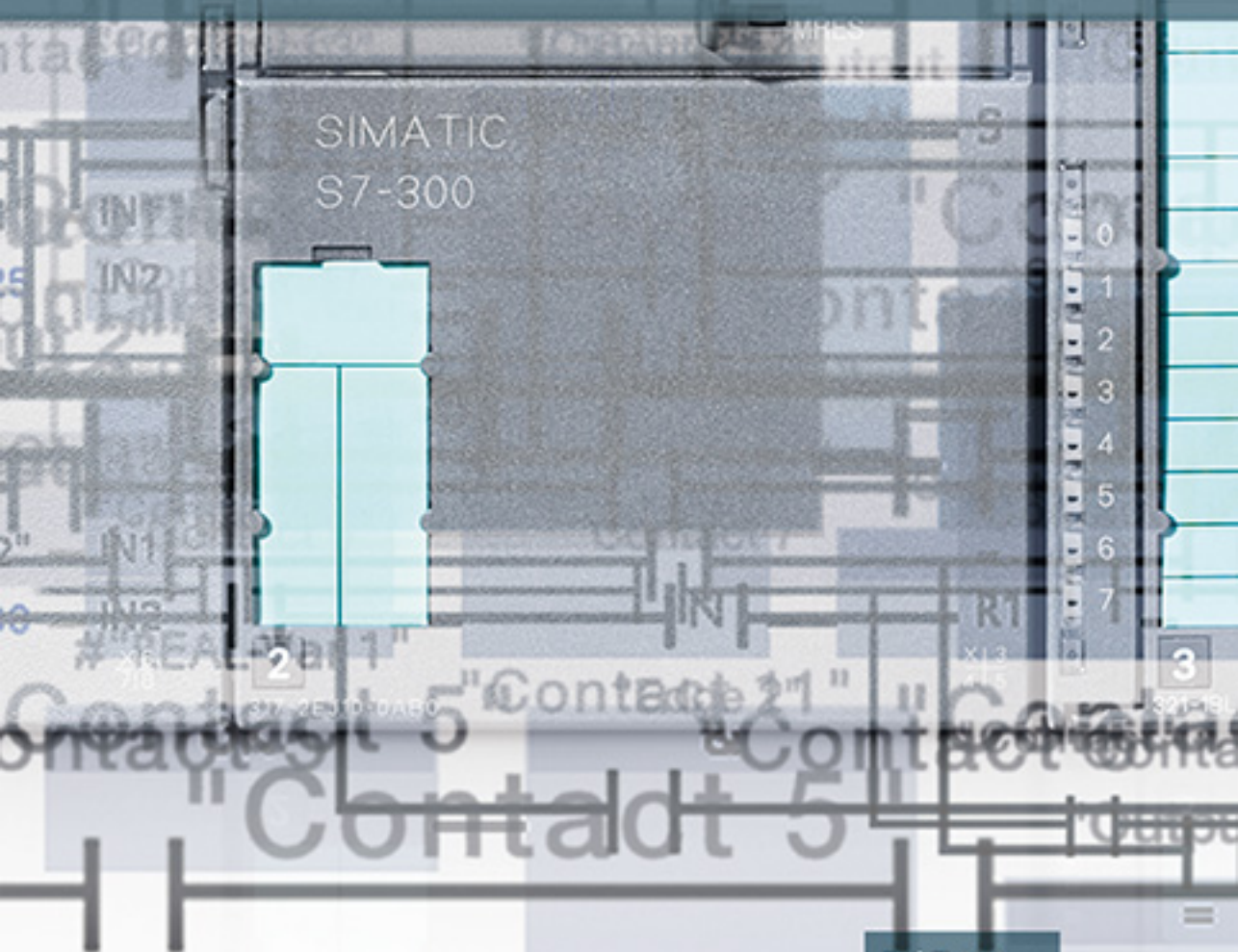
SIEMENS

Hans Berger

Automating with SIMATIC S7-300 inside TIA Portal

Configuring, Programming and Testing with STEP 7 Professional



2nd Edition

Berger Automating with SIMATIC S7-300 inside TIA Portal

Automating with SIMATIC S7-300 inside TIA Portal

Configuring, Programming and Testing
with STEP 7 Professional

by Hans Berger

2nd edition, 2014

Publicis Publishing

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

www.publicis-books.de

Print ISBN 978-3-89578-443-9
ePDF ISBN 978-3-89578-924-3

2nd edition, 2014

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 hardware components of the SIMATIC S7-300 automation system with standard controllers and the features provided for designing a distributed control concept with PROFIBUS and PROFINET. To permit communication with other automation systems, the controllers offer integrated bus interfaces for multipoint interface (MPI), PROFIBUS, and Industrial Ethernet.

The STEP 7 Professional engineering software inside TIA Portal makes it possible to use the complete functionality of the S7-300 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, statement list (STL) with formulation of the control task as a list of commands at machine level, a high-level Structured Control Language (SCL) similar to Pascal, 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 S7-300 automation system with the STEP 7 Professional engineering software Version 12 with Service Pack 1 Update 2.

Erlangen, June 2014 Hans Berger

The contents of the book at a glance

Start

Overview of the SIMATIC S7-300 automation system.

Introduction to the SIMATIC STEP 7 Professional V12 engineering software.

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

SIMATIC S7-300 automation system

Overview of the SIMATIC S7-300 modules: Design of an automation system, CPUs, signal, function 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 compound data types, data types for block parameters, pointers, and user 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 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 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 S7-GRAPH sequential controller

What a sequential control is, and what its elements are: sequencers, steps, transitions, and branches. How a sequential control is configured using S7-GRAPH.

Description of the control functions

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

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

Program flow control: Working with status bits, programming jump functions, calling and closing blocks, using the master control relay.

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.

How an actuator/sensor interface system is configured, and what properties it has.

Communication

The properties of S7 basic communication and of S7 communication, and with what communication functions they are programmed.

The communication functions used to implement open user communication.

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

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

How block parameters and local tags are saved in the memory.

Table of contents

1 Introduction

1.1 Overview of the S7-300 automation system

SIMATIC S7-300 is the modular mini PLC system for the lower and medium performance ranges (Fig. 1.1). 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 in up to four racks with eight modules each.

Further expansion with input/output modules is made possible by the distributed I/O over PROFIBUS or PROFINET. Special designs of these modules for increased mechanical demands allow their installation directly on site on the machine or plant. STEP 7 is used to configure and program the SIMATIC S7-300 controllers. Data exchange between the controllers, the distributed I/O, and the programming device is carried out over SIMATIC NET.

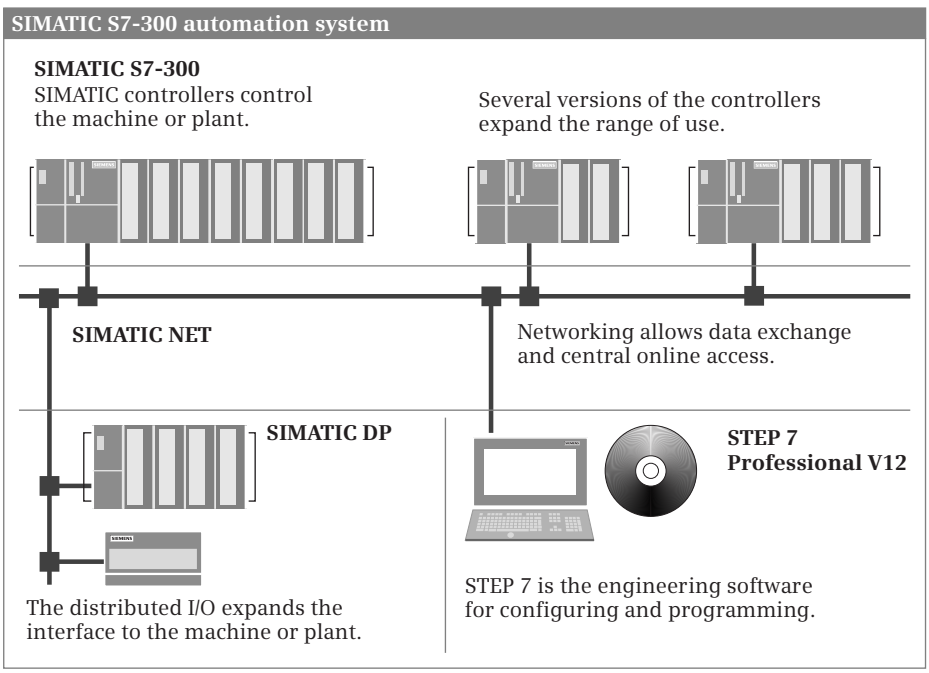


Fig. 1.1 Components of the SIMATIC S7-300 automation system

1.1.1 SIMATIC S7-300 programmable controller

The most important components of an S7-300 programmable controller are shown in Fig. 1.2.

The **CPU** contains the operating system and the user program. The user program is saved powerfail-proof on the *Micro Memory Card (MMC)*, 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 plant. These input and output modules are available for digital and analog signals.

The **function modules (FM)** are signal-preprocessing, "intelligent" I/O modules which prepare 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. Function modules are responsible for handling functions which the CPU cannot usually execute quickly enough, such as counting pulses, positioning, or controlling drives.

The **CP modules** allow data transfer in excess of the possibilities provided by the standard interfaces with regard to protocols and communication functions.

In the case of an expansion, the **interface modules (IM)** connect the central rack to a maximum of three expansion racks.

Finally, a **power supply module** provides the voltage required by the programmable controller.

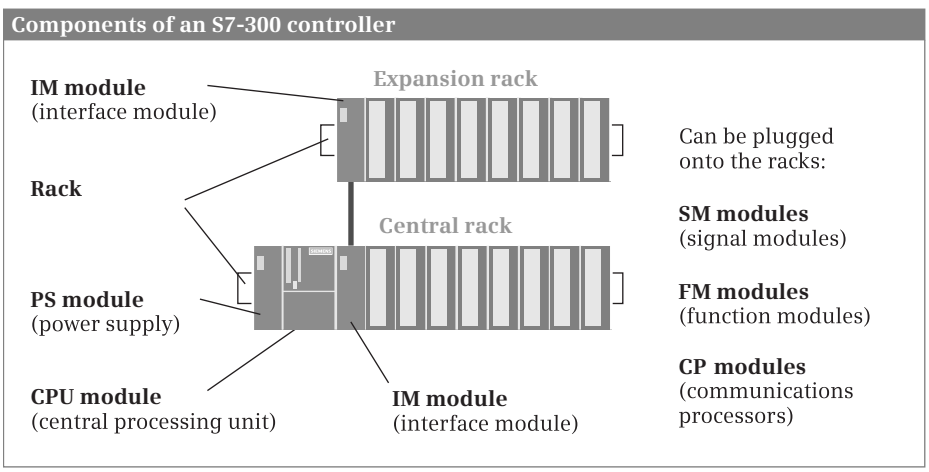


Fig. 1.2 Components of an S7-300 controller

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 S7-300 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 (Fig. 1.3):

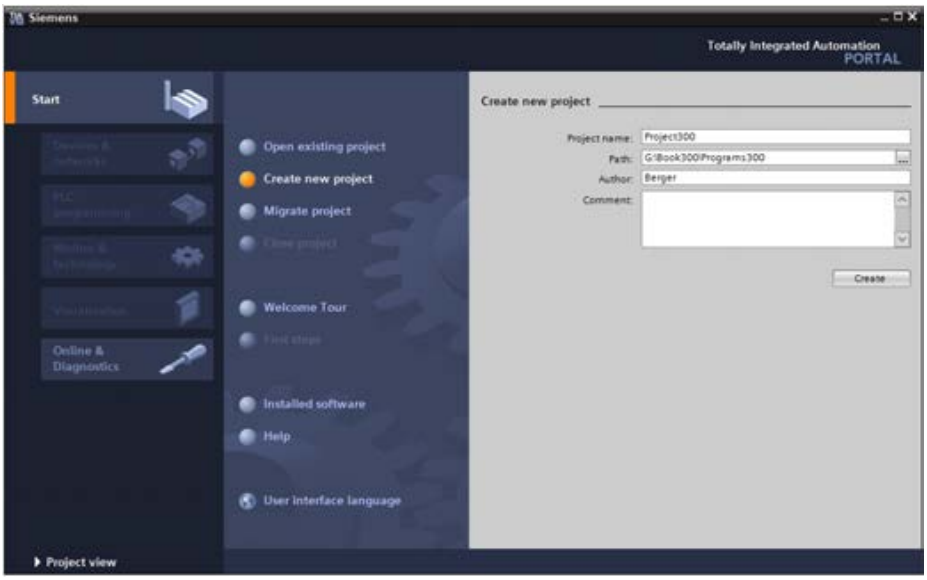


Fig. 1.3 Tools in the Start portal of STEP 7 Professional V12

- In the *Devices & networks* portal you configure the programmable controllers, i.e. you position the modules in a rack and set their parameters.
- In the *PLC programming* portal you create the user program in the form of individual sections referred to as "blocks".
- The *Visualization* portal provides the most important tools for configuration and simulation of HMI systems using SIMATIC WinCC.
- In the *Motion & Technology* portal, you insert a technology object for *PID Control* and edit it.

▷ The *Online & Diagnostics* portal allows you to connect the programming device online to a CPU. You can control the CPU's operating modes, and transfer and test the user program.

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

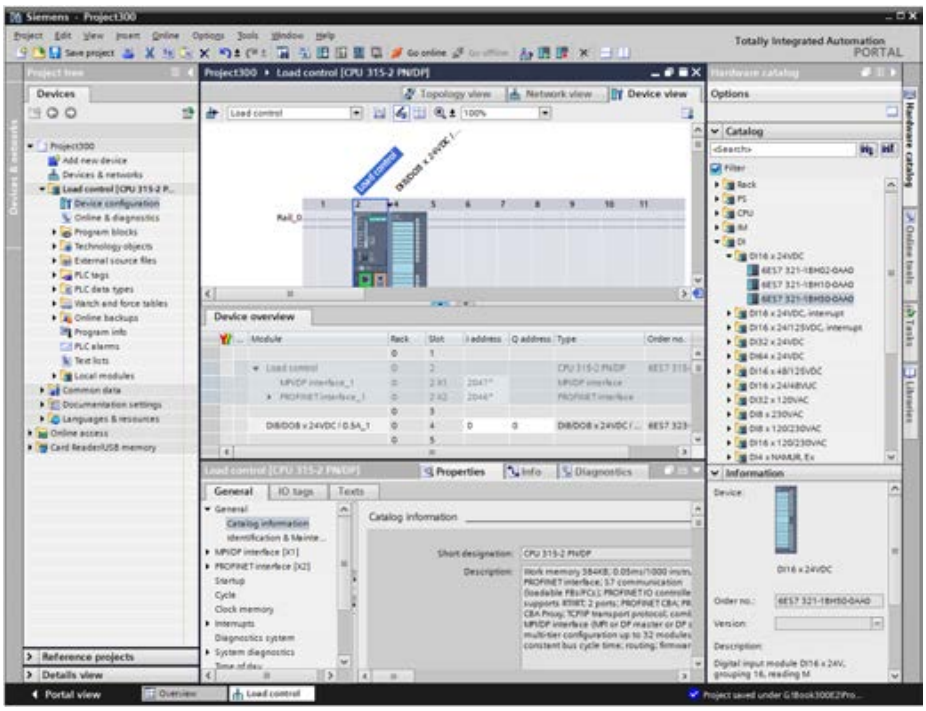


Fig. 1.4 Example of a Project view: working area of the device configuration

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 instructions. 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 Five programming languages

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

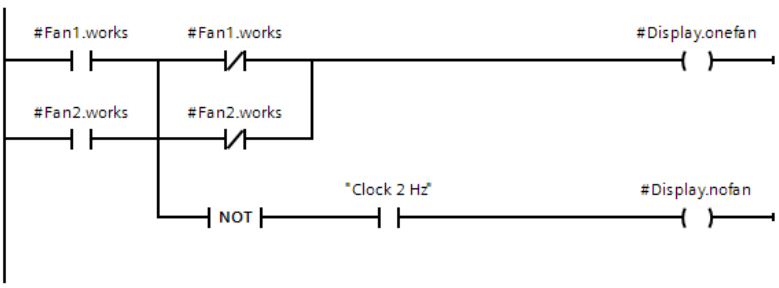


Fig. 1.5 Example of representation in ladder logic

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.

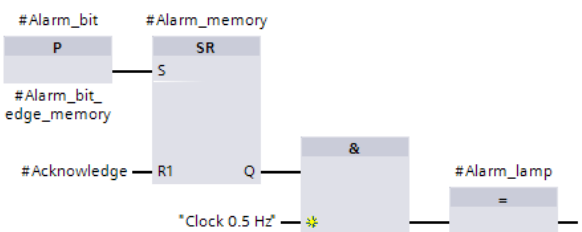


Fig. 1.6 Example of representation in function block diagram

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. 1.7).

```
1  //Motor memory
2        A     "Switch-on_manual"
3        A     "Manual_mode"
4        O
5        A     "Switch-on_automatic"
6        AN    "Manual_mode"
7        S     #Motor_memory        //Set memory
8
9        O     "Switch-off_manual"
10       O     "Switch-off_automatic"
11       ON    "Motor_fault"
12       R     #Motor_memory        //Reset memory
13
```

Fig. 1.7 Example of STL statements

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

Fig. 1.8 Example of SCL statements

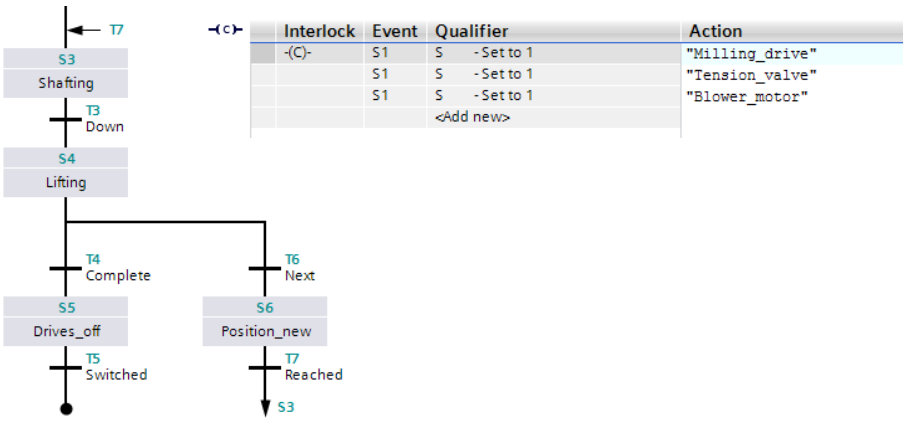


Fig. 1.9 Example of a GRAPH sequencer and step configuration

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. 1.8).

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 consistency of the hardware and parameterizes the modules. A startup program is then executed once, if present. The startup program belongs to the user program 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).

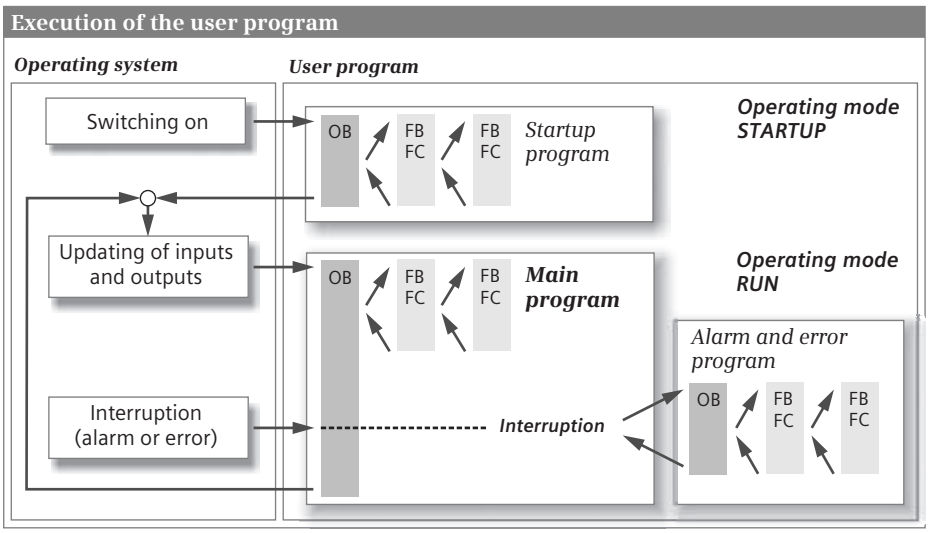


Fig. 1.10 Execution of the user program

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 instructions are available for calling function blocks and functions (start of execution). Each block call can be assigned inputs and outputs, referred to as “block parameters”. During calling, tags can be transferred with which the program in the block is to work. In this manner, a block can be repeatedly called with a certain function (e.g. addition of three tags), but with different parameters sets (e.g. for different calculations) (Fig. 1.11).

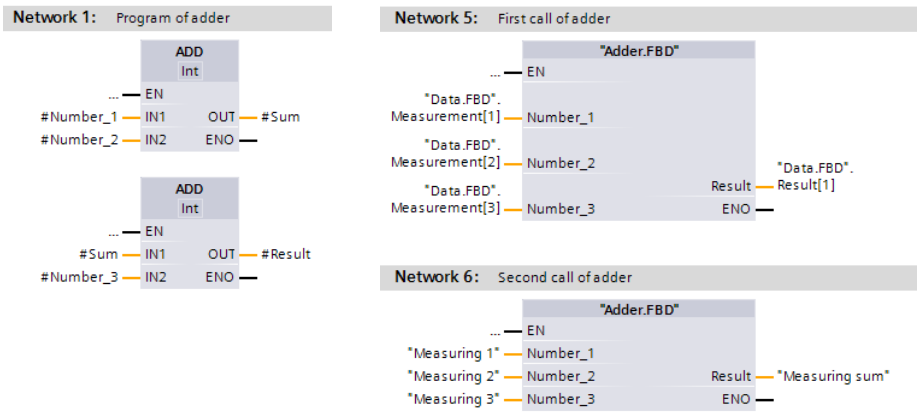


Fig. 1.11 Example of two block calls with different tags in each case

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.

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. Structuring is also possible (and recommended) in the main program. Once the main program has been processed, the control processor returns to the operating system, retains (for example) communication with the programming device, updates the input and output signals, and then recommences with execution of the main program.

Cyclic program execution is a feature of programmable 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.

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).

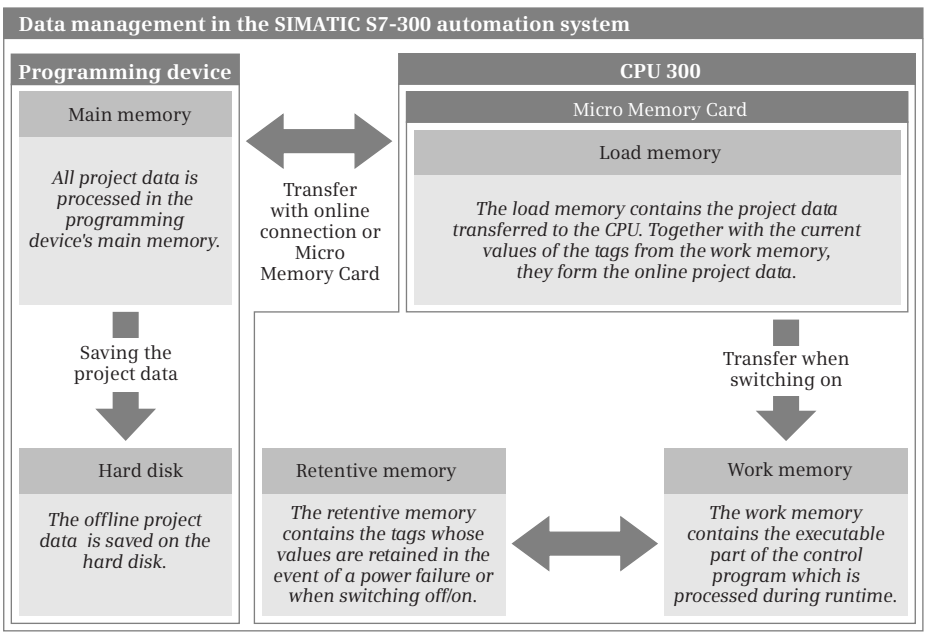


Fig. 1.12 Data management in the SIMATIC S7-300 automation system

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*.

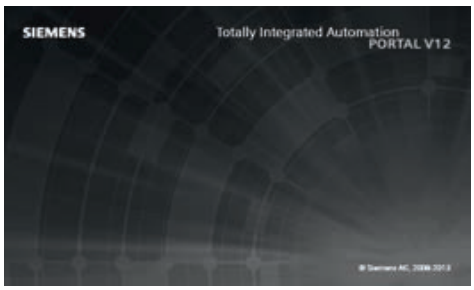
The user memory on the CPU is divided into three components: the *load memory* contains the complete user program including the configuration data, the *work*

memory contains the executable user program with the current control data, and the *retentive memory* contains the tags whose current values are saved powerfail-proof. The load memory is present on the Micro Memory Card, and therefore the CPU always requires a Micro Memory Card for operation.

The project data can be transferred between the programming device and CPU using the Micro 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. STEP 7 Professional requires approximately 2 GB on the hard disk.

An interface module with an appropriate port is required on the programming device for the online connection to the programmable controller. The connection can be established over MPI, PROFIBUS, or PROFINET (Ethernet). If you wish to work with a SIMATIC Micro Memory Card on the programming device, you require an SD 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, for example WinCC Professional 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 permits selection of the following portals (Fig. 1.13):

- 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.
- 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 messages.
- 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.

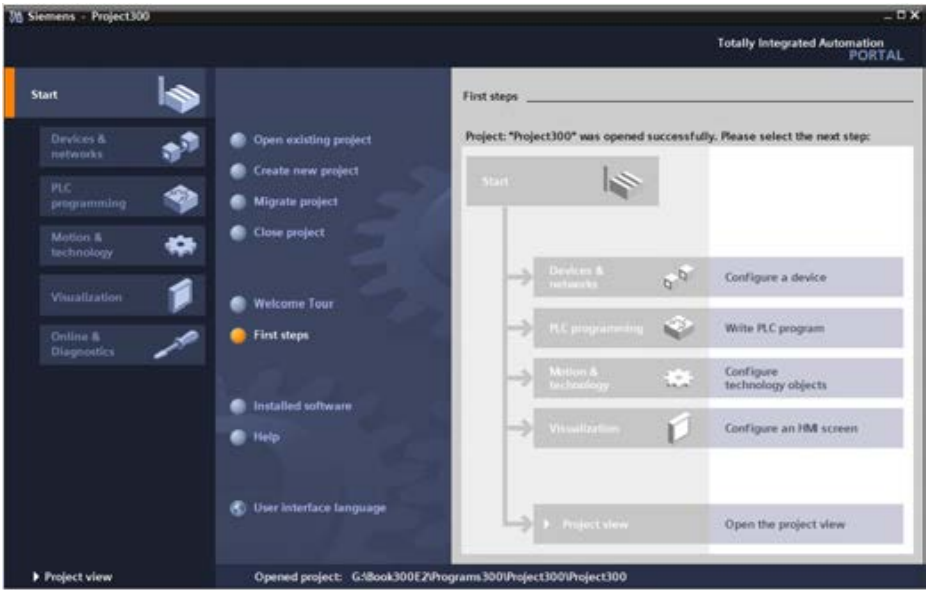


Fig. 1.13 Portal view: First steps after opening a project

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.

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 graphi-

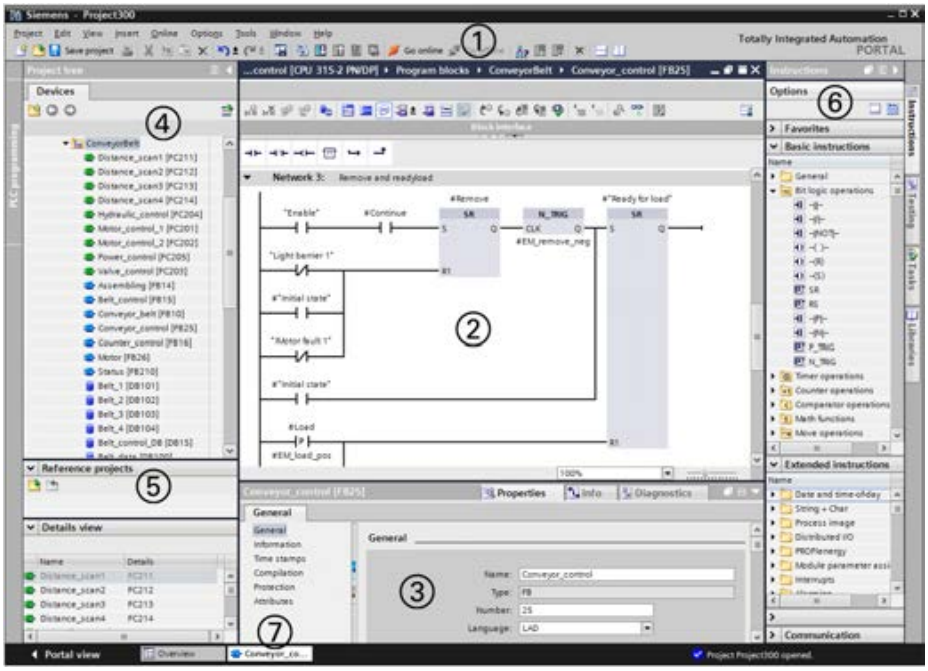


Fig. 1.14 Components of Project view using example of block programming

cally 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 programmable controller, generate the user program, or configure the process images for HMI devices.

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 and HMI stations included in the project, and further subfolders within these folders, e.g. for program blocks, PLC tags, and watch tables with a PLC station or, 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 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.

⑥ 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.

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. Clicking on a tab results in its contents being displayed in the top level of the working window. This makes it easy to change quickly between different window contents. The 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 processing steps: Configuration, parameterization, and networking of devices, structuring and programming of the user program, visualization of processes, and utilization of the online and diagnostics functions.

Readme provides general information on STEP 7 and further information which could not be included in the online help.

A comprehensive description of all available instructions, including extended instructions, can be found under *Programming a PLC* and *References*.

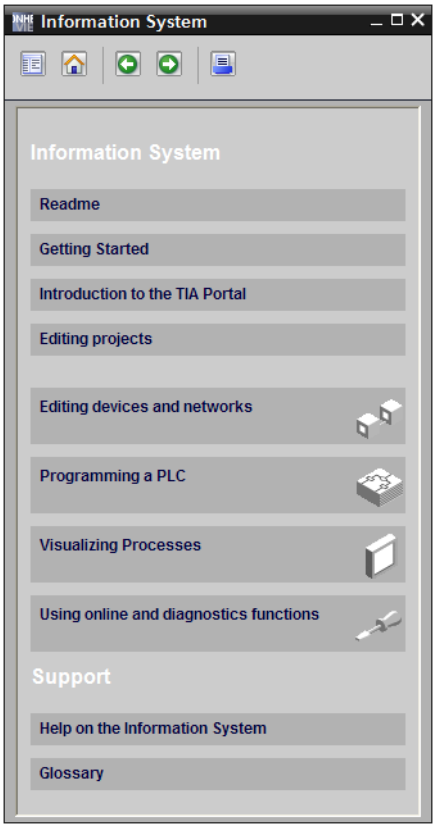


Fig. 1.15 Start page of the information system

1.2.7 Adapting the user interface

The language of the user interface can be changed. In the main menu, select *Options > Settings* and the *General* section. 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 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 station is described later in this book. *Common project data* includes, for example, centrally managed message texts or texts for multilingual projects.

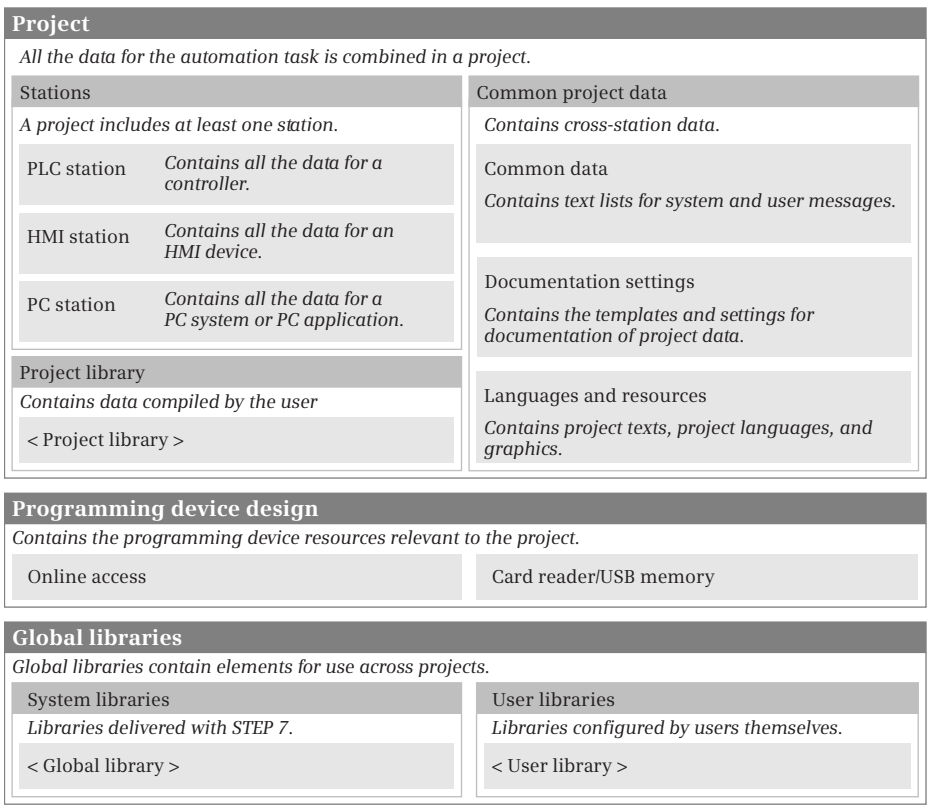


Fig. 1.16 Project components, libraries, and programming device design

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

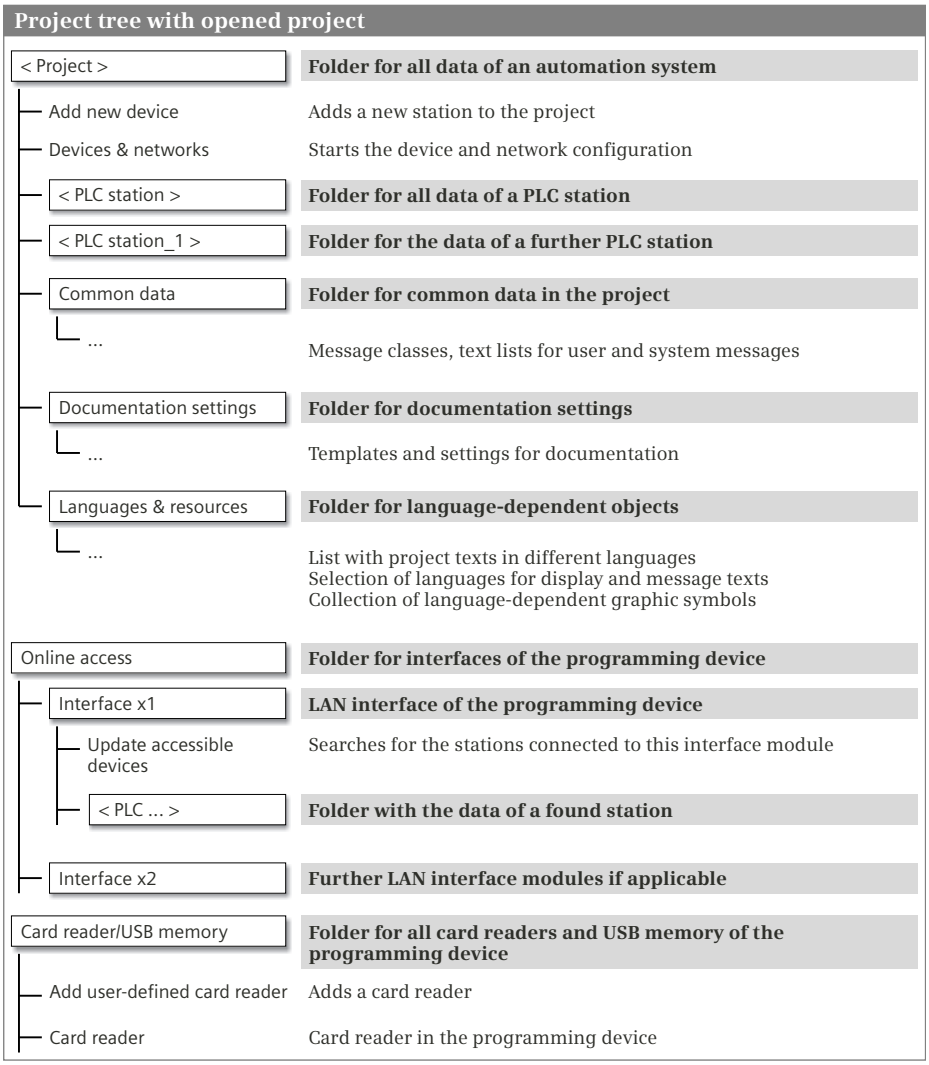


Fig. 1.17 Project structure in the project tree

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 S7-300 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 an “unspecified CPU 300” from the hardware catalog and replace it later with a “real” CPU 300 as required.

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.

The *Technology objects* folder contains the configuration data for control loop objects (PID controllers). A new PID controller technology object can be generated as a technology object using the *Add new object* editor.

The *External source 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 source 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 functions 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.

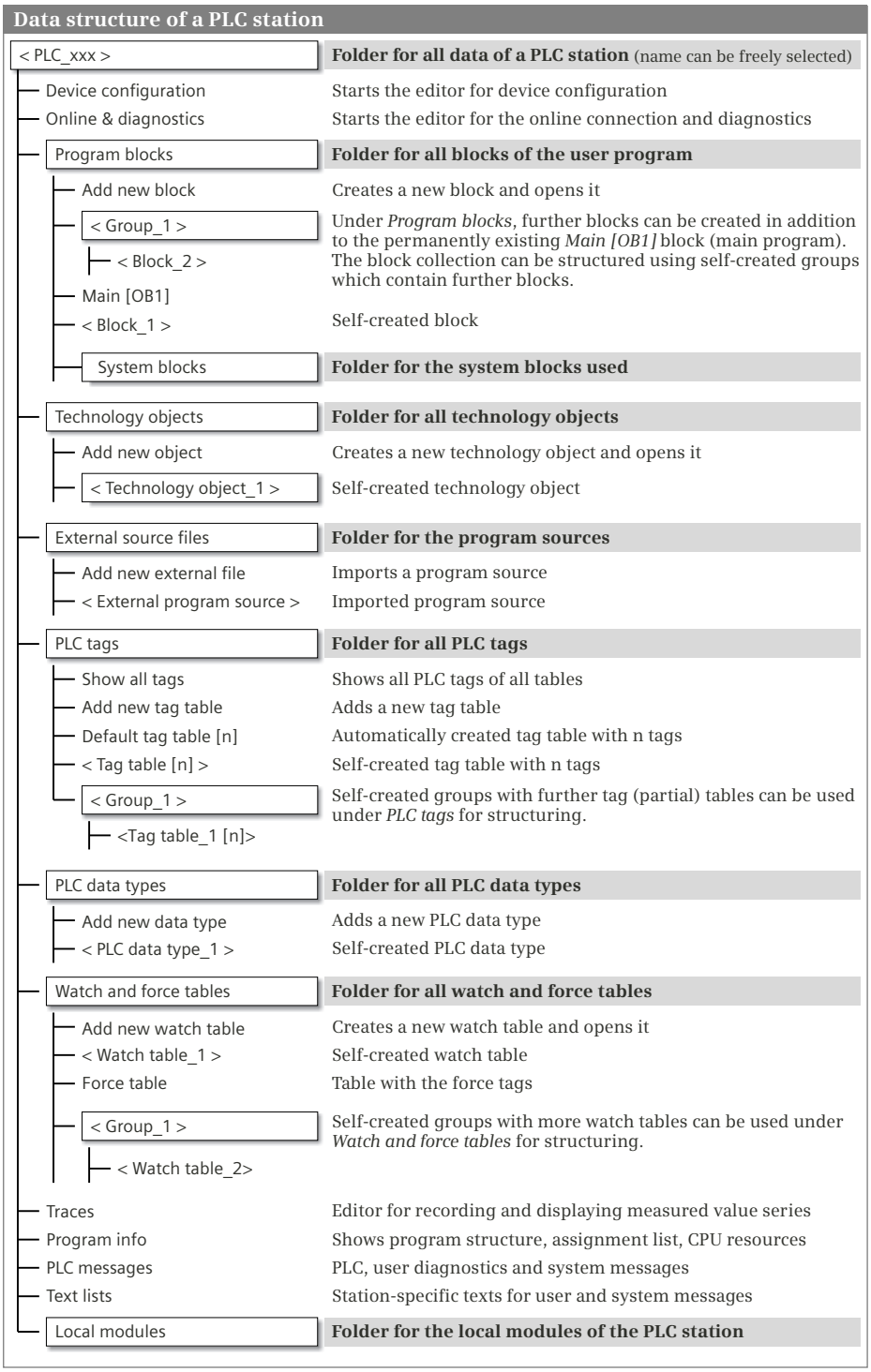


Fig. 1.18 Structure of the project data for a PLC station

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.

Program info provides information about

- the *call structure* shows the call sequence of the blocks – 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, work and retentive memories

Under *PLC alarms* you can find an overview of which PLC alarms, user diagnostic alarms, and system alarms are currently present.

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

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

You start configuration of a station using the *Device configuration* editor, which is located in the first position in the project structure 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 diagnostic 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 300 (a PLC station) into the project and open it for editing.
- Create a PLC program
  STEP 7 changes to the *PLC programming* portal in which you can edit the *Main* block (organization block OB 1) or insert a new block and open it for editing. Select a PLC station if applicable. This can also be an "unspecified CPU" which you can later replace by a CPU from the hardware catalog.
- Configure an HMI screen (if WinCC is installed in the TIA Portal)
  STEP 7 changes to the *Visualization* portal in which you can create a new HMI station or configure an already existing one. From this portal you start configuration of the process images, editing of HMI tags and messages, and the HMI simulator.
- Open the project view
  STEP 7 changes to the Project view in which you can carry out the next steps (insert and configure PLC station, insert and program block, or insert and configure 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 in 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 icon *Open or close the element view* in the *Details mode*, *List mode*, or *Overview mode*. The *Elements* pallet shows the contents of 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 S7-300 automation system

2.1 S7-300 station components

Fig. 2.1 S7-300 station with CPU 317-2PN/DP, power supply, and three signal modules

A programmable controller including all I/O modules is referred to as a “station”. An S7-300 station can contain the following components:

- Rack
- Power supply (PS)
- Central processing unit (CPU)
- Interface module (IM)
- Input/output module (signal module SM)
- Function module (FM)
- Communication module (communication processors CP)
- Special module such as the simulator module

A station can also comprise distributed I/O connected to it over a bus system.

Certain SIMATIC S7-300 modules are also available as SIPLUS version for particularly harsh environmental conditions.

Design variants

An automation system (station) can consist of several racks interconnected by means of bus cables. The power supply, CPU, and I/O modules (SM, FM, and CP) are fitted in the central rack. If the space in the central rack is insufficient for all I/O modules, or if you wish to position I/O modules remote from the central rack, you can use expansion racks which use interface modules to establish the connection to the central rack (Fig. 2.2). It is additionally possible to connect distributed I/O to a station.

With the S7-300, up to 8 I/O modules can be fitted in the central rack. A maximum current consumption of 1.2 A per rack must not be exceeded. The module width does not play any role here. If the space in this single-tier configuration is insufficient, you can select (with CPU 314 and higher)

- either a two-tier configuration (with IM 365 and a distance of up to 1 m between the racks)
- or a configuration with up to four tiers (with IM 360 and IM 361 and a distance of up to 10 m between the racks).

The connection between the I/O modules and the CPU is established via a serial backplane bus which is routed from module to module by means of connectors at the rear of the modules.

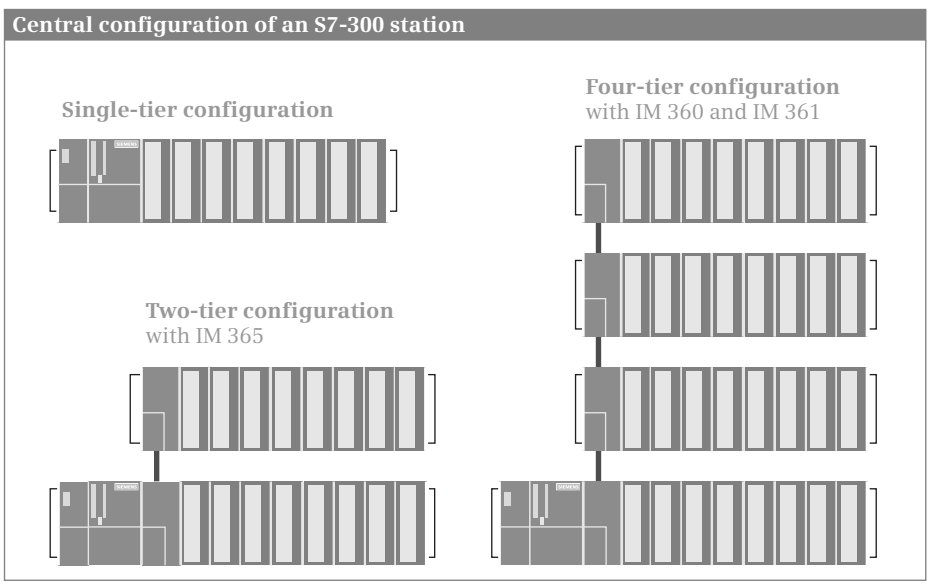


Fig. 2.2 Design variants of an S7-300 station

2.2 S7-300 CPUs

2.2.1 CPU versions

CPUs for S7-300 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. A Micro Memory Card (MMC) is required for operation.

Standard controllers

The controllers of standard design range from the “smallest” CPU 312 for lower-end applications with moderate processing speed requirements up to the CPU 319-3 PN/DP with its large program memory and high processing performance for cross-sector automation tasks. Equipped with the relevant interfaces, some CPUs can be used for central control of the distributed I/O via PROFIBUS and PROFINET.

Fig. 2.3 CPU 315-2DP

Compact controllers

The controllers of contact design can be used to set up compact mini PLCs. The CPU contains input and output channels so that the CPU is already sufficient as a station for very small applications.

The technological functions present in the compact controllers are system blocks integrated in the operating system which can be applied in the user program.

The digital and analog inputs/outputs present on the CPU have fixed assignments to the technological functions. Some of the assignments overlap, and therefore sometimes not all technological functions can be used together.

Fig. 2.4 CPU 314C-2DP

Fail-safe controllers

The fail-safe controllers are used in production plants with increased safety requirements. The relevant PROFIBUS and PROFINET interfaces allow the operation of safety-related distributed I/O using the PROFIsafe bus profile. Standard modules for normal applications can be used parallel to safety-related operation.

Fig. 2.5 CPU 315F-2 DP

Fail-safe automation systems control processes and machines with the objective of keeping the danger to persons and the environment as low as possible without limiting production more than absolutely necessary. Safety-related SIMATIC systems are used when the safe state can be achieved by switching off immediately. They comply with the following safety requirements: Safety integrity levels SIL1 to SIL3 in accordance with IEC 61508, and categories Cat. 1 to Cat. 4 in accordance with EN 954-1.

The enhanced safety functions are essentially achieved by means of a safety-related user program in an appropriately designed CPU 300F, by means of fail-safe I/O modules, and by the higher-level PROFIsafe profile on the PROFIBUS DP and PROFINET IO “standard” bus systems (Fig. 2.6).

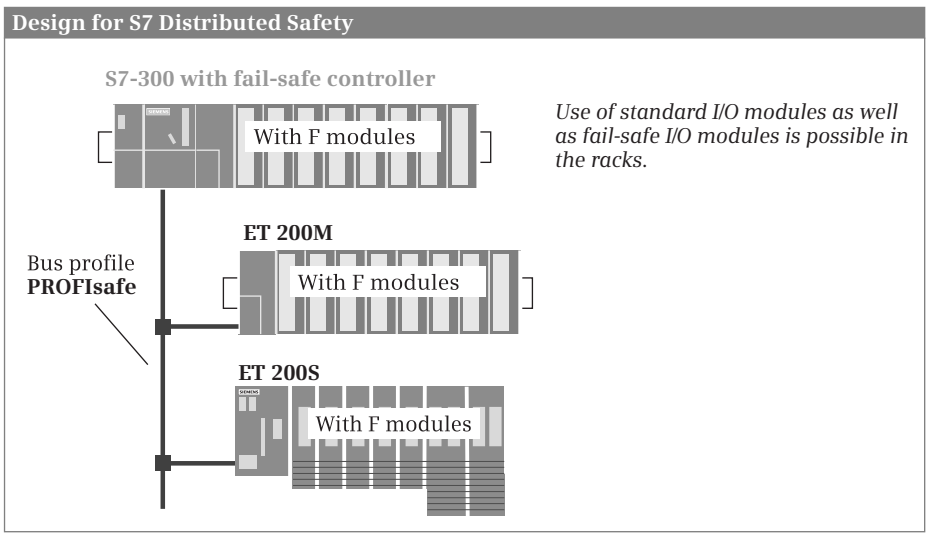


Fig. 2.6 Design versions for S7 Distributed Safety

2.2.2 Control and display elements

The mode switch and the status LEDs are located on the front side of the CPU (Fig. 2.7).

Fig. 2.7 Control and display elements on a CPU 317-2 PN/DP

Mode switch

The mode switch is designed as a toggle switch with the positions RUN, STOP, and MRES. The user program executes in the RUN position. 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 diagnostic 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.

Status LEDs

The status of the CPU is indicated by means of LEDs:

SF	red	lights up if there is a software error or hardware fault
BF	red	lights up if there is a bus fault; each bus interface has a fault LED, and the LEDs are numbered consecutively if more than one bus interface is present (BF1, BF2, etc.)
MAINT	yellow	lights up if there is a maintenance request
DC5V	green	lights up if the power supply is present on the CPU
FRCE	yellow	lights up if forcing is active, flashes at 2 Hz during the node flashing test
RUN	green	lights up in RUN mode, flashes at 0.5 Hz in HALT mode, flashes at 2 Hz in STARTUP mode
STOP	yellow	lights up in STOP mode, flashes at 0.5 Hz if the CPU requests a memory reset, flashes at 2 Hz when the CPU is carrying out a memory reset

If all LEDs flash, a CPU-internal system fault is present.

2.2.3 SIMATIC Micro Memory Card

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

Fig. 2.8 SIMATIC Micro Memory Card

The data is stored retentive on the MMC, 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 MMC, meaning that an MMC is always required to operate a CPU 300. The MMC 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 MMC by means of special system functions, for example read recipes from the MMC or create a measured value archive on the MMC and supply it with data.

The SIMATIC Micro Memory Card is available for memory capacities from 64 KB up to 8 MB. Please note that formatting the MMC using Windows tools makes it unusable for a CPU 300.

2.2.4 Memory areas in an S7-300 station

Fig. 2.9 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 and comments.

The online data consists of the user program and the system data which is located in three memory areas: the load memory on the Micro Memory Card, the retentive memory, and the work memory in the CPU. The system memory contains the global tags, the temporary local data, the buffers for diagnostics messages and communication jobs, and the stacks (buffers) for program execution.

Finally, the I/O modules contain memories for the signal states of the input and output signals.

Load memory

The load memory contains the complete user program including configuration data (system data). 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.

Access to the system data in the load memory is only possible using specific system functions, for example functions for changing the module parameters. Data blocks can be identified as “not relevant to execution”, and in this case they are not transferred to the work memory. (Non-cyclic) reading and writing of this data is never-

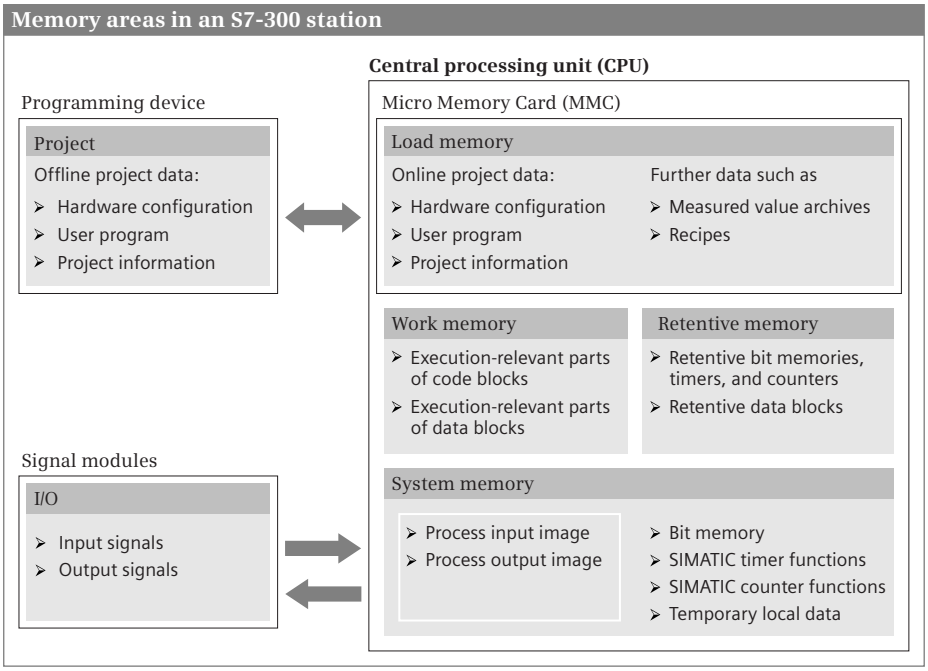


Fig. 2.9 Memory areas for the user program

theless possible, and therefore the data blocks can be used, for example, as recipe memories or measured value archives.

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.

Depending on the product, the work memory can be either a continuous area or divided into program and data memories, the latter of which can also be divided into retentive and non-retentive areas.

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

System memory

The system memory contains the operands (tags) which you address from your program. The operands are combined into (operand) areas which contain a specific quantity of operands for each CPU. Operands can be, for example, inputs with

which you query the signal states of pushbuttons and limit switches, and outputs with which you control contactors and lamps.

The following operand areas are present in the CPU's system memory:

- Inputs (I)
  These are an image (“process image”) of the digital input modules.
- Outputs (Q)
  These are an image (“process image”) of the digital output modules.
- Bit memories (M)
  These are information memories which can be addressed in the user program from any location.
- SIMATIC timer functions (T)
  These represent timers with which waiting and monitoring times are implemented.
- SIMATIC counter functions (C)
  These are software counters that can count up and down.
- Temporary local data (L)
  This serves as a dynamic buffer during block processing. The temporary local data is present in the local data stack, which is occupied in dynamic mode by the CPU during program execution.

The abbreviations with which you can address the various operands when programming are shown in parentheses. You can also assign a symbol to each tag and then work with the symbolic ID instead of the operand ID.

In addition, the system memory contains buffers for communication jobs and system alarms (diagnostic buffer).

2.2.5 Bus interfaces

The following bus interfaces can be present depending on the design of the CPU:

- MPI
  All CPUs 300 have an MPI connection (multi point interface) via which the programming device is connected. The MPI can also be used for data transfer (station-internal S7 basic communication) and for connecting operator control and display units. The MPI is always named X1.
- DP
  The DP interface connects the CPU to the PROFIBUS DP bus system. The CPU can be the DP master or also a DP slave. If a CPU has two DP interfaces, DP slave mode is not possible at both interfaces simultaneously. The station-external S7 basic communication can be used to configure data transfer via a DP interface. A programming device or an operator control and display unit can also be connected to a DP interface.

- MPI/DP
  Certain CPUs have a combined MPI/DP interface as the first connection. When delivered, the interface is configured as an MPI; if you wish to use it as a DP interface, you must change the parameter settings.

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

- PtP
  The PtP interface (point-to-point) connects an appropriately equipped compact CPU to another device over a point-to-point coupling. The following drivers are available: ASCII driver, 3964 (R) and RK 512 procedures.

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

The bus interfaces are numbered consecutively: X1 for the first interface (MPI or MPI/DP), then X2 for the second interface (DP or PN), and possibly a third interface X3 (PN) with the ports P1 and P2 (Fig. 2.10).

Fig. 2.10 The bus connections on a CPU 319-3 PN/DP

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 in the standard design are available for SIMATIC S7-300:

- SM 321 Digital input modules
- SM 322 Digital output modules
- SM 323, SM 327 Digital input/output modules
- SM 331 Analog input modules
- SM 332 Analog output modules
- SM 334, SM 335 Analog input/output modules

Each CPU 300 can control up to 8 modules per rack. The CPU 312 is designed for one (central) rack (max. 8 modules). With the other CPUs, up to three expansion racks (max. 24 modules) can be connected in addition to the central rack.

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. The process signals present with a DC voltage level of 24 V to 120 V or an AC voltage level of 120 V to 230 V are converted into signals with an internal level.

Fig. 2.11 SM 321 digital input module, DI 16

Depending on the module, the input channels are isolated either individually or in groups. The module types include simple input modules, modules with diagnostic capability, and modules for isochronous mode. All modules indicate the presence of a process signal by means of an LED on the input channel.

Digital input modules are available with one, two, or four bytes corresponding to 8, 16, or 32 input signals. A special feature is provided by the SM 321-1BP00 digital input module. Contrary to the address step of four bytes per module usual with the S7-300, this input module has eight bytes, in other words 64 input signals. A so-called default startup, i.e. without configuration, is therefore no longer possible when using this module. A special terminal block is required for the connection. The module is of single width, enabling a highly compact design.

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. The internal signals are amplified and output in the following current and voltage ranges (rated values):

- With electronic amplifiers from 24 V to 120 V DC and a current from 0.3 A to 2 A
- With electronic amplifiers from 120 V to 230 V AC and a current of 1 A or 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

Fig. 2.12 SM 322 digital output module, DO 32

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. All modules indicate a delivered process signal by means of an LED on the output channel.

The digital output modules have one, two or four bytes, corresponding to 8, 16 or 32 output signals. A special feature is provided by the SM 322-1BP00 (sourcing output) and SM 322-1BP50 (sinking output) digital output modules. Contrary to the address step of four bytes per module usual with the S7-300, these output modules have eight bytes, in other words 64 output signals.

A so-called default startup, i.e. without configuration, is therefore no longer possible when using these modules. A special terminal block is required for the connection. The module is of single width, enabling a highly compact design.

2.3.3 Digital input/output modules

The digital input/output modules are used by the CPU to record the operating states of the connected machine or plant and to control the machine or plant. These modules are signal conditioners for binary process input and output signals.

Process signals present with a DC voltage level of 24 V are converted into input signals with an internal level. The internal output signals are amplified electronically and output at 24 V with a current of up to 0.5 A.

Fig. 2.13 SM 323 digital input/output module, DI 8/DO 8

Depending on the module, the channels are isolated from each another. The outputs have integral short-circuit protection. All modules indicate a present or delivered process signal by means of an LED on the input or output channel.

The digital input/output modules have one or two bytes each corresponding to 8 or 16 input and output signals. A special feature is provided by the SM 327 digital input/output module. It has eight digital input channels and eight channels which can be parameterized individually as inputs or outputs.

Fig. 2.14 SM 331 analog input module, AI 8

2.3.4 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.

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 digital value is generated by integration, or with the SM 331-7FH01 by immediate conversion within 52 µs per channel. Depending on the module, the resolution is up to 16 bits including sign.

An analog value (an analog channel) occupies 16 bits, in other words two bytes. Analog input modules are available with 2, 4 or 8 channels, corresponding to an address range of 4, 8 or 16 bytes.

Depending on the module, the input channels are isolated either individually or in groups. The module types include simple input modules, modules with diagnostic capability, modules for isochronous mode, and modules which output a hardware interrupt if a limit is violated. All analog input modules indicate a missing load voltage or a fault on the module by means of the SF LED.

Fig. 2.15 SM 332 analog output module, AO 4

2.3.5 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.

The modules can output a voltage value (0 to 10 V, 1 to 5 V, or -10 to +10 V) or a current value (0 to 20 mA, -20 to +20 mA, or 4 to 20 mA). Depending on the module, the resolution is up to 16 bits including sign.

An analog value (an analog channel) occupies 16 bits, in other words two bytes. Analog output modules are available with 2, 4, or 8 channels corresponding to an address range of 4, 8, or 16 bytes.

The output channels are isolated from the load voltage and the backplane bus. All modules can output a diagnostic interrupt; SM 332-7ND02 supports isochronous mode. All analog output modules indicate a missing load voltage or a fault on the module by means of the SF LED.

2.3.6 Analog input/output modules

The CPU can use analog input/output modules to process analog variables and continuously provide actuators with analog setpoints. These modules are signal conditioners for analog process signals.

A voltage value (0 to 10 V), a current value (0 to 20 mA), or – with SM 334-0KE00 – also a Pt 100 thermoresistor can be connected to an input channel of the SM 334 modules. An output channel can output a voltage value of 0 to 10 V, with SM 334-0CE01 also a current of 0 to 20 mA. The resolution is 8 or 12 bits including sign. In the SM 334-0KE00, the output channels are isolated from the backplane bus.

Fig. 2.16 SM 334 analog input/output module, AI 4, AO 2

SM 335

SM 335 is a fast analog input/output module which converts an analog input signal into a digital value within 200 µs, and outputs a digital value as an analog value within 800 µs. The voltage input ranges are ±1 V, ±10 V, ±2.5 V, 0...2 V, and 0...10 V, the current input ranges ±10 mA, 0...20 mA, and 4...20 mA. The module can output ±10 V or 0...10 V on an output channel.

The resolution is 14 bits for an input channel and 12 bits for an output channel, in both cases including a sign where applicable.

An analog value (an analog channel) occupies 16 bits, in other words two bytes. The module has four input channels and four output channels, and therefore occupies eight bytes for input addresses and eight bytes for output addresses. A pulse input channel (counter input) is also available.

The SF LED indicates a fault on the module. The module can output a diagnostic interrupt.

2.4 Function modules

Function modules (FM) 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 at the CPU's internal interface. They are responsible for handling functions which the CPU cannot usually execute quickly enough, such as counting pulses, positioning, or controlling drives.

