of a CPU 1500

RUN/STOP LED green/yellow	ERROR LED red	MAINT LED yellow	Meaning
off	off	off	No or insufficient power supply.
off	flashes red	off	An error occurred.
illuminates green	off	off	RUN operating state.
illuminates green	flashes red	off	A diagnosis event occurred.
illuminates green	off	illuminates yellow	A maintenance request is present.
illuminates green	off	flashes yellow	Maintenance is required or a firmware update has been successfully completed.
illuminates yellow	off	off	STOP operating state.
illuminates yellow	flashes red	flashes yellow	The program on the memory card causes an error – or – the CPU is defective.
flashes yellow	off	off	The CPU carries out internal activities in STOP or the user program is being loaded.
flashes yellow/green	off	off	STARTUP operating state.
flashes yellow/green	flashes red	flashes yellow	The CPU is booting or LED and flash test.

operationally ready. A red ERROR LED flashes to indicate an error on the module. A green PWR LED indicates the load voltage is present on the module.

Status displays on power supply modules, technology modules and communication modules

The PS, TM and CM modules indicate the operating state of the module, an error or a maintenance requirement by means of a green RUN LED, a red ERROR LED and a yellow MAINT LED. Depending on the function of the module, there are additional LEDs at the inputs and outputs for status displays.

15.4.2 Diagnostic information

The diagnostic information is displayed in the working window when the programming device is switched to online mode using the *Online & diagnostics* command. The following diagnostic information is then available:

- General: Module designations, module and vendor information.
- Diagnostics status: Status information of selected module, e.g. *Module exists and OK*, differences between configured and existing modules.

- Cycle time: Display of preset or configured minimum cycle time and cycle (monitoring) time and – in RUN mode – the cycle time diagram and the shortest, current, and longest cycle (processing) times.
- Memory: Display of the memory utilization for the load memory, the work memory separated by program code and data, and the retentivity memory.
- Diagnostics buffer: Display of diagnostics buffer content.
- Display: Display of order number and manufacturer data.
- Interface: Properties of the corresponding interface.

Cycle times and resources are displayed in parallel in the online tools. Under *Online access*, the online connection can be confirmed with the flash test, the online connection can be activated or disconnected, and the receipt of diagnostics alarms in the programming device can be enabled or blocked. The diagnostic functions are located in the same window (see Chapter 15.4.4 “Diagnostic functions” on page 675).

15.4.3 Diagnostics buffer

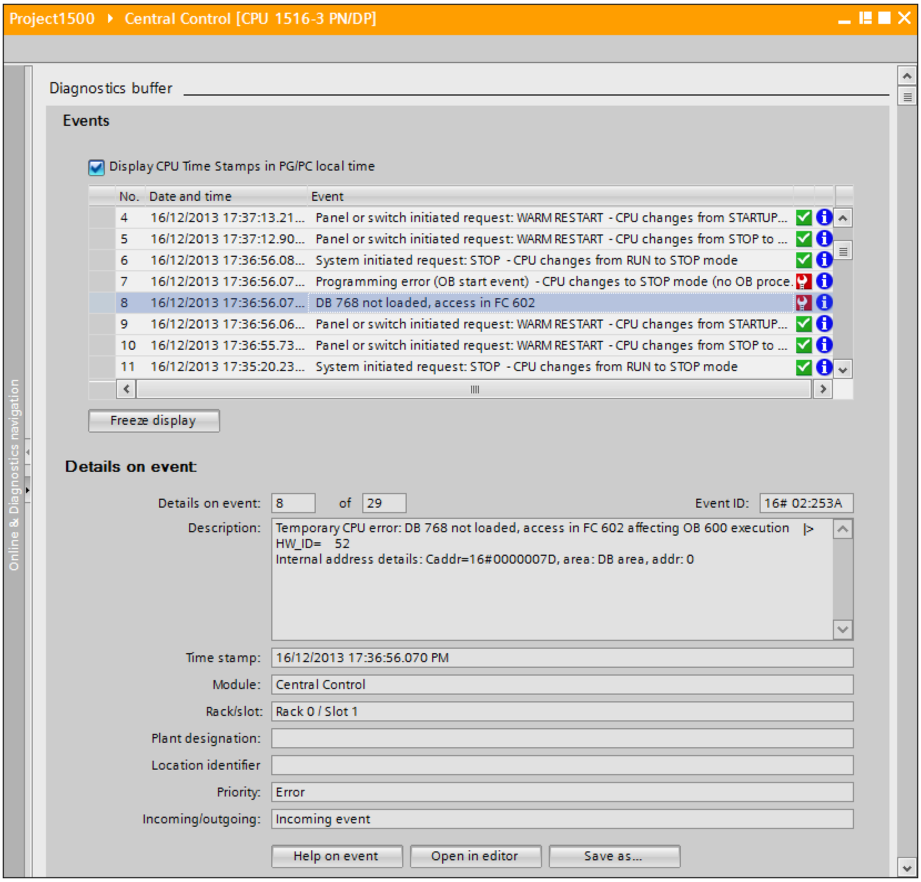
The diagnostics buffer contains the faults detected by the CPU and the modules with diagnostic capability, the triggered hardware and diagnostic interrupts, and the changes in CPU modes in the sequence of occurrence. The diagnostics buffer is designed as a ring buffer: when it is full, the oldest entries are overwritten. The entries can only be erased by resetting the CPU to its factory settings (Fig. 15.8).

The most recent event is present in the first line in the diagnostics buffer. A diagnostics buffer 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 CPU's time is up-to-date. An event ID can be called up for every event. This is an identification which exactly specifies the event. Select a line and the event ID will be displayed on the right underneath the table.

The *Freeze display* button stops the display of entries; you can then call up information on a specific event or study the sequence of displayed events at your own rate. Clicking on the button again (now labeled: *Cancel freeze*) changes to the updated display. You can use the *Save as* ... button to save the contents of the diagnostics buffer as a text file.

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.

In the *Settings* area (not shown in the figure), you can set a filter for the events to be displayed and import this filter as standard for future display of the diagnostics buffer.



Project1500 ▸ Central Control [CPU 1516-3 PN/DP]

Diagnostics buffer

Events

☑ Display CPU Time Stamps in PG/PC local time

No.	Date and time	Event		
4	16/12/2013 17:37:13.21...	Panel or switch initiated request: WARM RESTART - CPU changes from STARTUP...	✓	ℹ
5	16/12/2013 17:37:12.90...	Panel or switch initiated request: WARM RESTART - CPU changes from STOP to ...	✓	ℹ
6	16/12/2013 17:36:56.08...	System initiated request: STOP - CPU changes from RUN to STOP mode	✓	ℹ
7	16/12/2013 17:36:56.07...	Programming error (OB start event) - CPU changes to STOP mode (no OB proce...	🟥	ℹ
8	16/12/2013 17:36:56.07...	DB 768 not loaded, access in FC 602	🟥	ℹ
9	16/12/2013 17:36:56.06...	Panel or switch initiated request: WARM RESTART - CPU changes from STARTUP...	✓	ℹ
10	16/12/2013 17:36:55.73...	Panel or switch initiated request: WARM RESTART - CPU changes from STOP to ...	✓	ℹ
11	16/12/2013 17:35:20.23...	System initiated request: STOP - CPU changes from RUN to STOP mode	✓	ℹ

Details on event:

Details on event: 8 of 29 Event ID: 16# 02:253A

Description: Temporary CPU error: DB 768 not loaded, access in FC 602 affecting OB 600 execution
HW_ID= 52
Internal address details: Caddr=16#0000007D, area: DB area, addr: 0

Time stamp: 16/12/2013 17:36:56.070 PM

Module: Central Control

Rack/slot: Rack 0 / Slot 1

Plant designation:

Location identifier

Priority: Error

Incoming/outgoing: Incoming event

Fig. 15.8 Example of display of diagnostics buffer

15.4.4 Diagnostic functions

The diagnostic functions are displayed in the working window when the programming device is switched to online mode using the *Online & diagnostics* command. The following functions are then available:

- Set time: Display of programming device and module time, setting of real-time clock on CPU, time synchronization.
- Assign IP address: Setting of the IP address, subnet mask, router address.
- Reset to factory settings: The user memory, operand areas, diagnostics buffer and– upon request– IP address are deleted. All parameters including the time are reset to the default settings.
- Format memory card: Formats the memory card for use in a CPU 1500.

- Assign name: Enter or change the PROFINET device name.
- Firmware update: Display of current firmware of the CPU and CPU display and preparations for updating the firmware.

15.4.5 Online tools

You can use the *Online & diagnostics* command from the project tree in the task window to start the task card with the online tools.

CPU operator panel

The CPU operator panel shows the current status of the LED on the front side of the CPU. The RUN and STOP buttons can be used to set the CPU – following confirmation – to the corresponding state. A pressed (bright) button symbolizes the currently set state. The CPU can only be switched to RUN mode using the operator panel if the mode switch on the CPU is at RUN and if no faults which prevent starting are present.

The MRES button is used to trigger a memory reset. A memory reset can only be carried out in the STOP state. During the memory reset, the contents of the work memory and all operand areas are deleted. The contents of the load memory are retained. 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 the IP address remain uninfluenced.

The existing (logic) connections to the CPU are cleared. Following a CPU memory reset, the programming device must switch to online mode again using the *Online & diagnostics* or *Go online* command.

Cycle time

Cycle time shows the shortest, current, and longest cycle (processing) times in milliseconds and presents these graphically.

Memory

Memory displays the utilization of the load, work and retain memories in percent as bars.

15.4.6 Further diagnostic information via the programming device

Diagnostics icons in the device and network views

In online mode, the device configuration editor shows the device status of every PLC station connected online by means of diagnostics icons in the device or network view. For example, a green tick indicates that the station does not signal any faults. The operating state is indicated by a colored square: green for RUN and yellow for STOP.

Diagnostics icons in the project tree

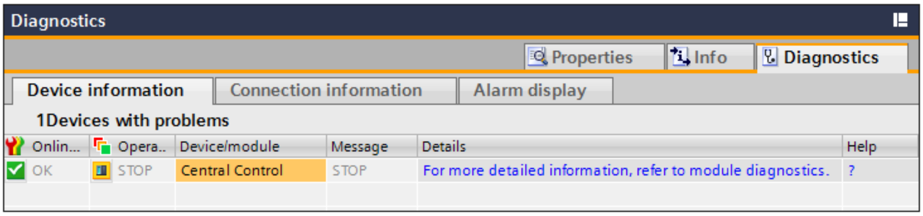
In online mode, diagnostics icons are also shown in the project tree. If everything is OK in the PLC station, the name is followed by a white tick on a green background.

The project tree also shows the result of the comparison between offline and online project data. If an orange circle with exclamation mark is shown, the folder contains objects which differ in the online and offline versions. The following identifications apply to individual objects:

- Green, filled circle: no difference between online version and offline version
- Blue/orange semicircle: the online version and offline versions of the object are different
- Blue/orange semicircle, right half (orange) filled: only the online object is present
- Blue/orange semicircle, left half (blue) filled: only the offline object is present

Device information in the 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. 15.9). Via the link in the *Details* column you can access the *Go online* dialog or the online and diagnostics view of the faulty device.



1Devices with problems

Onlin...	Opera..	Device/module	Message	Details	Help
✓ OK	STOP	Central Control	STOP	For more detailed information, refer to module diagnostics.	?

Fig. 15.9 *Diagnostics* tab in the inspector window

15.5 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
- ▷ 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.

You can use the S7-PLCSIM option software to simulate a CPU in the programming device and thus test your program without additional hardware (see Chapter 18.6 “Simulation with PLCSIM” on page 819).

15.5.1 Defining the call environment

If you wish to test the user program at a specific position in the program status, you open that position of the program in the working window and switch on the test function. If the program position to be tested is in a block which is called repeatedly in the user program, you must define the block call you wish to test.

You set the call environment in the tasks window on the *Testing* task card in the *Call environment* pallet. If the condition applies, the program status that is located in the specified call of the block is recorded. You can make the following settings when you click the *Change* button:

- ▷ *No trigger applied*
  Default option; for several block calls, the program status of any call is displayed.
- ▷ *Instance data block*
  The condition is fulfilled if the function block is called with the specified instance data block.
- ▷ *Call environment*
  The condition is satisfied if the call of the code block is made from the specified block or from a specific path.

15.5.2 Testing with program status

The program status shows the program execution during runtime. You can monitor the current signal state 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!

Please note that the program status requires considerable resources, which means that, under certain circumstances, the test function will only be carried out to a limited extent.

Switching the program status on and off

To switch on the program status, open the block to be monitored, move on to the program position you wish to test, and click on the *Monitoring on/off* icon in the toolbar of the working window.

If an online connection to the CPU has not yet been established, STEP 7 searches for accessible devices. If necessary, set the LAN adapter 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* icon in the toolbar. You will be asked whether the online connection which was created when switching on the program status is to be canceled. If you click on the *No* button, the program status will be exited but the online connection remains established.

Display format with digital tags

The display format of digital tags is set as standard to *Automatic*, but you can change it by selecting the digital tag and then *Modify > Display format >* ... from the shortcut menu. ... > *Automatic*, ... > *Decimal*, ... > *Hexadecimal* and ... > *Floating-point* are available.

In the case of LAD and FBD you set the display format for the complete network if you click with the right mouse button on a free space in the network and then select *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 as is the case with inputs which receive their signal state from the peripheral input channel during the automatic updating of the process image.

Select the tag and then the command *Modify > Modify to 0* 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 command *Modify > Modify operand...* from the shortcut menu and specify the desired value.

Block calls in the program status (LAD, FBD)

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 move on to the called block and continue the program status there: Select the block call and then the *Open and monitor* command from the shortcut menu. The program status then changes to the called block.

Program status in LAD representation

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. 15.10).

Program elements with unknown status or those which are not processed are identified by continuous gray lines. Tags shown in black mean that the displayed value is from the current monitoring cycle, those in gray display a value from a previously processed cycle.

You can determine at which position the program status is to be executed: Select the program element or tag and then the *Modify > Monitor from here* command from the shortcut menu. The *Modify > Monitor selection* command from the shortcut menu means that only the selected program element is monitored.

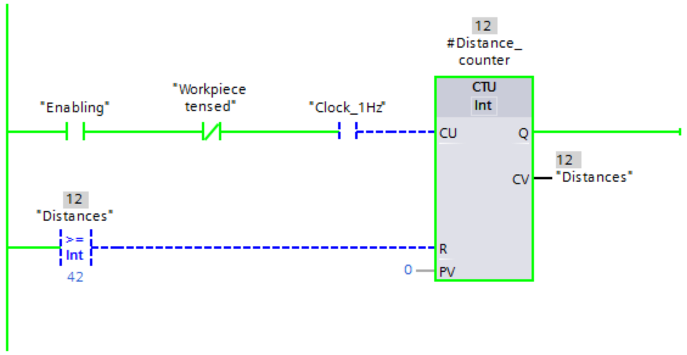


Fig. 15.10 Program status in LAD representation

Program status in FBD representation

In the FBD program status, green continuous lines are used to identify the boxes of the binary program elements and the connections if they have signal state “1” and blue dashed lines if they have signal state “0” (Fig. 15.11). In addition to the colored identification, the signal state (TRUE or FALSE) is displayed for the binary inputs.

Program elements with unknown status or those which are not processed are identified by continuous gray lines. Tags shown in black mean that the displayed value is from the current monitoring cycle, those in gray display a value from a previously processed cycle.

You can determine at which position the program status is to be executed: Select the program element or tag and then the *Modify* > *Monitor from here* command from the shortcut menu. The *Modify* > *Monitor selection* command from the shortcut menu means that only the selected program element is monitored.

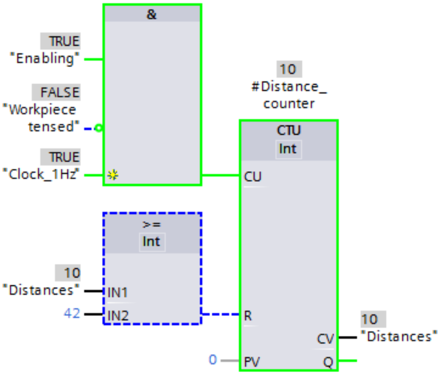


Fig. 15.11 Program status in FBD representation

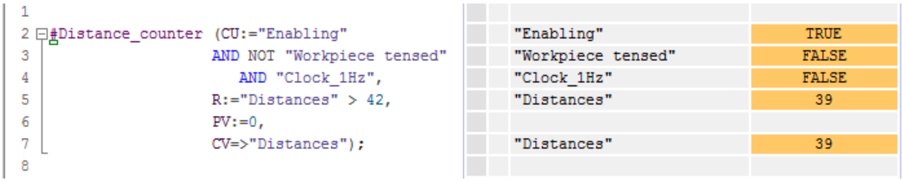
Program status in SCL representation

The program status is shown in tabular form next to the statements. The line in the table contains the name and value of the (first) tag in the statement line. If the statement line contains several tags, a table with all tags is displayed when you position the cursor in the statement line.

If the line contains one of the IF, WHILE, or REPEAT statements, the result of the condition (TRUE, FALSE) is shown in the line.

You can use the *Absolute/symbolic operands* icon to select the displayed type of addressing. If the tag name is shown in gray, the corresponding program is not processed.

If no value can be shown for a tag or event, the table contains three question marks on a yellow background in the *Value* column. In this case, activate the *Create ex-*



```
#Distance_counter (CU:="Enabling"
                   AND NOT "Workpiece tensed"
                       AND "Clock_1Hz",
                   R:="Distances" > 42,
                   PV:=0,
                   CV=>"Distances");
```

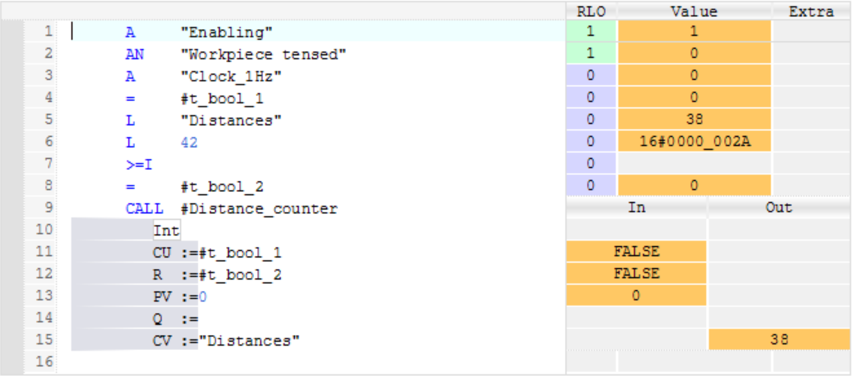
"Enabling"	TRUE
"Workpiece tensed"	FALSE
"Clock_1Hz"	FALSE
"Distances"	39
"Distances"	39

Fig. 15.12 Program status in SCL representation

panded status information Checkbox under *Options > Settings and PLC programming > SCL*, compile the block, and load the block again into the CPU (Fig. 15.12).

Program status in STL representation

The program status is shown in tabular form next to the statements so that the tag value can be read for each statement line. The RLO column shows the result of logic operation: “0” has a purple background and “1” has a green background. The *Value* column shows the current status or the current value of the operand. The *Extra* column shows additional information, if applicable (Fig. 15.13).



		RLO	Value	Extra
A	"Enabling"	1	1	
AN	"Workpiece tensed"	1	0	
A	"Clock_1Hz"	0	0	
=	#t_bool_1	0	0	
L	"Distances"	0	38	
L	42	0	16#0000_002A	
>=I		0		
=	#t_bool_2	0	0	

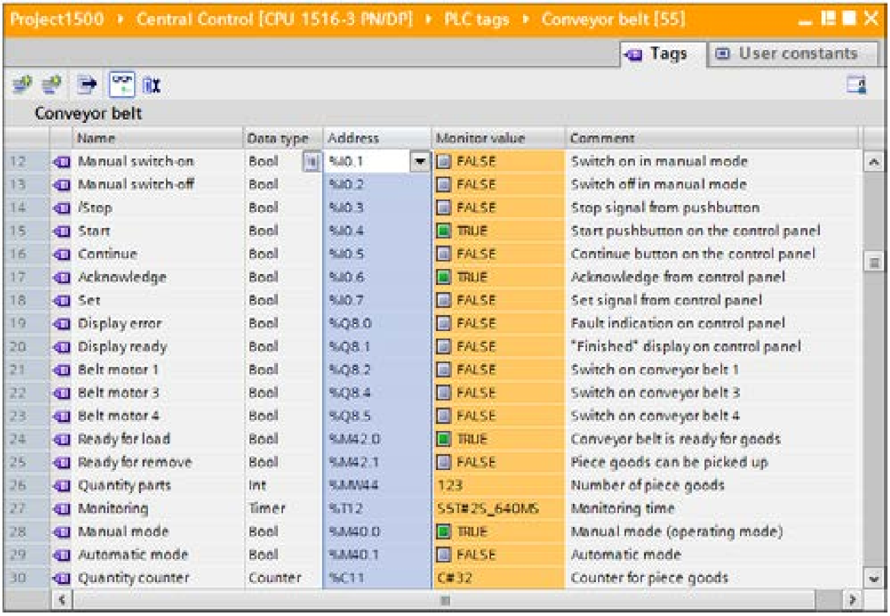
		In	Out
CALL	#Distance_counter		
	Int		
	CU :=#t_bool_1	FALSE	
	R :=#t_bool_2	FALSE	
	PV :=0	0	
	Q :=		
	CV :="Distances"		38

Fig. 15.13 Program status in STL representation

You can use the *Absolute/symbolic operands* icon to select the displayed type of addressing.

15.5.3 Monitoring of PLC tags

To monitor using the tag table, double-click on the PLC tag table in the project tree. Click on 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.



Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ PLC tags ▸ Conveyor belt [55]
Tags
User constants

Conveyor belt

	Name	Data type	Address	Monitor value	Comment
12	Manual switch-on	Bool	%I0.1	FALSE	Switch on in manual mode
13	Manual switch-off	Bool	%I0.2	FALSE	Switch off in manual mode
14	/Stop	Bool	%I0.3	FALSE	Stop signal from pushbutton
15	Start	Bool	%I0.4	TRUE	Start pushbutton on the control panel
16	Continue	Bool	%I0.5	FALSE	Continue button on the control panel
17	Acknowledge	Bool	%I0.6	TRUE	Acknowledge from control panel
18	Set	Bool	%I0.7	FALSE	Set signal from control panel
19	Display error	Bool	%Q8.0	FALSE	Fault indication on control panel
20	Display ready	Bool	%Q8.1	FALSE	"Finished" display on control panel
21	Belt motor 1	Bool	%Q8.2	FALSE	Switch on conveyor belt 1
22	Belt motor 3	Bool	%Q8.4	FALSE	Switch on conveyor belt 3
23	Belt motor 4	Bool	%Q8.5	FALSE	Switch on conveyor belt 4
24	Ready for load	Bool	%M42.0	TRUE	Conveyor belt is ready for goods
25	Ready for remove	Bool	%M42.1	FALSE	Piece goods can be picked up
26	Quantity parts	Int	%MW44	123	Number of piece goods
27	Monitoring	Timer	%T12	S5T#2S_640MS	Monitoring time
28	Manual mode	Bool	%M40.0	TRUE	Manual mode (operating mode)
29	Automatic mode	Bool	%M40.1	FALSE	Automatic mode
30	Quantity counter	Counter	%C11	C#32	Counter for piece goods

Fig. 15.14 Monitoring with the PLC tag table

The current time and count values are displayed with the SIMATIC timer functions (data type TIMER) and the SIMATIC counter functions (data type COUNTER).

Fig. 15.14 shows an online PLC tag table where monitoring is activated. The *Retain*, *Accessible from HMI*, and *Visible in HMI* columns which are not required are hidden.

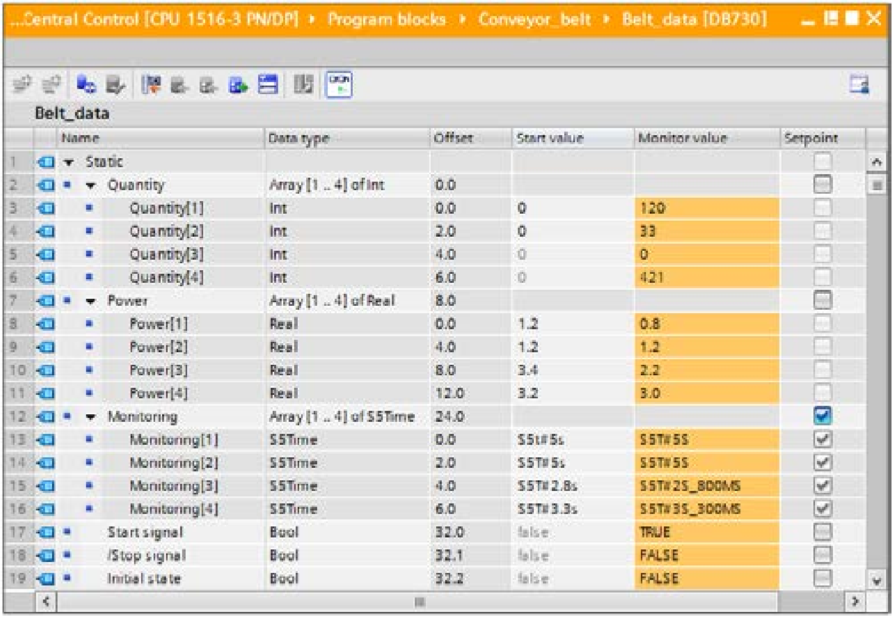
15.5.4 Monitoring of data tags

To monitor the data tags, you 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. A further click on the *Monitor all* icon exits monitoring mode.

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. Chapter "Uploading actual values" on page 667 describes how the actual values displayed in this way can be imported as start values.

Fig. 15.15 shows the monitoring function for a data block in expanded mode. The combined tags are "opened" so that the individual values can be monitored. Columns which are not required, for example *Default value*, *Retain*, and *Visible in HMI*, can be hidden. The *Snapshot* column is not shown in the figure.

Please note that tag values displayed in monitoring mode can originate from different program cycles.



...Central Control [CPU 1516-3 PN/DP] ▸ Program blocks ▸ Conveyor_belt ▸ Belt_data [DB730]

Belt_data

	Name	Data type	Offset	Start value	Monitor value	Setpoint
1	▼ Static					□
2	▼ Quantity	Array [1 .. 4] of Int	0.0			□
3	Quantity[1]	Int	0.0	0	120	□
4	Quantity[2]	Int	2.0	0	33	□
5	Quantity[3]	Int	4.0	0	0	□
6	Quantity[4]	Int	6.0	0	421	□
7	▼ Power	Array [1 .. 4] of Real	8.0			□
8	Power[1]	Real	0.0	1.2	0.8	□
9	Power[2]	Real	4.0	1.2	1.2	□
10	Power[3]	Real	8.0	3.4	2.2	□
11	Power[4]	Real	12.0	3.2	3.0	□
12	▼ Monitoring	Array [1 .. 4] of S5Time	24.0			☑
13	Monitoring[1]	S5Time	0.0	S5t#5s	S5T#5S	☑
14	Monitoring[2]	S5Time	2.0	S5T#5s	S5T#5S	☑
15	Monitoring[3]	S5Time	4.0	S5T#2.8s	S5T#2S_800MS	☑
16	Monitoring[4]	S5Time	6.0	S5T#3.3s	S5T#3S_300MS	☑
17	Start signal	Bool	32.0	false	TRUE	□
18	/Stop signal	Bool	32.1	false	FALSE	□
19	Initial state	Bool	32.2	false	FALSE	□

Fig. 15.15 Example of monitoring of data tags

15.5.5 Testing with watch tables

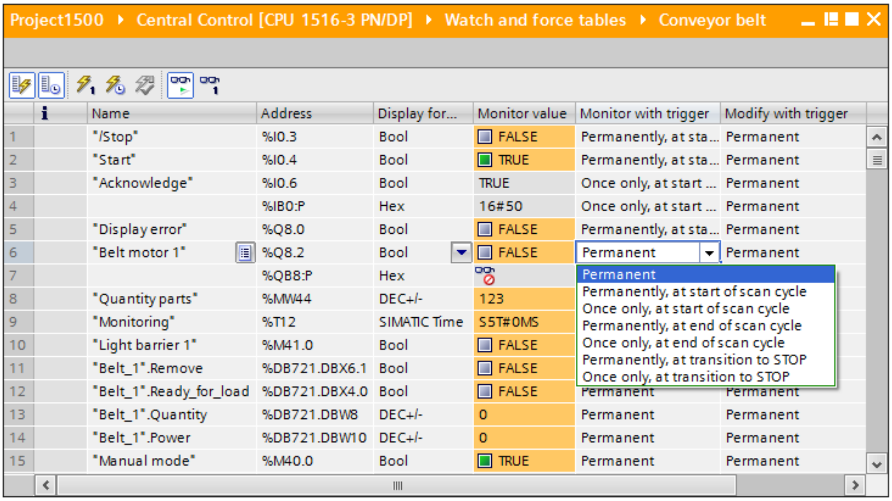
The watch tables contain tags whose values can be monitored and controlled during runtime. The tags can be combined in any manner so that a specially tailored watch table can be created for each test case. You can call the test functions in the shortcut menu or using the icons in the toolbar of the working window shown in Fig. 15.17 on page 686.

Tags from data blocks can be used in watch tables as well as tags from the areas: peripheral inputs/outputs, inputs, outputs, and bit memories. The current time and count values are displayed with the SIMATIC timer functions (data type TIMER) and the SIMATIC counter functions (data type COUNTER).

Creating a watch table

Underneath a PLC station in the project tree there is the *Watch and force tables* folder with the watch tables. 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 format from a drop-down list. You can enter a short explanatory text for each tag in the comment column.



Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ Watch and force tables ▸ Conveyor belt

	i	Name	Address	Display for...	Monitor value	Monitor with trigger	Modify with trigger
1		"/Stop"	%I0.3	Bool	□ FALSE	Permanently, at sta...	Permanent
2		"Start"	%I0.4	Bool	🟩 TRUE	Permanently, at sta...	Permanent
3		"Acknowledge"	%I0.6	Bool	TRUE	Once only, at start ...	Permanent
4			%IB0:P	Hex	16#50	Once only, at start ...	Permanent
5		"Display error"	%Q8.0	Bool	□ FALSE	Permanently, at sta...	Permanent
6		"Belt motor 1"	%Q8.2	Bool	□ FALSE	Permanent	Permanent
7			%QB8:P	Hex			
8		"Quantity parts"	%MW44	DEC+/-	123		
9		"Monitoring"	%T12	SIMATIC Time	S5T#0MS		
10		"Light barrier 1"	%M41.0	Bool	□ FALSE		
11		"Belt_1".Remove	%DB721.DBX6.1	Bool	□ FALSE		
12		"Belt_1".Ready_for_load	%DB721.DBX4.0	Bool	□ FALSE	Permanent	Permanent
13		"Belt_1".Quantity	%DB721.DBW8	DEC+/-	0	Permanent	Permanent
14		"Belt_1".Power	%DB721.DBW10	DEC+/-	0	Permanent	Permanent
15		"Manual mode"	%M40.0	Bool	🟩 TRUE	Permanent	Permanent

Permanent
Permanently, at start of scan cycle
Once only, at start of scan cycle
Permanently, at end of scan cycle
Once only, at end of scan cycle
Permanently, at transition to STOP
Once only, at transition to STOP

Fig. 15.16 Example of monitoring of tags in expanded mode

The tags entered with names must previously have been defined in the PLC tag table or in a data block. You can also enter the memory location (absolute address) in the *Address* column.

Fig. 15.16 shows monitoring in expanded mode. In the *Monitor with trigger* column, the possible settings are "opened".

Monitoring and modifying with triggers

The watch tables permit specification of the monitoring and control time. The following can be selected:

- Permanent
  In each program cycle, the inputs are monitored and controlled at the start of the cycle prior to processing of the main program and the outputs at the end of the cycle following processing of the main program.

- Permanently, at start of scan cycle
  In each program cycle, the tags are monitored and controlled prior to processing of the main program (meaningful for inputs or tags which control functions).

- Once only, at start of scan cycle
  The tags are monitored and controlled once prior to processing of the main program (meaningful for inputs or tags which control functions).

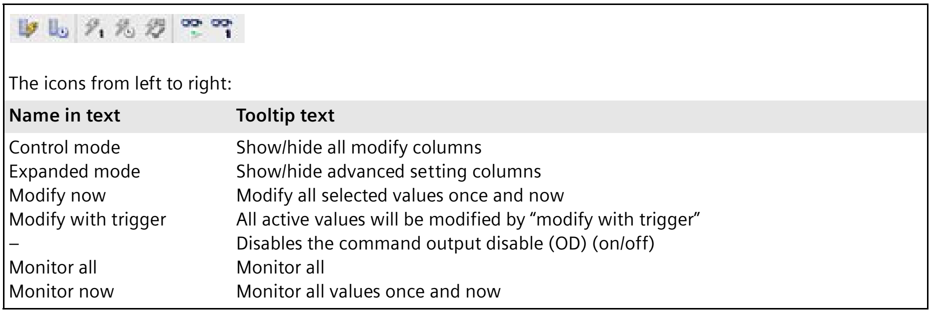
- Permanently, at end of scan cycle
  In each program cycle, the tags are monitored and controlled following processing of the main program (meaningful for outputs or tags which are controlled by functions).

- ▷ Once only, at end of scan cycle
  The tags are monitored and controlled once following processing of the main program (meaningful for outputs or tags which are controlled by functions).
- ▷ Permanently, at transition to STOP
  The tags are monitored and controlled permanently at the transition to the STOP state.
- ▷ Once only, at transition to STOP
  The tags are monitored and controlled once at the transition to the STOP state.

It is also possible to control tags using the *Online > Modify > Modify now* command in the main menu or the *Modify > Modify now* command in the shortcut menu. The selected tags are then updated as rapidly as possible. Tags can even be controlled using the listed commands if the CPU is in the STOP state.

Monitoring of tags with watch table

You can call the test functions of a watch table in the shortcut menu or using the icons in the toolbar of the working window shown in Fig. 15.17.



The icons from left to right:

Name in text	Tooltip text
Control mode	Show/hide all modify columns
Expanded mode	Show/hide advanced setting columns
Modify now	Modify all selected values once and now
Modify with trigger	All active values will be modified by “modify with trigger”
–	Disables the command output disable (OD) (on/off)
Monitor all	Monitor all
Monitor now	Monitor all values once and now

Fig. 15.17 Icons in the toolbar of the watch table

Double-click to open the watch table and select *Monitor all* or *Monitor now*. An online connection to the CPU will be established.

If neither the control mode nor the expanded mode are activated, 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 685). You can stop the current monitoring by clicking again on the *Monitor all* icon.

Please note that peripheral outputs can never be monitored. Monitoring the peripheral inputs is very time-intensive and can lead to the cycle time being exceeded under certain circumstances.

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 switch on the *Expanded mode* function, the *Monitor with trigger* column is 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.

Controlling of tags with watch tables

Double-click to open the watch table and switch the *Control mode* function on. 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.

In the *Modify value* column, enter the value to which the tag is to be set; in the *Tag selection* column, activate the checkbox 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 permanent monitoring prior to the modification. An online connection to the CPU is then already established and the success of the modification can be monitored.

Caution! Make sure that no dangerous states can occur when modifying tags!

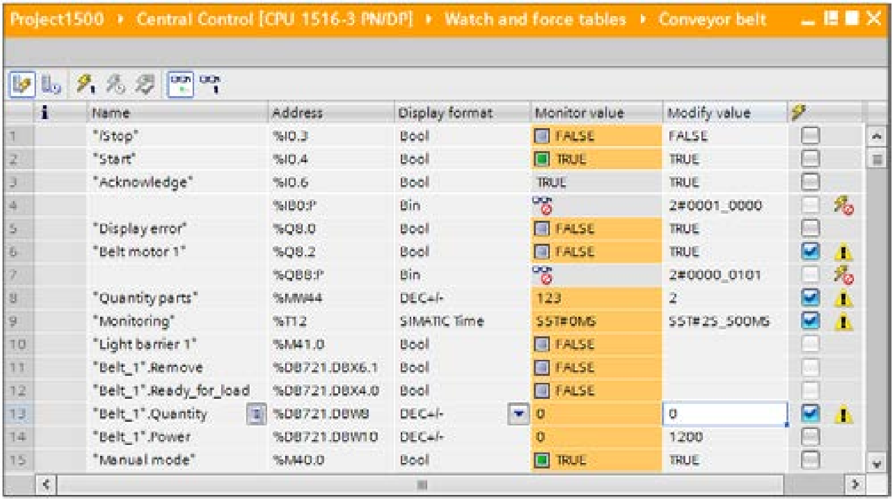
To modify the activated tags, click on *Modify now*. The tags activated in the *Tag selection* column are immediately set (as fast as possible) to the control value. If a tag is immediately overwritten after the modification by a value from the program – for example if a switched-on input has been controlled to “0” and the process image updating overwrites the control value again – the yellow triangle appears again in the *Tag selection* column.

In Fig. 15.18, the tags *“Belt motor_1”, “Quantity_parts”, “Monitoring”* (the time value of the SIMATIC timer function) and *“Belt_1”.Quantity* have been selected for modification.

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 tag selected in the watch table.

Please note that peripheral inputs can never be modified. Controlling the peripheral outputs is very time-intensive and can lead to the cycle time being exceeded under certain circumstances.

Only the tags visible in the table are modified. Multiple modification (multiple input) of a tag in the watch table is not permissible.



Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ Watch and force tables ▸ Conveyor belt

	i	Name	Address	Display format	Monitor value	Modify value		
1		"/Stop"	%I0.3	Bool	FALSE	FALSE	☐	
2		"Start"	%I0.4	Bool	TRUE	TRUE	☐	
3		"Acknowledge"	%I0.6	Bool	TRUE	TRUE	☐	
4			%IB0:P	Bin		2#0001_0000	☐	
5		"Display error"	%Q8.0	Bool	FALSE	TRUE	☐	
6		"Belt motor 1"	%Q8.2	Bool	FALSE	TRUE	☑	⚠
7			%QB8:P	Bin		2#0000_0101	☐	
8		"Quantity parts"	%MW44	DEC+/-	123	2	☑	⚠
9		"Monitoring"	%T12	SIMATIC Time	S5T#0MS	S5T#2S_500MS	☑	⚠
10		"Light barrier 1"	%M41.0	Bool	FALSE		☐	
11		"Belt_1".Remove	%DB721.DBX6.1	Bool	FALSE		☐	
12		"Belt_1".Ready_for_load	%DB721.DBX4.0	Bool	FALSE		☐	
13		"Belt_1".Quantity	%DB721.DBW8	DEC+/-	0	0	☑	⚠
14		"Belt_1".Power	%DB721.DBW10	DEC+/-	0	1200	☐	
15		"Manual mode"	%M40.0	Bool	TRUE	TRUE	☐	

Fig. 15.18 Example of controlling tags

Modify with trigger

In expanded mode, you can select the trigger time at which the tag values are modified in the CPU. If you switch on the *Expanded mode function*, the *Monitor with trigger* and *Modify with trigger* columns are displayed. You can then define the control time for each tag from a drop-down list.

If you click *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, modification can be triggered or exited by means of the *Online* > *Modify* > *Modify with trigger* command from the main menu or the *Modify* > *Modify with trigger* command from the shortcut menu.

Modify now

You can use the *Modify now* function to assign values once to tags independent of the monitoring and control modes. This job is executed as rapidly as possible. With the CPU at STOP, this command can be used to assign default values to tags.

Enter the modify values in the watch table and activate the checkbox after the modify value in the column in which the tag to be controlled is present. A yellow triangle indicates that this tag has been selected for modification but has not yet been controlled.

To carry out the modification, select the *Online* > *Modify* > *Modify now* command in the main menu or shortcut menu.

15.5.6 Testing with the force table

Tags can be preassigned fixed values. This action is referred to as “forcing”. A CPU 1500 can force tags from the peripheral inputs and peripheral outputs area. 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.

Please note: Forcing is sent to the CPU by means of a force job. *The force job remains active even if online mode is terminated and the online connection to the programming device canceled! The force job also remains active after switching the CPU off and on!* Forcing can only be exited using the *Force > Stop forcing* command; this command deletes the force job in the CPU.

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.

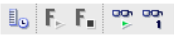
The tags entered with names must previously have been defined in a PLC tag table or in a data block. The tags from the PLC table and from data blocks with the *Optimized block access* attribute deactivated can also be entered with their memory address (absolute address) in the *Address* column. Inputs and outputs can only be entered with the I/O address (I:P or Q:P).

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 15.5.5 “Testing with watch tables” on page 684 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. 15.19.



The icons from left to right:

Name in text	Tooltip text
Expanded mode	Show/hide advanced setting columns
Start forcing	Starts or replaces forcing of the visible addresses in the Force table
Stop forcing	Stop forcing of the selected addresses
Monitor all	Monitor all
Monitor now	Monitor all values once and now

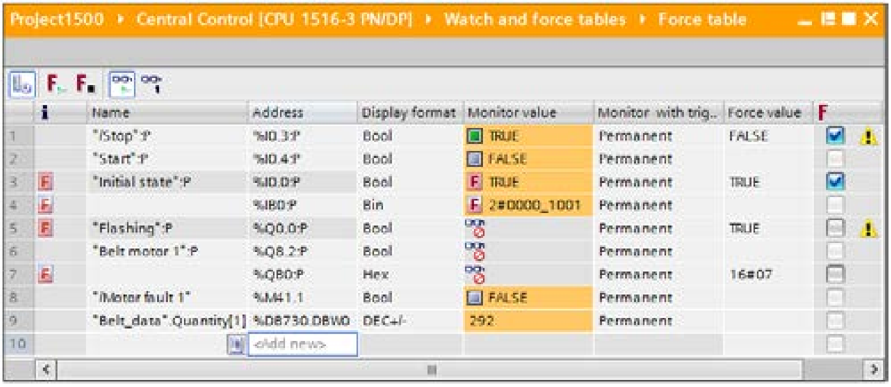
Fig. 15.19 Icons in the toolbar of the force table

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. Multiple forcing of an address is not possible if, for example, an I/O byte is forced and you want to force an individual bit of this byte again.

It is recommendable to switch on monitoring mode prior to forcing. An online connection to the CPU is then already established and the success of forcing can be monitored.

Caution: Make sure that no dangerous states can occur when forcing tags!

The *Start forcing* icon sends a force job to the CPU which contains the tags selected for forcing. Forcing is effective immediately (Fig. 15.20). A forced tag is marked with a red "F" in the first column of the force table. A CPU indicates an active force job with a continuous yellow MAINT LED.



Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ Watch and force tables ▸ Force table

	i	Name	Address	Display format	Monitor value	Monitor with trig..	Force value	F	
1		"/Stop":P	%I0.3:P	Bool	🟩 TRUE	Permanent	FALSE	☑	⚠
2		"Start":P	%I0.4:P	Bool	▢ FALSE	Permanent		□	
3	F	"Initial state":P	%I0.0:P	Bool	F TRUE	Permanent	TRUE	☑	
4	F		%IB0:P	Bin	F 2#0000_1001	Permanent		□	
5	F	"Flashing":P	%Q0.0:P	Bool		Permanent	TRUE	□	⚠
6		"Belt motor 1":P	%Q8.2:P	Bool		Permanent		□	
7	F		%QB0:P	Hex		Permanent	16#07	□	
8		"/Motor fault 1"	%M41.1	Bool	▢ FALSE	Permanent		□	
9		"Belt_data".Quantity[1]	%DB730.DBW0	DEC+/-	292	Permanent		□	
10			<Add new>					□	

Fig. 15.20 Example of forcing of peripheral inputs and outputs

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 then 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* > … .

The MAINT LED on the CPU turns off when no more force jobs are present in the CPU.