Fig. 2.17 FM 350 fast counter

The following function modules are available:

- FM 350-1AH03 Fast counter

  Count up/down at a frequency up to 500 kHz, counting range ±31 bits, 1 counter channel
- FM 350-2AH01 Fast counter

  Count up/down at a frequency up to 20 kHz, counting range ±31 bits, 8 counter channels
- FM 351-1AAH01 Positioning module

  Positioning for rapid-feed and creep-feed drives, adjustment of 2 or 3 independent axes with 4 digital outputs per axis for controlling contactors or frequency converters. Position detection is carried out either incrementally or synchronous-serially.
- FM 352-1AH01 Electronic cam controller

  32 cam tracks with 32/64/128 cams and parameterizable cam characteristics (e.g. position-based or time-based cams) control 13 or 16 digital outputs. Position detection is carried out either incrementally or synchronous-serially.

Additional function modules in the design for SIMATIC S7-300 are:

- SM 338-4BC01 POS input module

  Connection of maximum 3 absolute value encoders (SSI), with 2 additional digital inputs for self-clocking encoder value acquisition or encoder value acquisition isochronous to the PROFIBUS DP cycle.
- 7MH4 900-2AA01 SIWAREX FTA

  Weighing modules for non-automatic and automatic weighing operation, e.g. suitable for the manufacture of mixtures, filling, loading, monitoring, and sacking, also for calibratable systems.
- 7MH4 900-3AA01 SIWAREX FTC

  Weighing module for use with belt scales, differential dosing scales, and solids flow meters, can be used also for weight recording and force measurement.
- 7MH4 950-1AA01 SIWAREX U, single-channel
  7MH4 950-2AA01 SIWAREX U, two-channel

  Versatile weighing module for the connection of one or two scales, for all simple weighing and force measurement tasks.

2.5 Communication modules

The communication modules (or communication processors, CP) 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 communications services for the CPU and the user program.

Fig. 2.18 PtP connection CP 340 RS 232C

The following communication modules are available:

- CP 340 Communication module for point-to-point coupling

  One interface; with the physical transmission characteristics RS 232C (V.24), 20 mA (TTY) or RS 422/RS 485 (X.27) depending on the type of module; ASCII driver, 3964 (R) (not for RS 485) and printer driver as the implemented protocols

- CP 341 Communication module for point-to-point coupling

  One interface; with the physical transmission characteristics RS 232C (V.24), 20 mA (TTY) or RS 422/RS 485 (X.27) depending on the type of module; ASCII driver, 3964 (R) and RK 512 as the implemented protocols with the option for subsequent downloading of customer-specific protocols

- CP 343-2 AS-Interface master
  CP 343-2P AS-Interface master with configuration support

  Up to 62 AS-i slaves can be connected; supports all AS-i master functions in accordance with AS-i specification V3.0; status and fault displays by means of LEDs on the front panel; CP 343-2P: Supports configuration of the AS-i subnet with STEP 7

- CP 342-5 Connection to PROFIBUS DP with electrical interface
  CP 342-5FO Connection to PROFIBUS DP with optical interface

  Used as DP master or DP slave; communications services: PROFIBUS DP-V0, PG/OP communication, S7 communication (client, server), open communication (SEND/RECEIVE)

- CP 343-5 Connection to PROFIBUS FMS

  Communication services: PROFIBUS FMS, PG/OP communication, S7 communication, open communication (SEND/RECEIVE)

- CP 343-1 Lean Connection to Industrial Ethernet

  One interface with two ports and integral switch; with autosensing and autocrossover functions; communication services: PROFINET IO device, PG/OP communication, S7 communication (server), open communication (TCP/IP and UDP);

IT communication (Web function); diagnostics option in STEP 7 and via Web browser

- CP 343-1 Connection to Industrial Ethernet

  One interface with two ports and integral switch; with autosensing/autonegotiation and autocrossover functions; communication services: PROFINET IO controller or PROFINET IO device, PG/OP communication, S7 communication (client, server, multiplexing), open communication (ISO, TCP/IP and UDP); IT communication (Web function); diagnostics option in STEP 7 and via Web browser; IP address assignment via DHCP or the user program (e.g. with HMI)

- CP 343-1 Advanced Connection to Industrial Ethernet

  Two separate interfaces: Gigabit interface with one port, with autosensing function, PROFINET interface with two ports and integral switch, with autosensing and autocrossover functions

  Communication services via both interfaces: PG/OP communication, S7 communication (client, server, multiplexing), open communication (TCP/IP and UDP); IT communication (HTTP communication, e-mail client, FTP communication, access to data blocks via FTP server)

  Communication services via PROFINET interface: PROFINET IO controller or PROFINET IO device with real-time characteristics, PROFINET CBA; IP address assignment via DHCP or the user program (e.g. with HMI)

2.6 Other modules

2.6.1 Interface modules (IM)

In the case of a multi-tier configuration, the interface modules (IM) connect the central rack to an expansion rack or the expansion racks to each other.

The IM 365 interface module can be used to connect an expansion rack to a central rack. One IM 365 each is present in both racks, and these are connected together by a cable which is 1 meter long. The use of modules is limited in one of the expansion racks connected to the IM 365, e.g. no FM or CP modules are permissible.

Fig. 2.19 IM 360 and IM 361 interface modules

Up to three expansion racks can be connected to the IM 360 interface module in the central rack, and an IM 361 interface module is present in each of these expansion racks. The connection cables can be 1 m, 2.5 m, 5 m, or 10 m long.

2.6.2 Power supply modules (PS)

The power supply modules (PS) provide a 24 V DC for powering the CPU and as load voltage. The secondary voltage is regulated, short-circuit-proof, and open-circuit-proof.

Depending on the power supply module, the primary voltage is either an AC voltage of 120/230 V (PS 307) or a DC voltage in the range from 24 to 110 V (PS 305).

Power supply modules are also available which deliver a secondary current of 2 A, 5 A or 10 A.

An LED on the front indicates that the 24 V output voltage is present. The 24 V output voltage can be switched on and off.

Fig. 2.20
PS 307 power supply

2.6.3 Simulator module

The SM 374 simulator module is used to test the user program. It has 16 binary channels which can be connected as input or output channels. A mixed setting with eight input channels and eight output channels is also possible.

Switches are present on the module with which the external signals can be simulated. The status of the controlled outputs is displayed on LEDs.

Note on configuration: The simulator module is not included in the hardware catalog of STEP 7. In order to configure the simulator module, you use a substitute module with the same address characteristics depending on the I/O setting: e.g. a 16 channel digital input module if the simulator module is set to 16 inputs, or a mixed 8 DI/8 DO module if eight inputs and eight outputs are set.

Fig. 2.21
SM 374 simulator

2.6.4 Dummy module

The DM 370 dummy module reserves a slot for an unconfigured interface module or signal module. It is used if an interface module or signal module is to be added to the hardware configuration at a later point in time without having to change the configuration. A switch on the dummy module determines whether it is configured (position “A”) or only serves as a “spacer” (position “NA”).

If the dummy module reserves the slot for an interface module, configuration in STEP 7 is unnecessary.

If the dummy module reserves the slot for a single-width signal module, the occupied addresses envisaged for the subsequently used signal module must be configured.

Fig. 2.22
DM 370 dummy module

Two dummy modules are used if the space is to be kept free for a double-width signal module. The addresses envisaged for the subsequent signal module must be configured for the first module in the slot with the lower number. The second dummy module bridges the (mechanical) distance to the next inserted module, and also connects the backplane bus further. This module is not configured.

2.7 SIPLUS S7-300

SIPLUS extreme is the product range with hardened components for use in harsh environments. Many components of the standard SIMATIC S7-300 range are offered in the SIPLUS S7-300 range in a form adapted to extreme ambient conditions.

Fig. 2.23
SIPLUS CPU 317-2PN/DP

The standard products are adapted individually. Two refined SIPLUS versions are available for SIMATIC S7-300 for:

- Extended ambient temperature range (−25 ... +60 °C, partly +70 °C), extraordinary medial load (conformal coating) and electronic equipment on rail vehicles (compliance with EN 50155)
- Extraordinary medial load (conformal coating).

Several standard controllers, compact controllers, and fail-safe controllers from the SIMATIC S7-300 range are available for an extended ambient temperature range, and also a large range of digital and analog signal modules, communication modules, the IM 365 interface module, the FM 350-1 function module, the SIPLUS DCF77 radio clock module, and power supply modules.

The following are available for an extraordinary medial load: Digital and analog signal modules, communication modules, and the FM 350-2 function module.

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

Configuration of SIPLUS modules

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

Since the SIPLUS modules have the same functions as existing modules, you can use the corresponding equivalent type (the standard module) when configuring.

This equivalent type can be found on the device's nameplate, in the SIPLUS data sheets, and on the Internet in the Siemens A&D Mall area.

If you select a module in the hardware catalog of STEP 7, the information on the module also shows whether this module is available as a SIPLUS type.

3 Device configuration

3.1 Introduction

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

This chapter primarily describes the configuration of an individual PLC station with a CPU 300 standard controller and provides an overview of the networking options. Configuration of the distributed I/O is described in Chapters 16.4 “PROFIBUS DP” on page 628 and 16.3 “PROFINET IO” on page 613.

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 by double-clicking on the *Devices & networks* editor under the project or on the *Device configuration* editor under the PLC station.

The *Device view* tab shows the station (device) racks. You add a station to the project in this view and configure it. The *Network view* tab shows the networking between multiple stations. In this view you can add further subnets and stations, and configure their networking (described in Chapter 3.4 “Configuring the network” on page 76). In the *Topology view* tab you configure the geographic arrangement of the Ethernet network (described in Chapter 16.3.6 “Real-time communication in PROFINET” on page 624).

Working in the Device view

The device configuration shows one or more racks with the modules which have already been positioned. If several PLCs are present in the project, select the one you wish to edit in the toolbar of the working window.

You can now drag new modules with the mouse from the hardware catalog to a rack, or remove existing ones. In the inspector window you can set the properties of the selected module such as the interrupts of the CPU or the addresses of the input/output modules.

Working in the Network view

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

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

Working in the Topology view

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

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

Save, compile, and download

You save the entered data on the hard disk by saving the complete project (using the *Project > Save* command in the main menu). In order to download the configuration data to a CPU, 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), 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.

Working area of the device configuration

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

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

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

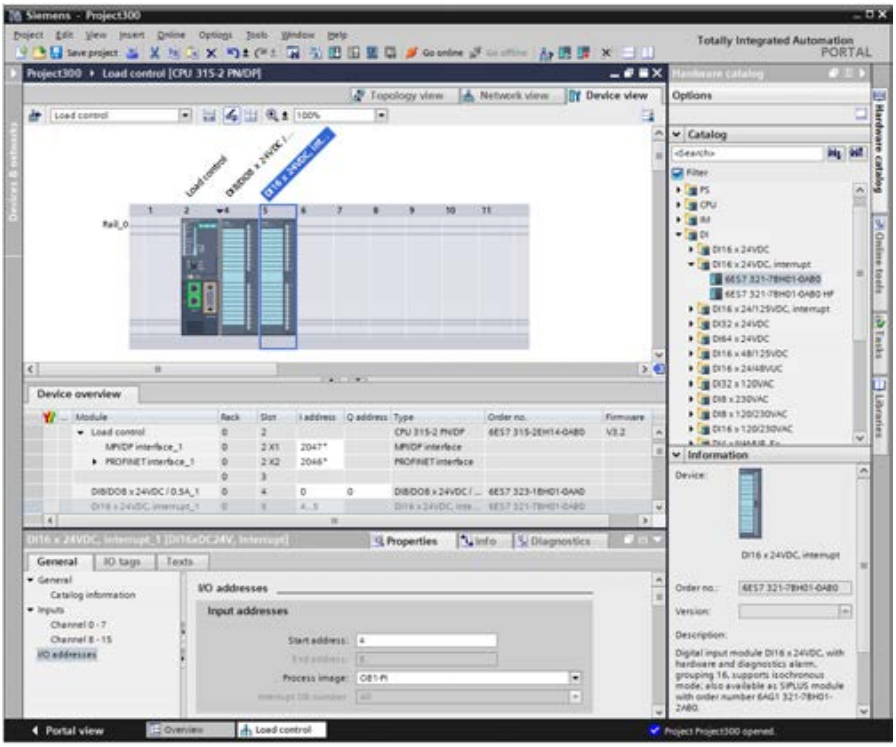


Fig. 3.1 Example of working area of device configuration (Device view)

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.

3.2 Configuring a station

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

3.2.1 Adding a PLC station

When creating a new project, you normally add a PLC station at the same time. You can add further PLC 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 2. Slot 1 on the left of the CPU is envisaged for the power supply module, slot 3 on the right of the CPU is for an interface module for connection of a further rack. I/O modules can be fitted in slots 4 to 11.

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

The I/O modules must be inserted without gaps, starting at slot 4. Each module occupies one slot independent of its width.

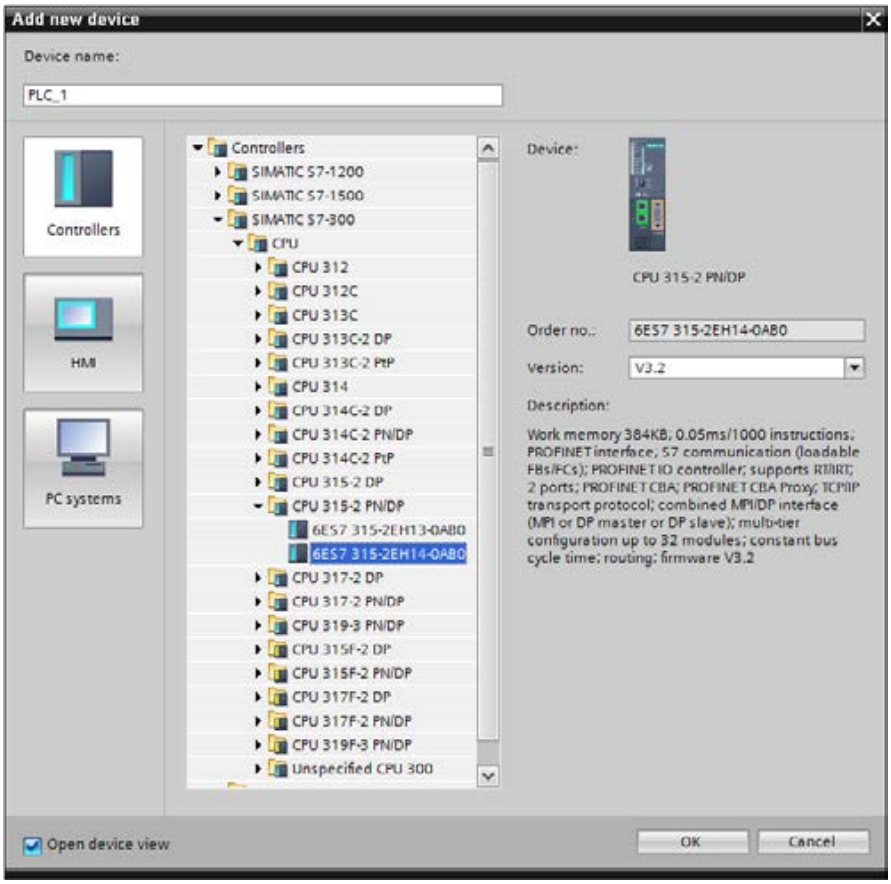


Fig. 3.2 Selection window *Add new device*

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

3.2.3 Adding an expansion rack

In the hardware catalog, double-click on the mounting rail under *Rack*, or drag it with the mouse into the working window. You are provided with an empty rack with slots 1 to 11. The connection from this expansion rack to the central rack is made by interface modules.

You use the IM 365 interface module if the PLC station has only one expansion rack, if this is not further than 1 m from the central rack, and if only signal modules are inserted in the expansion rack. Double-clicking on the IM 365 in the hardware catalog under *Interface module* positions it in slot 3 in the rack, an additional double-click positions the module in the other rack and connects the two interface modules.

If there is more than one additional rack, select the IM 360 as the interface module in the central rack (rack 0) and the IM 361 for the expansion rack.

Position the I/O modules in the expansion rack in slots 4 to 11 without gaps. In slot 1 you can position a power supply module for the load voltage for the modules present in the rack; slot 2 remains vacant.

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, 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. You can read out this data online using a programming device or evaluate it in a program using the system function RDSYSST (reading of system status list with SSL_ID W#16#011C and the index W#16#0003 for the higher level designation and W#16#000B for the location designation).

In the **MPI** section you set the connection to an MPI subnet, the node address, the diagnostic address, and the network parameters. For more information, refer to Chapter 3.4.5 “Configuring an MPI subnet” on page 82.

In the **DP interface** section you set the connection to a PROFIBUS subnet, the node address, the diagnostic address, the operating mode (master/slave), and further properties such as sync/freeze. You define 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 3.4.6 “Configuring a PROFIBUS subnet” on page 83.

With a combined **MPI/DP interface**, first select the interface type (MPI or DP) and then set the corresponding properties.

In the **PROFINET interface** section, you set the connection to an Ethernet subnet and define the IP address, the subnet mask, and the diagnostic address of the interface. Under *Port* you can find the transmission settings and the diagnostic address of the port (connection). For more information on the format of the IP address, refer to Chapter 3.4.7 “Configuring a PROFINET subnet” on page 85.

You can set the startup characteristics of the CPU under **Startup** (Fig. 3.3). With the *Startup if preset configuration does not match actual configuration* checkbox activated, the CPU starts even if the actual hardware configuration deviates from the set configuration. “Warm restart” is the fixed setting for *Startup after POWER ON* with S7-300 controllers. The duration for module parameterization is monitored during a startup; you can set the monitoring times. A module is considered to be absent if the monitoring time for it expires.

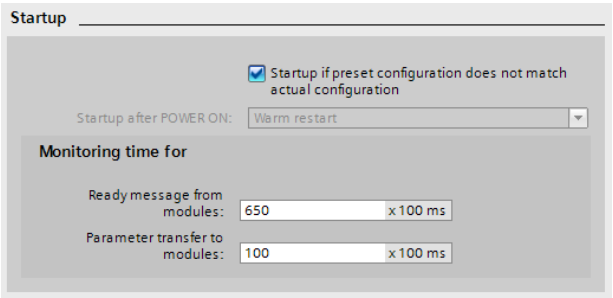


Fig. 3.3 Startup parameters with a CPU 300

In the **Cycle** section, you set the *Cycle monitoring time*, which is then signaled when exceeded and can lead to the STOP operating state. In the *Cycle load due to communication* section, you define the time share for communication. In addition to execution of the user program, the CPU also carries out communication tasks, for example data transmission to another PLC station or downloading of blocks from a programming device. This communication requires time, some of which has to be added to the execution time of the main program. Specification of the communication load can be used to control influencing of the cycle time to a certain extent. The time available for communication is entered as a percentage with this parameter (communication load). The cycle time is then extended by the factor 100 / (100 – communication load).

In the Cycle section you can also set the size of the process image for the inputs and outputs – if the CPU supports this function – and define the response of the organization block OB 85 in the event of an I/O access error during automatic updating of the process image.

Clock memory change their signal state controlled by the system at frequencies from 0.5 Hz to 10 Hz. During parameterization of the CPU you activate the clock memories and assign an address to them. Further information on bit memories in general and on clock memories can be found in Chapter 4.1.3 “Operand area: bit memory” on page 93.

In the **Interrupts** section you activate and parameterize event-driven processing. Further information on interrupts can be found in Chapter 5.6 “Interrupt processing” on page 186.

Under **Diagnostics system** you can use the *Signal cause of STOP* checkbox to activate output of a message to the logged-on display units when the CPU changes to the STOP state.

Under **System diagnostics** you can activate the system-internal diagnostics functions (see Chapter 5.8.5 “System diagnostics with Report System Errors” on page 216). Under *Advanced settings* you can define which organization blocks are to be created and with which events the CPU is to switch to the STOP operating state.

In the **Time of day** section you can set the correction factor and the type of synchronization.

In the **Web server** section, you can activate the web server and set its properties. Further details can be found in Chapter 18.4 “Web server” on page 707.

Under **Retentive memory** you can set which areas of the bit memories and the SIMATIC timer and counter functions are to be retentive.

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

In the **Connection resources** section you can assign the available number of connections to the individual communication types. One connection should always be reserved for PG communication so that the programming device can access the CPU. The maximum number of connections available for the selected CPU is shown in the last line.

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

3.3.2 Addressing modules

Slot address, geographic address

Every slot in a PLC station has a fixed address. This slot address is made up of the rack number and the slot number. A module is unequivocally defined by the slot address (“geographic address”).

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 DP master system or PROFINET IO 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.

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 logical addresses commences at zero and ends at a CPU-specific upper limit.

In the hardware configuration, a logical address is assigned to each byte of a used module. As standard, STEP 7 assigns the addresses starting at zero, but you can change the proposed address.

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 DP slave or an intelligent IO device.

In the case of modules which have input and output areas, the lower area start address is defined as the module start address. If the input and output areas have the same start address, use the input address.

The module start address is used in many 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. 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 addresses (Fig. 3.4).

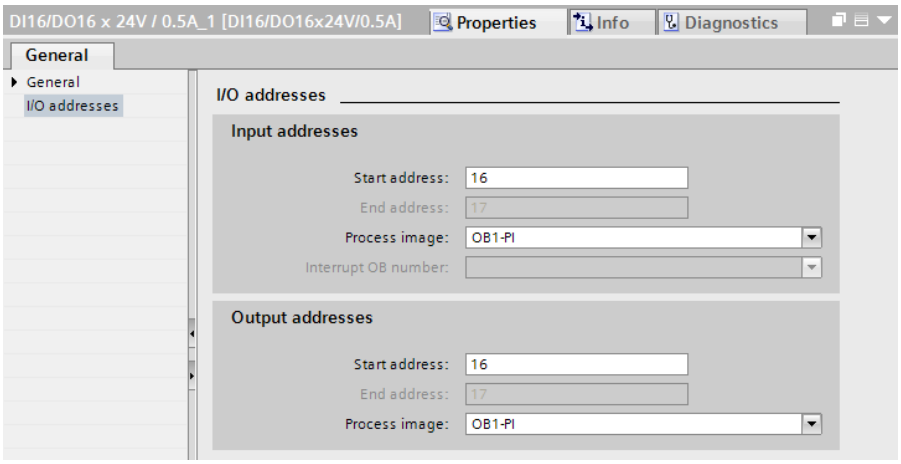


Fig. 3.4 Example of parameterization of I/O addresses

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.

The digital modules are usually located in the process image with regard to their addresses, meaning that their signal states are updated automatically and that they can be addressed with the input (I) and output (Q) operand areas. Analog modules, FM modules, and CP modules usually have an address which is not in the process image. You must access the user data of these modules via the peripheral operand area (I:P or Q:P). However, you are always able to freely select the module address within the CPU's address volume.

If the CPU supports isochronous mode of the distributed I/O, you can assign the process image partition PIP 1 instead of the OB1 process image to an involved module.

Diagnostic address

Appropriately configured modules can deliver diagnostic data which you can evaluate in the user program. If central modules have a user data address (module start address), you access the module by means of this address when reading the diagnostic data. If a module or submodule does not have a user data address, for example an interface module, a diagnostic address is available for this purpose.

The diagnostic address is always an address in the I/O input range and takes up one byte. The user data length of this address is zero; if it is located in the process image (which is certainly permissible), it is not taken into consideration by the CPU when updating the process image.

During hardware configuration, STEP 7 automatically assigns the diagnostic addresses in descending order starting with the highest possible logical address. You can change the diagnostic address with the hardware configuration.

The diagnostic data can only be read using special system functions; access to the diagnostic address using load statements has no effect.

3.3.3 Assigning parameters to signal modules

You set the following parameters in addition to the module start address for appropriately designed signal modules.

Digital input modules: In the case of an interrupt-triggering input module, enable the diagnostics interrupt in the *Inputs* section and define the trigger event: absence of the sensor power supply and/or with open-circuit. Enable the diagnostics and/or hardware interrupt for a NAMUR module, and set the input delay and the type of voltage. You can select for the channels which event is to trigger the diagnostics and hardware interrupts.

Digital output modules: In the case of an interrupt-triggering output module, enable the diagnostics interrupt in the *Outputs* section and define the trigger event: open-circuit, absence of load voltage, short-circuit to ground, or discrepancy test

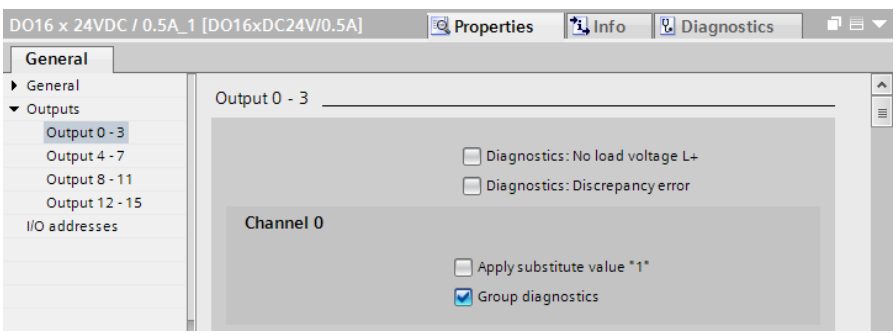


Fig. 3.5 Example of parameterization of a digital output channel

(comparison of reference and actual state of an output signal). You can define the response in the STOP operating state: retain the last value or connect a substitute value (Fig. 3.5).

Analog input modules: Use *Enable* to enable the triggering of a diagnostics interrupt and/or a hardware interrupt. For example, you can set the measuring type, measuring range, interference frequency suppression, and smoothing for the channels (Fig. 3.6).

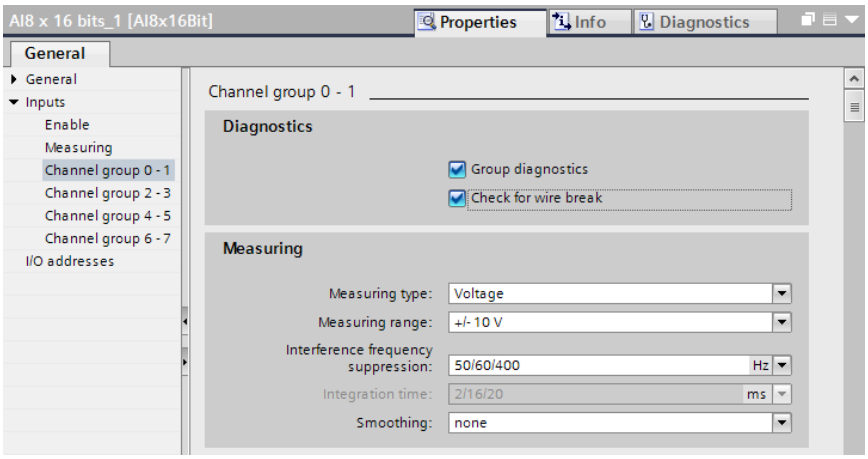


Fig. 3.6 Example of parameterization of an analog input channel

Analog output modules: Use *Enable* to enable the triggering of a diagnostics interrupt and/or a hardware interrupt. For example, you can set the output type (voltage or current), output range, and response of the output channel in STOP mode (retain last value or connect substitute value) for the channels.

3.4 Configuring the 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. Configuration 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. 3.7).

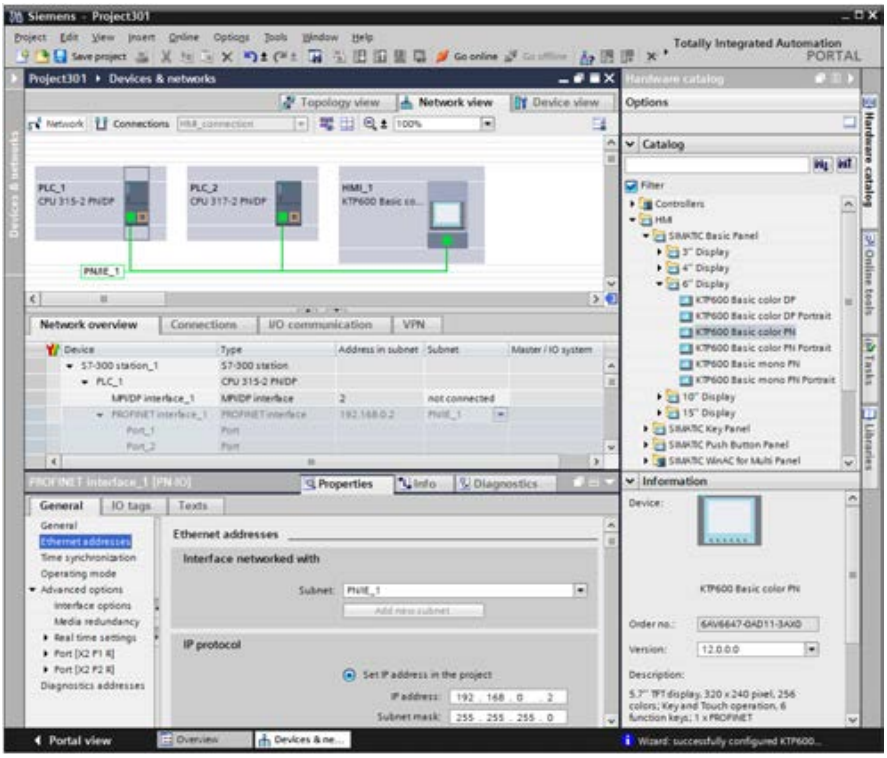


Fig. 3.7 Example of working area of network configuration (Network view)

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

3.4.2 Networking stations

"Networking" of stations corresponds to the wiring of modules with communication capability, i.e. a *mechanical* connection is established. A *logical* connection is

additionally required in order to transfer data via the cable. The logical connection defines the transmission parameters between the modules.

The working window of the network configuration editor shows the existing stations with the modules with communication capability. The interfaces for the MPI, PROFIBUS, PROFINET, and AS-Interface subnets are highlighted. Any existing CP modules are located on the right next to the CPUs, even if they are inserted in an extension rack.

Adding a station in the network configuration

In the hardware catalog under *Controllers > SIMATIC S7-300 > CPU > [folder: CPU 3xx...] > [CPU]*, select the desired CPU and drag it with the mouse into the working area. The graphic shows the CPU as a representative for the complete PLC station with the existing bus interfaces.

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

Adding a communication module in the network configuration

In the hardware catalog under *Controllers > SIMATIC S7-300 > Communications 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 CP module added in this manner is positioned by the editor in the lowest vacant slot in the rack.

If you drag the CP module to an existing subnet and if the CP module has an interface matching the subnet, the interface is directly connected to the subnet when adding and the CP module is displayed individually as a graphic. In the Device view, the CP 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.

Before networking a combined MPI/DP interface, set MPI or DP as the interface type in the interface properties.

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

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

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. 3.8).

Setting node addresses

When networking a module, the 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.

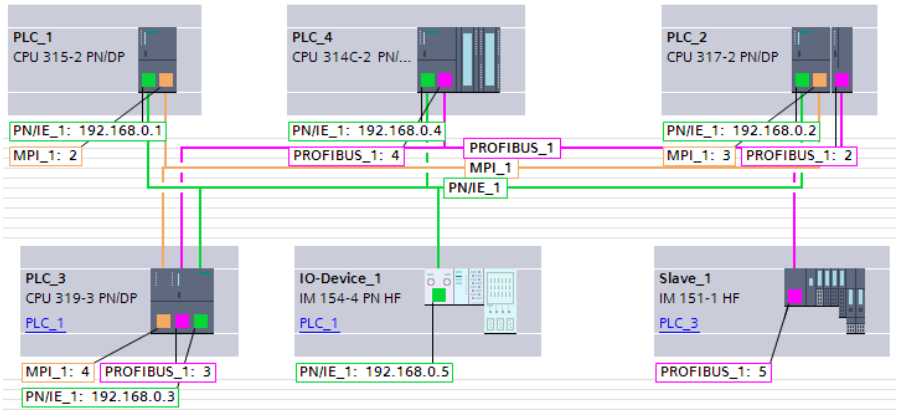


Fig. 3.8 Display of node addresses in the Network view

3.4.4 Connections

Introduction

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

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

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

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

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

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

Connection types

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

You use the other connection types, for example, if you wish to transmit data to third-party devices. This usually takes place by means of a communication module. Various versions of the CP 343-1 communication module are available for communication over Industrial Ethernet. In addition to an S7 connection, you can use the protocols for TCP connection, ISO transport connection, ISO-on-TCP connection, UDP connection, e-mail connection, and FTP connection. The CP 343-5 communication module is available for the PROFIBUS subnet with which you can additionally exchange data via PROFIBUS FMS.

You use the HMI connection for communication with an HMI station.

Connection resources

Every connection requires connection resources on the communication partners involved for the end point of the connection and for the transition point in a 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. One connection is always reserved for a programming device, and another for an HMI station (these cannot be used for anything else).

Temporary connection resources are also required for the “non-configured connections” during S7 basic communication.

Configuring connections

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 (Fig. 3.9).

Click with the left mouse button on a station, drag the connection line with the mouse button pressed to the other station, and release the button. A connection with the connection name is displayed as a blue/white patterned line. Several logical connections can be created 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.

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

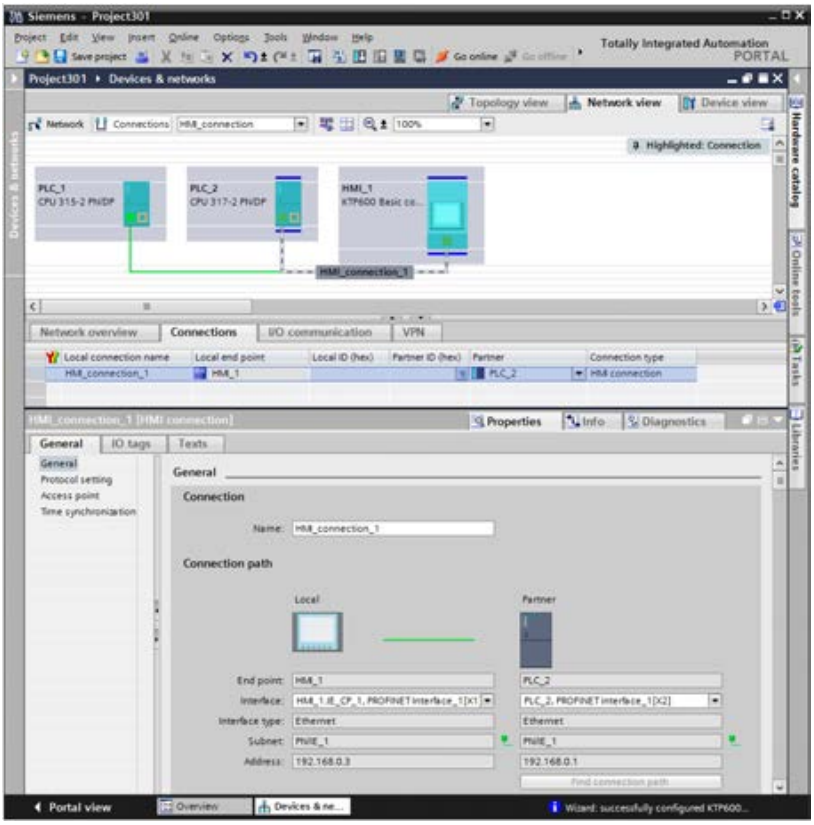


Fig. 3.9 Representation of an HMI connection in the network configuration

click on one of the connections listed in the tooltip window, this connection is displayed highlighted in the Network view.

Connection properties

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

3.4.5 Configuring an MPI subnet

To configure an MPI subnet, drag the MPI of one station to the MPI of the other station with the mouse. An MPI subnet will be created automatically. You can also drag an MPI to an existing MPI subnet.

Setting the properties of an MPI subnet

To set the properties, select the MPI 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 and the transmission speed in this subnet. You must observe the technical specifications of the involved modules when doing this.

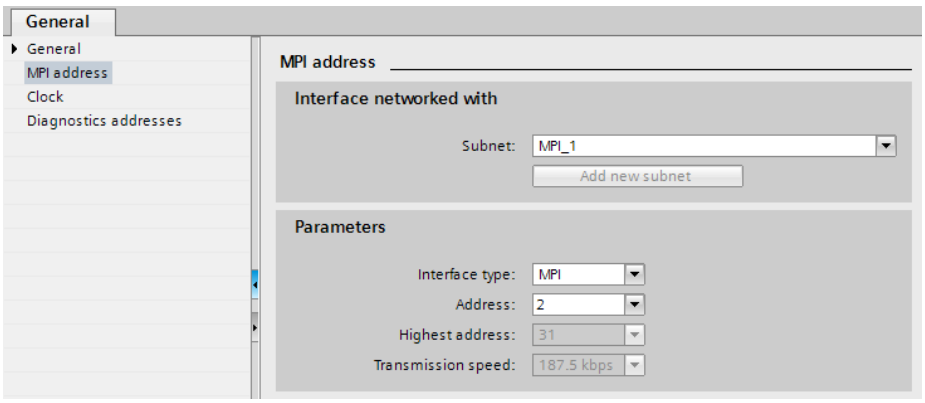


Fig. 3.10 Example of the properties of an MPI

Setting the properties of an MPI interface

To set the properties, select the MPI and then the *Properties* tab in the inspector window. Under *General* you can set a different name for the interface. Under *MPI address* you set the MPI address of the CPU (Fig. 3.10). Note with CPUs with the Order No. 6ES7 3xx-xxxxx-0AB0 that the FM and CP modules with an MPI which are present in the station occupy the MPI address following the CPU.

The MPI address can be assigned as desired. It must not be higher than the highest node address for the MPI subnet set in the CPU properties during hardware configuration.

The MPI address 0 is reserved as standard for a programming device, which can be connected temporarily to the MPI subnet for servicing purposes. An operator station with MPI connection has the MPI address 1 as the factory setting, and a CPU 300 has the MPI address 2.

3.4.6 Configuring a PROFIBUS subnet

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. With a combined MPI/DP interface, first set *PROFIBUS* as the interface type in the interface properties.

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. 3.11).

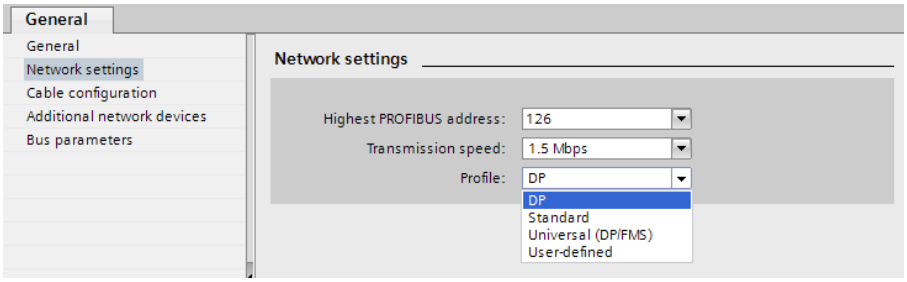


Fig. 3.11 Example of network settings on the PROFIBUS

The selectable bus profiles have the following properties:

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

Setting the properties of a DP interface

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

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

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

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

Under *Operating mode* 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 functions for PROFIBUS DP” on page 640.

You can change the diagnostic address of the interface under *Diagnostics addresses*. Further information can be found in Chapter 16.4.2 “Addresses with PROFIBUS DP” on page 632.

3.4.7 Configuring a PROFINET subnet

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

Setting the properties of a PROFINET subnet

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

Setting the properties of a PN interface

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

Ethernet address (MAC address)

The MAC (Media Access Control) address is an unambiguous address assigned to the device and defined by the manufacturer. It consists of three bytes with the manufacturer ID and three bytes with the device ID. The MAC address is usually printed on the device and is assigned to the latter during the configuration – if this has not already been carried out in the factory. The bytes are assigned in hexadecimal form

(symbols 0 to F), 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. 3.12).

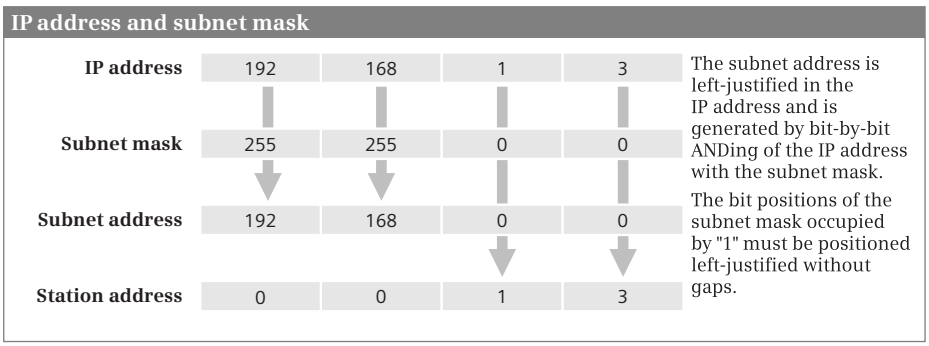


Fig. 3.12 Example of the structure of an IP address

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, separat-

ed 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 (see Fig. 3.13).

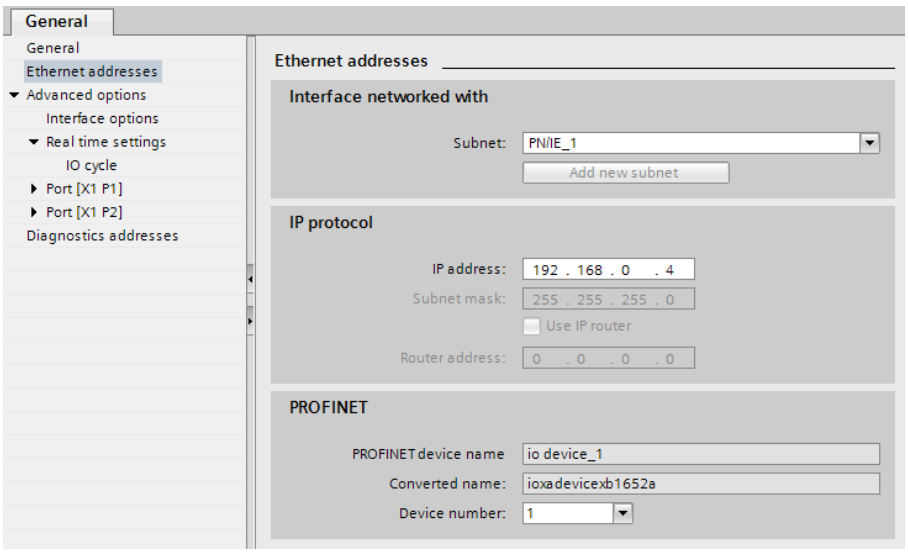


Fig. 3.13 Example of Ethernet addresses for an IO device

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.

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

You set the *Operating mode* for a CPU with integral PN interface. In addition to operation as an IO controller, you can also activate operation as an IO device and determine from which device the interface is to be parameterized: from the assigned IO controller or from the IO device itself.

Under *Advanced options* you can set, among others, the options for real-time mode. Refer to Chapter 16.3.4 “Configuring PROFINET IO” on page 619 (Fig. 3.14) for how to configure a PROFINET IO system.

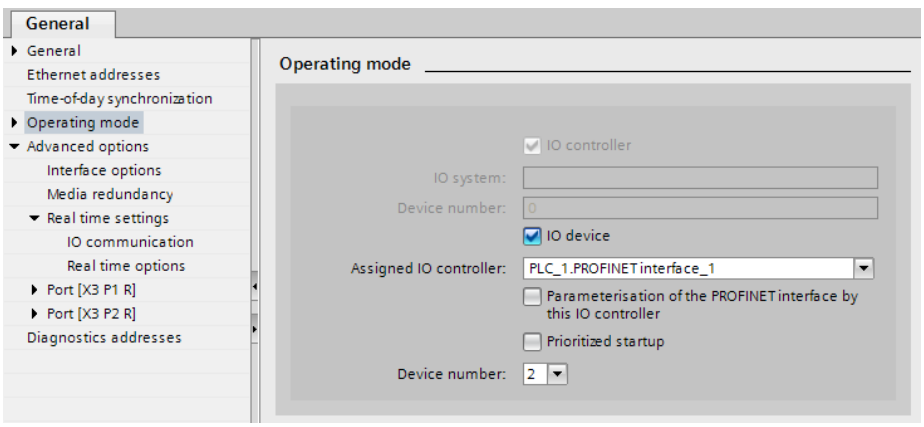


Fig. 3.14 Example of the operating mode settings for an integral PN interface

You can change the diagnostic addresses of the interface under *Diagnostics addresses*. Further information can be found in Chapter 16.3.2 “Addresses with PROFINET IO” on page 615.

3.4.8 Configuring an AS-i subnet

You can configure an AS-i subnet using STEP 7 if you use the CP 343-2P communication module as AS-i master (Fig. 3.15).

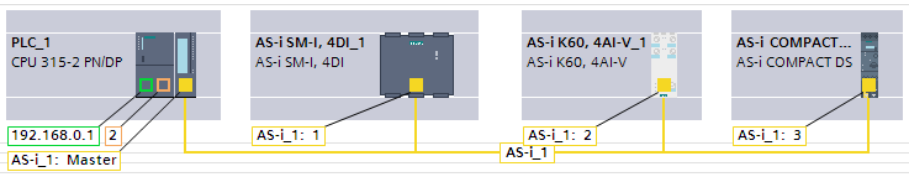


Fig. 3.15 Example of an AS-i master system with CP 343-2P

Setting the properties of an AS-i subnet

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

Setting the properties of an AS-i master interface

To set the properties, select the AS-i master and then the *Properties* tab in the inspector window. Under *General* you can set a different name for the interface. Under *Operating parameters* you enable the diagnostics interrupt for faults in the AS-i configuration, and activate or deactivate automatic address programming. If automatic address programming is activated, a failed AS-i slave can simply be replaced by a brand-new AS-i slave in protected mode: The AS-i master automatically programs the new AS-i slave with the address of the failed AS-i slave.

Setting the properties of an AS-i slave interface

To set the properties, select the AS-i interface in the field device and then the *Properties* tab in the inspector window. Under *General* you can set a different name for the interface. Under *AS Interface* you set the AS-i subnet used and the node address. The other settings specify operation on the AS-i subnet (see Chapter 16.7 “Actuator/sensor interface” on page 657).

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):

- *Peripheral inputs* are the memory areas on the input modules. They constitute the direct interface to the controlled machine, e.g. in order to scan the settings of control elements or sensors.
- *Inputs* are an image of the peripheral inputs in the CPU's system memory. These are normally processed by the user program when signal states of the machine are to be scanned and linked. The totality of the inputs is the process image input.
- *Peripheral outputs* are the memory areas on the output modules. They constitute the direct interface to the controlled machine, e.g. in order to control displays, valves, or contactors.

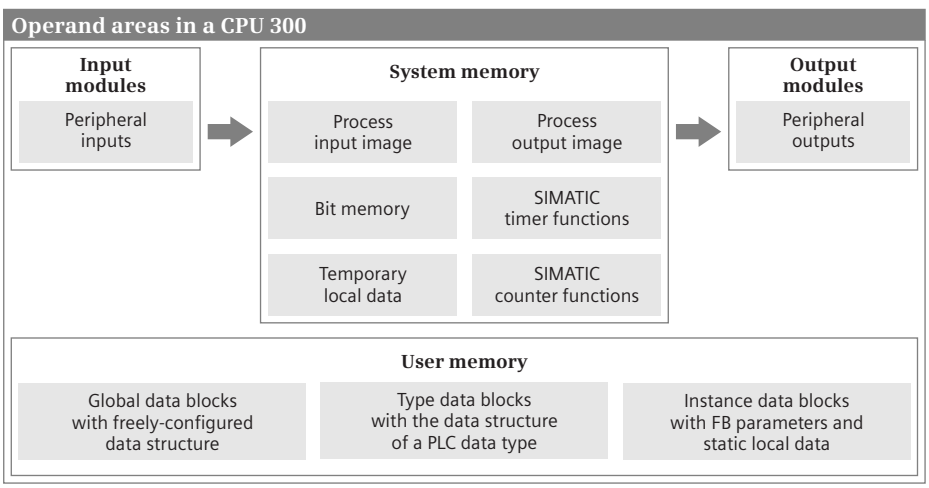


Fig. 4.1 Operand areas in a PLC station

- *Outputs* are an image of the peripheral outputs in the CPU's system memory. These are normally processed by the user program if the results of the control functions are to be output. The totality of the outputs is the process image output.
- *Bit memories* are a memory area in the CPU's system memory and are used as a global intermediate memory for signal states and numerical values.
- *Data* refers to memory areas in the user memory. Data is organized in *data blocks*, which can either be addressed globally from all parts of the user program or which locally manage the data of a function block. They are then called *static local data*.
- *Temporary local data* refers to memory areas assigned by the CPU to a 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 SIMATIC timer and counter functions save their data – contrary to the IEC timer and -counter functions – at a fixed position in the system memory. Therefore the SIMATIC timer and 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 12.3 “SIMATIC timer functions” on page 443 and 12.5 “SIMATIC counter functions” on page 462.

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), or one doubleword (32 bits) are referred to as *digital operands* or *digital tags*. The data types for digital tags are extremely diverse. For example, the data type INT (integer) refers to a 16-bit wide fixed-point number, the data type CHAR to a character in ASCII code, and the data type ARRAY to a combination of several tags with the same type of data under one tag name.

4.1.2 Operand areas: inputs and outputs

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 5.5.2 “Process image updating” on page 177).

The process image input is located in the CPU's system memory. It contains the operand area *Inputs*. The inputs are used to scan binary signals in the user program

and to link their signal states. This means that the input modules are not directly scanned in the normal case, it is the process image input which is scanned.

Access to the peripheral inputs is read-only. Inputs can be read and written. Inputs not occupied by peripheral inputs 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 5.5.2 “Process image updating” on page 177).

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

Consistent user data 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. For example, if you transfer four bytes individually, the transfer program can be interrupted between each byte by a program of higher priority with the facility for changing the data in the source or destination area.

With a direct access to user data (loading and transferring), the data is read and written as byte, word, or doubleword. The load and transfer statements which are also taken as the basis for the MOVE box with LAD/FBD and for the assignment of tags with elementary data types with SCL cannot be interrupted during execution. If you wish to transfer a data block with more than four bytes between the system and work memories without interruption, use the system function UBLKMOV.

The data transfer for PROFINET IO between IO controller and IO device is consistent for up to 1024 bytes; for PROFIBUS DP between DP slave and DP master, it is 32 bytes. This applies regardless of the distribution of the user data interface into several consistent transfer areas for I-slaves or I-devices. The data consistency with direct data exchange is as with direct access (1, 2 and 4-byte consistency).

You can specify consistent user data areas when configuring stations of the distributed I/O with three bytes or more than four bytes of user data. You then transfer these areas consistently to the parameterized destination area, for example a data area in the work memory, using the system functions DPRD_DAT and DPWR_DAT.

Note that the “normal” updating of process images can be interrupted following each transmitted doubleword. An exception is the transfer of user data blocks for distributed I/O with the system functions SYNC_PI and SYNC_PO, which transfer the process image partition when there is a isochronous mode interrupt.

CPU-specific values apply to the maximum size of a consistency area for data transfer with S7 basic communication and S7 communication by the operating system.

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

4.1.3 Operand area: bit memory

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

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

Bit memories can be read and written without limitation.

Retentive bit memories

Some of the bit memories can be set “retentive”, i.e. this part retains its signal state even when deenergized. Retentivity always starts at memory byte 0 and ends at the set upper limit. You can set the retentivity when assigning the CPU parameters.

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.

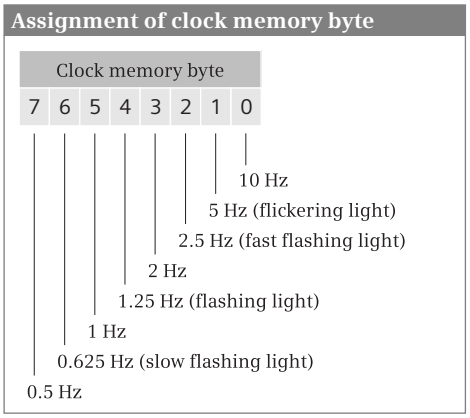


Fig. 4.2 Assignment of clock memory byte

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

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

4.1.4 Operand area: data

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

- *Global data blocks* have a data structure which is defined when configuring the data block.
- *Type data blocks* are derived from PLC data types. The data structure of a type data block is based on a PLC data type.
- *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.

Data blocks are global objects which can be addressed in absolute mode using their number, or symbolically using a name. The name of a data block must be unique

on the CPU. The data operands within a data block are local data. The name of a data operand must be unique in the data block. In association with the data block, a data operand has the character of a global tag.

Data tags of every data block can be read and written without restrictions – independent of the version as global, type or instance data block – unless the *Data block write-protected in device* attribute has been activated for the data block.

The data present in data blocks can be retentive, i.e. it retains its value even when deenergized. With a CPU 300, only a complete data block can be either retentive or non-retentive.

4.1.5 Operand area: temporary local data

Temporary local data includes operands present in the local data stack (L stack) in the CPU's system memory. Each code block can use temporary local data for intermediate storage. Temporary local data is only available during block processing, and its contents are lost when the block is left.

The tags in the operand area "temporary local data" are declared in the block interface (see Chapter 5.2.5 "Block interface" on page 161).

The operating system of the CPU 300 provides 2 KB of temporary local data in every block for block processing. A maximum of 32 KB is available per execution level, i.e. per organization block with all blocks called within it.

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

Use of temporary local data

In order to use temporary local data for meaningful purposes, it must be written before being read. Within the block, the temporary local data can be read and written without limitations.

The temporary local data is addressed symbolically as standard. Absolute addressing of temporary local data is only possible in the programming languages STL, LAD, and FBD via operand area L.

All elementary, complex and PLC data types are permissible for the tags in the temporary local data.

All operations which also apply to the bit memories are permissible for the temporary local data. However, please note that a temporary local data bit is not suitable as an edge trigger flag since it does not retain its signal state beyond block processing.

The organization blocks pass on start information in the temporary local data. The general data structure of the start information is described in Chapter 4.8 "Start in-

formation” on page 143, the special characteristics along with the individual organization blocks.

A tag in the temporary local data can be declared – as an exception – with the data type ANY. You can then generate an ANY pointer using STL which can be changed during runtime (for more details, see Chapter 4.6.3 ““Variable” ANY pointer with STL” on page 139). Using SCL you can assign the address of another (complex) tag to a temporary ANY tag during runtime (for more details, see Chapter 4.6.4 ““Variable” ANY pointer with SCL” on page 140).

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.

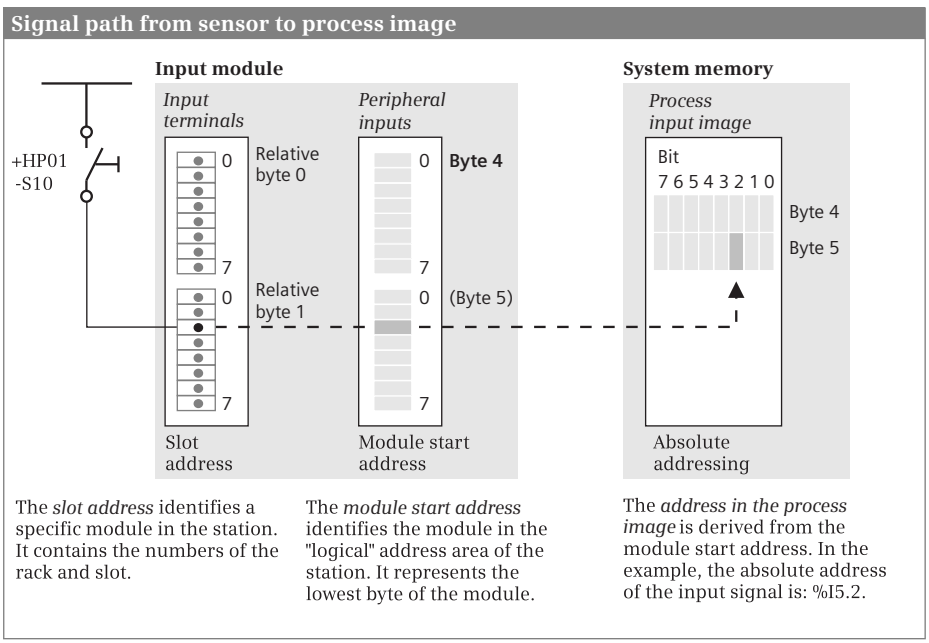


Fig. 4.3 Signal path from sensor to process image

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

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

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

4.2.2 Absolute addressing of tags

The absolute address of a signal state or numerical value consists of the specification of the operand area and operand width supplemented by a number which defines the position within the operand area. Examples are:

%I2.5	Bit 5 in byte 2 in the operand area “Inputs”.
%QB34	Byte 34 (8 bits) in the operand area “Outputs”.
%IW128:P	Word 128 (16 bits) in the operand area “Peripheral inputs”.
%MD200	Doubleword 200 (32 bits) in the operand area “Bit memory”.

An absolute address is identified via a preceding percent sign (%).

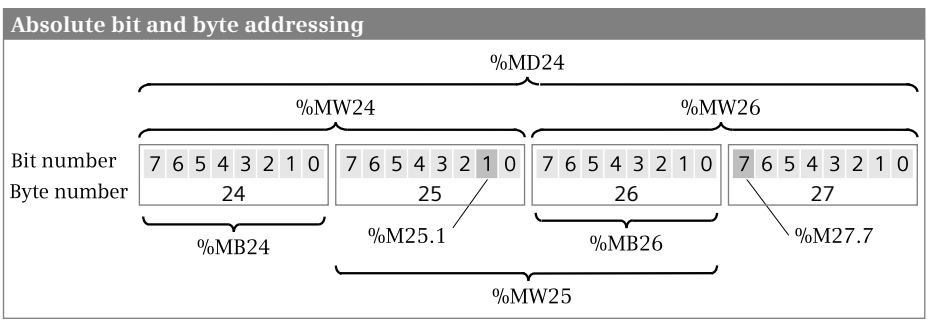


Fig. 4.4 Example of bit and byte assignments

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.

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 the peripheral addresses, ":P" is appended to the input or output address (Table 4.1).

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.

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	I:P	–	%IBy:P	%IWy:P	%IDy:P
Output	Q	%Qy.x	%QBy	%QWy	%QDy
Peripheral output	Q:P	–	%QBy:P	%QWy:P	%QDy:P
Bit memory	M	%My.x	%MBy	%MWy	%MDy

y = byte address; x = bit address

Complete addressing of data operands

Data operands are combined into data blocks in the user memory. 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. A data block can have a size up to 64 KB (with the CPU 312: 32 KB).

The absolute address of a data operand is shown in the *Offset* column of the block interface once the data block has been compiled.

Partial addressing of data operands

Partial addressing of data operands is possible in the programming languages LAD, FBD, and STL.

"Partial addressing" means that data operands are addressed individually. 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 14.5.1 "Open data block" on page 555).

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 4.3).

Table 4.3 Operand IDs with partially addressed data operands

Operand area	ID	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

The absolute address of a data operand is shown in the *Offset* column of the block interface once the data block has been compiled.

Complete addressing should be preferred for the absolute addressing of data operands. Partial addressing is only considered for special applications since, in order to use partial addressing error-free, it is necessary to know how the program editor compiles the user program into machine code. In the following cases the program editor generates additional statements which change the contents of the data block register – but not visible for you in the user program of the respective programming language:

Complete addressing of data operands

With each complete addressing of data operands, the program editor first opens the data block and then accesses the data operands. The DB register is overwritten in each case. This also applies to the writing of block parameters with completely addressed data operands.

Access to block parameters

Access to the following block parameters changes the content of the DB register: With functions (FC), all block parameters with complex data type, and with function blocks (FB), in-out parameters with complex data type.

Calls of function blocks and system function blocks

The CALL statement or the call box saves the number of the current instance data block in the DB register prior to the actual block call (by swapping the data block registers) and opens the instance data block for the called function block. As a result, the associated instance data block is always open in a called function block. Following the actual block call, the CALL statement or the call box again swaps the data block registers so that the current instance data block is again available in the calling function block. The CALL statement or the call box thus changes the content of the DB register.

DI register in function blocks

In function blocks, the DI register always has the number of the current instance data block. All access operations to block parameters or static local data are carried out via the DI register and, as a side note, also via the address register AR2 for function blocks with “multi-instance capability”. If you change the content of the DI or AR2 register (possible with STL), all subsequent access operations to block parameters and static local data are carried out incorrectly. Every change to these registers *must* be reversed before block parameters or static local data are accessed.

Absolute addressing of static local data

The static local data – just like the block parameters – are local tags in a function block. Local tags are usually addressed symbolically.

The programming language STL makes it possible to use indirect addressing to address static local data operands in absolute mode. For more details, refer to Chapter 4.3.3 “Working with the address registers with STL” on page 109.

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 using *“Data block”. Data operand* just like with “normal” data tags.

Absolute addressing of temporary local data

The temporary local data are local tags in a code block. Local tags are usually addressed symbolically.

Absolute addressing is also possible with the programming languages LAD, FBD, and STL. The operand ID is L (Table 4.4). 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.

Table 4.4 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

Absolute addressing of SIMATIC timer and counter functions

The SIMATIC timer and 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. Table 4.5 shows the operand IDs of these functions.

Table 4.5 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 of tags

During symbolic addressing, an operand is assigned a name 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.

Letters, digits, and special characters – except double quotes – are permissible as tag names. No distinction is made between upper and lower case when checking the name.

Symbolic addressing of global tags

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

ripheral outputs, bit memories, SIMATIC timer functions, and SIMATIC counter functions.

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 L stack for all 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. Example: In the tag named *"Motor_1".Switch_on_motor*, *"Motor_1"* is the name of the data block and *Switch_on_motor* the name of the data operand.

Partial addressing of the data block and data operand can be entered independent of each other either absolute or symbolic. However, the program editor indicates the complete address – depending on the setting – either as absolute and/or symbolic address.

4.2.4 Addressing constants

A "constant" is a fixed numerical value. When programming a constant, you must observe the correct notation depending on the constant's data type (Table 4.6).

Symbolic addressing of constants

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

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

Table 4.6 Notation for constant values

Data type	Length	Description	Examples of representation
BOOL	1 bit	Bit value	FALSE, TRUE, 16#0, 16#1
BYTE	8 bits	Bit string with 8 bits	B#16#0, B#16#FF
CHAR	8 bits	Character in ASCII code	'a', 'Z'
WORD	16 bits	Bit string with 16 bits	W#16#0000, W#16#FFFF
DWORD	32 bits	Bit string with 32 bits	DW#16#0000_0000, DW#16#FFFF_FFFF
INT	16 bits	16-bit fixed-point number	–32_768, 0, +32_767
DINT	32 bits	32-bit fixed-point number	–2_147_483_648, 0, +2_147_483_647
REAL	32 bits	32-bit floating-point number	Exponential representation:+1.234567E+02 [1)] Decimal representation: –123.4567 [1)]
S5TIME	16 bits	Time value for SIMATIC times	S5T#10ms, S5TIME#2h46m30s
TIME	32 bits	Time value in IEC format	T#–24d20h31m23s647ms, T#0ms, TIME#24d20h31m23s647ms
DATE	16 bits	Date	D#1990-01-01, DATE#2168-12-31
TIME_OF_DAY	32 bits	Time of day	TOD#00:00:00.000, TIME_OF_DAY#23:59:59.999
DATE_AND_TIME	64 bits	Date and time	DT#1990-01-01-00:00:00.000, DATE_AND_TIME#2168-12-31-23:59:59.999
STRING	variable	Character string	'ABCD', '012345'

[1)] For range of values, see Chapter 4.4.5 "Floating-point data type REAL" on page 125

4.3 Indirect addressing

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

The statements required for indirect addressing are present in the programming languages STL and SCL, but not in LAD and FBD.