15.6 Measured value recording with the trace function

15.6.1 Introduction

Using the trace functions, a CPU 1500 records tag values in chronological order. The recordings are read out by the programming device and displayed in a curve graph if necessary. Because the CPU makes the recordings directly, the trace function is suitable for analyzing highly dynamic processes.

A CPU 1500 can handle up to 4 trace jobs simultaneously, each with up to 16 tags. The recordings are stored in the system memory of the CPU.

The trace configuration is retained when the power is off. The recording begins again when the system is started up. Previously recorded data is discarded.

Overview of the activities

You create a trace configuration using the trace editor. It contains the trace job with the tags used, the trigger condition, the time point at which recording is to begin, and information on the duration of the measurement.

You load the trace configuration into the CPU and activate the recording. If the trigger condition occurs, the CPU begins recording. You can monitor the recorded values with a programming device.

To store the recording, load it into the programming device. You can graphically display the data series as a curve graph using the logic analyzer function.

The measurements saved in the project (trace configuration + data series) are saved when the project is saved. You can also export and import the measurements.

15.6.2 Creating the trace configuration

The prerequisite is a project with a station, which supports the trace function. In the project view, double-click on the *Traces* editor under the station in the project tree.

To create a new trace configuration, click in the working window in the *Trace handling* table on *<Add new>* and enter the name of the trace job. In the inspector window, define the tags whose values are to be recorded under *Configuration > Signals*. The tags have an elementary data type and can be absolutely or symbolically addressed; the tags with “long” data types can only be symbolically addressed. Define a color for the curve graph for each tag.

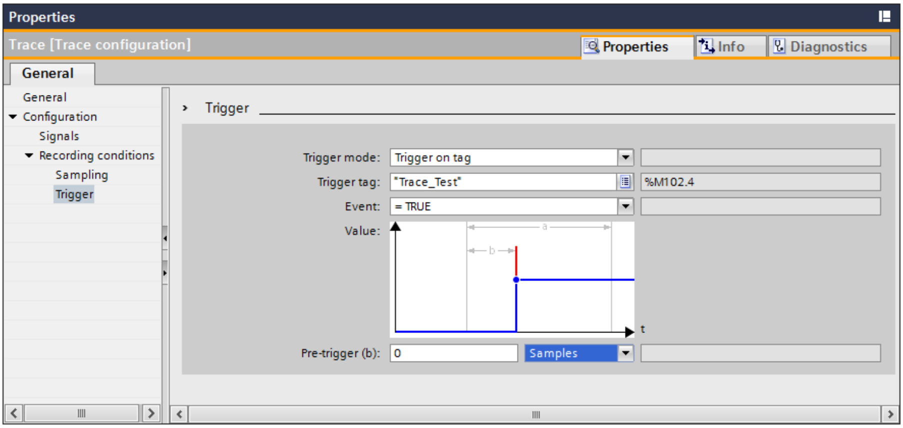
Now define the recording conditions. The recording is tied to an execution level (an organization block). The organization blocks of the main program, of a time-of-day interrupt, of a time-delay interrupt, of a cyclic interrupt, and of the isochronous mode interrupt are available. The organization block must be present in the program and be processed. The values are recorded at the end of the respective organization block.

Under *Configuration > Recording conditions > Sampling*, enter the organization block (selected from a drop-down list) in the *Sample with* field. Specify the grid in which the recording is to take place (how many cycles) and define the end of the recording (*number of samples* or *maximum recording duration*).

You define the beginning of recording under *Configuration > Recording conditions > Trigger*. If you select the entry *Record immediately* under *Trigger mode*, recording begins after downloading the configuration into the CPU. If you select *Trigger on tag*, you must define the tag. It can be located in the operand areas Inputs, Outputs, Bit memories, or Data. The trigger event depends on the data type of the tag. They can select:

- For the data type BOOL: TRUE, FALSE, rising or falling edge
- For a bit-serial or fixed-point data type: Equal or not equal to a value or a bit pattern
- For a fixed-point or floating-point data type: Value within a range or outside of a range, increasing or decreasing value (“rising” or “falling” signal)

Fig. 15.21 shows an example of the configuration of the trigger conditions for a tag with a fixed-point data type. With the pre-trigger you specify how many measuring points before the actual trigger signal the recorded values are to be saved. You save the trace configuration together with the project.



Trigger mode: Trigger on tag

Trigger tag: "Trace_Test" %M102.4

Event: = TRUE

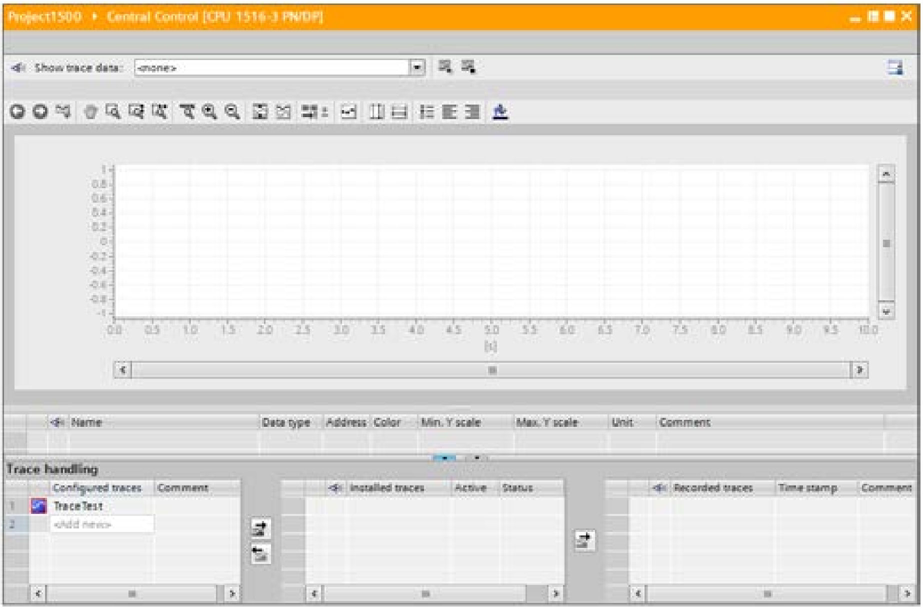
Value:

Pre-trigger (b): 0 Samples

Fig. 15.21 Example of trigger condition in a trace configuration

15.6.3 Loading a trace and recording

You use the trace handling function to transfer the trace configuration to the station and activate recording. The trace handling function is present in the bottom part of the working window. Fig. 15.22 shows the user interface of trace handling. The configured traces which are already set in the project are listed in the *Configured traces* table on the left. In the center table, *Installed traces*, you have the configured traces which have been loaded into the station. The table on the right,



Project1500 ▸ Central Control [CPU 1516-3 PN/DP]

Show trace data: <none>

Name	Data type	Address	Color	Min. Y scale	Max. Y scale	Unit	Comment

Trace handling

	Configured traces	Comment
1	TraceTest	
2	<Add new>	

Installed traces	Active	Status

Recorded traces	Time stamp	Comment

Fig. 15.22 Trace handling

Recorded traces, shows the combination of a configured trace and the associated recording.

The prerequisite for transferring the configured traces is the online mode. Select the trace configuration in the left table and click the icon *Transfer trace configuration to device*. You have the capability of uploading trace configurations that are in the station back into the project.

If you activate the trace in the *Active* column in the *Installed traces* table, recording will begin as soon as the trigger condition is fulfilled. In the column with the eye icon, you can display the recording data so that it is displayed in the curve chart or in the signal display.

You delete an installed trace by selecting the trace in the *Installed traces* table and choosing the *Delete* command from the shortcut menu.

15.6.4 Saving and evaluating recorded traces

Using the *Add to measurements* icon, save the trace configuration selected in the *Installed traces* table and the associated recording data in the project. The *Recorded traces* table shows the trace configurations present in the project with the associated recording data. You save the recorded traces together with the project.

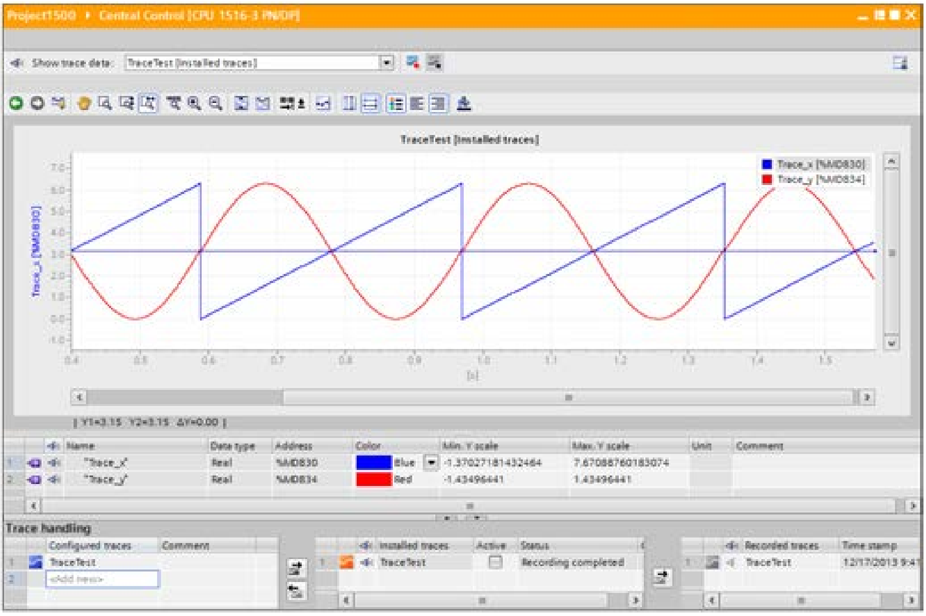
You export a recorded trace when you select the recorded trace in the *Recorded traces* table and select the *Export measurement* command from the shortcut menu. You are prompted to name the memory location (the file) in the file system. A recorded trace is saved with the file extension *.ttrc. You can also import a recorded trace back into the trace handling system.

Settings in the signal display

The signal display provides setting options of the signals used in the trace. Click on the eye icon in order to show or hide the associated measured value curve in the curve graph. You can select the color of each measured value curve from a drop-down list. The minimum and maximum value for the range of the Y scale can be specified for each measured value curve.

Settings in the curve chart

The curve graph graphically displays the recorded values of a trace. You can adapt the display of the measured values using the icons in the toolbar of the working window (Fig. 15.23).


Project1500 ▸ Central Control [CPU 1516-3 PN/DP]
Show trace data: TraceTest [Installed traces]
TraceTest [Installed traces]
Trace_x [%MD830]
Trace_y [%MD834]
Trace_x [%MD830]
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
-1.0
0.4
0.5
0.6
0.7
0.8
0.9
1.0
1.1
1.2
1.3
1.4
1.5
[s]

| Y1=3.15 Y2=3.15 ΔY=0.00 |

	Name	Data type	Address	Color	Min. Y scale	Max. Y scale	Unit	Comment
1	"Trace_x"	Real	%MD830	🟦 Blue	-1.37027181432464	7.67088760183074		
2	"Trace_y"	Real	%MD834	🟥 Red	-1.43496441	1.43496441		

Trace handling

	Configured traces	Comment
1	TraceTest	
2	<Add new>	

	Installed traces	Active	Status
1	TraceTest	□	Recording completed

	Recorded traces	Time stamp
1	TraceTest	12/17/2013 9:41

Fig. 15.23 Display of the recorded values

Using the *Zoom selection*, you can select any area by pressing and holding the mouse button. It will then be enlarged and displayed in the working area. With *Vertical zoom selection* and *Horizontal zoom selection*, you can scale the height and width of the curve graph. *Zoom in* and *Zoom out* enlarge or reduce the display. *Display all* scales the display in such a way that the entire recording range and all of the values are displayed. With *Scale automatically* you display all of the measured values for the selected recording range.

You can arrange the measured value series in lanes so that they do not overlap and you can display all of the measured points in the curve. The recording time or the measuring points can be selected for the units of the horizontal time axis.

The *Display vertical measurement cursors* icon displays two vertical guide lines in the curve graph, which can be moved using the mouse. The position of the guide lines (in units of the corresponding measured values) and their interval are displayed in the signal display. With *Display horizontal measurement cursors*, two horizontal guide lines are displayed, which can be moved using the mouse.

You can display a legend of the measurement series and define whether the display is to be displayed on the left or right upper edge of the curve graph. You can switch between various background colors for the curve graph.

You can copy the curve graph into the Windows clipboard and save it as a bitmap or print it.

16 Distributed I/O

16.1 Introduction, overview

Distributed I/O is the term used for input/output modules connected to the central PLC station over a bus system. The bus systems PROFINET IO and PROFIBUS DP are used in combination with an S7-1500 station.

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 input/output modules can be addressed via the operand areas Inputs (I), Peripheral inputs (I:P), Outputs (Q), and Peripheral outputs (Q:P).

Transfer of the user data 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: this is the IO controller for PROFINET IO and the DP master for PROFIBUS DP. The distributed stations – these are the IO devices with PROFINET IO and the DP slaves with PROFIBUS DP – 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” IO devices (I-devices) or DP slaves (I-slaves). 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 and PROFIBUS DP are represented as subnet. The connections required for data transfer are then present “automatically”.

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.

16.2 ET 200 distributed I/O system

ET 200 is the device family for the distributed I/Os on PROFINET IO and PROFIBUS DP. Depending on their use locally on the machine or in the process, the mechanical properties can be highly different, especially the degrees of protection: IP 20 for installation in a control cabinet and IP 65/67 for mounting directly on the machine.

The range of ET 200 stations extends from a simple compact station practically corresponding to an I/O module, to a station with modular design and several modules, up to the “intelligent” station which can execute a user program with its own CPU.

16.2.1 ET 200MP

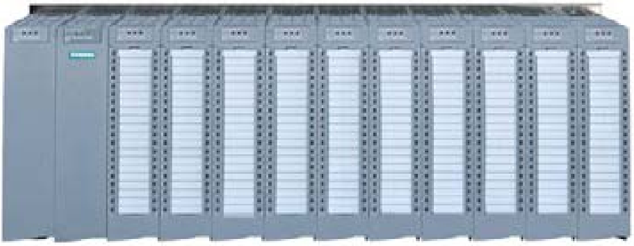


Fig. 16.1 ET 200MP

ET 200MP is a modular, distributed I/O system with degree of protection IP 20 for connection to PROFINET IO. Up to 30 I/O modules from the S7-1500 device range can be used on a mounting rail in a station. The internal bus signals are passed on from module to module via U-type connectors.

An ET 200MP station is comprised of an interface module and up to 30 modules, which are arranged to the right of the interface module. These can be input/output modules and technology or communication modules and, depending on the power balance, one or two power supply modules. Optionally, a power supply module can be inserted to the left of the interface module. The layout of an ET 200MP station corresponds to the layout of an S7-1500 station, as described in Chapter 2.1 “S7-1500 station components” on page 47.

With the interface module IM 155-5 PN ST, an ET 200MP station is operated as an IO device. The interface module has a PROFINET interface with two ports, which are connected to an integrated switch so that a linear topology can be set up without additional devices. The data transfer rate is 100 Mbit/s. The operating state of the interface module is indicated using LEDs (RUN, ERROR, MAINT).

In addition to the “normal” transfer of process data, isochronous mode with a minimum cycle of 250 µs and a maximum cycle of 4 ms is also possible. Other functions include device replacement without a programming device, media redundancy, IRT communication (isochronous real time), a firmware update and resetting to factory settings via PROFINET IO.

A prioritized startup in 500 ms is possible if the following prerequisites are met: A maximum of 12 modules, which all support the prioritized startup in 500 ms, and no power supply module may be plugged-in. The prioritized startup is not available when using IRT communication and media redundancy.

16.2.2 ET 200M

ET 200M is a modular distributed I/O system with degree of protection IP 20 and is particularly suitable for individual and complex automation tasks. Depending on the interface module, up to 8 or 12 modules from the S7-300 range can be used (the High-Feature version also allows the use of function and communication modules).

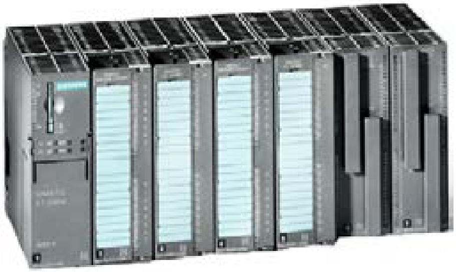


Fig. 16.2 ET 200M with IM 153-4 PN

The internal bus signals are passed on from module to module over a bus connector. If active bus modules are used onto which the modules are snapped, the latter can be replaced during ongoing operation.

The maximum data transfer rate on the PROFIBUS DP is 12 Mbit/s and 100 Mbit/s on the PROFINET IO. With the integral 2-port switch, a linear topology can be implemented without additional devices.

The ET 200M is also available in a hardened SIPLUS version and can be used with S7-300 modules with the same properties in environments with increased demands.

ET 200M can also be used in fault-tolerant systems for redundant operation. The fail-safe S7-300 modules can be used in the ET 200M – also mixed with standard modules. Together with Ex digital and analog modules, intrinsically-safe sensors and actuators can be connected from zones 1 and 2 of hazardous plants.

16.2.3 ET 200SP

ET 200SP is a modular, distributed I/O system with degree of protection IP 20 for connection to PROFINET IO. Up to 32 or 64 I/O modules can be used in one station. The internal bus signals are passed on from module to module over bus connectors (BaseUnits). It is possible to operate with equipment gaps and to replace I/Os during operation (“single hot swap”).

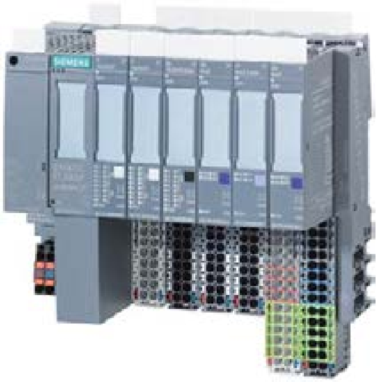


Fig. 16.3 ET 200SP

An ET 200SP station is comprised of an interface module with a bus adapter and I/O modules in a quantity dependent upon the power needs. The length of the backplane bus must not exceed 1 m. The I/O modules can be connected to potential groups with individual rooting of the power supply.

The IM 155-6 PN ST interface module control an ET 200SP station as IO device with up to 32 I/O modules. The IM 155-6 PN HF interface module control an ET 200SP station as IO device with up to 64 I/O modules. The interface modules have a PROFINET interface with two ports, which are connected to an integrated switch so that a linear topology can be set up without additional devices. The data transfer rate for PROFINET IO is 10/100 Mbit/s. The operating state of the interface modules is indicated using LEDs (RUN, ERROR, MAINT, PWR).

In addition to the "normal" transfer of the process data via PROFINET IO, a device replacement without a programming device, prioritized startup, shared device, media redundancy, IRT communication (isochronous real time) with send clocks of 250 µs to 4 ms, a firmware update and resetting to the factory settings via PROFINET IO are also possible.

16.2.4 ET 200S

ET 200S is a versatile I/O system with degree of protection IP 20 whose bit modular design allows exact adaptation to the automation task. Digital input/output modules, analog input/output modules, technology modules, motor starters, and frequency converters are available. Up to 63 I/O modules can be connected to the ET 200S interface module. The I/O modules can be replaced during ongoing operation; they are snapped onto terminal modules which contain the wiring. ET 200S is available with a PROFIBUS DP interface and a maximum data transfer rate of 12 Mbit/s or with PROFINET IO interface and a maximum data transfer rate of 100 Mbit/s.

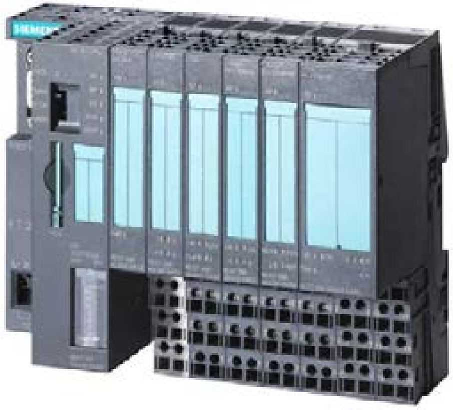


Fig. 16.4 ET 200S with IM 151 CPU

Together with the IM 151-7 CPU interface module, ET 200S can be used as a mini PLC. In association with the DP master module, the IM 151-7 CPU also has DP master functionality. The PLC functionality corresponds to that of a CPU S7-314. ET 200S with the IM 151-8 PN/DP CPU interface module can additionally be operated as an IO controller on PROFINET IO.

ET 200S is available with integral safety technology, where standard modules and fail-safe modules can be used together. A fail-safe mini PLC can be implemented using the IM 151-7 F-CPU interface module and the S7 Distributed Safety option package.

The ET 200S is also available as a PROFIBUS DP slave with digital inputs and outputs in a hardened SIPLUS version.

ET 200S COMPACT is a range of interface modules with onboard I/O, either with 32 digital inputs or with 16 digital inputs and outputs. Up to 12 ET 200S I/O modules (except F modules) can be connected to these interface modules so that a station can have up to 128 channels (mixed digital and analog).

ET 200S can also be used in fault-tolerant systems downstream of a Y-link (bus coupler for transition from a redundant to a single-channel PROFIBUS DP).

16.2.5 ET 200pro

ET 200pro is a modular I/O system with degree of protection IP 65/67 for use without a control cabinet. It consists of a module support and connection modules which accommodate the interface module for the bus connection and the electronic modules. Power modules for the load power supply combine the electronic modules into potential groups.

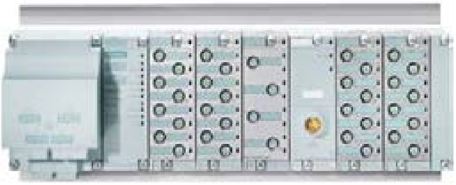


Fig. 16.5 ET 200pro with digital modules

The electronic modules are digital inputs/outputs and analog inputs/outputs. They can be replaced during ongoing operation. A frequency converter and motor starter (direct-on-line and reversing starter) as well as a pneumatic interface module with 16 outputs for the FESTO CPV 10 valve terminal are also available in this design.

Interface modules are available for ET 200pro with a PROFIBUS DP interface (maximum data transfer rate 12 Mbit/s) or a PROFINET IO interface (data transfer rate 100 Mbit/s) with the facility for wireless connection to a PROFINET IO controller. The PROFINET interface module has a 2-port switch for setting up a linear topology without additional devices.

Together with the IM 154-8 PN/DP CPU interface module, ET 200pro can be used as a mini PLC on site. Operation as a DP master or DP slave is possible on the PROFIBUS DP and as an IO controller on the PROFINET IO. The PLC functionality of the interface module corresponds to that of a CPU 315-2 PN/DP.

16.2.6 ET 200eco and ET200eco PN

ET 200eco with degree of protection IP 65/67 is the low-cost solution for processing digital and analog signals at machine level. ET 200eco is operated on PROFIBUS DP and ET 200eco PN on PROFINET IO.

ET 200eco

ET 200eco comprises a basic module and a connection block of different designs. Modules are available with 8 or 16 digital inputs, 8 or 16 digital outputs, 8 digital inputs and outputs each, and in fail-safe versions with 4 or 8 digital inputs.

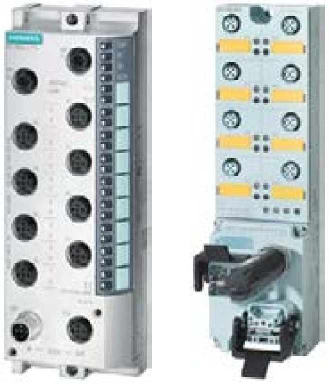


Fig. 16.6
ET 200eco PN (left) and ET 200eco with ECOFAST connection

The maximum data transfer rate on PROFIBUS is 12 Mbit/s. During commissioning and servicing, the modules can be disconnected interruption-free from the PROFIBUS and reconnected.

ET200eco PN

ET 200eco PN is the compact block I/O for processing digital, analog and IO-Link signals for connection to the PROFINET IO bus system. The design of the digital input and output modules is as with the PROFIBUS version of ET 200eco. Additionally available are an analog input module with 8 channels (4 × U/I, 4 × TC/RTD), an analog output module with 4 channels (U/I), and an IO-Link master with 4 IO-Link signals, 8 digital inputs, and 4 digital outputs.

ET 200eco PN is equipped with a 2-port switch so that a linear topology can be set up without additional devices. The data transfer rate is 100 Mbit/s.

16.3 PROFINET IO

16.3.1 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 stations which are referred to as IO devices (Fig. 16.7).

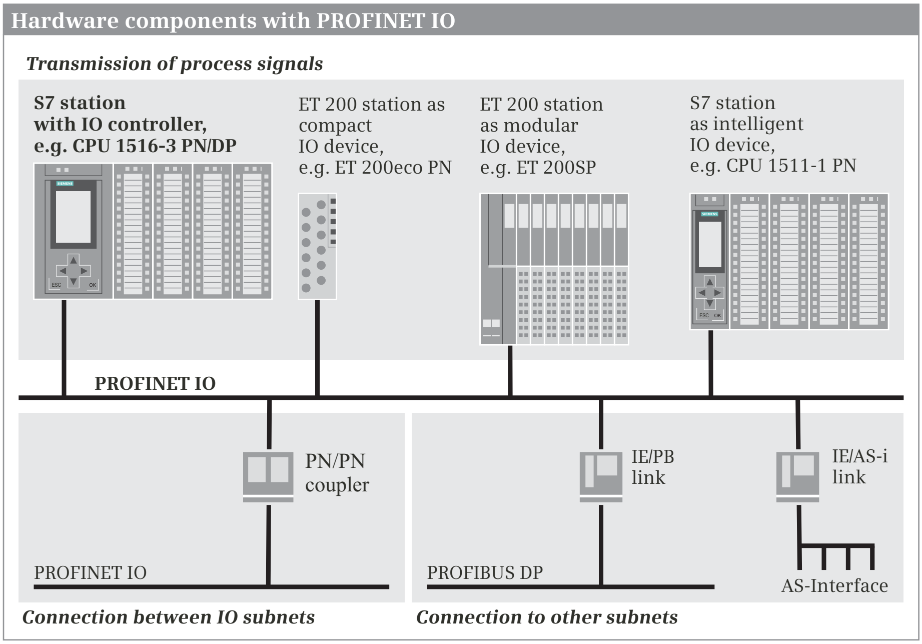

Hardware components with PROFINET IO
Transmission of process signals
S7 station with IO controller, e.g. CPU 1516-3 PN/DP
ET 200 station as compact IO device, e.g. ET 200eco PN
ET 200 station as modular IO device, e.g. ET 200SP
S7 station as intelligent IO device, e.g. CPU 1511-1 PN
PROFINET IO
PN/PN coupler
IE/PB link
IE/AS-i link
PROFINET IO
PROFIBUS DP
AS-Interface
Connection between IO subnets
Connection to other subnets

Fig. 16.7 Components of a PROFINET IO system

Industrial Ethernet can be designed physically as an electrical, optical, or wireless network. FastConnect Twisted Pairs (FC TP) with RJ45 connections, or Industrial Twisted Pairs (ITP) 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. Every CPU 1500 has an integrated 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 from the ET 200 distributed I/O system are the ET 200MP and ET 200SP.

More precisely, the PROFINET interface modules are the IO devices that communicate with the IO controller. For the sake of simplicity, the entire station is designated as an IO device in the following. Whenever this difference plays a role, it will be explicitly referred to.

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

Intelligent IO devices contain a CPU with a user program which controls the subordinate (own) modules. The user data interface to the IO controller is a transfer area which can be divided into different address areas. Examples of intelligent IO devices are S7 stations with CPUs with integral IO device 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.

An intelligent IO device can simultaneously be the IO controller for a subordinate PROFINET IO system.

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:

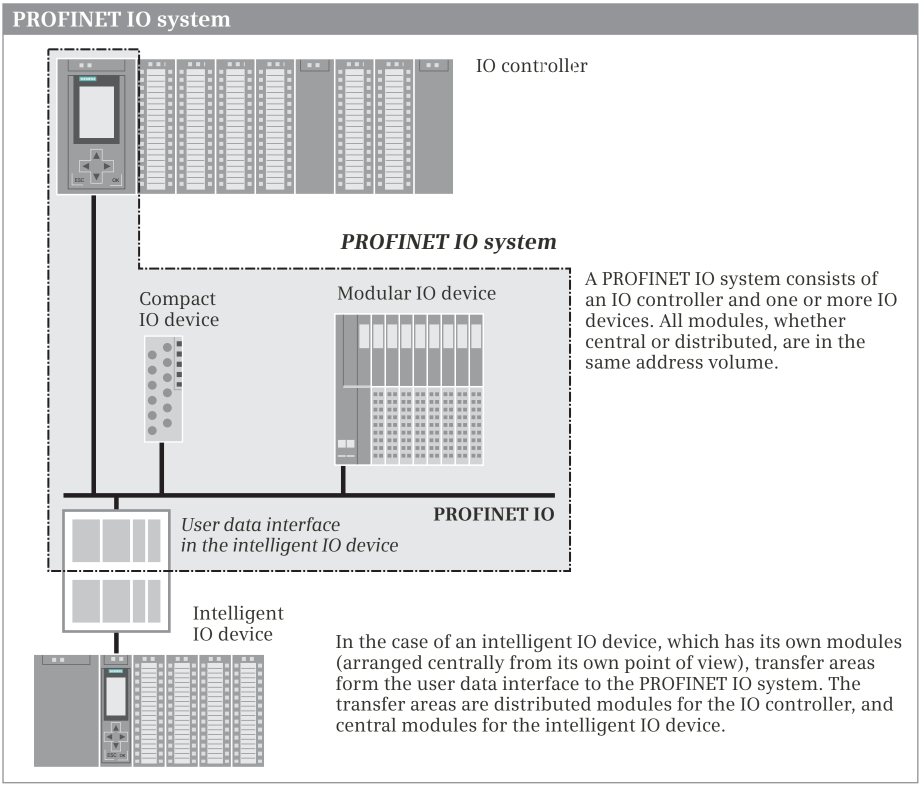
- ▷ PN/PN coupler for connecting two Ethernet subnets
- ▷ IE/PB Link PN IO for connecting an Ethernet subnet to a PROFIBUS subnet
- ▷ IE/AS-i Link for connecting an Ethernet subnet to an AS-i subnet

The coupling modules are described in more detail in Chapter 16.3.4 “Coupling modules for PROFINET IO” on page 708.

PROFINET IO system

The IO controller and all IO devices controlled by it constitute a PROFINET IO system (Fig. 16.8). An IO device is supplied with data by its IO controller within an update time which is calculated by the configuration editor 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.



A PROFINET IO system consists of an IO controller and one or more IO devices. All modules, whether central or distributed, are in the same address volume.

In the case of an intelligent IO device, which has its own modules (arranged centrally from its own point of view), transfer areas form the user data interface to the PROFINET IO system. The transfer areas are distributed modules for the IO controller, and central modules for the intelligent IO device.

Fig. 16.8 Schematic representation of a PROFINET IO system

16.3.2 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 – in address format IPv4 – 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 address for the above-mentioned IP address is 192.168.1 and the station address 3. Each station on the PROFINET is additionally assigned a device name and device number. Further information on the station addresses in an Ethernet subnet can be found in Chapter 3.4.6 "Configuring a PROFINET subnet" on page 80.

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 the configuration editor. It ranges from 100 to 115 and can be changed – not to be confused with the hardware identifier of the PROFINET I/O system in the system constants.

Hardware identifier

The configuration editor assigns an ID to each addressable hardware object, the hardware identifier or HW ID for short. The hardware identifier is used in the user program to address, for example, modules or interfaces if diagnostic information is to be read. All hardware identifiers of the automation station are listed in the default tag table in the *System constants* tab. You can also find the hardware identifiers in the properties of the hardware object. Further details can be found in Chapter 4.4 "Addressing of hardware objects" on page 107.

Logical addresses with PROFINET IO

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 of operands and tags" on page 94.

The user data of the IO devices shares the range of logical addresses with the user data of the central modules in the PLC station with the IO controller. This means that the addresses of the centrally arranged modules must not overlap with the user data addresses of the compact and modular IO devices and the addresses of the transfer areas of I-devices.

Consistent user data transfer to and from IO devices

Data consistency means that a block of user data is handled together.

With direct access, for example when loading and transferring, you can consistently transfer an area of one byte, one word, or one doubleword. With a user data area of three bytes or more than four bytes, you use the system blocks DPRD_DAT (read) and DPWR_DAT (write) for consistent data transfer. These and other system blocks for the consistent transfer of data between the CPU and an IO device are described in Chapter 16.5.1 “Read and write user data” on page 730.

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 transfer” on page 89.

User data interface with intelligent IO devices

With the compact and modular IO devices, the addresses of the inputs and outputs are together with the addresses of the central modules in the address volume of the IO controller. With intelligent IO devices (abbreviated to: I-devices), the input/output modules of the IO device are assigned to the device CPU. Every intelligent IO device therefore has a user data interface as common memory area with the IO controller whose size depends on the device CPU used.

The user data interface can be divided into several areas of different length. The individual areas then respond like modules whose lowest address is the module start address. From the viewpoint of the IO controller, the intelligent IO device then appears like a compact or modular IO device depending on the division.

A transfer area which is represented as an input module from the viewpoint of the IO controller is an output module from the viewpoint of the IO device and vice versa. The logical addresses on the controller side are in the address volume of the IO controller and the logical addresses on the device side in the address volume of the IO device. The addresses on the controller side can be different from those on the device side.

16.3.3 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 of the configuration is the IO controller integrated in a CPU 1500. *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 last two 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.
- ▷ To configure an intelligent IO device, drag it as a PLC station into the working window, set *IO device* mode in the properties of the PN interface, assign the IO controller, and configure the transfer areas of the user data interface.

The result is networking of the IO controller with the assigned IO devices to a PROFINET IO system (Fig. 16.9).

You then make the parameter settings for the stations and the fitting with input/output modules in the Device view.

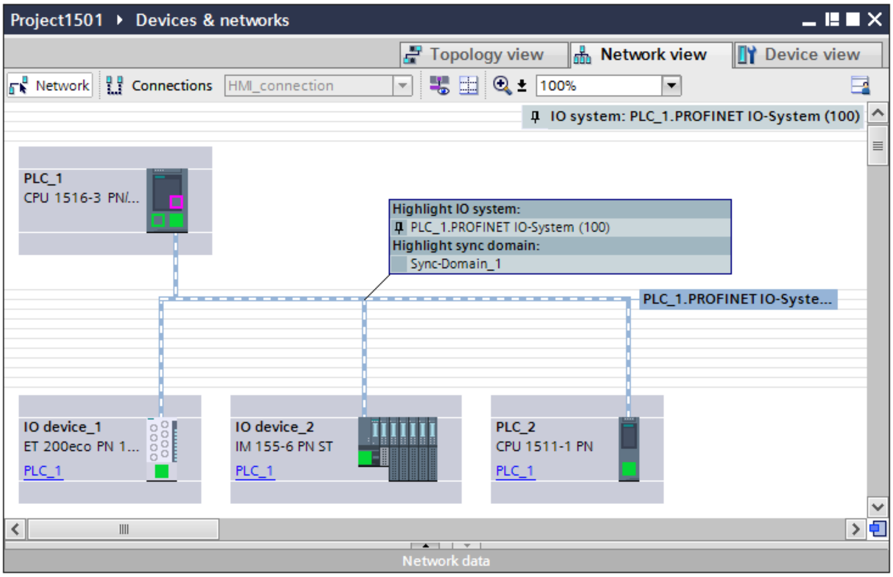


Fig. 16.9 Example of representation of a PROFINET IO system

Configuring the IO controller in the Network view

Prerequisite: You have created a project and a PLC station, for example a CPU 1500 with PN interface. 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 the project* option and change the preset IP address and subnet mask if necessary. Information on the IP address can be found in Chapter 3.4.6

"Configuring a PROFINET subnet" on page 80. Activate the *Set IP address using a different method* option if you wish, for example, to set the IP address per user program.

Set the mode: Select the *Operating mode* group in the interface properties and activate the *IO controller* checkbox if this is not already preset.

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 addresses* 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 *Add 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. 16.9 shows two stations of the distributed I/O: An ET200eco station from the object tree *Distributed I/O > ET 200eco PN* > PROFINET > *DI/DO > 8DI/8DO x 24VDC* > ... and an ET 200SP station from the object tree *Distributed I/O > ET 200SP > Interface modules > PROFINET > IM 155-6 PN ST* > ...

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.

You set the Ethernet addresses in the properties of the PROFINET interface. In the *Advanced options* group you can additionally set – depending on the application – for example the prioritized startup, the device replacement without removable medium, or participation in media redundancy.

Coupling an intelligent IO device to the PROFINET IO system

You initially create an intelligent IO device ("I-device") as a stand-alone PLC station and then connect the PN interface of the I-device to the PROFINET IO system. You can find the I-devices in the hardware catalog in the *Controllers* folder.

For example, if you want to create an S7-1500 station as an I-device, press and hold the left mouse button and drag the CPU from the *Controllers > SIMATIC S7-1500 > CPU > CPU 1511-1 PN >* ... object tree to the working area.

You establish a connection to the existing subnet if you drag the PN interface of the I-device to a PN interface of another device on the subnet with the left mouse button pressed, for example to the PN interface of the IO controller.

In the properties of the PN interface of the I-device, activate the *IO device* checkbox under the *Operating mode* entry and select the assigned IO controller from the drop-down list. The station is then added as an IO device to the PROFINET IO system.

Configuring the user data interface

You configure the user data interface to the IO controller in the module properties of the I-device. Select the CPU or ET station in the working window and then the *Operating mode > I-device communication* group in the inspector window in the *Properties* tab under the *PROFINET interface* group.

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 *Data direction* column (↔), click on the arrow to set the type of transfer area (arrow to the right → means input area, arrow to the left ← means output area from the viewpoint of the I-device).

Now set the start address in the *Address in I-device* column and the length of the transfer area in the *Length* column. In the *Address in IO controller* column, set the start address which the transfer area has from the viewpoint of the IO controller.

In this manner you can configure further transfer areas. The configured transfer areas are displayed in the *I-device communication* properties group. If you click a transfer area here, you obtain its details (Fig. 16.10). In this display, you can select the association to a process image: *Automatic update*, if the process image update is to take place during execution of the main program, or *PIPn* for a process image partition.

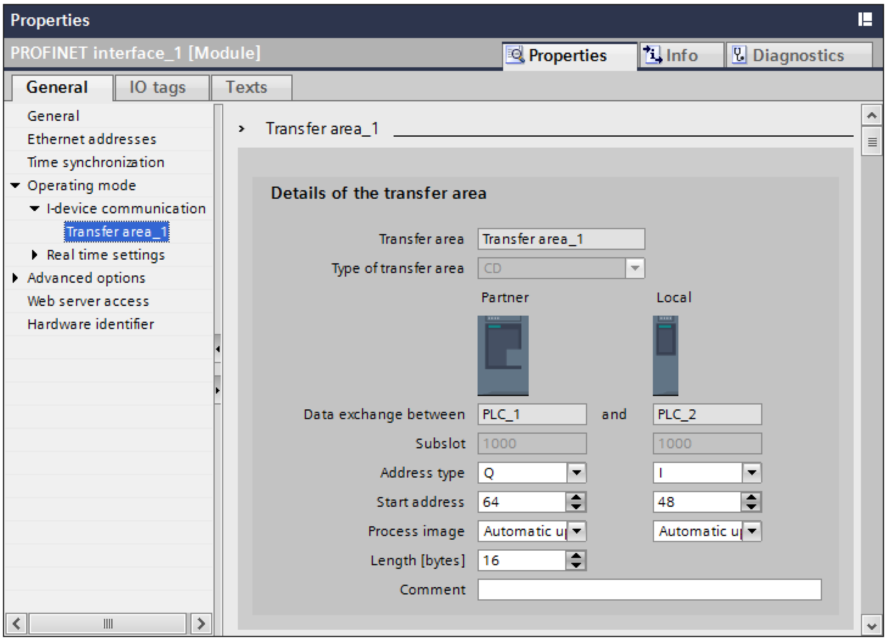
16.3.4 Coupling modules for PROFINET IO

PN/PN coupler: connection of two Ethernet subnets

A PN/PN coupler connects two Ethernet subnets in order to exchange data between the IO controllers of the two subnets. There is galvanic isolation between the subnets.

The two sides of the PN/PN coupler each represent an IO device when configuring. One side (one IO device) is coupled to one of the PROFINET IO systems, the other side to the other system.

You can find the PN/PN coupler in the hardware catalog under *Other field devices > PROFINET IO > Gateway > Siemens AG > PN/PN couplers > PN/PN coupler Vx.0 > ...*.



Details of the transfer area

Transfer area: Transfer area_1

Type of transfer area: CD

	Partner		Local
Data exchange between	PLC_1	and	PLC_2
Subslot	1000		1000
Address type	Q		I
Start address	64		48
Process image	Automatic u		Automatic u
Length [bytes]	16		
Comment			

Fig. 16.10 Example of configuration of a transfer area

The modules underneath this represent the two sides of the PN/PN coupler (X1 for the left side and X2 for the right side of the module).

In order to connect the PN/PN coupler, drag the symbol for one side of the PN/PN coupler with the left mouse button pressed to the PROFINET IO system. You can set the properties of the PN/PN coupler, for example IP address, device name and device number, in the inspector window with the module selected. You configure the second side (X2) of the PN/PN coupler on the other PROFINET IO system in the same manner.

IE/PB Link PN IO: Connection of PROFINET IO to PROFIBUS DP

An IE/PB Link PN IO connects the Industrial Ethernet and PROFIBUS subnets. In standard mode, the link permits cross-subnet PG/OP communication and communication via S7 connections, parameterization of field devices via data record routing, and the network transition to a DP master system with constant bus cycle time.

When operating as PROFINET IO proxy, the IE/PB Link PN IO takes over the role of a proxy for the DP slaves on the PROFIBUS. The IO controller on the PROFINET can then address the DP slaves on the PROFIBUS like IO devices in its PROFINET IO system.

The IE/PB Link PN IO is a double-width module of S7-300 design. You connect the IE/PB Link to Industrial Ethernet using an 8-pole RJ45 socket and to PROFIBUS using a 9-pole SUB-D socket.

The IE/PB Link PN IO is configured as an IO device to which a DP master system is connected. You can find the link in the hardware catalog under *Network components > Gateways > IE/PB Link >* ... . In order to add it to the PROFINET IO system, drag it with the left mouse button pressed to the PROFINET IO system in the working window.

You set the operating mode – standard mode or PROFINET IO proxy – in the IE/PB link properties under *Network gateway*. You configure the Ethernet addresses and the real-time settings in the *PROFINET interface* group.

You can also set the PROFINET device number and the assignment to the PROFIBUS station number in the properties of the IE/PB link. The table shown under *PROFINET device number* contains the PROFIBUS station number in the *PB address* column and the device number assigned by the configuration editor in the *PROFINET device number* column. To change the device number, click in the cell with the device number and select an unused device number from the drop-down list. If you activate the checkbox in the *Device number = PB address* column, the PB address and the device number are set the same.

The IE/PB Link PN IO is the DP master of the subordinate PROFIBUS DP master system. How to configure a DP master system with the assigned DP slaves is described in Chapter 16.4.3 “Configuring PROFIBUS DP” on page 721.

IE/AS-i Link PN IO: Connecting PROFINET IO to the AS-Interface

An IE/AS-i Link PN IO connects PROFINET IO with AS-Interface. On PROFINET IO, the link is an IO device. On the AS-Interface, it is an AS-i single or double master in accordance with the AS-i specification V3.0. You can find the link in the hardware catalog under *Other field devices > PROFINET IO > Gateway > Siemens AG > IE/AS-i Link PN IO >* ... . In order to add it to the PROFINET IO system, drag it with the left mouse button pressed to the PROFINET IO system in the working window.

Connection to the IO controller is via a user data interface with 62 bytes digital inputs and 62 bytes digital outputs. A programming device can be connected via the integral Ethernet port for commissioning, testing, and diagnostics via a web interface with a standard browser. The link allows uploading of the AS-i configuration to the programming device.