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!*

Overview of types of addressing with STL and SCL

STL and SCL use different methods for indirect addressing. STL distinguishes between memory-indirect and register-indirect addressing:

- Memory-indirect addressing,
  example: *IW [%MD200]*, the number of the input word is present in the bit memory doubleword %MD200.
- Register-indirect, area-internal addressing,
  example: *IW [AR1, P#2.0]*, the number of the input word is present in the address register AR1; it is incremented by the offset P#2.0 when the operation is executed.
- Register-indirect, area-crossing addressing,
  example: *W [AR1, P#0.0]*, the operand area and the number of the operand are present in the address register AR 1; the number is not incremented when the operation is executed.

Memory-indirect addressing uses doublewords from the operand areas “Data” (DBD and DID), “Bit memories” (MD), and “Temporary local data” (LD) as “address registers”. These operands can be addressed absolutely or symbolically. With symbolic addressing, the data types must have the required width of 16 bits or 32 bits.

Register-indirect addressing uses the two address registers AR1 and AR2.

STL allows determination of the absolute address of a symbolically addressed local tag during runtime and to then change this address if applicable. The required procedure is described in Chapter 4.3.4 “Direct access to complex local tags with STL” on page 116.

SCL allows indirect addressing of operands and arrays:

- With indirect addressing of operands, SCL considers an operand area like an array whose elements can be addressed individually. Example: %*MW(#index)*, the number of the bit memory word is present in the *#index* tag.
- In the case of a tag with the data type ARRAY, SCL permits a tag as index. Example: *#Array[#index]*, the number of the array element is present in the *#index* tag.

The index tags can be global or local tags addressed absolutely or symbolically. With symbolic addressing, the index tags must be of data type INT.

4.3.1 Memory-indirect addressing with STL

“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”. 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. Associated with this are all disadvantages of partial addressing of data operands, since the program editor also uses the data block registers DB and DI, which is not visible on the programming interface. Refer to section “Partial addressing of data operands” on page 99 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 4.6.2 “Pointer” on page 136. Refer to section “Partial addressing of data operands” on page 99 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”.

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. 4.5 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.

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. 4.5 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. 4.5 uses the "*Number*" tag as address register. The tag could be, for example, the bit memory word %MW204 with the data type INT.

4.3.2 Register-indirect addressing with STL

Register-indirect addressing uses one of the address registers AR1 or AR2 in order to determine the address of the operand. Operations used in conjunction with register-indirect addressing are located in the following operand areas: 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). Refer to section "Partial addressing of data operands" on page 99 for information on what you must observe when addressing data operands.

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 4.6.2 "Pointer" on page 136 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 4.3.3 "Working with the address registers with STL" on page 109.

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. 4.6 (top) show area-internal, register-indirect addressing.

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. 4.6 (bottom) show cross-area, 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  MW[AR1,P#0.0]`	The statement L %MW10 is executed.
`L  MW[AR1,P#2.0]`	The statement L %MW12 is executed.
`+AR1  P#20.0`	The byte address is incremented by 20 in address register AR1.
`L  MW[AR1,P#0.0]`	The statement L %MW30 is executed.
`L  MW[AR1,P#4.0]`	The statement L %MW34 is executed.
`LAR2  P#Q16.3`	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  W[AR1,P#0.0]`	The statement L %MW64 is executed.
`L  W[AR1,P#2.0]`	The statement L %MW66 is executed.
`+AR1  P#12.0`	The byte address is incremented by 12 in address register AR1.
`L  B[AR1,P#0.0]`	The statement L %MB76 is executed.
`L  B[AR1,P#4.0]`	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  D[AR2,P#0.0]`	The statement T %DBD32 is executed.
`A  [AR2,P#0.1]`	The statement A %DBX32.1 is executed.
`L  MW[AR2,P#4.0]`	The statement L %MW36 is executed.

Fig. 4.6 Examples of register-indirect addressing with STL

4.3.3 Working with the address registers with STL

The statements available for working with the address registers AR1 and AR2 are listed in Table 4.7. 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. 4.7. All statements are executed independent of conditions, and do not influence the status bits.

Table 4.7 Overview of address register functions

Operation	Operand	Function
LAR1		Load address register AR1 with the content of accumulator 1
LAR1	Operand	Load address register AR1 with the content of the operand
LAR1	Pointer	Load address register AR1 with the pointer
LAR1	AR2	Load address register AR1 with the content of address register AR2
LAR2		Load address register AR2 with the content of accumulator 1
LAR2	Operand	Load address register AR2 with the content of the operand
LAR2	Pointer	Load address register AR2 with the pointer
TAR1		Transfer the content of address register AR1 to accumulator 1
TAR1	Operand	Transfer the content of address register AR1 to the operand
TAR1	AR2	Transfer the content of address register AR1 to address register AR2
TAR2		Transfer the content of address register AR2 to accumulator 1
TAR2	Operand	Transfer the content of address register AR2 to the operand
CAR		Swap the contents of the address registers
+AR1		Add the content of accumulator 1 to address register AR1
+AR1	Pointer	Add the pointer to address register AR1
+AR2		Add the content of accumulator 1 to address register AR2
+AR2	Pointer	Add the pointer to address register AR2

Loading into an address register

The LAR*n* statement loads an area pointer into address register AR*n* (*n* = 1 or 2). You can select an area-internal or cross-area pointer or a doubleword from operand areas (bit memories, temporary local data, global data, and instance data) as the source. The content of the doubleword must correspond to the format of an area pointer.

If you do not specify an operand, LAR*n* loads the content of accumulator 1 into address register AR*n*.

Using the LAR1 AR2 statement, you copy the content of address register AR2 into address register AR1.

Transferring from an address register

The TAR*n* statement transfers the complete area pointer from address register AR*n* (*n* = 1 or 2). You can specify a doubleword from operand areas (bit memories, temporary local data, global data, and instance data) as the destination.

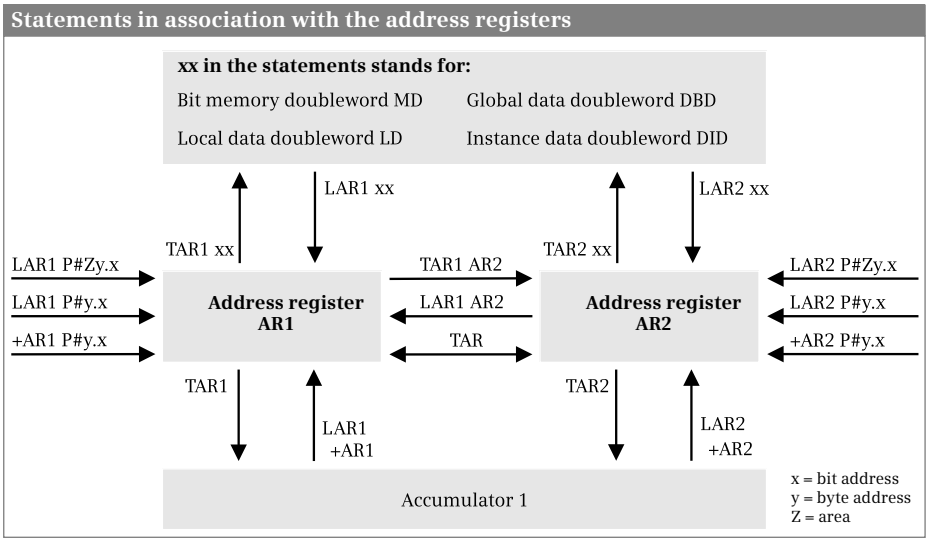


Fig. 4.7 Statements for working with address registers

If you do not specify an operand, TAR*n* transfers the content of address register AR*n* 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. 4.8 (top) shows an example of application of the statement.

Note: The system functions BLKMOV and UBLKMOV are available for transferring larger data areas.

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. 4.8 (middle) shows an example of application of the statements.

Example of swapping address register contents

8 bytes of data are transferred between the memory area starting at %MB100 and the data area starting at %DB20.DBB200. The transfer direction is determined by bit memory %M126.6. The contents of the address registers are swapped if %M126.6 has the signal state “0”.

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 CDB statement.

```
      LAR1  P#M100.0
      LAR2  P#DBX200.0
      OPN   %DB20
      A     %M126.6
      JC    swap
      CAR
swap: L     D[AR1,P#0.0]
      T     D[AR2,P#0.0]
      L     D[AR1,P#4.0]
      T     D[AR2,P#4.0]
```

Example of adding a pointer to the address register

A data area of length *#Number_ data* in data block *%DB14* 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”.

```
      OPN   %DB14
      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
```

Example of adding the accumulator content to the address register

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).

```
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]
```

Fig. 4.8 Examples of register-indirect addressing with STL

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. 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. 4.9).

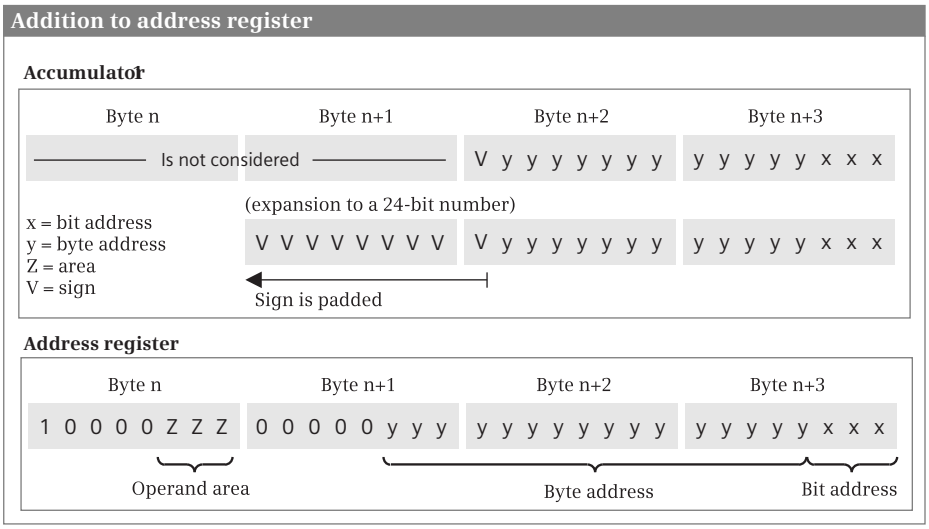


Fig. 4.9 Addition to address register

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. 4.8 (bottom) shows an example of application of the statement.

Note: The system function FILL is available for filling in larger data areas with bit patterns.

Notes on use of address register AR1

STL uses address register AR1 to access block parameters which are transferred as DB pointers. With functions (FC), these are all block parameters with complex data type, and with function blocks (FB), these are in/out parameters with complex data type.

If you therefore access such a type of block parameter, e.g. in order to scan a bit component of a structure or to write an INT value to an array component, the content of address register AR1 is changed and – as a side note – also the content of the DB register. This also applies if you "pass on" block parameters with this data type to called blocks.

If you use address register AR1, access to a block parameter as described above must not be carried out between loading of the address register and indirect addressing. Otherwise you must save the content of AR1 prior to access, and load it again following access (see Fig. 4.10 top).

Notes on use of address register AR2

With function blocks with “multi-instance capability”, STEP 7 uses address register AR2 as the “base address register” for instance data. When calling a single instance, P#DBX0.0 is present in AR2 and all access operations to block parameters or static local data in the FB use the register-indirect, area-internal addressing with operand area DI via this register.

Calling of a local instance increases the “base address” with +AR2 P#y.x so that access is possible relative to this address within the called function block which uses the instance data block of the calling function block. In this manner, function blocks can be called either as an independent instance or as a local instance (and in this case at any position in a function block, and also more than once).

If you therefore wish to use address register AR2 in a function block with “multi-instance capability”, you must previously save the content and then restore it following use. You must not program an access operation to block parameters or static local data in the area in which you are working with the address register AR2 (see Fig. 4.10 bottom).

With a function block without “multi-instance capability”, the program editor does not use address register AR2. The multi-instance capability can be activated or deactivated in the block attributes for a function block generated via a source file where the keyword CODE_VERSION1 is used.

There are no limitations to working with the address register AR2 within functions (FC).

Notes on absolute addressing of static local data

Static local data and block parameters are normally addressed symbolically in the program of the “own” function block. In the case of absolute addressing, you must note that the function block can be called either as a single instance or as a local instance.

The address of the block parameters and static local data specified in the interface of the function block is the offset from the beginning of the instance data. It applies if you call the function block as a single instance with a separate instance data block; assignment of the data operands then commences at address byte zero.

Example: A 16-bit tag “Setpoint” in the static local data has the address 32.0, which can be read in the block interface in the *Offset* column following compilation of the function block. You can address this tag in absolute mode with %DIW32 in the function block's program. However, this only works if you call the function block as a single instance.

Example “Save address register AR1”

`Temp`
`  AR1Memory : DWORD`
`  DBMemory : WORD`
`...`
	Declaration of used intermediate memories

`...`
`LAR1  P#y.x`
`OPN  DBz`
`...`
	Any indirect addressing with AR1 and DB register

`L  DBNO`
`T  #DBMemory`
`TAR1  #AR1Memory`
	Save register contents

`L  "Motor1".Actual_value`	Access to a tag with complex data type
`OPN  DB[#DBMemory]` `LAR1  #AR1Memory`	Restore register contents
`T  DBW[AR1,P#0.0]`	Save loaded value

Example “Save address register AR2”

`Temp`
`  AR2Memory : DWORD`
`  DIMemory : WORD`
`...`
	Declaration of used intermediate memories

`L  DINO`
`T  #DIMemory`
`TAR2  #AR2Memory`
	Save register contents

`...`
`LAR2  P#y.x`
`OPN  DIz`
`...`
	Any indirect addressing with AR2 and DI register

`OPN  DI[#DIMemory]` `LAR2  #AR2Memory`	Restore register contents
`L  #Local_tag`	Access to block parameters and local tags

Fig. 4.10 Examples of saving address registers

If you call a function block as a local instance, its data is present “in the middle” of the instance data block of the calling block (the “multi-instance”). The offset from the start of the multi-instance data is present in address register AR2. The absolute address of a local tag is then the total of the contents of address register AR2 and the address in the called function block (of the local instance).

Example: A 16-bit tag “Setpoint” in the static local data has the address 32.0. This is the offset from the start of the local instance data. The offset from the start of the multi-instance data of the calling function block is present in address register AR2. The absolute address of this tag is then DIW[AR2,P#32.0] in the program of a function block with multi-instance capability.

Limitations when addressing static local data

In the case of function blocks without multi-instance capability, you can use all statements described in this chapter without limitation.

In the case of function blocks with multi-instance capability, the program editor accesses instance data addressed symbolically by means of the address register AR2, i.e. all access operations are carried out indirectly. Symbolic addressing of instance data is not possible in conjunction with the address registers or further indirect addressing, since in this case the instance data would be addressed indirectly twice.

If you wish to use local data addressed symbolically in function blocks with multi-instance capability as pointers, you must use a “detour” via an intermediate memory or accumulator 1.

Program in function block without multi-instance capability	Program in function block with multi-instance capability	Program in function block with multi-instance capability and absolute addressing

`Static`
`  Pointer : DWORD`
`...`
	`Static`
	`  Pointer : DWORD`
	`...`
	`Temp`
	`  tPointer : DWORD`
		`Static  Offset`
		`  Pointer : DWORD  32.0`
		`...`

`L  MW[#Pointer]`	`L  #Pointer`
	`T  #tPointer`
	`L  MW[#tPointer]`
		`L  MW[DID32]`

`LAR1  #Pointer`	`L  #Pointer` `LAR1`	`LAR1  %DID32`
`TAR1  #Pointer`	`TAR1` `T  #Pointer`	`TAR1  %DID32`

Fig. 4.11 Examples of use of local data as area pointer

Fig. 4.11 (left example) shows the program of a function block without multi-instance capability. The example in the middle shows the same program in a function block with multi-instance capability and symbolic addressing of the local data. The example on the right shows the program with absolute addressing of the local data.

4.3.4 Direct access to complex local tags with STL

You can access local tags with elementary data types using "normal" STL statements. Local tags with complex 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 complex data types.

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), and global data operands (DB).

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 4.8 are approved for *name*.

Table 4.8 Permissible operands for loading the tag start address

Operation	*name* is a	OB	FC	FB with multi-instance capability	FB without multi-instance capability
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

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.

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 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. Chapter 18.5 “Storage of local tags” on page 712 describes how the block parameters are saved in the memory and their respective contents.

Note that `LAR2 P##name` overwrites address register AR2, which function blocks with multi-instance capability use as “start address register” for addressing the instance data!

The two examples at the top in Fig. 4.12 show the program in a function block 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.

`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 complex 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. 4.12 Examples of loading a tag address

4.3.5 Indirect addressing with SCL

SCL allows indirect addressing of operands and arrays. The addresses of the operands and array elements are present in index tags which can be global or local tags addressed absolutely or symbolically. With symbolic addressing, the index tags must be of data type INT. A constant or an expression with data type INT as a result can also be used instead of the index tags.

Data types of the index tags used

```
Data   : BLOCK_DB     //In declaration section Input

byte_adr : INT
bit_adr : INT
k        : INT
i        : INT
db_no_i : INT
db_no_w : WORD
```

Examples of indirect addressing of global operands

```
#var_bool := %MX(#byte_adr,#bit_adr);  //Address a memory bit
#var_byte := %IB(#byte_adr);
#var_word := %IW(#byte_adr):P;         //Address an I/O word
#var_dword := %QD(#byte_adr);

(* Copy an I/O area into a memory area *)
#k := 120;
FOR #i := 48 TO 62 BY 2 DO %MW(#k) := %IW(#i):P; #k := #k + 2;
END_FOR;
```

Examples of indirect addressing of data operands

```
//The data block is addressed absolutely
#var_bool := %DB10.DX(#byte_adr,#bit_adr);
//The data block is an input parameter
#var_bool := #Data.DX(#byte_adr,#bit_adr);
//The DB index is present with data type WORD
#var_bool := WORD_TO_BLOCK_DB(#db_no_w).DX(#byte_adr,#bit_adr);
//The DB index is present with data type INT
#var_word := WORD_TO_BLOCK_DB(INT_TO_WORD(#db_no_i)).DW(#byte_adr);
```

Fig. 4.13 Examples of indirect addressing of global operands with SCL

Indirect addressing of operands

With indirect addressing of operands, SCL considers an operand area like an array whose elements (bit, byte, word or doubleword) can be addressed individually. Example: When addressing %*MW(#byte_adr)*, the number of the bit memory word is present in the *#byte_adr* tag.

Two tags are required to address bit operands, one for the byte address and one for the bit address. An operand bit is identified by an "X". Example: %*IX(#byte_adr, #bit_adr)*.

You can address the following operand areas in this manner:

- Inputs (I), outputs (Q), and bit memories (M)
- Peripheral inputs (I:P) and peripheral outputs (Q:P), in each case without bit addressing
- Data operands (D) in conjunction with a data block which is absolutely or indirectly addressed (complete addressing)

SIMATIC timer functions (T) and SIMATIC counter functions (C) cannot be indirectly addressed with SCL.

In the middle section, Fig. 4.13 shows examples of indirect addressing of global operands with SCL.

Indirect addressing of data blocks and data operands

If a data operand is indirectly addressed, the data block can be addressed with the absolute address, as block parameter with the parameter type BLOCK_DB, or with the conversion function WORD_TO_BLOCK_DB (Fig. 4.13, lower section).

If a data block is indirectly addressed, the data operand can only be indirectly addressed. Absolute or symbolic addressing is not possible.

A data block is indirectly addressed by the conversion function WORD_TO_BLOCK_DB(#db_no_w), where the #db_no_w tag is present in data type WORD. The addressing is WORD_TO_BLOCK_DB(INT_TO_WORD(#db_no_i)) for a #db_no_i tag with data type INT, which one can handle better with arithmetic functions, for example.

Indirect addressing with data type ARRAY

With SCL, the component of a tag with data type ARRAY can be dynamically addressed. The array index need not only be a constant but can also be a tag or expression with data type INT (Fig. 4.14).

This indirect addressing is also possible with multi-dimensional arrays and with the addressing of partial arrays.

Examples of indirect addressing of array components

```
Array : ARRAY[1..12] OF REAL
Array_3dim : ARRAY[1..4,1..4,1..4] OF WORD
Array_1dim : ARRAY[1..4] OF WORD
...
index1 : INT
index2 : INT
index3 : INT
...
//Indirectly address an array component
#var_real := #Array[#index1];
#var_word := #Array_3dim[#index1,#index2,#index3];

//Indirectly address a partial array
#Array_1dim := #Array_3dim[#index1,#index2];
```

Fig. 4.14 Examples of indirect addressing of array components with SCL

4.4 Elementary data types

4.4.1 Introduction

Overview

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

- Elementary data types, which are predefined and can have a width of up to one doubleword (32 bits)
- Complex data types, comprising a combination of elementary data types
- Parameter types as additional data types for transfer of block parameters to functions and function blocks
- PLC data types, which a user can compile from existing data types
- System data types with a fixed structure and defined in STEP 7
- Hardware data types defined by the hardware configuration

Elementary data types have a width of 1, 8, 16 or 32 bits (Table 4.9). The data types BCD16 and BCD 32 are not data types in the closer sense – they cannot be assigned to a tag; they are only relevant to data type conversion. The elementary data types can be used together with tags from all operand areas.

Table 4.9 Overview of elementary data types

Bit-serial data types			**Durations and points in time**		
BOOL	1 bit	1-bit binary value (0, 1, FALSE, TRUE)	TIME	32 bits	Time value in IEC format (T#–24d20h31m23s648ms, TIME#24d20h31m23s647ms)
BYTE	8 bits	8-bit binary value (16#00…16#FF)	S5TIME	16 bits	Duration for SIMATIC timers (S5T#0ms, S5TIME#2h46m30s)
WORD	16 bits	16-bit binary value (16#0000…16#FFFF)	DATE	16 bits	Date (D#1990-01-01, DATE#2168-12-31)
DWORD	32 bits	32-bit binary value (16#0000 0000… 16#FFFF FFFF)	TIME_OF_DAY	32 bits	Time of day (TOD#00:00:00.000, TIME_OF_DAY#23:59:59.999)
Fixed-point and floating-point numbers			**BCD numbers and characters**		
INT	16 bits	16-bit fixed-point number (–32 768…+32 767)	BCD16 [1)]	16 bits	3 decades with sign (–999…+999)
DINT	32 bits	32-bit fixed-point number (–2 147 483 648… +2 147 483 647)	BCD32 [1)]	32 bits	7 decades with sign (–9 999 999…+9 999 999)
REAL	32 bits	32-bit floating-point number ($\pm 1.18\times10^{-38}$…$\pm 3.40\times10^{38}$)	CHAR	8 bits	A character in ASCII code ('a', 'A', '1', …)

[1)] Not data types in the closer sense; only relevant to data type conversion

Overlaying tags (data type views with SCL)

In the programming language SCL, additional data types can be "overlaid" on block parameters and local data. It is then possible to address the contents of tags completely or partially using various data types.

Example: You declare an input parameter with *Station* as the name and STRING as the data type. You transfer the *Station* tag to a called block in order to process it further, e.g. extend it by a number. You additionally wish to determine the current length of *Station*. To do this, you overlay an additional data type definition, for example in the form of a structure with two bytes, on the *Station* tag. The second byte then contains the current length of the string. The additional data type definition is to be called *length*, the components are called *maximum* and *current* (Fig. 4.15).

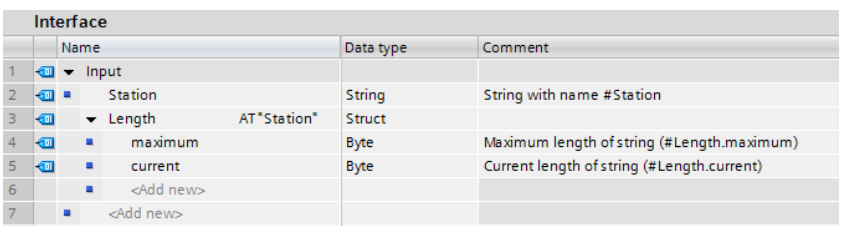


Fig. 4.15 Example of declaration of a "overlaid" data type

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.

The memory requirements of the overlaying data type definition must not be greater than the “original” tag (the new data type must “fit” into the tag).

You use an overlaying data type definition like any other tag, but only locally in the block. In the above 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.

Table 4.10 shows which overlaying data type definitions you can position over a tag with specific data type. If, for example, the tag is in the temporary local data of an FC and is a complex data type, the data type definitions overlaid on it can be of data types Elementary, Complex, and ANY.

Tags of type TIMER, COUNTER, and BLOCK_xx as well as the function value with functions (FC) cannot be overlaid by data types.

Table 4.10 Permissible data types for overlaying

Block	The tag is declared in the block	The tag is of data type											
		Elementary		Complex				POINTER		ANY			
FC	INPUT	E			C								
	OUTPUT	E			C								
	INOUT	E			C								
	TEMP	E	C	E	C		A				C		
FB	INPUT	E	C	E	C	P	A		C		C		
	OUTPUT	E	C	E	C								
	INOUT	E			C								
	STATIC	E	C	E	C								
	TEMP	E	C	E	C		A				C		

Overlaying data type:
E Elementary (BOOL, CHAR, BYTE, WORD, DWORD, INT, DINT, REAL, S5TIME, TIME, DATE, TIME_OF_DAY)
C Complex (DATE_AND_TIME, STRING, ARRAY, STRUCT) and PLC data types
P POINTER
A ANY

4.4.2 Bit-serial data types BOOL, BYTE, WORD, and DWORD

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

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) or as a 2×8-bit unsigned decimal number B#(0,0) to B#(255,255).

A tag with data type DWORD occupies 32 bits. The individual bits have no 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) or as a 4×8-bit unsigned decimal number B#(0,0,0,0) to B#(255,255,255,255).

Fig. 4.16 shows the structure of the data types BYTE, WORD, and DWORD.

4.4.3 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.

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. 4.16).

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.

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.

4.4.4 Fixed-point data types with sign INT and DINT

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 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.

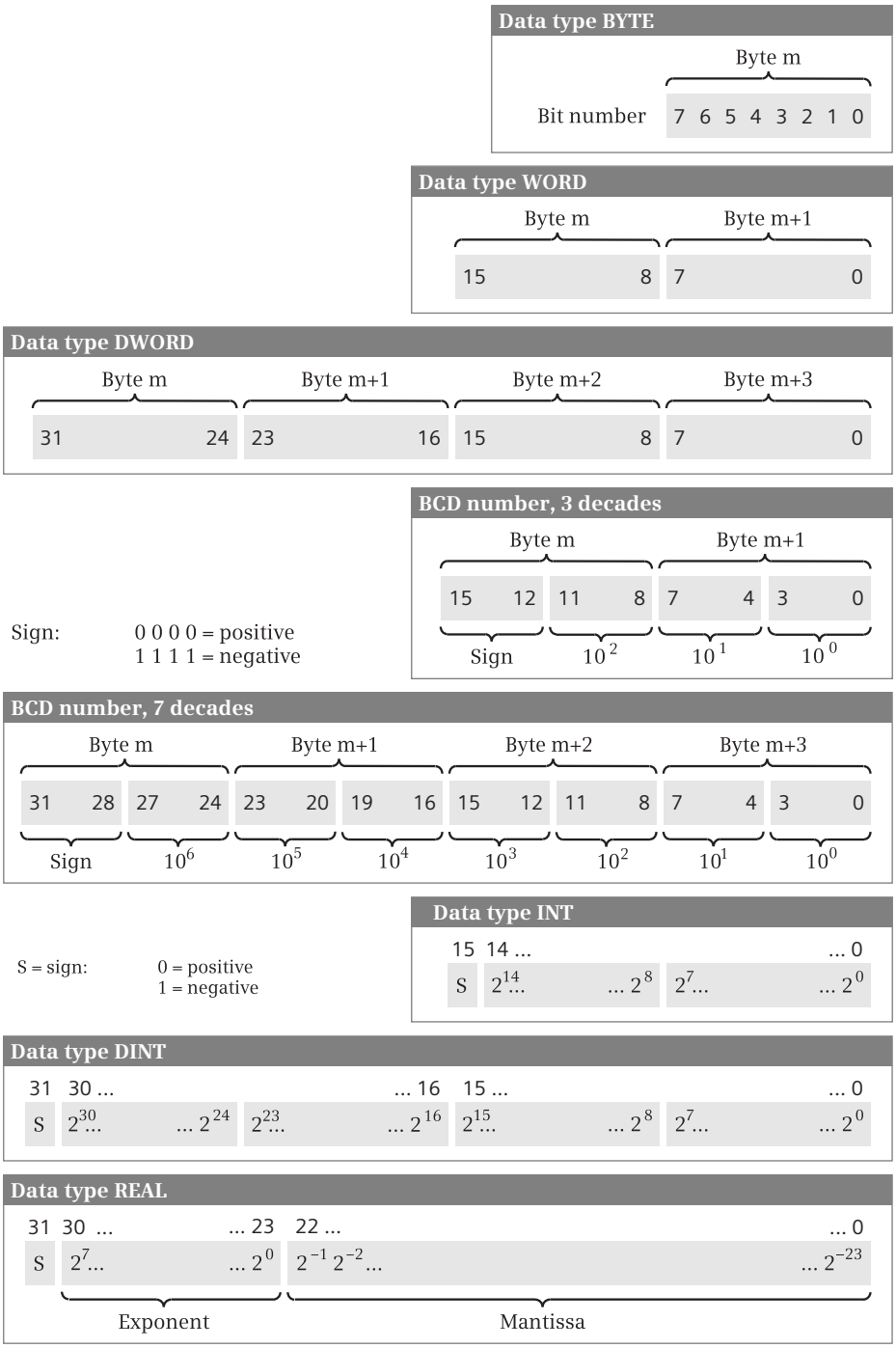


Fig. 4.16
Structure of bit-serial and BCD data types as well as of data types INT, DINT, and REAL

The data type DINT (double integer or fixed-point number) occupies one doubleword. The numerical range extends from -2^{31} to $+2^{31}-1$, i.e. from –2 147 483 648 to +2 147 483 647 or in hexadecimal notation from DW#16#8000_0000 to DW#16#7FFF_FFFF (Fig. 4.16).

An integer is saved as a DINT tag if it is larger than +32 767 or smaller than –32 768, or if the number is preceded by L# as type ID.

Example with STL: Using L –100 you load an INT number; using L L#–100 a DINT number. The difference is in the occupation of the left word in accumulator 1: In the example of the INT number –100, the value W#16#0000 is present here; with the DINT number –100, the sign W#16#FFFF.

Example with SCL: If you specify the constant value –100, the program editor automatically converts the value into a DINT number if it is linked to a DINT tag (“implicit” data type conversion).

4.4.5 Floating-point data type REAL

A tag with data type REAL 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 relevant positions (digits) which are positioned in exponential form in front of the “e” or “E”. The value following “e” or “E” is the exponent to base 10.

Conversion of the REAL tags into the internal representation of a floating-point number is handled by the program editor. Table 4.11shows the internal range limits of a floating-point number.

Table 4.11 Internal range limits of a floating-point number

Sign	Exponent	Mantissa	Meaning
0	255	Not equal to 0	Not a valid floating-point number (+NaN, Not a Number)
0	255	0	+Inf, Infinity
0	1 ... 254	Any	Positive, normalized floating-point number
0	0	Not equal to 0	Positive, denormalized floating-point number
0	0	0	+ zero
1	0	0	– zero
1	0	Not equal to 0	Negative, denormalized floating-point number
1	1 ... 254	Any	Negative, normalized floating-point number
1	255	0	– Inf, Infinity
1	255	Not equal to 0	Not a valid floating-point number (–NaN, Not a Number)

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

REAL data type

A distinction is made for the REAL tags between numbers which can be represented with full precision (“normalized” floating-point numbers) and numbers with a reduced precision (“denormalized” floating-point numbers).

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 $-1.175\,494 \times 10^{-38}$
± 0
$+1.175\,494 \times 10^{-38}$ to $+3.402\,823 \times 10^{+38}$

A denormalized floating-point number can be between the limits:

$-1.175\,494 \times 10^{-38}$ to $-1.401\,298 \times 10^{-45}$
and
$+1.401\,298 \times 10^{-45}$ to $+1.175\,494 \times 10^{-38}$

S7-300 CPUs cannot calculate with denormalized floating-point numbers. The bit pattern of a denormalized number is interpreted like a zero. If the result of a calculation is in this range, it is represented as zero, and status bits OV and OS are set (downward violation of number range).

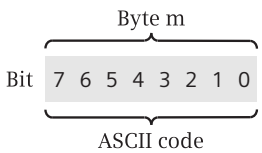
A tag with data type REAL consists internally of three components: the sign, the 8-bit exponent to base 2, and the 23-bit mantissa. The sign can have the values “0” (positive) or “1” (negative). The exponent is saved increased by a constant (bias, +127) so that it has a range of values from 0 to 255. 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. 4.16).

4.4.6 Data type CHAR

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

The notation shown in Fig. 4.17 is available for a number of special characters in association with an STL load statement. Example: L ‘$$’ loads a dollar sign in ASCII code. One, two or four characters can also be present for a load statement: The load statement L 'a' loads one character (in this case an a), L 'aa' loads two characters, and L 'aaaa' loads four characters right-justified into accumulator 1.

Data type CHAR



Special characters for CHAR

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

Fig. 4.17 Structure of the CHAR data type

4.4.7 Data types for durations and points in time

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).

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. Examples:

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

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. 4.18). Examples:

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

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). Individual time units can be omitted. If more than one time unit is specified, the values for the time units are limited: Days from 0 to 24, hours from 0 to 23, minutes and seconds from 0 to 59, and milliseconds from 0 to 999 (Fig. 4.18).

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

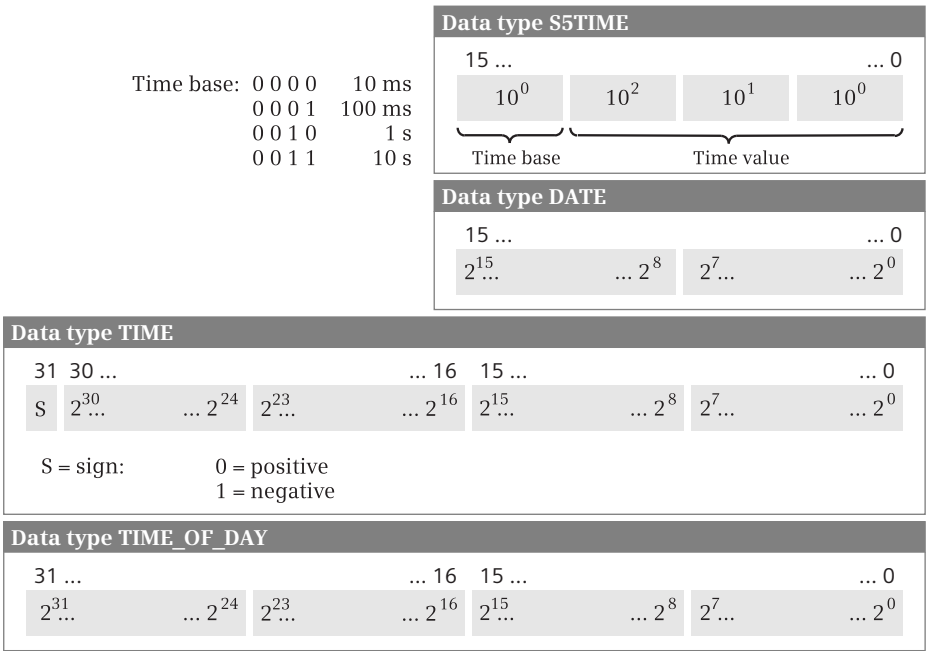


Fig. 4.18 Bit assignment of data types S5TIME, DATE, TIME, and TIME_OF_DAY

Data type TIME_OF_DAY

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 seconds, each separated by a colon. The specification of milliseconds, which follows the seconds and is separated by a dot, can be omitted (Fig. 4.18). Examples:

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

4.5 Complex data types

Complex data types consist of a combination of elementary data types under one name (Table 4.12). 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.

4.5.1 Data type DATE_AND_TIME

The data type DATE_AND_TIME represents a specific point in time consisting of a date and time. You can use the abbreviation DT instead of DATE_AND_TIME.

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. 4.19).

Table 4.12 Overview of complex data types

Data type	Length	Meaning, remark
DATE_AND_TIME	8 bytes	Date and time Example: DT#1990-01-01-00:00:00
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.

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	All data in BCD format
Byte n+5	10^1	10^0	Seconds	0 to 59	*) n = even
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	

Fig. 4.19 Structure of data type DATE_AND_TIME (DT)

4.5.2 Data type STRING

The data type STRING represents a string consisting of two bytes for the length data and up to 254 bytes for the characters in ASCII code. Saving in the memory commences at a byte with even address. The program editor reserves an even number of bytes for a string.

When declaring a STRING tag, its maximum length is defined in square brackets. The current length is entered for the default setting or when processing the string (the actually used length of the string = number of valid characters). The maximum

length is present in the first byte of the string, the second byte contains the current length; this is followed by the characters in ASCII format (Fig. 4.20).

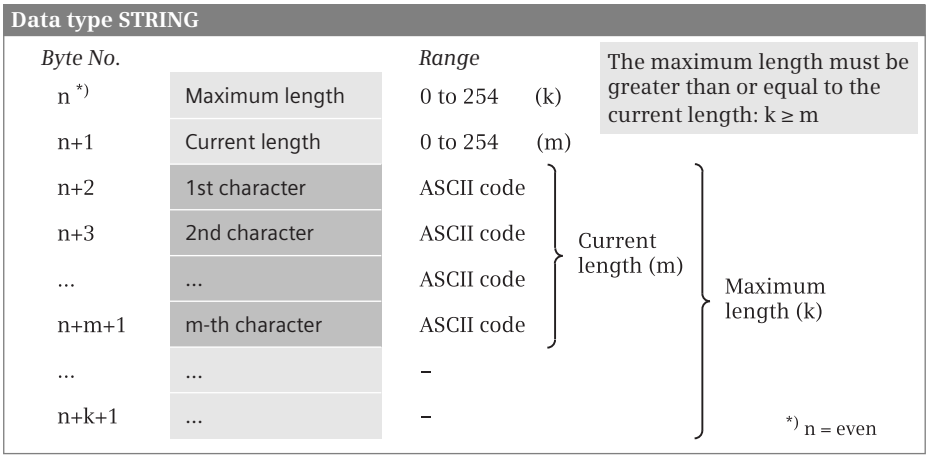


Fig. 4.20 Structure of STRING data type

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.

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

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

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

The characters in a STRING tag can also be addressed individually (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 'i'.

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 “Functions for strings” on page 526).

4.5.3 Data type ARRAY

The data type ARRAY represents a data structure comprising a fixed number of components with the same data type (Fig. 4.21). All data types are permissible except ARRAY.

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.

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

The index is a fixed value with LAD/FBD/STL and cannot be changed during runtime (variable indexing not possible). With SCL, the index can also be a tag or an expression with data type INT (see Chapter 4.3.5 “Indirect addressing with SCL” on page 118).

Example: A tag with 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 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.

Multi-dimensional arrays

Arrays 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. 4.21).

When addressing the components of multi-dimensional arrays, you must always specify the indices of all dimensions for LAD/FBD/STL.

Addressing of partial arrays is possible with SCL: 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.

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,#j].

The partial array #Array1[#i] can now be assigned to a correspondingly dimensioned array, e.g. #Array2 := #Array1[#i], where #i = 1 to 8 and #Array2 : ARRAY [1..16] OF INT.

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 bit-wide components

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

Array of byte-wide components

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

Array of word-wide or doubleword-wide components

*) 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]

2nd dimension
2nd dimension
2nd dimension
1st dimension

Example of the byte assignments of the *#Arraytag* tags with the data type ARRAY[1..2,1..3,1..2] OF BYTE

*) n = even

Fig. 4.21 Structure of data type ARRAY

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).

4.5.4 Data type STRUCT

The STRUCT data type represents a data structure comprising a fixed number of components with different data types (Fig. 4.22). All data types are permissible, including the PLC data types.

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.

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

Example: A tag with the name *Fan* is to comprise four components:
Switch_on_fan (BOOL), *Switch_off_fan* (BOOL), *Speed* (INT), and *Delay* (TIME).

```
Fan : STRUCT
   Switch_on_fan  : BOOL := FALSE
   Switch_off_fan : BOOL := FALSE
   Speed          : INT  := 0
   Delay          : TIME := T#0ms
```

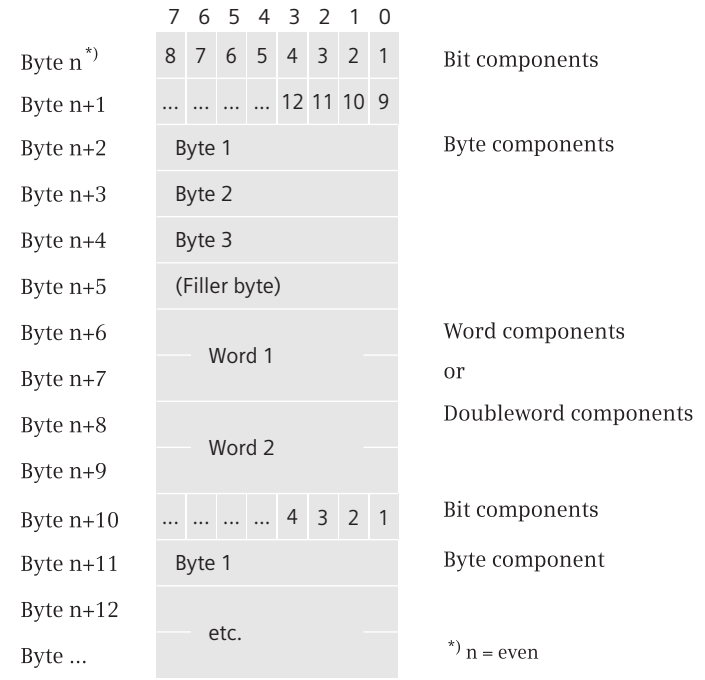
A component of a STRUCT tag can also be addressed individually by positioning the name of the structure, separated by a dot, in front of the component name. A STRUCT component can be handled like a tag with the same data type. For example, the component *#Fan.Speed* can be created on a block parameter 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 provided they have an elementary data type. The individual names are each separated by a dot.
Example: *StructureTag.Structure_Level2.Component_Level2*.

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

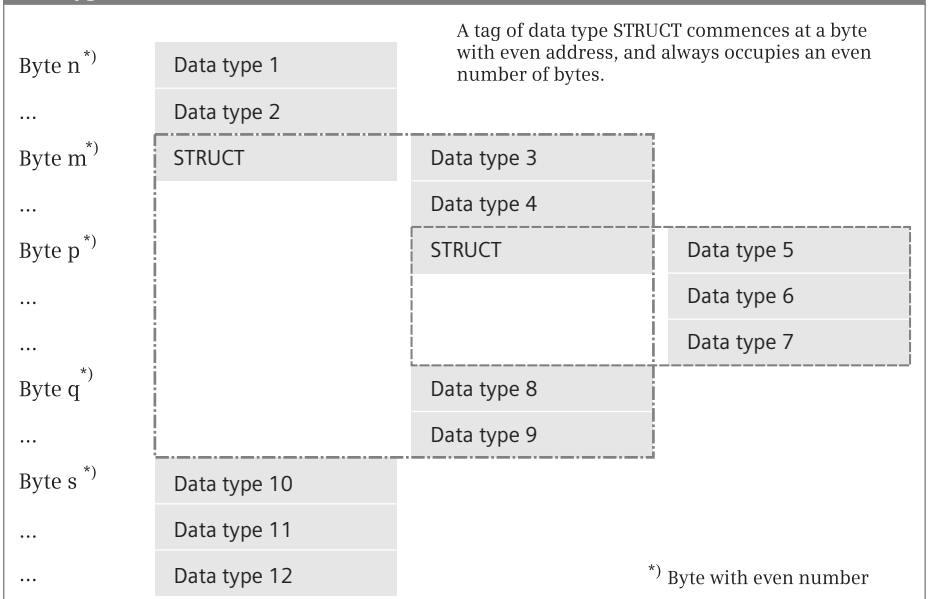


Fig. 4.22 Structure of STRUCT data type

4.6 Parameter types and pointers

4.6.1 Parameter types

The parameter types are additional data types for block parameters (Table 4.13). The length data in the table refers to the memory requirements of the block parameter when transferring to the called block.

Parameter types TIMER and COUNTER

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.

Parameter types BLOCK_xx

The blocks transferred at the block interface are of the parameter types BLOCK_xx (xx stands for the block type, see Table 4.13). Blocks transferred with BLOCK_FC or BLOCK_FB can only be processed to a limited extent in the called block (see Chapter 14.4.4 “Change to a block without block parameter” on page 551). The BLOCK_DB parameter type is of practical significance since it can be used to transfer and call data blocks as block parameters (see Chapter 14.5.1 “Open data block” on page 555).

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

Table 4.13 Overview of parameter types

Parameter type	Description		Examples of actual parameters
TIMER	SIMATIC timer function	16 bits	T 15 or name
COUNTER	SIMATIC counter function	16 bits	C 16 or name
BLOCK_DB	Data block	16 bits	DB 19 or name
BLOCK_FC	Function	16 bits	FC 17 or name
BLOCK_FB	Function block	16 bits	FB 18 or name
VOID	Function value without type	–	Without actual parameter (only with functions FC)
POINTER	DB pointer	48 bits	As pointer: P#M10.0 or P#DB20.DBX22.2 As operand: MW 20 or E 1.0 or #Name
ANY	ANY pointer	80 bits	As area: P#DB10.DBX0.0 WORD 20 or any (complete) tag

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 166.

Parameter type POINTER

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 section *Out*. The content of the block parameter is a DB pointer which points to the actual parameter to be transferred (see Chapter 4.6.2 “Pointer” on page 136).

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 section *Out*. The content of the block parameter is an ANY pointer which points to the actual parameter to be transferred (see Chapter 4.6.2 “Pointer” on page 136).

4.6.2 Pointer

Introduction

An address for indirect addressing must be formatted such that it contains the bit address, the byte address, and possibly also the operand area. It therefore has a special format referred to as pointer. A pointer is used to point as it were to an operand.

Three types of pointers are possible with STEP 7:

- Area pointers; these have a length of 32 bits and contain a specific operand or its address
- 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

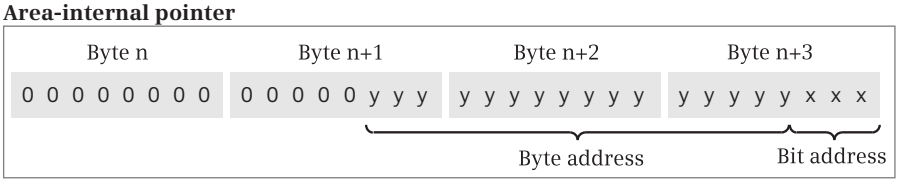
The area pointer is used in STL for indirect addressing, the DB pointer for a block parameter of the type POINTER, and the ANY pointer for a block parameter of the type ANY.

Area pointer

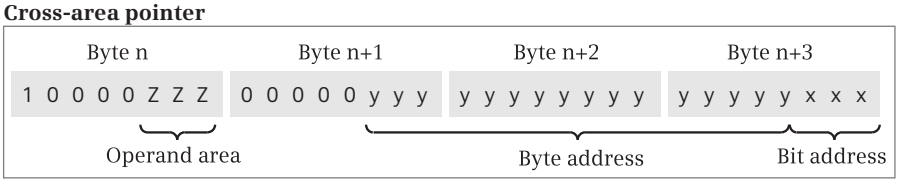
The area pointer contains the operand address and possibly also the operand area. Without an operand area, it is an *area-internal* pointer. If the pointer also contains the operand area, one refers to a *cross-area* pointer. The two types of pointer are distinguished by the assignment of bit 31 (Fig. 4.23).

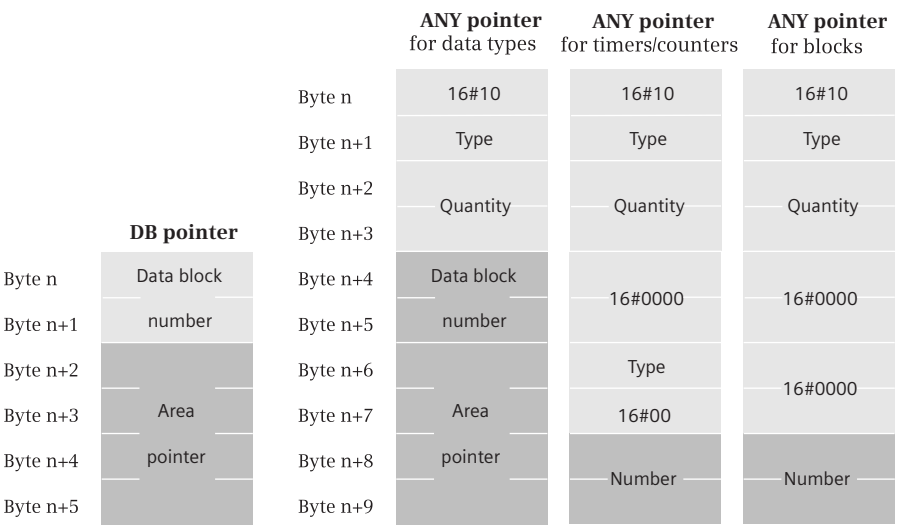
Pointers for indirect addressing

Area-internal pointer



Cross-area pointer





Operand area in the area pointer:

0	0	0	Peripherals (P)
0	0	1	Inputs (I)
0	1	0	Outputs (Q)
0	1	1	Bit memories (M)
1	0	0	Global data (DBX)
1	0	1	Instance data (DIX)
1	1	0	Temporary local data (L)[1)]
1	1	1	Temporary local data of preceding block (V) [2)]

1) Not with cross-area addressing
2) Only with transfer of block parameters

Type in the ANY pointer

Elementary data types		Compound data types
01	BOOL	0E DT
02	BYTE	13 STRING
03	CHAR	
04	WORD	**Parameter types**
05	INT	
06	DWORD	19 BLOCK_DB
07	DINT	1C COUNTER
08	REAL	1D TIMER
09	DATE	
0A	TOD	**Zero pointer**
0B	TIME	
0C	S5TIME	00 NIL

Fig. 4.23 Structure of pointers for indirect addressing

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). With the peripheral operand area (P), the operation (load or transfer) used during application of the pointer distinguishes whether the peripheral inputs (I:P) or the peripheral outputs (Q:P) are addressed.

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.

DB pointer

A DB pointer also contains, supplementary to the area pointer, a data block number as a positive integer. 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. 4.23).

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.

Example: P#DB10.DBX20.5

You cannot load this pointer using a load statement; however, you can apply it to a block parameter of the type POINTER (not with SCL). The POINTER parameter type has the structure of a DB pointer.

ANY pointer

Supplementary to the DB pointer, the ANY pointer also contains the data type and a repetition factor. It is then possible to point to a data area (addressed absolutely) in addition to tags.

You can apply an ANY pointer to a block parameter of the type ANY (not with SCL). The ANY parameter type has the structure of an ANY pointer.

The ANY pointer is available in two versions: for tags with data types and for tags with parameter types. Tags with elementary data types, DT, and STRING are assigned the type shown in Fig. 4.23 and the quantity 1.

If you apply a tag with the data types ARRAY, STRUCT, or a PLC data type to an ANY parameter, the program editor generates an ANY pointer to the data array or the data structure. This ANY pointer contains the ID for BYTE (B#16#02) as the type, and as quantity the number of bytes of the tag length. The data type of the individual array or structure components is insignificant. Thus an ANY pointer points to a

WORD array with double the number of bytes. Exception: A pointer to an array of components with data type CHAR is also applied with CHAR type (B#16#03).

The representation of a constant for tags with data types or for data areas is as follows: P#[Data_block.]Operand Type Quantity

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

With parameter types you write the pointer: L# Number Type Quantity

Examples:

- L# 10 TIMER 1 Timer function %T10
- L# 2 COUNTER 1 Counter function %C2

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 for data types must always be a bit address.

You can also use the parameter type ANY to declare tags in the temporary local data. You use these tags to generate an ANY pointer which can be changed during runtime (“variable” ANY pointer).

4.6.3 “Variable” ANY pointer with STL

An ANY pointer in the representation of a constant *P#Operand Type Quantity* points to a data area with fixed address. If a tag is applied to an ANY block parameter, for example a tag with the data type ARRAY, the program editor generates a constant ANY pointer to this tag. In neither case is it possible to change or redefine 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 data in the user program prior to its use.

Like with indirect addressing, the addresses are only known during runtime, and the danger therefore exists that memory areas can be overwritten unintentionally. *The automation system could then react in an unexpected manner! Therefore be extremely careful when using the “variable” ANY pointer!*

Fig. 4.24 shows an example of application of the “variable” ANY pointer. It contains the program Function (FC) for transfer of variable data areas for a function (FC) which transfers a data area from one data block to another, where the address and length of the area can be changed during runtime. Writing of the ANY tag is programmed in two different manners depending on whether the address of the ANY

tag is known in the temporary local data or not. When calling the BLKMOV function, its function value is transferred to the function value of the FC so that this is provided with the error information of the BLKMOV function.

The “variable” ANY pointer can be changed as desired. For example, the operand area can be set such that the transfer is to or from the memory area.

Note: If the ANY pointer present in the temporary local data points to a tag which is also located in the temporary local data of the calling block, V must be entered as the operand area since, from the viewpoint of the called block, this tag is in the temporary local data of the preceding block.

4.6.4 “Variable” ANY pointer with SCL

Temporary local data of the type ANY can save the address of an operand or of a global or block-local tag:

```
var_any := %MW10;
var_any := #Setpoint;
var_any := "Data".Array1;
var_any := P#P0.0 VOID 0;
```

The pointer P#P0.0 VOID 0 contains only bits occupied by "0" and is a pointer "to nothing".

You can directly process the individual components of an ANY pointer with SCL, for example the data block number or the operand ID, by overlaying a data type (see section “Overlaying tags (data type views with SCL)” on page 121).

Example: In a function (FC) with the name “Copy”, two ANY pointers *#QANY* and *#ZANY* are put together from individual block parameters (Fig. 4.25).

4.7 PLC data types

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

Programming a PLC data type

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

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,

Example: Function (FC) for transfer of variable data areas

```
Input
   QDB  : INT //Source DB
   QANF : INT //Source start address
   ANZB : INT //Number of bytes
   ZDB  : INT //Destination DB
   ZANF : INT //Destination start address

Temp
   QANY : ANY //Pointer to source
   ZANY : ANY  //Pointer to destination

Return
   RET_VAL : INT
```

Declaration of input parameters in the declaration section *Input*.

Declaration of pointer tags used in the declaration section *Temp*.

Declaration of the function value

```
//Pointer for the source area
L     W#16#1002
T     %LW0
L     #ANZB
T     %LW2
L     #QDB
T     %LW4
L     #QANF
SLD   3
OD    DW#16#8400_0000
T     %LD6
```

The pointer is written with the source data area.

In the example, the pointer address is assumed to be known: It commences at byte 0 in the temporary local data, and is therefore the address of the *#QANY* tag.

```
//Pointer for the destination area
LAR1  P##ZANY
L     W#16#1002
T     LW[AR1,P#0.0]
L     #ANZB
T     LW[AR1,P#2.0]
L     #ZDB
T     LW[AR1,P#4.0]
L     #ZANF
SLD   3
OD    DW#16#8400_0000
T     LD[AR1,P#6.0]
```

The pointer is written with the destination data area.

In the example, the pointer address is assumed to be unknown and is first loaded into address register AR1. Addressing of the components of the *#ZANY* tag is then carried out via address register AR1.

```
CALL  BLKMOV
   Any
   SRCBLK  := #QANY
   RET_VAL := #RET_VAL
   DSTBLK  := #ZANY
```

Application of ANY tag as actual parameter for the system function BLKMOV.

Fig. 4.24 Example of a “variable” ANY pointer with STL

Example: Variable ANY pointer with SCL

```
Input                                  //Declaration of block parameters
   QDB  : INT                          //Source data block
   QANF : INT                          //Source start address
   ANZB : INT                          //Number of bytes
   ZDB  : INT                          //Destination data block
   ZANF : INT                          //Destination start address

Temp                                   //Pointer to temporary local data
   QANY              : ANY             //Source pointer
   QPointer   AT QANY: STRUCT          //Overlaying with components
     Type            : WORD
     Number          : INT
     DBNo            : INT
     Area            : DWORD

   ZANY              : ANY             //Destination pointer
   ZPointer  AT ZANY : STRUCT          //Overlaying with components
     Type            : WORD
     Number          : INT
     DBNo            : INT
     Area            : DWORD

Return
   Copy              : INT             //Function value
```

```
                                       //Compile source pointers
#QPointer.Type   := W#16#1002;
#QPointer.Number := #ANZB;
#QPointer.DBNo   := #QDB;
#QPointer.Area   := SHL(IN := #QANF, N := 3) OR DW#16#8400_0000;
```

```
                                       //Compile destination pointers
#ZPointer.Type   := W#16#1002;
#ZPointer.Number := #ANZB;
#ZPointer.DBNo   := #ZDB;
#ZPointer.Area   := SHL(IN := #ZANF, N := 3) OR DW#16#8400_0000;
```

```
                                       //Copy with BLKMOV
#Copy := BLKMOV (                      //The block has the name
   SRCBLK := #QANY,                    //"Copy". The function value
   DSTBLK := #ZANY);                   //is the error information
                                       //from BLKMOV
```

Fig. 4.25 Example of a "variable" ANY pointer with SCL

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.

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*.

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.

The PLC data types of the opened PLC station can be compared to the PLC data types of another station, e.g. from a reference project. To start the comparison, select the PLC station in the project tree or select the *Compare > Offline/offline* command from the shortcut menu and move the second PLC station to the right side of the compare editor. The compare editor then indicates, for example, whether the PLC data types of both stations agree.

4.8 Start information

When an organization block is called, the CPU's operating system transfers start information in the temporary local data. 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 names and comments are in English, but 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 4.14 can be found as a basic framework in all organization blocks. Deviating assignments for individual organization blocks are described in Chapters 5.4 “Startup program” on page 171, 5.5 “Main program” on page 176, 5.6 “Interrupt processing” on page 186, and 5.7 “Error handling” on page 200.

Table 4.14 Structure of the start information

Byte	Data type	Structure element	Meaning, remark
0	BYTE	EV_CLASS	Bits 0 to 3: Event identifier Bits 4 to 7: Event class
1	BYTE	EV_NUM	Event number
2	BYTE	PRIORITY	Priority class, number of execution level
3	BYTE	NUM	OB number
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	Event time	Beginning of event

5 Program execution

5.1 Operating states of the CPU

A CPU 300 recognizes the following operating states:

- Deenergized, when the power supply is switched off
- 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
- HOLD, when a user program with breakpoints is being tested
- Faulty, if an internal error prevents further execution

Fig. 5.1 illustrates the operating state transitions: After switching on ①, the CPU is in the STOP operating state. If the mode switch on the front panel of the CPU is at RUN, the CPU changes to the STARTUP operating state ② and then to the RUN operating state ③. If a "serious" error occurs during execution in STARTUP or RUN, or if you set the mode switch to STOP, the CPU returns to the STOP operating state ④ ⑤.

In the HOLD operating state, you can test the user program with breakpoints in single step mode. You can access this operating state either from RUN or STARTUP and return to the original mode when you exit testing ⑥ ⑦. You can also switch the CPU immediately to STOP from the HOLD operating state ⑧.

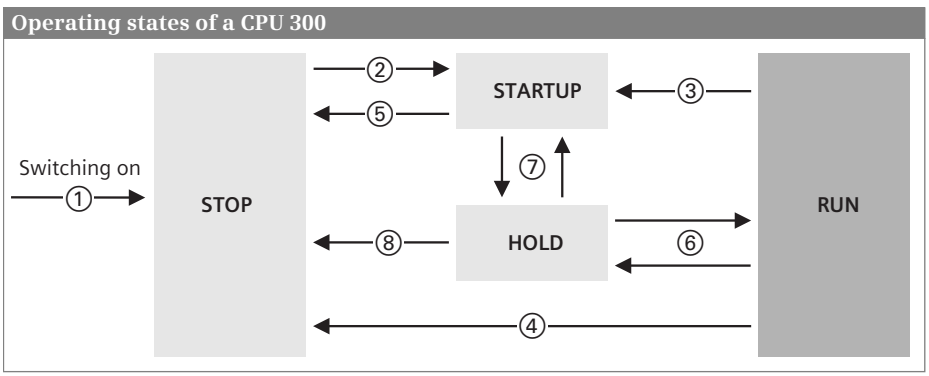


Fig. 5.1 Operating states and operating state transitions of a CPU 300

The STOP operating state has the highest priority, followed by HOLD, STARTUP, and RUN with descending priority. For example, if a stop request comes from the programming device while the mode switch is at RUN, the CPU switches to the STOP operating state.

5.1.1 STOP operating state

The STOP operating state is reached

- when the CPU is switched on,
- after changing the mode switch from RUN to STOP,
- if a “serious” error occurs during program execution,
- if the STP system block is executed, and
- following a request from a communication function (stop request from programming device or by communication function blocks of a different CPU).

The CPU enters the cause of the STOP operating state into the diagnostic 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. The real-time clock is running.

You can parameterize the CPU in the STOP operating state, for example set the MPI 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, HOLD, and STARTUP operating states (OD or BASP signal, output disable or disable command output). Disabled output modules output a zero signal or – if configured accordingly – the substitute value.

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

In the STOP operating state, you can cancel disabling of the output modules for testing purposes (see Chapter 15.5.7 “Monitoring and modifying in the STOP operating state” on page 602).

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.

A CPU 300 carries out a warm restart in the STARTUP operating state.

Warm restart startup mode

A *manual warm restart* is triggered

- by the mode switch on the central processing unit on a transition from STOP to RUN, or
- by a communication function from a programming device or another CPU; 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. The automatic warm restart is carried out if

- the CPU is not at STOP when the voltage is switched off and the mode switch is at RUN, or
- the CPU is interrupted during a warm restart by a power failure.

Warm restart is the fixed setting in the properties of a CPU 300 for *Startup after POWER ON*.

CPU activities upon warm restart

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 reset or retain their last value depending on the parameterization (Fig. 5.2). This is followed by disabling of the peripheral outputs by the OD signal (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 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.

The modules are parameterized as was defined by the hardware configuration.

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.

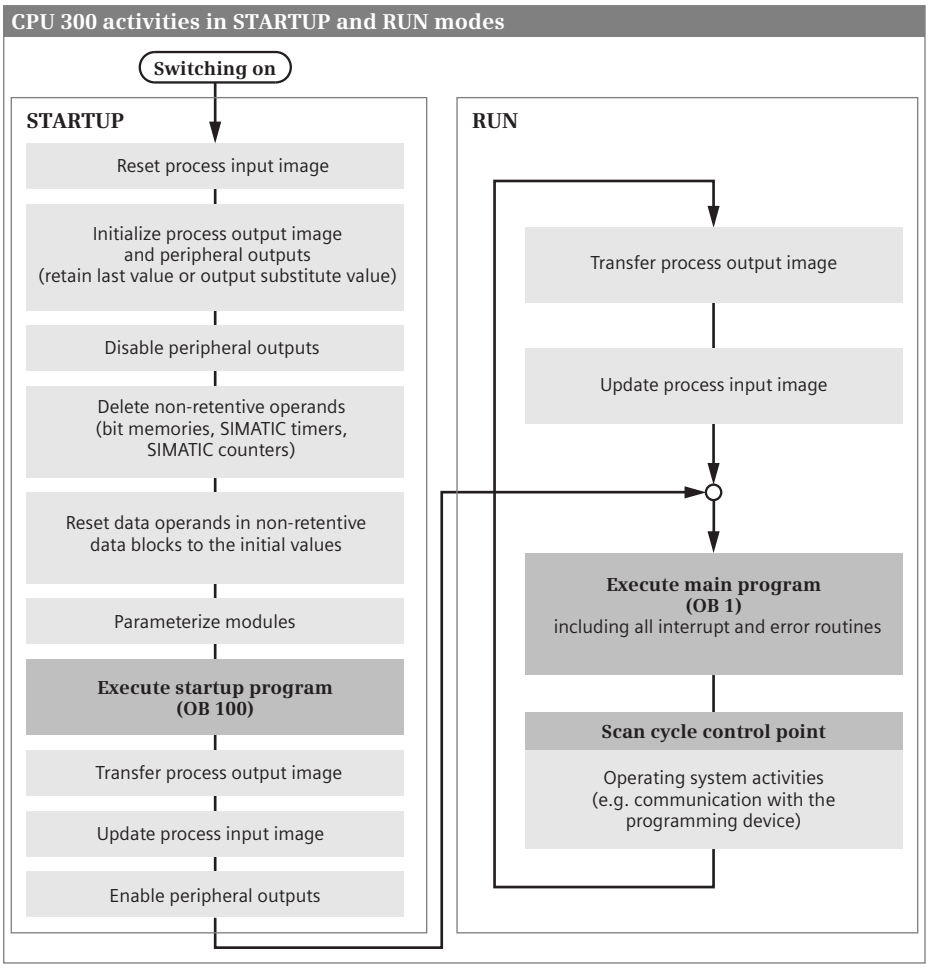


Fig. 5.2 CPU activities in the STARTUP and RUN operating states

If the startup organization block OB 100 is present, it is called once (see Chapter 5.4 “Startup program” on page 171). 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.

Execution of a start-up routine can be aborted, e.g. by repositioning the mode switch or by a power supply failure. The aborted start-up routine is then executed from the beginning when the CPU is switched on.

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):

- Transmission of process image output to the output modules
- Updating of process image input
- Execution of main program, including interrupt and error programs
- Communication with the programming device and with other stations on the so-called cycle control point.

The main program is present in organization block OB 1 and the blocks called within it.

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 5.5 “Main program” on page 176 (including process images, cycle time, response time, time-of-day), in Chapter 5.6 “Interrupt processing” on page 186 (time-delay interrupts, cyclic interrupts, and hardware interrupts), and in Chapter 5.7 “Error handling” on page 200 (including OB 82 *diagnostics interrupt* and OB 80 *time error*).

5.1.4 HOLD operating state

The CPU enters the HOLD operating state if you test the program with breakpoints in single-step mode. The STOP LED is then lit, and the RUN LED flashes.

The output modules are disabled when in the HOLD operating state, and therefore a zero signal or – if configured accordingly – the parameterized substitute value is output.

All time procedures are stopped by the operating system when at HOLD. This refers, for example, to the execution of SIMATIC timer functions, clock memories, runtime meters, cycle time monitoring, processing of cyclic and time-delay interrupts. Exception: The real-time clock continues as usual.

With each progression by one statement in test mode, the times for the duration of the single step continue a little, and thus simulate a time response based on the “normal” program execution.

In the HOLD operating state, the CPU has passive communication capability, i.e. it can participate in one-way data exchange, for example.

5.1.5 Reset CPU memory

A memory reset returns the CPU to its “initial state”. You can only trigger the memory reset using a connected programming device when in the STOP operating state or by using the mode switch: Hold switch in MRES position for at least 3 s, release, and then at the latest within 3 s hold in the MRES position again for at least 3 s.

The CPU deletes the complete user program present in the work memory and the operands present in the system memory independent of the retentivity setting. The contents of the load memory on the Micro Memory Card are retained during the memory reset.

The CPU resets the parameters of all modules – including its own – to the standard settings. The MPI parameters of the first interface are an exception. These are not changed, and therefore a CPU for which a memory reset has been carried out remains addressable on the MPI bus. The diagnostic buffer, the real-time clock, and the runtime meters are not reset either.

The CPU imports the configuration data present in the load memory on the Micro Memory Card, uses it to set the module parameters, and copies the parts of the user program relevant to execution from the load memory into the work memory.

5.1.6 Restoring the factory settings

You can restore the factory settings with a “Reset to factory settings”. Proceed as follows:

- ▷ Switch off the power supply and remove the Micro Memory Card.
- ▷ Hold the mode switch in the MRES position and switch the power supply on again.
- ▷ Once the SF, FRCE, RUN and STOP LEDs flash slowly, release the mode switch, set it to MRES again within 3 s, and hold it in this position.
- ▷ Wait until only the SF LED flashes. During this period (approx. 5 s), you can abort the reset procedure by releasing the mode switch.
- ▷ Once the SF LED shows a continuous light, release the mode switch.

The CPU starts up and all status LEDs light up. It executes a memory reset, then sets the MPI address to 2 and the MPI baud rate to 187.5 Kbit/s. As well as the memory reset, the real-time clock is set to the start date (DT#1990-01-01-00:00:00.000) and the runtime meters and diagnostic buffer are deleted.

Finally, the CPU enters the “Reset to factory settings” event into the diagnostic buffer, and goes to the STOP operating state.

5.1.7 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.

Retentive memory areas can be bit memories, SIMATIC timers, SIMATIC counters, and data blocks. These are saved in the retentive memory where they are retentive even without battery backup.

The maximum length of the retentive areas is CPU-specific. You define the number of retentive memory bytes, timers, and counters with the hardware configuration in the CPU properties under *Retentive memory*.

You define the retentivity of a data block – independent of its type – in the declaration table in the *Retain* column. Only the complete data block can be set retentive or non-retentive.

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.

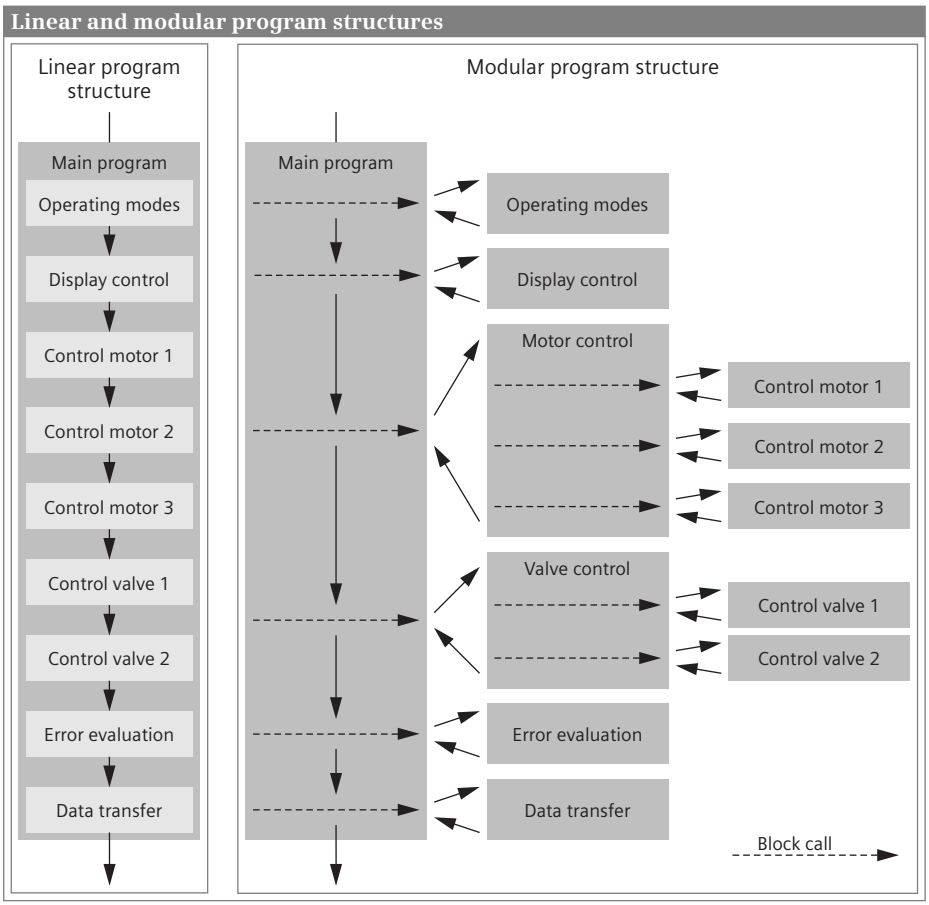


Fig. 5.3 Comparison between linear and modular program structures

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 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.

The example also shows the use of different types of block (further information on the types of block can be found in Chapter 5.2.3 "Block types" on page 156). 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 in-

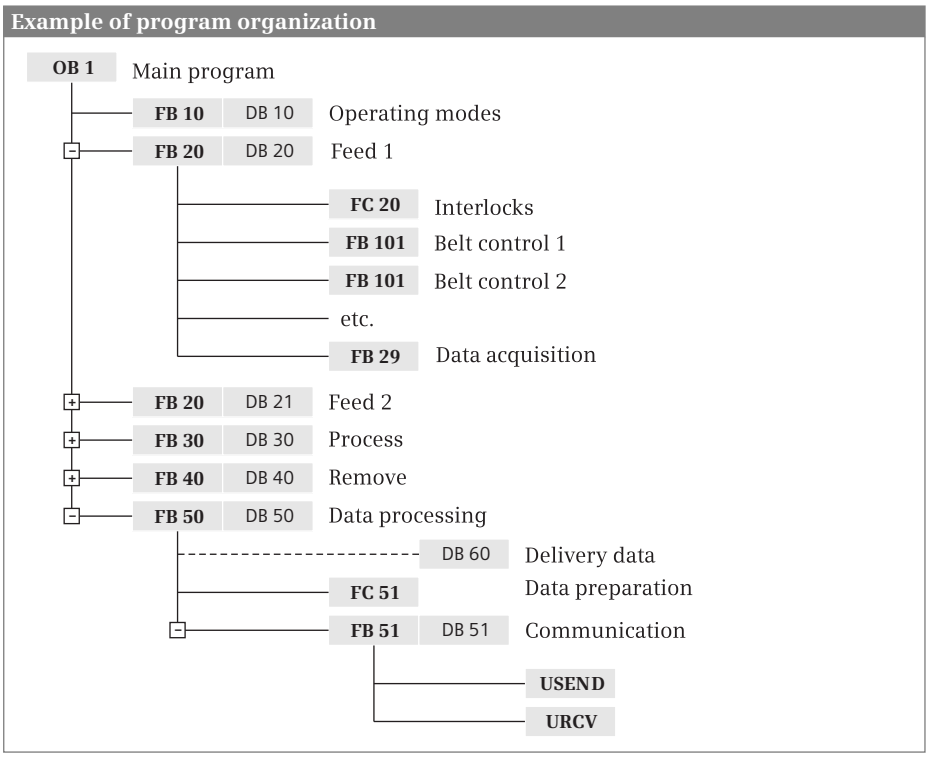


Fig. 5.4 Example of program organization

stance 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 8 in the startup program, in the main program, and in an interrupt routine, and 4 in an error program. If more blocks are called in the "horizontal" level, the CPU generates 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. The opening of a data block does not generate a new call level either.

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 Micro 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.

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 state-

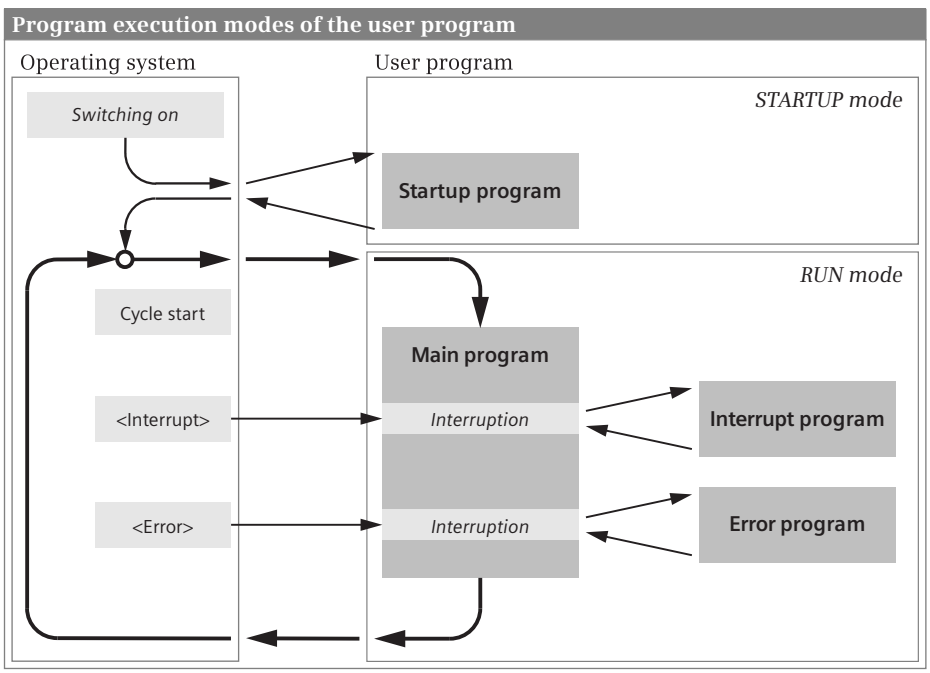


Fig. 5.5 Program execution modes of a SIMATIC user program

ment; 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. The quantity of organization blocks, together with their numbers and assignments to events, are fixed for a CPU 300. Each controller type has a certain range of organization blocks.

Program execution commences in the CPU with the **startup program**. A startup can be triggered by switching on the power supply or by an operator input on a connected programming device. The startup program is optional. If you wish to create a startup program, use the organization block OB 100 for a CPU 300. Further code blocks can be called in the OB 100. Following execution of the startup program, the CPU commences with execution of the main program.

The **main program** is present as standard in organization block OB 1, which is always executed by the CPU. The start of the program is identical to the first statement in OB 1. Further code blocks can be called in the OB 1. The main program comprises the program in OB 1 and the programs in all blocks called in OB 1.

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), it calls OB 1 again.

Events that can interrupt the main program are **interrupts** and **errors**. Interrupts have their origin in the controlled plant (hardware interrupts), in the CPU (time-delay interrupts and cyclic interrupts), or on the modules (DPV1 interrupts, for example).

A distinction is made between asynchronous and synchronous errors. An asynchronous error is one which is independent of the program execution, such as a power supply failure in an expansion unit. A synchronous error is one caused by the program execution, for example the addressing of a non-existent operand or an error during conversion of a data type.

5.2.3 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

certain events occur. The organization blocks have a fixed number corresponding to the call event, for example the main program is in organization block OB 1. The organization blocks provide so-called start information when called, and this 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 a global data block, as an instance data block, or as a type data block. With a global data block, you program the data tags directly in the data block. With an instance data block, the programming of the assigned function block determines the data tags present in the data block. A type data block has the structure of a PLC data type.

The number of organization blocks and their block numbers are defined 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.

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.

You handle system functions in the user program exactly like functions (FC), and system function blocks exactly like function blocks (FB).

System data blocks contain configuration data, for example module parameters. These blocks are generated and managed by STEP 7 itself. You can only read and write the contents of system data blocks in special cases, for example when "reparameterizing" modules using 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 instructions 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 instruction, 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.

5.2.4 Editing block properties

To display and change the block properties, select the block in the project tree and then the *Properties* command in the shortcut menu. Fig. 5.6 shows as example for the block properties the sections *General* and *Information* of a function block.

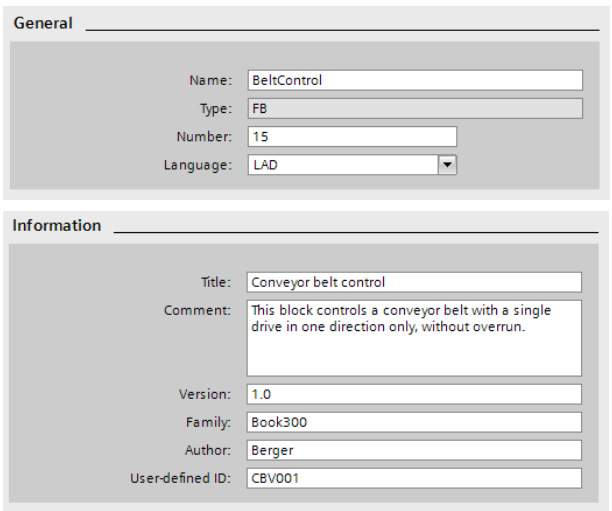


Fig. 5.6 Block properties: *General* and *Information* tabs

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 a PLC tag, a constant, a PLC data type, or another 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. 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, for data blocks DB. In the case of a function block with sequential control (GRAPH), you also set the programming language in the networks here (LAD or FBD).

With data blocks, the designation *DB* together with the type of data block is present in the *Type* field: *Global DB* in the case of a global data block, *Instance DB of <FB_name>* in the case of an instance data block of the function block *<FB_name>*, and *Data block derived from <Type_name>* if the structure of the data block is based on the data type *<Type_name>*.

The **Information** section contains the *Title* and the *Comment*; these are identical to the block title and the block comment which you can enter when programming the block 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 enter the author 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).

The time data in the **Time stamps** section indicates 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 scope of temporary local data and the memory requirements of the block in the load and work memories.

The **Protection** section indicates the block protection. A block can be protected so that the program can no longer be read out (know-how protection). In the case of a protected block, *Block is protected* is present here. For more information, refer to section “Configuring know-how protection” on page 161.

Attributes

Table 5.1 lists the block attributes for code blocks with LAD, FBD or STL programs and for data blocks. The *IEC check* attribute is present for each code block. Additional attributes for compilation of SCL blocks are described in Chapter 6.5.2 “Compiling SCL blocks” on page 241. 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 419.

The *IEC check* attribute indicates how strict the data type test is to be in the code block. With the attribute not activated, it is usually sufficient if the tags used have the data width required for execution of the function or instruction; with the attribute activated, the data types of the tags must correspond to the required data types.

Table 5.1 Block attributes

Attribute	For block	Meaning with attribute activated
IEC check	OB, FB, FC	A stricter test of the data types takes place.
Multiple instance capability	FB	The function block can be called in another function block as a local instance.
Data block write-protected in the device	Global DB, type DB	The data of the data block cannot be overwritten by means of a program during runtime.
Only store in load memory	Global DB, type DB	The data block is not transferred to the work memory; it is only present in the load memory.

The *Multiple instance capability* attribute is only present with function blocks. If the *Multiple instance capability* attribute is activated (this is the standard setting), you can call the block as a local instance in another function block. If the function block is generated via an external source file with the keyword CODE_VERSION1, the *Multiple instance capability* attribute can be activated or deactivated. The advantage of a "not capable of multi-instance" function block is – since address register AR2 is not used – the unlimited use of instance data for indirect addressing, which is only of significance with STL programming.

Data block write-protected in the device is an attribute only 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 *Data block write-protected in the device* attribute is switched off as standard, but this can be changed at any time using the program editor. The keyword for programming with a source file for switching on the write protection is READ_ONLY.

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 Micro 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 are suitable for data which is only accessed rarely, e.g. recipes or archives. This attribute is switched off as standard and can be changed at any time using the program editor. The keyword for programming with a source file is UNLINKED.

The "Retentive" property with data blocks is not set using a block attribute but in the declaration table in the *Retain* column. The keyword for programming via a source file for non-retentive data blocks is NON_RETAIN.

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.

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 (thus overwriting the offline version!) cannot be opened, not even with the correct password.

5.2.5 Block interface

Components of the block interface

The block interface contains the declarations of the local tags that are used solely within the block. 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.

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 complex data types TIMER, COUNTER, BLOCK_xx, POINTER, ANY FB: STRING of adjustable length FC: STRING with standard length 254	FC and FB
Output	Output parameters may only be written in the program of the block Elementary and complex data types FB: STRING of adjustable length FC: STRING with standard length 254, POINTER, ANY	FC and FB
InOut	In/out parameters may be read and written in the program of the block Elementary and complex data types STRING with standard length 254, POINTER, ANY	FC and FB
Temp	Temporary local data may be read and written in the program of the block, are only valid during the current block processing Elementary and complex data types, STRING of adjustable length, BLOCK_xx, POINTER, ANY with special function	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 complex data types, STRING of adjustable length	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 and complex data types, STRING of adjustable length, ANY, VOID	FC

Input parameters

An input parameter transfers a value to the program in the block and may only be read. 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.

Blocks can also be transferred at the interface: A function block with BLOCK_FB, a function with BLOCK_FC, and a data block with BLOCK_DB. The transferred code blocks must not have any parameters themselves.

Output parameters

An output parameter transfers a value to the calling block and may only be written. 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/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.

5.2.6 Example of use of block parameters

By means of block parameters you enable parameterization of the processing specification (the block function) present in a block.

The example shows an adder with three summands which can be used repeatedly in the user program with different tags. The tags are transferred as block parameters – in our example, three input parameters and one output parameter. Since the adder need not permanently save values internally, a function FC is suitable as the block type (Fig. 5.7).

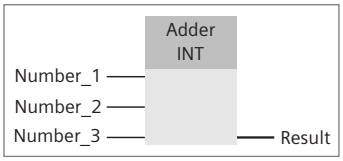
The values to be transferred are declared as input parameters in the *Input* section with name and data type, the calculated value as an output parameter in the *Output* section, also with name and data type. If the program is written in LAD or FBD in the block, another tag is required as intermediate memory. This is declared in the *Temp* section, since its value is not required outside the block. A tag for intermediate storage is not required with an STL or SCL program.

The program in the block can be written in the language with which the block function is best mapped, independent of the programming language with which the block is subsequently called. The block parameters used in the block program are called *formal parameters*. They are handled like tags which have the same data type.

The "Adder" function can then be called repeatedly in the user program. Different values are transferred to the adder at the block parameters with each call. These values can be constants, operands, or tags; they are referred to as *actual parameters*. During runtime, the control processor replaces the formal parameters by the actual parameters.

Program in the called block "Adder"

Template



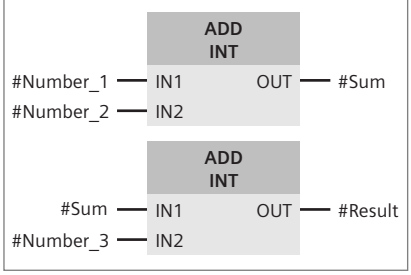
Block interface

Input	Number_1	INT
	Number_2	INT
	Number_3	INT
Output	Result	INT
Temp	Sum	INT

The block interface contains the input and output parameters and a temporary tag which is used with LAD and FBD as an intermediate memory in the program. This can be omitted with STL and SCL.

Program

Graphic representation (LAD, FBD)



SCL

```
#Result := #Number_1 + #Number_2 + #Number_3;
```

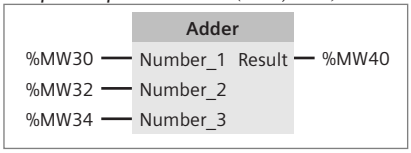
STL

```
L    #Number_1
L    #Number_2
+I
L    #Number_3
+I
T    #Result
```

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 in the block interface as local tags.

Call of block "Adder"

Graphic representation (LAD, FBD)



During the block call, the block parameters are supplied with tags which are referred to as **actual parameters**.
The called block works with the values of the actual parameters.

SCL

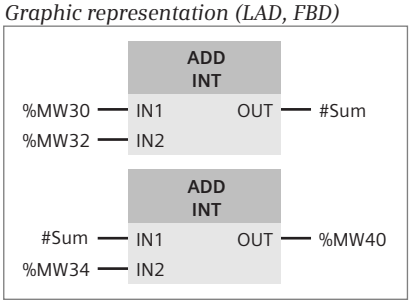
```
"Adder"      (Number_1 := %MW30,
              Number_2  := %MW32,
              Number_3  := %Mw34,
              Result => %MW40);
```

STL

```
CALL "Adder"
     Number_1      := %MW30
     Number_2      := %MW32
     Number_3      := %MW34
     Result        := %MW40
```

Executed program in the block "Adder"

Graphic representation (LAD, FBD)



SCL

```
%MW40 := %MW30 + %MW32 + %MW34;
```

STL

```
L    %MW30
L    %MW32
+I
L    %MW34
+I
T    %MW40
```

Fig. 5.7 Block call with block parameters

5.3 Calling blocks

5.3.1 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 FBD and LAD, 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. With SCL, the enable input EN and the enable output ENO are parameters which are implicitly present. With STL, this "EN/ENO mechanism" can be mapped using STL statements.

A 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" (see Chapter 6.6.5 "Consistency check" on page 247).

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.

5.3.2 Calling functions (FC)

When calling a function, all block parameters must be supplied with actual operands, i.e. you must connect operands or tags to all block inputs and outputs. You can supply EN and ENO as required with the graphic programming languages.

Fig. 5.8 shows an example of calling a function (FC) in the various programming languages. The block parameter *Result* is configured as an output parameter; the function has no function value.

Supplying the block parameters

You can use tags from the inputs, outputs, and bit memories operand areas for all block parameters. Constants and peripheral inputs are only permissible for input parameters, peripheral outputs only for output parameters. When using data tags, you are strongly recommended to use complete addressing (*"Data_block".Data_tag*) since the programming interface does not indicate which data block has been currently opened by the program editor.

The data type of the actual parameter must correspond to the data type of the block parameter. The data types must agree exactly if the *IEC check* attribute is activated in the calling block, otherwise matching widths of the data type or operand are usually sufficient.

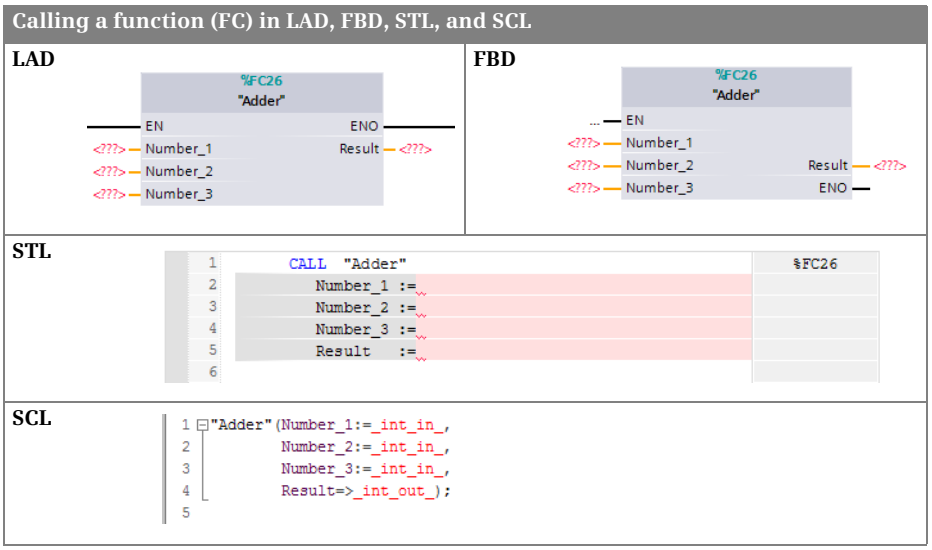


Fig. 5.8 Calling a function (FC) in the various programming languages

Any maximum length of an actual parameter is possible for a block parameter with data type STRING. Note that an actual parameter with data type STRING which has been declared in the temporary local data cannot be assigned a default value and therefore has any content. It must be provided with meaningful values before being used as an actual parameter.

Tags with elementary data type and pointers (e.g. P#DB10.DBX20.5) are permissible on block parameters with parameter type POINTER. These can also be completely addressed data tags or components of an array or data structure.

Tags of all data types are approved for block parameters with 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 Chapters 4.6.3 ““Variable” ANY pointer with STL” on page 139 and 4.6.4 ““Variable” ANY pointer with SCL” on page 140).

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 shown in LAD, FBD, and STL as the first output parameter and it is also handled like an output parameter.

SCL handles a function with function value like a tag with the data type of the function value. Fig. 5.9 shows an example: The function “Adder2” adds three numbers

Declaration of the function value in the “Adder2” function

Block interface

Input	Number_1	INT
	Number_2	INT
	Number_3	INT
Return	**Result**	**INT**

The block interface contains the three input parameters and the function value as the result of the addition
#Result := #Number_1 + # Number_2 + # Number_3;

The declaration as function value has no effects for the program in the "Adder2" block. The program could also have been created using one of the programming languages LAD, FBD or STL.

Call of the “Adder2” block in the SCL program

SCL

```
//Call within an expression
#Tag_1 := "Adder2" (Number_1 := %MW30, Number_2 := %MW32, Number_3 := %MW34) + #Tag_2;
```

The "Adder2" function can now be used in an expression in the programming language SCL.
The function has the data type which was assigned to the function value in the RETURN declaration.

Fig. 5.9 Use of the function value with SCL

and returns the total as a function value with data type INT. The total can be directly processed further in an expression.

5.3.3 Calling function blocks (FB)

When calling a function block, you are requested to specify the storage location of the instance data. This is the data with which the function block works internally: the block parameters and the static local data.

Specify a data block if the call takes place in an organization block or a function. The call then takes place as a “single instance”, and the data block is the instance data block for this call. If you call the function block as a single instance for a second time, enter a different data block as the instance data block. This then contains the data for the second call. Assign a separate data block to each call of a function block as single instance.

Fig. 5.10 shows an example of calling a function block (FB) as a single instance in the various programming languages. The function block in the example is named *“Motor”*; the instance data block assigned to the call is named *“Motor_DB”*.

When calling a function block with “multi-instance capability”, you can choose the following: You can call the function block as a “single instance” or as a “local instance” (“multi-instance”). With a single instance, the call is assigned a separate data block as instance data block. When calling a local instance, the called function block stores its instance data in the instance data block of the calling function block. You then specify the name with which the local instance can be addressed in the static local data of the calling function block. You can repeatedly call a function block as a local instance using different names in each case.

Fig. 5.11 shows an example of calling a function block (FB) as a local instance in the various programming languages. The function block in the example is named "*Motor*", the assigned instance data is named *#Motor_Instance*.

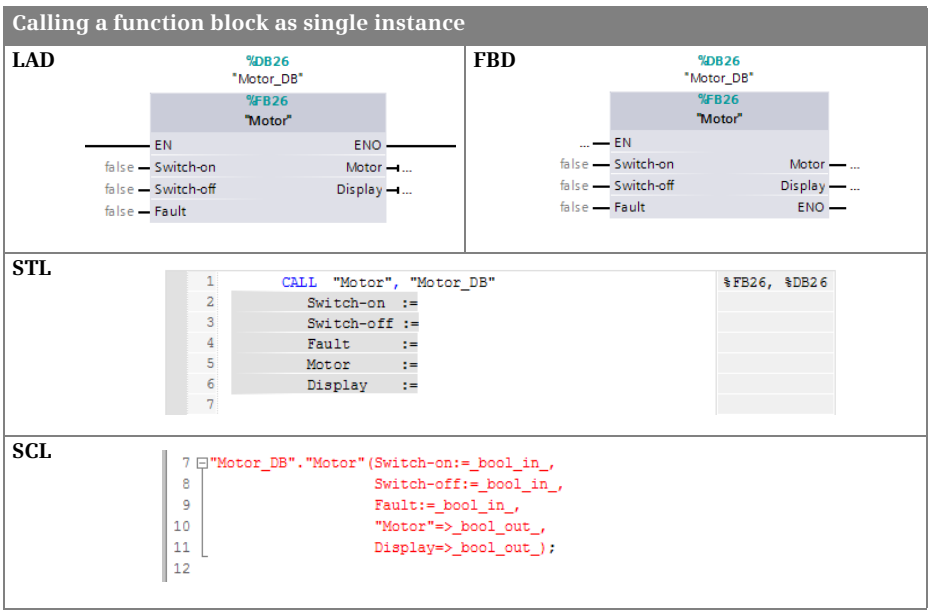


Fig. 5.10 Calling a function block as single instance

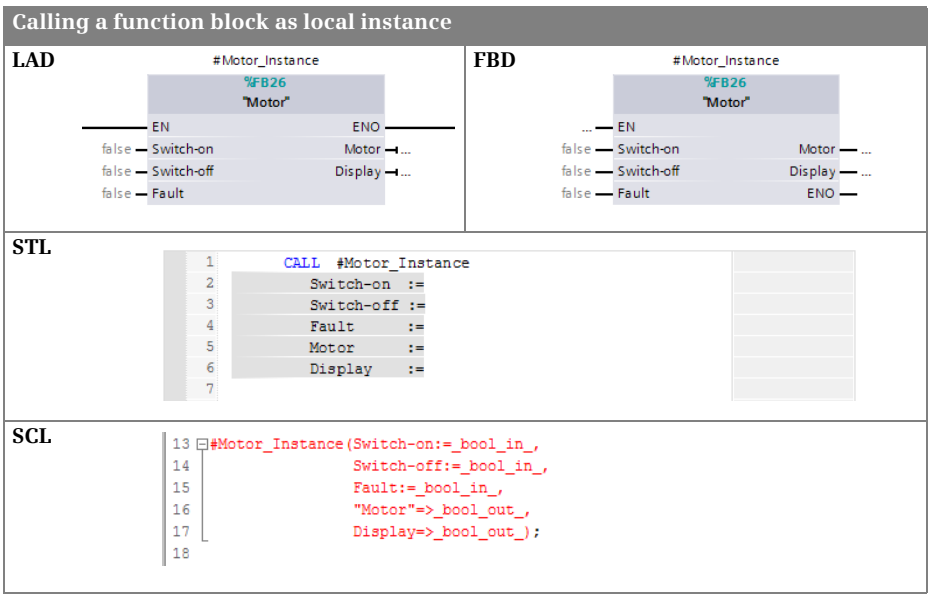


Fig. 5.11 Calling a function block as local instance

If a function block is not capable of multi-instance – the *Multiple instance capability* attribute is not activated – it can only be called as a single instance. A function block with multi-instance capability can be called as a single instance and as a local instance. A function block without multi-instance capability can contain local instances.

Supplying the block parameters

The block parameters of a function block are located in the instance data. Therefore not all block parameters have to be supplied when calling the function block. If the supply is omitted, the function block works with the "old" values from its last call or with the default settings. In/out parameters with complex data type are an exception; these must be supplied during the first call so that a valid pointer is entered into the instance data. You can supply EN and ENO as required with the graphic programming languages.

You can use tags from the inputs, outputs, and bit memories operand areas for all block parameters. Constants and peripheral inputs are only permissible for input parameters, peripheral outputs only for output parameters. When using data tags, you are strongly recommended to use complete addressing (*"Data_block".Data_tag*) since the programming interface does not indicate which data block has been currently opened by the program editor.

The data type of the actual parameter must correspond to the data type of the block parameter. The data types must agree exactly if the *IEC check* attribute is activated in the calling block, otherwise matching widths of the data type or operand are usually sufficient.

An input or output parameter of a function block with data type STRING can only be supplied with STRING tags whose maximum length corresponds to that of the block parameter. Any maximum length of the STRING tag is possible on an in/out parameter. Note that an actual parameter with data type STRING which has been declared in the temporary local data cannot be assigned a default value and therefore has any content. It must be provided with meaningful values before being used as an actual parameter.

Tags with elementary data type and pointers (e.g. P#DB10.DBX20.5) are permissible on block parameters with parameter type POINTER. These can also be completely addressed data tags or components of an array or data structure.

Tags of all data types are approved for block parameters with 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 Chapters 4.6.3 ""Variable" ANY pointer with STL" on page 139 and 4.6.4 ""Variable" ANY pointer with SCL" on page 140).

Chapter 18.5.6 “Data storage of a local instance in a multi-instance” on page 718 describes how the block parameters and the static local data are saved when calling as a local instance in a multi-instance.

“External” access to local data

The block parameters of a function block are located in a data block. If a block parameter is saved as a value (not as a pointer), you can address it from any position in the user program like a global data tag. The address for a single instance is *“Data_block”.Parameter_name* and for a local instance *“Data_block”.Instance_name. Parameter_name*.

Block parameters with data types POINTER and ANY as well as in/out parameters with complex data types are saved as pointers.

5.3.4 “Passing on” of block parameters

The “passing on” of block parameters is a special form of access and supply of block parameters. The parameters of the calling block are “passed on” to the parameters of the called block. In this case, the formal parameter of the calling block is then the actual parameter of the called block.

It always applies here that the actual and formal parameters must be of the same type, i.e. the associated block parameters must agree with regard to their data types. Note in this context that the maximum length may have to be considered with data type STRING.

It additionally applies that you can only connect an input parameter of the calling block to an input parameter of the called block, and an output parameter only to an output parameter. You can connect an in/out parameter of the calling block to all declaration types of the called block.

Exceptions: Complex data types in input and output parameters can only be passed on if the calling block is a function block. Block parameters with parameter types TIMER, COUNTER and BLOCK_xx can only be passed on by an input parameter to another input parameter if the called block is a function block.

The “passing on” of block parameters also applies in the same manner to instructions (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.

5.4 Startup program

A CPU 300 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 147.

5.4.1 Organization block OB 100

The startup program is present in organization block OB 100 and the blocks called within it. A startup program is not essential. If a startup program is not required, simply omit the organization block OB 100.

The startup program can have any length. There is no time limit for executing the startup program; the cycle time monitoring is not active. Application examples include the parameterization of modules if the parameter settings made by the CPU are to be changed and the programming of default settings for the main program.

Start information

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), the start information of the OB 100 contains the tags

OB100_STOP	Number of the STOP event
OB100_STRT_INFO	Additional information for the current startup

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 OB 100 in the STEP 7 help). With this information you can create an event-dependent startup program.

5.4.2 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.

You can use the following system blocks to determine the geographic address from the logical address, and vice versa:

- GEO_LOG Determine logical base address (SFC 70)
- GADR_LGC Determine logical address of a module channel (SFC 5)
- RD_LGADR Determine all logical addresses of a module (SFC 50)
- LOG_GEO Determine geographic address (SFC 71)
- LGC_GADR Determine slot address of a module (SFC 49)

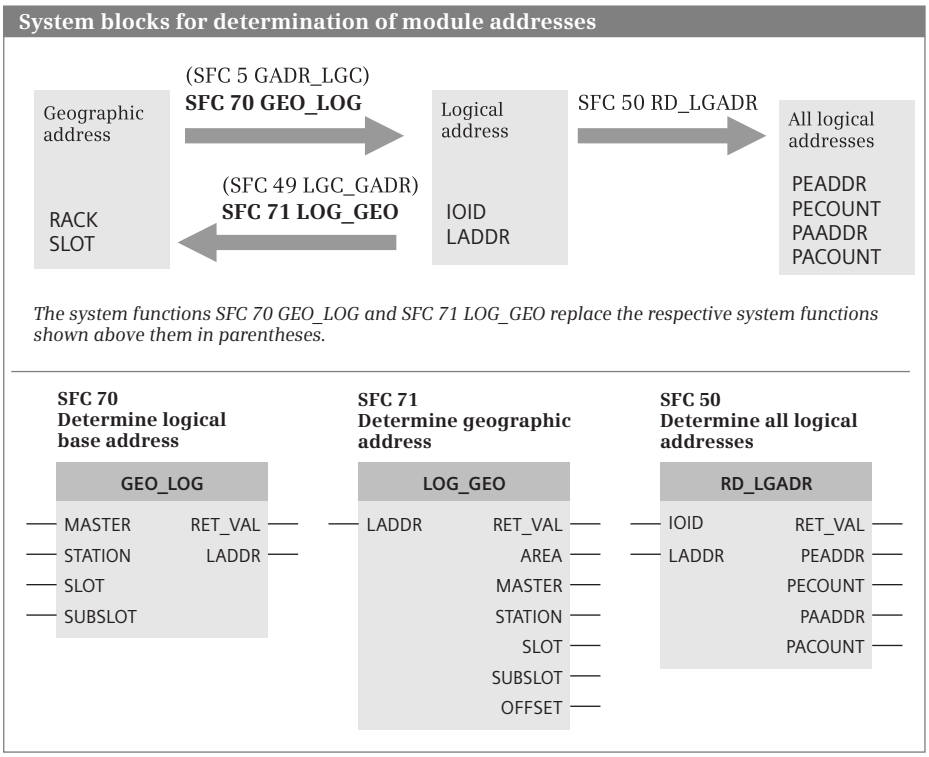


Fig. 5.12 Determining logical and geographic module addresses

Fig. 5.12 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.

The system function GEO_LOG supersedes the system function GADR_LGC, and the system function LOG_GEO supersedes the system function LGC_GADR. GEO_LOG and LOG_GEO have an extended functionality, e.g. they can also be used in the distributed I/O in conjunction with PROFINET IO. The replaced system blocks will therefore not be described below.

GEO_LOG Determine logical start address

The system function GEO_LOG delivers the logical base address of a module or station. The assignment of the MASTER parameter indicates whether the station or module is inserted in a rack (central design) or whether the station is operated in a distributed PROFIBUS or PROFINET system.

Specify the slot number in the rack or station in the SLOT parameter, and the number of the submodule in the SUBSLOT parameter. The LADDR parameter then delivers the base address of the submodule. The assignment of bit 15 decides whether the address is assigned to an input (= 0) or output (= 1). With SUBSLOT = 0, the diagnostic address of the module or station is delivered.

LOG_GEO Determine geographic address

The system function LOG_GEO delivers the geographic address of a module or station if you specify the logical base address for it in the LADDR parameter. The assignment of bit 15 decides whether the address is assigned to an input (= 0) or output (= 1).

The value in the AREA parameter specifies the system in which the module is used (1 = S7-300, 2 = distributed I/O).

RD_LGADR Determine all logical addresses of a module

The system function RD_LGADR returns all logical addresses of a module if any address from the user data range is specified for it in the IOID and LADDR parameters. IOID is either B#16#54 (corresponds to the inputs) or B#16#55 (for the outputs).

Connect an area comprising WORD components (a word-by-word ANY pointer, e.g. P#DBzDBXy.x WORD nnn) to the PEADDR and PAADDR parameters. The system function RD_LGADR then shows the number of entries returned in these areas in the PECOUNT and PACOUNT parameters.

5.4.3 Parameterization of modules

Most S7 modules can be parameterized, i.e. values can be set on the module which are different from the default settings. To set the parameters, open the module in the hardware configuration and complete the tabs in the displayed dialog. When started, the CPU automatically transfers the module parameters to the modules and for the distributed I/O following the “return” of a station.

Static and dynamic module parameters

Module parameters are distinguished by static and dynamic properties. You can set both types of parameter offline in the hardware configuration. You can also change the dynamic parameters by calling a system block during runtime. 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.

Asynchronous processing of system blocks

The system blocks for module parameterization and transmission of data records work “asynchronously”. This means that the result of the block function is not immediately available following processing 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 for the system functions is in the RET_VAL parameter and for the system function blocks in bytes 2 and 3 of the STATUS parameter.

A certain job for a module is specified by the module start address 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.

An error which occurs when triggering a job is signaled by the error information and BUSY remains “0”.

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 an undesirable delay in the cycle processing time and thus in the response time, and the cycle monitoring time may then be triggered.

System blocks for module parameterization

The following system blocks are available with a CPU 300 for module parameterization:

- WR_PARM Write dynamic parameters (SFC 55)
- WR_DPARM Write predefined parameters (SFC 56)
- PARM_MOD Assign module parameters (SFC 57)
- WR_REC Write data record (SFC 58)
- RD_REC Read data record (SFC 59)
- RD_DPARA Read predefined parameters (SFC 102)
- RD_DPAR Read predefined parameters (SFB 81)
- RDREC Read data record (SFB 52)
- WRREC Write data record (SFB 53)

Fig. 5.13 shows a graphic representation of the available system blocks. The highlighted system blocks supersede the system blocks above them in parentheses. Some have an extended functionality, e.g. they can also be used in the distributed I/O in conjunction with PROFINET IO. The replaced system blocks will therefore not be described below.

Module and data record addressing

You use the module start address for transmission of data records. In the case of hybrid modules which have input and output areas, you use the lower area start address. If the input and output areas have the same start address, use the ID for an input address. You use the I/O ID irrespective of whether you wish to carry out a read or write operation.

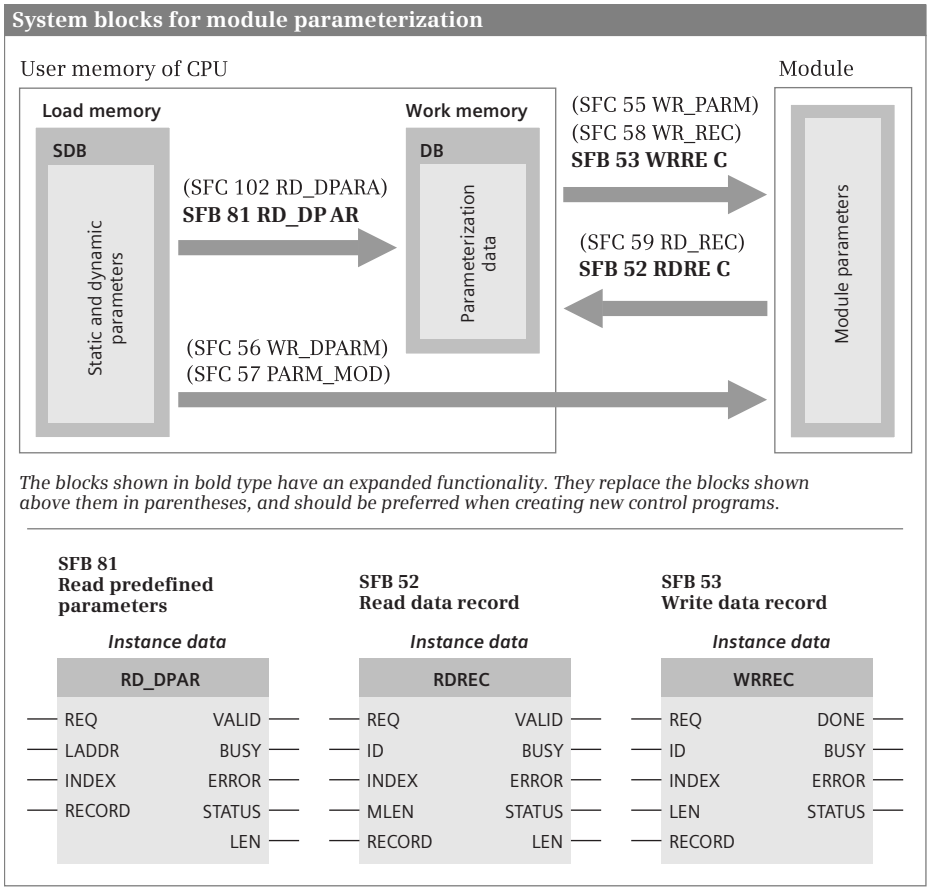


Fig. 5.13 System blocks for module parameterization and transmission of data records

Parameterization of the module start address is carried out using the ID or LADDR parameter. The assignment of bit 15 determines whether it is an input (= 0) or output (= 1).

You apply an actual parameter which corresponds to an area of BYTE components to the RECORD parameter with data type ANY. This can be a tag with data type ARRAY, STRUCT or with a PLC data type, or a byte-serial ANY pointer (for example P#DBzDBXy.x BYTE nnn). If you use a tag, it can only be a "complete" tag; individual array or structure components are not permissible.

RD_DPAR Read predefined parameters

The system function block RD_DPAR transfers the data record with the number specified in the INDEX parameter from the corresponding system data block SDB to the destination area specified in the RECORD parameter.

The transmission is carried out asynchronously and can be divided between several program cycles; the BUSY parameter has signal state “1” during the transmission. Following a successful transmission, the VALID parameter has signal state “1” and the number of data bytes transferred is present in the LEN parameter.

The read data record can then be evaluated, for example, or modified and written by the WRREC system block to the module.

RDREC Read data record

With “1” in the REQ parameter, the system function block RDREC reads the data record INDEX from the module and saves it in the destination area RECORD. The destination area must have the same length as the data record, or longer. The MLEN parameter specifies how many bytes are to be read.

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

With “1” in the REQ parameter, the system function block WRREC writes the data record INDEX from the source area RECORD to the module. The LEN parameter specifies how many bytes are to be written.

The 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.

5.5 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.

5.5.1 Organization block OB 1

The main program is present in organization block OB 1 and the blocks called within it. It executes at the lowest priority level and can be interrupted by all other types of program execution.

The length of the main program is limited by the available memory space. The time available for execution of the main program with all interrupt events occurring in the current processing cycle is limited by the cycle time monitoring (see Chapter 5.5.3 “Cycle time and response time” on page 178).

Prior to commencement of a new processing cycle – quasi at the start of the main program – the process image of the outputs is transferred to the I/O modules and the process image of the inputs is updated (see Chapter 5.5.2 “Process image updating” on page 177).

Execution of the main program can be interrupted by interrupt or error events. The corresponding organization blocks are then called and processed. Following processing of such an interruption, processing is continued in the main program at the point of interruption (see Chapters 5.6 “Interrupt processing” on page 186 and 5.7 “Error handling” on page 200).

Start information

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), the start information of the OB 1 contains the tags

OB1_PREV_CYCLE	Runtime of previous cycle (in ms)
OB1_MIN_CYCLE	Minimum cycle time since the last startup (in ms)
OB1_MAX_CYCLE	Maximum cycle time since the last startup (in ms)

Using these tags you can determine the current cycle time and its fluctuation per program.

5.5.2 Process image updating

The process image is part of the CPU's internal system memory. The process image consists of the process image input (operand area “Inputs I”) and the process image output (operand area “Outputs Q”). It commences at address 0 (zero) and ends at a CPU-dependent upper limit.

Following a CPU restart and prior to initial execution of the main program, the operating system transfers the signal states of the process image output to the output modules, and accepts the signal states of the input modules into the process image input. This is followed by execution of the main program where the signal states of the inputs are combined with each other and the outputs are controlled. Following termination of the main program, a new cycle begins with updating of the process image (Fig. 5.14).

If an error occurs during automatic updating of the process image, e.g. because a module can no longer be addressed, the organization block OB 85 *Program execution error* is called. If OB 85 is not present, the CPU switches to STOP.

Adjustable size of process image

The memory areas for the process images – the operand areas “Inputs” and “Outputs” – are always present in their full length and can be addressed by all functions and statements.

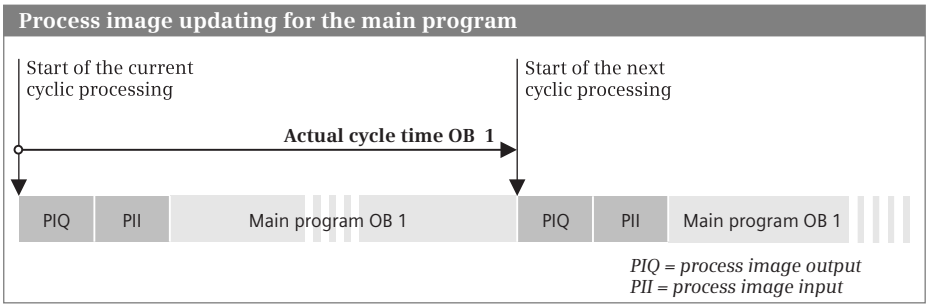


Fig. 5.14 Process image updating

With automatic process image updating, only the existing input and output modules are considered whose addresses are within the set range. In the properties of a CPU, under *Cycle* and *Size of process image of input* or *Size of process image of output* you can set how many bytes starting at byte 0 are used by the automatic updating (Table 5.3). The smaller this area can be kept, for example by assigning the input and output addresses without gaps starting at byte 0, the smaller is the cycle (processing) time.

Process image partition

A CPU 300, which supports isochronous mode, has the process image partition 1 (PIP 1). This process image partition is updated by system blocks in the isochronous mode program OB 61. Further details on isochronous mode can be found in Chapter 5.6.8 “Isochronous mode interrupt, organization block OB 61” on page 197 and in Chapter 16.4.5 „Special functions for PROFIBUS DP“ in the section “Constant bus cycle time” on page 645.

5.5.3 Cycle time and response time

Cycle monitoring time

Processing of the main program with regard to timing is carried out by means of the so-called *Cycle monitoring time*. The default value for the monitoring time is 150 ms. You can change this value within the range from 1 ms to 6 s when parameterizing the CPU.

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, the CPU switches to STOP.

The cycle processing time comprises:

- The complete execution time of the main program (execution time of program in OB 1)
- The processing times for higher priority classes which interrupt the main program (in the current cycle)
- The time required to update the process images
- The time for communication processes by the operating system, e.g. access operations of programming devices to the CPU (program status!)

Table 5.3 Size of the process image for a standard CPU 300 (in bytes)

Process image updating		CPU 312	CPU 314 IM 151	CPU 315 IM 154	CPU 317	CPU 319
Inputs	Preset Adjustable up to	128 1024	128 1024	128 2048	256 8192	256 8192
Outputs	Preset Adjustable up to	128 1024	128 1024	128 2048	256 8192	256 8192

Cycle statistics

If you are connected online with the programming device to 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 organization block OB 1.

Communication load

The CPU's operating system requires a certain time for communication with the programming device or with other stations. In the CPU properties, you can set the percentage of the cycle time which is to be available for communication tasks. If you set a high percentage, it may be necessary to adapt the cycle monitoring time. 20% is set by default.

Independent of this setting, the CPU carries out communication tasks every 100 ms. This should guarantee that the CPU can still be accessed and switched to STOP, for example, in the event of an endless loop with restarting of the cycle monitoring time.

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 program execu-

tion time (the cycle time). The response time lies between one and two cycle times, as demonstrated in the following example.

If a limit switch is activated, for example, it changes its signal state from “0” to “1”. The PLC detects this change during subsequent updating of the process image and sets the input allocated to the limit switch to “1”. The program evaluates this change by resetting an output, for example in order to switch off the corresponding drive. The new signal state of the output that was reset is transferred at the end of program execution; only then is the corresponding bit reset on the digital output module.

In a best-case situation, the process image is updated immediately following the change in the limit switch's signal (Fig. 5.15). 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.

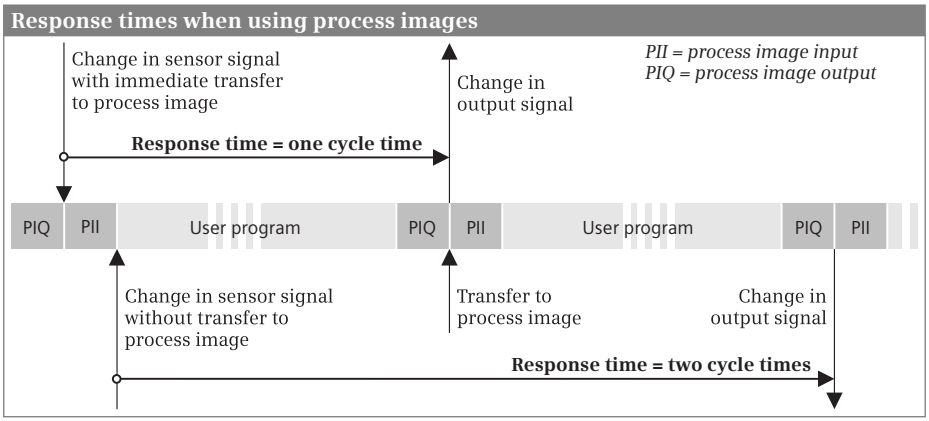


Fig. 5.15 Response times of PLCs

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 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 PROFIBUS DP master system also results in calculable response times.

RE_TRIGR Restart cycle monitoring time

RE_TRIGR restarts the cycle monitoring time. This then starts with the value set during CPU parameterization. RE_TRIGR does not have any parameters (Fig. 5.16).

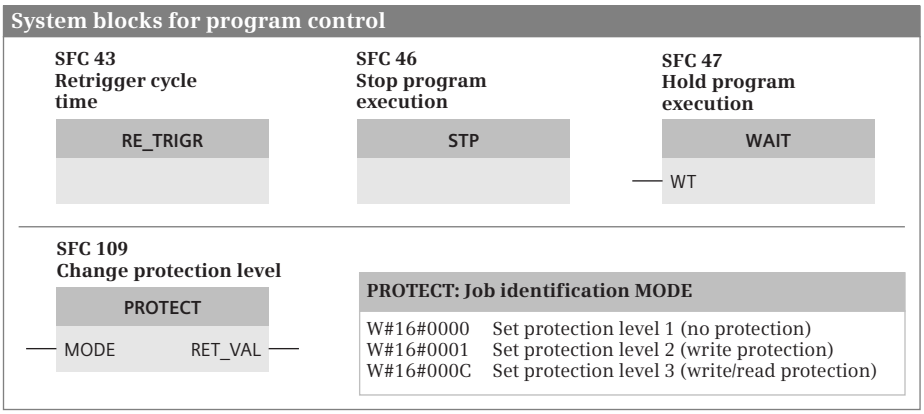


Fig. 5.16 System blocks for program control during runtime

The RE_TRIGR function is only effective when called in the main program. The cycle monitoring time is not restarted by a call in the startup program or in an interrupt routine, and ENO has the signal state “0”.

5.5.4 Hold, stop, and protect program

WAIT Hold program execution

The system function WAIT holds program execution for a defined duration (Fig. 5.16).

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. With a CPU 300, the hold time is then extended by the execution time of the interrupt program of higher priority.

STP Stop program execution

The system function STP terminates program execution; the CPU then switches to the STOP operating state. STP does not have any parameters (Fig. 5.16).

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: *Keep last value* or *Substitute a 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.

PROTECT Change program protection

The user program in a CPU can be protected against access in three protection levels: no protection, write protection, and read/write protection. You can set the protection level when parameterizing the CPU.

Program-driven toggling between protection levels “No protection” and “Write protection” or “Write/Read protection” is possible with the system function PROTECT (Fig. 5.16). Calling the system function PROTECT is only effective if you have set the protection level “No protection” with the hardware configuration. It has no effect if write protection or write/read protection is set.

If you have set the protection level “No protection” with the hardware configuration and activated the option *Can be canceled with password*, you can cancel the protection levels “Write protection” and “Read/write protection” by entering the correct password.

The protection level set with PROTECT remains unchanged if

- the CPU goes to STOP due to a (program) error, an STP call, or operator intervention, or
- after switching the power supply off and on again.

In all other cases, the protection level “No protection” is set in the case of an operating mode transition. Even if you switch the mode switch to STOP, protection level “No protection” is (re)set.

During runtime, you can scan the current protection level with system function RDSYSST via the system state partial list W#16#0232 with the index W#16#0004.

5.5.5 Time

A CPU 300 has a buffered real-time clock which can continue operating for approximately 6 weeks without a power supply (exception: CPU 312 has a non-buffered software clock). When the power supply is switched on after the buffering period, the clock continues with the time it had when switched off. The time is updated every 10 ms with a CPU 300. The accuracy is typically 2 s per day and can be corrected by a factor.

The clock can be synchronized and can be set or queried using a programming device or system blocks. The time is represented in the user program in DATE_AND_TIME format, thus comprising the date, time, and day of week.

Setting and reading the time in the user program

You can set and read the CPU clock using the following system functions:

- WR_SYS_T Set time and date (SFC 0)
- RD_SYS_T Read time and date (SFC 1)

Fig. 5.17 shows the graphic representation of the system functions.

The **WR_SYS_T** function sets the clock in the CPU to the value specified by the IN parameter. If a master clock is parameterized in the CPU – for example at the MPI – the time synchronization is started in addition. The error information is output in the RET_VAL parameter (0 = no error).

The **RD_SYS_T** function reads the current time of the CPU and outputs it in the OUT parameter. The error information is output in the RET_VAL parameter (0 = no error).

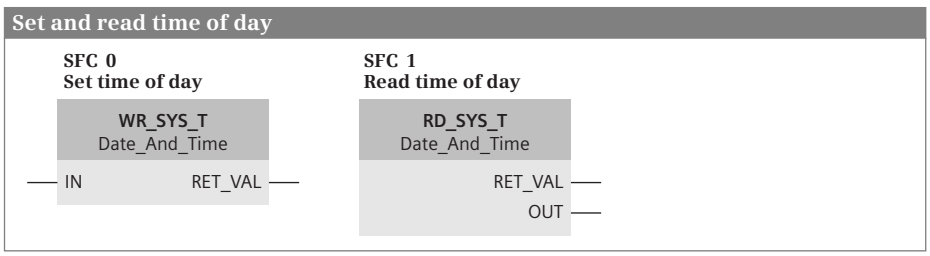


Fig. 5.17 System blocks for time control

Master clock and slave clock

Time synchronization is possible over the MPI and – if present – over the DP and PROFINET interfaces. The time can also be synchronized with correspondingly designed central FM or CP modules.

The CPU can be the master clock or slave clock. The setting as slave clock is only possible on one interface. The CPU then receives the synchronization frame over this interface, which it can be passed on via the other interfaces. These interfaces are then set as master clock.

There can only be one master clock within a subnet.