16.3.5 Real-time communication in PROFINET

PROFINET 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). The stations participating in the IRT communication (synchronized stations), are grouped together in a sync domain.

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.

Send clock

Cyclic data exchange is handled within a specific time frame, the send clock. The configuration editor calculates the send clock from the configuration information on the PROFINET IO system. The send clock is the shortest possible update time.

You can configure the send clock for an unsynchronized IO controller in its interface properties. With the PN interface selected, select a value in the properties tab under *Advanced options* > *Real-time settings* > *IO communication* from the drop-down list *Send clock*. If the IO controller is the sync master in a sync domain, set the send clock using the *Domain settings* button in the properties of the sync domain.

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 IO 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 update time and “Accepted updating cycles without IO data”.

If the IO device is assigned to an unsynchronized IO controller, 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. If the IO device is assigned to a sync domain, the update time corresponds to the send clock in the properties of the sync domain.

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 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.

Isochronous real-time

Isochronous real time (IRT) is hardware-supported real-time communication designed, for example, for motion control applications. IRT message frames are deterministically transmitted via planned communication paths in a specified order. IRT communication therefore requires network components that support this planned data transmission.

To be able to configure IRT communication, set up a sync domain (see next section) and determine a sync master, which will take over the synchronized distribution of the IRT message frames to the sync slaves. IRT requires a topology configuration (see section "Topology editor") and thus a defined structure that takes account of the transmission properties of the cables and the switches used.

SYNC domain

A sync domain is a group of PROFINET I/O stations which exchange synchronized data with each other. A station, which can be an IO controller or an IO device, assumes the role of the sync master. The others are the sync slaves.

A sync domain can contain several I/O systems, where a complete I/O system is always assigned to a single sync domain. Several sync domains can exist on an Ethernet subnet.

A default domain is automatically created with the name *Sync-Domain_1* when an I/O system is configured. All of the configured IO systems, IO controllers and IO devices are initially located in this sync domain, but they are unsynchronized. You can now use the default domain for IRT communication or you can create a new sync domain.

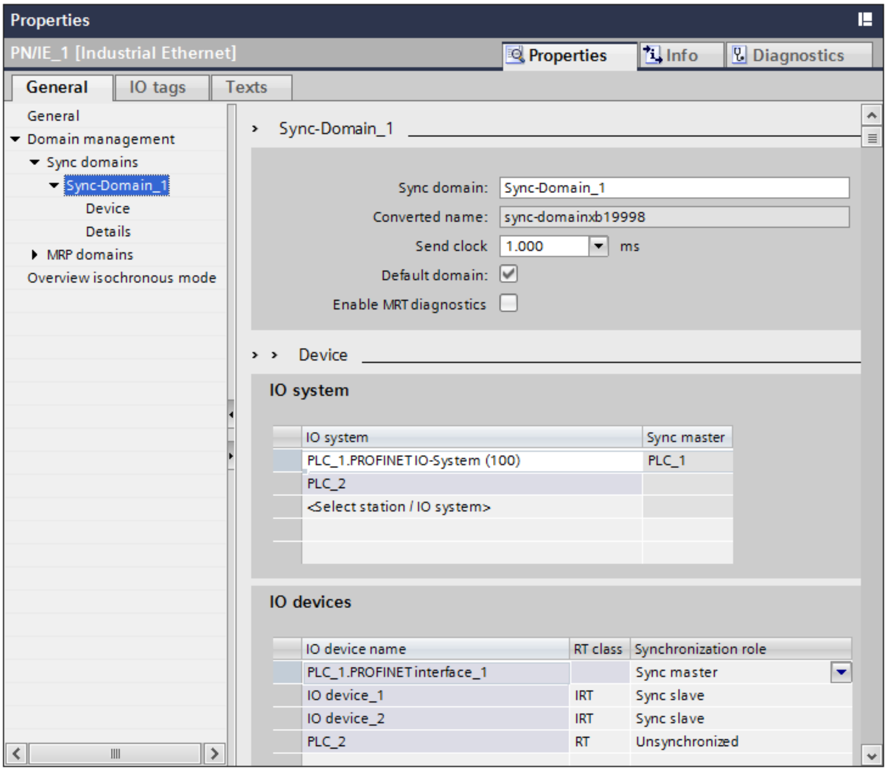
Configuration of a new SYNC domain

Prerequisite: You have configured the Ethernet subnet with one or several PROFINET IO systems. The stations involved in IRT communication must also support this function.

To create a new sync domain, select the Ethernet subnet in the network view and select *Properties* from the shortcut menu. In the inspector window of the *General* tab, open the *Domain management* group. The *Sync domains* table contains the already configured sync domains. You set up a new sync domain when you overwrite the entry *<New sync domain>* with the name of the new sync domain.

You can set the send clock in the properties of the sync domain (Fig. 16.11). To add the station, select the entry *Device* under the sync domain. The *IO system* table shows the configured IO systems and PLC stations. If you select an IO system, the

IO devices table shows the configured devices. In the *Synchronization role* column, set the sync master. For the IO devices that you want to synchronize, set the entry IRT in the *RT class* column.



Sync domain: Sync-Domain_1

Converted name: sync-domainxb19998

Send clock 1.000 ms

Default domain: ☑

Enable MRT diagnostics □

IO system

IO system	Sync master
PLC_1.PROFINET IO-System (100)	PLC_1
PLC_2	
<Select station / IO system>	

IO devices

IO device name	RT class	Synchronization role
PLC_1.PROFINET interface_1		Sync master
IO device_1	IRT	Sync slave
IO device_2	IRT	Sync slave
PLC_2	RT	Unsynchronized

Fig. 16.11 Configuration of a SYNC domain

Topology editor

The topology editor allows the configuring of wiring for devices on the Industrial Ethernet subnet. In the network view, the logical connections between the PROFINET devices are configured; with the topology editor the physical connections with the properties length and cable type for determining the signal runtimes. Use of the topology editor is a prerequisite for using IRT communication.

The physical connections between devices on the Ethernet subnet are point-to-point connections. The connections on a PN interface are called ports. The Ethernet cable connects a device port with a port on the partner device. To enable several nodes to communicate with each other, they are connected to a switch that has several connections (ports) and that distributes signals. If a PN interface has two ports connected by an integrated switch, you can implement a linear topology without external switches.

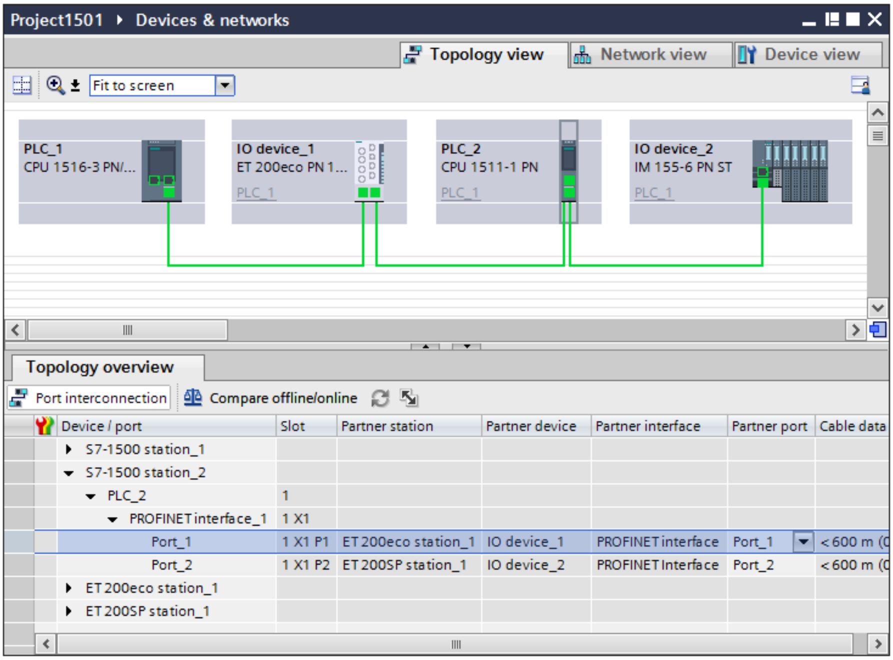
You can also configure the connection of two ports in the device view beforehand. In the graphic, select the PN interface and select in the properties *Advanced options > Port [X...] > Port interconnection*. In the *Partner port* field, select the desired connection in the drop-down list. Here, you can also set the cable properties that are relevant for determining the send clock. You set the connection properties and the boundaries (limits) under *Port options*:

▷ *End of the detection of accessible stations*
The stations located behind this port are no longer displayed in STEP 7 under *Accessible devices*.

▷ *End of the topology discovery*
The detection of the topology ends at this port.

▷ *End of the sync domain*
The sync domain ends at this port.

In the *Topology view* tab of the hardware configuration, you can graphically configure the port interconnection (with graphs or tables) (Fig. 16.12).



Device / port	Slot	Partner station	Partner device	Partner interface	Partner port	Cable data
▸ S7-1500 station_1						
▾ S7-1500 station_2						
▾ PLC_2	1					
▾ PROFINET interface_1	1 X1					
Port_1	1 X1 P1	ET 200eco station_1	IO device_1	PROFINET interface	Port_1	< 600 m (
Port_2	1 X1 P2	ET 200SP station_1	IO device_2	PROFINET Interface	Port_2	< 600 m (
▸ ET 200eco station_1						
▸ ET 200SP station_1						

Fig. 16.12 Graphical and tabular view of the PROFINET topology

The ports of the configured stations are displayed in the working area. To interconnect two ports, press and hold the right mouse button and drag one port to the other. You can delete the interconnection by highlighting the line and pressing the [Del] key. The *Topology overview* table shows the port interconnection in tabular form. You can compare the configured connection with the actual connection in online mode using the *Compare offline/online* button.

If you select a port in the graphic or in the table, the inspector window will show the interconnection and the configured options of this port in the *Properties* tab.

16.3.6 Special PROFINET configurations

In the properties of the PROFINET interface, you can activate the PROFINET functions described below when configuring an IO controller or IO device (depending on the device configuration).

Device replacement without removable medium or programming device

When replacing an IO device, a device name must be assigned to the new IO device in order to make it known (again) to the IO controller. This can be carried out – depending on the IO device – using a removable medium (e.g. a memory card) or the programming device.

Under certain conditions, the new IO device can be identified by means of neighbor relationships between the other IO devices and the IO controller and assigned a new device name by the IO controller. One of the requirements is that a port connection is configured and the *Support device replacement without exchangeable medium* checkbox is activated when configuring the interface properties under *Advanced options* > *Interface options*. Only new IO devices or IO devices which have been reset to the factory settings should be used as replacement devices.

Prioritized startup, docking systems

With a prioritized startup, the startup of IO devices in a PROFINET IO system with RT and IRT communication is carried out faster. Special cabling conditions must be observed. The maximum possible number of IO devices controlled with prioritized startup depends on the IO controller used.

The prioritized startup minimizes the time until the cyclic exchange of user data can be started after restoration of power, after the return of a station, or after activation of an IO device. One possible application is the swapping of machine parts or tools that are controlled with IO devices.

You configure the prioritized startup in the properties of the PROFINET interface of an IO device using the *Prioritized startup* checkbox. You can find the checkbox under *Advanced options* > *Interface options* or – with an intelligent IO device – under *Operating mode* (with *IO device* mode switched on and assigned IO controller). In the properties of the port, you set the data transfer rate under *Port options* to *TP 100 Mbps full duplex* and deactivate the checkbox *Enable autonegotiation*.

For a docking system, under *Port interconnection*, activate the checkbox *Alternative partner* in the properties of the (unconnected) port, which is operated with alternating partners during runtime. In the table under this, double-click on <*Add alternative partner*> to select the desired partner ports. The alternative connections are displayed with dashed lines in the technology view.

Media redundancy

The media redundancy is used to increase the network availability by means of a special topology. The ends of a linear topology are connected into a ring topology in a station at the two connections of the PN interface. This station is the redundancy manager and the connections are the ring ports. If a station in the ring network fails, an alternative communication path can be made available.

Up to 50 stations can participate per ring by means of the Media Redundancy Protocol (MRP) used with SIMATIC S7. The stations are grouped in an MRP domain. All partner ports must have identical settings. IRT communication and prioritized startup cannot be used if media redundancy is configured.

To configure the media redundancy, interconnect all of the stations to a ring in the topology view. The network editor automatically creates a default MRP domain with the name *mrpdomain_1*, in which no stations yet exist. To set the MRP manager, select the desired station and select the media redundancy role *Manager* or *Manager (Auto)* in the properties of the PROFINET interface under *Expanded options* > *Media redundancy* from a drop-down list. For the other stations in the media redundancy, select *Client*.

You are given an overview of the participating stations if you highlight the PROFINET IO system in the network view and select the group *Domain management* > *MRP domains* in its properties. The available MRP domains are displayed. The list under *<MRP domain name>* > *Device*, shows you all configured devices of the selected IO system.

16.4 PROFIBUS DP

16.4.1 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. 16.13).

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).

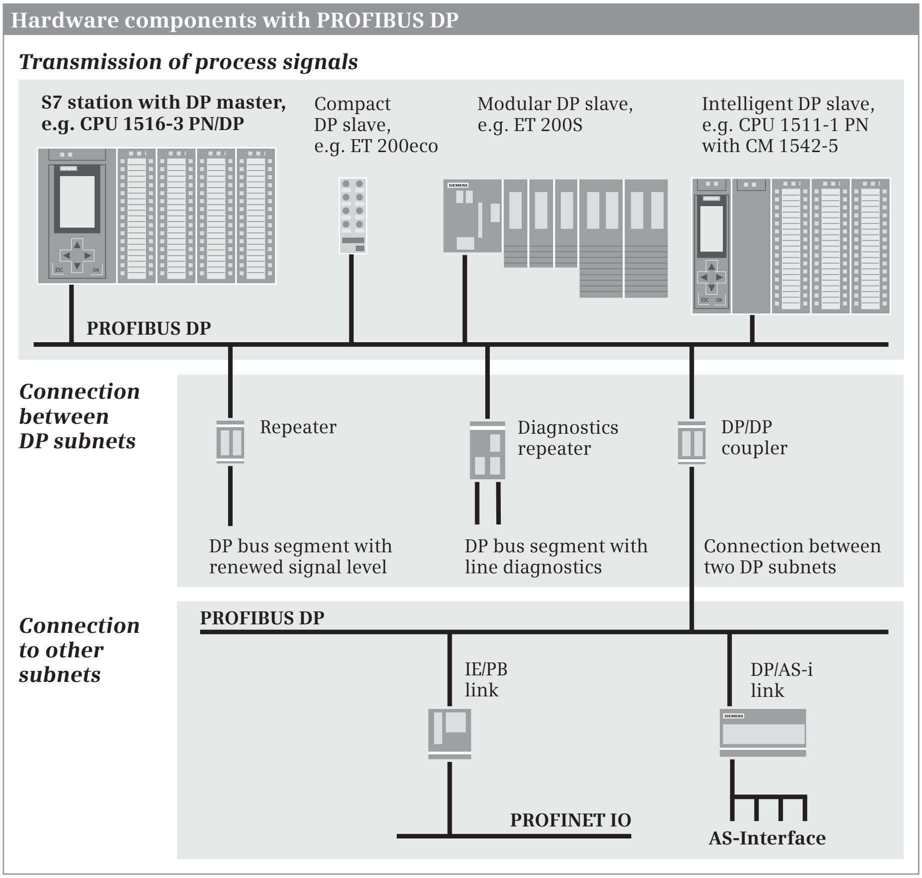

Hardware components with PROFIBUS DP
Transmission of process signals
S7 station with DP master, e.g. CPU 1516-3 PN/DP
Compact DP slave, e.g. ET 200eco
Modular DP slave, e.g. ET 200S
Intelligent DP slave, e.g. CPU 1511-1 PN with CM 1542-5
PROFIBUS DP
Connection between DP subnets
Repeater
Diagnostics repeater
DP/DP coupler
DP bus segment with renewed signal level
DP bus segment with line diagnostics
Connection between two DP subnets
Connection to other subnets
PROFIBUS DP
IE/PB link
DP/AS-i link
PROFINET IO
AS-Interface

Fig. 16.13 Hardware components with PROFIBUS DP

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 1516-3 PN/DP)
- ▷ A communication module in the PLC station (e.g. CM 1542-5)
- ▷ The IE/PB Link PN IO

DP slaves

The DP slaves are the passive stations on the PROFIBUS DP. These can be stations with process inputs and outputs, repeaters, couplers, 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. Some examples of intelligent DP slaves are S7-1500 stations with the CM 1542-5 communication module, the distributed I/O stations ET 200S with the IM 151-7 CPU interface module, and ET 200pro with the IM 154-8 PN/DP CPU interface module.

A CPU which is configured as an intelligent DP slave cannot be a DP master at the same time. However, a communication module can be operated as DP master in the station with an intelligent DP slave.

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:

- RS 232 repeater for regeneration of the bus signals
- Diagnostics repeater for diagnostics of bus faults
- 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

The repeater modules, the DP/DP coupler, and the DP/AS-i link are described in more detail in Chapter 16.4.4 “Coupling modules for PROFIBUS DP” on page 724, the IE/PB Link PN IO in Chapter 16.3.4 “Coupling modules for PROFINET IO” on page 708.

PROFIBUS DP master system

The DP master and all DP slaves controlled by it form a PROFIBUS DP master system (Fig. 16.14). 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.

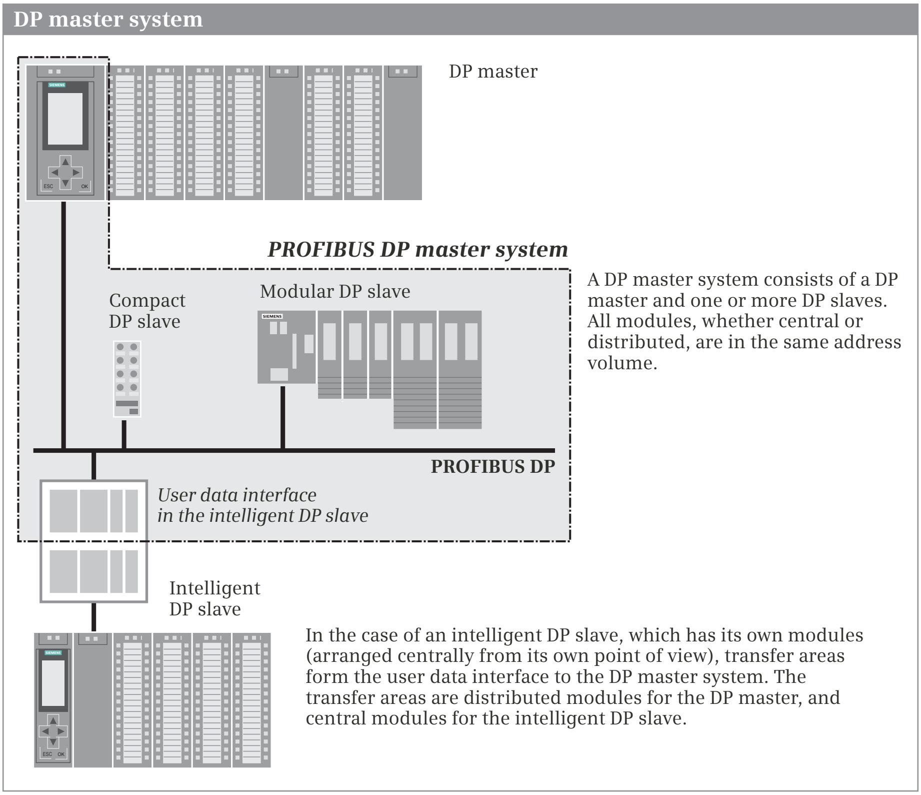

DP master system
DP master
PROFIBUS DP master system
Compact DP slave
Modular DP slave
A DP master system consists of a DP master and one or more DP slaves. All modules, whether central or distributed, are in the same address volume.
PROFIBUS DP
User data interface in the intelligent DP slave
Intelligent DP slave
In the case of an intelligent DP slave, which has its own modules (arranged centrally from its own point of view), transfer areas form the user data interface to the DP master system. The transfer areas are distributed modules for the DP master, and central modules for the intelligent DP slave.

Fig. 16.14 Schematic representation of a PROFIBUS DP 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 the use of data records transferred acyclically or the use of additional 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.

16.4.2 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.

The configuration editor 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 the configuration editor and ranges from 1 to 32 – not to be confused with the hardware identifier of the DP master system in the system constants. The name and the ID can be changed in the properties of the DP master system under *General*.

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 of operands and tags" on page 94.

Consistent user data transfer to and from DP slaves

Data consistency means that a block of user data is handled together without interruption.

With direct access, for example when loading and transferring, you can consistently transfer an area of one byte, one word, or one doubleword. With a user data area of three bytes or more than four bytes, you use the system blocks DPRD_DAT (read) and DPWR_DAT (write) for consistent data transfer. These and other system blocks for the consistent transfer of data between the CPU and a DP standard slave are described in Chapter 16.5.1 "Read and write user data" on page 730.

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 transfer" on page 89.

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.

16.4.3 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 for configuring is the DP master – either integrated in a CPU 1500 with DP interface or as the CM 1542-5 communication module.
- Activate the *DP master* mode of the DP interface if it is not the default setting.
- 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.
- Drag an intelligent DP slave as an independent PLC station into the working window, set the *DP slave* mode in the properties of the DP interface, assign the DP master, and configure the transfer areas 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. 16.15).

You then make the parameter settings for the stations and the fitting with input/output modules in the Device view.

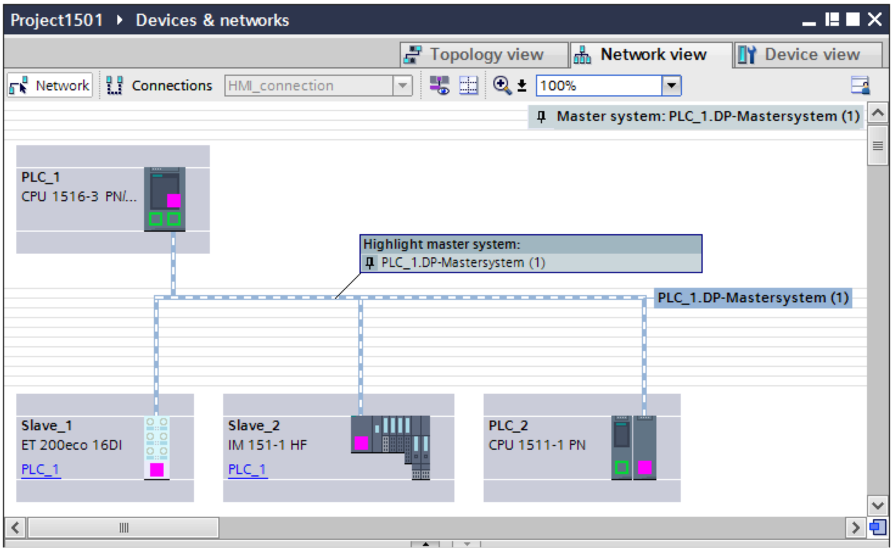


Fig. 16.15 Example of representation of a PROFIBUS DP master system

Configuring the DP master in the Network view

Prerequisite: You have created a project and a PLC station, for example a CPU 1516-3 PN/DP with DP interface. 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 *Add 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 name and 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. 16.15 shows two stations of the distributed I/O: An ET200eco station from the object tree *Distributed I/O > ET 200eco > PROFIBUS > DI/DO > 8DI/8DO >* ... and an ET 200S station from the object tree *Distributed I/O > ET 200S > Interface modules > PROFIBUS > IM 151-1 HF > ...*.

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 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 and, depending on the DP slave and application, in the *Module parameters* group, for example, the startup property *Operation 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 *Controllers* folder.

For example, if you want to create an S7-1511 station as an I-slave, press and hold the left mouse button and drag the CPU 1511-1 PN into the working window and equip it with the CM 1542-5 communication module from the object tree *Controllers > SIMATIC S7-1500 > Communication modules > PROFIBUS > CM 1542-5 > ...*.

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. With an S7 or ET 200 station with combined MPI/DP interface as I-slave, you must first set *PROFIBUS* as the interface type in the interface properties.

In the properties of the DP interface of the I-slave, activate the *DP slave* option under the *Operating mode* entry and 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 CPU or the ET station in the working window and then the *Operating mode > I-slave communication* entry in the inspector window in the *Properties* tab in the *DP interface* group.

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 *Data direction* column (↔), click on the arrow to set the type of transfer area (arrow to the right → means input area, arrow to the left ← 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 32 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. 16.16).

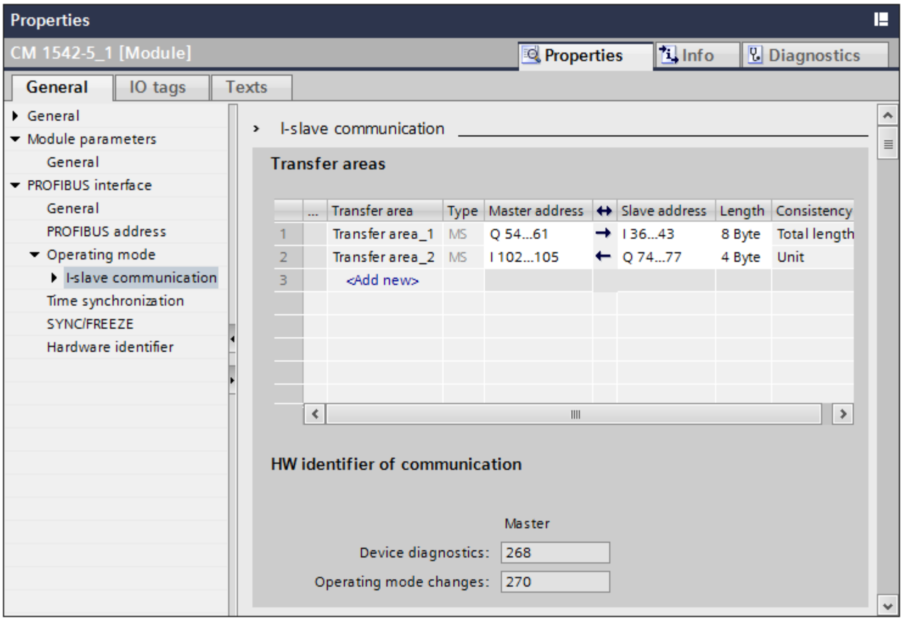
In this manner you can configure further transfer areas. The configured transfer areas are displayed in the *I-slave communication* properties group. If you select a transfer area here, you obtain its details. In this display, you can select the association with a process image: *Automatic update*, if the process image update is to take place during execution of the main program, or *PIPn* for a process image partition. You can assign a hardware interrupt to each transfer area, which is triggered when the lowest bit has a signal state change.

16.4.4 Coupling modules for PROFIBUS DP

RS485 repeater for PROFIBUS DP

The RS485 repeater connects two bus segments together in a PROFIBUS subnet. The number of stations and the size of the subnet can then be increased. The repeater provides signal regeneration and galvanic isolation. It can be used at data transfer rates up to 12 Mbit/s – including 45.45 Kbit/s for PROFIBUS PA.

It is not necessary to configure the RS 485 repeater; it need only be considered when calculating the bus parameters.



I-slave communication

Transfer areas

	...	Transfer area	Type	Master address	↔	Slave address	Length	Consistency
1		Transfer area_1	MS	Q 54...61	→	I 36...43	8 Byte	Total length
2		Transfer area_2	MS	I 102...105	←	Q 74...77	4 Byte	Unit
3		<Add new>						

HW identifier of communication

Master

Device diagnostics: 268

Operating mode changes: 270

Fig. 16.16 Example of configuration of the transfer areas of an I-slave

Diagnostics repeater for PROFIBUS DP

The diagnostics repeater can determine the topology in a PROFIBUS segment (RS 485 copper cable) during ongoing operation and carry out line diagnostics. It provides signal regeneration and galvanic isolation for the connected segments. The maximum segment length is 100 m in each case; the data transfer rate can be between 9.6 Kbit/s and 12 Mbit/s.

The diagnostics repeater has connections for 3 bus segments. The cable from the DP master is connected to the supply terminals of the DP1 bus segment. The two other connections DP2 and DP3 contain the measuring circuits for determination of the topology and for cable diagnostics on the bus segments connected to them. Up to nine further diagnostics repeaters can be connected in series.

The diagnostics repeater is handled like a DP slave in the master system. In the event of a fault, it sends the determined diagnostic data to the DP master. This includes the topology of the bus segment (stations and cable lengths), the contents of the segment diagnostic buffers (last ten events with fault information, location, and cause), and the statistics data (information on the quality of the bus system). In addition, the diagnostics repeater provides monitoring functions for isochronous mode.

The diagnostic data is displayed in the navigation window of the online and diagnostics view of the diagnostics repeater in the *Segment diagnostics* folder. System blocks in the user program permit line diagnostics. The system function DP_TOPOL triggers diagnostics on the repeater and RD_REC or RDREC is used to read the diagnostic data. READ_CLK reads the CPU time and WR_REC or WRREC transfers it to the diagnostics repeater in order to set the time on the latter.

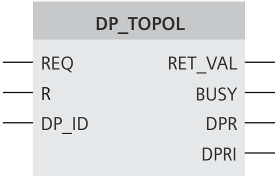
The diagnostics repeater is configured and parameterized with the configuration editor. You can find it in the hardware catalog under *Network components* > *Diagnostics repeaters* > ....

DP_TOPOL Determine bus topology

DP_TOPOL uses a diagnostics repeater to determine the bus topology of the DP master system whose ID you specify in the DP_ID parameter. Fig. 16.17 shows the graphic representation of the system block. You find it in the program elements catalog under *Extended instructions* > *Distributed I/O* > *Others*.

System block for determining the PROFIBUS topology

Determine bus topology



DP_TOPOL determines the bus topology on a diagnostics repeater.

Parameter assignment:

REQ	Job initiation	RET_VAL	Error information
R	Cancel determination	BUSY	Job finished
DP_IP	Hardware identifier of the DP master system	DPR	Station number of the diagnostics repeater
		DPRI	Error messages

Fig. 16.17 Determine the topology of a DP master system

The determination is triggered by REQ = "1" and is finished when BUSY signals "0". You can use R = "1" to cancel determination of the topology.

If an error is signaled by a diagnostics repeater, determination of the bus topology is prevented and this is shown in the DPR and DPRI parameters. If several diagnostics repeaters signal errors, the error message of the first one is displayed and the complete diagnostic information can be read with DPNRM_DG or the programming device.

A distinction is made between temporary and permanent faults in the error information in the DPRI parameter. In certain circumstances it may not be possible to conclusively identify temporary faults such as a loose contact and these may disappear on their own. You must eliminate permanent faults before you call DP_TOPOL again to determine the topology.

Following processing of DP_TOPOL, the determined data is available on the diagnostics repeater and can be read using RDREC. The data comprises the topology of the bus segment (stations and cable lengths), the contents of the segment diagnostic buffers (last ten events with fault information, location, and cause), and the statistics data (information on the quality of the bus system).

DP/DP coupler

The DP/DP coupler (Version 2) connects two PROFIBUS subnets to each other and can exchange data between the DP masters. The two subnets are electrically isolated and can be operated at different data transfer rates up to a maximum of 12 Mbit/s. In both subnets, the DP/DP coupler is assigned to the relevant DP master as a DP slave with a freely selectable station address in each case.

The maximum size of the transfer memory is 244 bytes of input data and 244 bytes of output data, divisible into a maximum of 16 areas. Input areas in one subnet must correspond to output areas in the other. Up to 128 bytes can be transferred consistently. If the side with the input data fails, the corresponding output data on the other side is maintained at its last value.

The DP/DP coupler is configured with the configuration editor. You can find it in the hardware catalog under *Other field devices > PROFIBUS DP > Gateways > Siemens AG > DP/DP Coupler, Release 2 > …*.

You configure the transfer area in the device view. This shows the graphics of the DP/DP coupler in the top part of the working window and the configuration table of the interface in the bottom part. Now drag an I/O module present under the DP/DP coupler from the hardware catalog into the table (the modules are displayed directly if the *Filter* checkbox is activated in the hardware catalog). The user data addresses that you specify in the module properties are in the address volume of the DP master.

Configure the second part of the DP/DP coupler in the same way. Add a DP/DP coupler to the second DP master system and configure the transfer area. Make sure that the structure of the transfer area matches that of the first part. Inputs on one side correspond to outputs on the other side and vice versa. The addresses in both parts of the DP/DP coupler are oriented to the address assignments of the relevant master CPU and can differ from each other.

DP/AS-i link: Connection between PROFIBUS DP and AS-Interface

The **DP/AS-i Link 20E** connects PROFIBUS DP with AS-Interface. On the PROFIBUS DP, the link is a modular DP slave in accordance with EN 50170. On the AS-Interface, it is an AS-i master in accordance with the AS-i specification V2.1.

Connection to the DP master is via a user data interface with 32 bytes digital inputs and 32 bytes digital outputs. The link allows uploading of the AS-i configuration to the programming device.

The **DP/AS-i Link Advanced** connects PROFIBUS DP with AS-Interface. On the PROFIBUS DP, the link is a modular DP slave in accordance with EN 50170. On the AS-Interface, it is an AS-i single or double master in accordance with the AS-i specification V3.0.

The DP/AS-i link is configured with the configuration editor. You can find it in the hardware catalog under *Other field devices > PROFIBUS DP > Gateways > Siemens AG > AS-I > ...* .

Connection to the DP master is via a user data interface with 62 bytes digital inputs and 62 bytes digital outputs. A programming device can be connected via the integral Ethernet port for commissioning, testing, and diagnostics via a web interface with a standard browser. The link allows uploading of the AS-i configuration to the programming device.

16.4.5 Special PROFIBUS configurations

You can configure the following special functions in a PROFIBUS DP master system if the devices are designed accordingly:

- SYNC/FREEZE groups for synchronous output of output signals and synchronous reading in of input signals
- Direct data exchange between stations on the PROFIBUS

Operation with equidistant bus cycles and isochronous mode for deterministic response times is described in Chapter 16.7.3 “Isochronous mode with PROFIBUS” on page 742.

Configuring SYNC/FREEZE groups

The SYNC control command requests the DP slaves combined into a group to simultaneously (synchronously) output the output states. The FREEZE control command requests the DP slaves combined into a group to simultaneously (synchronously) freeze the current input signal states to allow them to then be cyclically fetched by the DP master. The UNSYNC and UNFREEZE control commands respectively cancel the effects of SYNC and FREEZE.

You can generate up to eight SYNC/FREEZE groups per DP master system which are to execute either the SYNC command, the FREEZE command, or both. Each DP slave can only be assigned to one group.

Using the system block DPSYC_FR in the user program, you can trigger the output of a command to a group. The DP master then sends the corresponding command simultaneously to all DP slaves in the specified group.

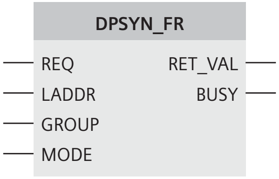
To assign a DP slave to a SYNC/FREEZE group, open its interface properties and assign the DP slave to a group under *SYNC/FREEZE*. You can find the list with the groups in the interface properties of the DP master under SYNC/FREEZE and can set the properties (SYNC, FREEZE, or both) there for each group.

DPSYC_FR Send SYNC/FREEZE commands

DPSYC_FR sends the SYNC, UNSYNC, FREEZE, and UNFREEZE commands to a SYNC/FREEZE group which you have configured with the hardware configuration. Fig. 16.18 shows the graphic representation of the system block. You find it in the program elements catalog under *Extended instructions > Distributed I/O > Others*.

System block for synchronizing DP slaves

Send SYNC/FREEZE command



DPSYN_FR sends SYNC/FREEZE commands to DP slaves.

Parameter assignment:

REQ	Job initiation
LADDR	Hardware identifier of the DP master interface
GROUP	SYNC/FREEZE group
MODE	Command at "1" Bit 2 UNFREEZE Bit 3 FREEZE Bit 4 UNSYNC Bit 5 SYNC
RET_VAL	Error information
BUSY	Job is being processed

The following commands are possible on the MODE parameter:

B#16#04	UNFREEZE	B#16#14	UNSYNC + UNFREEZE
B#16#08	FREEZE	B#16#18	UNSYNC + FREEZE
B#16#10	UNSYNC	B#16#24	SYNC + UNFREEZE
B#16#20	SYNC	B#16#28	SYNC + FREEZE

Fig. 16.18 Synchronous reading and writing from and to DP slaves

The send procedure is triggered by REQ = “1” and is finished when BUSY signals “0”. In the GROUP parameter, each group occupies one bit (from bit 0 = group 1 to bit 7 = group 8). At least one bit must always be set. The commands are specified at the MODE parameter. SYNC and UNSYNC commands or FREEZE and UNFREEZE commands must not be triggered simultaneously in a call.

Following a startup, SYNC mode and FREEZE mode on the DP slaves are initially switched off. The inputs of the DP slaves are scanned in sequence by the DP master and the outputs of the DP slaves are controlled; the DP slaves immediately output the received output signals at the output terminals.

If you wish to “freeze” the input signals of several DP slaves at a certain time, output the FREEZE command to the associated group. The input signals read by the DP master in succession have the signal states which they had when “freezing”. These input signals retain their values until you use a further FREEZE command to request the DP slaves to read in and freeze updated input signals, or until you switch the DP slaves back to “normal” mode using the UNFREEZE command.

If you wish to output the output signals of several DP slaves synchronously at a certain time, first output the SYNC command to the associated group. The addressed DP slaves then retain the current signals at the output terminals. You can then transfer the desired signal states to the DP slaves. Output the SYNC command again following completion of transfer; in this manner you request the DP slaves to connect the received output signals simultaneously to the output terminals. The DP slaves retain the signals at the output terminals until you connect the new output signals using a further SYNC command, or until you switch the DP slaves back to “normal” mode using the UNSYNC command.

Note that the SYNC and FREEZE commands are still valid following a start.

Configuring direct data exchange

In a DP master system, the DP master only controls the slaves assigned to it. With correspondingly designed stations, only a different station (master or intelligent slave, referred to as receiver or subscriber) on the PROFIBUS subnet can “listen in” to find out what input data a DP slave (the sender or publisher) is sending to “its” master. This direct data exchange is also referred to as direct communication.

You can also use direct data exchange between two DP master systems on the same PROFIBUS subnet. For example, the master in master system 1 can “listen in” in this manner to the data of a slave in master system 2.

A prerequisite for configuration of direct data exchange is configuration of the sender station with input modules. First define the partner stations. Select a partner in the Network view – with two I-slaves as partners, this must be the sender – and open the *I/O communication* tab in the configuration table in the bottom part of the working window. The DP slaves which have already been configured are listed here. The *Drop the device here or select* cell is present in the *Partner 2* column. Click in this cell and select the partner station for direct data exchange from the drop-down list or drag the partner station from the graphic into this cell using the mouse. The partner station is entered in a new line in the configuration table with the operating mode *Direct data exchange*.

Select the line with the partner station and enter the desired transfer areas in the inspector window under *Direct data exchange* in the *Transfer areas* table. Select the desired module in the *Partner module* column from the drop-down list and define the input address in the receiver station, the length of the transfer area, and the data consistency.

16.5 System blocks for distributed I/O

16.5.1 Read and write user data

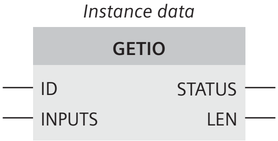
The following system blocks can be used for PROFIBUS DP and PROFINET IO for reading and writing user data:

- ▷ GETIO Read all inputs of an input area
- ▷ SETIO Write all outputs of an output area
- ▷ GETIO_PART Read some inputs of an input area
- ▷ SETIO_PART Write some outputs of an output area
- ▷ DPRD_DAT Read user data consistently
- ▷ DPWR_DAT Write user data consistently

Fig. 16.19 shows the graphic representation of these system blocks. The program elements catalog contains the system blocks under *Extended instructions > Distributed I/O*.

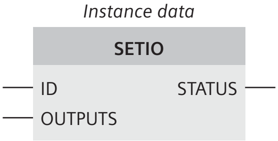
System blocks for reading and writing of user data

Read all inputs of an input area



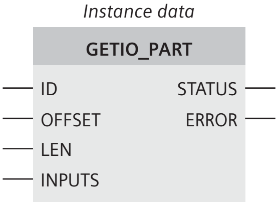
GETIO reads all inputs of an input area that is located in a distributed station.

Write all outputs of an output area



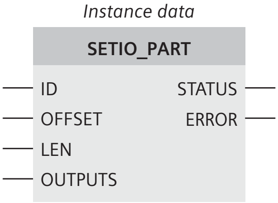
SETIO writes all outputs of an output area that is located in a distributed station.

Read some inputs of an input area



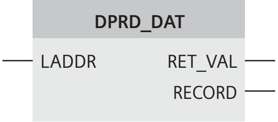
GETIO_PART reads some inputs of an input area that is located in a distributed station.

Write some outputs of an output area



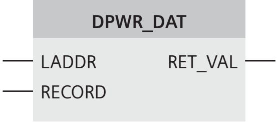
SETIO_PART writes some outputs of an output area that is located in a distributed station.

Read user data consistently



DPRD_DAT reads the user data from a distributed station consistently.

Write user data consistently



DPWR_DAT writes the user data to a distributed station consistently.

Parameter assignment:

ID	Hardware identifier of the object	STATUS	Error information
INPUTS	Destination area for the read data	LEN	Number of bytes
OUTPUTS	Source area for the data to be written		
OFFSET	Number of the first byte	ERROR	With "1": completion with error
LADDR	Hardware identifier of the object	RET_VAL	Error information
RECORD	Destination area for the read data or source area for the data to be written		

Fig. 16.19 Reading and writing user data with distributed I/Os

Common parameters

The parameters ID and LADDR use the hardware identifier to specify the components from which the data is to be read or to which the data is to be written. For a compact station, it is the hardware identifier of the user data. For a modular station, it is the hardware identifier of a module in the station and, for an intelligent station, it is the hardware identifier of a transfer area.

The parameters INPUTS, OUTPUTS and RECORD have the data type VARIANT. The following are permitted as actual parameters:

- ▷ an absolutely or symbolically addressed tag,
- ▷ an area that is absolutely addressed with an ANY pointer, or
- ▷ a type data block.

The prerequisite for an error-free transfer is that the user data that is to be written or read be addressed in the process image and belong to a single component. No boundaries to adjacent user data areas must not be violated with the parameters OFFSET and LEN.

GETIO Read all inputs of an input area

Using DPRD_DAT, GETIO consistently reads all of the inputs from the component of a DP standard slave or IO device addressed with the parameter ID.

The destination area specified with the INPUTS parameter must be exactly the same length as the configured length of the input area read that is also output with the LEN parameter.

Error information is output at the STATUS parameter.

SETIO Write all outputs of an output area

Using DPWR_DAT, SETIO consistently writes all of the outputs of the component of a DP standard slave or IO device addressed at the parameter ID.

The source area specified with the OUTPUTS parameter must be exactly the same length as the configured length of the output area to be written to.

Error information is output at the STATUS parameter.

GETIO_PART Read some inputs of an input area

Using DPRD_DAT, GETIO_PART consistently reads a part of the inputs of the component of a DP standard slave or IO device addressed at the parameter ID. The number of the first byte to be read is present at the OFFSET parameter and the amount of bytes to be read is present at the LEN parameter.

The destination area specified with the INPUTS parameter must be exactly as long as or longer than the number of bytes to be read. If the destination area is larger, only the first LEN bytes of the area are written to and ERROR has signal state “1”. If an error occurs during the data transfer, ERROR likewise has signal state “1”.

SETIO_PART Write some outputs of an output area

Using DPWR_DAT, SETIO_PART consistently reads a part of the outputs to the component of a DP standard slave or IO device addressed at the parameter ID. The number of the first byte to be written is present at the OFFSET parameter and their quantity is present at the LEN parameter.