A CPU – as the master clock – also sends a synchronization frame if the time is set, e.g. with the system function WR_SYS_T. Note that the clock is in the default setting when delivered or following a firmware update of the CPU and must first be set in order to be able to send a synchronization frame.

Configuration of time synchronization

You configure the settings for time synchronization in the properties of the CPU during hardware configuration.

If time synchronization is to be carried out on modules within the station, open the *Time of day* section in the CPU properties and set the synchronization mode *As master* under *Synchronization on PLC*. Select the time interval from a drop-down list (in the range from 1 second to 24 hours).

If time synchronization is to be carried out over an interface, open the *Time synchronization* section in the interface properties and set the synchronization mode. When setting as master clock, select the time interval from a drop-down list (in the range from 1 second to 24 hours). The PROFINET interface can only be a slave clock.

Correction factor

You use the correction factor to correct any variation in the time occurring within a 24 hour period. Set a negative correction factor if the clock is fast. You set the correction factor in the hardware configuration in the *Time of day* section in the properties of the CPU.

5.5.6 Read system time

The system time of a CPU is a milliseconds counter which is updated every 10 ms with a CPU 300. 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 or HOLD.

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

The TIME_TCK system function reads the current system time. The RET_VAL parameter contains the read system time in the TIME data format. Fig. 5.18 shows the graphic representation of the system function.

5.5.7 Runtime meter

An runtime meter counts the hours while running. You can use the runtime meter, for example, to record the CPU runtime or to determine the operating hours of connected devices.

Fig. 5.18 Reading the system time with system function TIME_TCK

The count value of an runtime meter is also retained with a restart, failure of the backup voltage, and following a memory reset.

The CPU 312, CPU 314, and CPU 315 each have one runtime meter, the CPU 317 and CPU 319 have four each. The range of values is 32 bits (2^{31}–1 hours). The runtime meter also stops when the CPU is at STOP or HOLD; if the CPU restarts, the runtime meter must be restarted if required.

If the maximum duration has been reached, the runtime meter remains stationary and signals an overflow. An runtime meter can only be set to a new value or zero using an SFC call.

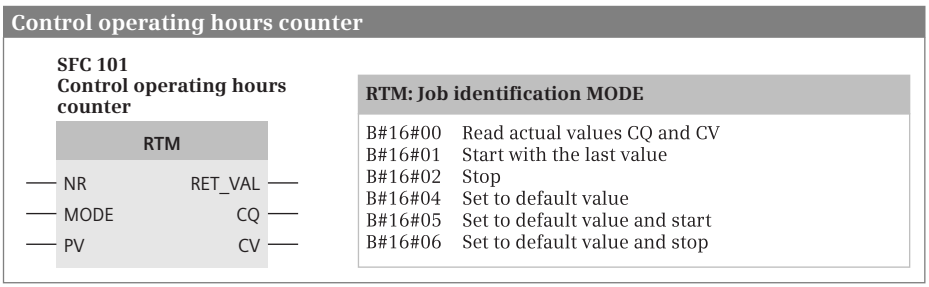


Fig. 5.19 System block for controlling the runtime meter

RTM Control runtime meter

The system function RTM controls an runtime meter. Fig. 5.19 shows the graphic representation of the system function.

RTM controls the runtime meter in 32-bit mode whose number is specified in the NR parameter. The MODE parameter defines the function to be executed. The value to which the runtime meter is to be set (default value or start value in hours) is present in the PV parameter. The 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 replaces the system functions SET_RTM (SFC 2), CTRL_RTM (SFC 3) and READ_RTM (SFC 4), which control the 32-bit runtime meter in 16-bit mode.

5.6 Interrupt processing

5.6.1 Introduction to interrupt processing

Interrupt processing is event-driven program execution. When such an event occurs, the operating system interrupts execution of the main program and calls the routine allocated to this particular event. Once this routine has been processed, the operating system resumes execution of the main program at the point of interruption. Such an interruption can take place after every operation (statement).

Applicable events may be interrupts and errors. A priority scheduler controls the execution order if interrupt events occur virtually simultaneously.

Each routine associated with an interrupt event is written in an organization block in which further blocks can be called. An event of higher priority interrupts execution of the routine in an organization block with a lower priority. You can influence the interruption of a program by events of higher priority using system blocks (Chapter 5.7.6 “Disable, delay, and enable interrupts and asynchronous errors” on page 209).

Events

The response of the operating system is based on events. If an organization block is assigned to the event, the block is called when the event occurs. If calling is not possible at this moment, the event is placed in the queue that corresponds to its priority.

If no organization block is assigned to an event, the preset system response is carried out when the event occurs: The operating system either calls the organization block OB 85 or changes to the STOP operating state.

Current start and interrupt information

Every organization block contains information concerning the start event in the first 20 bytes of the temporary local data. A detailed description of the start information can be found in Chapter 4.8 “Start information” on page 143.

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 5.6.9 “Reading additional interrupt information” on page 199).

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. 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.

5.6.2 Priority classes

Table 3.1 shows the organization blocks present with SIMATIC S7-300 with their execution priority. The execution priority is predefined for a CPU 300 and is not adjustable.

Table 5.4 Organization blocks available with the standard controllers of S7-300

OB	Prio.	Start event	Standard name	Remark
OB1	1	Start of main program	Main	–
OB10	2	Time-of-day interrupt	TOD_INT0	–
OB20 OB21	3 4	Time-delay interrupt	DEL_INT0 DEL_INT1	–
OB32 OB33 OB34 OB35	9 10 11 12	Cyclic interrupt	CYC_INT2 CYC_INT3 CYC_INT4 CYC_INT5	–
OB40	16	Hardware interrupt	HW_INT0	–
OB55 OB56 OB57	2	Status interrupt Update interrupt Manufacturer-specific interrupt	DP: STATUS ALARM DP: UPDATE ALARM DP: MANUFACTURE ALARM	Not with CPU 312 and CPU 314 Not with CPU 312 and CPU 314 Not with CPU 312 and CPU 314
OB61	25	Isochronous mode interrupt	SYNC_1	Not with CPU 312 and CPU 314
OB80 OB82 OB83 OB85 OB86 OB87	26 *)	Time error Diagnostics error interrupt Insert/remove module interrupt Program execution error Rack failure Communication error	CYCL_FLT I/O_FLT1 I/O_FLT2 OBNL_FLT RACK_FLT COMM_FLT	– – Not with CPU 312 and CPU 314 – Not with CPU 312 –
OB100	27	Warm restart	COMPLETE RESTART	–
OB121 OB122	**)	Synchronous error I/O access error	PROG_ERR MOD_ERR	–

*) The priority is 28 in the STARTUP operating state
**) This error OB has the priority of the error-causing OB

The main program has the lowest execution priority and can be interrupted by all other OB start events. The startup program is present in organization block OB 100 (warm restart) and has priority 27. Asynchronous errors occurring during the

startup belong to priority class 28. The diagnostic interrupt is also an asynchronous error.

If the start event happens and the corresponding organization block is not present, the CPU calls the OB 85 *Program execution error* or switches to STOP.

5.6.3 Time-of-day interrupt, organization block OB 10

You use a time-of-day interrupt if you wish to execute a program once at a particular time or periodically, for example daily. The organization block OB 10 is provided with S7-300 for execution of a time-of-day interrupt.

You can configure the time-of-day interrupt in the hardware configuration or control it from the program during runtime using system functions. A prerequisite for correct execution of the time-of-day interrupt is a correctly set real-time clock on the CPU.

Start information

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), the start information of the OB 10 contains the tag

OB10_PERIOD_EXE Processing interval

The processing interval specifies the interval with which the organization block is processed (see PERIOD parameter of system function SET_TINT).

Using the time-of-day interrupt

To start the 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 using the hardware configuration or also with system functions. Note that activation with the hardware configuration means that the time-of-day interrupt is automatically started following parameterization of the CPU.

You can start the 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.

Configuring the time-of-day interrupt

The time-of-day interrupt is configured using the hardware configuration. *Time-of-day interrupts* are also present in the *Interrupts* group in the CPU properties. Activate the time-of-day interrupt on the *Time-of-day interrupts* tab and select the processing interval from a drop-down list in the *Interval* column. Under *Start time* you can enter the time of the first execution.

System functions for processing the time-of-day interrupt

You can use system functions to set, cancel, and activate the 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. 5.20 shows the graphic representation of the system functions.

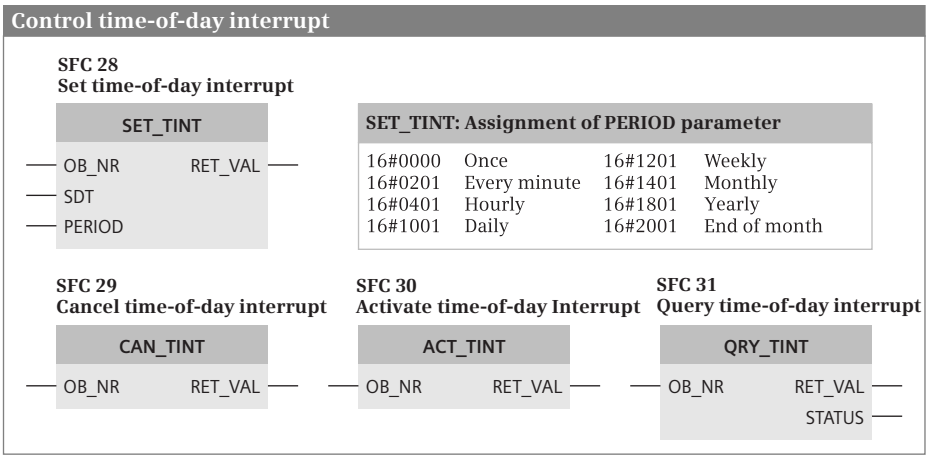


Fig. 5.20 System blocks for controlling the time-of-day interrupt

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. The operating system ignores any specified seconds and milliseconds and sets these values to zero. 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 the 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 the 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).

Table 5.5 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.
Other	Always "0"	

QRY_TINT provides information on the status of the time-of-day interrupt. The STATUS parameter contains the desired information and the individual bits have the significance shown in Table 5.5.

Behavior during startup

During a warm restart, the operating system deletes all settings you have made using a system function. The settings parameterized with the hardware configuration are retained.

You can obtain information in the startup program on the status of the time-of-day interrupt using QRY_TINT and cancel or reset and activate the time-of-day interrupt as required. Processing of the time-of-day interrupt OB only takes place in the RUN operating state.

Error response

If the time-of-day interrupt OB is missing in the user program when called, the operating system calls the organization block OB 85 *Program execution error*. If OB 85 is not present, the CPU switches to STOP.

A time-of-day interrupt which has been deactivated by parameterization of the CPU cannot be executed even if the time-of-day interrupt OB is present. The CPU then switches to STOP.

If you activate the time-of-day interrupt for single processing and if the start time (from the viewpoint of the real-time clock) is in the past, the operating system calls the organization block OB 80 *Time error*. If this is not present, the CPU switches to STOP.

If you activate the time-of-day interrupt for periodic processing and if the start time (from the viewpoint of the real-time clock) is in the past, the time-of-day interrupt OB is processed at the next due time.

If you advance the real-time clock by more than approximately 20 s, either through a correction or synchronization, such that the start time for the time-of-day interrupt OB is bypassed, the operating system calls the organization block OB 80 *Time error*. The time-of-day interrupt OB is subsequently processed once.

If you have set the real-time clock back by more than approximately 20 s, either through a correction or synchronization, an activated time-of-day interrupt OB at the points in time which have already passed is no longer processed.

If the time-of-day interrupt OB is still being processed and the next (periodic) call already occurs, the operating system calls the organization block OB 80 *Time error*. Following processing of the OB 80 and the time-of-day interrupt OB, the time-of-day interrupt OB is started again.

5.6.4 Time-delay interrupts, organization blocks OB 20 and OB 21

A time-delay interrupt allows you to implement a delay time independent of the timer functions and asynchronous to cyclic program execution. With a CPU 300, organization blocks OB 20 and OB 21 are set aside for processing a time-delay interrupt.

Start information

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), the start information of the time-delay interrupt OB contains the tags

OBxx_SIGN	Job ID
OBxx_DTIME	Parameterized delay time

xx stands for the OB number 20 or 21. The OBxx_SIGN tag contains the job ID which you have parameterized in the SIGN parameter of system function SRT_DINT. The OBxx_DTIME tag contains the delay time in ms.

Using time-delay interrupts

A time-delay interrupt is started by calling SRT_DINT; this also passes on the delay interval and the delay organization block. When the delay interval has expired, the corresponding organization block is called. You can query the status of a time-delay interrupt with QRY_DINT.

You can also use CAN_DINT to cancel execution of a time-delay interrupt that has not yet started. The associated organization block is then no longer called.

Configuring time-delay interrupts

Configuration of the time-delay interrupts is carried out in two steps:

- You create an organization block for a time-delay interrupt in the *Program blocks* folder using *Add new block*.
- Then program the SRT_DINT function and possibly the CAN_DINT function and assign the number of the time-delay interrupt OB to the OB_NR parameter.

You can find the functions for the time-delay interrupts in the program elements catalog under *Extended instructions* > *Interrupts*.

System functions for time-delay interrupts

You can use system functions to start and cancel a time-delay interrupt and also to query the status. Fig. 5.21 shows the graphic representation of the system functions.

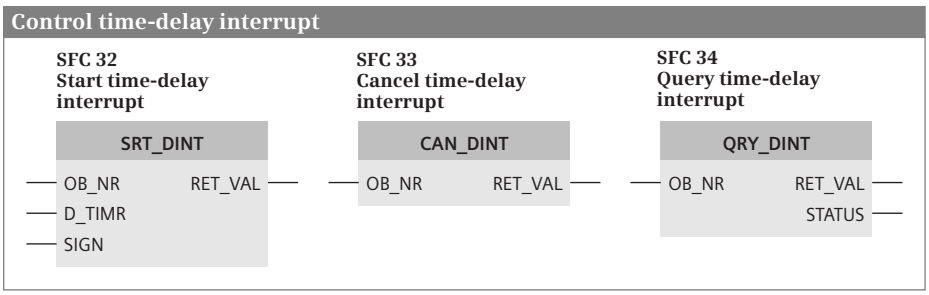


Fig. 5.21 Start, cancel, and query a time-delay interrupt

SRT_DINT starts 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 delay time value and a job ID in the start information for this OB. You define the job ID in the SIGN parameter; you can read the same value in bytes 6 and 7 of the start information of the associated time-delay interrupt 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 a started 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 5.6.

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

Table 5.6 STATUS parameter of system function QRY_DINT

Bit	Meaning with signal state "0"	Meaning with signal state "1"
0	The CPU is in 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"	

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 calls the organization block OB 85 *Program execution error*. If OB 85 is not present, the CPU switches to STOP.

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 enters the STOP operating state if the OB 80 is not present.

5.6.5 Cyclic interrupts, organization blocks OB 32 to OB 35

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. With a CPU 300, organization blocks OB 32 to OB 35 are set aside for processing the cyclic interrupts.

Start information

In addition to the standard data (see Chapter 4.8 "Start information" on page 143), the start information of the cyclic interrupt OB contains the tags

OBxx_PHASE_OFFSET Phase offset

OBxx_EXC_FREQ Parameterized time interval

xx stands for the OB number 32 to 35. If the OBxx_STRT_INF tag has the value B#16#3A, the phase shift and the time interval are output in microseconds (µs), otherwise in milliseconds (ms).

Activation of cyclic interrupt

You can set the time interval and the phase offset from 1 ms to 1 min in 1 ms steps during parameterization of the CPU (with CPU 319 from 500 µs in steps of 500 µs).

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 results in higher accuracy of the processing intervals.

The start time of the time interval and the phase offset is the transition from the STARTUP operating state to RUN. The call instant for a cyclic interrupt OB is thus the time interval plus the phase offset. An example is shown in Fig. 5.22. No phase offset is set in the left 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.

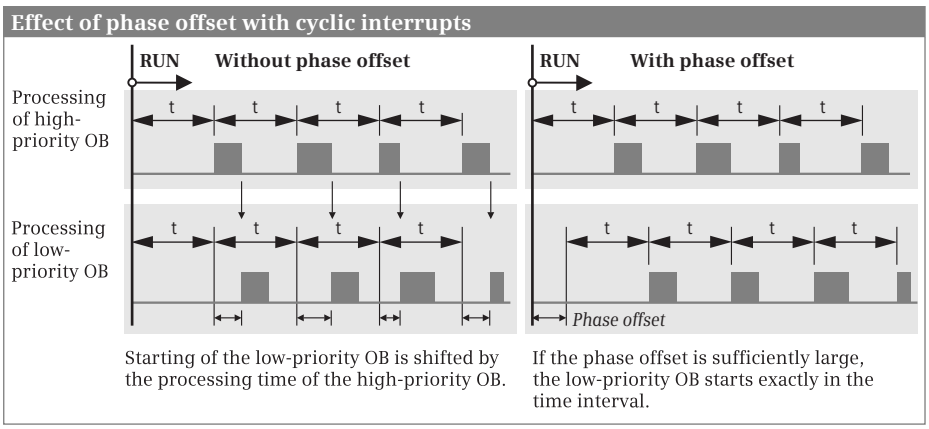


Fig. 5.22 Processing of cyclic interrupts with and without phase offset

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.

Configuring cyclic interrupts

Configuration of the cyclic interrupts is carried out in two steps:

- You create an organization block for a cyclic interrupt in the *Program blocks* folder using *Add new block*.
- You set the interval and the phase offset in the CPU properties under *Interrupts* > *Cyclic interrupts*.

Behavior during startup

Processing of cyclic interrupts is not possible in the startup program. The time intervals only commence upon transition to the RUN operating state.

Error response

If the associated cyclic interrupt is repeated during an ongoing cyclic interrupt OB, the operating system calls the organization block OB 80 *Time error*. The cyclic interrupt that caused the error is executed later.

If OB 80 is not present, the CPU switches to STOP.

5.6.6 Hardware interrupt, organization block OB 40

You use a hardware interrupt to allow immediate detection in the user program of an event in the controlled process, making it possible to respond with an appropriate routine. With a CPU 300, the organization block OB 40 is set aside for processing a hardware interrupt.

Start information

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), the start information of the hardware interrupt OB contains the tags

OB40_IO_FLAG	I/O ID
OB40_MDL_ADDR	Module start address
OB40_POINT_ADDR	Interrupt information

With this information you can identify the component triggering the interrupt. The OB40_IO_FLAG tag identifies with B#16#54 an input module, and with B#16#55 an output module. The module start address is contained in the OB40_ADL_ADDR tag. OB40_POINT_ADDR identifies in a bit array which channel of the module has triggered the hardware interrupt.

Enabling hardware interrupts

A hardware interrupt is enabled on a module designed for this. This can be, for example, a digital input module which records a signal coming from the process, or a function module which triggers a hardware interrupt due to a process on the module.

Activation of a hardware interrupt is initially disabled by default. You enable processing of the hardware interrupt in the parameterization (static parameter). You can select whether the hardware interrupt is to be triggered by an incoming event, an outgoing event, or by both (dynamic parameter). You can change dynamic parameters during runtime by calling a function.

In a PROFIBUS DP master system, an intelligent DP slave can trigger a hardware interrupt in the master CPU by means of the system function DP_PRAL. The interrupt

ID which you pass on in the AL_INFO parameter to the DP_PRAL function is present in the start information of OB 40 in the OB40_POINT_ADDR tag.

The module acknowledges the hardware interrupt following processing of the organization block belonging to the hardware interrupt.

Detecting hardware interrupts

If an event occurs during processing of the hardware interrupt OB which would again trigger the hardware interrupt, this hardware interrupt is lost if the event is no longer present following acknowledgment. It is irrelevant whether the event comes from the module whose hardware interrupt is currently being processed or from another module.

A diagnostic interrupt may be triggered during processing of the hardware interrupt. If a further hardware interrupt occurs on the same channel in the period between triggering of the hardware interrupt and its acknowledgment, the loss of the hardware interrupt is signaled for system diagnostics by means of a diagnostic error interrupt.

Behavior during startup

The modules do not generate hardware interrupt events in the startup program. Interrupt processing commences with the transition to the RUN operating state. Hardware interrupts present during the transition are lost.

Error handling

If the hardware interrupt OB is missing in the user program when the hardware interrupt event occurs, the operating system calls the organization block OB 85 *Program execution error*. The hardware interrupt is acknowledged. If OB 85 is not present, the CPU switches to STOP.

5.6.7 Interrupts for DPV1 organization blocks OB 55 to OB 57

PROFIBUS DPV1 slaves (PROFIBUS) and correspondingly designed IO devices (PROFINET IO) can trigger the following interrupts:

- Status interrupt if, for example, the station changes its operating state; the interrupt organization block OB 55 is called.
- Update interrupt if, for example, the station parameters are changed over the bus system or directly; the interrupt organization block OB 56 is called.
- Manufacturer-specific interrupt if an event envisaged for this by the manufacturer occurs; the interrupt organization block OB 57 is called. The events triggering the interrupt are defined by the station manufacturer.

Start information

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), the start information of the DPV1 interrupt OBs contains further tags in the area

from byte 5 to byte 11 which identify the components triggering the interrupts. 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 5.6.9 “Reading additional interrupt information” on page 199).

Behavior in the STOP and STARTUP operating states

Distributed I/O stations can also generate interrupts even if the central CPU is in the STOP or STARTUP operating state. The CPU cannot call or hardware interrupt organization blocks when at STOP; processing of the interrupts is not carried out later either when the CPU goes to RUN. DPV1 interrupts occurring in the STARTUP operating state are processed in the transition from STOP to RUN.

The received interrupt events are entered into the diagnostic buffer and into the module status data both at STOP and STARTUP. You can read the module status data with the system function RDSYSST.

Error handling

If the corresponding DPV1 interrupt OB is missing in the user program when the DPV1 interrupt is triggered, the operating system calls the organization block OB 85 *Program execution error*. The DPV1 interrupt is acknowledged. If OB 85 is not present, the CPU switches to STOP.

5.6.8 Isochronous mode interrupt, organization block OB 61

The “Isochronous mode” function permits synchronous reading, processing and output of I/O signals in a fixed (equidistant) cycle for PROFIBUS DP or PROFINET IO. The user program executed in isochronous mode is present in organization block OB 61. The system functions SYNC_PI and SYNC_PO are available for isochronous updating of the process image.

The isochronous mode interrupt is triggered by the *Global_Control* (GC) command of the DP master for PROFIBUS DP and by a time scale of the IRT communication derived from the send clock for PROFINET. Isochronous processing is described in Chapter 16.5 “Isochronous mode” on page 641.

Start information

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), the start information of the isochronous mode interrupt OB contains the tags

OB61_GC_VIOL	GC violation with PROFIBUS DP
OB61_FIRST	First execution following STARTUP or HOLD
OB61_MISSED_EXEC	Number of discarded OB calls
OB61_DP_ID	ID of the DP master system or PROFINET IO system

The OB61_GC_VIOL and OB61_MISSED_EXEC tags allow you to recognize faulty isochronous processing. Default settings can be programmed in the first cycle identified by the OB61_FIRST tag. The OB61_DP_ID tag indicates the DP master system or PROFINET IO system from which the OB 61 has been called.

Configuring isochronous mode interrupts

To activate the isochronous mode interrupt, set the number of the DP master system or PROFINET IO system in the properties of the "master CPU" under *Interrupts* > *Isochronous mode interrupts*. Enter the number 1 in the *Process image partition(s)* column.

To carry out interrupt processing, add the isochronous mode interrupt OB to the user program in the *Program blocks* folder. In the OB, call the system function SYNC_PI prior to the interrupt routine and the system function 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 PIP 1.

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!

System functions for the isochronous mode interrupt

You can use system functions to update the used process image partition in the program of a isochronous mode interrupt OB. Fig. 5.23 shows the graphic representation of the system functions.

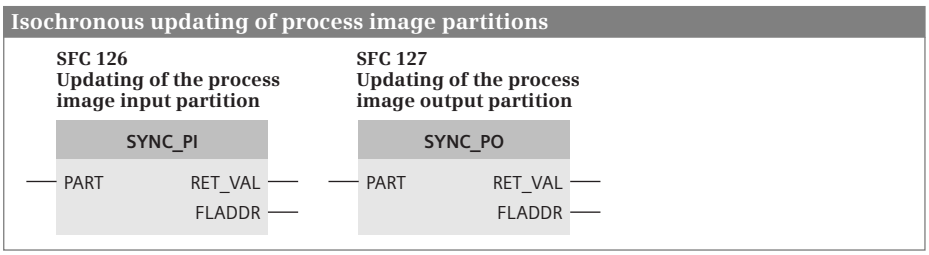


Fig. 5.23 System blocks for isochronous updating of process image partitions

The system function **SYNC_PI** updates the process image partition PIP 1 of the inputs. The system function **SYNC_PO** updates the process image partition PIP 1 of the outputs. Updating is carried out isochronously and data-consistent. The two system functions may only be called in the isochronous mode interrupt OB. In the PART parameter you define the process image partition to be updated (with a CPU 300 using B#16#01).

The process image partitions are not updated if an error is detected. Exceptions:

- If an access error occurs when updating the process image input partition, the inputs of faulty modules are set to signal state “0”; and the OB 85 *Program execution error* is not called.
- A consistency warning is output if the complete data could not be transferred consistently to the outputs. However, the data of individual slaves or devices are consistent.
- If an access error occurs during updating of the process image output partition, the data of the faulty modules is not transferred; it remains unchanged in the process image partition. The updating of unaffected modules is divided between two cycles (with consistency warning).

Behavior in the STOP, HOLD, and STARTUP operating states

The isochronous mode interrupt is only processed in the RUN operating state. A isochronous mode interrupt in the STOP, HOLD 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 OB called for the first time in RUN.

Error handling

If a isochronous mode interrupt arrives before the associated isochronous mode interrupt OB has been completed, the organization block OB 80 *Time error* is called. This can happen if the user program in a isochronous mode interrupt OB takes too long or if processing has been interrupted for too long because of higher-priority program parts. The OB requested by the “too early” interrupt is rejected and the OB 80 *Time error* is called. Here it is then possible to respond to the time error. The number of failed isochronous mode interrupts is output in the start information of the next isochronous mode interrupt OB processed.

5.6.9 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. Fig. 5.24 shows the graphic representation of RALRM.

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 (task information) contains the complete start information of the organization block in which RALRM was called, independent of the nesting depth in which it was called. The system block RALRM thus partially replaces the system function RD_SINFO. Management information is present in bytes 20 to 27, e.g. which component has triggered the interrupt.

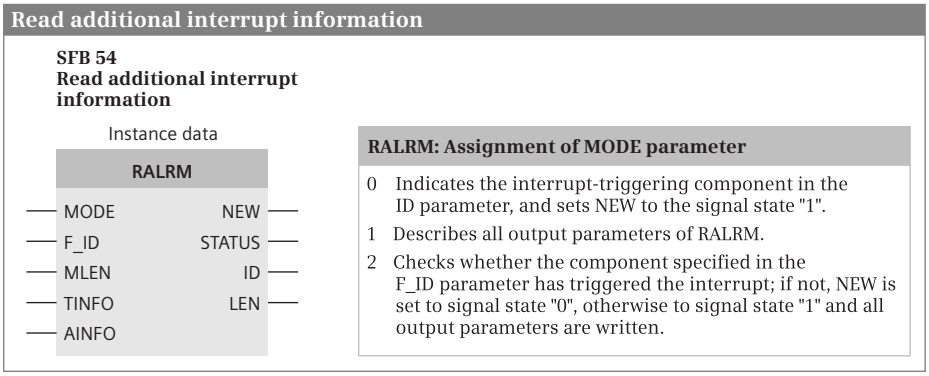


Fig. 5.24 System block for reading additional interrupt information

In bytes 0 to 3 (bytes 0 to 25 with PROFINET), the destination area AINFO (alarm information) contains the header information, e.g. the number of received bytes of the additional interrupt information or interrupt type. Bytes 4 to 223 (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.

5.7 Error handling

5.7.1 Causes of errors and error responses

The CPU can detect and signal errors in the program execution and from the modules. These errors include:

- Error in arithmetic operations (overflow, invalid REAL number) with setting of the status bits, e.g. status bit OV with number range overflow
- Error during execution of the user program (synchronous error) with calling of organization blocks OB 121 and OB 122
- Error in automation system independent of program execution (asynchronous error) with calling of organization blocks OB 80 to OB 87

The occurrence of an error and possibly the reason are indicated by error LEDs on the front panel of the CPU. In the event of serious errors, for example an impermissible operation code, the CPU is directly set to STOP.

The system diagnostics can detect errors on the modules and enters these errors into a diagnostic buffer. The diagnostic buffer also contains information on the mode transitions of the CPU, e.g. the reasons for STOP.

5.7.2 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.

Programming error, organization block OB 121

A programming error is present if program execution is faulty. This includes, for example, BCD conversion errors, errors with indirect addressing, addressing of missing SIMATIC timers, counters or blocks. Organization block OB 121 is called in the event of a programming error.

If the organization block OB 121 is not present when a programming error occurs, the CPU switches to STOP.

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), with a CPU 300 the **Start information** of the programming error OB contains the tags

OB121_SW_FLT	Start request for OB 121 (error code)
OB121_FLT_REG	Error source depending on the error code

The error code is provided in the OB121_SW_FLT tag. Example: If the OB121_SW-FLT tag is occupied by B#16#32 (= access to a non-existent global data block), the OB121_FLT_REG tag contains the number of the missing data block.

I/O access error, organization block OB 122

An I/O access error is present if a faulty module, a non-existent module, or an I/O address unknown on the CPU is accessed. The operating system responds differently depending on the type of access:

- The I/O access takes place from the user program. The I/O access error organization block OB 122 is then called.
- The I/O access error occurs during automatic updating of a process image (partition). The preset response for a CPU 300 is that neither a diagnostic buffer entry nor an OB call is carried out (see “Program execution error OB 85” on page 208 in Chapter 5.7.5 „Asynchronous errors“).

▷ The I/O access error occurs if a process image partition is updated by a system function. The error and address of the first error-signaling byte are returned via their parameters (SYNC_PI and SYNC_PO).

If an error occurs during updating of the process image input, the corresponding input bytes are set to zero until the error has been eliminated. If an I/O access error occurs during a write access to the peripheral outputs, a CPU 300 does not update the process image output.

If the organization block OB 122 is not present when a programming error occurs, the CPU switches to STOP.

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), with a CPU 300 the **Start information** of the I/O access error OB contains the tags

OB122_SW_FLT	Start request for the OB 122 (error code)
OB122_MEM_AREA	Memory area (bits 0 to 3) and type of access (bits 4 to 7)
OB122_MEM_ADDR	Memory address at which the error occurred

In the OB122_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 OB122_MEM_AREA tag can be bit (with value 1), byte (2), word (3), and doubleword (4). The memory area can be the I/O area PI or PQ (with value 0), the process image input, (1) or the process image output (2). The error-causing memory address is then in the OB122_MEM_ADDR tag.

Priority class, block nesting depth, and L stack

A synchronous error OB has the same priority (class) as the block which caused the error. The accumulators and address registers of the synchronous error OB contain the values which the error-causing block had at the break point. The data block registers are deleted and the indicator word is assigned undefined.

It is similar with the block nesting depth. The nesting depth per priority class permissible for a CPU is the total of the nesting depth of “normal” processing and the nesting depth of synchronous error processing. The additional block nesting depth in a synchronous error OB is 4 for a CPU 300.

Note when calling a synchronous error OB that its 20 bytes of start information are additionally stored in the L stack of the error-causing priority class, as well as the further temporary local data of the synchronous error OBs and all blocks called in these OBs. The area reserved for the temporary local data must be designed for this in every priority class affected (program execution level) (preset with a CPU 300).

5.7.3 Enabling and disabling synchronous error processing

You can use system functions to disable calling of the error OB in the event of a synchronous error event, or enable it again, and also query which synchronous error events have occurred. Fig. 5.25 shows the graphic representation of the system functions.

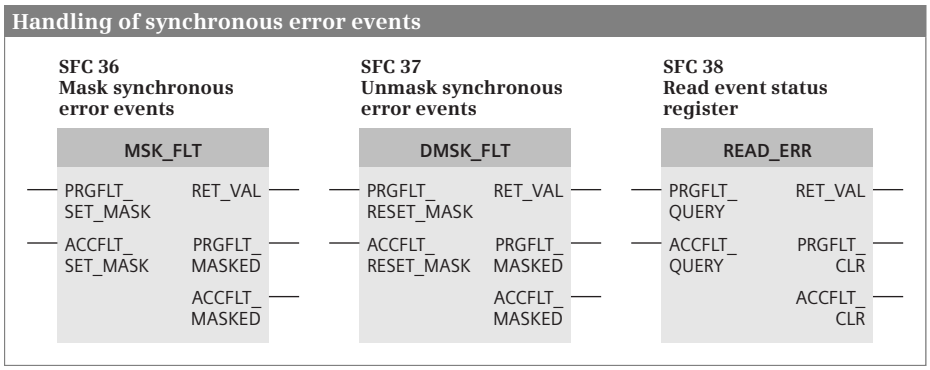


Fig. 5.25 System blocks for handling of synchronous error events

With the CPU 300, regardless of whether an error event is masked or unmasked, the error is entered in the diagnostic buffer and the group error LED of the CPU lights up.

Error masks

You use the error masks to control the system functions for handling synchronous errors. 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”.

The assignments of the access error mask are shown in Table 5.7, where the “Error code” column shows the assignment of the OB122_SW_FLT tag in the start information of OB 122.

The assignments of the programming error mask are shown in Table 5.8. The “Error code” column shows the assignment of the OB121_SW_FLT tag in the start information of OB 121. The bits of the error masks not listed in the table are not relevant to the handling of synchronous errors.

Table 5.7 Assignment of access error mask

Bit	Error code	Assignment
2	B#16#42	I/O access error when reading The module is not present or does not acknowledge
3	B#16#43	I/O access error when writing The module is not present or does not acknowledge

Table 5.8 Assignment of programming error mask

Bit	Error code	Assignment
1	B#16#21	BCD conversion error (pseudo tetrad during conversion)
2	B#16#22	Area length error when reading (operand outside permissible range)
3	B#16#23	Area length error when writing (operand outside permissible range)
4	B#16#24	Area error when reading (incorrect area in area pointer)
5	B#16#25	Area error when writing (incorrect area in area pointer)
6	B#16#26	Faulty number of a timer function
7	B#16#27	Faulty number of a counter function
8	B#16#28	Address error when reading (bit address <>0 with byte, word or doubleword access with indirect addressing)
9	B#16#29	Address error when writing (bit address <>0 with byte, word or doubleword access with indirect addressing)
16	B#16#30	Write error with global data block (read-only block)
17	B#16#31	Write error with instance data block (read-only block)
18	B#16#32	Faulty number of a global data block (DB register)
19	B#16#33	Faulty number of an instance data block (DI register)
20	B#16#34	Faulty number of a function FC
21	B#16#35	Faulty number of a function block FB
26	B#16#3A	Called data block DB does not exist
28	B#16#3C	Called function FC does not exist
30	B#16#3E	Called function block FB does not exist

MSK_FLT Mask synchronous error events

By means of the error masks, the system function MSK_FLT disables calling of the synchronous error OBs. By means of signal state “1” you identify in the error masks for which synchronous errors the OBs 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 OB is not called and the event is not entered in the event status register. Masking applies to the current priority class (program execution level). If you mask the call of a synchronous error

OB in the main program, for example, the synchronous error OB is nevertheless called if the error occurs in an interrupt routine.

DMSK_FLT Unmask synchronous error events

By means of the error masks, the system function DMSK_FLT enables calling of the synchronous error OBs. By means of signal state “1” you identify in the error masks the synchronous errors for which the OBs 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 OB is called and the event is entered in the event status register. Unmasking applies to the current priority class (program execution level).

READ_ERR Read event status register

The system function 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 (program execution level) are signaled.

5.7.4 Enter substitute value

REPL_VAL is called in a synchronous error OB and enters a substitute value into accumulator 1. Fig. 5.26 shows the graphic representation of the system function.

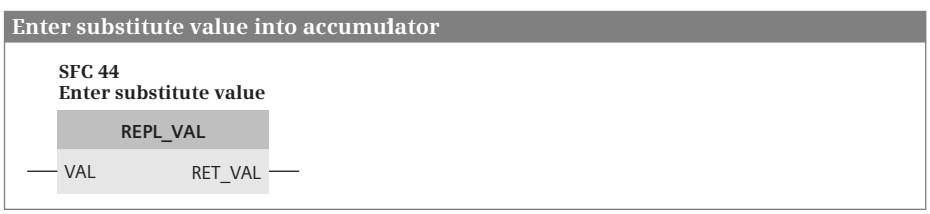


Fig. 5.26 System block for entering a substitute value

You use REPL_VAL if no value can be read from a module, for example because the module is defective. The OB 122 *Access error* is then called with each access operation. The start information indicates which module has caused the error. You can then load a substitute value into accumulator 1 by calling REPL_VAL; program execution is then continued with this substitute value.

REPL_VAL may only be called in a synchronous error OB (OB 121 or OB 122).

5.7.5 Asynchronous errors

Asynchronous errors are errors which can occur independent of program execution. If an asynchronous error occurs, the operating system calls one of the following organization blocks:

OB 80	Time error
OB 82	Diagnostic error interrupt
OB 83	Insert/remove module interrupt
OB 85	Program execution error
OB 86	Rack failure
OB 87	Communication error

The organization block OB 82 (diagnostics error interrupt) is described in Chapter 5.8.1 „Diagnostic error interrupt, organization block OB 82“.

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:

- Cycle monitoring time exceeded
- OB request error (the requested OB is still being processed, or an OB is requested too frequently within a priority class)
- Time-of-day error interrupt (time-of-day interrupt expired through setting ahead of time or following transition to RUN)

If OB 80 is not present, the CPU switches to STOP in the event of a time error. The CPU also switches to STOP if the OB is called for a second time in the same program cycle because of a cycle time violation.

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), with a CPU 300 the **Start information** of the time error OB contains the tags

OB80_FLT_ID	Start request (error code)
OB80_ERROR_EV_CLASS	Error-triggering event class

OB80_ERR_EV_NUM	Error-triggering event number
OB80_OB_PRIORITY	Error information depending on the error code
OB80_OB_NUM	Error information depending on the error code

You can use these tags to determine the cause of the time error. For example, if the OB80_FLT_ID tag is B#16#02 (OB request error), the priority class of the OB that has caused the error is present in the OB80_OB_PRIORITY tag and the number of this OB in OB80_OB_NUM.

Insert/remove module interrupt OB 83

An insert/remove module interrupt with a CPU 300 only exists for PROFINET IO components.

The operating system monitors the module configuration every second. Every insertion or removal of a module in the RUN, STOP and STARTUP states results in an entry in the diagnostic buffer and in the system state list.

In addition, the operating system calls the operation block OB 83 in RUN. If OB 83 is not present, the CPU switches to STOP in the event of an insert/remove module interrupt.

One second can pass until triggering of the insert/remove module interrupt. It is therefore possible that, when removing a module, an access error or an error in process image updating is signaled in the meantime.

If a suitable module is inserted into a configured slot, automatic parameterization of the module is carried out by the CPU using the data records present on the latter. The OB 83 is only called after this in order to signal the inserted module as being ready for operation again.

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), with a CPU 300 the **Start information** of the insert/remove module interrupt OB contains the tags

OB83_MDL_ID	Interrupt-triggering peripheral area I:P (B#16#54) or Q:P (B#16#55)
OB83_MDL_ADDR	Module start address of interrupt-triggering module
OB83_RACK_NUM	Number of rack or number of distributed station
OB83_MDL_TYPE	Type of interrupt-triggering module

You can determine the interrupt-triggering module using these tags.

The RALRM instruction (read additional interrupt information) can provide further interrupt information with an insert/remove module interrupt in a DPV1 or PROFINET-capable CPU (see Chapter 5.6.9 “Reading additional interrupt information” on page 199).

Program execution error OB 85

The operating system calls the organization block OB 85 when one of the following events occurs:

- Start request for an organization block that is not loaded
- Error during access of operating system to a block (e.g. instance data block missing when calling a system function block SFB)
- I/O access error with system (automatic) updating of process image (configurable)

The OB 85 is not called by default if an I/O access error occurs during automatic updating of the process image. The substitute value or zero is entered into the associated byte upon the first faulty access; it is no longer updated thereafter.

You can set how the OB 85 call is to be handled in the event of an I/O access error on the system side during hardware configuration in the *Cycle* group of the CPU properties:

- OB 85 is called each time. The associated input byte is written each time with the substitute value or zero.
- OB 85 is called upon the first access error with the attribute “Incoming”. An affected input byte is only written the first time with the substitute value or zero; it is no longer updated thereafter. Once the error has been eliminated, the OB 85 is called with the attribute “Outgoing”; an affected input byte is subsequently updated “normally” again.
- OB 85 is not called in the event of an access error. Affected input bytes are only written once with the substitute value or zero and no longer updated thereafter.

If OB 85 is not present, the CPU switches to STOP in the event of a program execution error.

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), with a CPU 300 the **Start information** of the program execution error OB contains the tags

OB85_ERR_EV_CLASS	Error-triggering event class
OB85_ERR_EV_NUM	Error-triggering event number
OB85_OB_PRIOR	Priority class of the OB that caused the error
OB85_OB_NUM	Number of the OB that caused the error

You can use these tags, for example, to determine a called organization block which is not present in the program.

Rack failure OB 86

The operating system calls the organization block OB 86 if

- a DP master system fails or becomes available again,
- a distributed station (PROFIBUS DP or PROFINET IO) fails or returns, and

- a distributed station (PROFIBUS DP or PROFINET IO) is activated with the system function A_ACT_DP with MODE = 3 or deactivated with MODE = 4.

If OB 86 is not present, the CPU switches to STOP upon one of the above-mentioned events.

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), with a CPU 300 the **Start information** of the rack failure OB contains the tags

OB86_FLT_ID	Start request (error code)
OB86_MDL_ADDR	Module address depending on the error code
OB86_RACKS_FLTD	Additional information depending on the error code

You can use these tags, for example, to determine the IO system and the station number of a failed or returned IO device.

Communication error OB 87

The operating system calls the organization block OB 87 if a communication error occurs. Communication errors can be, for example:

- Sending of diagnostics entries not possible
- Error in time-of-day synchronization

If OB 87 is not present, the CPU switches to STOP in the event of a communication error.

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), with a CPU 300 the **Start information** of the communication error OB contains the tags

OB87_FLT_ID	Start request (error code)
OB87_RESERVED_3	Information depending on the error code
OB86_RESERVED_4	Information depending on the error code

You can use these tags, for example, to determine an illegal jump in time due by means of time synchronization.

5.7.6 Disable, delay, and enable interrupts and asynchronous errors

Disable, delay, and enable system functions for interrupt events influences all interrupts and all asynchronous errors. Fig. 5.27 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 asynchronous error occurs following disabling, the associated organization block is no longer processed; if the OB does not exist, the CPU does not switch to STOP.

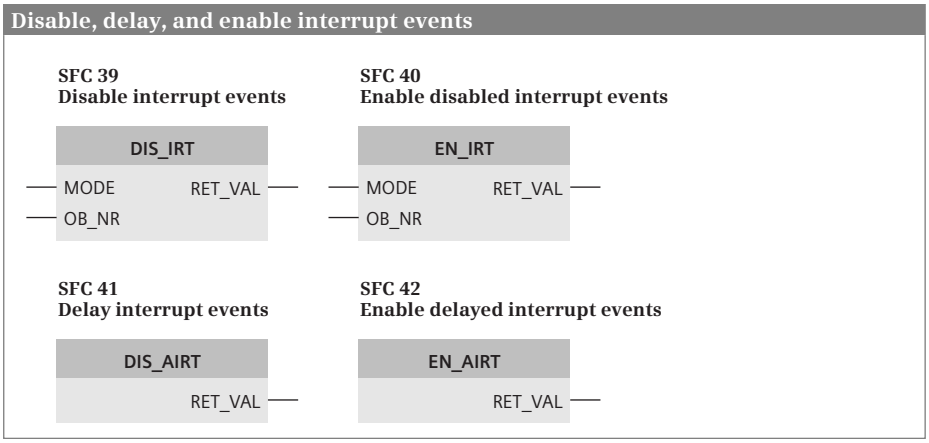


Fig. 5.27 System blocks for handling interrupt events

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 5.9). Depending on the assignment of the MODE parameter, the disabled interrupt events are also entered into the diagnostic 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 CPU switches to STOP.

You can use the MODE and OB_NR parameters to specify which interrupts and asynchronous errors are to be enabled (Table 5.9).

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 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.

Table 5.9 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 diagnostic buffer.	–
B#16#81	The newly occurring interrupt events of an interrupt class are disabled without entry into the diagnostic buffer.	–
B#16#82	The newly occurring interrupt events of an interrupt are disabled without entry into the diagnostic buffer.	–

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.

5.8 Diagnostics

System diagnostics is the detection, evaluation, and reporting of errors that occur within the programmable controller. Examples of such errors are those in the user program or module failures, but also an open-circuit with signal modules.

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 583.

5.8.1 Diagnostic error interrupt, organization block OB 82

You use a diagnostic error interrupt in order to respond to a change in the diagnostics status of a module by means of an interrupt routine. The interrupt routine for an incoming or outgoing diagnostic event is in organization block OB 82.

Evaluating a diagnostic error interrupt

In the event of a diagnostic error interrupt, the operating system interrupts execution of the user program and calls the organization block OB 82. If OB 82 is not present, the CPU switches to STOP. You can disable or delay processing of the OB 82 (see Chapter 5.7.6 “Disable, delay, and enable interrupts and asynchronous errors” on page 209).

In addition to the standard data (see Chapter 4.8 “Start information” on page 143), the start information of the diagnostic error interrupt OB contains further tags in the area from byte 5 to byte 11 which identify the components triggering the interrupts. You can use these tags, for example, to determine the interrupt-triggering event and the module.

The additional interrupt information can be read using the system block RALRM. Module diagnostic data can be read using the system block RDSYSST. System block WR_USMSG is available for entering a user diagnostic event into the diagnostic buffer.

The diagnostics information on the modules is consistent until OB 82 is left, i.e. it remains frozen. The module acknowledges the diagnostic error interrupt when the program exits the organization block OB 82.

Configuring diagnostic error interrupts

A diagnostic error interrupt is configured using the hardware configuration. You activate the diagnostics interrupt and the associated event for correspondingly designed modules.

With PROFINET IO controllers you can set whether communication interrupts (diagnostic events of the PROFINET interface) are to call the diagnostic error interrupt OB 82: In the CPU properties under *PROFINET interface > Advanced options > Interface options*, activate the checkbox *Call the user program if communication errors occur*.

Behavior in the STOP and STARTUP operating states

Distributed I/O stations can also generate diagnostics interrupts even if the master or controller station is in the STOP or STARTUP operating state. The central CPU cannot call or process the diagnostics organization block when at STOP; processing of the diagnostics interrupt is not carried out later either when the CPU goes to RUN. Diagnostics interrupts occurring in the STARTUP operating state are processed in the transition from STOP to RUN.

The received interrupt events are entered into the diagnostic buffer and into the module status data both at STOP and STARTUP. You can read the module status data with the system function RDSYSST.

5.8.2 Read system state list

The system state list (SSL) describes the current status of an automation system. The contents of the SSL can only be read using information functions, contents can-

not be modified. Since the complete system state list is extremely comprehensive, reading is carried out in partial lists and partial list extracts. The partial lists are only compiled by the CPU's operating system when requested.

The SSL ID is available for identification of a partial list. This contains the module type class to which the list applies, the number of the partial list extract, and the actual system state partial list number (Fig. 5.28). Together with the index which specifies an object of a partial list, you are provided with the desired information. The CPU always provides information on the automation system as standard, but FM and CP modules can also use this service to provide information (see documentation on module). You can find the possible system state lists of a CPU in the description of the operation.

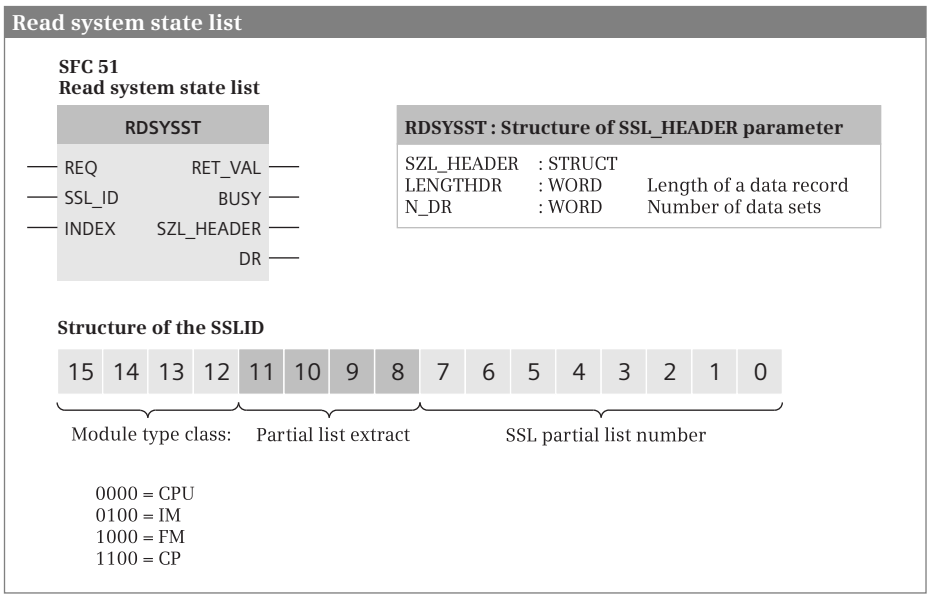


Fig. 5.28 Read system state list and structure of the SSL ID

Reading the header information

With the SSL_ID W#16#0Fxx you read the header information of a system state partial list without the associated data record (xx = system state partial list number). The SSL_HEADER.N_DR parameter (number of data records) then returns the maximum possible number of data records of the partial list extract which the module can supply with an SSL job. With dynamic partial lists, the value can be larger than the currently readable number. The length of the data record is indicated in *SSL_HEADER.LENGTHDR*. With this data in the header information it is possible, for example, to configure a sufficiently large data buffer for the associated system state partial list during startup.

RDSYSST Read system state partial list

The system function RDSYSST reads a partial list or a partial list extract of the system state list (SSL) (Fig. 5.28).

You trigger reading with REQ = "1", and BUSY = "0" indicates completion of the read operation. The operating system can process several asynchronously triggered read operations quasi-simultaneously; the number depends on the CPU. If RDSYSST signals a shortage of resources (W#16#8085) in the function value, trigger the read operation again.

You can obtain the assignments of the SSL_ID and INDEX parameters from the operation list for the CPU 300. Example: SSL_ID = W#16#0111 and INDEX = W#16#0001 are used to read out the CPU type and the version number. If the INDEX parameter is not required for information, its assignment is irrelevant.

In the DR parameter you specify the tag or data area in which RDSYSST is to enter the data records. For example, P#DB200.DBX0.0 WORD 256 provides an area of 256 data words in data block DB 200, starting at DBB 0. If the provided area is too small, as many data records as possible are supplied. Only complete data records are transmitted. However, the area must be able to accommodate at least one data record.

5.8.3 Read start information

RD_SINFO Read start information

RD_SINFO provides the start information of the current organization block – this is the OB at the top of the call tree – and also that of the last executed startup OB in a lower call level (Fig. 5.29).

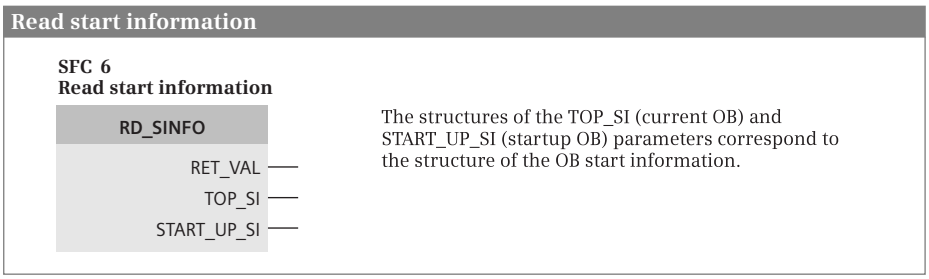


Fig. 5.29 Read start information RD_SINFO

The TOP_SI output parameter contains the first 12 bytes of the start information of the current OB, the START_UP_SI output parameter contains the first 12 bytes of the start information of the last executed startup OB. The time stamp is not included in either case. The structure of the start information is described in Chapter 4.8 "Start information" on page 143.

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.

5.8.4 Write user diagnostic event to the diagnostic buffer

WR_USMSG writes an entry to the diagnostic buffer and can send it to all logged-on display units (Fig. 5.30).

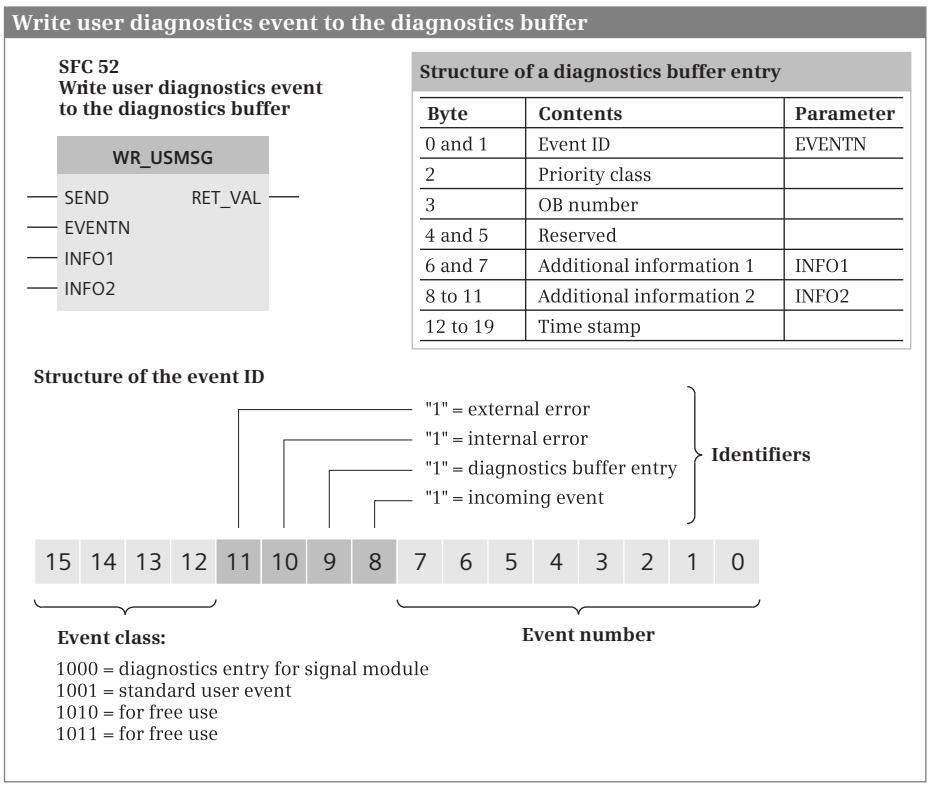


Fig. 5.30 Event ID with diagnostic buffer entries

The structure of the entry in the diagnostic buffer corresponds to the start information of an organization block. You can freely select the event ID (EVENTN parameter) and the additional information (INFO1 and INFO2 parameters) in the context of the permissible assignment.

The event ID is identical to the first two bytes of the buffer entry (Fig. 5.30). Event classes 8 (predefined diagnostics entries for signal modules), 9 (predefined stan-

dard user events), A and B (freely available user events) are permissible for a user entry.

The additional information 1 corresponds to the bytes 6 and 7 of the buffer entry (one word), and additional information 2 to the bytes 8 to 11 (one doubleword). The contents of both tags are freely selectable.

Using SEND = “1” you define that the diagnostic buffer entry is to be sent to the logged-on nodes. Even if sending is not possible (e.g. no display unit is logged-on, or the send buffer is full), the entry is nevertheless made in the diagnostic buffer (if bit 9 of the event ID is set).

5.8.5 System diagnostics with Report System Errors

System diagnostics with “Report System Errors” (RSE) offers system blocks which evaluate information provided by the operating system in the event of an error and prepare corresponding messages. In order to use the RSE diagnostics, activate the corresponding PLC properties and generate the required system blocks.

Settings for the RSE diagnostics

During the hardware configuration, the properties of the selected CPU are displayed in the inspector window on the *Properties* tab. In the CPU properties, select the *System diagnostics* group and activate under *General* the *Activate system diagnostics for this PLC* checkbox.

Under *Alarm settings* you select the alarm category for which an alarm is to be output and whether this alarm must be acknowledged. The following categories are available: *Fault*, *Maintenance demanded*, *Maintenance required*, and *Info*.

Under *System diagnostic blocks* adapt the block names and block numbers of the system blocks used to your project. You define the name and number of the status data block under *Diagnostic support*. If the Web server is activated on the CPU, the diagnostics status DB must also be activated. Further settings apply to the status scanning of I/O stations during startup and the output of a message if an I/O station is activated or deactivated during runtime using the system function D_ACT_DP.

Blocks for the RSE diagnostics

Fig. 5.31 shows the data structure for system diagnostics with Report System Errors. The error organization blocks supported by the respective CPU are created during compilation. The RSE system diagnostics is called in organization block OB 1 and in the associated error organization blocks. The system blocks for RSE diagnostics are generated in accordance with the settings in the CPU properties for system diagnostics, and saved in the project tree under *Program blocks* > *System blocks* > *System diagnostics*.

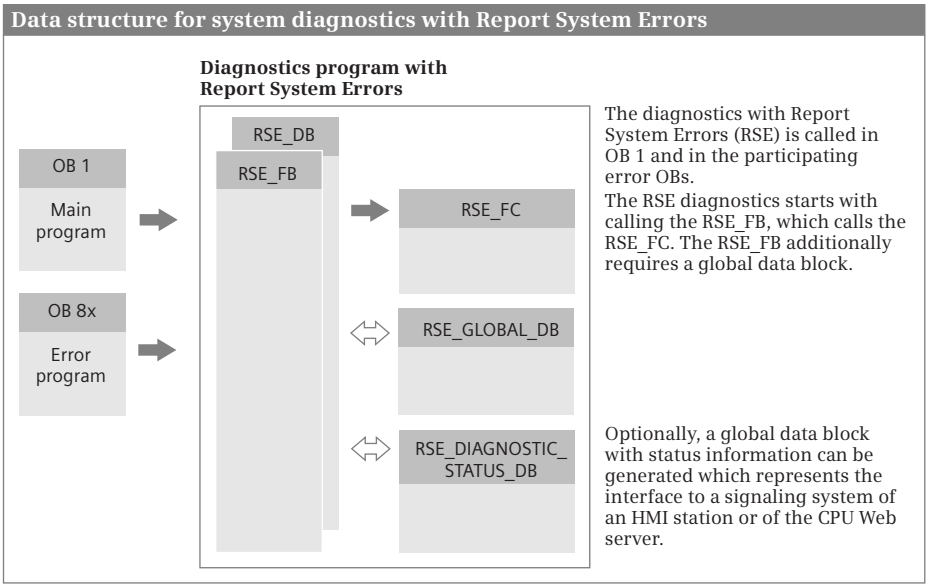


Fig. 5.31 Data structure for the RSE diagnostics

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 249, 8 “Function block diagram FBD” on page 282, 9 “Statement list STL” on page 315, 10 “Structured Control Language SCL” on page 363, and 11 “S7-GRAPH sequential control” on page 397.

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 687.

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”.

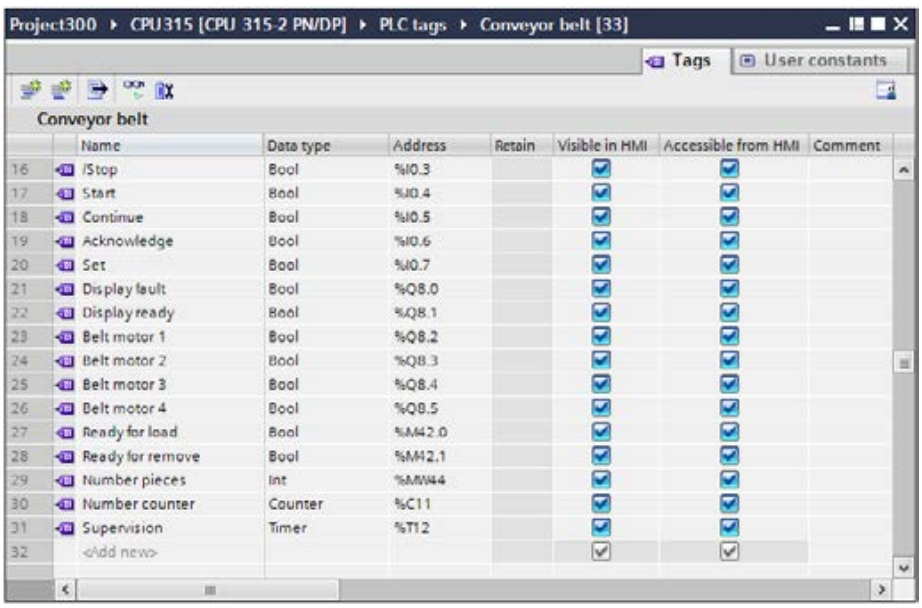
When writing the user program, a distinction is made between *local* and *global* tags. A local tag is only known in the block in which it has been defined. You can use local tags with the same name in different blocks for different purposes. A global tag is known throughout the entire user program and 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 242 for how to create a cross-reference list of the PLC tags. Monitoring of tags using the PLC tag table is described in Chapter 15.5.4 “Monitoring of PLC tags” on page 596.

6.2.1 Working with PLC tag tables

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.



	Name	Data type	Address	Retain	Visible in HMI	Accessible from HMI	Comment
16	!Stop	Bool	%I0.3		☑	☑	
17	Start	Bool	%I0.4		☑	☑	
18	Continue	Bool	%I0.5		☑	☑	
19	Acknowledge	Bool	%I0.6		☑	☑	
20	Set	Bool	%I0.7		☑	☑	
21	Display fault	Bool	%Q8.0		☑	☑	
22	Display ready	Bool	%Q8.1		☑	☑	
23	Belt motor 1	Bool	%Q8.2		☑	☑	
24	Belt motor 2	Bool	%Q8.3		☑	☑	
25	Belt motor 3	Bool	%Q8.4		☑	☑	
26	Belt motor 4	Bool	%Q8.5		☑	☑	
27	Ready for load	Bool	%M42.0		☑	☑	
28	Ready for remove	Bool	%M42.1		☑	☑	
29	Number pieces	Int	%MW44		☑	☑	
30	Number counter	Counter	%C11		☑	☑	
31	Supervision	Timer	%T12		☑	☑	
32	<Add new>				☑	☑	

Fig. 6.1 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.

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 a block, another PLC tag, 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. 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. The SIMATIC timer/counter functions are addressed by 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 300 and the detailed description can be found in Chapter 4 "Tags, addressing, and data types" on page 90.

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.

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).

Table 6.1 Approved operands and data types for PLC tags

Operand	ID	Data types
Input Output Bit memory	%Iy.x, %IBy, %IWy, %IDy %Qy.x, %QBy, %QWy, %QDy %My.x, %MBy, %MWy, %MDy	1 bit: BOOL 8 bits: BYTE, CHAR 16 bits: WORD, INT, S5TIME, DATE 32 bits: DWORD, DINT, REAL, TIME, TIME_OF_DAY
Peripheral input Peripheral output	%IBy:P, %IWy:P, %IDy:P %QBy:P, %QWy:P, %QDy:P	
SIMATIC timer function SIMATIC counter function	%Tn %Cn	TIMER COUNTER

y = byte address, x = bit address, n = number

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 6.3.6 “Editing tags” on page 232).

6.2.3 Comparing PLC tags

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.

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.

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.

With a detailed comparison you compare the contents of the PLC tag tables. To do so, select the tag table and choose the *Start detailed comparison* command from the shortcut menu, or alternatively click on the *Start detailed comparison* icon in the toolbar. 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.

6.2.4 Exporting and importing a PLC tag table

A PLC tag table can also be created or edited using an external editor. The external file is present in .xlsx format.

To export, open the PLC tag table and select the *Export* icon in the toolbar. Set the file name and path in the dialog, and select the data to be exported (tags or constants). The contents of the opened PLC tag table are exported. To export all PLC tags, open the complete table by double-clicking on *Show all tags* and then select the *Export* icon.

The external file contains the *PLC Tags* worksheet for the PLC tags and the *Constants* worksheet for the symbolically addressed user constants (Table 6.2).

To import, double-click on *Show all tags* under the PLC station in the *PLC tags* folder in the project tree. Select the *Import* icon in the toolbar. Set the file name and path in the dialog and select the data to be imported (tags or constants). The contents of the external file are imported into the tag table specified in the *Path* column. Existing entries are identified by a consecutive number in parentheses appended to the name and/or by an address highlighted in color.

Table 6.2 Columns in the external file for the PLC tag table

***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		

6.2.5 Constants tables

A PLC tag table in the *User constants* tab contains symbolically addressed constant values which are valid throughout the CPU. You define a constant in the table in that you assign a name, data type, and fixed value to it and you can then use this constant in the user program with the symbolic name.

The constant name must not already have been assigned to 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.

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.

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.

Select the programming language for the block. 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.

You set the default assignment when creating a new block in the main menu in the Project view using the *Options > Settings* command in the *PLC programming* section. 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.

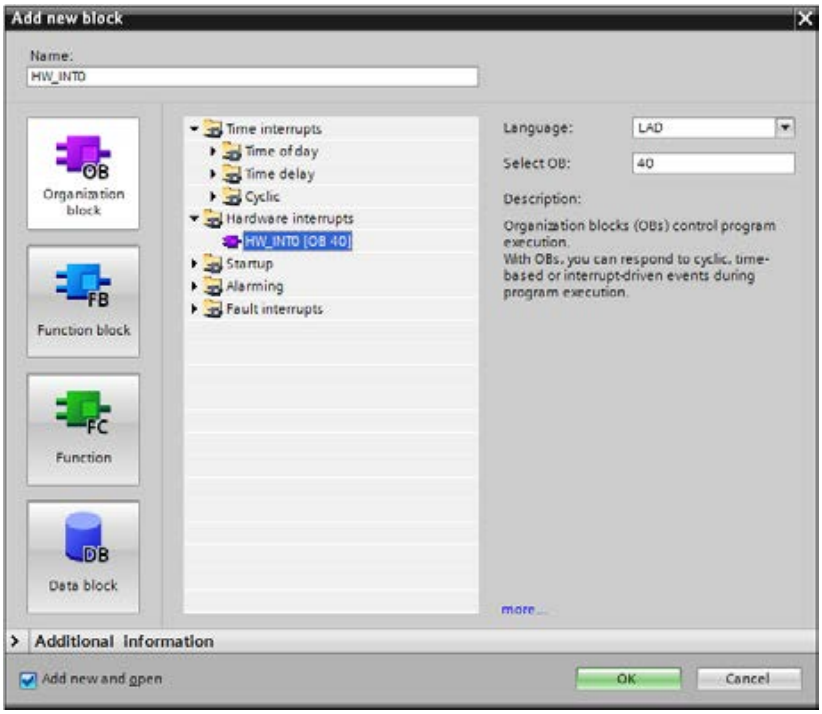


Fig. 6.2 *Add new block* window with organization block selected

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.

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.

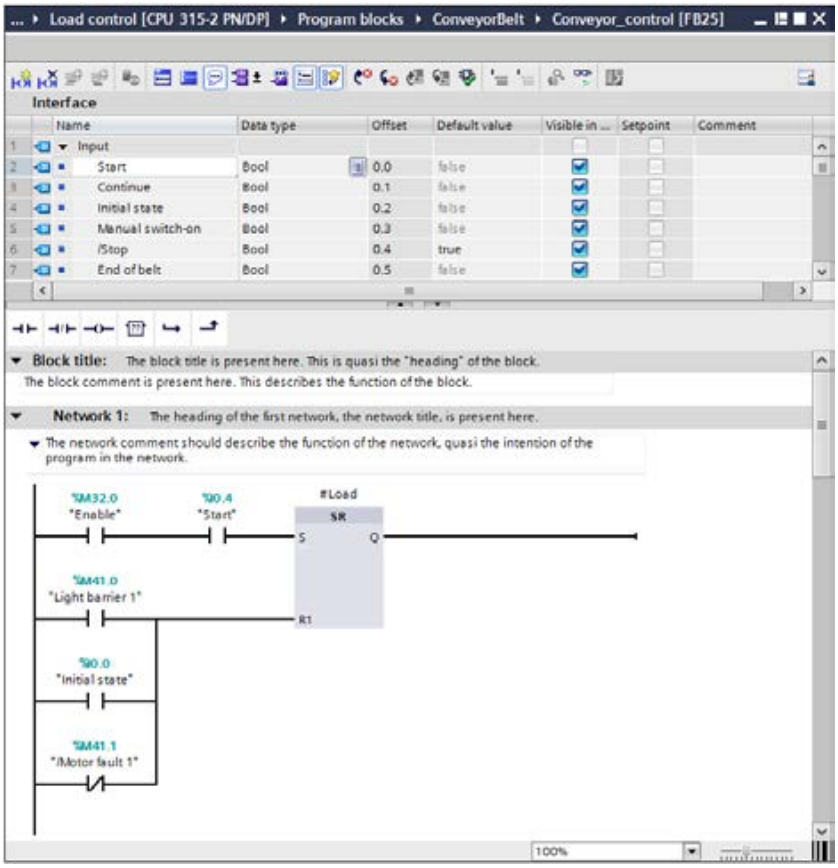


Fig. 6.3 Example of the program editor's working window in ladder logic

- The interface
  shows the block interface with the block parameters and the block-local tags.
- The favorites bar
  provides the favorite program elements (instructions), which can also be found in the *Favorites* section of the program elements catalog. You can activate and deactivate the display in the editor: Click with the right mouse button in the favorites catalog or favorites bar and 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 pos-

sible: 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 can be changed are the block name, block number (not with organization blocks), block title, block comment, block version, block family, author, and a block ID.

A block can be protected against illegal access (know-how protection).

There are attributes depending on the type of block, for example the setting for data type testing when programming the block. The block properties are described in detail in Chapter 5.2.4 “Editing block properties” on page 158.

6.3.4 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 5.2.5 “Block interface” on page 161.

Table 6.3 Declaration sections for code blocks

Declaration section	Meaning	Permissible with block type		
Input	Input parameters	–	FC	FB
Output	Output parameters	–	FC	FB
InOut	In/out parameters	–	FC	FB
Static	Static local data	–	–	FB
Temp	Temporary local data	OB	FC	FB
Return	Function value	–	FC	–

You can increase or decrease the size of the block interface window by dragging on the bottom edge with the mouse. Two arrows at the bottom can be used to open or close the window. Fig. 6.4 shows an example of a function block interface.

You can click on the triangle to the left of the declaration mode to open the declaration section or to close it. If you select a line with the right mouse button, in the shortcut menu you can delete the line, insert an empty line above it, or add an empty line after it.

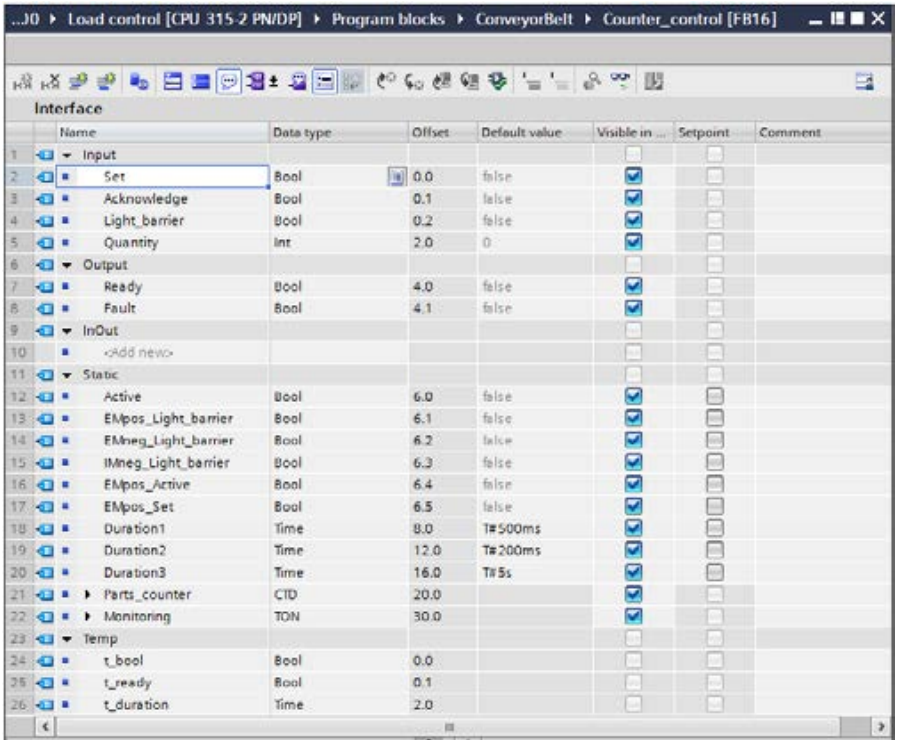


Fig. 6.4 Example of function block interface

The name can contain letters, digits, and special characters (but not quotation marks). No distinction is made between upper and lower case when checking the name. A drop-down list shows the currently permissible data types. You can use the comment to describe the purpose of the respective tag.

The *Default value* column is displayed for a function block (FB). You can enter a default value here which is saved in the instance data block. 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 15.3.6 “Working with setpoints” on page 579.

Each organization block (OB) of a CPU 300 provides start information for the user program. This start information is present in the first 20 bytes of the declaration section *Temp*.

In the case of a function (FC), the function value with the name of the block and data type VOID is displayed in the interface in the *Return* section. The function value has no significance when programming with LAD, FBD, and STL. 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 is displayed as the first output parameter.

Using the SCL programming language, you can integrate a function in an expression instead of a tag with the data type of the function value (see section “Using a function value of a function (FC)” on page 166).

SCL additionally permits an overlay of the tags in the block interface by other data types (see section “Overlaying tags (data type views with SCL)” on page 121).

6.3.5 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.

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 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 a CPU 300C)
- *Communication* (communication functions for data transmission and for communication modules)

You can combine a selection of frequently used program elements in the *Favorites* catalog and display them in the favorites bar of the program editor to allow rapid selection.

General procedure when programming

To enter the program code, position the program elements in the desired arrangement and subsequently supply them with tags or enter the statement lines. The program editor immediately checks your inputs and indicates faulty entries.

You can interrupt block programming at any time – even if the program is still incomplete or faulty – and continue later. You can store a block by saving the complete project using the *Project > Save* command from the main menu.

You can save the structure of the windows and tables using the *Save window settings* icon in the top right corner of the working window. This structure is reestablished the next time the working window is opened.

Programming a control function with ladder 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. 6.5). With the ladder logic, the Q or ENO output is positioned in the display at the top edge of the box in order to be able to "hang" the box into the current path. The structure of an LAD current path is described in Chapter 7 "Ladder logic LAD" on page 249.

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. 6.6). The Q and ENO outputs are positioned at the bottom edge where they can be connected to the input of the following program element. The structure of an FBD logic operation is described in Chapter 8 "Function block diagram FBD" on page 282.

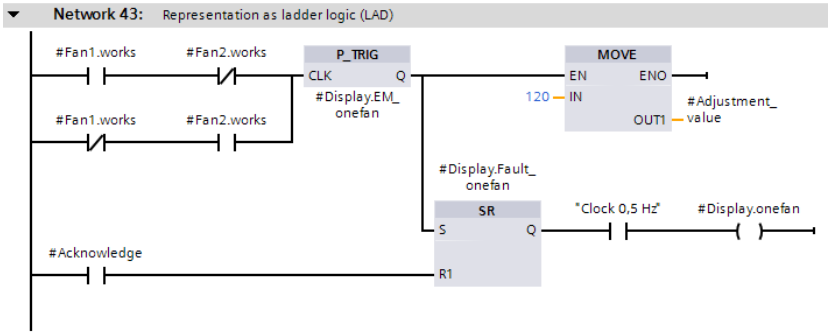


Fig. 6.5 Example of ladder logic representation with contacts, coils, and boxes

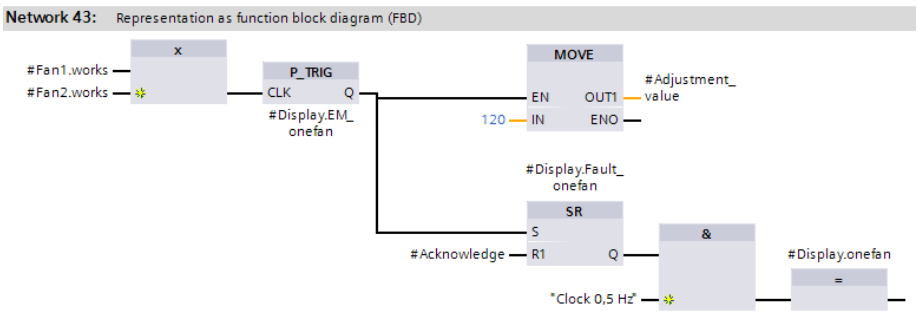


Fig. 6.6 Example of function block diagram with boxes

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. In this case, the drop-down list shows the data types permissible at this point, from which you can select the desired data type.

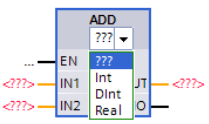


Fig. 6.7 Selection of data type using drop-down list

If a small yellow triangle is displayed in the top right corner of the program element (contact, coil, box), you can select a different function here for the program element from a drop-down list.

The empty box – which can be found in the favorites or in the program elements catalog under *General* – is particularly flexible here. Here you can select almost all program elements from the (function) drop-down list.

Programming a control function with 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.

Binary logic operations are implemented in the representation as statement list by AND, OR, and exclusive OR logic operations (Fig. 6.8). 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 9 “Statement list STL” on page 315.

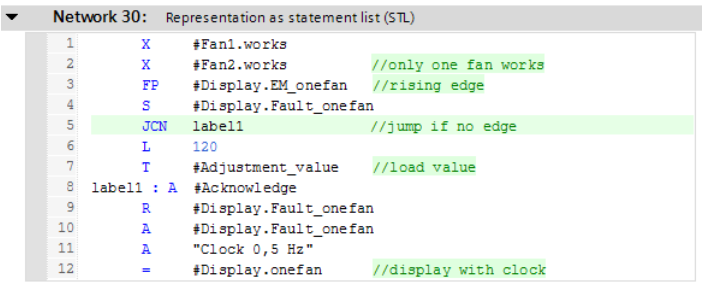


Fig. 6.8 Example of representation as statement list STL

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.

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

```
213  //Representation as structured control language (SCL) *********************************
214 IF (#Fan1.works XOR #Fan2.works) AND NOT #Display.EM_onefan THEN
215    #Display.Fault_onefan := TRUE;
216    #Adjustment_value := 120;
217  END_IF;
218  #Display.EM_onefan := #Fan1.works XOR #Fan2.works;
219 IF #Acknowledge THEN
220    #Display.Fault_onefan := FALSE;
221  END_IF;
222  #Display.onefan := #Display.Fault_onefan & "Clock 0,5 Hz";
223
```

Fig. 6.9 Example of representation as structured control language (SCL)

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 10 “Structured Control Language SCL” on page 363.

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 (Fig. 6.10). You can drag all instructions (steps, transitions, etc.) from the program elements catalog into the working area. 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 397.

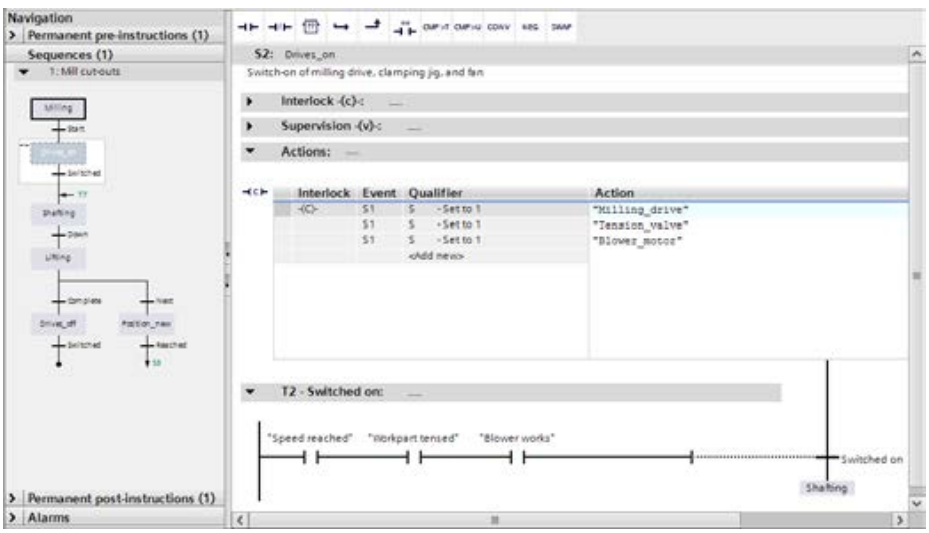


Fig. 6.10 Example of representation as sequence control GRAPH

6.3.6 Editing tags

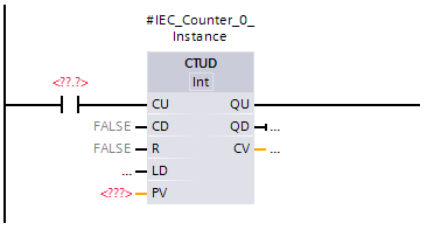
Almost all program elements require tags in order to execute their function. Following insertion in the working area, a program element must be supplied with tags. Fig. 6.11 shows the insertion of an up/down counter as local instance (*#IEC_Counter_0_Instance*) in a function block. The example shows the representation in LAD, FBD, STL, and SCL.

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.

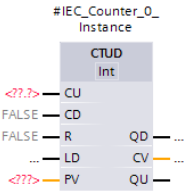
If you set the cursor to a block parameter or function parameter in STL, the declaration mode and the data type of the parameter are shown.

Supply of an IEC counter function with tags

Call in LAD



Call in FBD



Call in STL

```
1
2       CALL  #IEC_Counter_0_Instance_1
3         Int
4         CU :=
5         CD :=
6         R  :=
7         LD :=
8         PV :=
9         QU :=
10        QD :=
11        CV :=
12
```

Call in SCL

```
239 #IEC_Counter_0_Instance_1(CU:=_bool_in_,
240                           CD:=_bool_in_,
241                           R:=_bool_in_,
242                           LD:=_bool_in_,
243                           PV:=_in_,
244                           QU=>_bool_out_,
245                           QD=>_bool_out_,
246                           CV=>_out_);
247
```

Fig. 6.11 Supply with tags

With SCL, the missing tags are occupied by dummy values which have to be replaced by “real” tags.

The program editor displays the global tags enclosed by quotation marks. Local tags are preceded by a number 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 program editor provides support for the input of tags: If you enter the first letters of a tag when entering those which are still missing, the editor provides a list of (previously defined) tags which can be considered for the current data type. You can then choose the desired tag.

The data type of the tag must correspond to the data type of the supply position. If the *IEC check* attribute is activated in the block, it must be exactly the same data type. If the attribute is deactivated, it is sufficient if the tag has the appropriate data width.

If you enter an operand with the appropriate data width which is not present in the PLC tag table, the editor creates a new "*Tag_x*" in the PLC tag table, with x as a consecutive number. By clicking with the right mouse button on a tag and selecting *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 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. 6.12). You can also select the operand area in which the tag is to be positioned: Input, output or in/out parameter, static or temporary local data, bit memories, inputs or outputs.

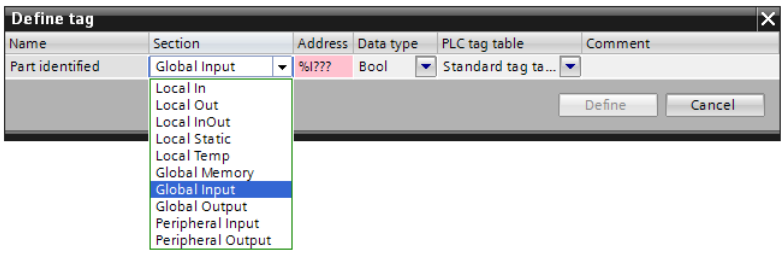


Fig. 6.12 Defining tags during program input

6.3.7 Working with program comments

With LAD and FBD as the programming languages, you can enter a "free-form comment" for each coil or box (LAD) and for each non-binary box (FBD). Right-click on the program element and select *Insert comment* from the shortcut menu. The program editor displays a comment box with an arrow pointing to the selected program element. You can then enter a comment in the box. You can shift the box within the network or increase its size using the triangle at the bottom right corner (Fig. 6.13).

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

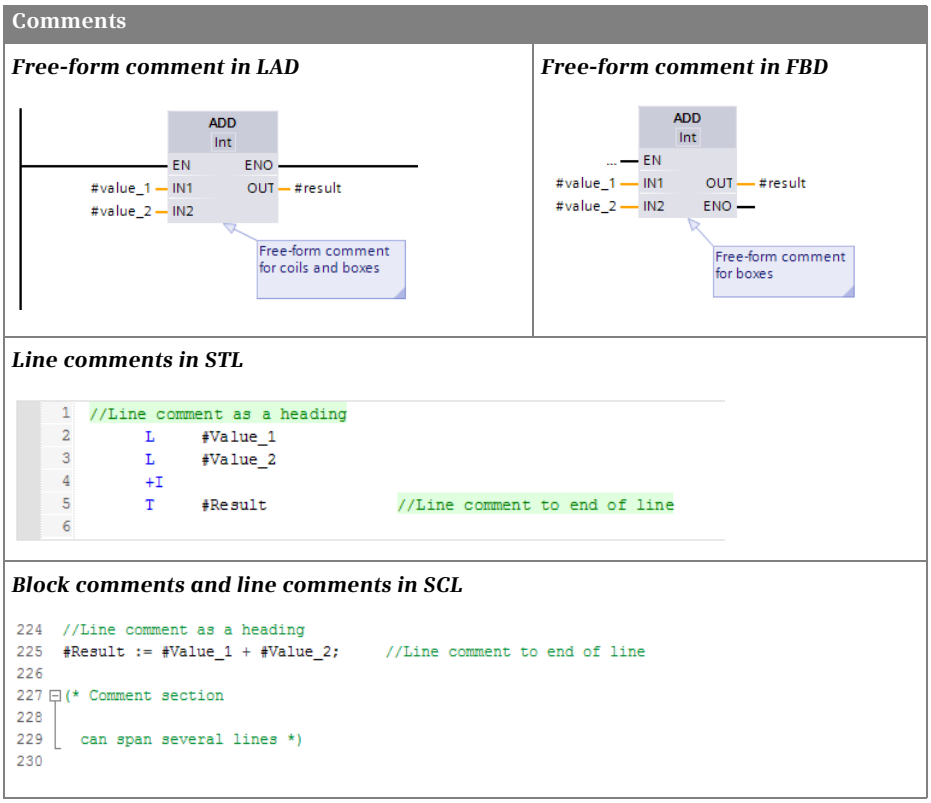


Fig. 6.13 Comments in the various programming languages

position it after the STL statement. You can eliminate the comment in a code line by positioning the cursor in the line 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.

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 eliminate the comment in a code line by positioning the cursor in the line 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.

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 function block (SFB). The data structure is defined during programming of the block interface (for an FB) or is prespecified and cannot be changed (for an SFB).
- 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. 6.14):

- 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*.

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 which can be changed are the block name, block number, block title, block comment, block version, block family, author, and a block ID. A data block can be protected against illegal access (know-how protection).

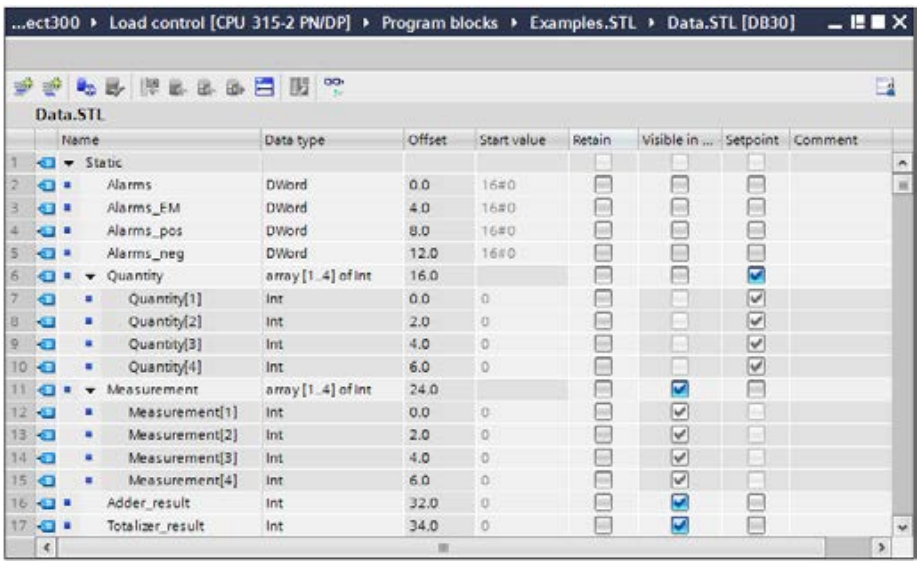


Fig. 6.14 Example of the program editor's working window for data blocks

Attributes also exist depending on the block type, for example the setting for write protection. The block properties are described in detail in Chapter 5.2.4 “Editing block properties” on page 158.

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 shows – following compilation – the relative address of the tag at the beginning of the data block.
- 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 (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 tags in the data block are retentive. The setting applies to the complete data block.
- Visible in HMI: A checkmark in this column means that the tag is visible in the drop-down list of HMI stations by default.
- 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 15.3.6 “Working with setpoints” on page 579.
- 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, all tags with data types ARRAY, STRUCT as well as the PLC data types 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 defined PLC data types which you can also assign to a data tag.

6.5 Compiling blocks

Compilation generates a program code which can execute in the CPU. A compilation process is always triggered prior to downloading the user program to the PLC station. Only blocks which have been compiled without errors can be downloaded.

It is recommendable to also trigger compilation while generating the user program to enable a quick response to any programming errors.

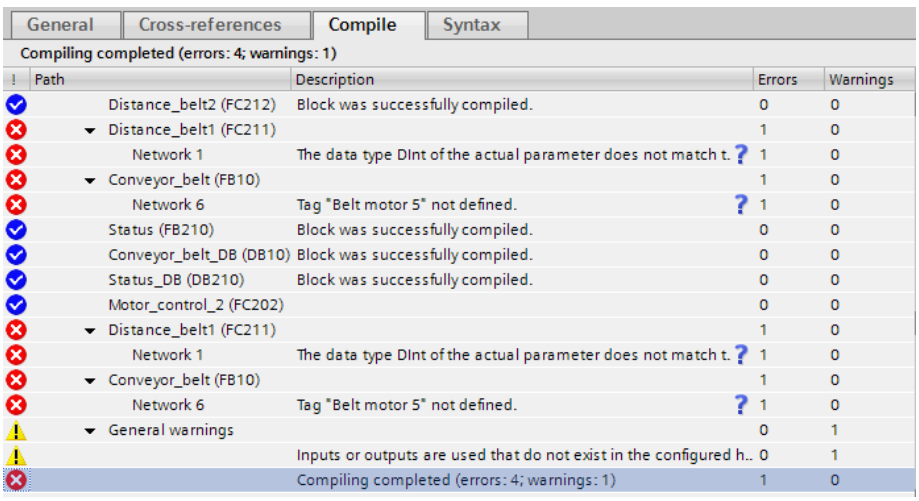
6.5.1 Starting the compilation

You start the compilation using a command from the shortcut menu.

▷ To compile a block opened in the program editor, click with the right mouse button on the white background of the working area and select the *Compile* command 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.
- 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 (changes only)* and
  - *... Software (rebuild all blocks)*.
- If you select the *PLC station* folder and then *Compile > ...* from the shortcut menu, you can select between
  - *... Hardware and software (changes only)*,
  - *... Hardware (changes only)*,
  - *... Software (changes only)* and
  - *... Software (rebuild all blocks)*.

The result of the compilation is displayed in the inspector window in the *Info* tab under *Compile* (Fig. 6.15). Any warnings which have been detected do not prevent continuation of the compilation. Any errors which have been detected are displayed in the result of the compilation and end the compilation.



!	Path	Description	Errors	Warnings
✓	Distance_belt2 (FC212)	Block was successfully compiled.	0	0
✗	Distance_belt1 (FC211)		1	0
✗	Network 1	The data type DInt of the actual parameter does not match t.	1	0
✗	Conveyor_belt (FB10)		1	0
✗	Network 6	Tag "Belt motor 5" not defined.	1	0
✓	Status (FB210)	Block was successfully compiled.	0	0
✓	Conveyor_belt_DB (DB10)	Block was successfully compiled.	0	0
✓	Status_DB (DB210)	Block was successfully compiled.	0	0
✓	Motor_control_2 (FC202)		0	0
✗	Distance_belt1 (FC211)		1	0
✗	Network 1	The data type DInt of the actual parameter does not match t.	1	0
✗	Conveyor_belt (FB10)		1	0
✗	Network 6	Tag "Belt motor 5" not defined.	1	0
⚠	General warnings		0	1
⚠		Inputs or outputs are used that do not exist in the configured h..	0	1
✗		Compiling completed (errors: 4; warnings: 1)	1	0

Fig. 6.15 Example of compilation information in the inspector window

6.5.2 Compiling SCL blocks

You can set the attributes for compilation in the properties of SCL blocks. The setting for all newly created blocks is specified in the main menu under *Options* > *Settings* and *PLC programming* > *SCL* > *Compile*.

- Create extended status information
  Permits monitoring of all tags in a block.
- Check ARRAY limits
  Checks the limits of ARRAY tags during runtime and sets ENO to signal state “0” in the event of a limit violation.
- Set ENO automatically
  Checks whether errors have occurred in program execution during runtime and sets ENO to signal state “0” in the event of an error.

Activation of one of the attributes increases the memory requirements and processing time of the block.

6.5.3 Eliminating errors following compilation

An error is indicated by a white cross on a red circle in the line of the faulty block. Click on the triangle to the left of the block name to open the list with the compilation messages.

Click on the blue question mark in an error message to display more information about the error. Double-clicking on an error message displays the program environment of the selected error in the working window so that you can correct the error directly.

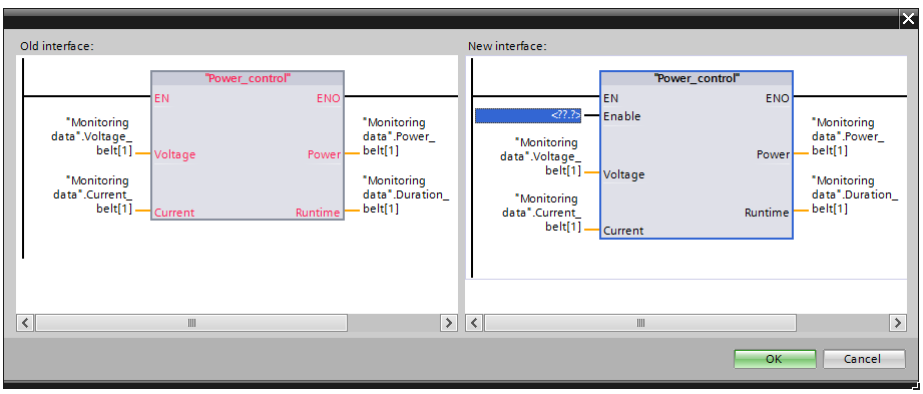


Fig. 6.16 Interface update in the case of faulty block calls

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 dialog which you can import unchanged or following modification (Fig. 6.16).

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. 6.17). 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 expanded, a link in the *Point of use* column leads directly to the program position at which 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*.

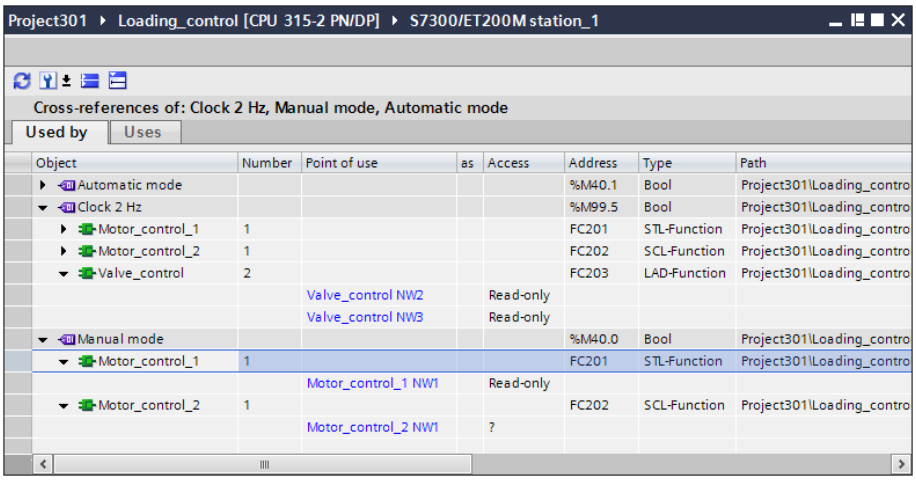


Fig. 6.17 Example of a cross-reference list in the *Used by* view

Cross-reference list *Uses*

The *Uses* view displays the objects used by the referenced object. It shows which objects are used (Fig. 6.18). 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 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.

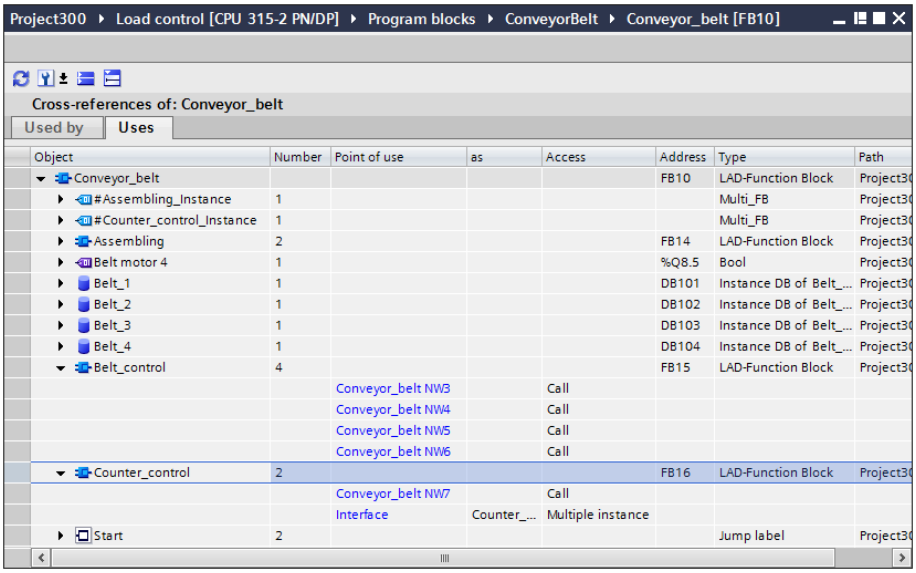


Fig. 6.18 Example of a cross-reference list in the *Uses* view

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. 6.19).

Display of input/output assignment

A yellow background for inputs and outputs (in the left window Fig. 6.19, grey background for the diamonds) 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.

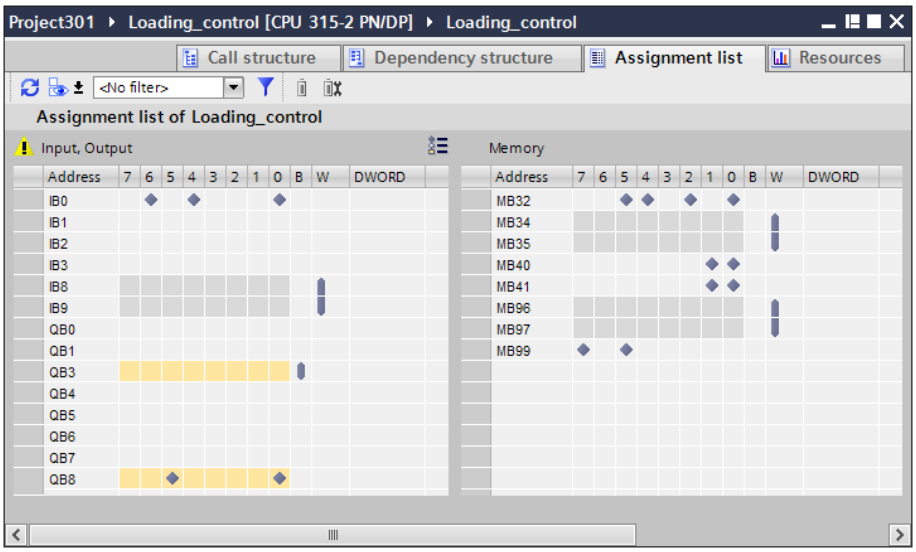


Fig. 6.19 Example of an assignment list

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 area (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. 6.20). 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.

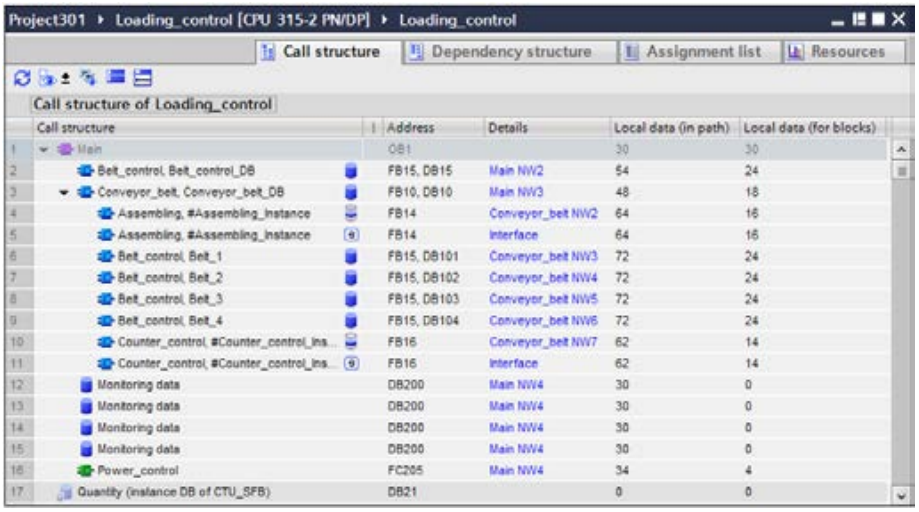


Fig. 6.20 Example of the call structure

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 247.

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.

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. 6.21).

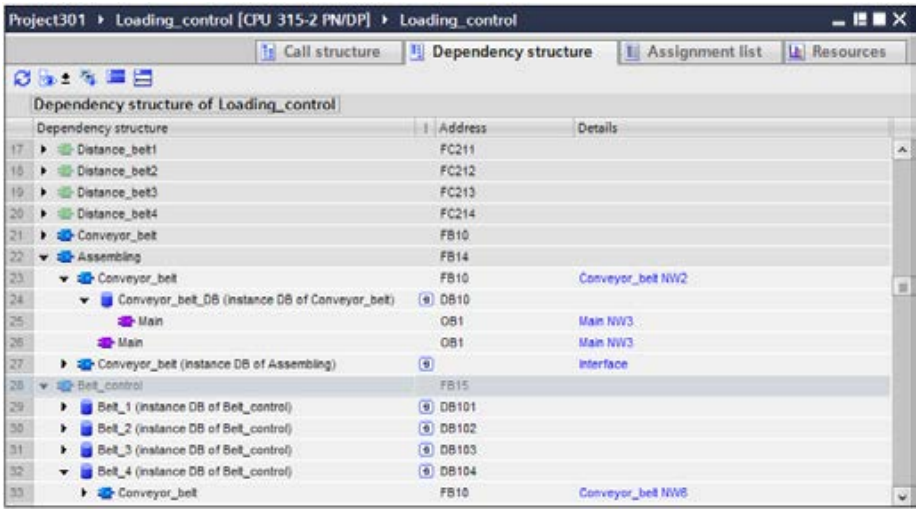


Fig. 6.21 Example of dependency structure

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 247.

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.

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 Memory utilization of the CPU

Under Resources you can see the utilization of the user memory and of the existing input/output modules (Fig. 6.22). To display the resources, first select the *PLC station* folder, *Program blocks* folder, or individual blocks and then select the *Resources* command from the shortcut menu or the *Tools* > *Resources* command from the main menu.

The resources function shows in three columns the maximum available and actual utilization of the load memory, work memory, and retentive memory. You can see the utilization for each type of block, for individual blocks, and for the PLC tags. marks represent blocks which have not yet been compiled. The values are then displayed in red in the total lines.

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.

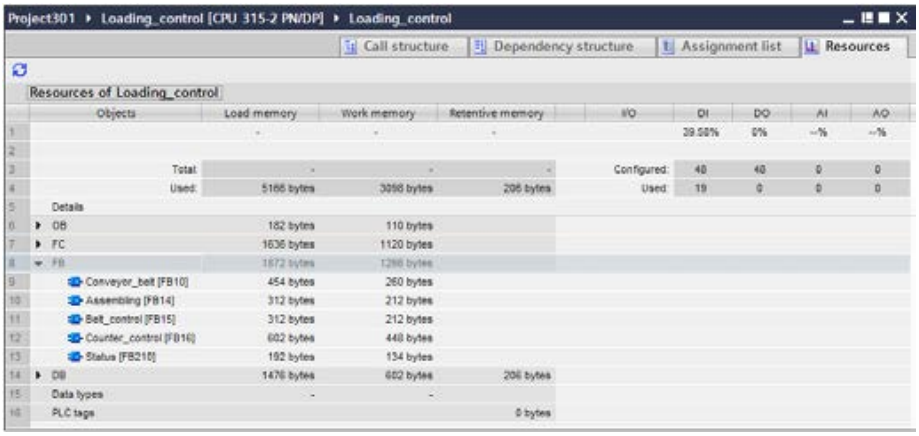


Fig. 6.22 Example of display of memory utilization

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 LAD. You can find a description of the individual functions, e.g. comparison functions, in Chapters 12 “Basic functions” on page 427, 13 “Digital functions” on page 475, and 14 “Program flow control” on page 530.

Use of the program and symbol editor, which generally applies to all programming languages, is described in Chapter 6 “Program editor” on page 218.

LAD is used to program the contents of blocks (the user program). What blocks are and how they are created is described in Chapters 5.2.3 “Block types” on page 156 and 6.3 “Programming a code block” on page 223.

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 structure of a block with the LAD program. Located at the beginning of the program is the block header (block title) and the block comment. Heading and comment are optional. These are followed by the first network with its number, heading and comment. Heading and comment are also optional for the networks. The first network shows a current path as an example with series and parallel connection of contacts, a memory function within the current path, and two coils as termination of the current path. The second network shows the processing of boxes which can be arranged in series or parallel. A block is not terminated by a special network or function; you simply finish the program input.

The LAD editor establishes a network in accordance with the principle of the “main current path”: This is the highest branch which commences directly 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.

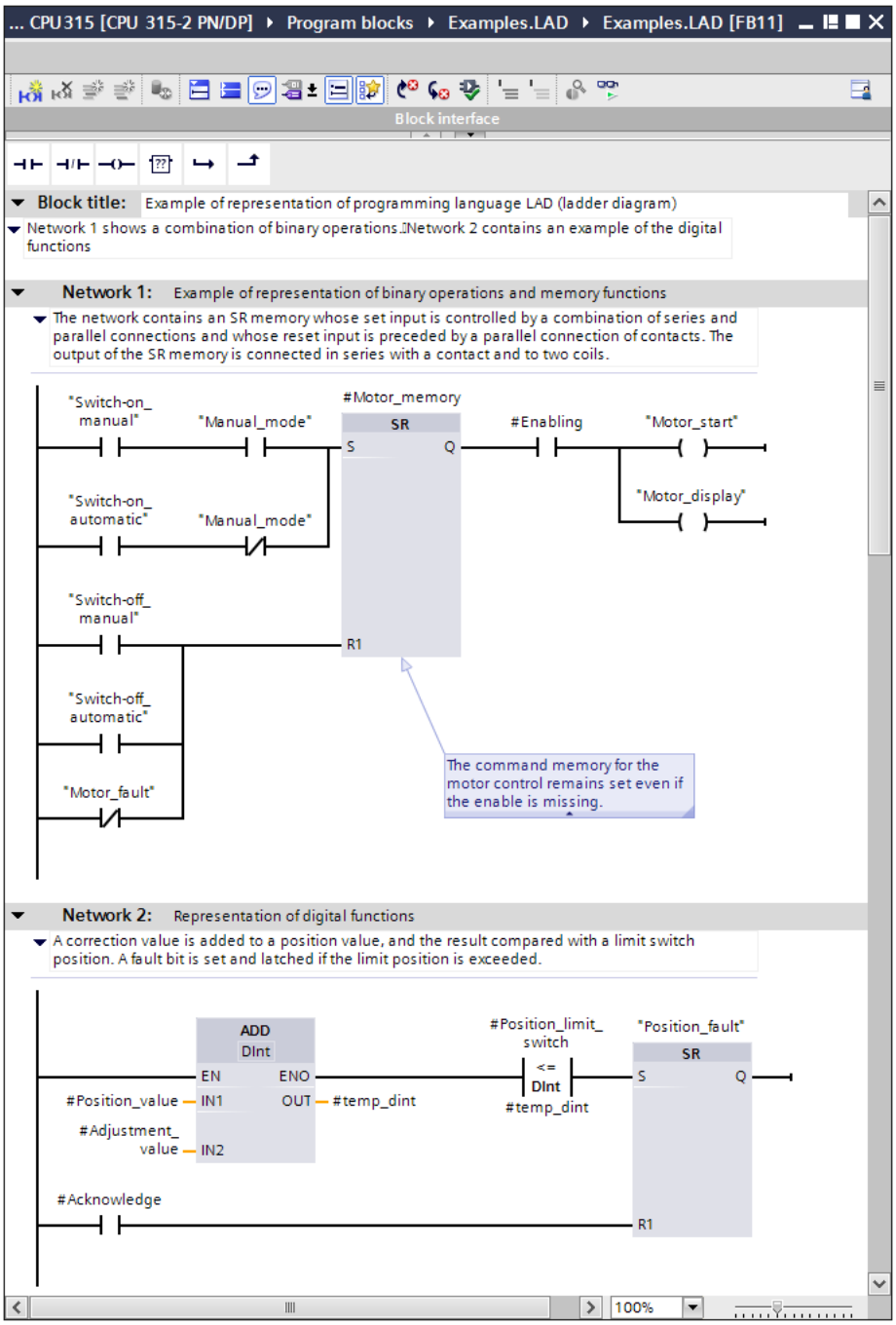


Fig. 7.1 Structure of a block with LAD program

"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.

Where additional rules apply to the arrangement of special LAD elements, these are described in the corresponding sections.

7.1.2 Program elements of ladder logic

Fig. 7.2 shows which types of LAD elements exist: Contacts and coils for processing binary signals, Q boxes for implementing memory, timer and counter functions, and EN/ENO boxes for "complex" functions which, for example, carry out calculations, manipulate strings, or convert numbers into text.

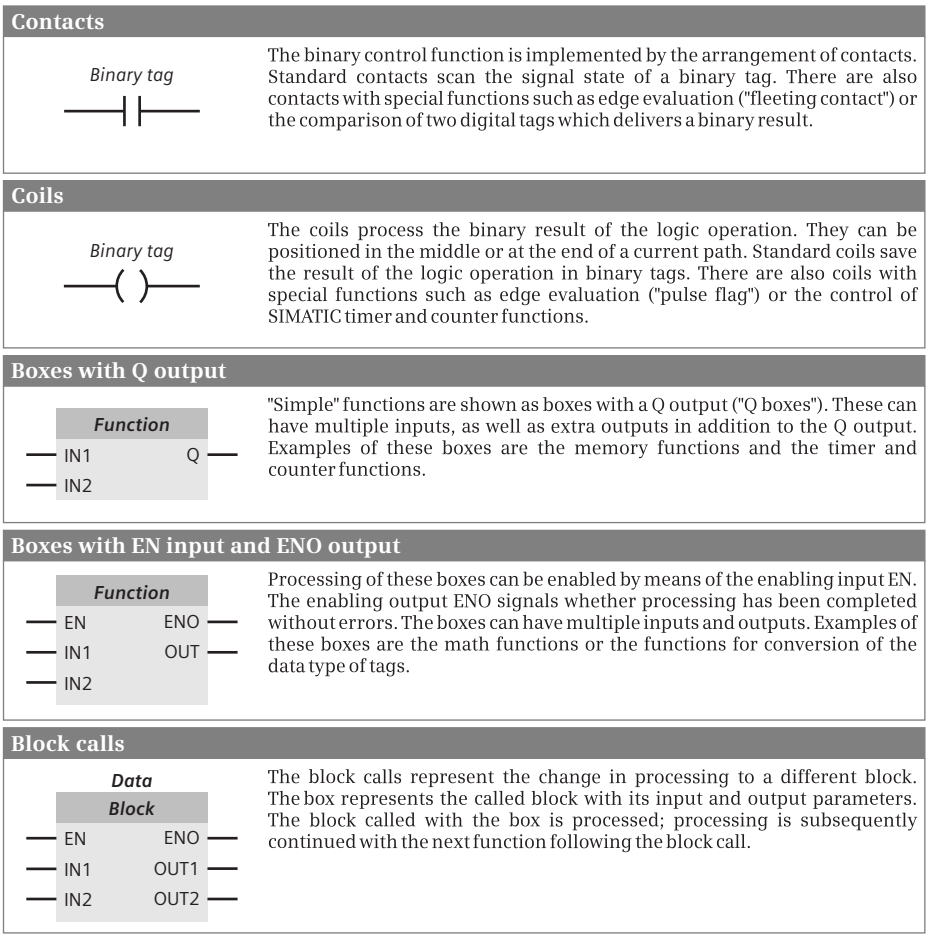


Fig. 7.2 Overview of ladder logic program elements

Most program elements must be provided with tags or operand addresses. With contacts and coils, the tags are assigned by means of the program element. If further tags are required for the function, these are present under the element. In the case of the boxes, the tags are present at the box inputs and outputs.

It is best if you initially arrange all program elements in a current path and subsequently label them.

7.2 Programming 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 for a binary tag, and the comparison function (Fig. 7.3).

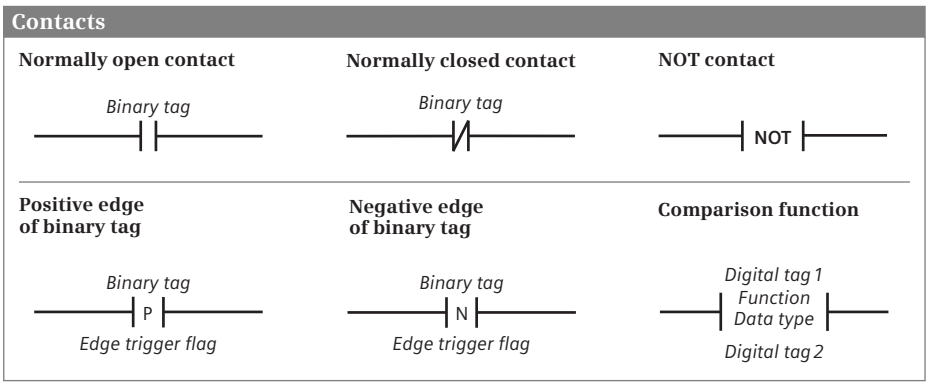


Fig. 7.3 Overview of the contacts described in this chapter

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 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.

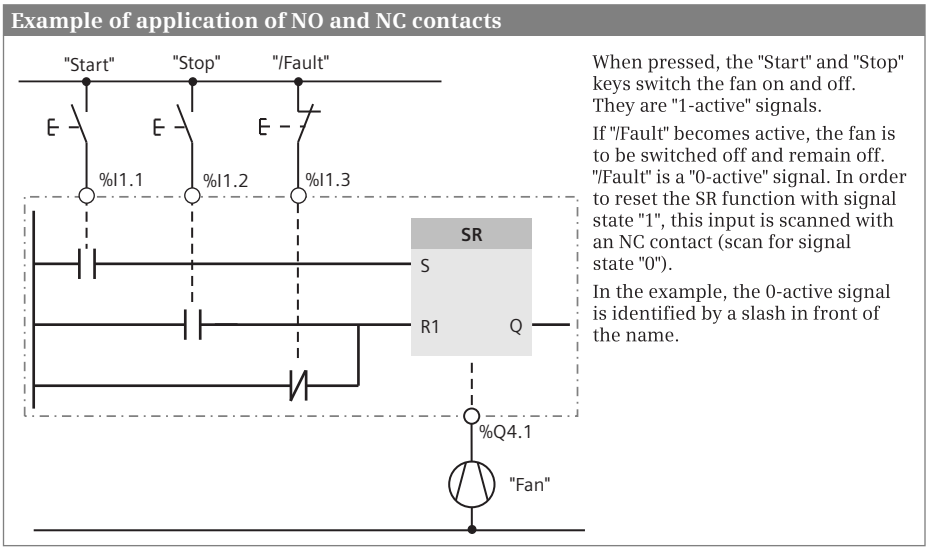


Fig. 7.4 Principle of operation of NO and NC contacts

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 431).

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 432).

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. The logic operation in network 3 is: (*#Fan1.works* AND not *#Fan2.works*) OR (not *#Fan1.works* AND *#Fan2.works*). Network 4 solves the task with the logic operation (*#Fan1.works* OR *#Fan2.works*) AND (not *#Fan1.works* OR not #*Fan2.works*).

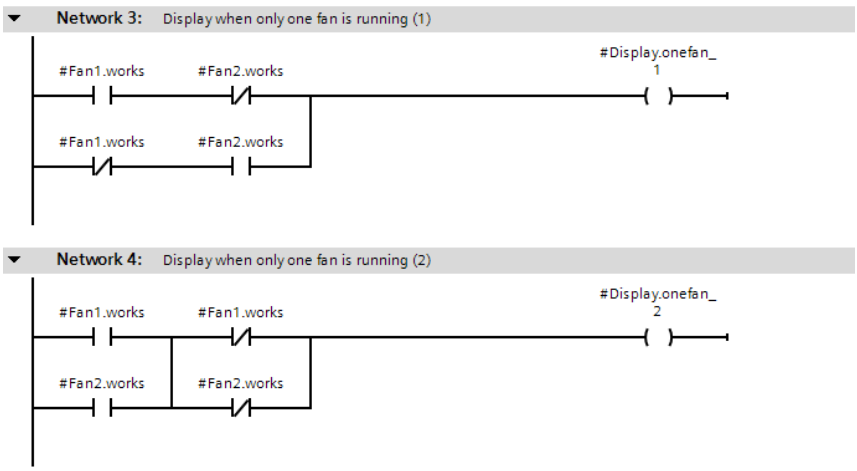


Fig. 7.5 Example of series and parallel connection of contacts

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.

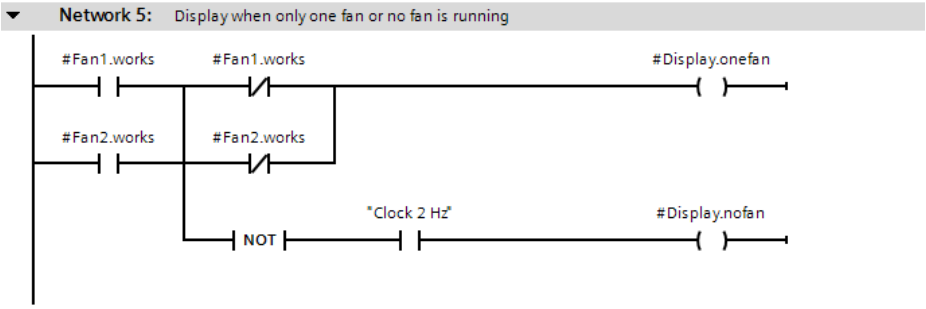


Fig. 7.6 Example of a T branch (open parallel branch) and the NOT contact

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 Negating result of logic operation

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

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 12.2.5 “Edge evaluation” on page 438).

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 *#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”).

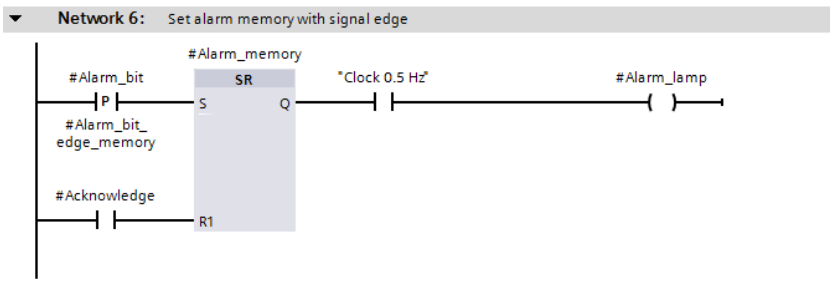


Fig. 7.7 Example of an edge evaluation of a binary tag

7.2.6 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 487.

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 set the comparison mode and data type (Fig. 7.8).

Fig. 7.9 shows two comparison contacts. If the *#Measurement_temperature* tag is above a lower limit and below an upper limit (series connection), the coil is activated and the *#Measurement_in_range* tag is set.

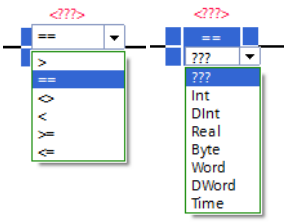


Fig. 7.8 Drop-down lists for setting the comparison mode and data type

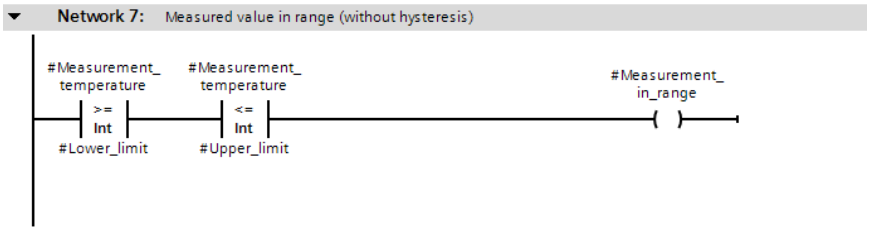


Fig. 7.9 Example of comparison contacts

7.3 Programming memory functions with LAD

Coils control binary tags such as outputs or bit memories. Coils are available with an additional inscription and a special functionality such as coils with timer or counter functions (Fig. 7.10). Further coils such as those for jumps or the opening of a data block are described in Chapter 14 “Program flow control” on page 530.

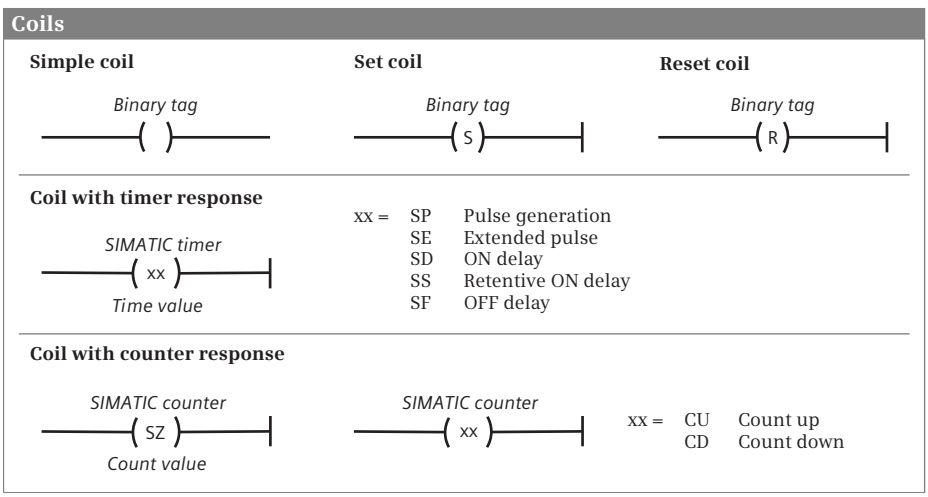


Fig. 7.10 Overview of the coils described in this chapter

7.3.1 Simple coil, assignment

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.

To program a simple coil, drag it with the mouse from the program elements catalog under *Basic instructions* > *Bit logic operation* to the working area. Gray boxes in-

dicate the permissible positioning, a green box identifies the position at which the coil will be inserted if you release the mouse button.

The simple coil requires a preceding logic operation; it 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 coils can be connected in series or – at the end of a current path – in parallel. Simple coils do not change the result of the logic operation (the “current flow”).

Fig. 7.11 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.