The source area specified with the OUTPUTS parameter must be exactly as long as or longer than the number of bytes to be written. If the source area is larger, only the first LEN bytes are transferred and ERROR has signal state "1". If an error occurs during the data transfer, ERROR likewise has signal state "1".

DPRD_DAT Read user data consistently

DPRD_DAT reads consistent user data from a DP standard slave or an IO device.

The hardware identifier of the component from which the user data is to be read is present at the LADDR parameter. The RECORD parameter specifies the destination area in which the read data is saved. The destination area must be at least as large as the user data area that is read.

DPWR_DAT Write user data consistently

DPWR_DAT writes consistent user data to a DP standard slave or an IO device.

The hardware identifier of the component to which the user data is to be written is present at the LADDR parameter. The RECORD parameter specifies the source area from which the written data is retrieved. The source area must be at least as large as the user data area that is to be written.

16.5.2 Read diagnostic data from a DP standard slave

DPNRM_DG Read diagnostic data

DPNRM_DG reads the diagnostic data of a DP standard slave. Fig. 16.20 shows the graphic representation of the system block. You find it in the program elements catalog under *Extended instructions > Distributed I/O > Others*.

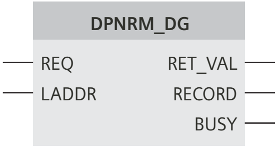
The read procedure is triggered by REQ = "1" and is finished when BUSY signals "0". The number of read bytes is then present in the function value RET_VAL. Depending on the slave, the diagnostic data is at least 6 bytes and a maximum of 240 bytes long. The first 240 bytes are transferred if the diagnostic data is longer and then the corresponding overflow bit is set in the data.

The RECORD parameter describes the area in which the read data is saved. The actual parameter can be an absolutely or symbolically addressed tag, an area that is absolutely addressed with an ANY pointer, or a type data block.

Note that DPMRM_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.

System block for reading diagnostic data

Read diagnostic data



DPNRM_DG reads diagnostic data from a DP standard slave.

Parameter assignment:

REQ	Job initiation	RET_VAL	Error information
LADDR	Hardware identifier of the DP slave	RECORD	Receive mailbox for the diagnostic data
		BUSY	Job is being processed

Fig. 16.20 Read diagnostic data from a DP standard slave

16.5.3 Receive and provide a data record

An intelligent IO device can receive a data record from the IO controller and provide a data record for the IO controller. The system blocks required for this are graphically displayed in Fig. 16.21. The program elements catalog contains the system blocks under *Extended instructions* > *Distributed I/O* > *Others*.

RCVREC Receive data record from an IO controller

RCVREC receives a data record from the IO controller in the program of an intelligent IO device. The MODE parameter defines the operating mode:

▷ MODE = 0: Check whether a request for receiving a data record is present. If NEW = “1”, a new data record is present.

▷ MODE = 1: Receive a data record for any transfer area in the user data interface. The MLEN parameter specifies the maximum number of bytes to be received. If NEW = “1”, the data record has been written into the data area defined by the RECORD parameter.

▷ MODE = 2: Receive a data record for a specific transfer area in the user data interface defined by the F_ID parameter. The MLEN parameter specifies the maximum number of bytes to be received. If NEW = “1”, the data record has been written into the data area defined by the RECORD parameter.

▷ MODE = 3: Accept the received data record and send a positive reply to the IO controller. The CODE1 and CODE2 parameters must be occupied by zero.

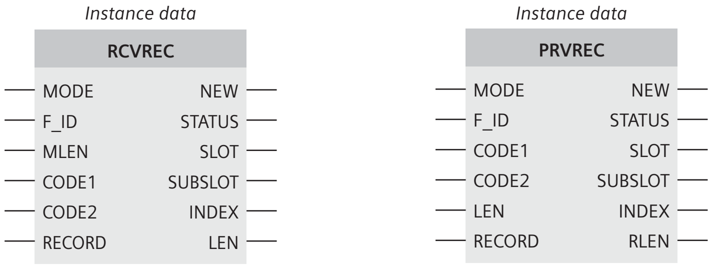
▷ MODE = 4: Reject the received data record and send a negative reply to the IO controller. You transfer the error code at the CODE1 and CODE2 parameters.

RCVREC must first be called with MODE = 1 or MODE = 2 and subsequently with MODE = 3 or MODE = 4 within certain periods which depend on the CPU.

System blocks for receiving and providing data records

Receive data record from an IO controller

Provide data record for an IO controller



Parameter assignment:

MODE	Job ID NEW	New data record	
F_ID	Hardware identifier for the transfer area	STATUS	Error information
MLEN	Maximum number of bytes	SLOT	(like F_ID)
CODE1	Error code	SUBSLOT	(like F_ID)
CODE2	Error code	INDEX	Number of the data record
RECORD	Receive mailbox for the data record	LEN	Length of the data record
		RLEN	Length of the send data record

Bild 16.21 Datensatz empfangen und bereitstellen

The number of the received data record is output at the INDEX parameter and its length at the LEN parameter. The STATUS parameter contains the error information. SLOT and SUBSLOT are occupied identical to F_ID.

PRVREC Provide data record for an IO controller

RCVREC provides a data record in the program of the intelligent IO device upon request by the IO controller. The MODE parameter defines the operating mode:

- MODE = 0: Check whether a request for providing a data record is present. If NEW = “1”, a new request is present. The SLOT parameter then identifies the transfer area in the user data interface, the data record number is at the INDEX parameter, and the number of bytes to be sent at the RLEN parameter.
- MODE = 1: Receive a request for a data record for any transfer area in the user data interface. The data record number is at the INDEX parameter and the number of bytes to be sent at the RLEN parameter.
- MODE = 2: Receive a request for a data record for a specific transfer area in the user data interface defined by the SLOT parameter. The data record number is at the INDEX parameter and the number of bytes to be sent at the RLEN parameter.
- MODE = 3: Provide the requested data record at the RECORD parameter and send a positive reply to the IO controller. The CODE1 and CODE2 parameters must be occupied by zero.

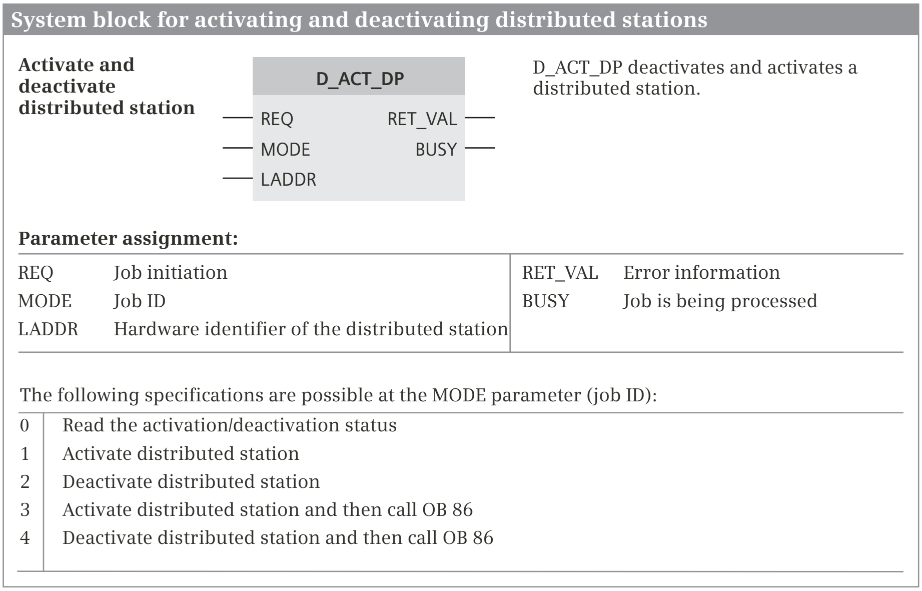
- ▷ MODE = 4: Reject the requested data record and send a negative reply to the IO controller. You transfer the error code at the CODE1 and CODE2 parameters.

PRVREC must first be called with MODE = 1 or MODE = 2 and subsequently with MODE = 3 or MODE = 4 within certain periods which depend on the CPU. The STATUS parameter contains the error information. SLOT and SUBSLOT are occupied identical to F_ID.

16.5.4 Activate/deactivate distributed station

D_ACT_DP Activate/deactivate distributed station

D_ACT_DP deactivates and activates a DP slave or an IO device and allows scanning of the deactivated or activated status. The system block is graphically displayed in Fig. 16.22. You find it in the program elements catalog under *Extended instructions* > *Distributed I/O*.



System block for activating and deactivating distributed stations

Activate and deactivate distributed station

D_ACT_DP deactivates and activates a distributed station.

Parameter assignment:

REQ	Job initiation	RET_VAL	Error information
MODE	Job ID	BUSY	Job is being processed
LADDR	Hardware identifier of the distributed station		

The following specifications are possible at the MODE parameter (job ID):

0	Read the activation/deactivation status
1	Activate distributed station
2	Deactivate distributed station
3	Activate distributed station and then call OB 86
4	Deactivate distributed station and then call OB 86

Fig. 16.22 Activate and deactivate distributed station

D_ACT_DP is called in the cyclic program; calling in the startup program is not supported. D_ACT_DP works asynchronously, i.e. processing of a job can extend over several program cycles. An activation or deactivation job is started by “1” in 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”.

After deactivation, a configured (and existing) station is no longer addressed by the DP master or the IO controller. The output terminals of deactivated output modules carry zero or a substitute value. The process image input of deactivated input modules is set to “0”.

A deactivated station can be removed from the bus without generating an error message; it is not signaled as faulty or missing. The call of the organization block OB 86 is omitted (station failure/restoration). You must not address the station from the program once it has been deactivated, since otherwise the organization block OB 122 (I/O access error) will be called with direct access operations, or the station will be signaled as not present when reading a data record with RDREC.

D_ACT_DP also activates a deactivated station again. The station is configured and parameterized by the DP master or IO controller as with a return of station. Upon activation, the organization block OB 86 is not started (Station failure/-restoration). If the BUSY parameter has signal state “0” following activation, the station can be addressed from the user program.

If you try to activate a station that is physically separated from the bus, an error message is output after the configured parameterization time elapses.

16.6 DPV1 interrupts

PROFIBUS DPV1 slaves (PROFIBUS) and correspondingly designed IO devices (PROFINET IO) can trigger the following interrupts:

- ▷ A status interrupt, if the station changes its operating state, for example.
- ▷ An update interrupt, if the station has been re-parameterized via the bus system or directly, for example.
- ▷ A manufacturer interrupt, if an event envisaged for this by the manufacturer occurs in the station.

Table 16.1 shows the number of the organization blocks and the event classes of the DPV1 interrupts. The constant names and the values are listed in the *System constants* tab of the default tag table. The name of the constant can be changed in the block properties.

Table 16.1 Organization blocks and event classes of the DPV1 interrupts

Interrupt	Status interrupt	Update interrupt	Manufacturer interrupt
OB number	55	56	57
Event class	Status	Update	Profile

Start information

A DPV1 interrupt organization block with the attribute *Optimized block access* activated provides the status information shown in Table 16.2 in the *Input* declaration section. A DPV1 interrupt organization block with the attribute *Optimized block*

Table 16.2 Start information for a DPV1 interrupt organization block

Declaration	Tag name	Data type	Description
The *Optimized block access* attribute is activated:			
Input	LADDR Slot Specifier	HW_IO UINT WORD	Hardware identifier Slot address of interrupt-triggering module Event identifier

access deactivated (OB with standard access) provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in 4.11.4 "Start information" on page 142. This contains additional tags, ranging from byte 5 to byte 11, which identify the component that triggers the interrupt. The assignment and occupation of the tags depends on the bus system used (PROFIBUS or PROFINET) (see operating instructions).

The additional interrupt information can be read using the system function block RALRM (see Chapter 5.7.7 "Reading additional interrupt information" on page 210).

Applying a DPV1 interrupt

A DPV1 interrupt can be triggered from a correspondingly configured distributed station. Under certain circumstances, alarm triggering must be enabled when the station is being configured or, if the DP slave can also handle the DPV0 mode, the DPV1 mode must be activated. Example: For an ET 200S station with the PROFIBUS interface IM 151-1 HF, DPV1 must be entered in the station properties under *Module parameters* > *Module parameters* in the *Interrupt mode* field.

Configuring a DPV1 interrupt

To configure a DPV1 interrupt in a CPU 1500, add an organization block with the event class *Status*, *Update* or *Profile* and enter the name, the programming language, and the number. In the properties of the organization block, you can also set the execution priority in addition to the general information under *Attributes*.

The interrupt information can be read in the DPV1 interrupt organization block with the system block RALRM *Read additional interrupt info*.

16.7 Isochronous mode

16.7.1 Introduction

Reference is made to isochronous mode if a program is executed synchronous to a PROFIBUS DP cycle or PROFINET IO cycle. In connection with equidistant (equally long) bus cycles, you thus obtain reproducible response times. The user program executed in isochronous mode is present in organization block OB 61 *Synchronous Cycle*. The system functions SYNC_PI and SYNC_PO are available for isochronous updating of the process image.

Isochronous mode interrupts are only processed in the RUN operating state. An isochronous mode interrupt in the STARTUP and STOP operating states is rejected.

16.7.2 Isochronous mode with PROFINET IO

The prerequisite for isochronous operation for PROFINET is IRT communication (Isochronous Real-Time). The send clock defined in the sync domain forms the basis for the time scale (data cycle) with which the I/O signals are read, processed and output (Fig. 16.23).

The data cycle is the interval at which the IRT transmission takes place on the subnet. The application cycle is the interval at which the isochronous mode OB is called.

Ti is the time required for reading the I/O signals. It includes the times for preparation of the I/O signals in the input modules or electronic modules, and for processing in the IO device.

Ti is followed by the data cycle. This begins with transmission of the I/O signals over the subnet. Transmission takes place in both directions; the input signals are transmitted to the controller station, and the output signals (from the previous application cycle) are transmitted to the IO devices.

The isochronous mode organization block assigned to the PROFINET IO system is called following a delay time during which the IRT transmission takes place. The

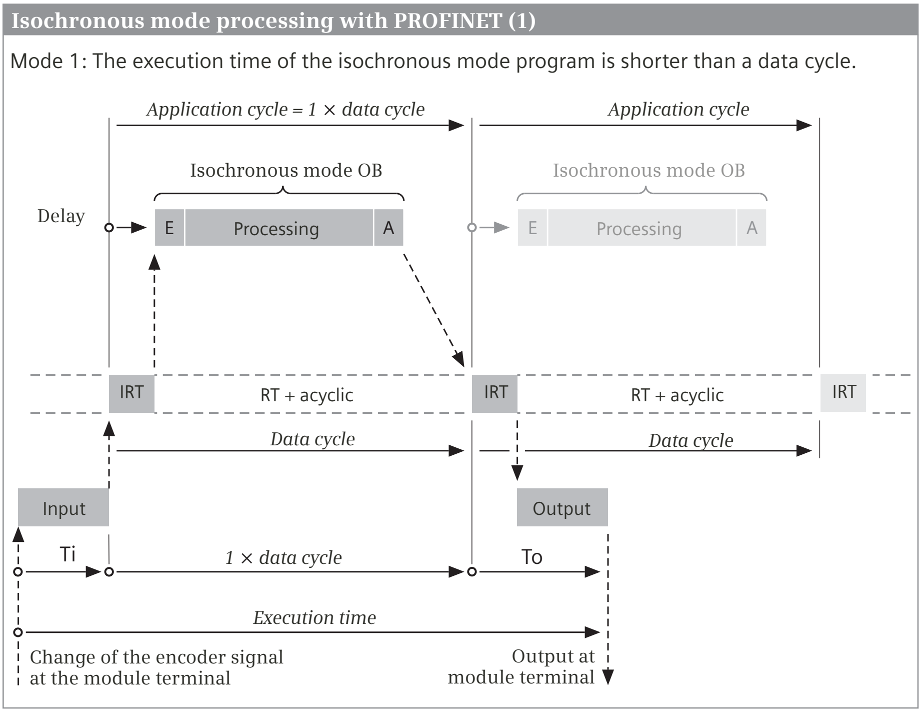

Isochronous mode processing with PROFINET (1)
Mode 1: The execution time of the isochronous mode program is shorter than a data cycle.
Application cycle = 1 × data cycle
Application cycle
Isochronous mode OB
Isochronous mode OB
Delay
E
Processing
A
E
Processing
A
IRT
RT + acyclic
IRT
RT + acyclic
IRT
Data cycle
Data cycle
Input
Output
Ti
1 × data cycle
To
Execution time
Change of the encoder signal at the module terminal
Output at module terminal

Fig. 16.23 Isochronous mode in the PROFINET IO system (1)

system block SYNC_PI must be called in the organization block in order to read the input signals in isochronous mode, and system block SYNC_PO in order to write the output signals in isochronous mode. The processing time of the isochronous mode OB must be (significantly) shorter than the application cycle time, for the main program is further processed during the differential time.

To begins at the end of the data cycle. To is the time required to output the I/O signals. It is made up of the transmission time on the subnet, the time for processing in the IO device, and the times for preparation of the I/O signals in the output modules or electronic modules.

With isochronous mode, a distinction is made between two types: The processing time of the isochronous mode program is (significantly) shorter than the time for one data cycle, or it is longer. In the first case, the isochronous mode OB can be called in every data cycle (shown in Fig. 16.23); in the second case, the cycle in which the isochronous mode OB is called – the application cycle – is a multiple of the data cycle (shown with factor 2 in Fig. 16.24).

If the isochronous mode OB is called in every data cycle – the "Application cycle factor" is then 1 – SYNC_PI for isochronous updating of the input signals is called first in the isochronous mode program. Then the signals are processed, followed by the output with SYNC_PO.

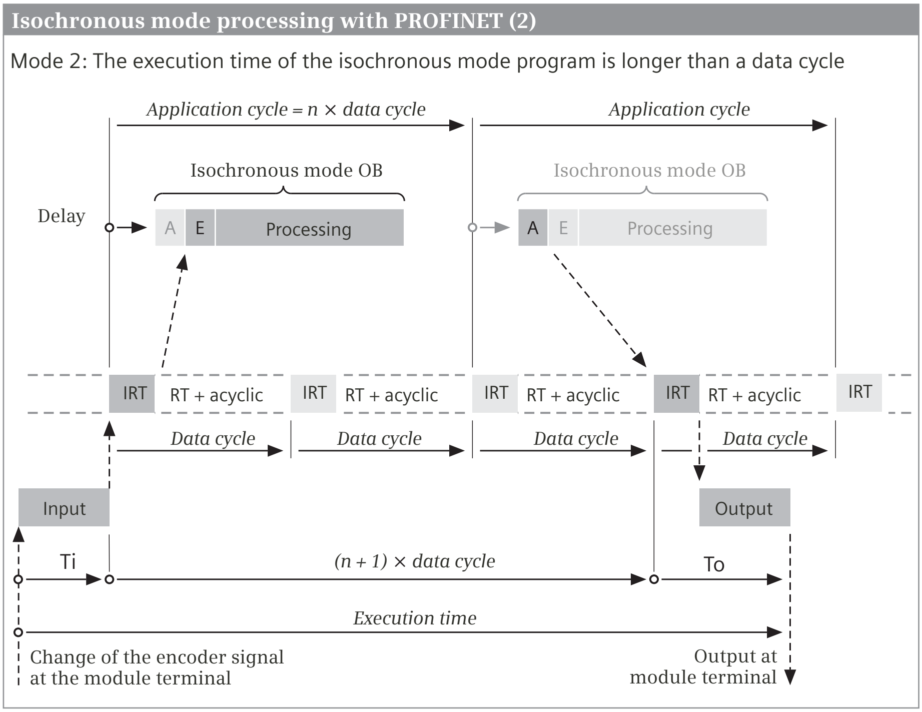

Isochronous mode processing with PROFINET (2)
Mode 2: The execution time of the isochronous mode program is longer than a data cycle
Application cycle = n × data cycle
Application cycle
Isochronous mode OB
Isochronous mode OB
Delay
A
E
Processing
A
E
Processing
IRT
RT + acyclic
IRT
RT + acyclic
IRT
RT + acyclic
IRT
RT + acyclic
IRT
Data cycle
Data cycle
Data cycle
Data cycle
Input
Output
Ti
(n + 1) × data cycle
To
Execution time
Change of the encoder signal at the module terminal
Output at module terminal

Fig. 16.24 Isochronous mode in the PROFINET IO system (2)

With this mode, the shortest response time between an input signal and the corresponding output signal is therefore the total of Ti, the data cycle time, and To. The longest response time occurs if the input signal changes shortly after the time for reading-in, and is the total of Ti, To, and twice the data cycle time.

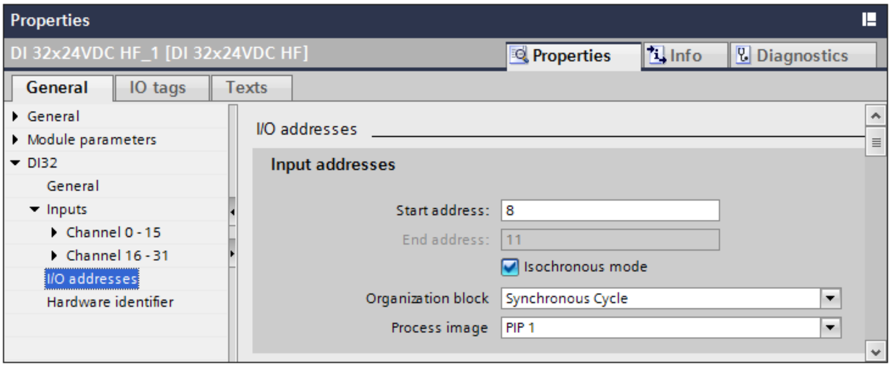
With an application cycle which takes longer than the data cycle (Fig. 16.24), you should select a different sequence for updating of the process image: Updating of the output signals first, then of the input signals, and then the processing. In this manner it is possible that the output signals are transmitted with the next possible data cycle (in the next application cycle) even if the data cycle time is short compared to the process image updating time.

With this mode, the shortest response time between an input signal and the corresponding output signal is therefore the total of Ti, the application cycle time, the data cycle time, and To. The longest response time occurs if the input signal changes shortly after the time for reading-in, and is the total of Ti, To, the data cycle time, and twice the application cycle time.

Configuring isochronous mode with PROFINET IO

A prerequisite for the configuration of isochronous mode is IRT communication and the corresponding functionality of the participating PROFINET components. Configure the PROFINET IO system with IRT communication as shown in Chapters 16.3.3 “Configuring PROFINET IO” on page 705 and 16.3.5 “Real-time communication in PROFINET” on page 710.

To activate the isochronous mode, open the IO device in the device view and activate the checkbox *Isochronous mode* in the properties of the input/output module under *I/O addresses*. In addition, assign the organization block for the isochronous mode interrupt (OB 61 *Synchronous Cycle*) and the relevant process image partition (e.g. PIP 1) to (Fig. 16.25).



I/O addresses

Input addresses

Start address: 8

End address: 11

☑ Isochronous mode

Organization block: Synchronous Cycle

Process image: PIP 1

Fig. 16.25 Configuration of isochronous mode in an IO device

Repeat the procedures for each module participating in the isochronous mode, in the other IO devices as well. In the properties of the IO device, set the method of determining the Ti/To values under *Isochronous mode*:

- *Automatic setting* if the configuration editor is to determine the values
- *From OB* if the settings in the isochronous mode organization block are to be imported
- *Manual* if you want to specify the values for Ti and To yourself

In the properties of the PROFINET subnet, you are given an overview of the set values and the list of the modules participating in the isochronous mode under *Overview isochronous mode*.

16.7.3 Isochronous mode with PROFIBUS

Constant bus cycle time

In the normal case, of the DP master controls the DP slaves assigned to it cyclically and without pauses. The time intervals may vary as a result of S7 communication, for example if the programming device carries out control functions over the PROFIBUS subnet. By using constant bus cycle times it is possible to achieve, for example, that outputs are always controlled via DP slaves at equal intervals. The DP master then always starts the bus cycles at equal intervals.

The use of constant bus cycle times is possible with the bus profiles “DP” and “User-defined”; SYNC/FREEZE groups must not be configured.

Isochronous mode

Reference is made to isochronous mode if a program is executed synchronous to the PROFIBUS DP cycle. In association with constant bus cycle times it is thus possible to achieve reproducible, response times of equal duration to the process I/O, which include the distributed recording of signals, signal transfer over PROFIBUS, and program execution including process image updating. The user program executed in isochronous mode is present in one of the organization blocks OB 61 to OB 64. The system functions SYNC_PI and SYNC_PO are available for isochronous updating of the process image.

The application of constant bus cycles is a prerequisite for isochronous mode. Isochronous mode is only possible with a DP master integrated in the CPU as the only active station on the PROFIBUS.

Fig. 16.26 shows the times involved in the isochronous mode. Ti is the time required for reading in the process values. It contains the execution time in the input modules or electronic modules and, in the case of modular DP slaves, the transfer time on the backplane bus. At the end of Ti, the input information for transfer using the global control command (GC) is available. The equidistant bus

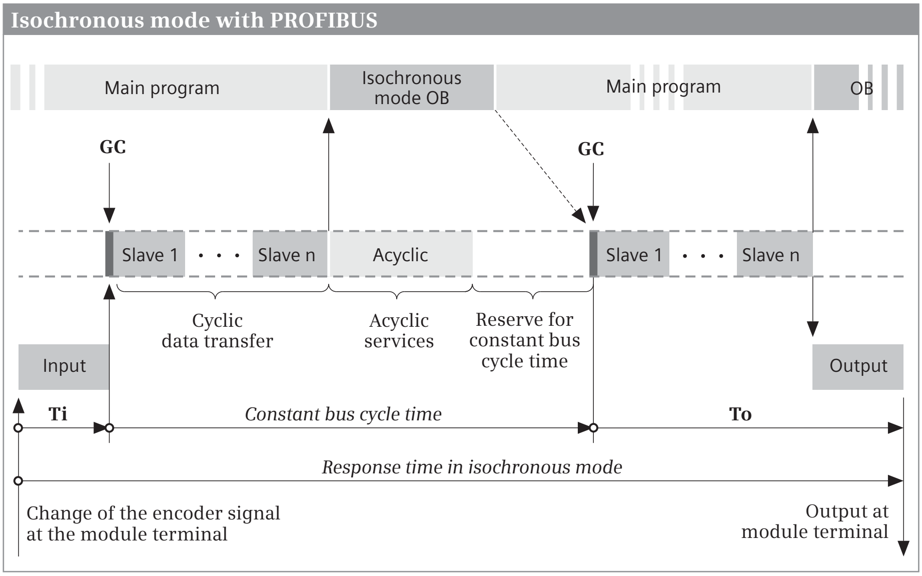

Isochronous mode with PROFIBUS
Main program
Isochronous mode OB
Main program
OB
GC
GC
Slave 1
Slave n
Acyclic
Slave 1
Slave n
Cyclic data transfer
Acyclic services
Reserve for constant bus cycle time
Input
Output
Ti
Constant bus cycle time
To
Response time in isochronous mode
Change of the encoder signal at the module terminal
Output at module terminal

Fig. 16.26 Response time with constant bus cycle time and isochronous mode

cycle then commences. This is the time between two global control commands and encompasses the transfer to the subnet as well as the execution of the isochronous interrupt OB. Between completion of the execution of this OB to the next global control command there must be time for execution of the main program.

To is the time required to output the process values. It begins with the global control command and comprises the transfer time on the subnet as well as the processing time in the output modules or electronic modules. In the case of modular DP slaves, the transfer time on the backplane bus is also added.

The minimum response time in the case of isochronous mode is the total of Ti, the bus cycle, and To. The maximum response time (Ti + To + 2 × bus cycle) occurs if a change in the input signal takes place shortly after the global control command.

Correspondingly designed DP slaves allow a reduction in the response time thanks to “overlapping isochronous mode”. This involves overlapped updating of the input and output signals (overlapping of Ti and To). In this case, the DP slave must not obtain the Ti/To values from the subnet. If isochronous modules have both inputs and outputs, overlapping of Ti and To is not possible.

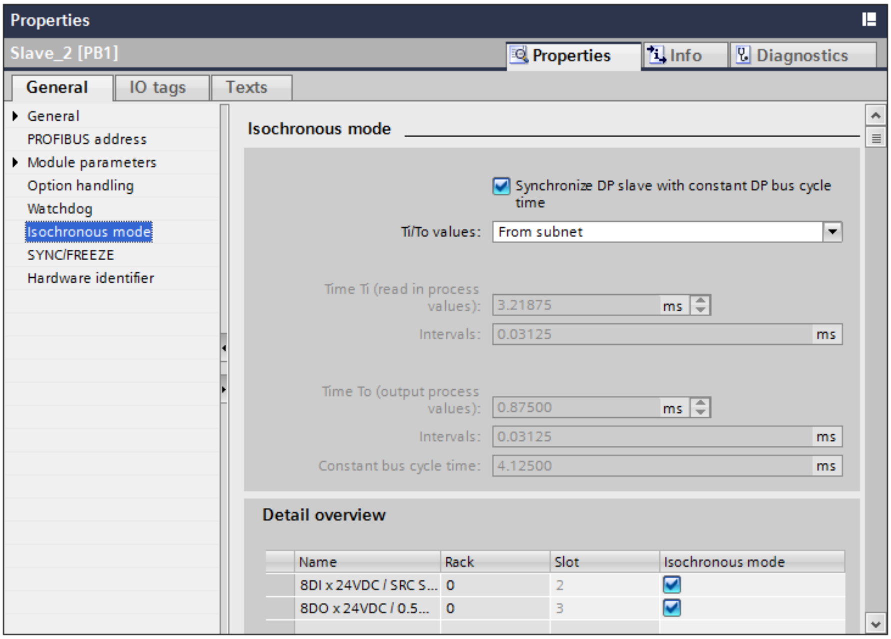
Configuring isochronous mode with PROFIBUS DP

A prerequisite for configuration of isochronous mode of the bus system is the constant bus cycle time and the corresponding functionality of the participating DP components. Configure the DP master system as shown in Chapter 16.4.3 “Configuring PROFIBUS DP” on page 721.

To switch on the constant bus cycle time and isochronous mode, activate the *Enable constant bus cycle time* checkbox in the properties of the PROFIBUS subnet or DP master system under *Constant bus cycle time*. Activate isochronous mode for the participating DP slaves in the *Detailed overview* section and, if you "open" a line with a DP slave, the isochronous mode of the individual I/O modules in the DP slave. In the *Ti/To values* column you can select the mode from a drop-down list for calculation of the Ti/To values:

- ▷ From subnet: The currently configured DP slave obtains the Ti/To values from the subnet and thus has the same values as the other DP slaves which also obtain their values from the subnet.
- ▷ Automatic minimum: If you manually change the Ti/To values of another DP slave when in this setting, any adaptations which may be necessary on the currently configured DP slave are carried out automatically.
- ▷ Manual: With this setting, you manually enter the Ti/To values for the currently configured DP slave.

You can also make these settings in the interface properties of the DP slave under *Isochronous mode*. Each module or submodule involved in isochronous mode must be addressed in a process image partition. You set the process image partition for the module in the Device view in the module properties under I/O addresses (Fig. 16.27).



Isochronous mode

☑ Synchronize DP slave with constant DP bus cycle time

Ti/To values: From subnet

Time Ti (read in process values): 3.21875 ms

Intervals: 0.03125 ms

Time To (output process values): 0.87500 ms

Intervals: 0.03125 ms

Constant bus cycle time: 4.12500 ms

Detail overview

Name	Rack	Slot	Isochronous mode
8DI x 24VDC / SRC S...	0	2	☑
8DO x 24VDC / 0.5...	0	3	☑

Fig. 16.27 Activation of isochronous mode in a DP slave

16.7.4 Isochronous mode interrupt

The “Isochronous mode” function permits synchronous reading, processing and output of I/O signals in a fixed (equidistant) cycle. The user program executed in isochronous mode is present in organization block OB 61 *Synchronous Cycle*. The system blocks SYNC_PI and SYNC_PO are available for isochronous updating of the process image.

Start information

A synchronous interrupt organization block with the attribute *Optimized block access* activated provides the start information shown in Table 16.3 in the *Input* declaration section. A synchronous interrupt organization block with the attribute *Optimized block access* deactivated (OB with standard access) provides 20-byte long start information in the *Temp* declaration section, the standard structure of which is described in 4.11.4 “Start information” on page 142. The GC_VIOL and MISSED_EXEC tags allow you to recognize faulty isochronous processing. Default settings can be programmed in the first cycle identified by the FIRST tag. The DP_ID tag indicates the master system from which the isochronous mode interrupt OB was called.

Table 16.3 Start information for an isochronous mode interrupt organization block

Declaration	Tag name	Data type	Description
The *Optimized block access* attribute is activated:			
Input	Initial_Call PIP_Input PIP_Output IO_System Event_Count SyncCycleTime	BOOL BOOL BOOL USINT INT LTIME	With ”1”: First call of the organization block With ”1”: The process image partition of the inputs is up to date With ”1”: The process image partition of the outputs is up to date Number of the interrupt-triggering I/O system Number of cycles lost Calculated cycle time
The *Optimized block access* attribute is deactivated (standard access):			
Temp	GC_VIOL FIRST MISSED_EXEC DP_ID	BOOL BOOL BYTE BYTE	GC violation First execution of the organization block Number of discarded OB calls Number of the interrupt-triggering I/O system

Configuring an isochronous mode interrupt

To configure an isochronous mode interrupt, add an organization block with the event class *Synchronous Cycle* and enter the name, programming language and number. In addition to the general information and the block attributes, you can set the execution priority and the overload behavior in the properties of the organization block. You can change the preset priority 21 from 16 to 26 under *Priority*. Under *Event queuing*, you define the response in the event of an overload (see section “Overload behavior” on page 194).

The application cycle and the delay time are set under *Isochronous mode*. The application time for PROFINET is an integral multiple of the send clock (possible values 1 to 14) and for PROFIBUS it is the global control command. If the *Automatic setting* checkbox is activated, the delay time is automatically calculated. If it is deactivated, you can manually specify the delay time between the send clock or global control command and the start of the organization block.

In the isochronous mode OB, call the system blocks SYNC_PI prior to the interrupt routine and SYNC_PO after the interrupt routine. These functions update the process image partition of those inputs and outputs you are using in the interrupt routine. When configuring these modules, you must apply their addresses to the process image partition assigned to the isochronous mode interrupt OB.

Caution: In the interrupt routine itself you may only work with the inputs and outputs of the process image partition. Direct access to the I/O addresses assigned to the process image partition is not permissible!

Behavior in the STOP and STARTUP operating states

An isochronous mode interrupt is only processed in the RUN operating state. A isochronous mode interrupt in the STOP or STARTUP operating states is rejected. The number of OB calls which have not been executed is indicated in the start information of the isochronous mode interrupt OB called for the first time in RUN.

Error handling

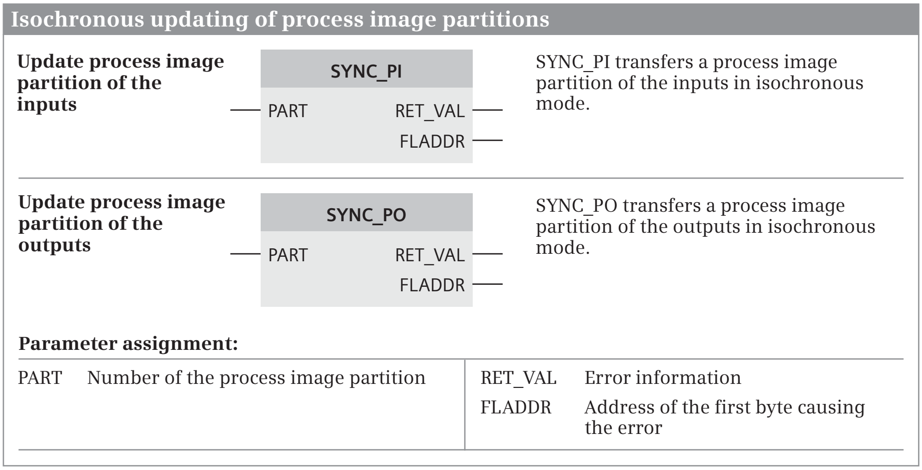
If the isochronous mode interrupt organization block is not present when an isochronous mode interrupt arrives, the operating system ignores the event with global error handling.

If an isochronous mode interrupt arrives before the associated isochronous mode interrupt OB has been terminated, the overload behavior configured in the block properties is activated (see also section “Overload behavior” on page 194).

16.7.5 Isochronous process image updating

For the isochronous and data-consistent update of the process image partitions, the system blocks SYNC_PI *Update inputs in isochronous mode* and SYNC_PO *Update outputs in isochronous mode* are available. You can find the system blocks in the program elements catalog under *Extended instructions* > *Process image*. Fig. 16.28 shows the graphic representation of the system functions.

The system function **SYNC_PI** updates a process image partition of the inputs. The system function **SYNC_PO** updates a process image partition of the outputs. Updating is carried out isochronously and data-consistent. The two system functions may only be called in an isochronous mode interrupt OB. At the PART parameter you define the process image partition to be updated and that is assigned to this organization block.



Isochronous updating of process image partitions

Update process image partition of the inputs

SYNC_PI transfers a process image partition of the inputs in isochronous mode.

Update process image partition of the outputs

SYNC_PO transfers a process image partition of the outputs in isochronous mode.

Parameter assignment:

PART	Number of the process image partition	RET_VAL	Error information
		FLADDR	Address of the first byte causing the error

Fig. 16.28 System blocks for isochronous updating of process image partitions

A process image partition which is updated with the system blocks SYNC_PI and SYNC_PO cannot be automatically updated or updated with the system blocks UPDAT_PI and UPDAT_PO and it cannot be accessed by means of direct accessing.

17 Communication

17.1 Overview

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 or an HMI device. The stations are connected either to a bus system or to a point-to-point connection. 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*.

A PLC station with a CPU 1500 can exchange data with other stations using various methods. The connection is made

- With Industrial Ethernet via the PROFINET interface on the CPU or via the CP 1543-1 communication module
- With PROFIBUS via the PROFINET interface on the CPU 1516 or via the CM 1542-5 communication module
- Point-to-point communication is implemented via the communication modules CM PtP RS232 BA, CM PtP RS232 HF, CM PtP RS422/485 BA, and CM PtP RS422/485 HF

The connection to a subnet is made via the interfaces integrated on the CPU or via communication modules. Communication is controlled by the operating system of the CPU or CM module, possibly supported by *communication functions*. These are either system blocks which are called in the control program, or loadable function blocks.

Note that a CPU 1500 has a limited number of “connection resources”. Table 2.1 on page 50 shows the possible number of simultaneously established connections.

Data exchange with the distributed I/O (PROFIBUS DP and PROFINET IO) is described in Chapter 16 “Distributed I/O” on page 696.

A prerequisite for configuration of communication is a created project with the PLC stations involved in the communication. Chapter 3 “Device configuration” on page 61 describes how to create a project and configure the PLC stations.

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 (switch) 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. Data can be transferred via Industrial Ethernet, for example, with open user communication, S7 communication, or Modbus TCP.

Data transmission with open user communication

Open user communication transfers data between two PLC stations that are connected together via an Ethernet network. Data transfer can be implemented using the TCP, UDP and ISO-on-TCP protocols. The communication function TSEND_C is available in the sending station and TRCV_C is available in the receiving station for setting up a connection and transferring data.

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”). For one-way data exchange, 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. For two-way data exchange, the communication functions USEND and BSEND are called in the transmitting station and URCV and BRCV are called in the receiving station.

Data transmission with a point-to-point connection

A serial, character-based point-to-point connection allows for a very free definition of the protocol. For example, a printer, barcode reader or modem on a CPU 1500 can be operated with the transmission standards RS232 or RS485 via a CM PTP RS 232 or CM PTP RS422/485 communication module. The ASCII protocol and the communication functions for Modbus RTU and USS drives are available for point-to-point communication.

Configuration of communication with STEP 7

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.

In Fig. 17.1 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 *PN/IE_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 82). The connection of a programming device is described in Chapter 15.1 “Connection of a programming device to the PLC station” on page 648.

The PLC and HMI stations to be networked must be located together in one project.

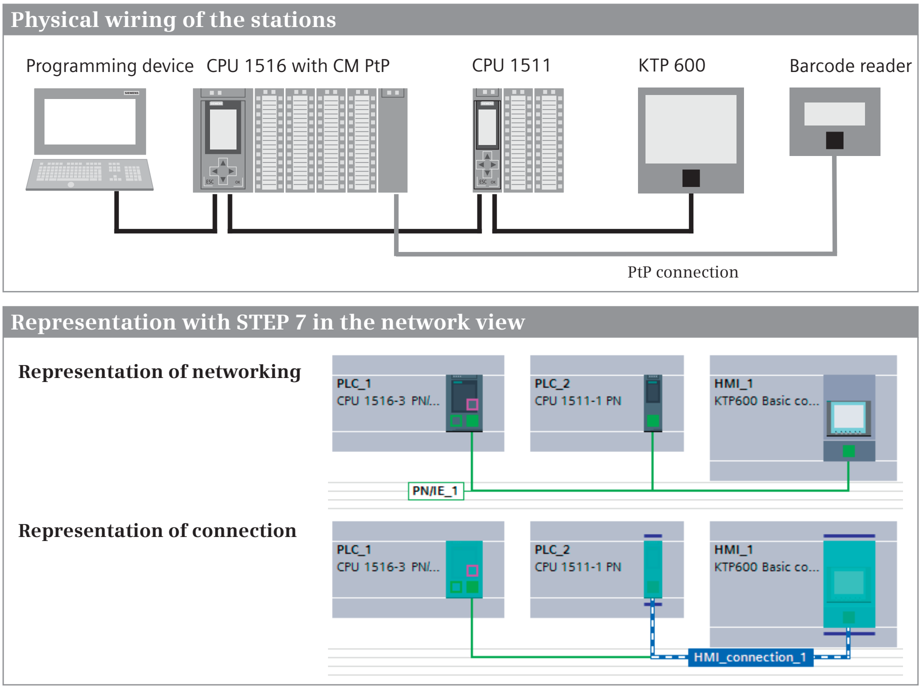


Fig. 17.1 Comparison of the wiring and configuration

17.2 Open user communication

17.2.1 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.

Prerequisite for open user communication is the networking of the participating stations via an Ethernet network, either using the integrated CPU interface and a communication module. The communication connection can be configured either during the network configuration or be programmed with the communication functions in a data block. If the TCP and UDP protocols are used, this data block has the structure of the system data type TCON_IP_v4; for the ISO-on-TCP protocol, it has the structure of the system data type TCON_IP_RFC.

For open user communication, you use the communication functions TSEND_C and TRCV_C in the user program. These functions establish a programmed connection, control the data traffic, and terminate a programmed connection or reset a configured connection.

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. The applications are addressed via port numbers. TCP is used if the communication partner does not support a connection via ISO-on-TCP. For such communication partners, enter “unspecified” as the partner end point in the connection parameterization.

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 without connection via the IP address and a port.

ISO Transport over TCP (RFC 1006)

With the RF 1006 (ISO-on-TCP) protocol, ISO applications can be adopted into the TCP/IP network. It is suitable for medium to large volumes of data (up to 8192 bytes) with a dynamic length. It is routing-capable and can be used in a wireless network. 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).

17.2.2 Data structure of open user communication

Fig. 17.2 shows the data structure of open user communication.

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 types TCON_IP_v4 (for the protocols TCP and UDP) or TCON_IP_RFC (for the protocol ISO-on-TCP) 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.

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. For these actions, TSEND_C and TRCV_C internally use, for example, the communication functions TCON, TDISCON, TSEND and TRCV or TUSEND and TURCV. These communication functions can also be used individually; however, they are not described in this book.

The communication functions TSEND_C and TRCV_C can be found in the program elements catalog under *Communication* > *Open user communication*. Fig. 17.3 shows the calls of the functions in ladder logic representation.

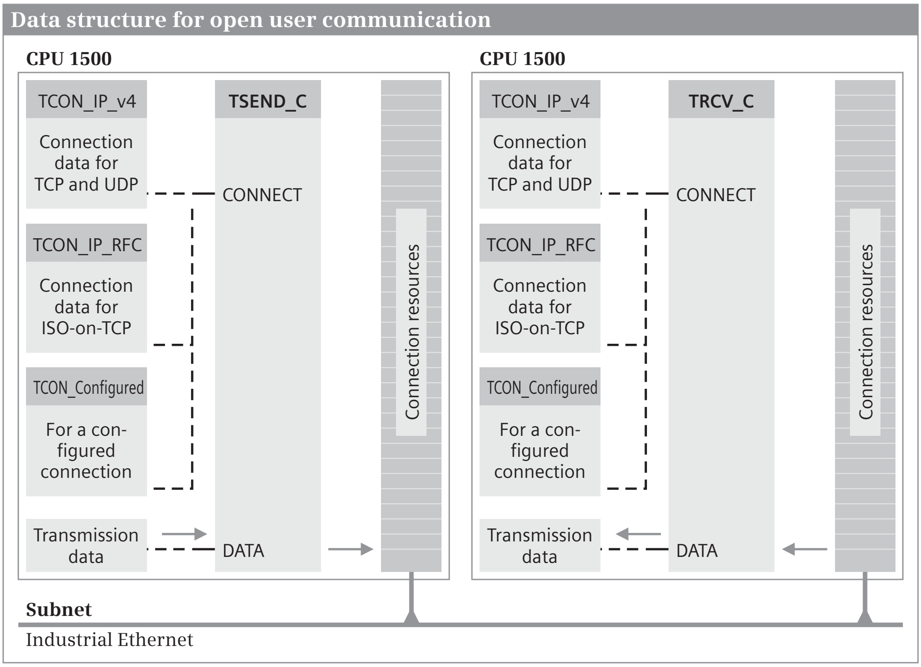
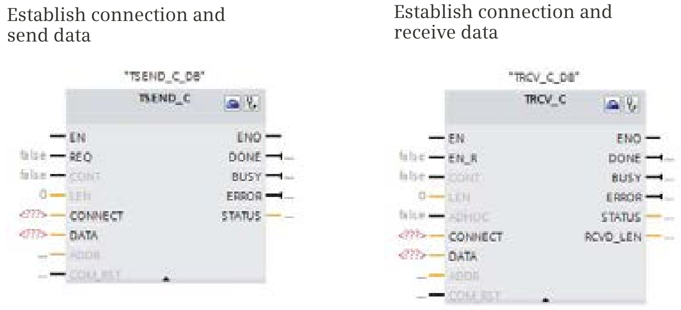


Fig. 17.2 Communication functions TSEND_C and TRCV_C

Calls of the communication functions for open user communication


Establish connection and send data

Establish connection and receive data

Fig. 17.3 Calls of the communication functions for open user communication (LAD)

17.2.3 Establish connection and send data with TSEND_C

The communication function TSEND_C establishes a connection, sends data with the TCP, UDP or ISO-on-TCP protocols, and also terminates the connection.

Control data transmission

TSEND_C works asynchronously, i.e. processing of a job may extend over several program cycles. You can control the establishing of the connection and the data transfer using the parameters CONT, REQ and COM_RST. The data transmission status is indicated in the parameters BUSY, NDR, DONE, ERROR, and STATUS. You must evaluate these parameters immediately after each execution of the communication function since they only remain valid until the next call.

In the initial state, the parameters CONT, REQ and COM_RST are assigned signal state “0” and no data is sent. If the signal state at CONT switches from “0” to “1”, a configured connection is checked or a programmed connection is established. After the job is successfully completed, the parameter DONE is set to “1” for the duration of one program cycle. Data is sent via an established connection if a rising edge occurs at the parameter REQ. After the transfer is successfully completed, DONE is set to “1” for the duration of one program cycle.

If the signal state at CONT switches from “1” to “0”, a configured connection is reset or a programmed connection is terminated. After the job is successfully completed, the parameter DONE is set to “1” for the duration of one program cycle. If the signal state at COM_RST switches from “0” to “1”, any running data transfer is aborted and the connection is reset. After this, COM_RST is reset to signal state “0”.

The parameter BUSY has signal state “1”, indicating that the send job has not yet been ended and a new job cannot be initiated. The ERROR parameter signals with signal state “1” that the started job has been completed with errors. It is only set for the duration of one program cycle. The STATUS parameter contains intermediate states or error information.

Specifying send data

The parameter DATA points to the send mailbox for the transmission data. An absolutely or symbolically addressed tag or a type data block can be used as actual parameter. If the send mailbox is located in a data block with standard access (the attribute *Optimized block access* is deactivated), then a data area that is absolutely addressed with an ANY pointer can also be addressed.

The parameter LEN specifies the maximum number of bytes sent. If the send mailbox is located in a data block with optimized access, the value zero must be used at LEN.

Specifying a connection

The CONNECT parameter points to a data block or a tag with the description of the connection parameters. The structure depends on the type of connection and the protocol used and is defined by system data types:

- For a programmed connection and the TCP and UDP protocols, it is the system data type TCON_IP_v4.
- For a programmed connection and the ISO-on-TCP protocol, it is the system data type TCON_IP_RFC.
- For a configured connection, it is the system data type TCON_Configured.

The ADDR parameter is used for a UDP connection and contains the IP address and the port number of the partner station with the data structure of the system data type TADDR_Param.

17.2.4 Establish connection and receive data with TRCV_C

The communication function TRCV_C establishes a connection, receives data with the TCP, UDP or ISO-on-TCP protocols, and also terminates the connection.

Control data transmission

TRCV_C works asynchronously, i.e. processing of a job may extend over several program cycles. You can control the establishing of the connection and the data transfer using the parameters EN_R, CONT, ADHOC and COM_RST. The data transmission status is indicated in the parameters BUSY, DONE, ERROR and STATUS. You must evaluate these parameters immediately after each execution of the communication function since they only remain valid until the next call.

In the initial state, the parameters CONT, EN_R, ADHOC and COM_RST are assigned signal state “0” and no data is received. If the signal state at CONT switches from “0” to “1”, a configured connection is checked or a programmed connection is established. After the job is successfully completed, the parameter DONE is set to signal state “1” for the duration of one cycle. Data is only received via an established connection if the parameter EN_R is assigned “1”. After the transfer is successfully completed, DONE is set to signal state “1” for the duration of one cycle.

If the signal state at CONT switches from “1” to “0”, a configured connection is reset or a programmed connection is terminated. After the job is successfully completed, the parameter DONE is set to “1” for the duration of one program cycle. If the signal state at COM_RST switches from “0” to “1”, any running data transfer is aborted and the connection is reset. After this, COM_RST is reset to signal state “0”.

The parameter BUSY has signal state “1”, indicating that the receive job has not yet been ended and a new job cannot be initiated. The ERROR parameter signals with signal state “1” that the started job has been completed with errors. It is only set for the duration of one program cycle. The STATUS parameter contains intermediate states or error information.

Specifying receive data

The parameter DATA points to the receive mailbox for the transmission data. An absolutely or symbolically addressed tag or a type data block can be used as actual parameter. If the send mailbox is located in a data block with standard access (the attribute *Optimized block access* is deactivated), then a data area that is absolutely addressed with an ANY pointer can also be addressed.

The LEN parameter specifies the maximum number of bytes to be received. When transferring data with the TCP protocol and ADHOC = “0” or with the ISO-on-TCP protocol, the data is received with the length specified at parameter LEN. The actual number of bytes received at the RCVD_LEN parameter corresponds to the number of bytes at the LEN parameter.

With the TCP protocol, the data is received with a dynamic length if the parameter ADHOC has signal state “1”. The actual number of bytes received is output at the parameter RCVD_LEN.

Specifying a connection

The CONNECT parameter points to a data block or a tag with the description of the connection parameters. The structure depends on the type of connection and the protocol used and is defined by system data types:

▷ For a programmed connection and the TCP and UDP protocols, it is the system data type TCON_IP_v4.

▷ For a programmed connection and the ISO-on-TCP protocol, it is the system data type TCON_IP_RFC.

▷ For a configured connection, it is the system data type TCON_Configured.

The ADDR parameter is used for a UDP connection and contains the IP address and the port number of the partner station with the data structure of the system data type TADDR_Param.

17.2.5 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 a network” on page 73. Open user communication requires either a connection that is configured with the network configuration or a connection that is programmed using the program editor.

Configuring a communication connection

In the network view, click on the *Connections* button and select the desired connection from the drop-down list: ISO-on-TCP, TCP or UDP connection. The stations that support the selected connection are highlighted. You can create a connection between two stations by left-clicking on one station and “dragging” the connecting line to the other station (see Chapter 3.4.5 “Configuring a connection” on page 78). The connection is displayed in the graphic and in the connection table in the lower part of the working window. If the connection is selected, the inspector window shows the connection properties in the *Properties* tab.

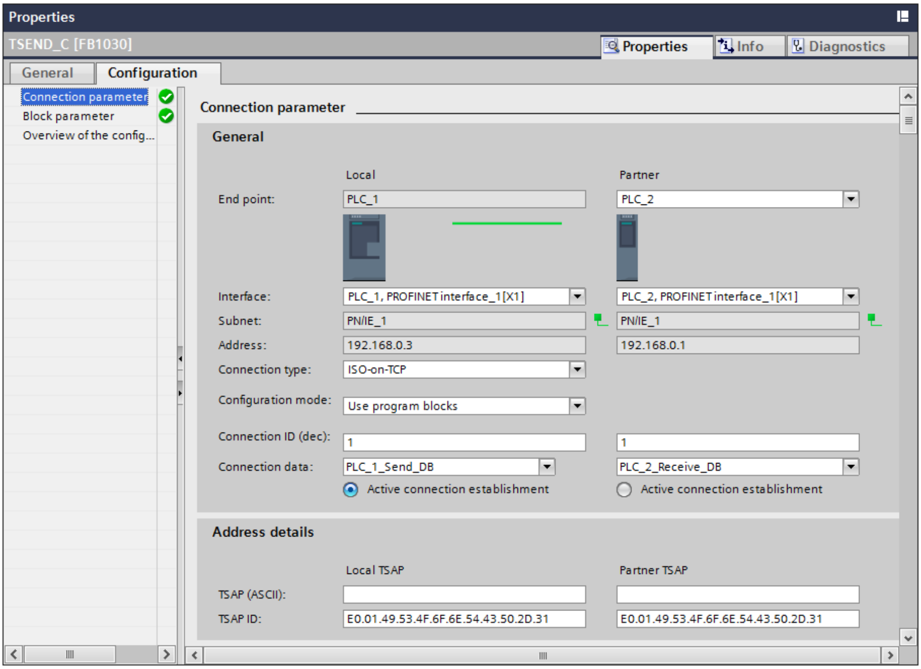
Programming the communication functions

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. Under *Properties* in the *Configuration* tab in the inspector window, the program editor shows the block parameters and the connection parameters (Fig. 17.4).

If you want to use a configured connection for open user communication, select *Use configured connection* from the *Configuration mode* drop-down list. Then click on the button with the three dots under *Connection name*. The *Select connection* dialog is displayed with the already configured connections. Select one of the displayed connections and confirm the selection by clicking on *OK*. After this, the data of the configured connection is imported into the connection parameters. Fields highlighted in red in the connection parameters must be filled in. You must now supply the remaining parameters at the block call in the working window of the program editor.

If you want to use a programmed connection for open user communication, select *Use program blocks* from the *Configuration mode* drop-down list. Then select the connection partner from the *Partner* drop-down list. If you select the entry *<new>* from the *Connection data* drop-down list for both stations, a new data block with the connection data is created. Then select the type of connection from the *Connection type* drop-down list. You must now supply the remaining parameters at the block call in the working window of the program editor.



Connection parameter

General

	Local	Partner
End point:	PLC_1	PLC_2
Interface:	PLC_1, PROFINET interface_1[X1]	PLC_2, PROFINET interface_1[X1]
Subnet:	PN/IE_1	PN/IE_1
Address:	192.168.0.3	192.168.0.1
Connection type:	ISO-on-TCP	
Configuration mode:	Use program blocks	
Connection ID (dec):	1	1
Connection data:	PLC_1_Send_DB	PLC_2_Receive_DB
	● Active connection establishment	○ Active connection establishment

Address details

	Local TSAP	Partner TSAP
TSAP (ASCII):		
TSAP ID:	E0.01.49.53.4F.6F.6E.54.43.50.2D.31	E0.01.49.53.4F.6F.6E.54.43.50.2D.31

Fig. 17.4 Configuring the connection parameters for open user communication

Further entries in the connection parameters

The *Connection ID* identifies the connection and must be the same in both stations. (Multiple connections between the partners can be created.)

For the connection data, you can also create a type data block or a tag in a global data block with the system data type TCON_IP_v4 or TCON_IP_RFC and preset it with the corresponding connection properties. Specify this data block or this tag as the actual parameter at the CONNECT parameter and enter the type or global data block in the *Connection data* field. At the ADDR parameter (for a UDP connection), you can create a type data block or a tag with the system data type TADDR_Param.

Use the *Active connection establishment* 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”.

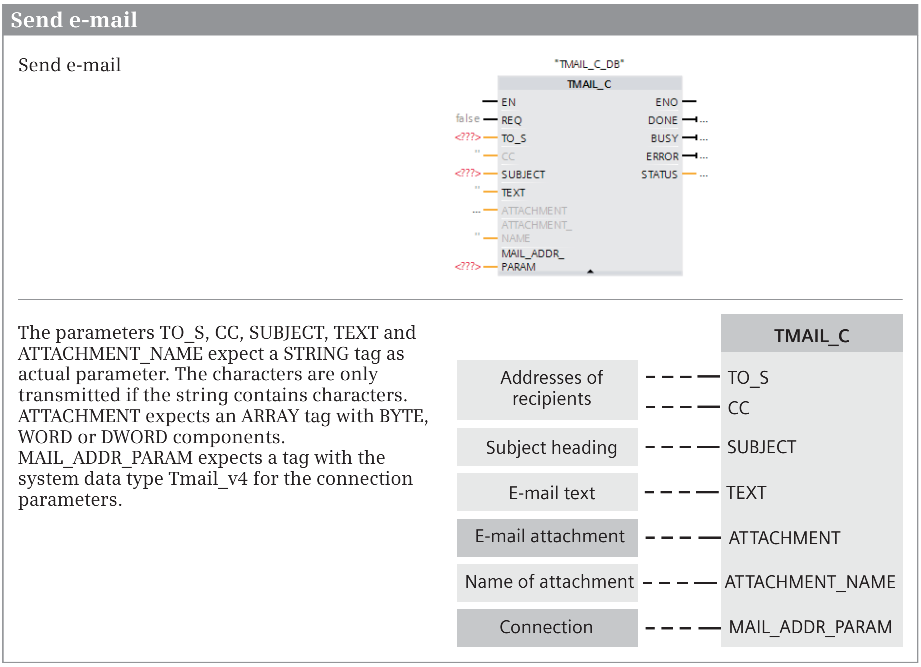
If you enter a new TSAP, first enter the character sequence in the field *TSAP (ASCII)*. You then add the characters “E0.01.” before it in the *TSAP ID* field. In the *TSAP (ASCII)* field, the TSAP is no longer displayed because the first character (E0) is not an ASCII character.

17.2.6 Further functions of open user communication

Send e-mail

TMAIL_C sends an e-mail via an Ethernet interface in an S7-1500 station. The station must have a connection to the dial-up server of the Internet Service Provider via the network. You can find TMAIL_C in the program elements catalog under *Communication* > *Open user communication*. Fig. 17.5 shows the call of TMAIL_C in LAD representation.

TMAIL_C works asynchronously, i.e. processing of a job extends over several program cycles. You initiate an e-mail transfer with a rising edge at parameter REQ. As long as the parameter BUSY has signal state “1”, the job is still in progress. If DONE = “1”, the job has been completed. If an error occurred while the job was processing, ERROR = “1” and STATUS has the associated error information.



The parameters TO_S, CC, SUBJECT, TEXT and ATTACHMENT_NAME expect a STRING tag as actual parameter. The characters are only transmitted if the string contains characters. ATTACHMENT expects an ARRAY tag with BYTE, WORD or DWORD components. MAIL_ADDR_PARAM expects a tag with the system data type Tmail_v4 for the connection parameters.

Fig. 17.5 Sending an e-mail with TMAIL_C

Write the addresses of the receivers at parameters TO_S and CC in a STRING tag with a maximum length of 180 characters. The e-mail address is in angle brackets, preceded by a space (blank). You can separate several addresses with a comma. The assignment of CC is optional.

The subject and the text of the e-mail at parameters SUBJECT and TEXT are each contained in a STRING tag with a maximum length of 180 characters. The assignment of TEXT is optional.

If a tag is specified at parameter ATTACHMENT, its contents are sent as an attachment with the e-mail. The tag must have the data type ARRAY with BYTE, WORD or DWORD components and can have a maximum length of 64 KB. The filename of the attachment is located at ATTACHMENT NAME in a STRING tag with a maximum length of 50 characters. Specifying this information is optional. For a blank string, the attachment is given the filename "attachment.bin".

MAIL_ADDR_PARAM expects a tag with the system data type Tmail_v4 if the integrated PN interface of the CPU is used (addressing via the IP address according to IPv4). If the e-mail is sent via a CP module, there are also tags with the system data types Tmail-v6 (addressing via the IP address according to IPv6) or Tmail_FQDN (addressing via fully qualified domain names).

Configuring a PN interface with T_CONFIG

T_CONFIG configures the integral PROFINET interface of the CPU. A prerequisite is that the *Set IP address using a different method* option was set during parameterization of the PROFINET interface with the hardware configuration when assigning the IP parameters. You can find T_CONFIG in the program elements catalog under *Communication* > *Open user communication* > *Others*. Fig. 17.6 shows the call of the function in LAD representation.

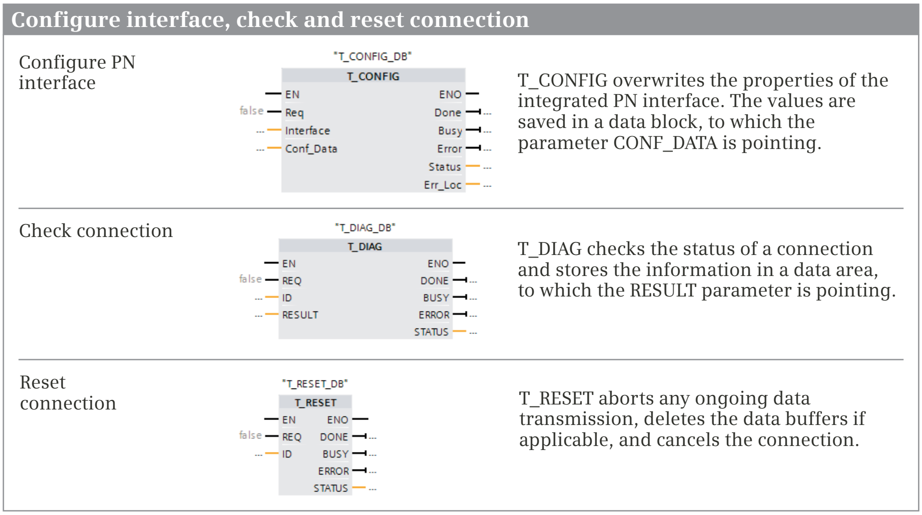
The adjustable parameters are the IP address, subnet mask, and router address. If the station is an IO device, the PROFINET device name can also be changed.

IP_CONF works asynchronously, i.e. processing of a job can extend over several program cycles. The job is initiated with a rising edge on the parameter REQ. 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 identifier of the PN interface at the INTERFACE parameter. STEP 7 specifies the hardware identifier during configuration and lists it in the *System constants* tab of the default tag table. The CONF_DB parameter is a pointer to the configuration data.

Checking the connection with T_DIAG

T_DIAG checks the status of the connection which has its connection ID specified at Parameter ID. You can find T_DIAG in the program elements catalog under



Configure interface, check and reset connection

Configure PN interface

T_CONFIG overwrites the properties of the integrated PN interface. The values are saved in a data block, to which the parameter CONF_DATA is pointing.

Check connection

T_DIAG checks the status of a connection and stores the information in a data area, to which the RESULT parameter is pointing.

Reset connection

T_RESET aborts any ongoing data transmission, deletes the data buffers if applicable, and cancels the connection.

Fig. 17.6 Setting a PN interface with T_CONFIG

Communication > *Open user communication* > *Others*. Fig. 17.6 shows the call of the function in LAD representation.

TDIAG works asynchronously, i.e. processing of a job extends over several program cycles. You initiate the check with a rising edge at parameter REQ. As long as the parameter BUSY has signal state “1”, the job is still in progress. If DONE = “1”, the job has been completed. If an error occurred while the job was processing, ERROR = “1” and STATUS has the associated error information.

The read information is saved in the data area to which the parameter RESULT is pointing.

Resetting a connection with T_RESET

T_RESET resets the connection which has its connection ID specified at Parameter ID. You can find T_RESET in the program elements catalog under *Communication* > *Open user communication* > *Others*. Fig. 17.6 shows the call of the function in LAD representation

T_RESET works asynchronously, i.e. processing of a job extends over several program cycles. You initiate the check with a rising edge at parameter REQ. As long as the parameter BUSY has signal state “1”, the job is still in progress. If DONE = “1”, the job has been completed. If an error occurred while the job was processing, ERROR = “1” and STATUS has the associated error information.

T_RESET interrupts any data transfer that might be running, empties the buffer for sending and receiving data, if applicable, and terminates the connection. For a configured or programmed connection, the end points of the connection are retained. The active connection partner then establishes the connection again.

17.3 S7 communication

17.3.1 Basics

S7 communication transfers large data quantities between PLC stations. The stations are connected to one another over an Ethernet or PROFIBUS subnet. Chapters 3.4.6 “Configuring a PROFINET subnet” on page 80 and 3.4.7 “Configuring a PROFIBUS subnet” on page 84 describe how to create such a subnet. The communication connections are static and are configured in the connection table.

For a one-way data exchange, only one PLC station requires a communication function. In the other (remote) PLC station, the operating system takes over the data transfer. For a two-way data exchange, the data transfer takes place between one communication function in one PLC station and a second communication function in the other (remote) PLC station.

17.3.2 One-way data exchange

In the case of one-way data exchange, the call of the communication function is only present in one CPU. In the partner CPU, the operating system controls the required data exchange (Fig. 17.7).

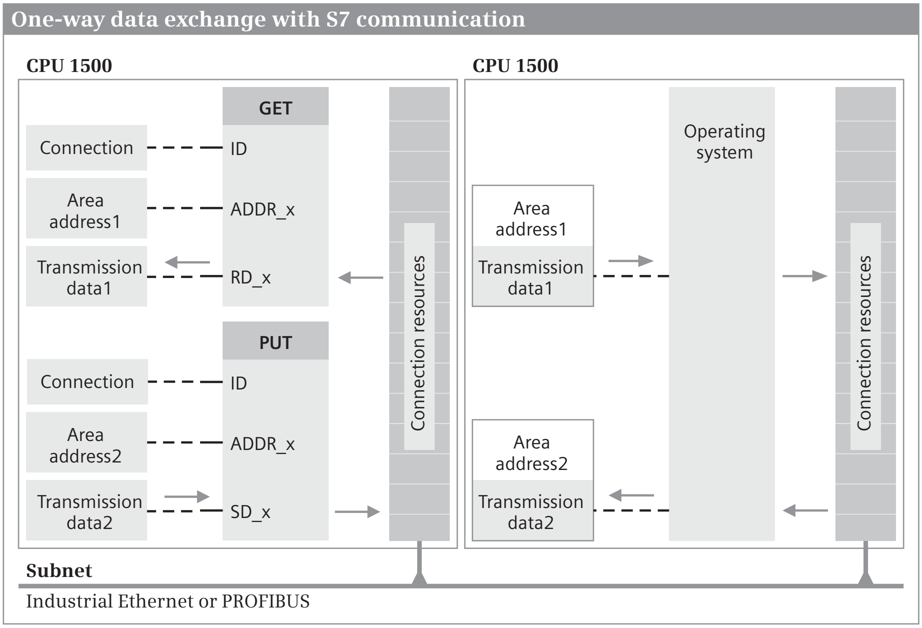


Fig. 17.7 Data structure for one-way data exchange

The following communication functions are available for one-way data exchange:

- ▷ GET Read data from a partner CPU
- ▷ PUT Write data to a partner CPU

The graphic representation of the block calls is shown in Fig. 17.8.

Calls of the communication functions for S7 communication

GET: Read data
for one-way data exchange

PUT: Write data
for one-way data exchange

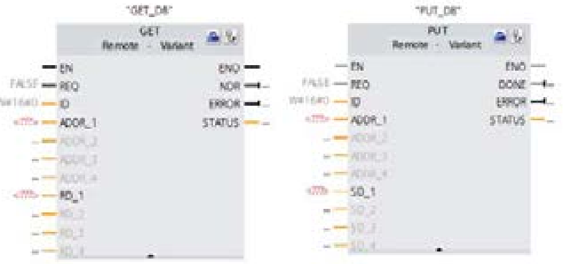


Fig. 17.8 Communication functions for one-way data exchange

The data read using GET is combined in the partner CPU by the operating system; the data written using PUT is distributed by the operating system in the partner CPU. A send or receive (user) program is not required in the partner CPU. The partner CPU can perform the required communication services both in RUN and STOP. The size of the consistently transferred data blocks depends on the partner CPU used and the number of parameters used (SD_n or RD_n).

In the partner CPU, access with GET or PUT must be permitted. You can find the setting in the CPU properties under *Protection* and *Connection mechanisms*: Activate the checkbox *Permit access with PUT/GET communication from remote partner*. GET and PUT can only address data areas in blocks with standard access; the attribute *Optimized block access* must be deactivated.

Common parameters

A positive edge at the REQ parameter starts the data exchange. You supply the ID parameter with the connection ID defined by STEP 7 in the connection table.

The block signals with "1" at the DONE or NDR parameter 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 block call.

GET Read data from a partner CPU

At the ADDR_n parameter you specify the memory area in the partner device from which you want to fetch data. You address the area using an ANY pointer as described in Chapter 4.9.4 “ANY pointer” on page 135. The read data is entered into the tag or the area which is specified at parameter RD_n (receive mailbox). If the read data area is larger than the receive mailbox, an error is displayed via ERROR and STATUS. Use the parameters ADDR_n and RD_n without gaps, beginning with 1. You do not supply parameters which are not required.

PUT Write data to a partner CPU

At the ADDR_n parameter you specify the memory area in the partner device to which you want to send data. You address the area using an ANY pointer as described in Chapter 4.9.4 “ANY pointer” on page 135. The data that is to be written is entered into the tag or the area which is specified at parameter SD_n (send mailbox). If the send mailbox is larger than the memory area addressed with ADDR_n, an error is displayed via ERROR and STATUS. Use the parameters without gaps, beginning with 1. You do not supply parameters which are not required.

17.3.3 Two-way data exchange

For two-way data exchange you require a send block and a receive block at each end of a connection. Both blocks have the connection IDs which are present in the same line in the connection table. You can also use several “pairs of blocks” for a connection which are then distinguished by the job ID (Fig. 17.9).

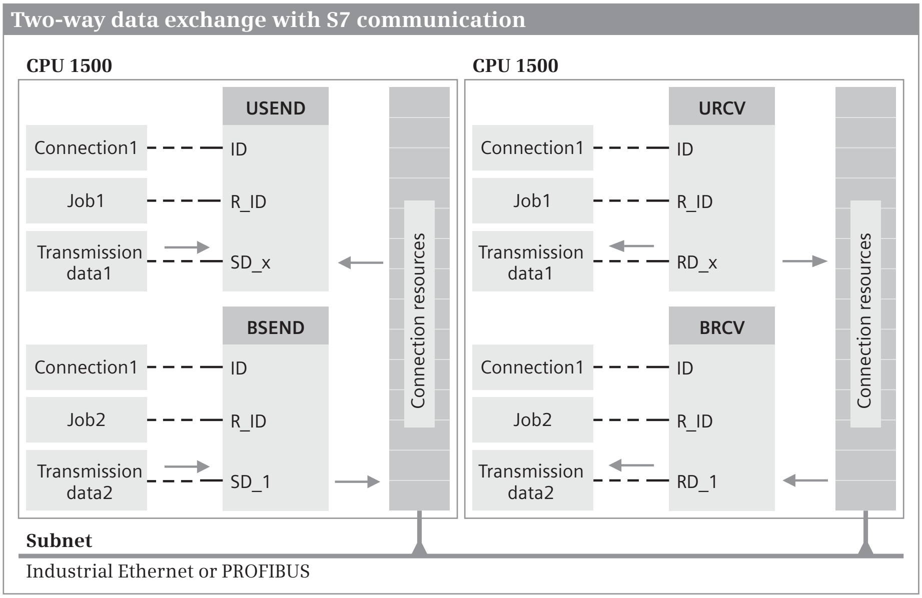


Fig. 17.9 Data structure for two-way data exchange

The following blocks are available for two-way data exchange:

- USEND Send data uncoordinated
- URCV Receive data uncoordinated
- BSEND Send a data block with a length of up to 64 KB
- BRCV Receive a data block with a length of up to 64 KB

The size of the consistently transferred data blocks for uncoordinated sending and receiving depends on the partner CPU used and the number of parameters used (SD_n or RD_n). The blocks can be found in the program elements catalog under *Communication > S7 communication*. The graphic representation of the block calls is shown in Fig. 17.10.

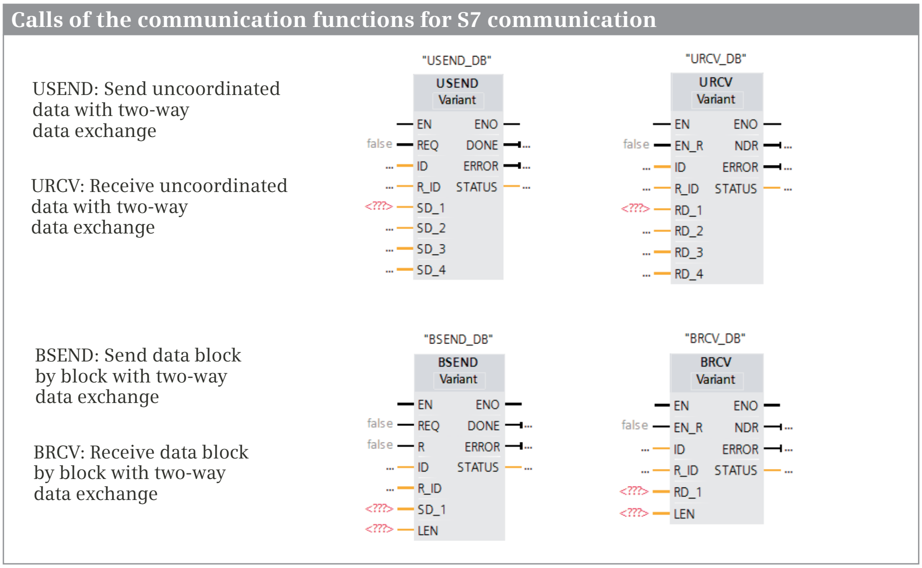


Fig. 17.10 Communication functions for two-way data exchange

Common parameters

A positive edge at the REQ parameter starts the data exchange, a positive edge at the R parameter aborts it. A “1” at the EN_R parameter signals the readiness to receive and a current job can be aborted by “0”.

You supply the ID parameter with the connection ID defined by STEP 7 in the connection table for both the local and partner devices. Use R_ID to define a freely-selectable yet unique job ID which must be the same for the send and receive blocks. In this manner, several pairs of send and receive blocks can use a single connection (specified by means of ID).

The block signals with “1” at the DONE or NDR parameter 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 block call.

USEND Send data uncoordinated

USEND sends data without coordination to a remote communication function URCV, i.e. without acknowledging the data transfer.

At the parameter SD_n (send mailbox), you enter the tag or the memory area from which the data to be transferred is to be taken. You address a memory area using an ANY pointer as described in Chapter 4.9.4 “ANY pointer” on page 135. Use the parameters SD_n without gaps, beginning with 1. You do not supply parameters which are not required.

At the start of the job, if USEND detects a positive signal edge at parameter REQ, the data is copied from the send mailboxes and the transfer begins. You can write new data to the send mailboxes immediately following the start of the job.

URCV Receive data uncoordinated

URCV receives data without coordination from a remote communication function USEND, i.e. the data receipt is not acknowledged.

At the parameter RD_n (receive mailbox), enter the tag or the memory area into which the received data is to be written. You address a memory area using an ANY pointer as described in Chapter 4.9.4 “ANY pointer” on page 135. The receive mailbox RD_n must correspond to the respective send mailbox SD_n. If the receive mailbox is smaller than the volume of the received data, an error is output via ERROR and STATUS.

After the data transfer is completed, the parameter NDR adopts signal state “1”. If you want to prevent new data from being written to the receive mailboxes during the data evaluation, call URCV again, this time with EN_R = “0”. After the evaluation of the received data, enable the data transfer again with EN_R = “1”.

BSEND Send data block by block

BSEND sends a data packet (segmented) to a remote communication function BRCV. The data packet can be up to 65 534 bytes in size.

At the parameter SD_1 (send mailbox), enter the tag or the start of the memory area from which the data to be transferred is to be taken. You address a memory area using an ANY pointer as described in Chapter 4.9.4 “ANY pointer” on page 135. Specify the number of the bytes to be sent at parameter LEN.

BSEND must be called until the transfer is ended with DONE = “1”. During this time, the data to be sent must not be changed.

BRCV Receive data block by block

BRCV receives a data packet (segmented) from a remote communication function BSEND. The data packet can be up to 65 534 bytes in size.

At the parameter RD_1 (receive mailbox), enter the tag or the start of the memory area into which the received data is to be written. You address a memory area using an ANY pointer as described in Chapter 4.9.4 “ANY pointer” on page 135. The size of the receive mailbox determines the maximum length of the received data block. The number of the currently received bytes is displayed at parameter LEN.

BRCV must be called until the end of the transfer of all segments is indicated with NDR = “1”.

17.3.4 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 a network” on page 73. 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*.

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. If you select the *Single instance* option, a separate data block is assigned to the call.

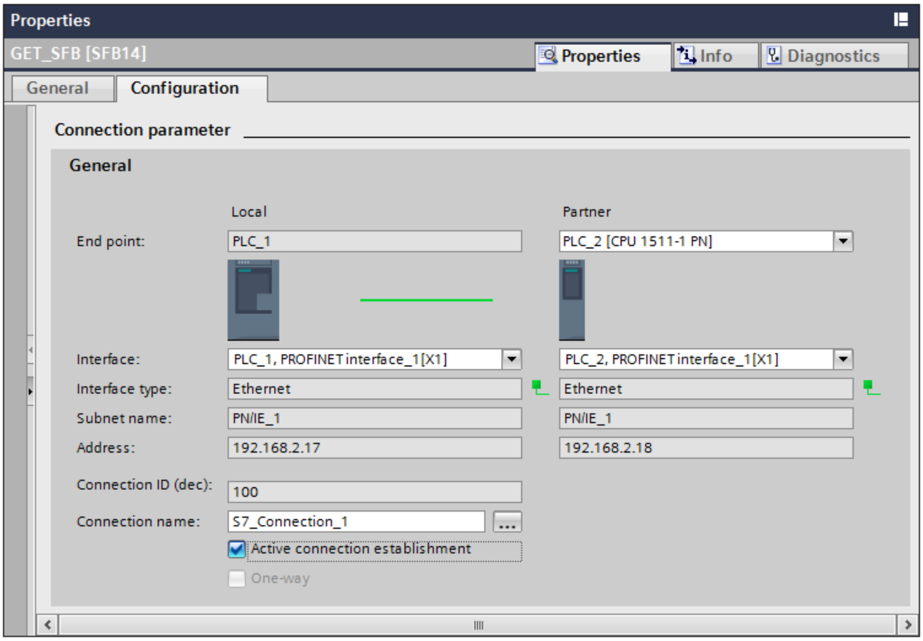
Configuring a connection

You configure the properties of an S7 connection in the inspector window. For one-way data exchange, the connection properties are displayed under *Properties* > *Configuration* when a communication function GET or PUT is inserted. You can also select the connection in the connection table in the lower part of the working window and configure the connection properties in the inspector window under *Properties* > *General* (Fig. 17.11).

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 connection is selected.

In the connection dialog, select the S7 connection for data transmission or create a new connection by clicking *Select connection* (the button with three dots). The connection ID identifies the connection and must be the same in both stations for a two-way data exchange. 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.



Connection parameter

General

	Local	Partner
End point:	PLC_1	PLC_2 [CPU 1511-1 PN]
Interface:	PLC_1, PROFINET interface_1[X1]	PLC_2, PROFINET interface_1[X1]
Interface type:	Ethernet	Ethernet
Subnet name:	PN/IE_1	PN/IE_1
Address:	192.168.2.17	192.168.2.18
Connection ID (dec):	100	
Connection name:	S7_Connection_1	

☑ Active connection establishment

□ One-way

Fig. 17.11 Configuring the connection parameters for S7 communication

17.4 Point-to-point communication

17.4.1 Introduction to point-to-point communication

With point-to-point communication (PtP communication), data can be exchanged via a serial interface with external devices such as printers or barcode readers. A CPU 1500 with a corresponding communication module supports the Freeport protocol for character-based, serial communication so that the data transmission protocol can be completely configured via the user program. Communication functions for the 3964(R) protocol, for the control of USS drives, and for Modbus RTU are also available.

The point-to-point communication is implemented with a CM PtP communication module, either via an RS232 interface or an RS422/RS485 interface. On the module are the indicators

- Diagnostics LED
  Flashes red after switching on until the module has been addressed (detected) by the CPU. After this, it flashes green until the module has been parameterized. In the operationally ready state, the LED illuminates with a steady green light.
- Send LED
  Illuminates if data is sent to the connected device.
- Receive LED
  Illuminates if data is received from the connected device.

For an RS232 interface, the communication can be coordinated by means of additional accompanying signals. An RS422/RS485 interface allows longer cable lengths through the use of differential voltages. During full duplex operation (RS422, simultaneous transfer in both directions), communication takes place via a four-wire cable. During half duplex operation (RS485, transfer in only one direction at a time), it takes place via a two-wire cable. The multipoint-capable coupling in full duplex mode allows master/slave operation with several stations.

The CM modules are available in the BA (Basic) and HF (High Feature) versions. Basically, the versions differ in terms of the maximum possible transmission speed, the maximum possible message frame length, and the capability of using the Modus protocol.

17.4.2 Configuring the CM PtP communication module

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* > *CM PtP ...* and drag it to the slot in the rack. Now set the configuration data in the inspector window.

Depending on the module, one of the following protocols can be used:

- Freeport (ASCII protocol) for transferring ASCII strings without a defined protocol format
- 3964(R), for example for data transmission between two PLC stations
- Modbus RTU, for example for data transmission between two PLC stations (operation as master or as slave)
- USS (universal serial interface) for controlling drives in master mode

The data flow control coordinates the transmission of the message frames. The software data flow control with XON/XOFF is possible with the Freeport protocol via the RS232 and RS422 interfaces. During the configuration, you define the start character (XON) and the end character (XOFF) of a message frame transmission. The hardware data flow control with the RTS/CTS signal is possible with the Freeport protocol via the RS232 interface. You can also configure automatic control of the accompanying signals with the Freeport and Modbus RTU protocols via the RS232 interface.

With PtP communication, the data transfer is character-based. A character can consist of 7 or 8 bits. In addition to the character bits, a parity bit can be transferred and used for error detection: When you have “even parity”, the signal state of the parity bit is selected such that the sum of the bits that have signal state “1” is even. For “odd parity”, the sum is odd. It is also possible to always set the parity bit to signal state “1” (mark) or “0” (space). 1 or 2 stop bits form the end of the transferred character.

You have the capability of controlling data traffic via the serial interface with a self-defined communication protocol. To do this, set the transmission parameters under *Configuration of message sending* and define how the start and end of a message frame can be detected under *Configuration of message receipt*.

17.4.3 Point-to-point communication functions

The following communication functions are available for PtP communication:

▷ Port_Config	Set the port configuration
▷ Send_Config	Set the send parameters
▷ Receive_Config	Set the receive parameters
▷ P3964_Config	Set the protocol parameters for 3964(R)
▷ Send_P2P	Initiate data transmission
▷ Receive_P2P	Enable data receipt
▷ Receive_Reset	Empty the receive buffer
▷ Signal_Get	Read RS232 signals
▷ Signal_Set	Write RS232 signals
▷ Get_Features	Read extended functions
▷ Set_Features	Activate extended functions

Fig. 17.12 shows the calls of the communication functions in LAD representation and Fig. 17.13 shows the data structure.

Programming communication functions

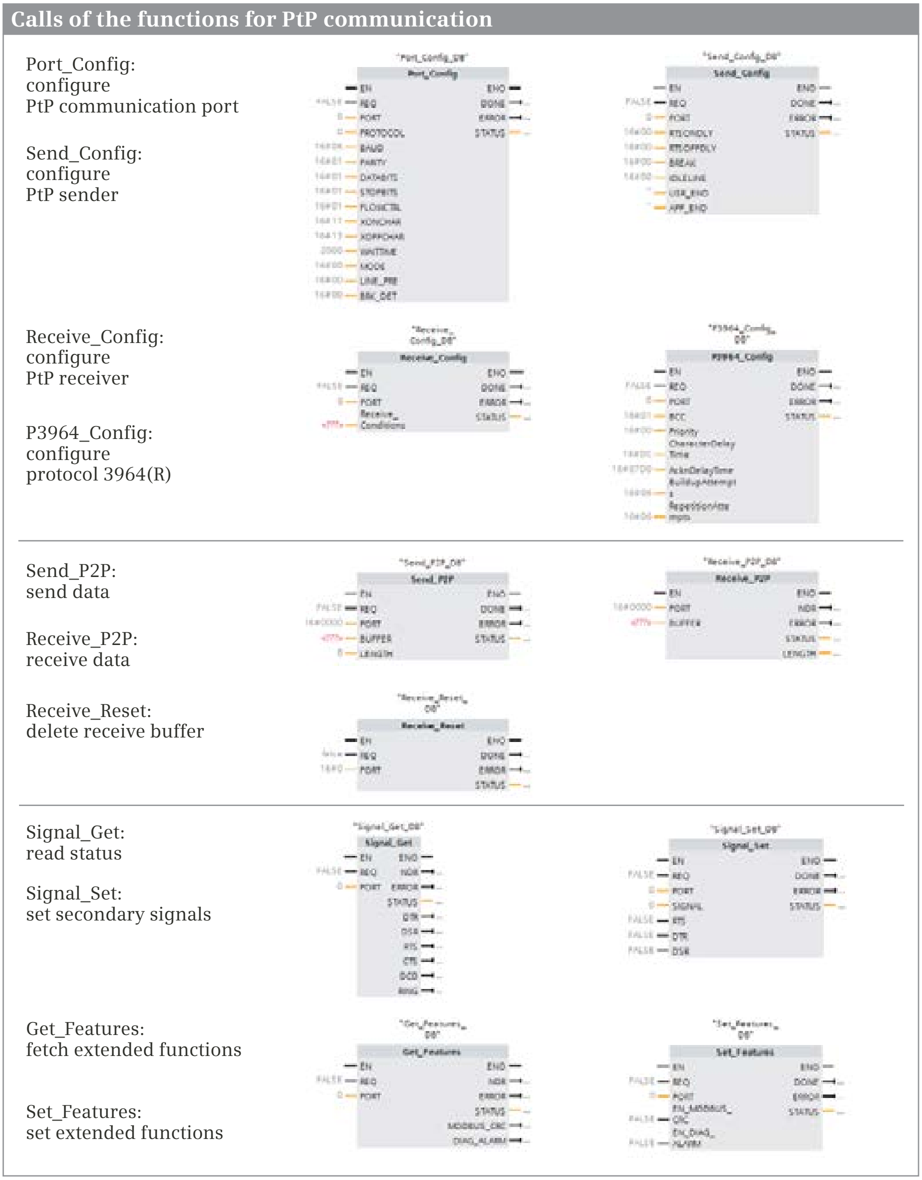
The communication functions for PtP communication are called in the main program, for example in a function block that is called in the organization block OB 1 or in another organization block with the event class *Program cycle*.