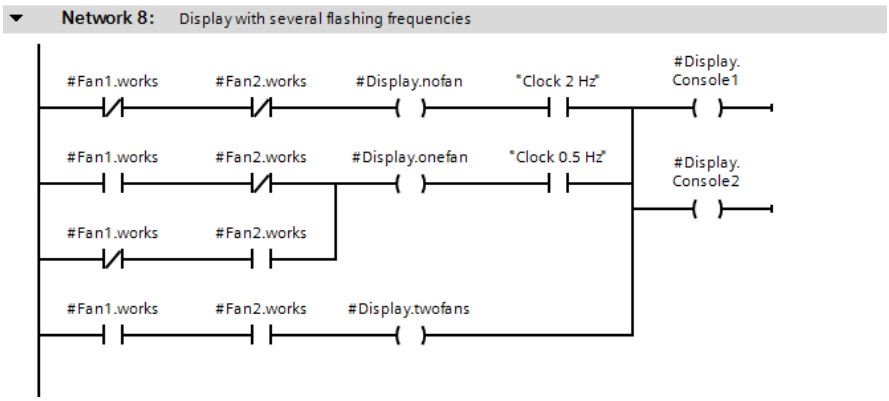


Fig. 7.11 Example of arrangement for a simple coil

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.

To program the corresponding coil, drag it 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.

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 or -counter function.

In Fig. 7.12, *#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.

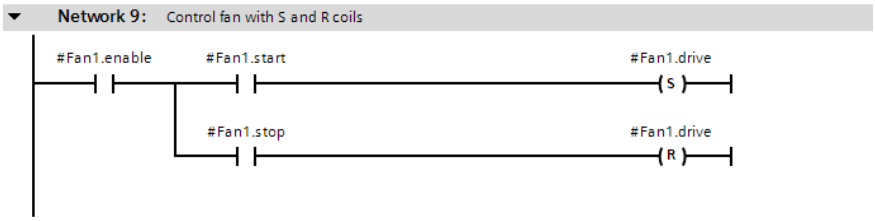


Fig. 7.12 Example of set and reset coils

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. 7.13 shows both types of memory function through latching, namely set dominant and reset dominant. Network 10: 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*, no current flows into the coil (reset dominant). This situation looks different in network 11: If signal state "1" is present at both *#Fan3.start* and *#Fan3.stop*, current flows into the coil (set dominant).

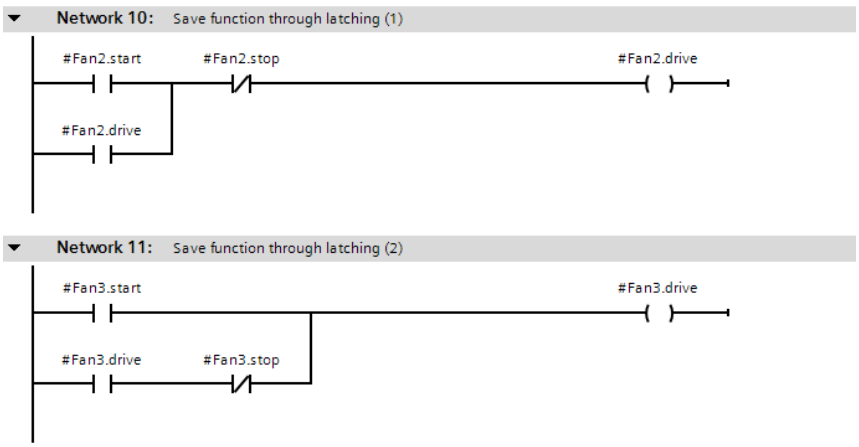


Fig. 7.13 Retentive response due to latching

7.3.4 Coils with time response

A coil with time response is a single element of a SIMATIC timer function. The timer function is usually applied as a box which contains all elements. The coil with time response corresponds to the S input of the time box. Attention must be paid to the sequence in the program when using the single elements. The time response of the coils is described in Chapter 12.3 “SIMATIC timer functions” on page 443.

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 and terminates a current path. It can be connected parallel to all other coils. Positioning at the end of a T branch is also possible.

The time tag is positioned above the coil with a time response. This is an operand from the range of SIMATIC timers (T). The time value is specified in the data format S5TIME underneath the coil.

In Fig. 7.14, the timer *“Fan4.delay”* is started by the positive edge of *#Fan4.start*. Following expiry of the duration (5 s in the example), the fan *#Fan4.drive* is switched on. If *#Fan4.start* has the signal state “0” prior to expiry of the duration, the fan is not even switched on.

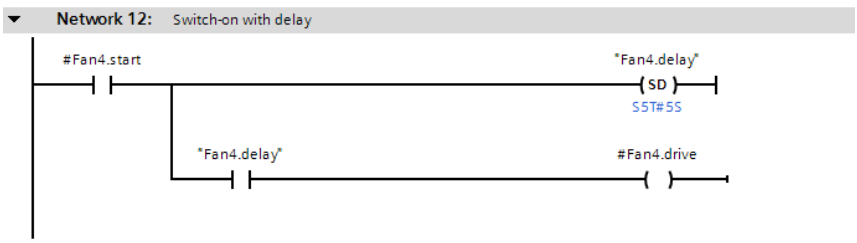


Fig. 7.14 Example of a coil with time response

7.3.5 Coils with counter response

A coil with counter response is a single element of a SIMATIC counter function. The counter function is usually applied as a box which contains all elements. The SC coil corresponds to the S input of the counter box, the CU coil to the CU input, and the CD coil to the CD input. Attention must be paid to the sequence in the program when using the single elements. The response of these coils is described in Chapter 12.5 “SIMATIC counter functions” on page 462.

For programming, drag the corresponding coil with the mouse from the program elements catalog under *Basic instructions* > *Counter operations* to the working area.

A current path is terminated by a coil with counter response. It can be connected parallel to all other coils. Positioning at the end of a T branch is also possible.

The counter tag is positioned above the coil with a counter response. This is an operand from the range of SIMATIC counters (C). The counter 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.

In Fig. 7.15, the switch-on processes of *#Fan1.start* are counted with the SIMATIC counter "*Fan1.number*". The *#Acknowledge* signal resets the counter to 0.

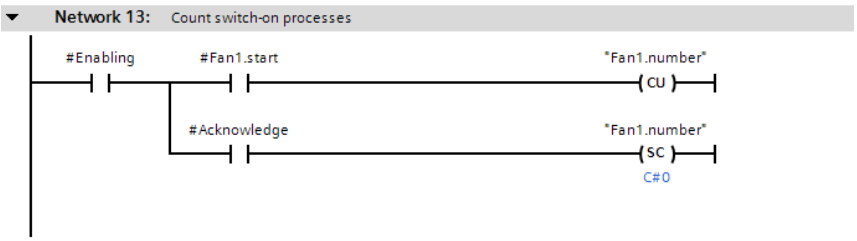


Fig. 7.15 Example of coils with counter response

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. 7.16).

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

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 435.

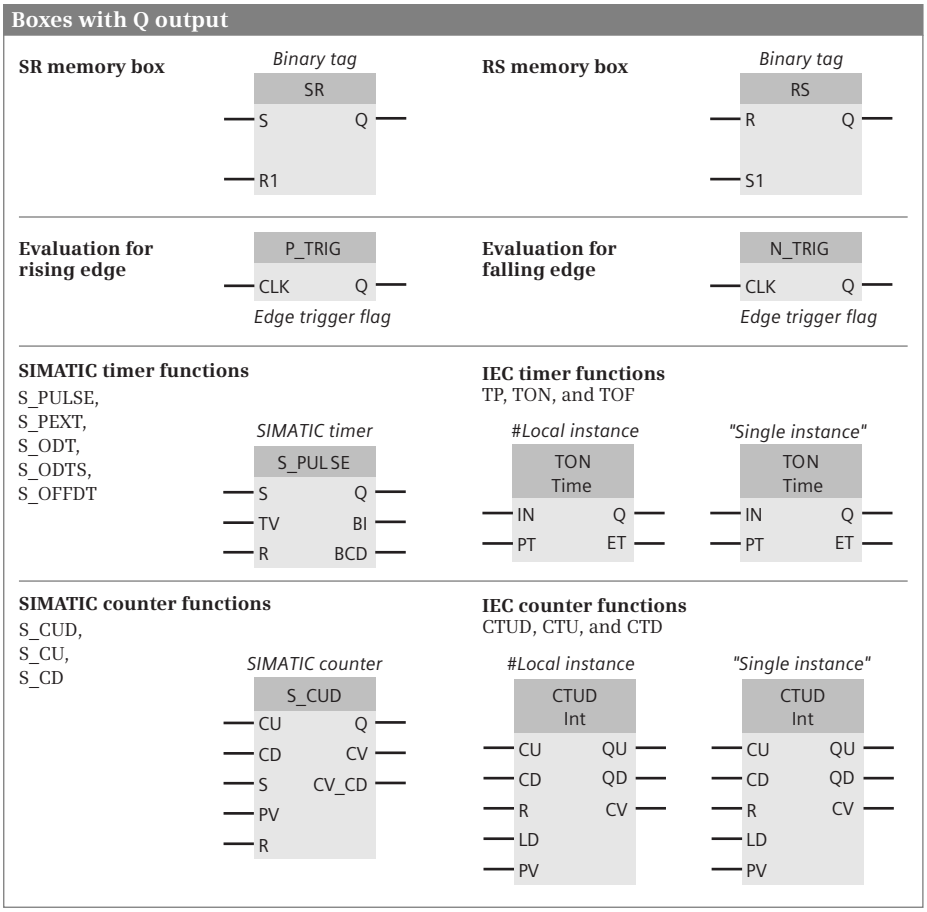


Fig. 7.16 Overview of Q boxes available with LAD

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. 7.17 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.

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 435.

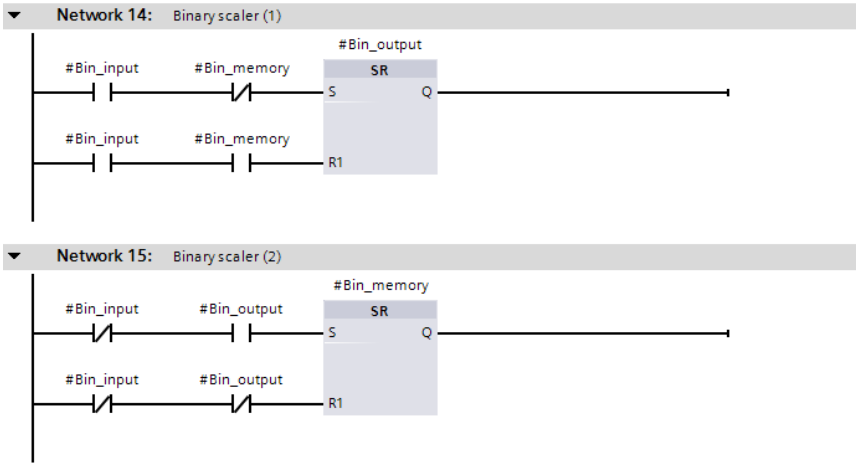


Fig. 7.17 Example of binary scaler

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. 7.18, *#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 message memory are set if the upper limit is exceeded again.

7.4.3 SIMATIC timer functions

Timer functions are used to implement dynamic processes in the user program. The box of a SIMATIC timer function contains all instructions required for the sequence. A detailed description of the SIMATIC timer functions is provided in Chapter 12.3 “SIMATIC timer functions” on page 443.

For programming, drag the corresponding symbol S_PULSE, S_PEXT, S_ODT, S_ODTS or S_OFFDT 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.

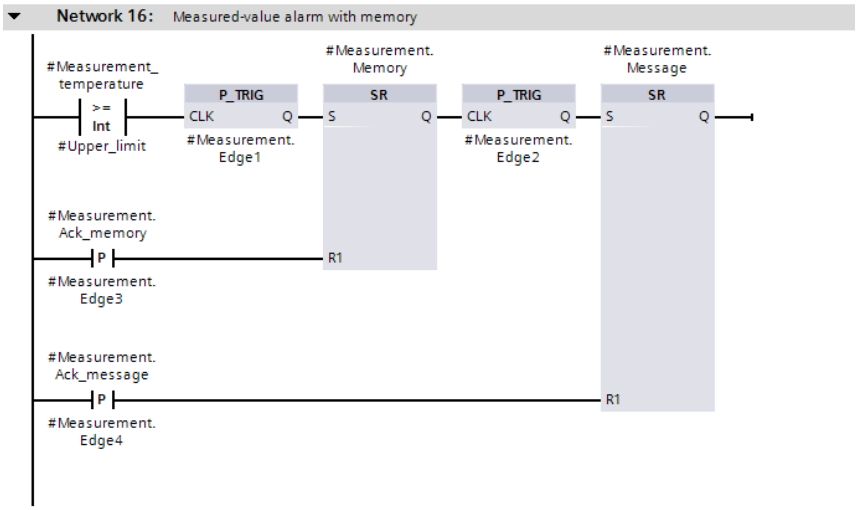


Fig. 7.18 Example of edge evaluations of current flow

Fig. 7.19 shows a switch-on and switch-off delay. The timer function *"Fan5.on-delay"* is started by *#Fan5.start*. The Q output has signal state "1" after 3 s, which starts the timer function *"Fan5.Off-delay"*. At the same time, the *#Fan5.drive* tag is started by the Q output of the box. The Q output still has signal state "1" for 5 s after *#Fan5.start* has signal state "0".

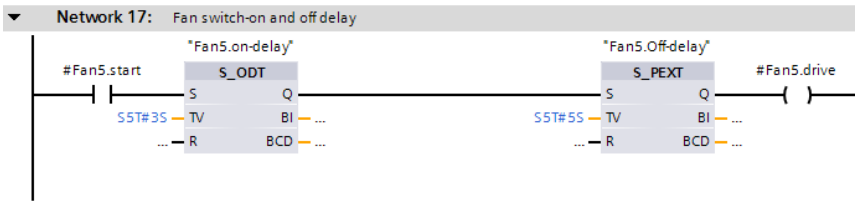


Fig. 7.19 Example of SIMATIC timer functions in the ladder logic

7.4.4 SIMATIC counter functions

Counter functions are used to implement counting tasks in the user program. The box of a SIMATIC counter function contains all instructions required for the sequence. A detailed description of the SIMATIC counter functions is provided in Chapter 12.5 "SIMATIC counter functions" on page 462.

For programming, drag the corresponding symbol (S_CUD, S_CU or S_CD) 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. 7.20 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.

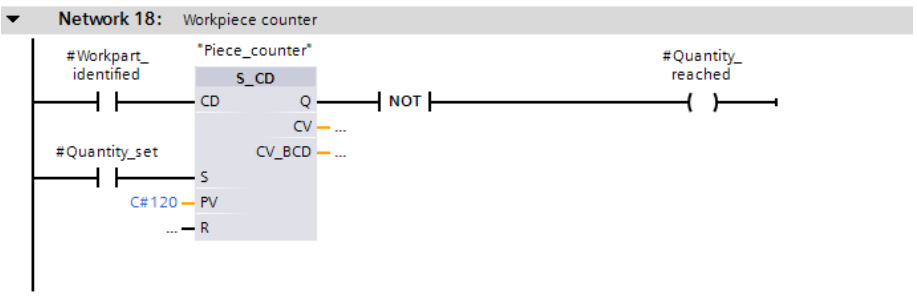


Fig. 7.20 Example of SIMATIC counter functions in the ladder logic

7.4.5 IEC timer functions

Timer functions are used to implement dynamic processes in the user program. With a CPU 300, an IEC timer function is a system function block (SFB) in the operating system. A detailed description of the IEC timer functions is provided in Chapter 12.4 “IEC timer functions” on page 459.

For programming, drag the corresponding symbol (TP, TON or TOF) 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. 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 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.

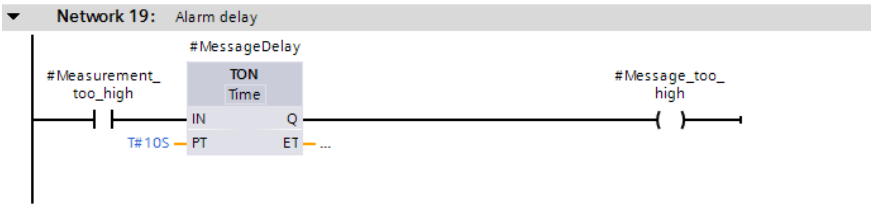


Fig. 7.21 Example of IEC timer functions

Fig. 7.21 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.

7.4.6 IEC counter functions

A counter function implements counting processes in the user program. With a CPU 300, an IEC counter function is a system function block (SFB) in the operating system. A detailed description of the IEC counter functions is provided in Chapter 12.6 "IEC counter functions" on page 470.

For programming, drag the corresponding symbol (CTUD, CTU or CTD) 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. 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 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*.

One can also directly access the output parameters using the instance data, for example with *"DB_name".QD* for a single instance.

Fig. 7.22 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.

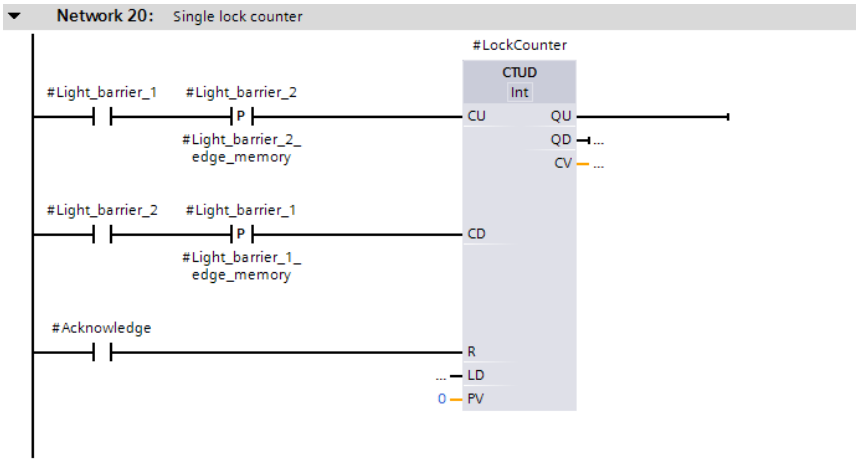


Fig. 7.22 Example of IEC counter functions

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. 7.23 provides an overview of the "basic" functions implemented with EN/ENO boxes.

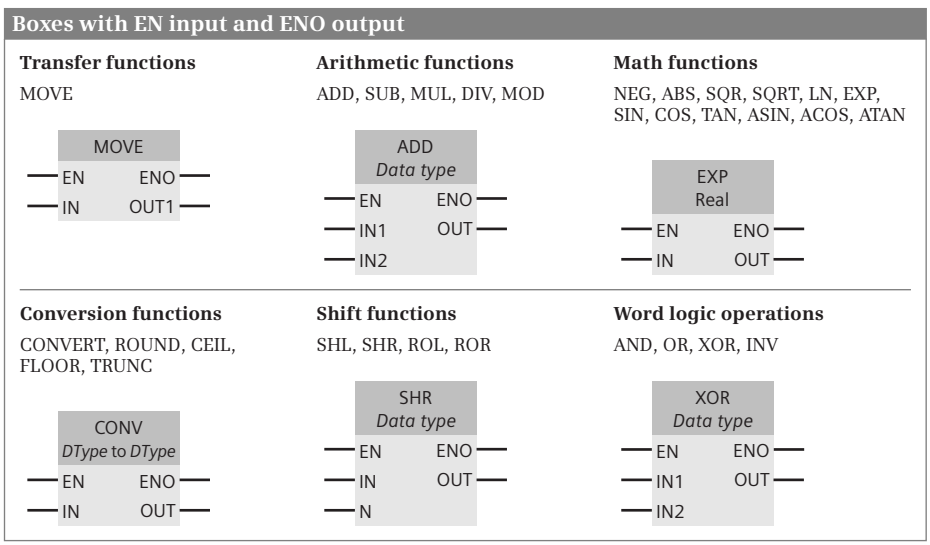


Fig. 7.23 Overview of boxes with enable input EN and enable output ENO

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.

An EN/ENO box can be positioned on its own in a network, with or without connection of the EN input or the ENO output. The ENO output can be connected to the EN input of the following box. By means of a contact at the beginning of this "main current path", it is possible to switch processing of the complete current path on and off. The signal state of the ENO output of the last box indicates by means of a "1" that the complete sequence has been processed without errors.

The ENO output of a box can be connected in parallel with the ENO output of a different box if the boxes are present in separate current paths which commence on the left-hand power rail. "Current" then flows in the subsequent path if one of the two boxes has completed the processing without errors.

If an EN/ENO box is positioned in a T branch, its ENO output can no longer be returned to the path at which the T branch commences.

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 7.6.2 "EN/ENO mechanism with LAD" on page 276. The block calls in the ladder logic, which are also shown as EN/ENO boxes, are described in Chapter 14.4 "Calling of code blocks" on page 547.

7.5.1 Transfer function, MOVE

The transfer function MOVE transfers the value of one tag to another.

For programming, drag the symbol of the MOVE function with the mouse from the program elements catalog under *Basic instructions* > *Move operations* to the working area.

A detailed description of the transfer function is provided in Chapter 13.2 "Transfer functions" on page 476.

In Fig. 7.24, the *#Messages* tag is transferred from the data block *"Data.LAD"* to the *"Message_bits"* tag in the bit memory address area.

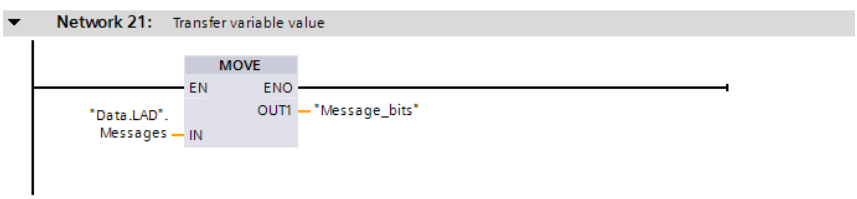


Fig. 7.24 Example of a transfer function in the ladder logic

7.5.2 Arithmetic functions

An arithmetic function for numerical values implements the basic arithmetical operations with the data formats INT, DINT, and REAL in the user program. A detailed description of these arithmetic functions is provided in Chapter 13.4 “Arithmetic functions” on page 491.

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 (ADD, SUB, MUL, DIV, or MOD) and the data type (INT, DINT, or REAL) 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. 7.25, the upper limit of a measured value is monitored. A hysteresis is introduced to ensure that the *#Measurement_too_high* message does not “pulsate” when the measured value changes rapidly around the upper limit. The message *#Measurement_too_high* is only canceled when the measured value has dropped again below the upper limit by the magnitude of the hysteresis.

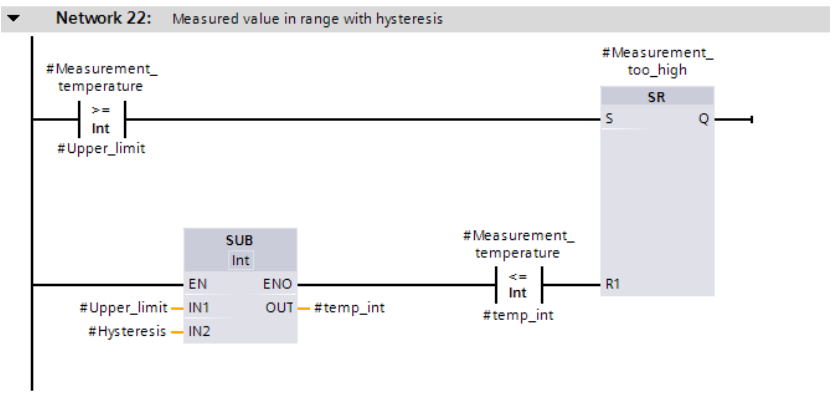


Fig. 7.25 Example of an arithmetic function in the ladder logic

7.5.3 Math functions

The mathematical functions comprise, for example, trigonometric functions, exponential functions, and logarithmic functions with tags in data format REAL. A detailed description of these math functions is provided in Chapter 13.5 “Math functions” on page 496.

For programming, drag one of the mathematical functions (NEG, ABS, SQR, SQRT, LN, EXP, SIN, COS, TAN, ASIN, ACOS, or ATAN) with the mouse from the program elements catalog under *Basic instructions* > *Math functions* to the working area. You can set the function (NEG, ABS, SQR, SQRT, LN, EXP, SIN, COS, TAN, ASIN, ACOS, or

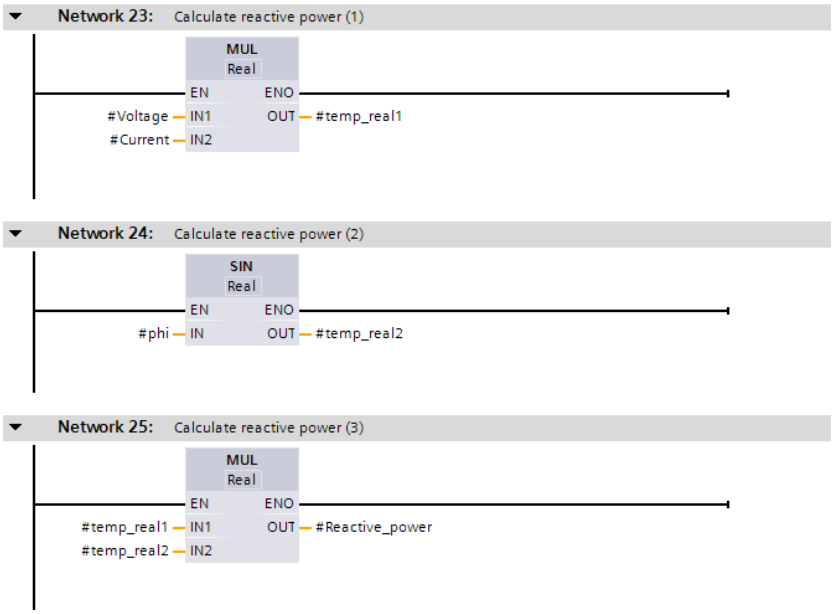


Fig. 7.26 Example of math functions in the ladder logic

ATAN) using drop-down lists which you can open using the small yellow triangle when the box is selected. The data type is permanently set to REAL.

Fig. 7.26 shows the calculation of the reactive power according to the equation *#Reactive_power* = *#Voltage* × *#Current* × sin(*#phi*).

7.5.4 Conversion functions

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 500.

Table 7.1 shows the data type conversions possible with LAD.

For programming, drag one of the conversion functions (CONVERT, ROUND, CEIL, FLOOR, or TRUNC) 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 S_CONV for data type conversion of character strings can be found in the program elements catalog under *Extended instructions* > *String + Char*.

Table 7.1 Data type conversion with LAD

from \ to	BOOL	BYTE	WORD	DWORD	INT	DINT	REAL	TIME	S5TIME	DT	TOD	DATE	CHAR	STRING	BCD16	BCD32
BOOL																
BYTE			IX	IX									IO			
WORD				IX	IO				IO			IO				
DWORD						IO		IO			IO					
INT			IO			C								S	C	
DINT				IO			C	IO						S		C
REAL						R								S		
TIME				IO		IO			T							
S5TIME			IO					T								
DT					T1)						T	T				
TOD				IO												
DATE			IO													
CHAR		IO														
STRING					S	S	S									
BCD16					C											
BCD32						C										

Data type conversion is possible:

- **IX** Implicitly and independent of attribute *IEC check*
- **IO** Implicitly with deactivated attribute *IEC check*
- **C** Explicitly with CONV
- **R** Explicitly with ROUND, CEIL, FLOOR, and TRUNC
- **T** Explicitly with T_CONV, 1) Conversion to day of week
- **S** Explicitly with S_CONV

Additionally: ATH, HTA, SCALE, UNSCALE

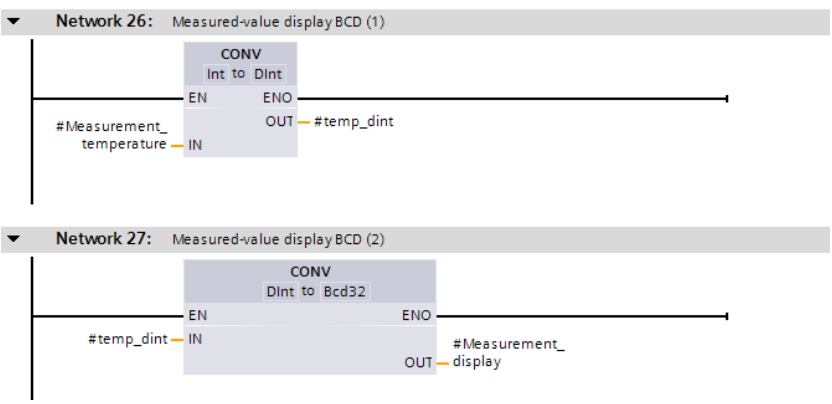


Fig. 7.27 Example of the conversion functions in the ladder logic

Fig. 7.27 shows an example of the conversion functions. A measured value present in data format INT is first expanded to the data format DINT and then converted into the BCD format.

7.5.5 Shift functions

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 514.

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 (SHL, SHR, ROL, and ROR) 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. 7.28, the respective three decades of two numbers present in BCD 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.

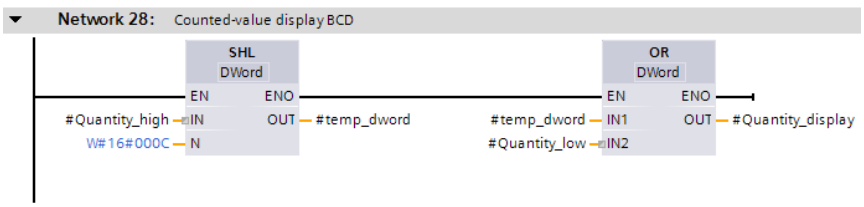


Fig. 7.28 Example of the shift functions in the ladder logic

7.5.6 Word logic operations

The word logic operations link each bit of two tags according to an AND, OR, or exclusive OR function. A detailed description of the word logic operations is provided in Chapter 13.8.1 “Word logic operations” on page 519.

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, 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. 7.29 shows how you can program 32 edge evaluations simultaneously for rising and falling edges. The message bits are collected in a doubleword *Messages*, which is present in data block "*Data.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 message bit has changed. In order to obtain the positive signal edges, the changes are linked to the messages by an AND logic operation. The bit is set for a rising signal edge wherever the message 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 message bits – the message bits with signal state "0" are now "1" – you obtain the pulse flags for a falling edge. At the end it is only necessary for the edge trigger flags to track the messages (last network in Fig. 7.29).

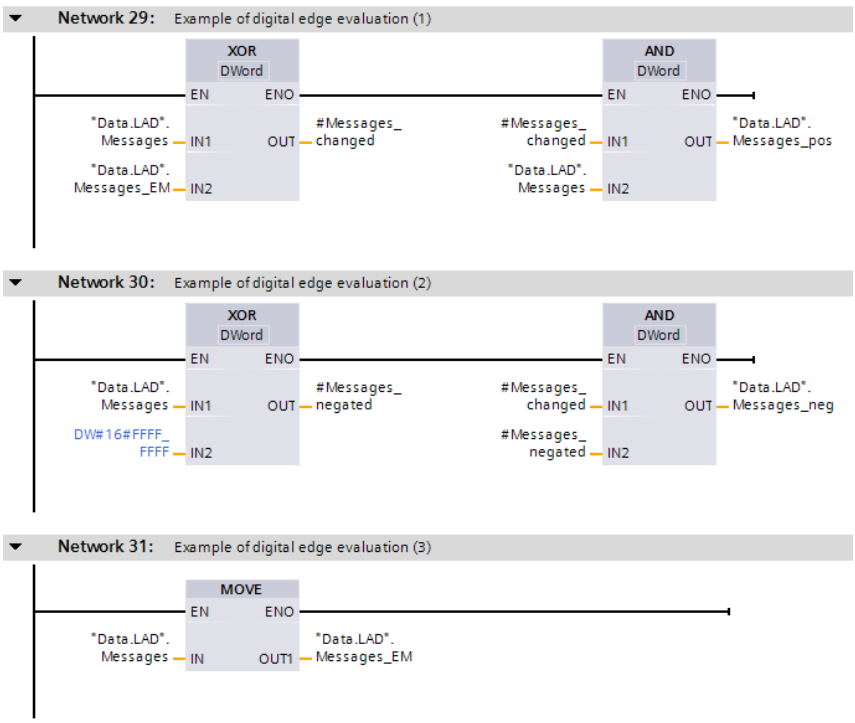


Fig. 7.29 Example of word logic operations in the ladder logic

7.6 Controlling the program flow with LAD

You can influence processing of the user program by means of the program flow control functions. The available functions are shown in Fig. 7.30.

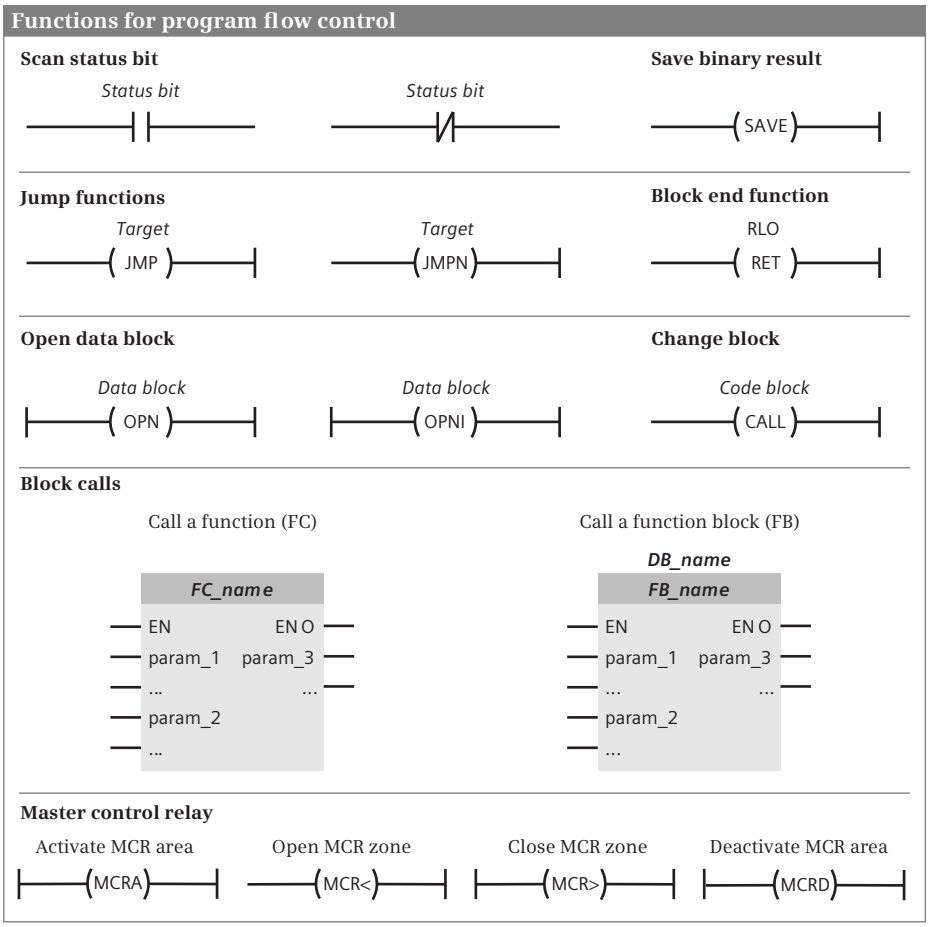


Fig. 7.30 Overview of functions for program flow control in the ladder logic

7.6.1 Working with status bits in the ladder logic

Scanning status bits

Status bits provide information on the result of an arithmetic function and on any errors, for example exceeding a numerical range. Chapter 14.1.5 “Evaluating the status bits” on page 538 describes how you can use contacts to scan the signal state of the status bits.

For programming, drag the NO or NC contact labeled *Status* with the mouse from the program elements catalog under *Basic instructions* > *Additional instructions* to the working area. You can set the status bit to be scanned (OV, OS, UO, BR, ==0, >=0, <=0, >0, <0, and <>0) via a drop-down list which you can open using the small yellow triangle when the contact is selected.

Example: In Fig. 7.31, 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. If the floating-point number is invalid, the comparison is incorrect in all cases and the status bit OV (overflow) is set. Thus if the comparison is incorrect and the overflow bit is set, the intermediate memory *#Floating_point_number_invalid* is set and the JMP (jump) to the *Error* label is carried out.

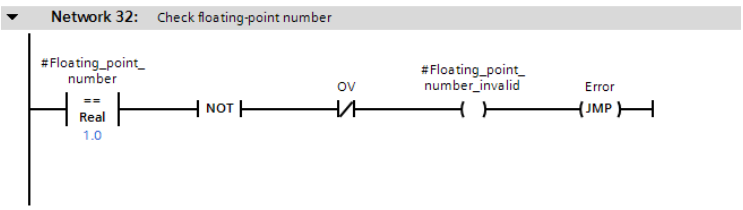


Fig. 7.31 Example of scanning a status bit with LAD

Save binary result

With the SAVE coil you can save the result of logic operation RLO in the binary result BR. A detailed function description of the SAVE coil is provided in Chapter 14.1.4 “Controlling the binary result” on page 535.

For programming, drag the SAVE coil with the mouse from the program elements catalog under *Basic instructions* > *Additional instructions* to the working area and, if applicable, set the SAVE function via a drop-down list which you can open using the small yellow triangle when the contact is selected.

The SAVE coil requires a preceding logic operation and is present alone in a current path. A T branch must not be programmed in the network with a SAVE coil. Note that the SAVE coil does not terminate the logic operation, and therefore the logic operation can be continued in the following network.

You can control the ENO output of the block with the SAVE coil if you call the coil in the last network of the block. In the example in Fig. 7.34 on page 278, the error messages are collected in the last network of the block: *#Floating_point_number_invalid* (error with “1”) and *#Adder_error* (error with “0”). The logic operation must not be fulfilled with an error; in this case signal state “0” is transferred by the SAVE coil to the ENO output.

7.6.2 EN/ENO mechanism with LAD

With LAD, all block calls and functions (instructions) for which an error can occur have an enable input EN and an enable output ENO.

The EN input and the ENO output are not block parameters and are not declared. They are statement sequences which the program editor generates in the program before and after a block or function call. They are not visible to the user. The EN input and the ENO output are both of data type BOOL.

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.

Controlling a processing sequence

In the example in Fig. 7.32, 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.

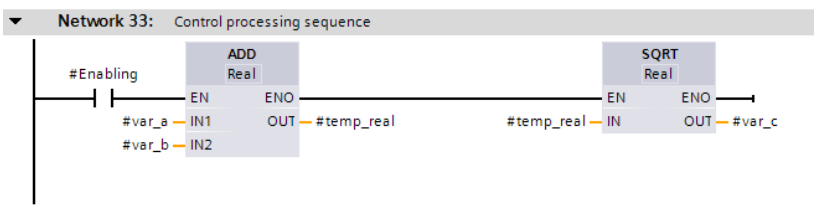


Fig. 7.32 Example of series connection of ENO and EN with LAD

Enable input EN

You can control the calling of a block using the enable input EN. If EN has signal state "1" or is connected to the left-hand power rail, the called block is processed. If EN has signal state "0", the called block is not processed. A jump is then made beyond the block call to the next following instruction (function).

Enable output ENO

You can scan the error status of the block using the enable output ENO. If ENO has signal state "1", processing has been carried out correctly. With signal state "0", the ENO output signals that the block was not called (EN was "0") or that an error is present in the block (Fig. 7.33).

The ENO output has the signal state which the binary result BR had in the block. With self-created blocks, you can control the assignment of the ENO output via the binary result in order, for example, to signal faulty processing in the block.

Fig. 7.33 Schematic diagram for setting of enable output ENO

<table>
<tr><td colspan="5">Is EN connected?</td></tr>
<tr><td colspan="3">YES</td><td colspan="2">NO</td></tr>
<tr><td colspan="2">Is EN = “1”?</td><td rowspan="2"></td><td colspan="2" rowspan="2">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><td colspan="2" rowspan="2"></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>ENO output is set to “0”</td><td>ENO output is set to “1”</td><td>ENO output is set to “0”</td><td>ENO output is set to “0”</td><td>ENO output is set to “1”</td></tr>
</table>

Example: In Fig. 7.34 on page 278, in the first network the *#Adder_error* tag is set to signal state “0” in the event of an error in the *“Adder.LAD”* block, and a jump is made to the *Error* label. The jump label *Error* is present in the last network of the block. The binary result BR, and thus also the ENO output of the current block, are set to signal state “0” here by means of the SAVE coil.

7.6.3 Jump functions

To program a jump function, 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 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.

The jump functions cannot be programmed in association with a T branch. Only one jump function is permissible per network. If you use the Master Control Relay (MCR), the jump destination must be located in the same MCR zone or in the same MCR area as the jump function.

A detailed description of the jump functions is provided in Chapter 14.2 “Jump functions” on page 539.

Fig. 7.31 on page 275 and Fig. 7.34 on page 278 show examples of the jump functions. In the first example, a jump is carried out by means of a JMP coil to the *Error* label upon a result of logic operation “1”. In the second example, a jump is also carried out by means of a JPMN coil to the *Error* label upon a result of logic operation “0”.

7.6.4 Block functions

Block end function, RET coil

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. Above the RET coil, RLO (result of logic operation) indicates that the result of the logic operation present in front of the RET coil (the "current flow") is assigned to the ENO output of the block which has been left.

A detailed description of the RET coil is provided in Chapter 14.3.1 "Block end function RET (LAD and FBD)" on page 545.

The RET coil requires a preceding logic operation and must only terminate a current path on its own.

In the second network in Fig. 7.34, the block with the RET coil is left if the *"Adder"* block does not signal an error.

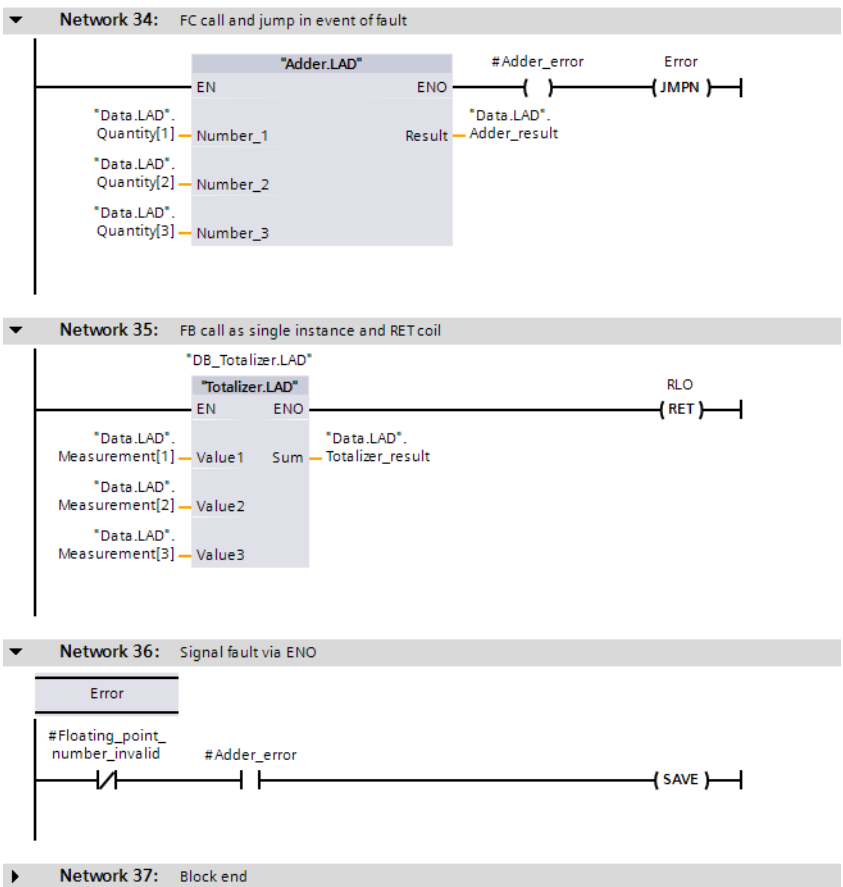


Fig. 7.34 Examples of functions for program flow control in the ladder logic

Open data block; OPN and OPNI coils

To program the OPN or OPNI coil, drag it with the mouse from the program elements catalog under *Basic instructions* > *Program control operations* to the working area. With OPN you open a data block via the DB register, with OPNI via the DI register.

You can subsequently set the function (OPN or OPNI) via a drop-down list which you can open using the small yellow triangle when the coil is selected.

The OPN or OPNI coil is present alone in a network without a preceding logic operation.

A detailed description of the OPN or OPNI coil is provided in Chapter 14.5.1 “Open data block” on page 555.

Opening a data block using block parameters

A block parameter with parameter type BLOCK_DB allows the transfer of a data block (or more precisely: a data block number) to the called block. You call this data block as a global data block in the called block with the OPN coil via the DB register.

Calling of a data block transferred with BLOCK_DB is not possible via the DI register.

Block change, CALL coil

To program the CALL coil, drag it with the mouse from the program elements catalog under *Basic instructions* > *Additional instructions* to the working area. With CALL, you call a code block which must not have any block parameters.

The CALL coil is present alone in a network. The CALL coil can be connected directly to the left-hand power rail – the block call then takes place without conditions, or it can follow a preceding logic operation.

A detailed description of the CALL coil is provided in Chapter 14.4.4 “Change to a block without block parameter” on page 551.

Block call functions

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 14.4 “Calling of code blocks” on page 547.

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. 7.34 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 “0” 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.

7.6.5 Master Control Relay (MCR)

The Master Control Relay controls write operations to the user memory. A detailed description of the MCR functions is provided in Chapter 14.6 “Master control relay” on page 560.

You use the MCRA and MCRD coils to define an MCR area. The two coils are each present alone in a network.

You use the MCR< coil to open an MCR zone. The coil requires a preceding logic operation and terminates a current path. If “current” flows into the coil, the MCR dependency is switched off (“normal” processing); if no “current” flows into the coil, the MCR dependency is switched on.

The MCR> coil closes an MCR zone and is present alone in a network.

To program the corresponding MCR function, drag it with the mouse from the program elements catalog under *Basic instructions* > *Additional instructions* to the working area. You can subsequently set the function (MCR<, MCR>, MCRA, or MCRD) via a drop-down list which you can open using the small yellow triangle when the coil is selected.

Fig. 7.35 shows the networks required to switch the MCR dependency on and off.

If MCR dependency is switched on, the following system blocks influence the operands in the I/O range, in the process image, and in the bit memory address area:

- SET, SETI, and SETP set the parameterized operands to signal state “1”
- RESET, RESETI, and RESETP reset the parameterized operands to signal state “0”.

The system blocks can be found in the program elements catalog under *Basic instructions* > *Additional instructions*. A detailed description of these blocks is provided in Chapter 13.2.7 “Control memory area with MCR dependency” on page 485.

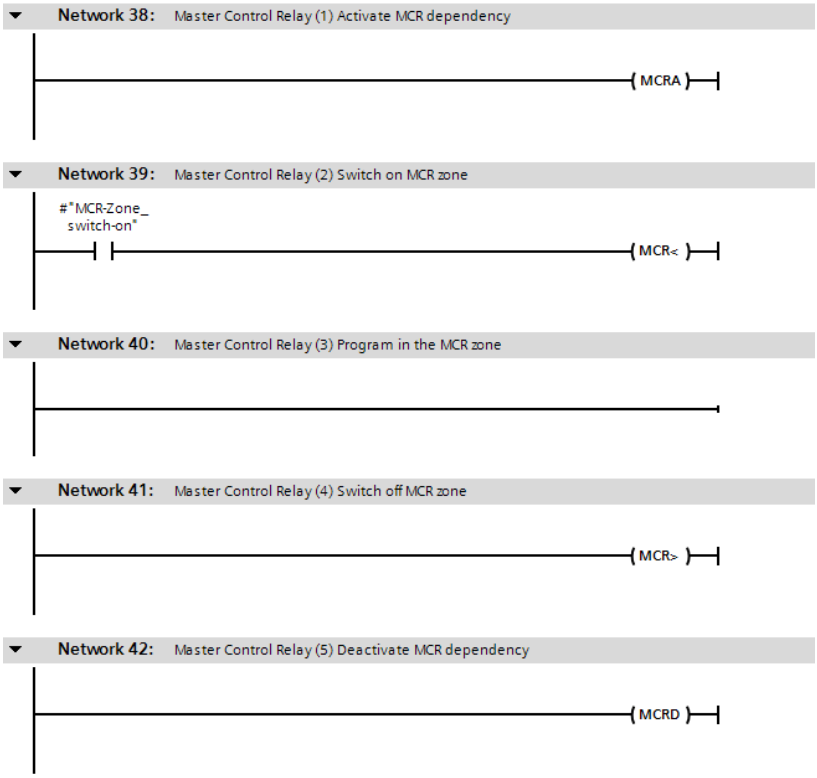


Fig. 7.35 MCR dependency and MCR zone in ladder logic

8 Function block diagram FBD

8.1 Introduction

This chapter describes programming with function block diagram (FBD); it uses examples to show how the program functions are represented in FBD. You can find a description of the individual functions, e.g. comparison functions, in Chapters 12 “Basic functions” on page 427, 13 “Digital functions” on page 475, and 14 “Program flow control” on page 530.

Use of the program and symbol editor, which generally applies to all programming languages, is described in Chapter 6 “Program editor” on page 218.

FBD is used to program the contents of blocks (the user program). What blocks are and how they are created is described in Chapters 5.2.3 “Block types” on page 156 and 6.3 “Programming a code block” on page 223.

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 structure of a block with the FBD program. Located at the beginning of the program is the block header (block title) and the block comment. Heading and comment are optional. These are followed by the first network with its number, heading and comment. Heading and comment are also optional for the networks. The first network shows a logic operation as example with AND and OR boxes, a memory function within the logic operation, and two assignments as termination of the logic operation. The second network shows the processing of EN/ENO boxes, of which two are arranged in series. A block is not terminated by a special network or function; you simply finish the program input.

The program editor constructs an FBD network from left to right: Position the first program element underneath the network comment and insert further program elements at 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.

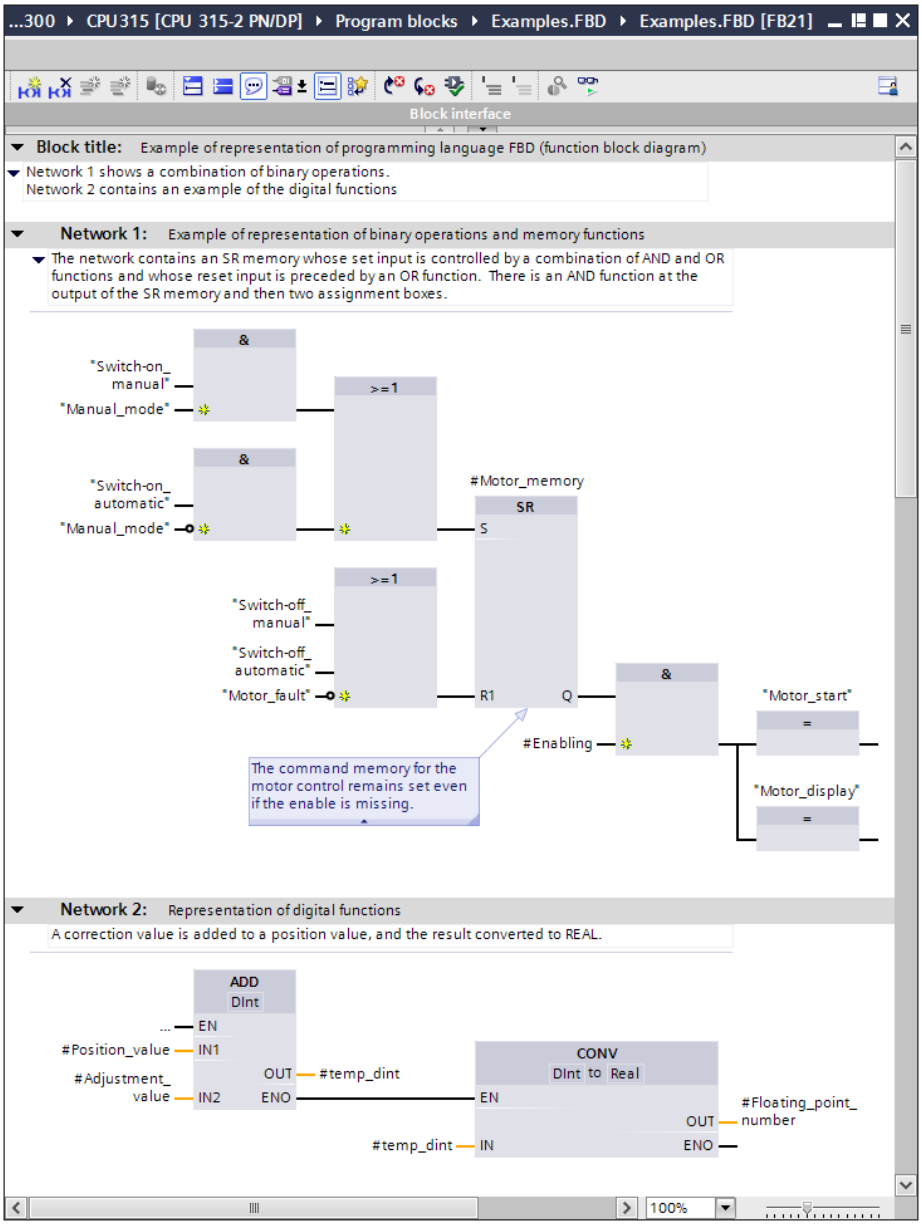


Fig. 8.1 Structure of a block with FBD program

"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.

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.

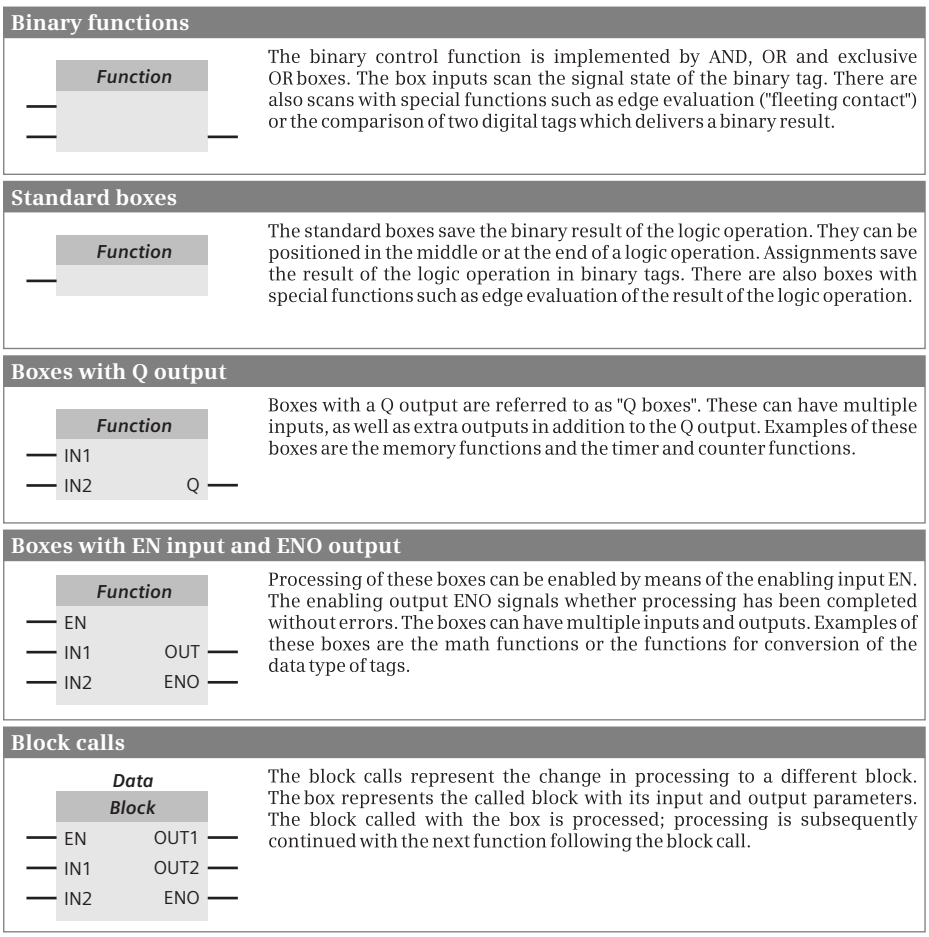


Fig. 8.2 Overview of program elements of the function block diagram

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 for the logic operation can be scanned for signal state “1” or “0”. The binary results of other boxes can also be included, e.g. the evaluation of a signal edge or the comparison of two digital tags (Fig. 8.3).

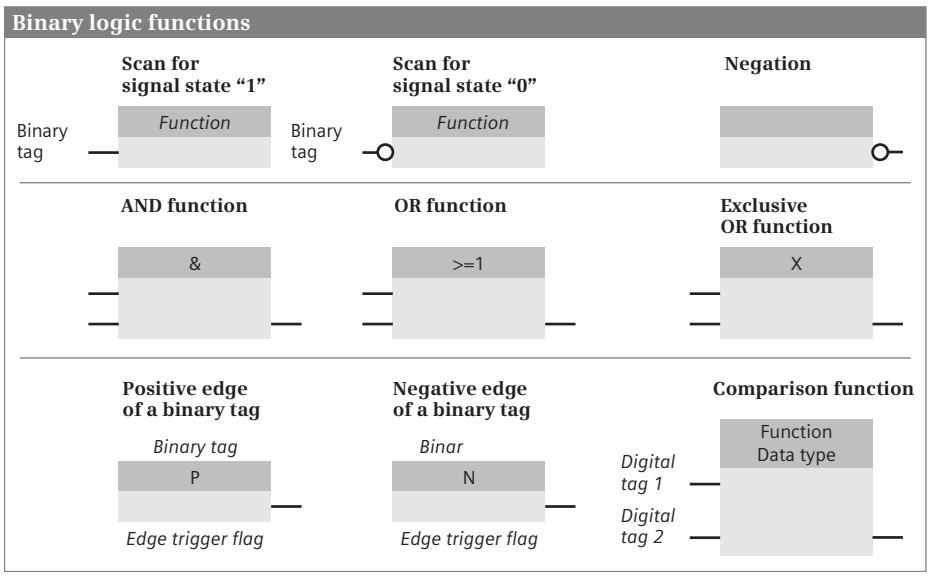


Fig. 8.3 Overview of binary logic operations in the function block diagram

8.2.1 Scanning for signal states “1” and “0”

The binary functions scan the binary tags at the function inputs before they link the signal states together. The scan can be made for signal state “1” or “0”. When scanning for signal state “1”, the function input leads directly to the box. You can recognize the scanning for signal state “0” by means of the negation circle at the input of the function.

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.

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”.

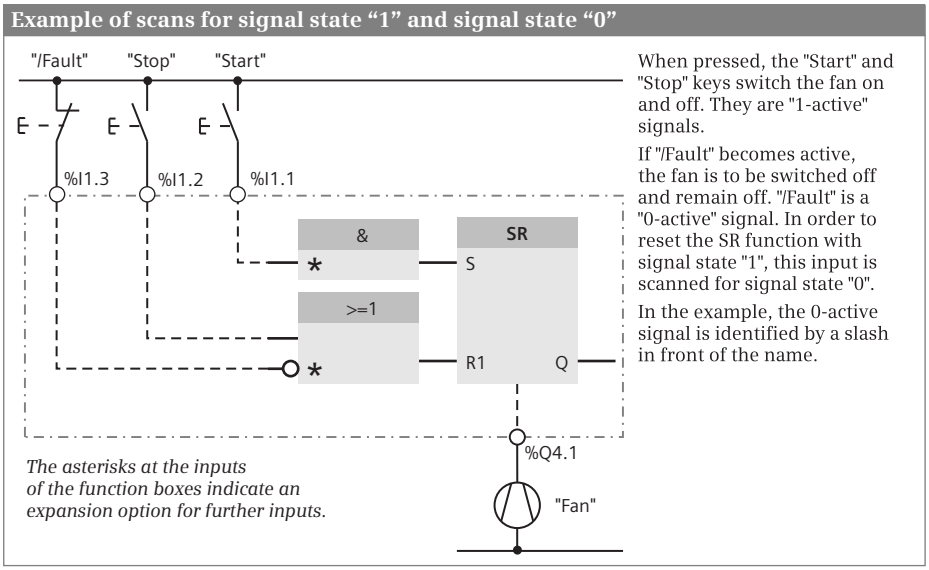


Fig. 8.4 Scanning for signal states “1” and “0”

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 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, 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 or -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, posi-

tion 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

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 431.

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.

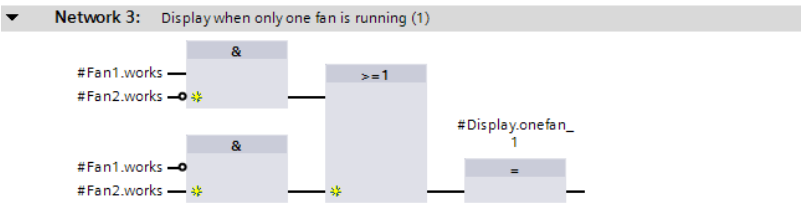


Fig. 8.5 Example of AND-before-OR logic operation

8.2.4 OR function

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 432.

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.

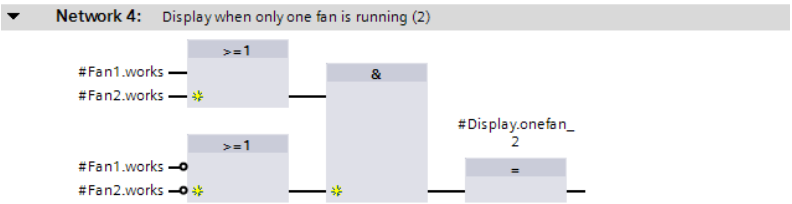


Fig. 8.6 Example of OR-before-AND logic operation

8.2.5 Exclusive OR function

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 432.

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.

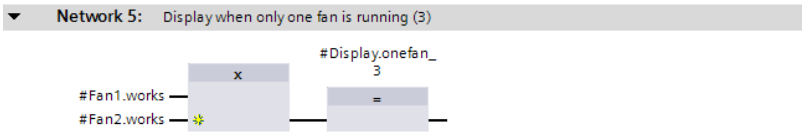


Fig. 8.7 Example of an exclusive OR function

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, 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”).

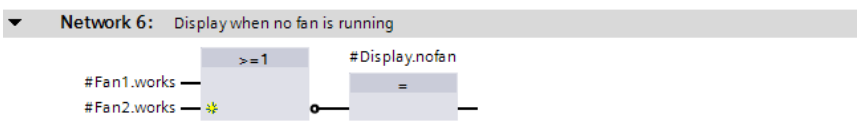


Fig. 8.8 Example of a negated function output

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.

8.2.7 T branch

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 2 Hz*” and controls the *#Display.nofan* tag.