To program a communication function, open the block, select the function in the program elements catalog under *Communication* > *Communication processor* > *Point-to-point* and drag it into the open block by pressing and holding the mouse key. 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 at parameter REQ, the task is started. As long as the signal state is “1”, no other task 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 DONE parameter signals with signal state “1” that the started job has been completed without errors. The NDR parameter signals with signal state “1” that the started job has been completed without errors and that new data has been received. The ERROR parameter signals with signal state “1” that the started job has been



Calls of the functions for PtP communication

Port_Config:
configure
PtP communication port

Send_Config:
configure
PtP sender

Receive_Config:
configure
PtP receiver

P3964_Config:
configure
protocol 3964(R)

Send_P2P:
send data

Receive_P2P:
receive data

Receive_Reset:
delete receive buffer

Signal_Get:
read status

Signal_Set:
set secondary signals

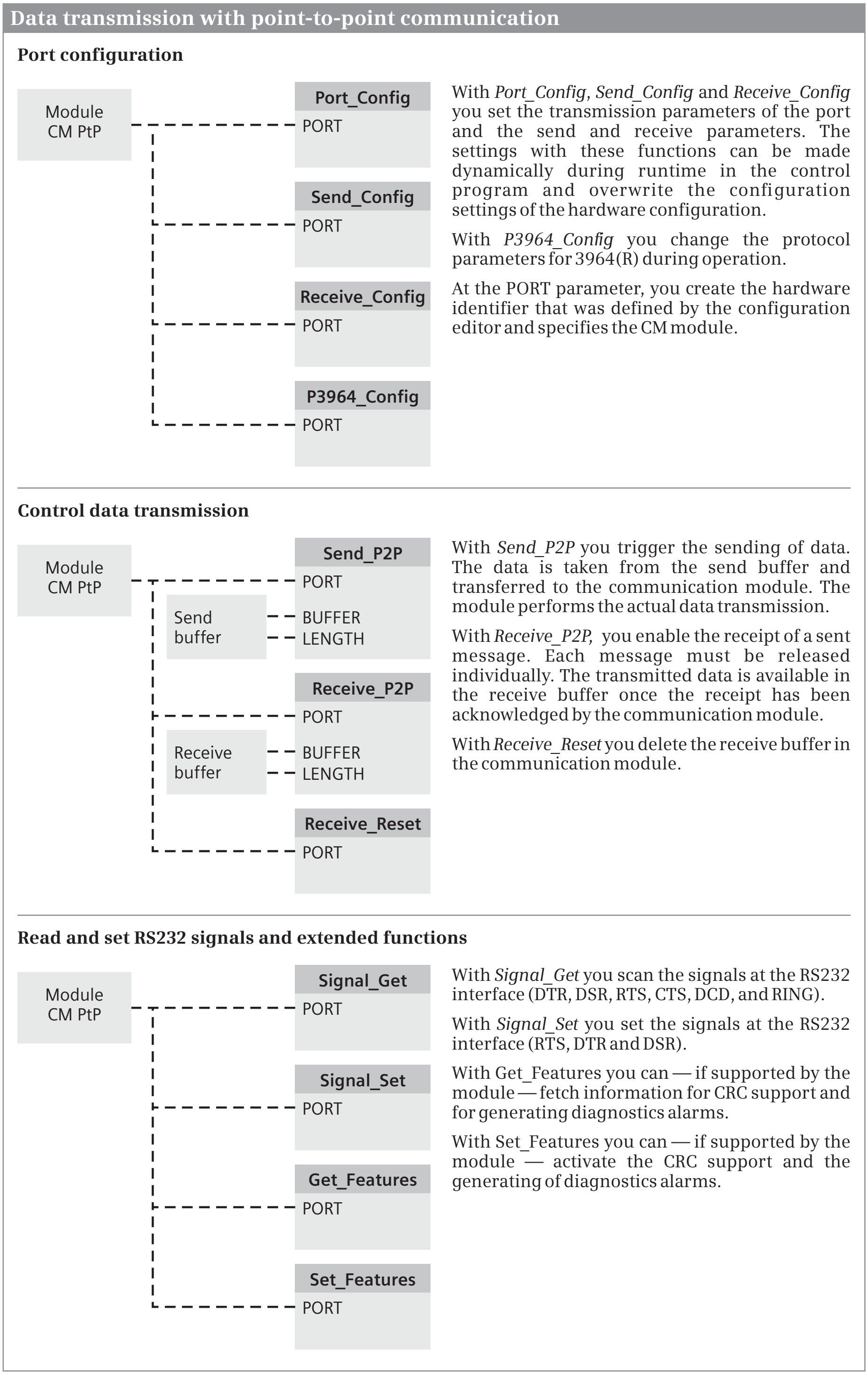
Get_Features:
fetch extended functions

Set_Features:
set extended functions

Fig. 17.12 Calls of the functions for PtP communication in LAD representation

completed with errors. The STATUS parameter contains intermediate states or error information. The parameters DONE, NDR and ERROR are only set for the duration of one program cycle each after the job has ended.

The parameter PORT specifies the interface on the CM module. It is assigned the hardware identifier, which can be found in the interface properties and in the sys-



With *Port_Config*, *Send_Config* and *Receive_Config* you set the transmission parameters of the port and the send and receive parameters. The settings with these functions can be made dynamically during runtime in the control program and overwrite the configuration settings of the hardware configuration.

With *P3964_Config* you change the protocol parameters for 3964(R) during operation.

At the PORT parameter, you create the hardware identifier that was defined by the configuration editor and specifies the CM module.

With *Send_P2P* you trigger the sending of data. The data is taken from the send buffer and transferred to the communication module. The module performs the actual data transmission.

With *Receive_P2P,* you enable the receipt of a sent message. Each message must be released individually. The transmitted data is available in the receive buffer once the receipt has been acknowledged by the communication module.

With *Receive_Reset* you delete the receive buffer in the communication module.

With *Signal_Get* you scan the signals at the RS232 interface (DTR, DSR, RTS, CTS, DCD, and RING).

With *Signal_Set* you set the signals at the RS232 interface (RTS, DTR and DSR).

With Get_Features you can — if supported by the module — fetch information for CRC support and for generating diagnostics alarms.

With Set_Features you can — if supported by the module — activate the CRC support and the generating of diagnostics alarms.

Fig. 17.13 Communication functions for PtP communication

tem constants tab in the default tag table. The parameters BUFFER and LENGTH contain the send or receive mailbox for the transferred data.

Changing configuration settings during runtime

The properties of a port are set when the CM module is configured. These properties are “static.” They are transferred from the load memory into the CM module when the PLC station is switched on and they are valid for continued operation. These properties can be modified during operation using communication functions. The “dynamically” changed properties are not permanent; they are replaced during the next startup of the “static” properties.

Port_Config changes the PORT properties such as baud rate, parity, number of data bits and stop bits.

Send_Config controls the time intervals between the activation of the RTS signal up to the start of data transfer and from the end of the transfer until the deactivation of the RTS signal and the breaks at the start and end of the message.

Receive_Config influences the conditions for the start and end of a message that is to be sent. Messages that meet these conditions can be received using the communication function *Receive_P2P*. The receive conditions are compiled in the CONDITIONS data structure.

P3964_Config changes parameters of the 3964(R) protocol such as character delay time, priority, and block check.

Sending and receiving data

Send_P2P transfers the send data from the send mailbox in the user memory to the CM module and initiates the sending of data. The CM module handles the actual transfer to the external device.

Receive_P2P enables the receipt of a sent message, where each message must be individually enabled. The data is transferred from the CM module into the receive mailbox in the user memory.

Receive_Reset empties the receive buffer of the CM module.

RS232 signals and expanded functions

If the CM module provides the corresponding support, the following functions can be used:

Signal_Get reads the signal states of the signals DTR, DSR, RTS, CTS, DCD and RING from the port of an RS232 interface.

Signal_Set writes the signal states of the signals RTS, DTR and DSR to the port of an RS232 interface.

Get_Features determines whether the CM module supports the generation of a checksum for Modbus (CRC) or the generation of diagnostics alarms.

Set_Features activates the generation of a checksum for Modbus (CRC) or the generation of diagnostics alarms on the CM module.

17.5 Further communication functions

17.5.1 USS protocol for drives

A CM PtP communication module can control per RS485 port up to 16 Siemens drives, which support the universal serial interface (USS), with the following functions:

▷ USS_Port_Scan	Communication via the USS network
▷ USS_Drive_Control	Prepare data for a drive
▷ USS_Read_Param	Read operating parameters from the drive
▷ USS_Write_Param	Write operating parameters to the drive

You can find the communication functions in the program elements catalog under *Communication* > *Communication processor* > *USS Communication*. Fig. 17.14 shows the calls of the functions for the USS protocol in the ladder logic representation, Fig. 17.15 shows the associated data structure.

USS_Drive_Control 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. A single data block is available for all controlled drives per communication module.

USS_Read_Param 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.

USS_Write_Param 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.

USS_Port_Scan transfers the drive data between the data block and the CM module. It is called only once per CM module.

The blocks *USS_Drive_Control*, *USS_Read_Param* and *USS_Write_Param* must be called in the main program; any organization block is possible for the block *USS_Port_Scan*. The processing of *USS_Port_Scan* 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.

Calls of the USS functions

USS_Port_Scan:
communication
via a USS network

USS_Drive_Control:
prepare and display
data for the drive

USS_Read_Param:
read data from the drive

USS_Write_Param:
change data in the drive

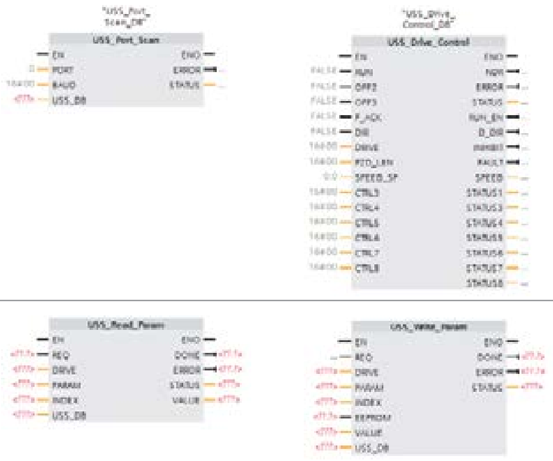
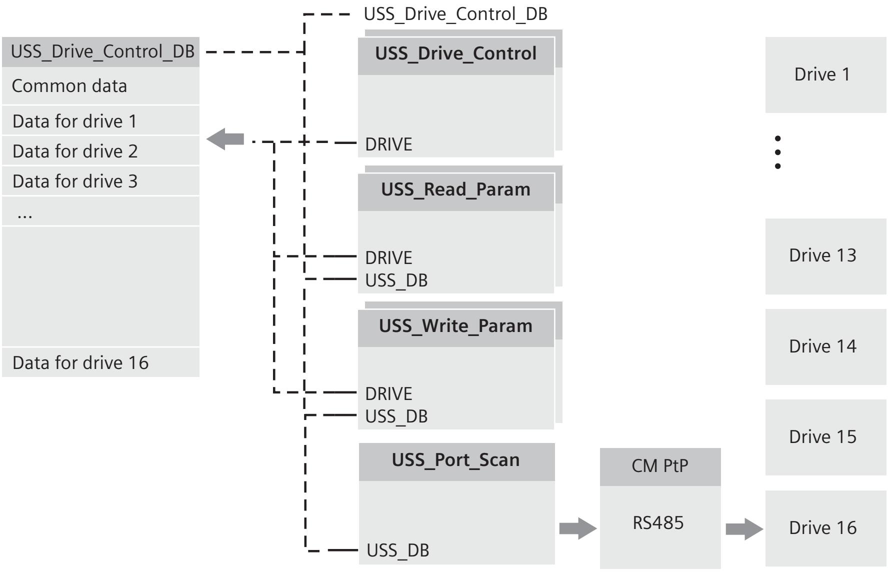


Fig. 17.14 Calls of the USS functions in LAD representation

Data structure for the USS protocol



The USS protocol is used for the control of up to 16 drives per port with a module CM PtP RS422/485.

A data block contains the data of a module for all drives of an RS 485 port.

Fig. 17.15 Data structure for the USS protocol

17.5.2 Modbus RTU

The standard protocol Modbus RTU (Remote Terminal Unit) uses the RS232 or RS422/RS485 interface for the serial data transfer between a Modbus master and one or more Modbus slaves. The communication via Modbus RTU is controlled using the following functions:

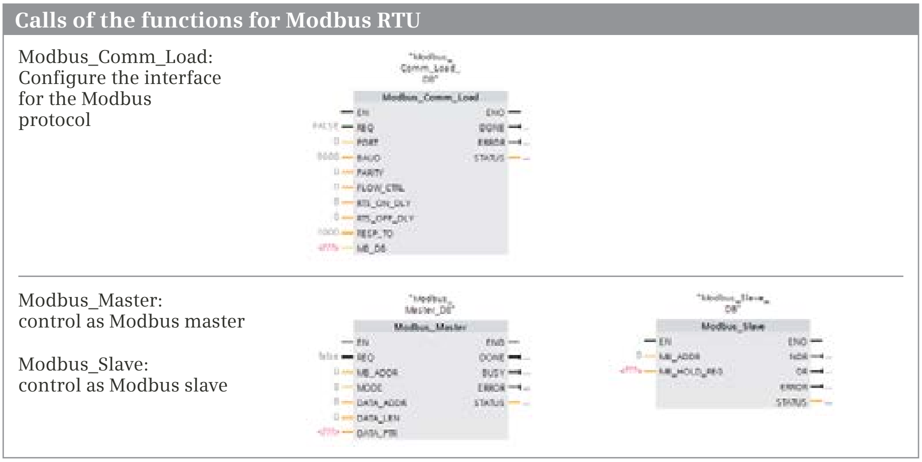
▷ Modbus_Comm_Load	Configure a CM module for the Modbus protocol
▷ Modbus_Master	Control for the Modbus master
▷ Modbus_Slave	Control for the Modbus slave

You can find the communication functions in the program elements catalog under *Communication* > *Communication processor* > *MODBUS (RTU)*. Fig. 17.16 shows the calls of the functions for the Modbus RTU protocol in the ladder logic representation, Fig. 17.17 shows the associated data structure.

Configuring a port with Modbus_Comm_Load

Modbus_Comm_Load configures the port (the connection) at the CM module for the Modbus RTU protocol. Executing Modbus_Comm_Load is a prerequisite for using Modbus_Master and Modbus_Slave.

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 error information is output at the STATUS parameter. The assigning of these status parameters is only valid for one cycle until the next processing of Modbus_Comm_Load. Modbus_Comm_Load is initialized with signal state “1” at the COM_RST parameter. After this, COM_RST is reset to “0”.


Calls of the functions for Modbus RTU

Modbus_Comm_Load:
Configure the interface for the Modbus protocol

Modbus_Master:
control as Modbus master

Modbus_Slave:
control as Modbus slave

Fig. 17.16 Calling the functions for Modbus RTU in LAD representation

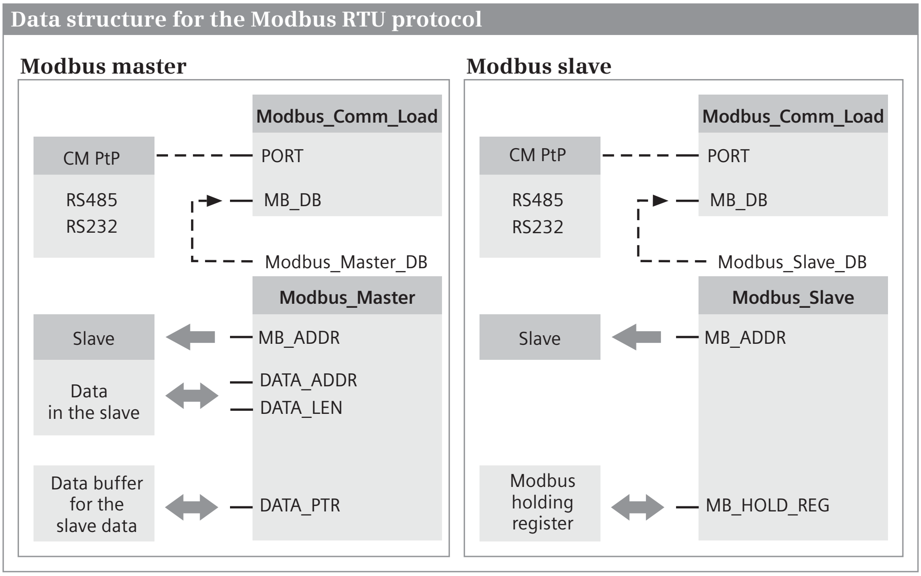


Fig. 17.17 Data structure for the Modbus RTU protocol

The parameter MB_DB is the reference to the data of the Modbus master or Modbus slave. This parameter is supplied with the data tag *MB_DB* from the static local data of the master or slave instance data block. Example for the actual parameter: "Modbus_Master_DB".MB_DB.

Controlling data traffic with Modbus_Master

Modbus_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 error information is output at the STATUS parameter. The assigning of these status parameters is only valid for one cycle until the next processing of Modbus_Master. Modbus_Master is initialized with signal state "1" at the COM_RST parameter. After this, COM_RST is reset to "0".

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.

Modbus_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 to master requests with Modbus_Slave

Modbus_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 status parameters is only valid for one cycle until the next processing of Modbus_Slave. Modbus_Slave is initialized with signal state “1” at the COM_RST parameter. After this, COM_RST is reset to “0”.

The parameter MB_HOLD_REG points to the Modbus holding register, which is used by the Modbus_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.

17.5.3 Modbus TCP

The standard protocol Modbus TCP (Transmission Control Protocol) uses a PROFINET interface for the data transfer between a Modbus client and a Modbus server. Communication via Modbus TCP uses the following functions:

▷ Modbus_Client	Control for the Modbus client
▷ Modbus_Server	Control for the Modbus server

You can find the communication functions in the program elements catalog under *Communication > Other > Modbus TCP*. Fig. 17.18 shows the calls of the functions for the Modbus TCP protocol in the ladder logic representation, Fig. 17.19 shows the associated data structure.

Calls of the functions for Modbus TCP

MB_CLIENT:
control as Modbus client

MB_SERVER:
control as Modbus server

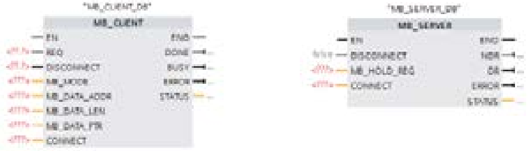


Fig. 17.18 Calling the functions for Modbus TCP in LAD representation

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 error information is output at the STATUS parameter.The

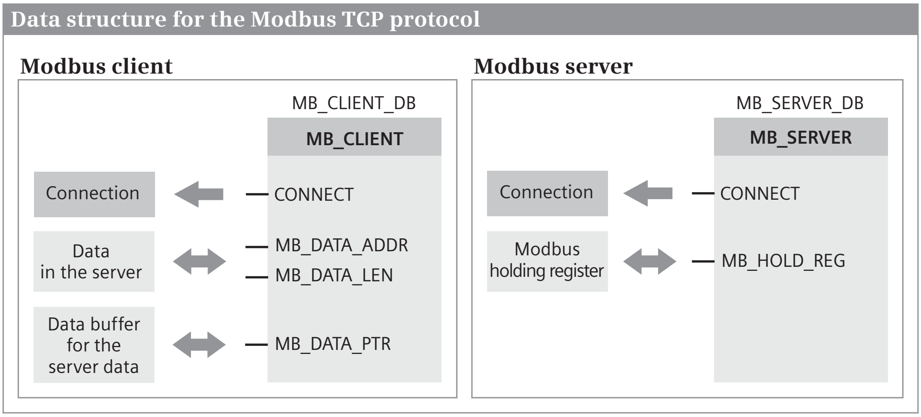


Fig. 17.19 Data structure for the Modbus TCP protocol

assigning of these status 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. Each connection requires its own function call (own instance data). 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 specifies the connection. The actual parameter can be a tag or a type data block with the structure of the system data type TCON_IP_v4 (then the connection is established by MB_CLIENT) or TCON_Configured (if the connection was configured with the network editor, see Chapter 17.2 "Open user communication" on page 751).

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 specifies the connection. The actual parameter can be a tag or a type data block with the structure of the system data type TCON_IP_v4 (then the connection of MB_CLIENT is established) or TCON_Configured (if the connection with the network editor has been configured, see Chapter

17.2 “Open user communication” on page 751). Each connection needs its own function call (own instance data) if, for example, the server station communicates with several Modbus clients.

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 error information is output at the STATUS parameter. The assigning of these status 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.

18 Appendix

18.1 Working with source files

Blocks with the programming languages STL or 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, for example the Windows text editor. 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.

18.1.1 General procedure

A source file can be generated in the following ways:

- You write the source file completely using a text editor.
- You copy an STL or SCL block as text to the Windows clipboard and create a source file from it.
- You take an SCL 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 with a text editor

In order to program a block, you must use keywords in a specific sequence in the source file, as described in the following chapters.

A source file can contain several blocks and these can be logic or data blocks as well as PLC data types. You can also call blocks in the source file which are present in the *Program blocks* folder or use system blocks from the program elements catalog. 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 252).

When working with source files, you must handle blocks programmed using STL separate from those programmed using SCL. A source file can contain either only STL blocks or only SCL blocks. In both cases, the source file can contain data blocks and PLC data types.

You save a source file with STL program with the file extension .stl and a source file with SCL program with the file extension .scl. If you only program data blocks or PLC data types, the file extension is irrelevant.

Generating source data by copying text

In the project tree, select the block(s) from which you wish to generate a source file in the *Program blocks* folder and then select the *Copy as text* command from the shortcut menu. The program is copied to the Windows clipboard. Paste the contents of the clipboard into a text editor, change the program if necessary, and save the program as a source file. If the program contains STL blocks, data blocks, and PLC data types, select the file extension .stl. If the program contains SCL blocks, data blocks, and PLC data types, select the file extension .scl.

For password-protected blocks, only the block header and the interface description are copied.

Generating a source file by exporting

In the project tree, select the block(s) with SCL program from which you wish to generate a source file in the *Program blocks* folder and select the *Generate source from blocks* command from the shortcut menu. Then define the name and storage location of the file with the file extension .scl in the dialog.

You generate the source file for one or more data blocks or PLC data types in the same way.

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 window, 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 in the project tree.

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 .stl or .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. In the dialog, select the editor which you wish to link to the file extension .stl or .scl.

You can then edit the source file using the linked editor by double-clicking on it in the *External sources* 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 from source* 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.

Special features

If a block in the source file has been programmed with a block number (absolute addressing), the block in the project is given the absolute address, without the percent sign, for a name.

If a block in the source file has been programmed with a block number (absolute addressing) and a block with the same number exists in the project, the block is overwritten in the project.

If a block in the source file has been programmed with a name (symbolic addressing) and a block with the same name exists in the project, the block is overwritten in the project.

The event class of an organization block is derived from its name. Example: If the organization block has the name “Cyclic interrupt”, it is generated as a cyclic interrupt organization block and is given the next permissible, free number (30 to 38 and from 123 on, see Table 5.7 on Page 194). If you want to program an additional block of an event class, supplement the name with a number. Example: An organization block with the name “Cyclic interrupt 1234” is generated as a cyclic interrupt organization block with the name *Cyclic interrupt 1234* and the next permissible, free number. The name and number can be changed in the project later. If no event class can be derived from the name, the organization block with the absolute address %OB0 is inserted into the project and cannot be edited.

If the access type in the source file is not explicitly defined, the attribute *Optimized block access* is deactivated (standard access) for an organization bock and for every other block, the attribute *Optimized block access* is activated.

If an operand is absolutely addressed in the external source file and no name is assigned in the PLC tag table, it is given a symbolic address with the prefix “Tag_” and a consecutive number.

There is no distinction between upper and lower case in the source file, exception: The designation for jump labels is case sensitive.

18.1.2 Programming a code block in the source file

The program of each code block consists of the block header with specification of the block type and block properties. This is followed by the declaration of the interface and the actual program. Terminate the programming of the block using a keyword for the block end.

Table 18.1 shows which keywords you require for block programming and the sequence in which the keywords are used.

Table 18.1 Keywords for code blocks

Section	Keyword	Meaning
Block type	ORGANIZATION_BLOCK “*OB_name*” FUNCTION_BLOCK “*FB_name*” FUNCTION “*FC_name*” : *Data type*	Start of an organization block Start of a function block Start of a function
Header	TITLE = *block title* //Block comment	Block property: Block title Block property: Block comment
	AUTHOR : *Created by* FAMILY : *Block family* NAME : *Block name* VERSION : *Version*	Block property: Created by Block property: Block family Block property: User-defined ID Block property: Block version
	{ S7_Optimized_Access := xxx }	Access type: xxx = ’TRUE’ : Optimized access xxx = ’FALSE’ : Standard access
	CODE_VERSION1	Only with FB (”not capable of multi-instance”), only with STL
	KNOW_HOW_PROTECT	No effect
Declaration Introduction	VAR_INPUT *Retentivity* VAR_OUTPUT *Retentivity* VAR_IN_OUT *Retentivity* VAR *Retentivity* VAR_TEMP	Input parameter (not with OB) Output parameter (not with OB) In/out parameter (not with OB) Static local data (only with FB) Temporary local data Retentivity: RETAIN = retentive DB_SPECIFIC = set in IDB (No specification) = not retentive
Declaration Tags	name {...; ...} : Data type := Default setting;	

	{ S7_HMI_Accessible := xxx }
	{ S7_HMI_Visible := xxx }
	{ S7_SetPoint := xxx }
		Accessible from HMI
		Visible in HMI
		Setpoint
		Value: xxx = ’TRUE’ : Property activated
		xxx = ’FALSE’ : Property deactivated

	name_ueb {...; ...} AT name : Data type;	Data type superimposition (only with standard access)
Declaration End	END_VAR	At the end of each declaration section
Program	BEGIN	Start of block program, can be omitted with SCL
	NETWORK	Network start, only with STL
	TITLE = *Network title*	Network title, only with STL
	//Network comment	Network comment; line comment with SCL
	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, only with SCL
	NETWORK	Start of next network, only with STL
	...	... etc.

Block end	END_ORGANIZATION_ BLOCK
	END_FUNCTION_BLOCK
	END_FUNCTION
		End of an organization block
		End of a function block
		End of a function

Block header and block properties

A code block commences with the keyword for the block type and with the specification of the block name. With symbolic addressing (e.g. FUNCTION_ BLOCK “FB_name”), the first vacant number of the block type is assigned when importing for absolute addressing. When specifying an absolute address (e.g. FUNCTION_BLOCK %FB102), the operand with the number is imported as the symbolic address.

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 data for the block properties is optional. You simply omit the surplus data together with the keywords. You can find the permitted assignment in Chapter 5.3.2 “Block properties” on page 157.

{ S7_Optimized_Access = ... } sets the attribute *Optimized block access* with the value ’TRUE’. The ’FALSE’ value resets the *Optimized block access* attribute.

For a function block, the keyword CODE_VERSION1 deactivates the capability of embedding in a multi-instance (only for STL). The block attribute *Multiple instance capability* is deactivated and cannot be reactivated later.

The keyword KNOW_HOW_PROTECT, which is programmable for a CPU 300/400, has no effect for blocks for a CPU 1500.

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 18.1). If you do not use a declaration section, omit it including the keywords.

You begin a declaration section with the introductory keyword and the retentivity settings, then you list the tags and end the declaration section with the keyword END_VAR. The retentivity setting applies until the end of the declaration section. With each change of the retentivity setting, you end the previous declaration section and begin a new one. Example: The input parameters *Par1* and *Par2* have the retentivity setting *Set in IDB*, *Par3* is not retentive, and *Par4* is retentive.

```
VAR_INPUT DB_SPECIFIC
Par1 {S7_HMI_Visible := 'FALSE' } : INT := 500; //Comment
Par2 {S7_HMI_Accessible := 'TRUE'} : BOOL; //Comment
END_VAR

VAR_INPUT
Par3 {S7_HMI_Accessible := 'FALSE'; S7_HMI_Visible := 'FALSE'} : INT ;
END_VAR

VAR_INPUT RETAIN
Par4 {S7_SetPoint := 'TRUE'} : REAL := 1.0; //Comment
END_VAR
```

The declaration of a tag consists of the name, the tag property, the data type, the default setting, and the tag comment. The tag property, default setting, and tag comment are optional. By default, the tag properties *S7_HMI_Accessible* and *S7_HMI_Visible* are predefined with the value “TRUE” and *S7_SetPoint* is predefined with the value “FALSE”.

Not all tags can have default start values, e.g. default settings are not possible for the temporary local data. Chapter 5.3.3 “Block interface” on page 157 describes the data types permissible for block parameters.

The sequence of individual declaration sections is defined as shown in the table. 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 a block with the property *{S7_Optimized_Access = ’FALSE}* (standard access), the tags can be superimposed with a different data type in a declaration section (Chapter 4.5.3 “Overlaying tags (data type views)” on page 111). You program the superimposition directly after the declaration of the tag to be superimposed with the keyword AT. The scheme is as follows: *var_new* AT *var_old*: new_data type. Example: The input parameter *DateTime* is superimposed with a structure *Date*, comprised of the components *Year*, *Month* and *Day*.

```
VAR_INPUT
   DateTime : DT;
   Date AT DateTime : STRUCT
      Year : WORD;
      Month : WORD;
      Day : WORD;
      END_STRUCT;
END_VAR
```

Program section

The program section of a code 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, except for jump labels. Refer to Chapter 10.1.2 “Structure of an STL statement” on page 397 for the syntax of an STL statement and to Chapter 9.1.2 “SCL statements and operators” on page 361 for that of an SCL statement. You can enter one or more spaces or tabulators between operation and operand. To achieve a clearer layout of the source text, you can enter any spaces and/or tabulators between the words.

You must conclude every statement 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. You can also program several statements per line, each separated by a semicolon.

```
FUNCTION_BLOCK “FIFO_STL”
TITLE = Intermediate memory for 4 values
//Example of a function block in STL
AUTHOR : Berger
FAMILY : Book1500
NAME : Memory
VERSION : 01.00

VAR_INPUT
  Import : BOOL := FALSE;          //Transfer with positive edge
  Input_value : REAL := 0.0;       //In data format REAL
END_VAR
VAR_OUTPUT
  Output_value : REAL := 0.0;      //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_memory_bit : BOOL := FALSE; //Edge memory bit for the transfer
END_VAR

BEGIN
NETWORK
TITLE = Program for transfer and output
//Transfer and output take place with a positive edge at Transfer
      A    Transfer;               //If Transfer changes to “1"
      FP   Edge_memory_bit;        //the RLO = “1” following FP
      JCN end;                     //Jump if no positive edge is present
//Transfer of values starting with the last value
      L    Value4;
      T    Output_value;           //Output of last value
      L    Value3;
      T    Value4;
      L    Value2;
      T    Value3;
      L    Value1;
      T    Value2;
      L    Input_value;            //Transfer of input value
      T    Value1;
End: BE;
END_FUNCTION_BLOCK

DATA_BLOCK “DB_FIFO_STL"
TITLE = Instance data block for “FIFO_STL”
//Example of an instance data block
AUTHOR : Berger
FAMILY : Book1500
NAME : FIFO_Dat
VERSION : 01.00
FIFO_STL                           //Instance for the FB “FIFO_STL”
BEGIN
  Value1 := 1.0;                   //Individual default setting
  Value2 := 1.0;                   //selected value
END_DATA_BLOCK
```

Fig. 18.1 Example of an STL source file

```
FUNCTION_BLOCK “FIFO_SCL"
TITLE = Intermediate memory for 4 values
//Example of a function block with static local data in SCL

AUTHOR : Berger
FAMILY : Book1500
NAME : Memory
VERSION : 01.00

VAR_INPUT
  Import  : BOOL := FALSE;         //Transfer with positive edge
  Input_value : REAL := 0.0;       //In data format REAL
END_VAR

VAR_OUTPUT
  Output_value : REAL := 0.0;      //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_memory_bit : BOOL := FALSE; //Edge memory bit for the transfer
END_VAR

BEGIN
//Transfer and output take place with a positive edge at Transfer
IF Transfer = TRUE AND Edge_memory_bit = FALSE
THEN Output_value := Value4;
     //Transfer of values starting with the last value
     Value4 := Value3;
     Value3 := Value2;
     Value2 := Value1;
     Value1 := Input_value;
END_IF;

Edge_memory_bit := Transfer;       //Update edge memory bit

END_FUNCTION_BLOCK

DATA_BLOCK “DB_FIFO_SCL"
TITLE = Instance data block for “FIFO_SCL”
//Example of an instance data block

AUTHOR : Berger
FAMILY : Book1500
NAME : FIFO_Dat
VERSION : 01.00

FIFO_SCL                           //Instance for the FB “FIFO_SCL”

BEGIN
  Value1 := 1.0;                   //Individual default setting
  Value2 := 1.0;                   //selected value
END_DATA_BLOCK
```

Fig. 18.2 Example of an SCL source file

A line comment commences with two slashes at the start of the line. A line comment can have up to 160 characters, but no tabulators or non-printable characters.

A block comment with SCL 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.

You can also program networks to structure the block program better in STL. Networks commence with the keyword NETWORK. You can assign a title to every network using the keyword TITLE, which is present in the next line. The line comments which directly follow the network title are the network comment.

The figures 18.1 and 18.2 show an example of an STL source file and of an SCL source file for a function block with the associated instance data block.

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.

```
//Example of a block call in STL
  CALL "FIFO_AWL","DB_FIFO_STL" (
    Import    := "Clock",
    Input_value := "Measurement",
    Output value := "Measurement_delayed");

//Example of a block call in SCL
  "DB_FIFO_SCL" (
    Import    := "Clock",
    Input_value := "Measurement",
    Output value := "Measurement_delayed");
```

18.1.3 Programming a data block in the source file

The program of a data block begins with the keyword DATA_BLOCK and consists of the block header with specification of the block type and block properties. Then specify the data operands and terminate the program for the block with the keyword END_DATA_BLOCK. Table 18.2 shows which keywords you require for block programming and the sequence in which the keywords are used.

Block header and block properties

A data block commences with the keyword DATA_BLOCK and with specification of the block name. With symbolic addressing (e.g. DATA_BLOCK “DB_name”), the first vacant data block number is assigned when importing for absolute addressing. When specifying an absolute address (e.g. DATA_BLOCK %DB102), the operand with the number is imported as the symbolic address. The data for the block properties is optional. You simply omit the surplus data together with the keywords.

{ S7_Optimized_Access = ... } sets the attribute *Optimized block access* with the value ’TRUE’. The ’FALSE’ value resets the *Optimized block access* attribute.

Table 18.2 Keywords for data blocks

Section	Keyword	Meaning
Block type	DATA_BLOCK "*DB_name*"	Start of a data block
Header	TITLE = *block title*	Block property: Block title
	//Block comment	Block property: Block comment
	AUTHOR : *Created by*	Block property: Created by
	FAMILY : *Block family*	Block property: Block family
	NAME : *Block name*	Block property: User-defined ID
	VERSION : *Version*	Block property: Block version
	{ S7_Optimized_Access := xxx }	Access type: xxx = 'TRUE' : Optimized access xxx = 'FALSE' : Standard access
	UNLINKED	Block attribute: not executable
	READ_ONLY	Block attribute: read-only
	NON_RETAIN	Block attribute: non-retentive
	KNOW_HOW_PROTECT	No effect
Declaration	VAR *Retentivity*	Declaration for a global data block
	name {...; ...} : Data type := Default setting;	
	{ S7_HMI_Accessible := xxx }	Accessible from HMI
	{ S7_HMI_Visible := xxx }	Visible in HMI
	{ S7_SetPoint = xxx }	Setpoint
		Value: xxx = 'TRUE' : Property activated xxx = 'FALSE' : Property deactivated
	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 start values
Block end	END_DATA_BLOCK	End of a data block

The keyword UNLINKED activates the *Only store in load memory* block attribute. The keyword READ_ONLY activates the *Data block write-protected in the device* block attribute. Using the keyword NON_RETAIN you deactivate the retentivity for all of the data operands in the data block. Using VAR RETAIN in the declaration, you can activate the retentivity of the following data tags (see next section).

The keyword KNOW_HOW_PROTECT, which is programmable for a CPU 300/400, has no effect for blocks for a CPU 1500.

Block interface for a global data block

The block interface contains the declaration of the data operands.

Using the keyword RETAIN after the introductory keyword of the declaration, you activate the retentivity for the following declared data tags. The retentivity setting applies until the end of the declaration section. If the keyword RETAIN is missing, the following data tags are not retentive. Example: The tags *Var1* and *Var2* are retentive, the tags *Var3* and *Var4* are not retentive.

```
VAR RETAIN
Var1 {S7_HMI_Visible := 'FALSE' } : INT := 500; //Comment
Var2 {S7_HMI_Accessible := 'TRUE'} : BOOL; //Comment
END_VAR

VAR
Var3 {S7_HMI_Accessible := 'FALSE'; S7_HMI_Visible := 'FALSE'} := INT;
Var4 := BOOL;
END_VAR
```

The declaration of a tag in a global data block consists of the name, the tag property, the data type, the default setting, and the tag comment. The tag property, default setting, and tag comment are optional. By default, the tag properties *S7_HMI_Accessible* and *S7_HMI_Visible* are pre-defined with the value 'TRUE' and the tag property *S7_SetPoint* is pre-defined with 'FALSE'.

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.

Block interface for a type data block

The declaration in a type data block consists only of the specification of the assigned PLC data type. Example: A data block based on the PLC data type *User data type_1* is generated.

```
DATA_BLOCK "DB_name"
   User data type_1
BEGIN
   Comp1 := 123;
END_DATA_BLOCK
```

After the keyword BEGIN, you can specify a start value for individual tags of the data block. In the top example, the PLC data type has a component with the name *Comp1* and the data type INT, which is preset with the value 123. 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 data blocks. BEGIN and the default setting are optional. Tags that do not have a default start value retain the default value from the PLC data type as the start value.

Block interface for an instance data block

The declaration in an instance data block consists only of the specification of the assigned function block. Example: A data block based on the function block *FB_name* is generated.

```
DATA_BLOCK "DB_name"
   FB_name
BEGIN
   Switch on :=TRUE;
END_DATA_BLOCK
```

After the keyword BEGIN, you can specify a start value for individual tags of the data block. In the top example, the function block has a block parameter or a static local tag with the name *Switch on* and the data type BOOL, which is preset with the value TRUE. By assigning default values to the start values, it is possible to assign individual values to each application (each instance) in the case of instance data blocks. BEGIN and the default setting are optional. Tags that do not have a default start value retain the default value from the function block as the start value.

18.1.4 Programming a PLC data type in the source file

The program of a PLC data type begins with the keyword TYPE and consists of the header with specification of the title and comment. Then specify the structure of the data type and end the programming of the data type with the keyword END_-TYPE.

Table 18.3 shows which keywords you require for data type programming and the sequence in which the keywords are used.

Table 18.3 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;
	{ S7_HMI_Accessible := xxx }
	{ S7_HMI_Visible := xxx }
	{ S7_SetPoint = xxx }
	END_STRUCT
		Declaration of data type components
		
		Accessible from HMI
		Visible in HMI
		Setpoint
		Value: xxx = 'TRUE' : Property activated
		xxx = 'FALSE' : Property deactivated

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 data type name. With symbolic addressing (e.g. TYPE "Type_name"), the first vacant data type number is assigned when importing for absolute addressing. When specifying an absolute address (e.g. TYPE %UDT102), the operand with the number is imported as the symbolic address.

The data for the header is optional. You simply omit the surplus data together with the keywords.

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 tag in a PLC data type consists of the name, the tag property, the data type, the default setting, and the tag comment. The tag property, default setting, and tag comment are optional. By default, the tag properties *S7_HMI_Accessible* and *S7_HMI_Visible* are pre-defined with the value 'TRUE' and the tag property *S7_SetPoint* is pre-defined with 'FALSE'.

Example of a source file for a PLC data type:

```
TYPE "User data type_1"
TITLE = Header
//Comment on data type
VERSION : 0.9
   STRUCT
   Comp1 : INT := 100; //First data type component
   Comp2 {S7_SetPoint := ’TRUE’} : BOOL; //Second data type component
   END_STRUCT
END_TYPE
```

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.

18.2 Migrating and upgrading projects

Automation projects which have been created using STEP 7 Version 5.4 SP5 or later can be migrated into the TIA Portal. The target project resulting from the original project can then be edited further in the TIA Portal using STEP 7.

Automation projects which have been created using STEP 7 Version 11 can be upgraded to Version 12. Projects created with STEP 7 V12 are upgraded with STEP 7 V12 SP1 upon request when opened and they can then be edited.

18.2.1 Migrating a project

Preparations and sequence of project migration

A prerequisite for migration is the installation of all applications with which the original project was created. This also includes the option packages and the Hardware Support Packages (HSP). If these applications are installed together with the TIA Portal on the programming device, you can migrate the original project directly. Otherwise you install the migration tool on the programming device which contains the original project with the required applications, create an intermediate project with the file extension .am12 from the original project, transfer this intermediate project to a programming device with the TIA Portal, and then migrate the project. You download the migration tool from the Service & Support section of the Siemens website or you can install the migration tool from the setup DVD of the TIA Portal.

A report with the result of the migration is displayed in the inspector window at the end of migration. Here you can find references to project components which were not migrated or were modified by the migration. Not all components of the original project can be migrated unchanged. For example, a hardware configuration whose components do not support the TIA Portal cannot be imported into the target project. You can also exclude the hardware configuration from the migration. In this case only the software is migrated into the target project and a non-specified device is generated in the target project for each device present in the original project.

Associated with a successful migration is that you systematically process the information in the migration report.

Prerequisites in the original project

The original project must not be a multi-project and must not be provided with access protection. It must be possible to compile the original project and – if present – the source files without error. The block folder must contain all called blocks and must not contain any uncalled blocks. The message number assignment must be set to CPU-wide.

Removing unsupported hardware components

If the original project contains hardware components that are no longer available for the suitable application or for which the required option package is missing in STEP 7 V12, delete the non-supported configuration manually from the project: To open the original project, use an installation of STEP 7 V5.4 or V5.5 containing only option packages and modules available in STEP 7 V12 and save the project with the option *With reorganization*. Any unsupported configurations are removed from the project.

If modules are used in the original project that are only available in STEP 7 V12 in a newer version or with a newer firmware version, replace the older module with migratable modules in the hardware configuration: With the SIMATIC station selected in the SIMATIC Manager, double-click on the *Hardware* object, select the module in the hardware configuration, and select *Exchange Object* from the shortcut menu.

Migrating a project

Select the *Project* > *Migrate project* command in the main menu. Enter either the intermediate project with the file extension .am12 or the original project in the *Source path* box. Insert the name with the storage location for the target project, and also the author and a comment if applicable. Clicking on the *Migrate* button starts the migration.

The report generated during migration is displayed in the inspector window directly following the end of migration. You can also obtain this report if you select the project in the project tree, followed by the *Properties* command in the shortcut menu, and then click on the *Report file* link in the *Project progress* group.

Special characteristics for the migration of program blocks

Functions and statements are converted into the representation which is standard for the TIA Portal and may therefore deviate from the previous representation. Compatible system and standard blocks from a standard library are replaced with statements, which can be found in the program elements catalog. Standard blocks with an incompatible range of functions are created as know-how-protected user blocks with the extension "_LF" in the *Program blocks* folder.

In the TIA Portal, an operand addressed in absolute mode is assigned a symbolic address (a name). Together with the name, the operand is also assigned a data type. This results in a type conflict if the operand addressed in absolute mode is used together with functions which require different data types, for example if a memory word addressed in absolute mode is used in both an integer addition and in a shift function.

The name of an I/O operand is not imported. Instead of this, a ":P" is appended to the name of the input or output operand. Undefined input and output operands are assigned a (new) name in the process.

You migrate programs in libraries in that you copy the programs into a project and migrate the latter.

Stricter rules for checking in accordance with IEC directives in the TIA Portal may lead to errors during migration. For example, a check is now carried out for functions (FC) that own input parameters may no longer be written and own output parameters may no longer be read.

With jump labels, a distinction is no longer made between upper and lower case during the check for uniqueness; if applicable, jump labels are assigned new names.

A start value defined by the user in global data blocks is replaced by a default value. A start value defined in a type data block (specified by the user-defined data type UDT) is retained.

Migration of LAD and FBD blocks

Blocks with jumps to downstream networks that were created in LAD or FBD are represented in STL following the migration, even though the programming language remains set to LAD or FBD. To correct the representation, you set the programming language in the block properties to STL and subsequently back to LAD or FBD again.

Program sources are not migrated. This has effects on the know-how protection: A block with know-how protection remains protected even following migration. However, the know-how protection can no longer be canceled since the program source is no longer available. Therefore remove the know-how protection prior to migration and protect the block following the migration using the *Edit > Know-how protection* command.

Migration of SCL blocks

A prerequisite for the migration of SCL blocks is that an S7-SCL V5.3 SP5 (or later) option package is installed.

If the program sources of SCL blocks are missing in the original project, these blocks are migrated into blocks with know-how protection. If the program sources are present, these blocks are migrated into non-protected blocks. If applicable, you must then protect the associated blocks again using the *Edit* > *Know-how protection* command.

The following are no longer present with SCL blocks in the TIA Portal:

- Jump labels in the declaration part (the jump labels are retained in the program)
- Symbolic constants in the declaration part (these are replaced by global user-defined constants, with a different name in the event of conflicting names)
- Symbolic constants as limits for the ARRAY declaration (these are replaced by fixed values)
- Nested ARRAY tags (they are replaced with multi-dimensional arrays)
- The DIV operator (this is replaced by the slash "/")
- The EXPD function (this is replaced by the notation "10**")
- The LOG function (this is replaced by the notation "LN(<Expression>)".

A floating-point number is always specified as a fractional number (for example, "12E2" becomes "12.0E2"). Absolute or indirect addressing of data operands is only possible with an absolutely addressed data block (for example, "DB_name".DW22 becomes %DB10.DW22).

Some of the standard functions from the *IEC Function Blocks* library are converted during the migration into functions which are available in the TIA Portal:

- S_COMP (comparison of STRING tags)
- S_CONV (data type conversion) or into the notation
  *Source data type*_TO_*Destination data type*
- T_COMP (comparison of time data types)
- T_CONV (data type conversion) or into the notation
  *Source data type*_TO_*Destination data type*
- T_ADD, T_SUB, and T_COMBINE

A syntax error is output if unambiguous conversion is not possible.

The EN/ENO mechanism with SCL programs is adapted to that of the TIA Portal: The OK tag is replaced by the ENO tag, which simultaneously controls the ENO output. Following the migration, the previous positions of use of OK tags must also be adapted to the new EN/ENO mechanism if applicable.

Migration of GRAPH blocks

A prerequisite for the migration of GRAPH blocks is that an S7-GRAPH V5.3 SP6 (or later) option package is installed.

The GRAPH-specific block settings are significantly reduced in the TIA Portal and this can result in changes in the interface. It may be necessary in this case to update the block call and to regenerate the instance data block.

18.2.2 Upgrading a project

A project created with STEP 7 V11 can also be edited with STEP V12. Backwards compatibility is retained, which means that you can continue to edit the project using STEP 7 V11. But this means that the range of functions will be limited to the STEP 7 V11 options. If you want to use the full range of functions of STEP 7 V12, you must upgrade the project. The same applies for global libraries.

To upgrade, open the project that was created with STEP 7 V11 using the command *Project > Upgrade* from the main menu. After confirming the prompt, the original project is closed without changes and the new version of the project is opened.

18.3 Web server

CPUs with an Ethernet interface have a Web server that provides information from the CPU. To read out the information you require a Web browser which displays the information of the HTML pages.

18.3.1 Enable Web server

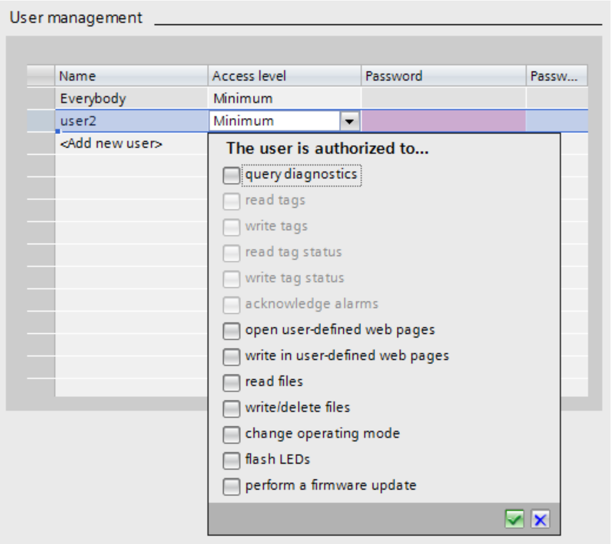
You enable the Web server with the hardware configuration using the *Activate web server on this module* checkbox in the CPU properties under the *Web server* group. For CPUs with several interfaces, you must also define the interface(s) in the *Overview of interfaces* group, via which access is to be allowed.

By activating the *Permit access only with HTTPS* checkbox you limit access to the secure hypertext transmission protocol. You additionally require a certificate for this which you can download and install via a link on the start page of the Web server. Furthermore, the time must be set on the CPU.

Further settings concern the time interval for automatic updating of the Web pages, the user administration, and the project language used (in the CPU properties under the group *User interface Languages*).

User management

If no users are configured, anyone can read all Web pages without being logged on. A user “Everybody” can access all Web pages enabled for the user “Everybody” without being logged on and without a password. Access privileges to the Web pages can be assigned individually to a configured user with password.



User management

Name	Access level	Password	Passw...
Everybody	Minimum		
user2	Minimum		
<Add new user>			

The user is authorized to...

- □ query diagnostics
- □ read tags
- □ write tags
- □ read tag status
- □ write tag status
- □ acknowledge alarms
- □ open user-defined web pages
- □ write in user-defined web pages
- □ read files
- □ write/delete files
- □ change operating mode
- □ flash LEDs
- □ perform a firmware update

Fig. 18.3 User administration of the Web server

To create a new user, enter a user name in the next line and click in the *Access level* cell. Select the required privileges from the list (Fig. 18.3).

18.3.2 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*.

To enable logging on, two input boxes are provided for the user name and password on the start page at the top left.

Automatic updating is disabled in the basic setting and the Web pages therefore deliver static information. You can switch the automatic updating on and off using the function key F5 or the *Enable/disable automatic refresh* symbol at the top right on the displayed page. If Web pages are printed, their contents are always up-to-date.

18.3.3 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. If you want to skip this intro page in the future, activate the *Skip Intro* option.

Start page

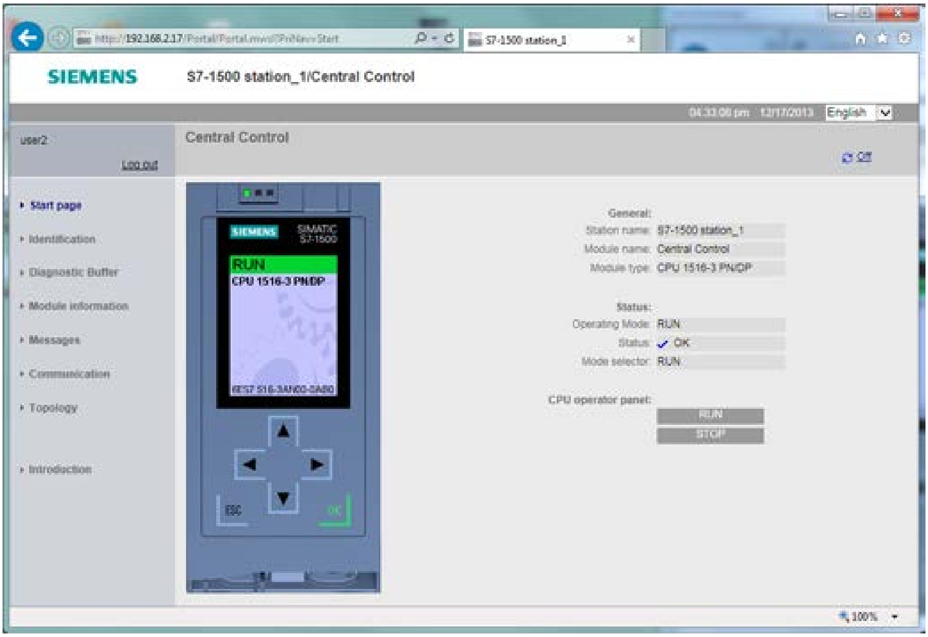
The *Start page* shows the station name, the module name, the module type of the CPU, and the status at the time of scanning: operating mode, diagnostics state, and position of the mode switch (Fig. 18.4).

Identification

The *Identification* page contains the plant designation and location identifier, the serial number, the order number, and the version information of the hardware, firmware, and boot loader.

Diagnostic Buffer

The *Diagnostic Buffer* page shows the contents of this buffer. The maximum size of the diagnostics buffer depends on the CPU used; the size used can be configured. Select the group to be displayed from the drop-down list. Detailed information is displayed on the selected event (Fig. 18.5).


S7-1500 station_1/Central Control

04:33:00 pm 12/17/2013 English

user2

Central Control

General:
Station name: S7-1500 station_1
Module name: Central Control
Module type: CPU 1516-3 PN/DP

Status:
Operating Mode: RUN
Status: ✓ OK
Mode selector: RUN

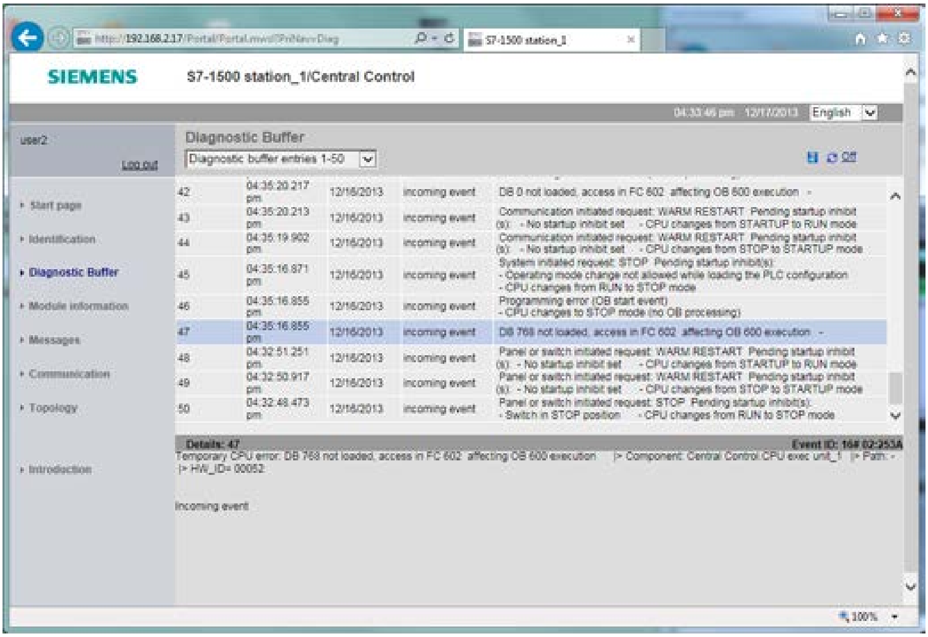
CPU operator panel:

Fig. 18.4 Start page of the Web server

You can select the display language in the window at the top right. If the selected language is not configured, the information is displayed in hexadecimal code.

Module information

The *Module information* page shows the status of the S7-1500 station. The status and the identification of individual modules and, if available, individual submodules of the modules can be called from here. Use the link in the “Heading” to access a higher object level, the link in the table column *Name* (under *Details*) to access a lower object level.



S7-1500 station_1/Central Control

04:33:46 pm 12/17/2013

Diagnostic Buffer

Diagnostic buffer entries 1-50

42	04:35:20.217 pm	12/16/2013	incoming event	DB 0 not loaded, access in FC 602 affecting OB 600 execution -
43	04:35:20.213 pm	12/16/2013	incoming event	Communication initiated request: WARM RESTART Pending startup inhibit (s): - No startup inhibit set - CPU changes from STARTUP to RUN mode
44	04:35:19.902 pm	12/16/2013	incoming event	Communication initiated request: WARM RESTART Pending startup inhibit (s): - No startup inhibit set - CPU changes from STOP to STARTUP mode
45	04:35:16.871 pm	12/16/2013	incoming event	System initiated request: STOP Pending startup inhibit(s): - Operating mode change not allowed while loading the PLC configuration - CPU changes from RUN to STOP mode
46	04:35:16.855 pm	12/16/2013	incoming event	Programming error (OB start event) - CPU changes to STOP mode (no OB processing)
47	04:35:16.855 pm	12/16/2013	incoming event	DB 768 not loaded, access in FC 602 affecting OB 600 execution -
48	04:32:51.251 pm	12/16/2013	incoming event	Panel or switch initiated request: WARM RESTART Pending startup inhibit (s): - No startup inhibit set - CPU changes from STARTUP to RUN mode
49	04:32:50.917 pm	12/16/2013	incoming event	Panel or switch initiated request: WARM RESTART Pending startup inhibit (s): - No startup inhibit set - CPU changes from STOP to STARTUP mode
50	04:32:48.473 pm	12/16/2013	incoming event	Panel or switch initiated request: STOP Pending startup inhibit(s): - Switch in STOP position - CPU changes from RUN to STOP mode

Details: 47 Event ID: 16# 02:253A

Temporary CPU error: DB 768 not loaded, access in FC 602 affecting OB 600 execution |> Component: Central Control.CPU exec unit_1 |> Path: - |> HW_ID= 00052

Incoming event

Fig. 18.5 Display of diagnostics buffer in the Web browser

The status of a component is indicated by various symbols: *Component is OK, deactivated DP slave or IO device, state cannot be determined, component failed or is not reachable, maintenance required, maintenance demanded, error, and a module in a lower module level not have the status "component OK".*

Messages

The *Messages* page displays the configured messages in chronological order, including the date and time. You cannot acknowledge the messages via the Web browser.

You can search for specific information with filter settings. With sort functions, you can sort the messages, for example according to message number or status. Detailed information about the selected message is displayed.

You can select the display language in the window at the top right. If the selected language is not configured, the information is displayed in hexadecimal code.

Communication

The *Communication* page contains the *Parameters* and *Resources* tabs.

Information on the PROFINET interfaces can be found in the *Parameters* tab. The MAC and the IP address are displayed, for example, as are physical interface properties. The *Resources* tab shows the number of available, reserved and occupied connections.

Topology

The *Topology* page shows the topological structure and the status of a PROFINET IO system. The *Graphic view* tab shows the reference topology and the actual topology in a graphic representation, the *Table view* tab shows only the actual topology.

The *Status overview* tab shows the status of all PROFINET IO devices present in the project without displaying the connection relationships and thus permits fast locating of the error location.

Customer pages

On the *Customer pages* page, the Web server shows the link to user-programmed Web pages. 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.

File browser

The file browser shows the directories and files that are located on the memory card, with the exception of the system files. The files can be downloaded, deleted, renamed, or uploaded. Directories can be created, deleted, and renamed.

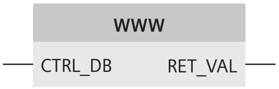
18.3.4 Read out service data

You can read out service data from the CPU, e.g. the contents of the diagnostics buffer, via the Web server. If a problem should occur with the CPU which cannot be resolved otherwise, you can send this service data to the Siemens Service&Support.

To read out the service data, open a Web browser and enter *https://<IP address>/save_service_data.html* as the address. On the page that appears for the service data, click on *Save ServiceData*. The service data is then saved in the file *<Order number><Serial number><Time stamp>.dmp*.

System block for synchronization of Web pages and user programs

Synchronize Web pages



The "Start data block" is created at the CTRL_DB parameter and contains user-defined Web pages and references to further data blocks with user-defined Web pages.

Fig. 18.6 Synchronize Web pages

18.3.5 Initialize Web server and synchronize Web pages (WWW)

The system function WWW initializes user-defined pages in the Web server of the CPU and synchronizes access between the pages and the user data. The system function is called cyclically in the user program. You can find the system function in the program elements catalog in the section *Communication* under *WEB server* (Fig. 18.6).

18.4 Technology functions

18.4.1 Technology modules TM Count 2×24V and TM PosInput 2

The technology modules TM Count 2×24V and TM PosInput 2 are suitable for counting pulses, for measuring a frequency, a duration of a period or a speed, and for position input for motion control.

The technology module **TM Count 2×24V** has two counter channels with a counting range of 32 bits and a maximum signal frequency of 200 kHz, which results in a maximum counter frequency of 800 kHz for a fourfold evaluation. 24 V incremental encoders with and without signal N, 24 V pulse transmitters with and without a direction signal, and 24 V pulse transmitters with separate signals for forwards and backwards can be connected to the module. Each of the two counter channels has three digital inputs and two digital outputs.

The technology module **TM PosInput 2** has two counter channels with a counting range of 32 bits and a maximum signal frequency of 1 MHz, which results in a maximum counter frequency of 4 MHz for a fourfold evaluation. SSI absolute value encoders, RS422/TTL incremental encoders with and without signal N, RS422/TTL pulse transmitters with and without a direction signal and RS422/TTL pulse transmitters with separate signals for forwards and backwards can be connected to the module. Each of the two counter channels has two digital inputs and two digital outputs.

Both technology modules are configured using the editor from the hardware configuration. The user data interface is comprised of the control interface (32 bytes inputs) and the feedback interface (24 bytes outputs or 8 bytes in the operating mode *Position input for Motion Control*). You configure the counting and measuring functions with the technology object *High_Speed_Counter* and control them in the user program with the system block *High_Speed_Counter*. Alternatively, you can also control the technology functions directly via the control and feedback interface.

Depending on the parameterization, a digital input for starting, stopping, synchronizing or saving the count value (capture function) can be used. The result of a comparison with the count value can be displayed at a digital output.

With corresponding parameterization, the technology modules generate a diagnostic interrupt and hardware interrupts, such as for a zero crossing of the count value or if the counting direction has changed.

In distributed mode, the technology modules support the isochronous mode function, which means that the count, measuring and position values can be synchronously (simultaneously) recorded.

Encoder signals

An incremental encoder provides two rows of pulses, A and B, which are offset from one another by 90°. The counting direction is detected by the sequence of the signal edges. Any possibly existing signal N can be used to set the start value or to save the count value (capture function). With RS422 incremental encoders, the signals are present as differential signals A and /A, B and /B as well as N and /N.

A pulse transmitter provides a row of pulses A. Should the counting direction be detected, a signal B is required, which will specify the counting direction with its signal state. The counting direction can also be specified via the user program.

An SSI absolute value encoder (SSI = Synchronous Serial Interface) provides the absolute position value and a clock signal serially via the difference signals DAT and /DAT and CLK and /CLK. The position value is present in dual code or in gray code depending on the encoder used.

Counter function of the technology modules

The counter function can count within the range -2^{31} to 2^{31-1}. Within these maximum count limits, the count limits used during operation can be parameterized or specified via the user program. The response of the counter function upon reaching a count limit can be parameterized.

Within the count limits, a start value can be parameterized, specified via the user program, or set for specific events such as for a signal edge of a digital input (synchronization).

Using a gate control, a time window in which the counting pulses are recorded can be defined. The gate can be controlled using a digital input of the count channel or via the user program.

For a signal edge of a digital input, the current count value can be saved (capture function). Then, for incremental encoders and pulse transmitters, the counting can be continued using the current count value or the start value.

Using the comparison function, two values can be defined which control the two digital outputs of the count channel when the comparison condition is fulfilled. The comparison values can be parameterized and they can be modified via the user program. A parameterizable hysteresis for the comparison values prevents frequent switching if the current count value fluctuates by the comparison value.

Measurement function of the technology modules

Within a measuring interval, the mean frequency, time period and speed can be determined from the chronological progress of count pulses or position values. A frequency can be measured in the range from 0.04 Hz to 800 kHz (4 MHz) and a time period from 1.25 µs (0.25 µs) to 25 s. The values apply for the evaluation of all four signal edges for 24 V incremental encoders or, in parentheses, for RS422 incremental encoders.

A gate control defines the time window in which the measured value is determined. The interval in which the measured value is updated can be parameterized as update time.

Using the comparison function, two values can be defined which control the two digital outputs of the count channel when the comparison condition is fulfilled. The comparison values can be parameterized and they can be modified via the user program.

Configuring technology modules

Using the hardware configuration, open the PLC station and position the technology module in the rack by pressing and holding the mouse key and "dragging and releasing" the module from the hardware catalog into the working window. The inspector window shows the properties of the module in the *Properties* > *General* tab. Under *General* you set the project information and I&M data. Under *Module parameters*, select the startup response of the CPU from a drop-down list if the plugged-in technology module differs from the configured module: *From CPU*, *Start up CPU only if compatible*, or *Startup CPU even if mismatch*.

You can parameterize the technology functions in the group *Count 2x24V* > *Basic parameter* or *PosInput 2* > *Basic parameters* for each of the two channels. Under *Operating mode*, you activate one of the following options:

- *Operating with technology object*
  The parameterization and control are handled via a technology object.
- *Position input for Motion Control*
  The technology module is used for position detection for a higher-level Motion Control controller (applies to both channels).
- *Manual operation*
  The parameterization is done using the hardware configuration, the control is implemented in the user program via the control and feedback interface.

Depending on the operating mode, there are different templates for the parameterization. In all of the modes, you define the response of the technology module for the CPU STOP and you can release the diagnostic interrupt.

For *Operating with technology object* you can enable hardware interrupts, for example for a zero crossing of the count value. With the enable action, you give the event a name and assign a hardware interrupt organization block.

For the *Position input for Motion Control* you select, for example, the signal type (e.g. incremental encoder), the signal evaluation (once, twice, or fourfold), the filter frequency, and the sensor type (P switch, M switch, push-pull) from drop-down lists under *Module parameters*.

For *Manual operation*, set the function *Counting/Position input* or *Measuring*. In addition, you can enable hardware interrupts, for example for a gate stop. With the enable action, you give the event a name and assign a hardware interrupt organization block.

Under *I/O addresses* you can set (separately for inputs and outputs) the logical start addresses, the process image in which the addresses are located, and the assigned organization block.

18.4.2 Technology objects for counting and measuring

With the technology object *High_Speed_Counter* you parameterize and control the counter and measurement functions of the technology modules TM Count 2×24V and TM PosInput 2. As a prerequisite, the technology module must be configured accordingly with the hardware configuration. To create the technology object, open the *Technology objects* folder in the project tree under the PLC station and double-click on *Add new object*. The *High_Speed_Counter* technology object is the instance data block for the *High_Speed_Counter* system block. In the dialog window, select the *Counting and Measurement* button and then click on *High_Speed_Counter*. You can change the displayed name and the data block number if you activate the *manual* option (Fig. 18.7).

Click the *OK* button to create the technology object and display the parameter assignment in the working window. You open the same parameterization window if you double-click on the *Configuration* editor in the project tree under the technology object.

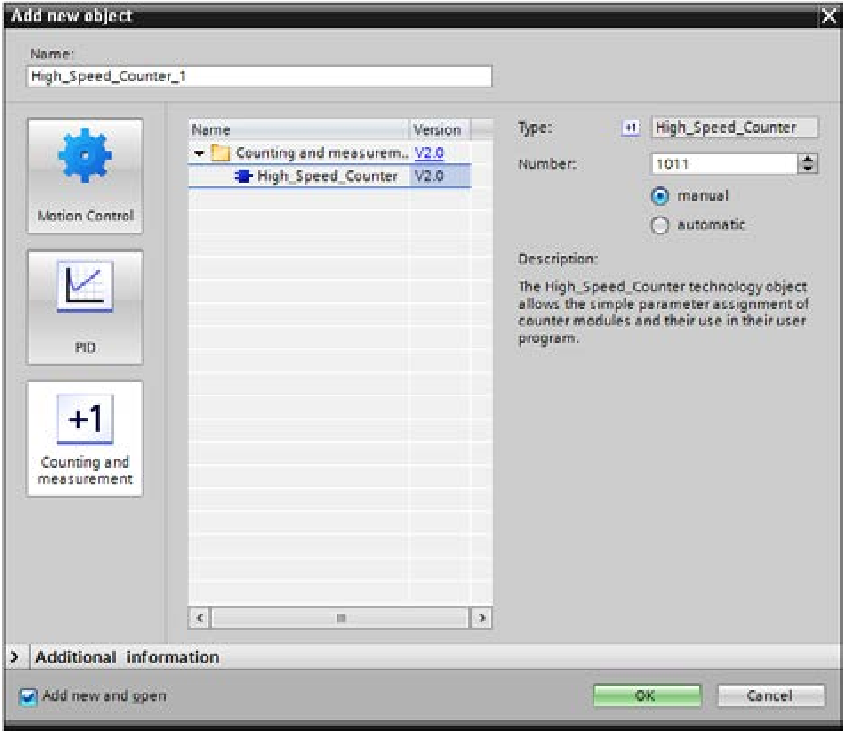
Under *Basic parameters* > *Module*, specify the technology module and the channel for which the technology object is determined. The counter channel is parameterized in the *Extended parameters* group.

Under *Counter inputs*, define the signal type of the connected encoder, the signal evaluation (once, twice, or fourfold), the filter frequency, and the sensor type (P switch, M switch, push-pull). For push-pull signals, you can activate the monitoring for a wire break.

Under *Counter behavior*, you set the counter limits, the start value, and the response when a count limit is exceeded and during a gate start.

Under *Behavior of DI0/DI1/DI2*, you set the response for each of the three digital inputs of the counter channel. You define which function is to trigger the digital input. Example: If the counter gate is only to be opened when the digital input has signal state “1”, set the function to *Gate start/stop (level-triggered)* and the option *Active with high level*.

Under *Behavior of DQ0/DQ1*, you set the response for each of the two digital outputs of the counter channel. You define the function for which the digital output is to be set, for example, when the current count value is between the comparison value 0 and the upper count limit. The selection depends on the set operating mode Counting or Measurement. In addition, you specify the replacement value for the digital output if the CPU switches to the STOP operating state.


Add new object

Name: High_Speed_Counter_1

Name	Version
Counting and measurem..	V2.0
High_Speed_Counter	V2.0

Type: High_Speed_Counter

Number: 1011

● manual

○ automatic

Description:

The High_Speed_Counter technology object allows the simple parameter assignment of counter modules and their use in their user program.

☑ Add new and open

Fig. 18.7 Add new object dialog window

Under *Hysteresis* you set the hysteresis value. Under *Measured value* you select the measured value (Frequency, Period duration, Velocity) from a drop-down list and set the refresh time. If you select *Velocity* as the measured value, you must define the time basis for the speed measurement and the increments per unit.

High_Speed_Counter system block

Using a High_Speed_Counter technology object, the High_Speed_Counter system block controls a counter channel on a TM Count 2×24V or TM PosInput 2 technology module. As a prerequisite for the use of the system block, the technology module and the technology object must be configured. To call the system block, drag it from the program elements catalog under *Technology* > *Counting and Measurement* to the open block. Enter the name of the technology block as instance data block (Fig. 18.8).

The input parameter *SwGate* controls the counter gate. A rising edge opens the gate and a falling edge closes the gate. For a rising edge, *SetCountValue* transfers the static local tag *NewCountValue* as the new count value to the module. The capture function is released if the signal state at *CaptureEnable* is “1”. It is carried out when the signal state of the corresponding parameterized digital input changes. The synchronization function is released if the signal state at *SyncEnable* is “1”. It is carried out when the signal state of the corresponding parameterized digital input

Call box of the High Speed Counter technology function

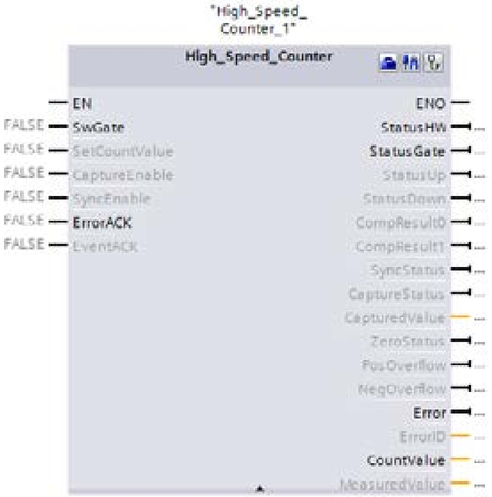


Fig. 18.8 Call box for a high-speed counter HSC

changes. The static local tags *SyncUpDirection* and *SynDownDirection* specify the direction for the synchronization. A rising edge at *ErrorACK* acknowledges the signaled error. A rising edge at *EventACK* resets the output parameters *CompResult0*, *CompResult1*, *ZeroStatus*, *PosOverflow*, and *NegOverflow*.

The output parameter *StatusHW* indicates the operational readiness of the module with signal state "1". *StatusGate* shows the enabling of the internal gate. *StatusUP* and *StatusDown* show that the last count pulse has increased or decreased the count value. *CompResult0* and *CompResult1* show that the comparison event for DQ0 or DQ1 has occurred. Both parameters are reset with a rising edge at *EventACK*.

SyncStatus has signal state "1" if a synchronization has occurred. With a falling edge at the input parameter *SyncEnable* or at the static local tags *SyncUpDirection* or *SyncDownDirection*, the parameter is reset. *CaptureStatus* has signal state "1" if the capture function has been executed. It is reset with a falling edge at the *CaptureEnable* parameter . *ZeroStatus* has signal state "1" if the current count value *CountValue* has reached the value zero. A rising edge at *EventACK* resets *ZeroStatus*.

PosOverflow and *NegOverflow* are set to signal state "1" if the current count value *CountValue* has overshot the upper count limit or undershot the lower count limit. A rising edge at *EventACK* resets *PosOverflow* and *NegOverflow*. With signal state "1", the *Error* parameter shows that an error has occurred, the ID of which can be read at the parameter *ErrorID*.

The current count value can be output at the *CountValue* parameter, the capture value can be output at the *CapturedValue* parameter, and the measured value for frequency, time period or speed can be output at the *MeasuredValue* parameter.

18.4.3 Technology objects for motion control

The technology function Motion Control in a CPU 1500 supports the closed-loop positioning and traversing of axes. Depending on the design, the drives can be controlled via a PROFIdrive message frame or an analog output. The encoder signals can be connected via PROFINET IO, PROFIBUS DP, or a technology module in the central controller. Motion control instructions control the technology objects in the user program.

The technology object *TO_SpeedAxis* calculates the setpoint value for the speed and outputs it via a PROFIdrive message frame or an analog channel. The technology object *TO_PositioningAxis* calculates the position in a controlled way and outputs a corresponding speed setpoint value via a PROFIdrive message frame or an analog channel. The technology object *TO_ExternalEncoder* detects a position and makes it available to the user program.

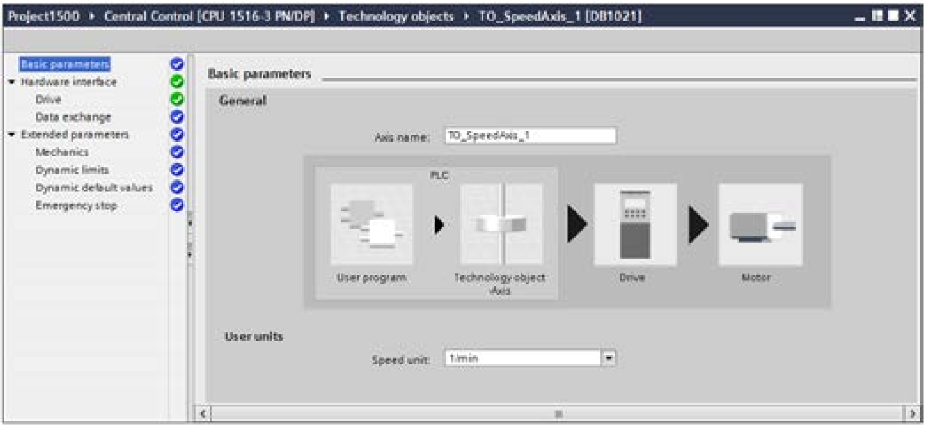
Adding a technology object

To create a technology object, open the *Technology objects* folder in the project tree under the PLC station and double-click on *Add new object*. Select the button *Motion Control* in the dialog window. Then click *TO_SpeedAxis* or *TO_PositioningAxis* in the folder *Motion Control* > *S7-1500 Motion Control* > *Axes* or click *TO_ExternalEncoder* in the folder *Motion Control* > *S7-1500 Motion Control* > *Other technology objects*.

The data for the technology objects is saved in data blocks. You can change the displayed name and the data block number if you activate the *manual* option. A technology object is stored in the project tree in the *Technology objects* folder.

Configuring the technology object *TO_SpeedAxis*

The *TO_SpeedAxis* technology object contains the editors *Configuration*, *Commissioning*, and *Diagnostics* in the *Technology objects* folder in the project tree. Double-click on *Configuration* to open the configuration window (Fig. 18.9). In the working


Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ Technology objects ▸ TO_SpeedAxis_1 [DB1021]

Basic parameters

General

Axis name: TO_SpeedAxis_1

User units

Speed unit: 1/min

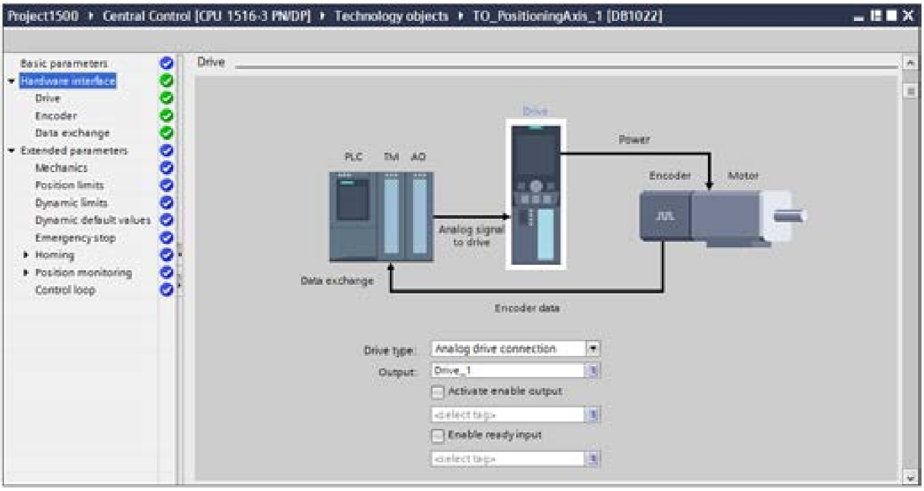
Fig. 18.9 Initial configuration screen for the *TO_SpeedAxis* technology object

window, select the desired parameter range and enter the configuration data for the axis.

To control a speed-controlled axis in the user program, the instructions referred to in Section "Instructions for motion control" on page 809 (*MC_Power*, *MC_MoveJog*, *MC_MoveVelocity*, *MC_Hold*, and *MC_Reset*) are available.

Configuring the technology object *TO_PositioningAxis*

The *TO_PositioningAxis* technology object contains the editors *Configuration*, *Commissioning*, and *Diagnostics* in the *Technology objects* folder in the project tree. Double-click on *Configuration* to open the configuration window (Fig. 18.10). In the working window, select the desired parameter range and enter the configuration data for the axis.


Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ Technology objects ▸ TO_PositioningAxis_1 [DB1022]

Drive type: Analog drive connection

Output: Drive_1

☐ Activate enable output

<select tag>

☐ Enable ready input

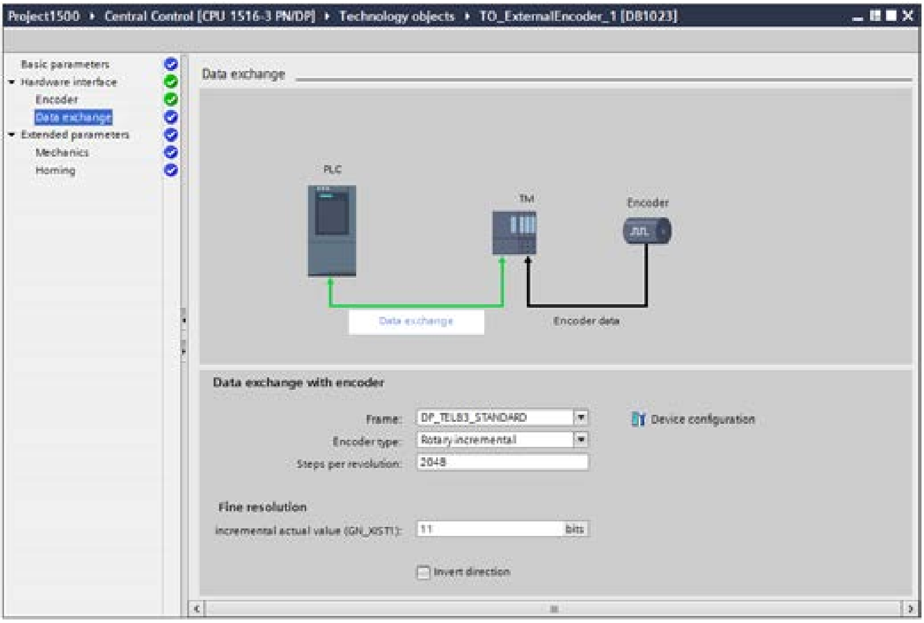
<select tag>

Fig. 18.10 Representation of the hardware interface of a positioning axis

To control a positioning axis in the user program, the instructions referred to in Section "Instructions for motion control" on page 809 (*MC_Power*, *MC_Home*, *MC_MoveJog*, *MC_MoveVelocity*, *MC_MoveRelative*, *MC_MoveAbsolute*, *MC_Hold*, and *MC_Reset*) are available.

Configuring the technology object *TO_ExternalEncoder*

The *TO_ExternalEncoder* technology object contains the editors *Configuration*, *Commissioning*, and *Diagnostics* in the *Technology objects* folder in the project tree. Double-click on *Configuration* to open the configuration window (Fig. 18.11). In the working window, select the desired parameter range and enter the configuration data for the axis.



Data exchange

Data exchange with encoder

Frame: DP_TEL83_STANDARD

Encoder type: Rotary incremental

Steps per revolution: 2048

Device configuration

Fine resolution

Incremental actual value (GN_XIST1): 11 bits

☐ Invert direction

Fig. 18.11 Configuring the data exchange for TO_ExternalEncoder

To control an external controller in the user program, the instructions referred to in the next Section "Statements for motion control" (*MC_Power*, *MC_Home*, and *MC_Reset*) are available.

Instructions for motion control

You control the *Axis* technology object and thus the drive with the user program using the motion control instructions. The instructions are available in the program elements catalog under *Technology* > *Motion control*. To call a instruction, 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. 18.12 shows the calls of the motion control instructions.

If a motion control instruction 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 instruction only once in the user program.

By assigning the AXIS parameter, you define the axis to be controlled by the motion control instructions. At AXIS you specify the data block which was generated when configuring the technology object.

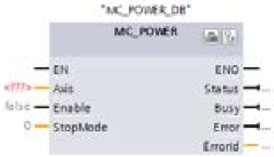
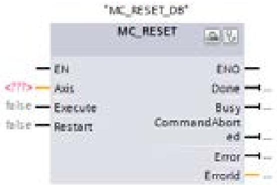
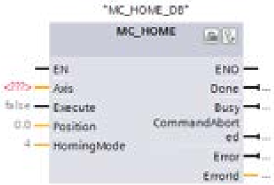
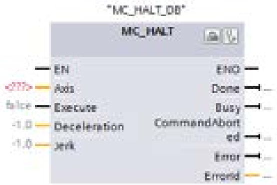
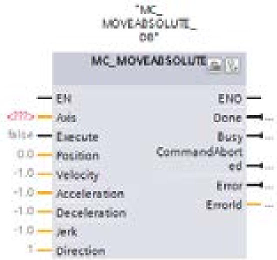
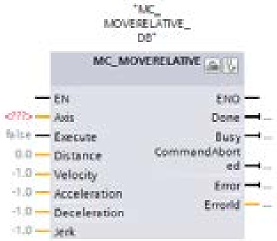
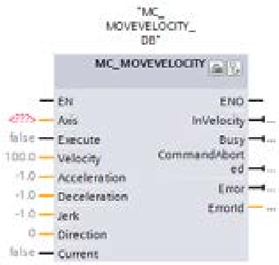
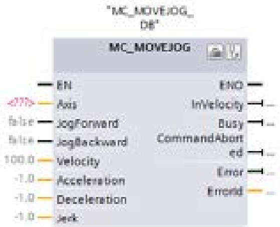
Call boxes of the instructions for motion control	
MC_Power activates and deactivates an axis for motion control. MC_Reset resets all errors and acknowledges all acknowledgeable errors.	 "MC_POWER_DB" MC_POWER EN ENO <???> Axis Status false Enable Busy 0 StopMode Error ErrorId "MC_RESET_DB" MC_RESET EN ENO <???> Axis Done false Execute Busy false Restart CommandAborted Error ErrorId
MC_Home sets a reference point, i.e. the (mechanical) positioning system of the axis is synchronized with the (logical) coordinate system of the controller. MC_Hold cancels all motion processes and stops the axis.	 "MC_HOME_DB" MC_HOME EN ENO <???> Axis Done false Execute Busy 0.0 Position CommandAborted 4 HomingMode Error ErrorId "MC_HALT_DB" MC_HALT EN ENO <???> Axis Done false Execute Busy -1.0 Deceleration CommandAborted -1.0 Jerk Error ErrorId
MC_MoveAbsolute starts positioning of the axis to an absolute position. MC_MoveRelative starts a positioning movement of the axis relative to the start position.	 "MC_MOVEABSOLUTE_DB" MC_MOVEABSOLUTE EN ENO <???> Axis Done false Execute Busy 0.0 Position CommandAborted -1.0 Velocity -1.0 Acceleration Error -1.0 Deceleration ErrorId -1.0 Jerk 1 Direction "MC_MOVERELATIVE_DB" MC_MOVERELATIVE EN ENO <???> Axis Done false Execute Busy 0.0 Distance CommandAborted -1.0 Velocity -1.0 Acceleration Error -1.0 Deceleration ErrorId -1.0 Jerk
MC_MoveVelocity starts the axis with the specified speed. MC_MoveJog starts the axis in jog mode during testing and commissioning.	 "MC_MOVEVELOCITY_DB" MC_MOVEVELOCITY EN ENO <???> Axis InVelocity false Execute Busy 100.0 Velocity CommandAborted -1.0 Acceleration -1.0 Deceleration Error -1.0 Jerk ErrorId 0 Direction false Current "MC_MOVEJOG_DB" MC_MOVEJOG EN ENO <???> Axis InVelocity false JogForward Busy false JogBackward CommandAborted 100.0 Velocity -1.0 Acceleration Error -1.0 Deceleration ErrorId -1.0 Jerk

Fig. 18.12 Calling the motion control instructions in LAD representation

18.4.4 Technology objects for PID control

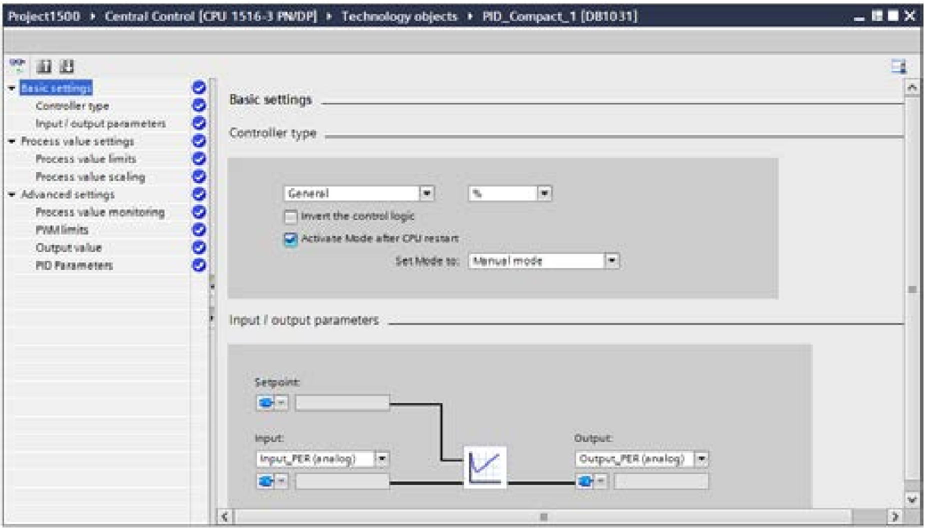
A PID controller continuously detects the measured *actual value* of the control variable in a control loop and compares it to the desired *setpoint value*. From the difference (of the control deviation), the PID controller calculates a *manipulated variable*, which conforms the control variable to the setpoint value.

For a PID controller, the control variable is made up of three parts: The *P part* is proportional to the control deviation, the *I part* is calculated by means of integration, it increases with the duration of the control deviation and finally leads to the offsetting of the control deviation, and the *D part* increases with the increasing speed of change of the control deviation in order to minimize the control deviation as quickly as possible.

With the technology objects for PID control, you create control loops with PID response and integrated optimization in manual and automatic mode. The instructions *PID_Compact* (universal PID controller) and *PID_3STEP* (PID controller with 3-point mode for valves) are available.

Configuring the *PID_Compact* technology object

The *PID_Compact* technology object is stored in the *Technology objects* folder in the project tree. It contains the editors *Configuration* and *Commissioning*. Double-click on *Configuration* to open the configuration window (Fig. 18.13). In the working window, select the desired parameter range and enter the configuration data for the PID controller.



Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ Technology objects ▸ PID_Compact_1 [DB1031]

Basic settings

Controller type

General ▾ % ▾

□ Invert the control logic

☑ Activate Mode after CPU restart

Set Mode to: Manual mode ▾

Input / output parameters

Setpoint:

Input: Input_PER (analog) ▾

Output: Output_PER (analog) ▾

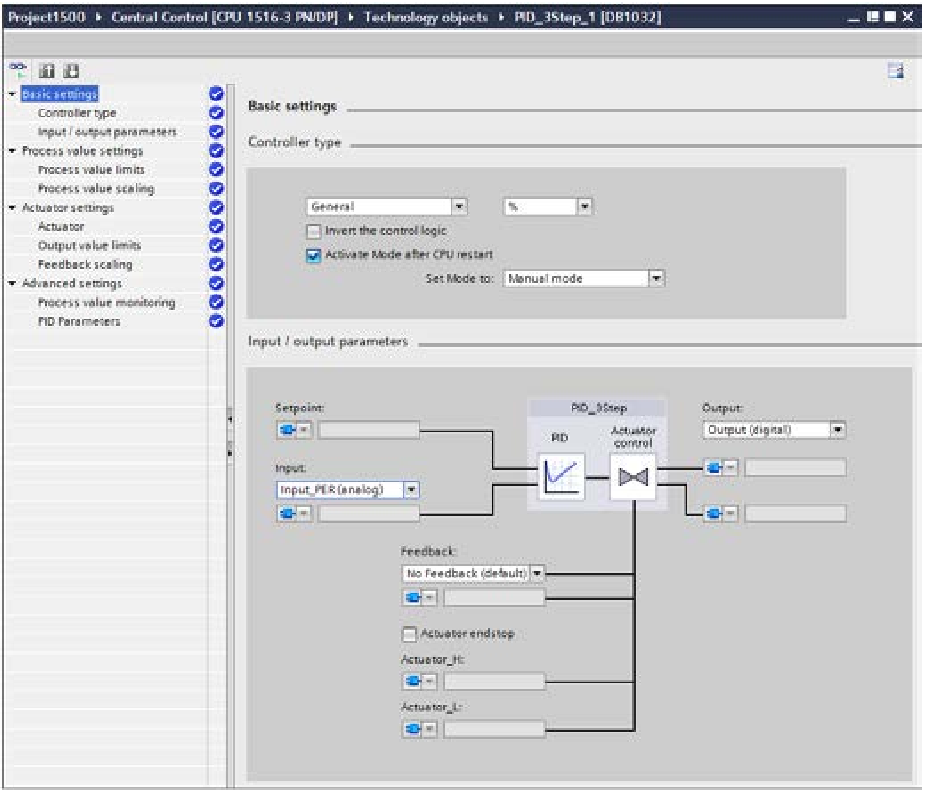
Fig. 18.13 Configuring the *PID_Compact* technology object

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 *PID_Compact* technology object is the instance data block for the *PID_Compact* instruction in the user program.

Configuring the *PID_3Step* technology object

The *PID_3Step* technology object is stored in the *Technology objects* folder in the project tree. It contains the editors *Configuration* and *Commissioning*. Double-click on *Configuration* to open the configuration window (Fig. 18.14). In the working window, select the desired parameter range and enter the configuration data for the PID controller.



Project1500 ▸ Central Control [CPU 1516-3 PN/DP] ▸ Technology objects ▸ PID_3Step_1 [DB1032]

Basic settings

Controller type

General %

□ Invert the control logic

☑ Activate Mode after CPU restart

Set Mode to: Manual mode

Input / output parameters

Setpoint:

Input: Input_PER (analog)

Output: Output (digital)

Feedback: No Feedback (default)

□ Actuator endstop

Actuator_H:

Actuator_L:

Fig. 18.14 Configuring the *PID_3Step* technology object

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).

The *PID_3Step* technology object is the instance data block for the *PID_3Step* instruction in the user program.

Instructions PID_Compact and PID_3Step

A 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 instruction must be called in a cyclic interrupt organization block whose call interval corresponds to the scanning time. The call can only be made as a single instance. The required instance data block corresponds to the technology object *PID_Compact* or *PID_3Step*.

To program the PID controller, create a cyclic interrupt OB with the desired scan time (see Chapter 5.7.4 “Cyclic interrupts” on page 203) 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 instruction from the *Instructions* 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. 18.15 shows the call of the controller instructions.

Call boxes of the technology objects PID_Compact and PID_3Step

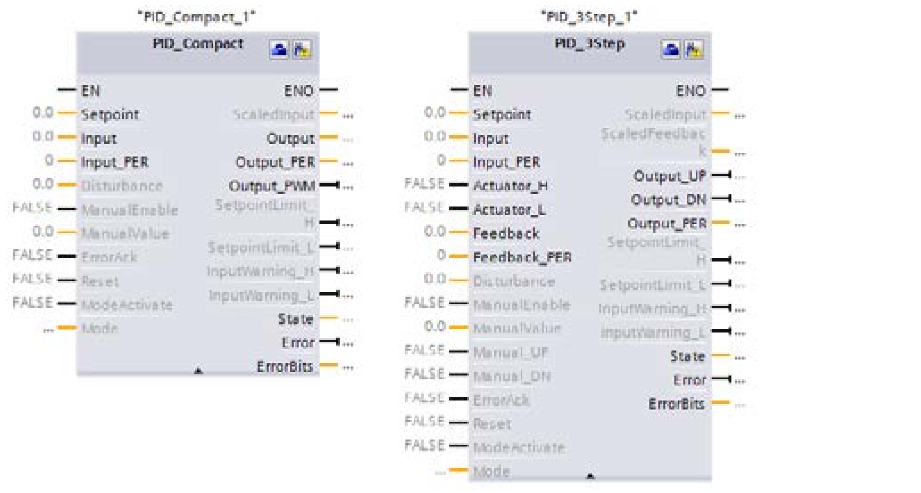


Fig. 18.15 Calls of the controller instructions *PID_Compact* or *PID_3Step* in LAD representation

18.5 Data logging and transferring recipes

With data logging, data blocks from the user program are written to the memory card in CSV format. With the recipe transfer, data blocks with recipes are transferred between the user program and the memory card.

18.5.1 Introduction to data logging

With data logging, selected process values from the user program are written to the data log file. The data log file for a CPU 1500 is on the memory card.

A data log file stores the values in CSV (Comma-Separated Values) format. The logged data can be read with a Web browser or with an SD card reader. A Web browser can read the data via the Web server that is available in the CPU, even if the

CPU is in the STOP operating state. The data on the memory card can also be directly accessed with an SD card reader on the programming device.

18.5.2 Using data logging

To use data logging, define a data buffer in a data area with the structure you require, for example, with a PLC data type which you created. 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.

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 during the next write action. To prevent this, using *DataLogNewFile*, an additional data log file can be created based on the current data log file.

18.5.3 Functions for data logging

Fig. 18.16 shows the structure of data logging. You find these functions in the program elements catalog under *Extended instructions* > *Recipe and data logging*. Fig. 18.17 shows the call of functions for data logging in LAD representation.

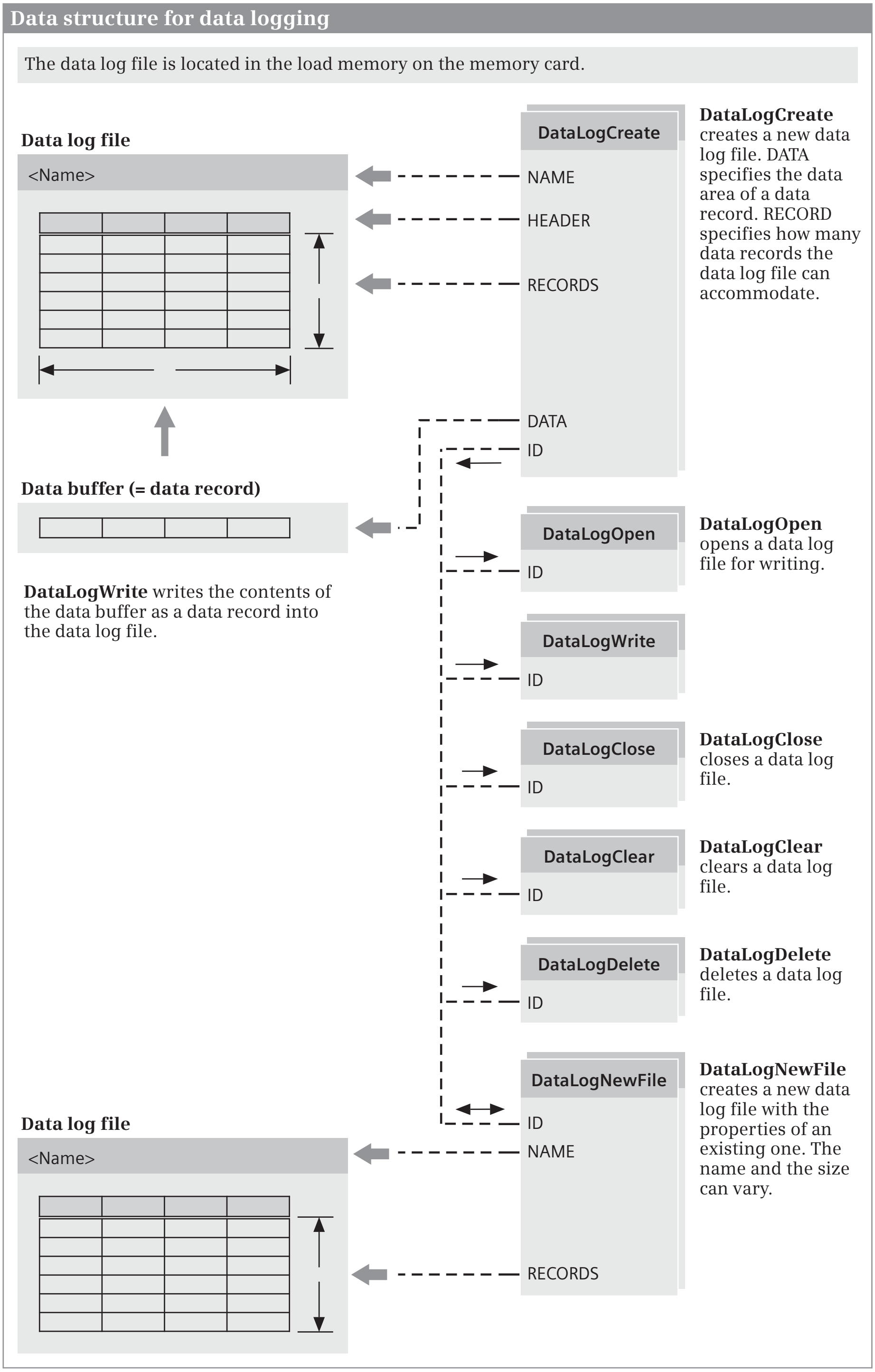
DataLogCreate creates a new (empty) data log file in the load memory. The parameters REQ, DONE, BUSY, ERROR, and STATUS control the execution of the function. At the NAME parameter, enter the name of the data log file, following the requirements for Windows file names. Further information on the data 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. Up to 10 data log files can be opened simultaneously. 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 states STARTUP and STOP.

DataLogClear deletes the data records in an open data log file. The HEADER is retained.



Data structure for data logging

The data log file is located in the load memory on the memory card.

DataLogCreate creates a new data log file. DATA specifies the data area of a data record. RECORD specifies how many data records the data log file can accommodate.

DataLogOpen opens a data log file for writing.

DataLogClose closes a data log file.

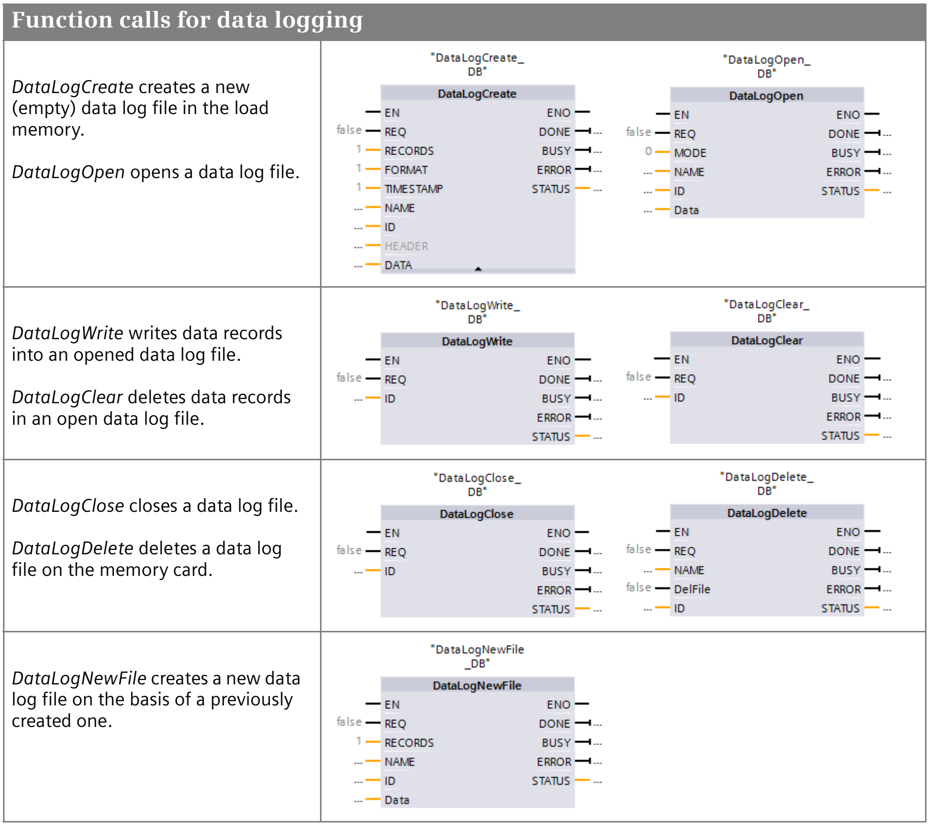
DataLogClear clears a data log file.

DataLogDelete deletes a data log file.

DataLogNewFile creates a new data log file with the properties of an existing one. The name and the size can vary.

DataLogWrite writes the contents of the data buffer as a data record into the data log file.

Fig. 18.16 Data structure for data logging



Function calls for data logging

DataLogCreate creates a new (empty) data log file in the load memory.

DataLogOpen opens a data log file.

DataLogWrite writes data records into an opened data log file.

DataLogClear deletes data records in an open data log file.

DataLogClose closes a data log file.

DataLogDelete deletes a data log file on the memory card.

DataLogNewFile creates a new data log file on the basis of a previously created one.

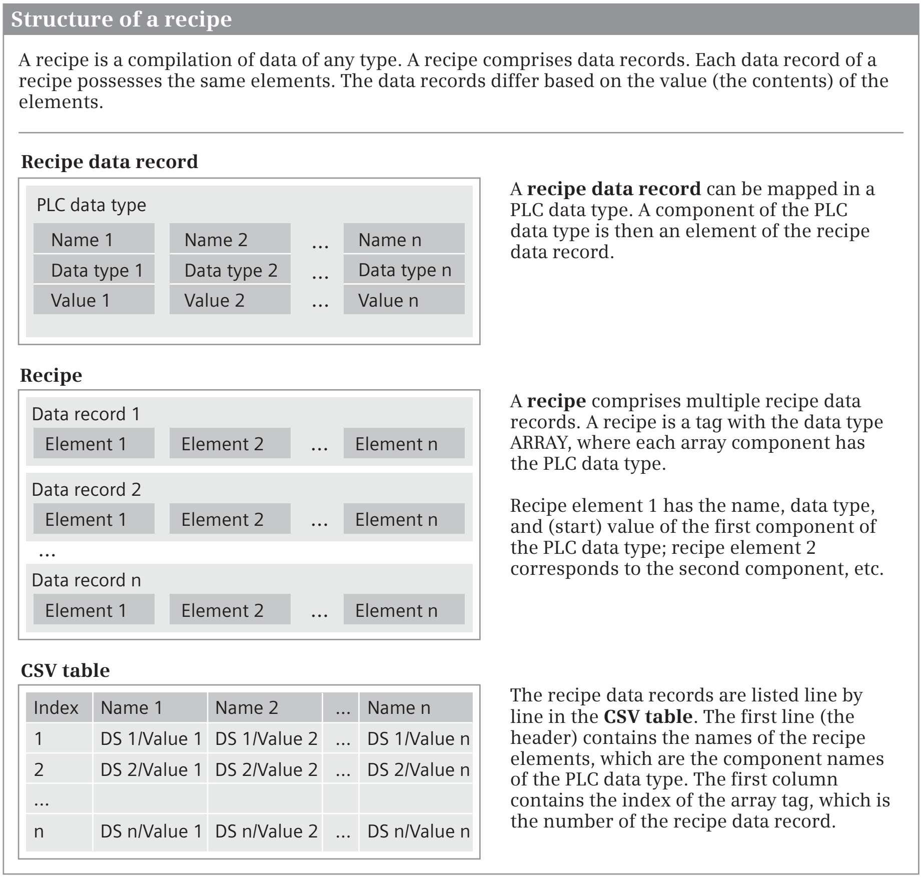
Fig. 18.17 Calls of the functions for data logging in LAD representation

DataLogDelete deletes a data log file on the memory card that was created using DataLogCreate or DataLogNewFile.

DataLogNewFile creates a new (empty) data log file with the same properties as the data log file whose identifier is specified at the ID parameter. With this, you can “expand” a full data log file, the oldest data record of which would have been overwritten during the next write action, with a new data log file. 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. After execution, the identifier of the newly created data log file is at the ID parameter.

18.5.4 Introduction to recipe transfer

Recipes contain data that belongs together such as data for a specific batch in production. A recipe is comprised of recipe data records, which differ in terms of values, but not in terms of their structure. A recipe data record is comprised of recipe elements, which can have different data types. (Fig. 18.18).



Structure of a recipe

A recipe is a compilation of data of any type. A recipe comprises data records. Each data record of a recipe possesses the same elements. The data records differ based on the value (the contents) of the elements.

Recipe data record

A **recipe data record** can be mapped in a PLC data type. A component of the PLC data type is then an element of the recipe data record.

Recipe

A **recipe** comprises multiple recipe data records. A recipe is a tag with the data type ARRAY, where each array component has the PLC data type.

Recipe element 1 has the name, data type, and (start) value of the first component of the PLC data type; recipe element 2 corresponds to the second component, etc.

CSV table

Index	Name 1	Name 2	...	Name n
1	DS 1/Value 1	DS 1/Value 2	...	DS 1/Value n
2	DS 2/Value 1	DS 2/Value 2	...	DS 2/Value n
...				
n	DS n/Value 1	DS n/Value 2	...	DS n/Value n

The recipe data records are listed line by line in the **CSV table**. The first line (the header) contains the names of the recipe elements, which are the component names of the PLC data type. The first column contains the index of the array tag, which is the number of the recipe data record.

Fig. 18.18 Recipe components

A recipe file stores the values in CSV (Comma-Separated Values) format. The recipe data can be read with a Web browser or with an SD card reader. A Web browser can read the data via the Web server that is available in the CPU, even if the CPU is in the STOP operating state. The data on the memory card can also be directly accessed with an SD card reader on the programming device.

To create a recipe in a data block, map the recipe data record in a PLC data type. The components of the PLC data type are the recipe elements. You then create a tag with the data type ARRAY in a data block and assign the PLC data type to the ARRAY components. Example: With the declaration <Recipe name>: ARRAY [1 .. 10] OF <PLC data type>, 10 recipe data records with the structure of the PLC data type are created in the <Recipe name> tag.

There is a recipe data record in each line of the CSV file. The first column contains the index, i.e. the number of the data record. The next columns contains the recipe elements, i.e. the components of the PLC data type. The header in the first line contains the names of the recipe elements, i.e. the component names of the PLC data type.

18.5.5 Functions for the recipe transfer

You can find the functions for the recipe transfer in the program elements catalog under *Extended instructions* > *Recipes and data logging*. Fig. 18.19 shows the calls of the functions in ladder logic representation.

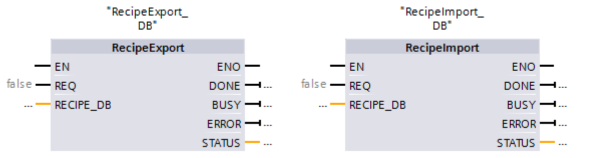
Function calls for the recipe transfer	
RecipeExport transfers a recipe from a data block in the user memory to a CSV file on the memory card. *RecipeImport* transfers a recipe from a CSV file on the memory card to a data block in the user memory.	 "RecipeExport_ DB" RecipeExport EN ENO false REQ DONE ... RECIPE_DB BUSY ERROR STATUS "RecipeImport_ DB" RecipeImport EN ENO false REQ DONE ... RECIPE_DB BUSY ERROR STATUS

Fig. 18.19 Calls of the functions for the recipe transfer in LAD representation

Exporting a recipe using RecipeExport

RecipeExport transfers a recipe from a data block in the user memory to a CSV file on the memory card.

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 error information is output at the STATUS parameter. The assigning of these status parameters is only valid for one cycle until the next processing of RecipeExport.

The parameter RECIPE_DB points to the recipe data block, the structure of which is described in Chapter 18.5.4 “Introduction to recipe transfer”.

Importing a recipe using RecipeImport

RecipeImport transfers a recipe from a CSV file on the memory card to a data block in the user memory.

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 error information is output at the STATUS parameter. The assigning of these status parameters is only valid for one cycle until the next processing of RecipeImport.

The parameter RECIPE_DB points to the recipe data block, the structure of which is described in Chapter 18.5.4 “Introduction to recipe transfer”.

18.6 Simulation with PLCSIM

The optional software S7-PLCSIM V12 simulates a PLC station in the main memory of the programming device. As a result, you can use the programming device to test the user program without additional hardware.

With STEP 7 you can configure and program the PLC station, then start the simulation, create a “simulation project”, and load the station data into the simulated PLC station. Now you can test the user program as described in Chapter 15.5 “Testing the user program” on page 677. SIM tables (watch tables) and sequence tables (tables with actions in a defined chronological order) are available as an interface to the simulated PLC station.

No connections to “real” programmable controllers must exist when using the simulation. Any existing online connections are cleared when starting the simulation.

You can carry out several simulations simultaneously. Each simulation runs independently, without being connected to any other simulation. All active simulations need different IP addresses.

18.6.1 Differences from a real CPU

S7-PLCSIM V12 supports the simulation of all of the CPU 1500 modules with firmware versions V1.0 and V1.1 and the associated signal modules (SM) with firmware versions V1.0 and V1.1. It is not possible, however, to emulate a real programmable controller 100%. Some differences from a “real” CPU 1500 are listed below. The following are not supported:

- The functionality of function modules (FMs) and communication modules (CPs and CMs)
- Network- and point-to-point communication
- An SD memory card and all of the associated functions such as data logging
- A Web server
- Blocks with know-how protection

Not all program functions which are based on system blocks are supported. Unsupported program functions are provided with valid output signals and are otherwise ignored. You can find a list of the supported program functions in the PLCSIM online help under *Simulations and STEP 7 > Instruction support*.

Configuration changes are not possible while a simulation is running. To change the hardware configuration, the simulated PLC station must be reloaded with the configuration that was changed (with STEP 7 in the TIA Portal).

18.6.2 Installing PLCSIM

S7-PLCSIM V12 runs under the Windows XP (32-bit) and Windows 7 (32 and 64-bit) operating systems. You must have administrator rights to install. If STEP 7 V12 or several instances of PLCSIM are to be executed simultaneously, the minimum recommended processor is a 2.2 GHz CORE 2 DUO (T7500) and 2 GB of main memory for Windows XP or 4 GB for Windows 7.

Installation, repair, and uninstalling are carried out using the setup program *start.exe* on the DVD. You can also uninstall PLCSIM normally in Windows using the *Software* application (Windows XP) or the *Programs and functions* application (Windows 7) in the Windows Control Panel.

No user authorization (license key) is required to install PLCSIM. However, PLCSIM only runs in connection with a valid license for STEP 7 in the TIA Portal.

18.6.3 Starting and saving the simulation

You have created a user program and compiled it without errors and it could therefore be executed on a CPU. You can then test it using PLCSIM.

If you are testing the user program of a PLC station with PLCSIM for the first time, you can start the simulation from STEP 7. Select the PLC station or the *Program blocks* folder in the STEP 7 project tree and then select the *Online* > *Simulation* > *Start* command from the main menu. PLCSIM is started and it creates a new “simulation project” with the file extension .sim12. You define the project name and path in the dialog window.

Alternatively, start the PLCSIM simultaneously with STEP 7 and create a new “simulation project” using the *Project* > *New* command from the PLCSIM main menu or open an existing one using the command *Project* > *Open...*.

Loading station data

A newly opened simulation project still does not contain any station data. Therefore, load the station data by selecting the PLC station in the project tree in STEP 7 and choosing the command *Download to device* > *Hardware and Software (only changes)* from the shortcut menu.

If there is still no connection to the simulated PLC station, specify the type of the PG/PC interface (*PN/IE*), the PG/PC interface (*PLCSIM S7-1200/S7-1500*) and, possibly, the interface in the PLC station in the *Extended download to device* dialog window. After the connection has been established, click on *Load*.

Before loading, STEP 7 compiles the station data and displays the result in the *Load preview* window. Fields highlighted in red in the *Action* column prevent loading. Clear the error messages, for example by selecting other actions, by eliminating errors, or by confirming by checking the checkbox, and click on *Load*. If you check the *Start all* checkbox when completing the loading process, the simulated PLC station starts up after you click on the *Finish* button.

The PLCSIM user interface

Fig. 18.20 shows the PLCSIM user interface. ① At the top you will find the main menu and the toolbar with the icons for frequently used commands and the specified IP address of the simulated PLC station. ② On the left side, the project tree win-

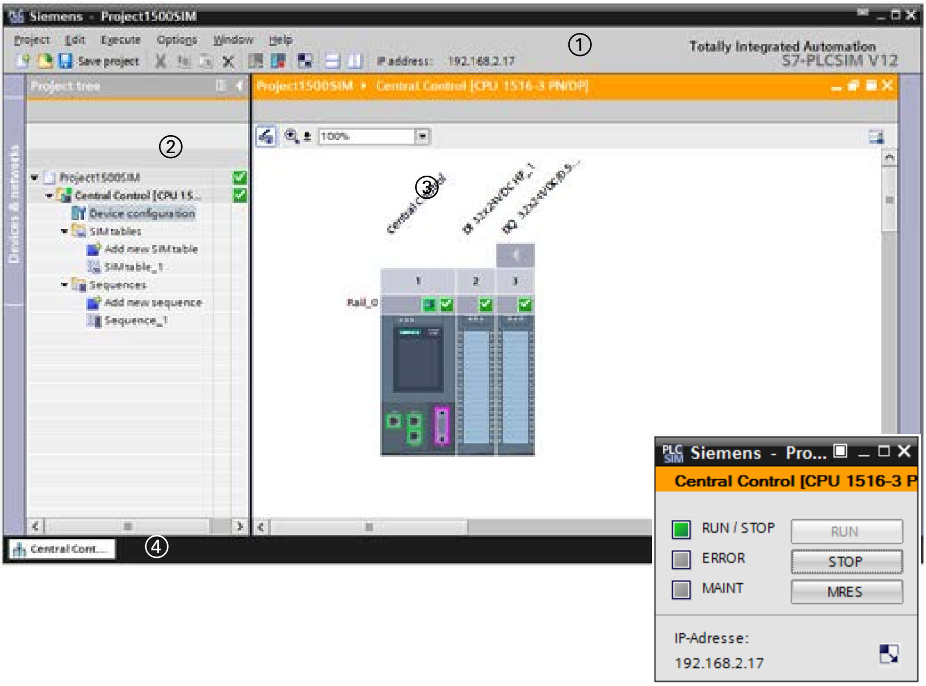


Fig. 18.20 Project window of the PLCSIM simulation software and compact view

dow shows the structure of the simulation project. ③ The objects selected in the project tree are displayed and edited in the working window. ④ The tabs of the objects opened in the working window are arranged on the lower edge of the user interface.

Use the *Window* > *Compact view* command or click on the corresponding icon to have the simulation window only display the CPU operator panel and the IP address. When you switch to the compact view, all of the information of the project view is retained and any unsaved editing work is saved. A consecutive sequence continues to execute.

Saving and restoring the simulation

The current state of a simulation is maintained in the SIM and sequence tables. You can save it and then load the saved state again later for continued processing.

Select the command *Project* > *Save* from the PLCSIM main menu. In addition to the SIM and sequence tables, the CPU type is also saved. The station data (configuration, user program) must be reloaded before the next simulation session.

To continue a simulation, start STEP 7 and PLCSIM. Open the respective projects and reload the station data from STEP 7 into the simulation project. Then you can continue with the simulation. The SIM tables and the sequence tables have the last displayed (saved) contents.

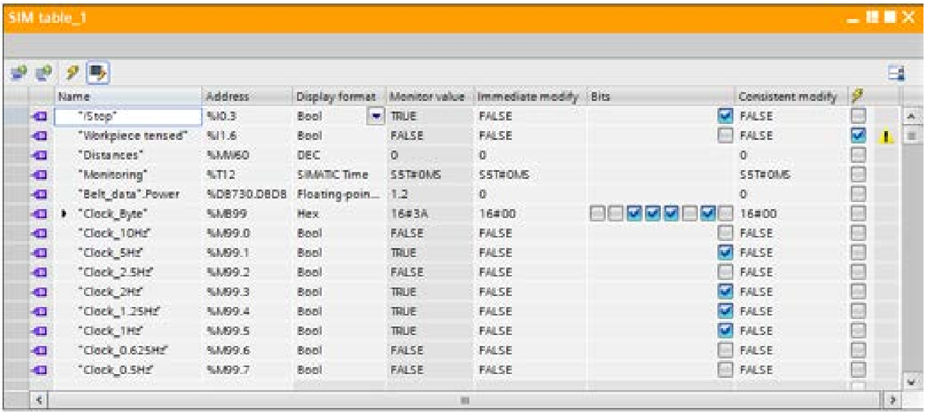
Change station data

In STEP 7 you can make changes to the station data such as supplementing the hardware configuration or program changes in blocks. Then load the changed data into the simulated PLC station using the command *Download to device* > ….

18.6.4 Testing with the SIM table

A SIM table contains tags whose values can be monitored and controlled in the simulation. The tags can be combined in any manner so that a specially tailored SIM table can be created for each test case. Several SIM tables, which can also be simultaneously opened, can be created for a simulated PLC station. Once a SIM table is opened, the monitoring of the tag values begins. Monitoring in an opened SIM table cannot be deactivated.

When the simulation project is created, an empty SIM table *SIMtable_1* is also created in the *SIM tables* folder. Open this SIM table with a double-click. You can create an additional SIM table by double-clicking on *Add new SIM table*. You can change the name of a SIM table with the command *Rename* from the shortcut menu. Fig. 18.21 shows an example of a SIM table.



SIM table_1

Name	Address	Display format	Monitor value	Immediate modify	Bits	Consistent modify		
"/Stop"	%I0.3	Bool	TRUE	FALSE	☑	FALSE	□	
"Workpiece tensed"	%I1.6	Bool	FALSE	FALSE	□	FALSE	☑	⚠
"Distances"	%MW60	DEC	0	0		0	□	
"Monitoring"	%T12	SIMATIC Time	S5T#0MS	S5T#0MS		S5T#0MS	□	
"Belt_data".Power	%DB730.DBD8	Floating-poin…	1.2	0		0	□	
"Clock_Byte"	%MB99	Hex	16#3A	16#00	□□☑☑☑□☑□	16#00	□	
"Clock_10Hz"	%M99.0	Bool	FALSE	FALSE	□	FALSE	□	
"Clock_5Hz"	%M99.1	Bool	TRUE	FALSE	☑	FALSE	□	
"Clock_2.5Hz"	%M99.2	Bool	FALSE	FALSE	□	FALSE	□	
"Clock_2Hz"	%M99.3	Bool	TRUE	FALSE	☑	FALSE	□	
"Clock_1.25Hz"	%M99.4	Bool	TRUE	FALSE	☑	FALSE	□	
"Clock_1Hz"	%M99.5	Bool	TRUE	FALSE	☑	FALSE	□	
"Clock_0.625Hz"	%M99.6	Bool	FALSE	FALSE	□	FALSE	□	
"Clock_0.5Hz"	%M99.7	Bool	FALSE	FALSE	□	FALSE	□	

Fig. 18.21 Example of a SIM table

Monitoring tags

Enter the tags to be controlled or monitored in the *Address* or *Name* column and select a different display format if applicable. The *Monitor value* column displays the current value of the tags.

If the tags are a bit or a byte, the *Bits* column displays a checkbox for each bit. If the checkbox is blank, the relevant bit has signal state “0”. Signal state “1” is indicated with a checkmark. For a byte, a triangle is displayed at the top of the *Name* column, which can be used to “open” the byte tag so that each bit occupies a line.

Modify tags

By default, only tags from the inputs and peripheral inputs areas can be controlled. If you would also like to assign a value to other tags, activate the icon *Enable/disable modification of non-inputs* in the toolbar of the SIM table.

Note that you cannot change the value of a tag which is controlled with an active force job.

Directly controlling tags

If you enter a value in the *Immediate modify* column, this value is immediately written into the tag. You can also achieve this immediate modification for a bit tag or byte tag by clicking on the checkbox that is assigned to the bit in the *Bits* column.

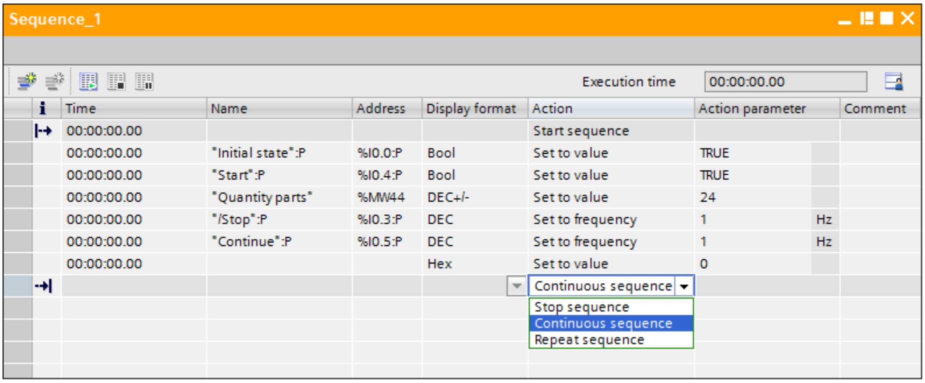
The *Bits* column always shows the current signal states. In contrast to this, the value in the *Immediate modify* column is not updated. You can use this behavior to conveniently always assign the same value to a tag with a value that changes during the simulation.

Consistently controlling tags

If you want to change the values of several tags at the same time, enter the values into the *Consistent modify* column. In the column with the lightning symbol, you define which values will be changed by checking the checkbox. The values will be changed if you click on the *Modify all selected values* icon in the toolbar of the SIM table.

18.6.5 Testing with the sequence table

In a sequence table, enter a chronological order of tag values (a “sequence”). When the sequence is started, the simulated PLC station is controlled with the tag values at the specified chronological interval. In this way, the chronological response of a “real” machine or system can be emulated.



Sequence_1

Execution time 00:00:00.00

i	Time	Name	Address	Display format	Action	Action parameter	Comment
|→	00:00:00.00				Start sequence		
	00:00:00.00	"Initial state":P	%I0.0:P	Bool	Set to value	TRUE	
	00:00:00.00	"Start":P	%I0.4:P	Bool	Set to value	TRUE	
	00:00:00.00	"Quantity parts"	%MW44	DEC+/-	Set to value	24	
	00:00:00.00	"/Stop":P	%I0.3:P	DEC	Set to frequency	1 Hz	
	00:00:00.00	"Continue":P	%I0.5:P	DEC	Set to frequency	1 Hz	
	00:00:00.00			Hex	Set to value	0	
→|					Continuous sequence		

Fig. 18.22 Example of a sequence table

You can create several sequence tables with one sequence each. Only one sequence can be carried at a time, however.

When the simulation project is created, an empty sequence table *Sequence_1* is created in the *Sequences* folder. You open this sequence table with a double-click. You can create an additional sequence table by double-clicking on *Add new sequence*. You can change the name of a sequence table with the command *Rename* from the shortcut menu. Fig. 18.22 shows an example of a sequence table.

Entering a sequence

The first step (initial step) and the last step (end of sequence) are defined for a sequence. You define the steps in-between. You can delete, copy and insert steps, and create new rows for additional steps using the *Insert step* and *Add step* icons in the toolbar of the sequence table.

A step consists of

- ▷ The point in time at which the specified action is started. The point in time is specified in the *Time* column in the format: Hours:Minutes:Seconds:Tenths of seconds.
- ▷ The tags, which you define in either the *Name* or *Address* column. The tag can be a peripheral input, an output, a bit memory, or a data tag. An entered input is converted into an peripheral input. In the address column, you can specify bit, byte, word or doubleword addresses.
- ▷ The action. Two actions are available: *Set to value* and *Set to frequency*. You specify the value or the frequency in Hz in the *Action parameter* column and you specify the format in the *Display format* column.

The action *Set to value* is carried out once at the specified time and can be applied to all of the tags. The action *Set to frequency* can only be applied to peripheral input bits. The peripheral input changes its signal state from the specified point in time until it is controlled with a new action or the sequence has elapsed.

The input sequence of the steps plays no role in this. When the sequence is executed, the steps are sorted in the specified chronological order (according to the entry in the *Time* column). You can sort the steps before execution by clicking in the header of the *Time* column.

Steps with different tags can be executed at the same point in time. The sequence of these actions is undefined.

In the last (defined) step of the sequence, you can specify a point in time for the end of the sequence. Prior to this, select the entry *Stop sequence* in the *Action* column. With the *Continuous sequence* action, the sequence will run until it is manually stopped. With *Repeat sequence*, the sequence continues to run from the beginning.

Executing a sequence

To start a sequence, open the corresponding sequence table and click on the *Start sequence* icon in the toolbar or select the *Start sequence* command from the shortcut menu.

After the sequence is started, PLCSIM removes all lines without a tag, sorts the lines according to the time of execution, and displays the steps in chronological order from top to bottom. In the information column on the far left of the table, the active step is displayed with a green arrow so that you can track the progress of sequence execution. In the upper right section of the table window, the execution time is running and the active sequence table is marked with a green triangle in the project tree.

To stop the sequence, click on the *Stop sequence* icon in the toolbar of the sequence table or select the command *Stop sequence* from the shortcut menu. A stopped sequence cannot be executed further.

Observe the responses to a sequence execution

You can track the responses of the user program while a sequence is executing using the test functions of STEP 7 (see next chapter) or in PLCSIM using a SIM table.

In PLCSIM, you can position a sequence table and a SIM table in parallel on the screen, for example by separating the tables from the PLCSIM window and displaying them as independent windows. You can also split the PLCSIM window into two sections with the sequence table and the SIM table using the *Split editor space horizontally* and *Split editor space vertically* icons. In this way, you can simultaneously track the changes of the tag values that interest you in the SIM table while the sequence is being executed.

18.6.6 Applying the test functions of STEP 7

The simulated PLC station almost behaves like a “real” PLC station, which you can virtually test online using the test functions of STEP 7. If it has not already taken place, start the PLCSIM, generate a new simulation project or open an existing one, and load the station data from STEP 7. Start the CPU when finishing the load process.

Connecting online

You can now establish an online connection from the (offline) project to the simulated PLC station, for example using the *Go online* icon in the toolbar. Then the offline and online version of the blocks are compared and the differences are displayed in the project tree of STEP 7, as described in section “Diagnostics icons in the project tree” on page 677.

Double-clicking on the *Online & diagnostics* command in the project tree starts the task card with the online tools. The CPU control panel, the cycle processing time, and the resources are displayed (see Chapter 15.4.5 “Online tools” on page 676).

Testing with program status

In the online mode, open the block to be tested and activate the program status, using the *Monitoring on/off* icon in the toolbar of the working window, for example. The signal states and tag values are displayed as described in Chapter 15.5.2 “Testing with program status” on page 679.

Monitoring with PLC tag table

In the online mode, open a PLC tag table and activate the monitoring mode, using the *Monitoring on/off* icon in the toolbar of the working window, for example. The signal states and the tag values are then displayed in the *Monitor value* column (see Chapter 15.5.3 “Monitoring of PLC tags” on page 682).

Monitoring of data tags

In the online mode, open the data block with the tags to be monitored and activate the monitoring mode, using the *Monitoring on/off* icon in the toolbar of the working window, for example. The signal states and the tag values are then displayed in the *Monitor value* column (see Chapter 15.5.4 “Monitoring of data tags” on page 683).

Testing with watch tables

In the online mode, open a watch table and activate the monitoring mode, using the *Monitoring on/off* icon in the toolbar of the working window, for example. You can then monitor and control the signal states and tag values as described in Chapter 15.5.5 “Testing with watch tables” on page 684.

Controlling with the force table

In the online mode, open the force table, specify the force values, and activate forcing as described in Chapter 15.5.6 “Testing with the force table” on page 689. The force job is transferred to the simulated PLC station and takes effect there. Forced I/O signals can now no longer be changed, for example with a watch or SIM table.

Index