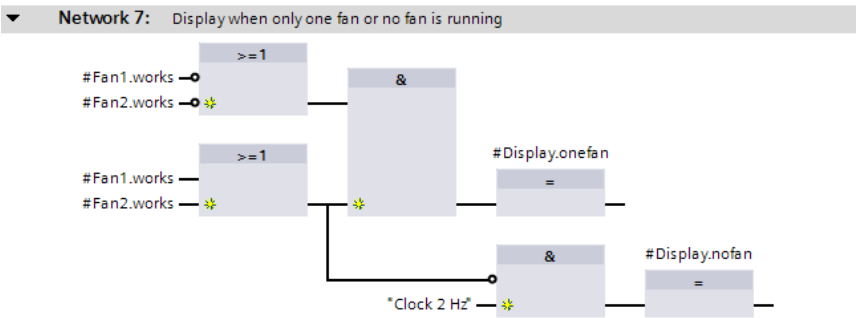


Fig. 8.9 Example of a T branch in the function block diagram

8.2.8 Edge evaluation of binary tags

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 12.2.5 “Edge evaluation” on page 438).

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").

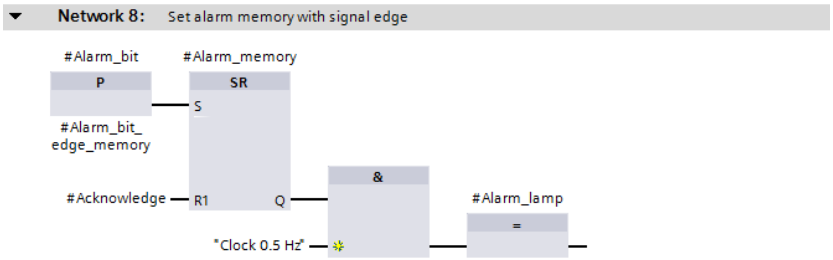


Fig. 8.10 Example of an edge evaluation of a binary tag

8.2.9 Comparison functions

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 487.

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 set the comparison mode and data type (Fig. 8.11).

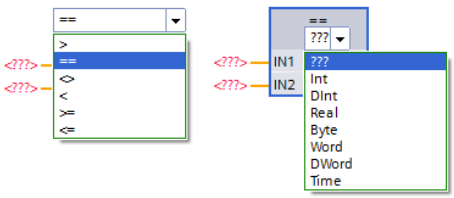


Fig. 8.11 Drop-down lists for setting the comparison mode and data type

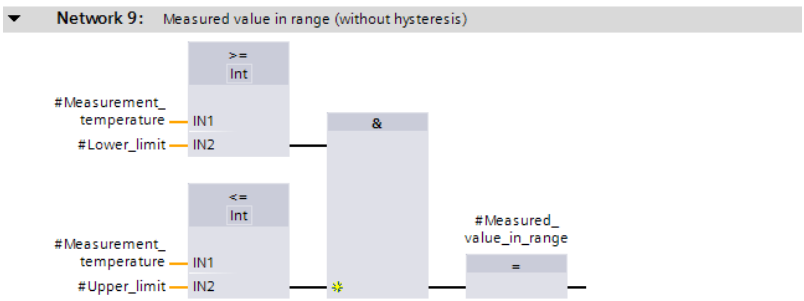


Fig. 8.12 Example of comparison functions

Fig. 8.12 shows two comparison boxes. 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”.

8.3 Programming standard boxes with FBD

Standard boxes control binary tags such as outputs or bit memories. Standard boxes exist for assigning, setting, and resetting a binary tag or for controlling a SIMATIC timer or counter function (Fig. 8.13).

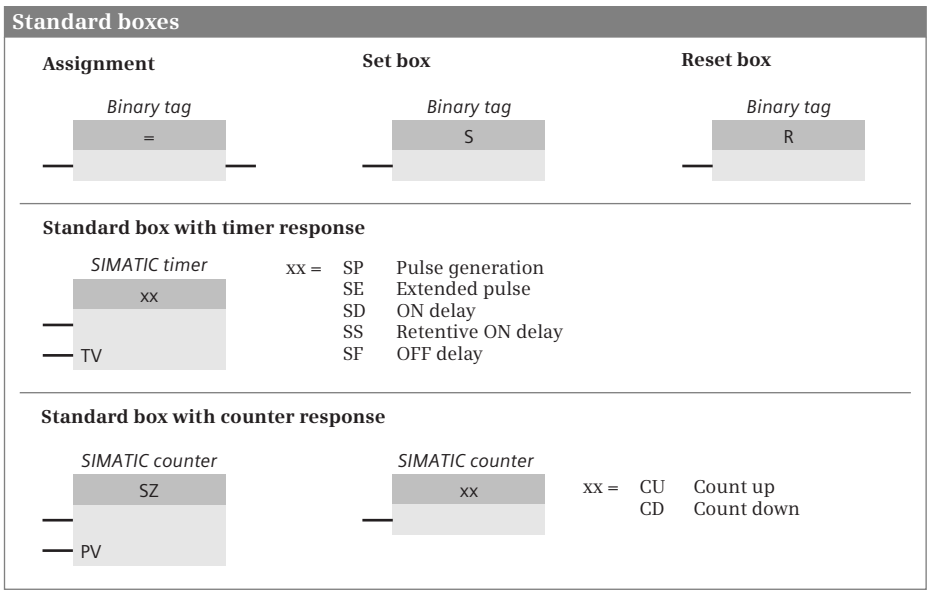


Fig. 8.13 Overview of standard boxes available with FBD

8.3.1 Assign box

The assign box directly assigns the result of logic operation to the tag above the box.

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 assign box can be used within a logic operation, following a T branch, or as the termination of an operation. It can be positioned in series or parallel. The assign box requires a preceding logic operation.

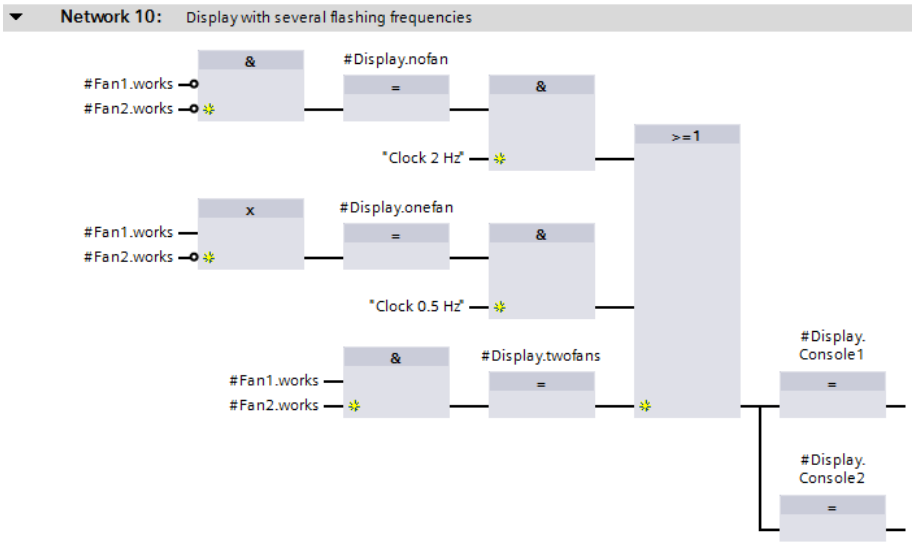


Fig. 8.14 Example of arrangement of the assign box

Fig. 8.14 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.

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 or -counter function.

In Fig. 8.15, *#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. As a result of positioning of the reset coil after the set coil, the memory response is “reset dominant”: If both tags have signal state “1”, *#Fan1.drive* is reset or remains reset. The enable for starting and stopping is not directly connected to the AND functions in this example, but permits the representation of both boxes in one network through the programming prior to a T branch.

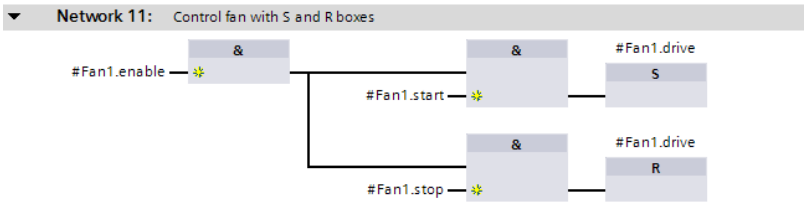


Fig. 8.15 Example of set and reset boxes

8.3.3 Standard boxes with time response

A standard box with time response is a single element of a SIMATIC timer function. The timer function is usually applied as a complete timer box which contains all elements. The standard box with time response corresponds to the S input of the complete timer box. Attention must be paid to the sequence in the program when using the single elements. The time response of these boxes is described in Chapter 12.3 “SIMATIC timer functions” on page 443.

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 and terminates a logic operation. It can be connected parallel to all other boxes. Positioning at the end of a T branch is also possible.

The time tag is positioned above the standard box with a time response. This is an operand from the range of SIMATIC timers (T). The time value is specified in the data format S5TIME at the TV input.

In Fig. 8.16, the timer *“Fan4.delay”* is started as a switch-on delay by the positive edge of *#Fan4.start*. Following expiry of the duration (5 s in the example), the fan is switched on by *#Fan4.drive*. If *#Fan4.start* has the signal state “0” prior to expiry of the duration, the fan is not even switched on.

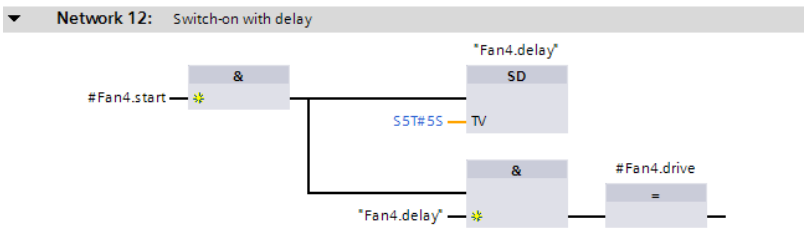


Fig. 8.16 Example of a standard box with time response

8.3.4 Standard boxes with counter response

A standard box with counter response is a single element of a SIMATIC counter function. The counter function is usually applied as a complete counter box which contains all elements. The SC box corresponds to the S input of the complete counter box, the CU box to the CU input, and the CD box to the CD input. Attention must be paid to the sequence in the program when using the single elements. The response of these boxes is described in Chapter 12.5 “SIMATIC counter functions” on page 462.

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 logic operation is terminated by a standard box with counter response. It can be connected parallel to all other boxes. Positioning at the end of a T branch is also possible.

The counter tag is positioned above the standard box with a counter response. This is an operand from the range of SIMATIC counters (C). The counter 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.

In Fig. 8.17, the switch-on processes of *#Fan1.start* are counted with the SIMATIC counter “*Fan1.number*”. The *#Acknowledge* signal resets the counter to 0.

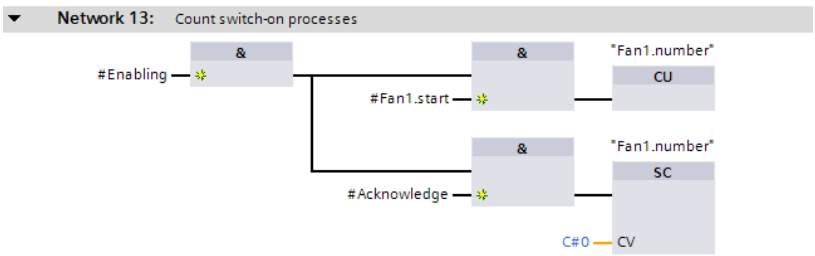


Fig. 8.17 Example of standard boxes with counter response

8.4 Programming Q boxes with FBD

“Q boxes” is the abbreviation for boxes with an output parameter named “Q”. These are the memory boxes SR and RS, the edge evaluations P_TRIG and N_TRIG, and the timer and counter functions (Fig. 8.18).

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.

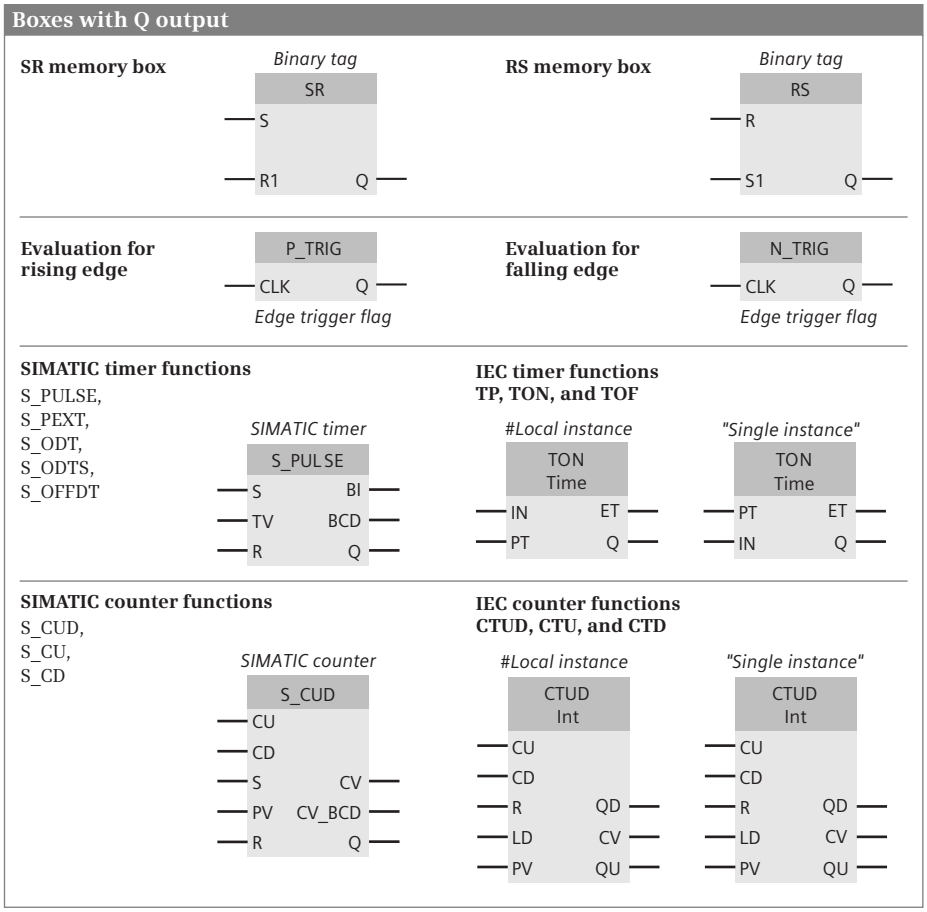


Fig. 8.18 Overview of Q boxes available with FBD

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

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 435.

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. 8.19 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.

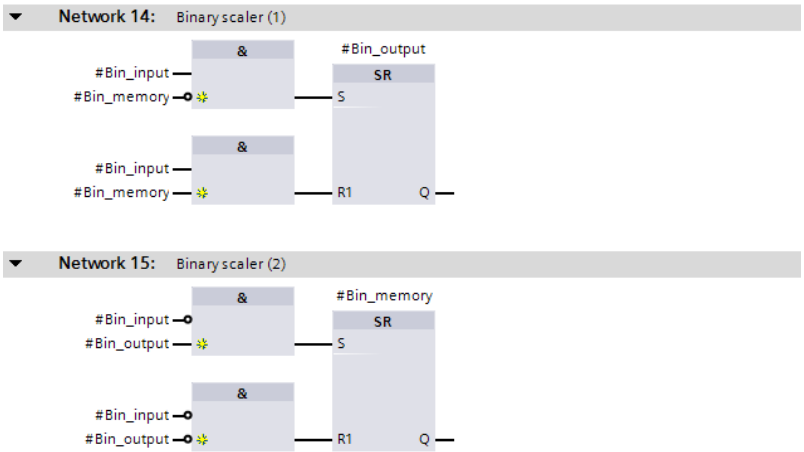


Fig. 8.19 Example of binary scaler

8.4.2 Edge evaluation of result of logic operation

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. 8.20, *#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 message memory are set if the upper limit is exceeded again.

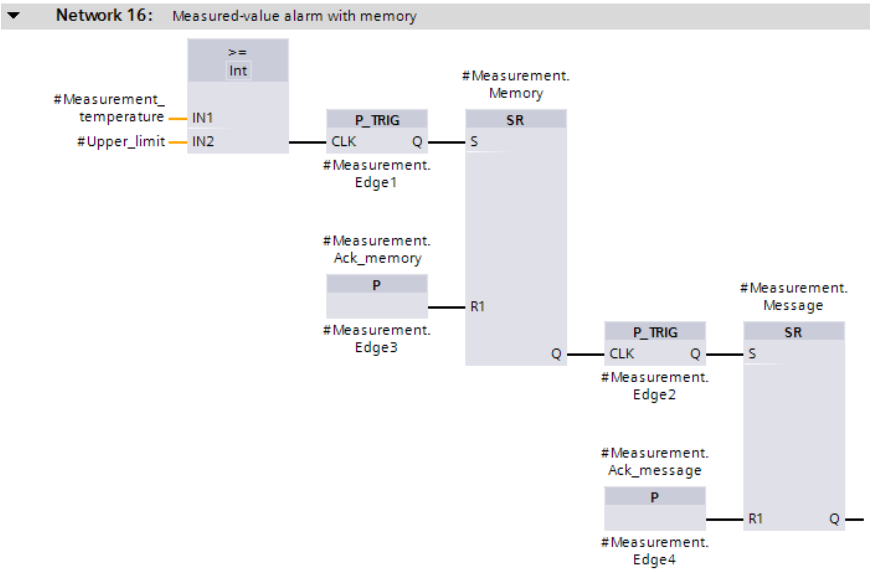


Fig. 8.20 Example of edge evaluation of the result of the logic operation

8.4.3 SIMATIC timer functions

Timer functions are used to implement dynamic processes in the user program. The box of a SIMATIC timer function contains all instructions required for the sequence. A detailed description of the SIMATIC timer functions is provided in Chapter 12.3 “SIMATIC timer functions” on page 443.

For programming, drag the corresponding symbol S_PULSE, S_PEXT, S_ODT, S_ODTS or S_OFFDT 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. 8.21 shows a switch-on and switch-off delay. The timer function “*Fan5.on-delay*” is started by *#Fan5.start*. The Q output has signal state “1” after 3 s, which starts the timer function *“Fan5.Off-delay”*. At the same time, the *#Fan5.drive* tag is started by the Q output of the box. The Q output still has signal state “1” for 5 s after *#Fan5.start* has signal state “0”.

8.4.4 SIMATIC counter functions

Counter functions are used to implement counting tasks in the user program. The box of a SIMATIC counter function contains all instructions required for the se-

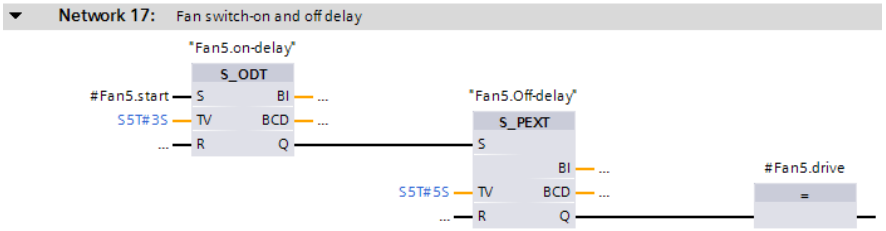


Fig. 8.21 Example of SIMATIC timer functions in the function block diagram

quence. A detailed description of the SIMATIC counter functions is provided in Chapter 12.5 “SIMATIC counter functions” on page 462.

For programming, drag the corresponding symbol (S_CUD, S_CU or S_CD) 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. 8.22 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.

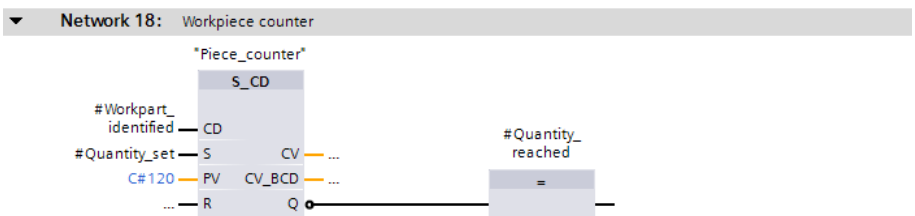


Fig. 8.22 Example of SIMATIC counter functions in the function block diagram

8.4.5 IEC timer functions

Timer functions are used to implement dynamic processes in the user program. With a CPU 300, an IEC timer function is a system function block (SFB) in the operating system. A detailed description of the IEC timer functions is provided in Chapter 12.4 “IEC timer functions” on page 459.

For programming, drag the corresponding symbol (TP, TON or TOF) 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 as local instance. The instance data block generated automatically when selecting as a single instance is saved in the project tree under *Program blocks* > *System blocks* > *Program resources*.

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 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. 8.23 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.

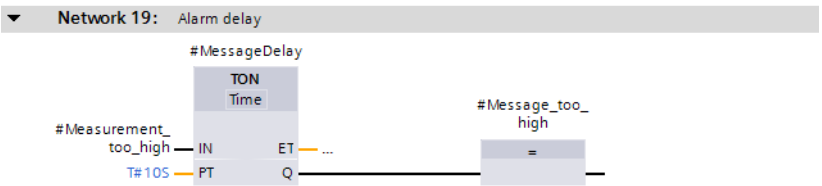


Fig. 8.23 Example of IEC timer functions in the function block diagram

8.4.6 IEC counter functions

A counter function implements counting processes in the user program. With a CPU 300, an IEC counter function is a system function block (SFB) in the operating system. A detailed description of the IEC counter functions is provided in Chapter 12.6 "IEC counter functions" on page 470.

For programming, drag the corresponding symbol (CTUD, CTU or CTD) 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 as local instance. The instance data block generated automatically when selecting as a single instance is saved in the project tree under *Program blocks* > *System blocks* > *Program resources*.

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. 8.24 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.

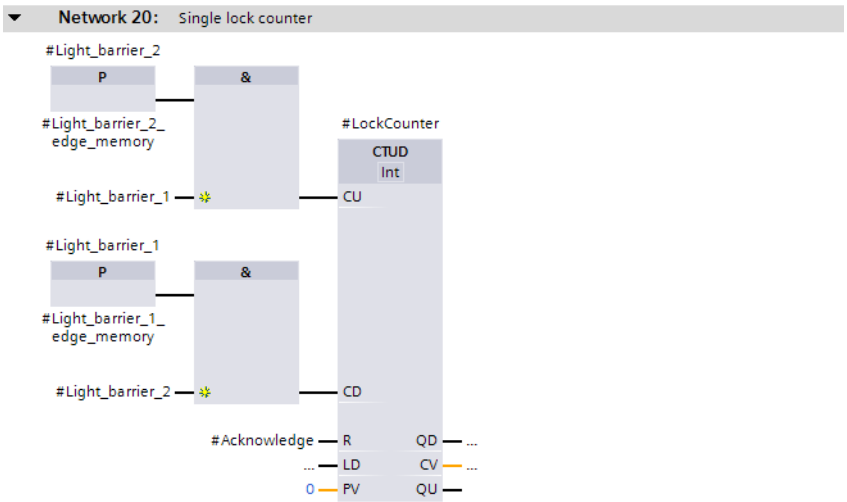


Fig. 8.24 Example of IEC counter functions in the function block diagram

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. 8.25 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.

An EN/ENO box can be positioned on its own in a network, with or without connection of the EN input or the ENO output. The ENO output can be connected to the EN input of the following box or to a binary logic operation.

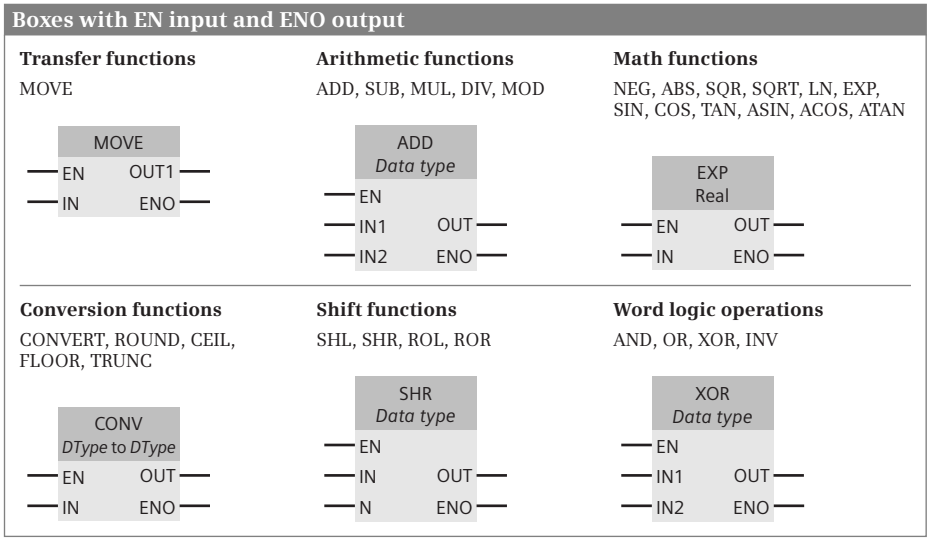


Fig. 8.25 Overview of boxes with enable input EN and enable output ENO

If an EN/ENO box is positioned in a T branch, its ENO output can no longer be returned to the path at which the T branch commences.

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 8.6.2 “EN/ENO mechanism with FBD” on page 309. The block calls in the function block diagram which are also shown as EN/ENO boxes are described in Chapter 14.4 “Calling of code blocks” on page 547.

8.5.1 Transfer function MOVE

The transfer function MOVE transfers the value of one tag to another.

For programming, drag the symbol for the MOVE function with the mouse from the program elements catalog under *Basic instructions* > *Move operations* to the working area.

A detailed description of the transfer function is provided in Chapter 13.2 “Transfer functions” on page 476.

In Fig. 8.26, the *#Messages* tag is transferred from the data block “*Data.FBD*” to the *#Message_bits* tag in the memory area.

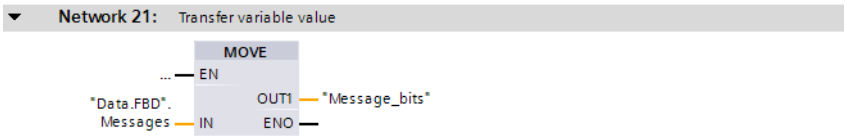


Fig. 8.26 Example of a transfer function in the function block diagram

8.5.2 Arithmetic functions

An arithmetic function for numerical values implements the basic arithmetical operations with the data formats INT, DINT, and REAL in the user program. A detailed description of these arithmetic functions is provided in Chapter 13.4 “Arithmetic functions” on page 491.

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 (ADD, SUB, MUL, DIV, or MOD) and the data type (INT, DINT, or REAL) 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. 8.27, the upper limit of a measured value is monitored. A hysteresis is introduced to ensure that the *#Measurement_too_high* message does not “pulsate” when the measured value changes rapidly around the upper limit. The message *#Measurement_too_high* is only canceled when the measured value has dropped again below the upper limit by the magnitude of the hysteresis.

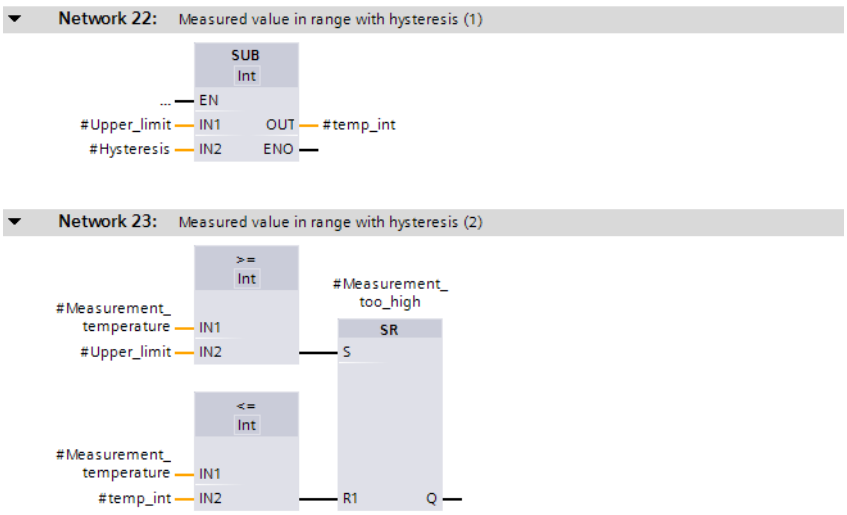


Fig. 8.27 Example of an arithmetic function in the function block diagram

8.5.3 Math functions

The mathematical functions comprise, for example, trigonometric functions, exponential functions, and logarithmic functions with tags in data format REAL. A detailed description of these math functions is provided in Chapter 13.5 “Math functions” on page 496.

For programming, drag one of the mathematical functions (NEG, ABS, SQR, SQRT, LN, EXP, SIN, COS, TAN, ASIN, ACOS, or ATAN) with the mouse from the program elements catalog under *Basic instructions* > *Math functions* to the working area. You can set the function (NEG, ABS, SQR, SQRT, LN, EXP, SIN, COS, TAN, ASIN, ACOS, or ATAN) using drop-down lists which you can open using the small yellow triangle when the box is selected. The data type is permanently set to REAL.

Fig. 8.28 shows the calculation of the reactive power according to the equation *#Reactive_power* = *#Voltage* × *#Current* × sin(*#phi*).

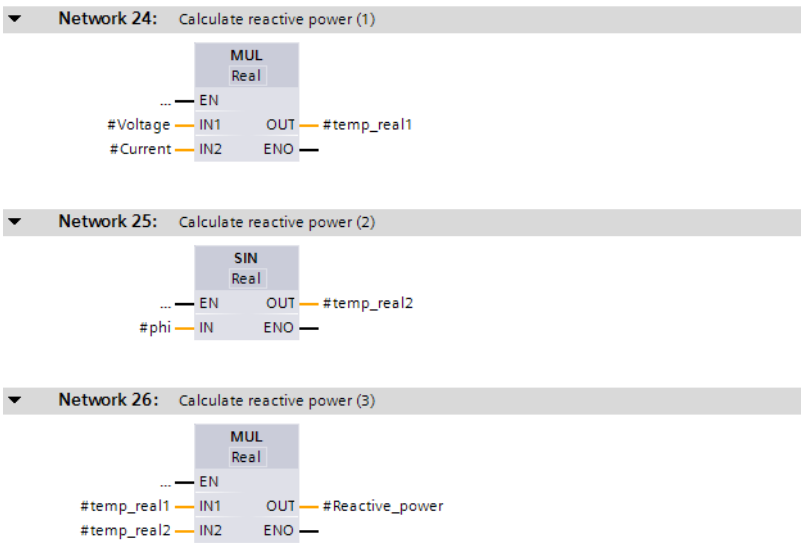


Fig. 8.28 Example of math functions in the function block diagram

8.5.4 Conversion functions

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 500.

Table 8.1 shows the data type conversions possible with FBD.

For programming, drag one of the conversion functions (CONVERT, ROUND, CEIL, FLOOR, or TRUNC) with the mouse from the program elements catalog under *Basic instructions* > *Conversion operations* to the working area. You can set the function

Table 8.1 Data type conversion with FBD

from \ to	BOOL	BYTE	WORD	DWORD	INT	DINT	REAL	TIME	S5TIME	DT	TOD	DATE	CHAR	STRING	BCD16	BCD32
BOOL																
BYTE			IX	IX									IO			
WORD				IX	IO				IO			IO				
DWORD						IO		IO			IO					
INT			IO			C								S	C	
DINT				IO			C	IO						S		C
REAL						R								S		
TIME				IO		IO			T							
S5TIME			IO					T								
DT					T1)						T	T				
TOD				IO												
DATE			IO													
CHAR		IO														
STRING					S	S	S									
BCD16					C											
BCD32						C										

Data type conversion is possible:
- **IX** Implicitly and independent of attribute *IEC check*
- **IO** Implicitly with deactivated attribute *IEC check*
- **C** Explicitly with CONV
- **R** Explicitly with ROUND, CEIL, FLOOR, and TRUNC
- **T** Explicitly with T_CONV, 1) Conversion to day of week
- **S** Explicitly with S_CONV

Additionally: ATH, HTA, SCALE, UNSCALE

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 S_CONV for data type conversion of character strings can be found in the program elements catalog under *Extended instructions > String + Char*.

Fig. 8.29 shows an example of the conversion functions. A measured value present in data format INT is first expanded to the data format DINT and then converted into the BCD format.

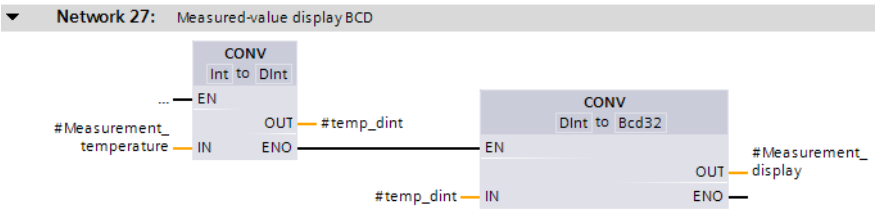


Fig. 8.29 Example of conversion functions in the function block diagram

8.5.5 Shift functions

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 514.

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 (SHL, SHR, ROL, and ROR) 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. 8.30, the respective three decades of two numbers present in BCD 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. Also on this box, a small gray square indicates the implicitly converted data type from WORD to DWORD.

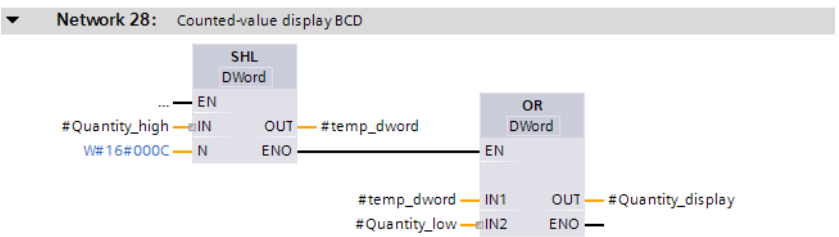


Fig. 8.30 Example of shift functions in the function block diagram

8.5.6 Word logic operations

The word logic operations link the individual bits of two tags according to an AND, OR, or exclusive-OR function. A detailed description of the word logic operations is provided in Chapter 13.8.1 “Word logic operations” on page 519.

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. 8.31 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 message bit has changed. In order to obtain the positive signal edges, the changes are linked to the

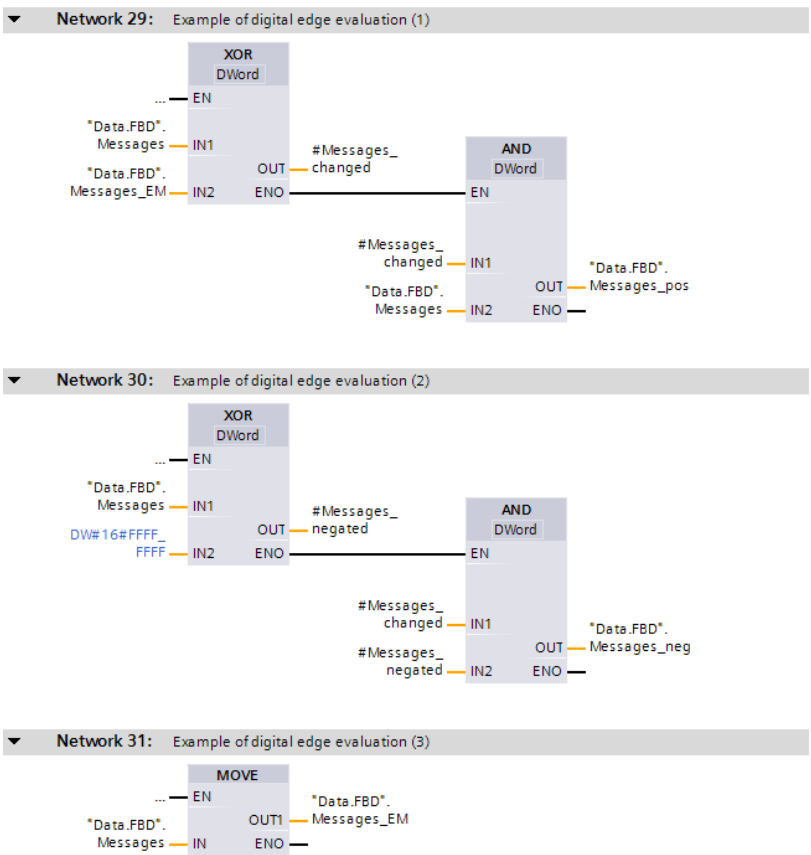


Fig. 8.31 Example of word logic operations in the function block diagram

messages by an AND logic operation. The bit is set for a rising signal edge wherever the message 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 message bits – the message bits with signal state “0” are now “1” – you obtain the pulse flags for a falling edge.

At the end it is only necessary for the edge trigger flags to track the messages.

8.6 Controlling the program flow with FBD

You can influence processing of the user program by means of the program flow control functions. The available functions are shown in Fig. 8.32.

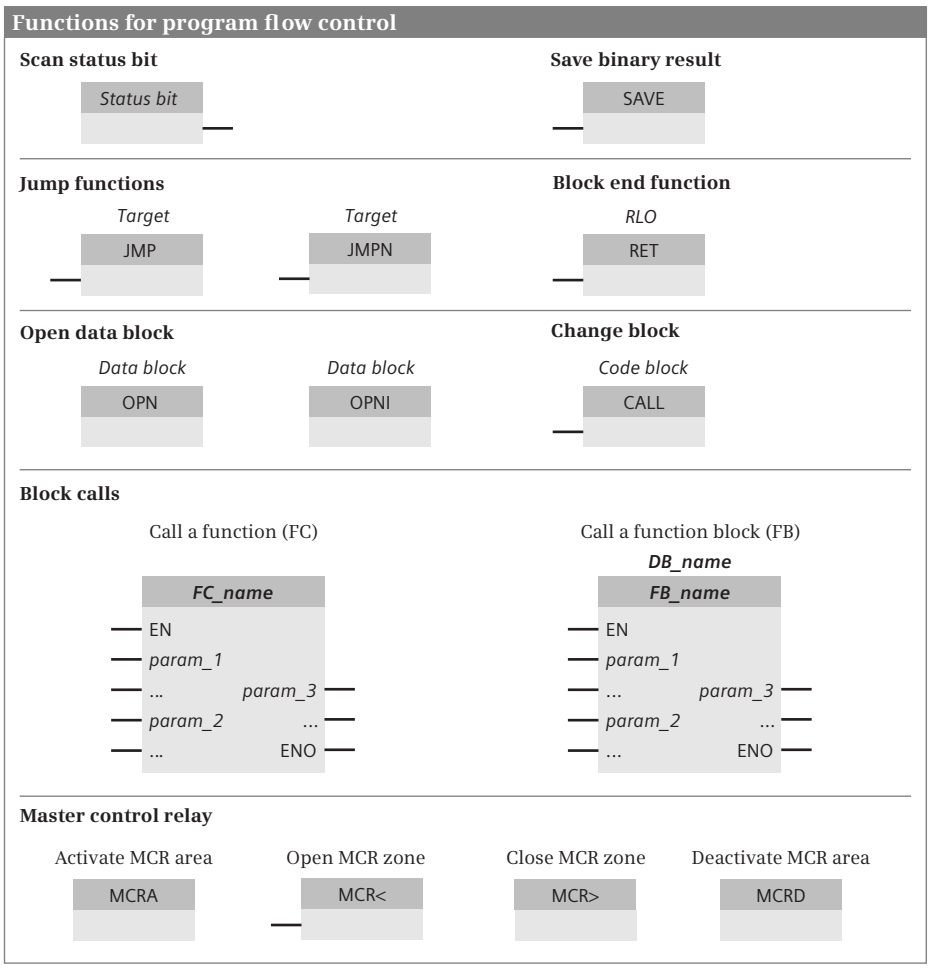


Fig. 8.32 Overview of functions for program flow control in the function block diagram

8.6.1 Working with status bits in the function block diagram

Scanning status bits

Status bits provide information on the result of an arithmetic function and on any errors, for example exceeding a numerical range. Chapter 14.1.5 “Evaluating the status bits” on page 538 describes how you can use scan boxes to scan the signal state of the status bits.

For programming, drag the symbol labeled *Status* with the mouse from the program elements catalog under *Basic instructions* > *Additional instructions* to the working area. You can set the status bit to be scanned (OV, OS, UO, BR, ==0, >=0, <=0, >0, <0, and <>0) via a drop-down list which you can open using the small yellow triangle when the box is selected.

Example: In Fig. 8.33, 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. If the floating-point number is invalid, the comparison is incorrect in all cases and the status bit OV (overflow) is set. Thus if the comparison is incorrect and the overflow bit is set, the intermediate memory *#Floating_point_number_invalid* is set and the JMP (jump) to the *Error* label is carried out.

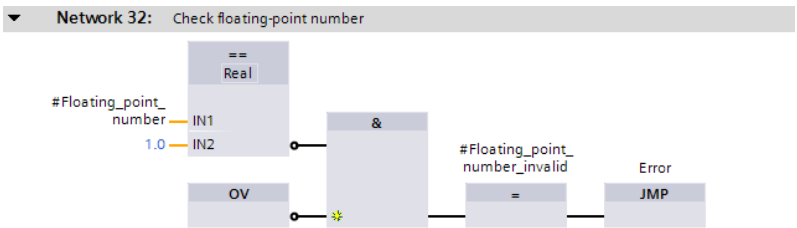


Fig. 8.33 Example of scanning a status bit

Save binary result

With the SAVE box you can save the result of logic operation RLO in the binary result BR. A detailed function description of the SAVE box is provided in Chapter 14.1.4 “Controlling the binary result” on page 535.

For programming, drag the SAVE box with the mouse from the program elements catalog under *Basic instructions* > *Additional instructions* to the working area and, if applicable, set the SAVE function via a drop-down list which you can open using the small yellow triangle when the box is selected.

The SAVE box requires a preceding logic operation and is present alone in a logic operation. A T branch must not be programmed in the network with a SAVE box. Note that the SAVE box does not terminate the logic operation, and therefore the logic operation can be continued in the following network.

You can control the ENO output of the block with the SAVE box if you call the box in the last network of the block. In the example in Fig. 8.36 on page 311, the error messages are collected in the last network of the block: *#Floating_point_number_invalid* (error with "1") and *#Adder_error* (error with "0"). The logic operation must not be fulfilled with an error; in this case signal state "0" is transferred by the SAVE box to the ENO output.

8.6.2 EN/ENO mechanism with FBD

With FBD, all block calls and functions (instructions) for which an error can occur have an enable input EN and an enable output ENO.

The EN input and the ENO output are not block parameters and are not declared. They are statement sequences which the program editor generates in the program before and after a block or function call. They are not visible to the user. The EN input and the ENO output are both of data type BOOL.

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.

Controlling a processing sequence

In the example in Fig. 8.34, 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.

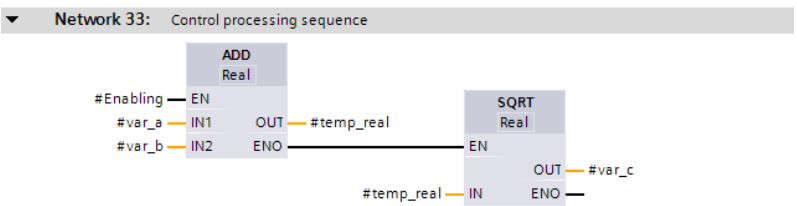


Fig. 8.34 Example of series connection of ENO and EN with FBD

Enable input EN

You can control the calling of a block using the enable input EN. If EN has signal state "1" or is not connected, the called block is processed. If EN has signal state "0", the called block is not processed. A jump is then made beyond the block call to the next following instruction (function).

Enable output ENO

You can scan the error status of the block using the enable output ENO. If ENO has signal state “1”, processing has been carried out correctly. With signal state “0”, the ENO output signals that the block was not called (EN was “0”) or that an error is present in the block (Fig. 8.35).

Fig. 8.35 Schematic diagram for setting of enable output ENO

<table>
<tr><td colspan="5">Is EN connected?</td></tr>
<tr><td colspan="3">YES</td><td colspan="2">NO</td></tr>
<tr><td colspan="3">Is EN = “1”?</td><td colspan="2" rowspan="2">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="2">Block/function not being processed</td><td colspan="2" rowspan="2"></td></tr>
<tr><td colspan="2">Has an error occurred?</td></tr>
<tr><td>YES</td><td>NO</td><td></td><td>YES</td><td>NO</td></tr>
<tr><td>ENO output is set to “0”</td><td>ENO output is set to “1”</td><td>ENO output is set to “0”</td><td>ENO output is set to “0”</td><td>ENO output is set to “1”</td></tr>
</table>

The ENO output has the signal state which the binary result BR had in the block. With self-created blocks, you can control the assignment of the ENO output via the binary result in order, for example, to signal faulty processing in the block.

Example: In Fig. 8.36 on page 311, in the first network the *#Adder_error* tag is set to signal state “0” in the event of an error in the *“Adder.FBD”* block, and a jump is made to the *Error* label.

The jump label *Error* is present in the last network of the block. The binary result BR, and thus also the ENO output of the current block, are set to signal state “0” here by means of the SAVE box.

8.6.3 Jump functions

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.

The jump functions cannot be programmed in association with a T branch. Only one jump instruction is permissible per network. If you use the Master Control Relay (MCR), the jump destination must be located in the same MCR zone or in the same MCR area as the jump function.

A detailed description of the jump functions is provided in Chapter 14.2 “Jump functions” on page 539.

Fig. 8.33 on page 308 and Fig. 8.36 show examples of the jump functions. In the first example, a jump is carried out by means of a JMP coil to the *Error* label upon a result of logic operation “1”. In the second example, a jump is also carried out by means of a JPMN coil to the *Error* label upon a result of logic operation “0”.

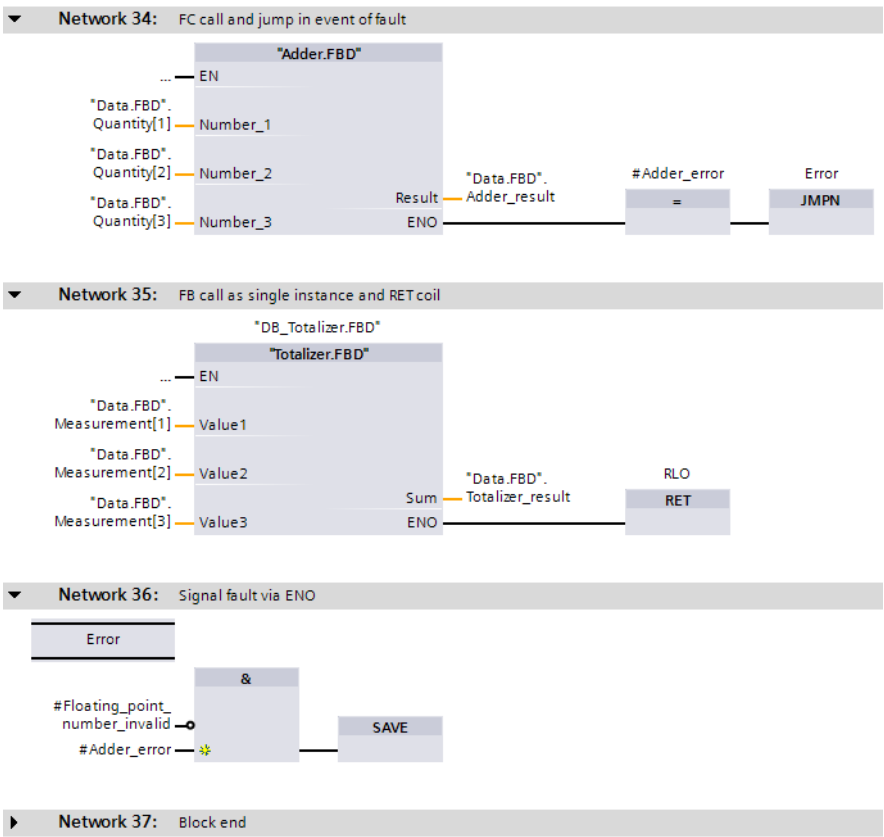


Fig. 8.36 Examples of functions for program flow control in the function block diagram

8.6.4 Block functions

Block end function, RET box

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. Above the RET box, RLO (result of logic operation) indicates that the result of the logic operation present in front of the RET box is assigned to the ENO output of the block which has been left.

A detailed description of the RET box is provided in Chapter 14.3 “Block end functions” on page 545.

The RET box requires a preceding logic operation and must only terminate a logic operation on its own.

In the second network in Fig. 8.36, the block with the RET box is left if the *“Totalizer”* block does not signal an error.

Open data block; OPN and OPNI boxes

To program the OPN or OPNI box, drag it with the mouse from the program elements catalog under *Basic instructions > Program control operations* to the working area. With OPN you open a data block via the DB register, with OPNI via the DI register.

You can subsequently set the function (OPN or OPNI) via a drop-down list which you can open using the small yellow triangle when the box is selected.

The OPN or OPNI box is present alone in a network without a preceding logic operation.

A detailed description of the OPN or OPNI box is provided in Chapter 14.5.1 “Open data block” on page 555.

Opening a data block using block parameters

A block parameter with parameter type BLOCK_DB allows the transfer of a data block (or more precisely: a data block number) to the called block. You call this data block as a global data block in the called block with the OPN box via the DB register.

Calling of a data block transferred with BLOCK_DB is not possible via the DI register.

Block change, CALL box

To program the CALL box, drag it with the mouse from the program elements catalog under *Basic instructions > Additional instructions* to the working area. With CALL, you call a code block which must not have any block parameters.

The CALL box is present alone in a network. If the CALL box does not have a preceding logic operation, the block call is carried out without conditions.

A detailed description of the CALL box is provided in Chapter 14.4.4 “Change to a block without block parameter” on page 551.

Block call functions

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 14.4 “Calling of code blocks” on page 547.

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.

Network 29 in Fig. 8.36 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 “0” and a jump is made to the network with the *Error* label. In network 30, 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.

8.6.5 Master Control Relay (MCR)

The Master Control Relay controls write operations to the user memory. A detailed description of the MCR functions is provided in Chapter 14.6 “Master control relay” on page 560.

You use the MCRA and MCRD boxes to define an MCR area. The two boxes are each present alone in a network.

You use the MCR< box to open an MCR zone. The box requires a preceding logic operation and terminates a logic operation. If the result of the logic operation is = “1”, the MCR dependency is switched off (“normal” processing); if the RLO = “0”, the MCR dependency is switched on.

The MCR> box closes an MCR zone and is present alone in a network.

To program the corresponding MCR function, drag it with the mouse from the program elements catalog under *Basic instructions* > *Additional instructions* to the working area. You can subsequently set the function (MCR<, MCR>, MCRA, or MCRD) via a drop-down list which you can open using the small yellow triangle when the coil is selected.

Fig. 8.37 shows the functions and networks required to switch the MCR dependency on and off.

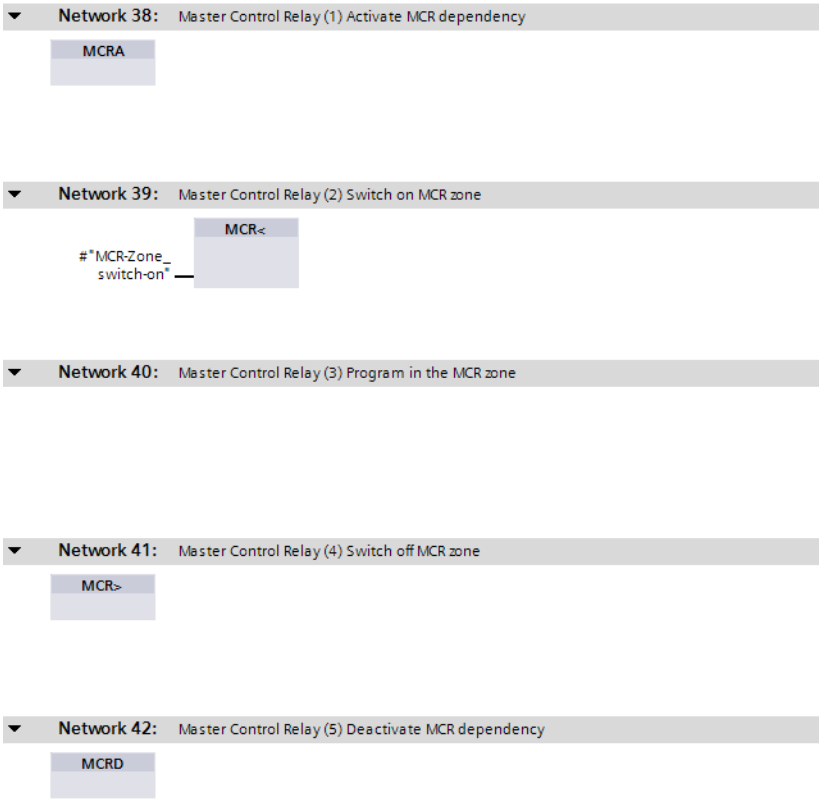


Fig. 8.37 MCR dependency and MCR zone in the FBD representation

If MCR dependency is switched on, the following system blocks influence the operands in the I/O range, in the process image, and in the bit memory address area:

- SET, SETI, and SETP set the parameterized operands to signal state “1”
- RESET, RESETI, and RESETP reset the parameterized operands to signal state “0”.

The system blocks can be found in the program elements catalog under *Basic instructions* > *Additional instructions*. A detailed description of these blocks is provided in Chapter 13.2.7 “Control memory area with MCR dependency” on page 485.

9 Statement list STL

9.1 Introduction

This chapter describes programming with statement list (STL); it uses examples to show how the program functions are represented in STL. You can find a description of the individual functions, e.g. comparison functions, in Chapters 12 “Basic functions” on page 427, 13 “Digital functions” on page 475, and 14 “Program flow control” on page 530.

Use of the program and symbol editor, which generally applies to all programming languages, is described in Chapter 6 “Program editor” on page 218.

STL is used to program the contents of blocks (the user program). What blocks are and how they are created is described in Chapters 5.2.3 “Block types” on page 156 and 6.3 “Programming a code block” on page 223.

9.1.1 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. 9.1 shows the structure of a block with the STL program. Located at the beginning of the program is the block header (block title) and the block comment. These are followed by the first network with its number, heading and comment. Further networks are optional. The first network shows a logic operation as example with AND and OR statements, a memory function, as well as an AND function with two assignments as termination. The second network shows the processing of digital values. Two digital values with data type DINT are added and the result converted to REAL before being transferred to a tag. A block need not be terminated by a special function, you simply terminate the program input.

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. Comments are commenced by two slashes, either as a line comment or a statement comment. You can insert empty lines to structure the sequence of statements. These and the comments have no effect on the control function.

In order to program an STL statement, use the keyboard to enter the operation in a line of the input field. Dragging the operation with the mouse from the program elements catalog under *Basic instructions* > *Basic instructions* > *Bit logic operations* is

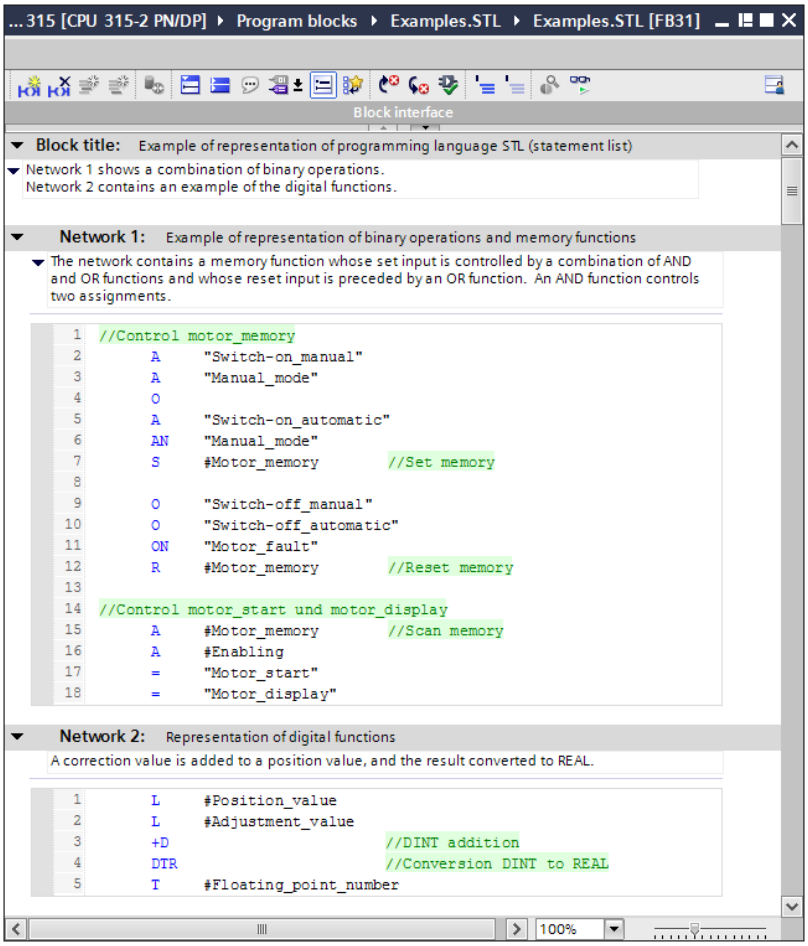


Fig. 9.1 Structure of a block with STL program

more laborious with STL then the direct input. However, the program elements catalog provides you with an overview of the existing operations.

9.1.2 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. 9.2 shows the structure 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.).

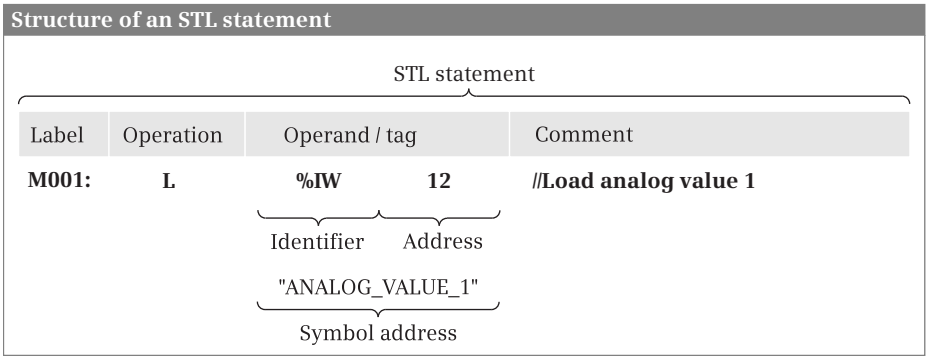


Fig. 9.2 Components of an STL statement

- An operand which contains the information necessary for executing the operation (e.g. an absolutely addressed operand %IW12, a symbolically addressed tag ANALOGVALUE_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 (only printable characters, no tabulators).

With a block call, the call operation is followed by the parameter list in round brackets.

9.2 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 9.1).

9.2.1 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. 9.3).

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.

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.

Table 9.1 Binary logic operations with STL

Operation	Operand	Function
A	Binary operand	Scan for signal state "1" and link according to AND logic operation
AN	Binary operand	Scan for signal state "0" and link according to AND logic operation
O	Binary operand	Scan for signal state "1" and link according to OR logic operation
ON	Binary operand	Scan for signal state "0" and link according to OR logic operation
X	Binary operand	Scan for signal state "1" and link according to exclusive OR logic operation
XN	Binary operand	Scan for signal state "0" and link according to exclusive OR logic operation
A(	–	Left parenthesis with AND logic operation
AN(		Left parenthesis with negation and AND logic operation
O(		Left parenthesis with OR logic operation
ON(		Left parenthesis with negation and OR logic operation
X(		Left parenthesis with exclusive OR logic operation
XN(		Left parenthesis with negation and exclusive OR logic operation
)		Right parenthesis
O	–	ORing of AND functions
NOT	–	Negation of result of logic operation
SET		Set result of logic operation to "1"
CLR		Set result of logic operation to "0"

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.

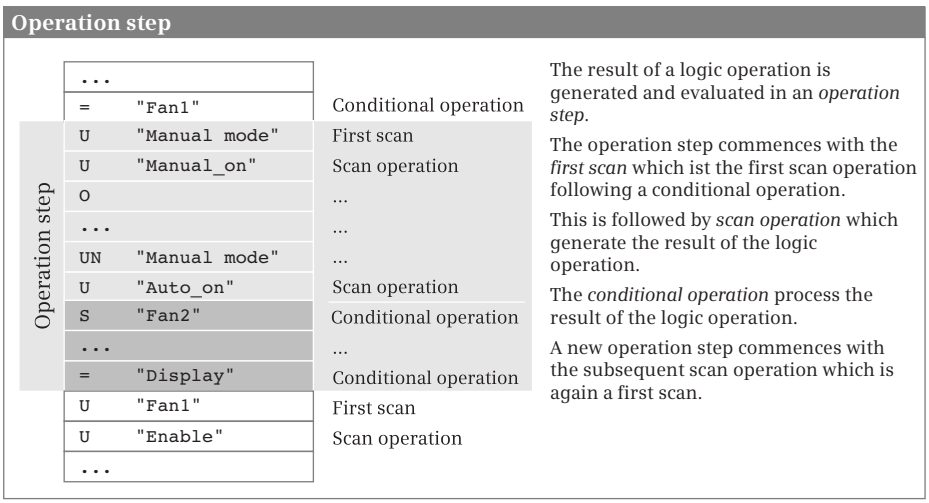


Fig. 9.3 Binary logic operation with STL, definition of operation step

Understandable programming

The logic operation specified for a first input bit scan is of no importance since the result of the scan is imported directly as the result of logic operation. To make the programming understandable, the logic operation specified for a first input bit scan should be identical to the desired function.

As an example, the sequence of statements

```
...
=   #Fan1.start
O   #Enable                    //First AND function
A   #Fan2.on
S   #Fan2.start
A   #Enable                    //Second AND function
A   #Fan2.off
R   #Fan2.start
...
```

represents two AND functions, where you should prefer the programming of the second AND function in which both scans are programmed according to AND.

For individual scan statements, the AND function is preferred, for example with

```
...
=   #Fan1.display
A   #Fan2.running              //Scan of #Fan2.running with
=   #Fan2.display              //assignment to #Fan2.display
...
```

9.2.2 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 428).

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. 9.4 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.

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”.

9.2.3 Programming a binary logic operation in the statement list

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, enter a binary tag from the operand areas: inputs, outputs, bit memories, and data. Binary logic operations can also be used to scan SIMATIC timer and counter functions, and status bits. The program editor supports you during input of operands by displaying all suitable, previously programmed tags following input of the first character.

In the same line you can enter a comment, separated by two slashes, up to the end of the line.

You terminate the binary logic operation with one or more conditional operations.

Example of scans for signal state “1” and signal state “0”

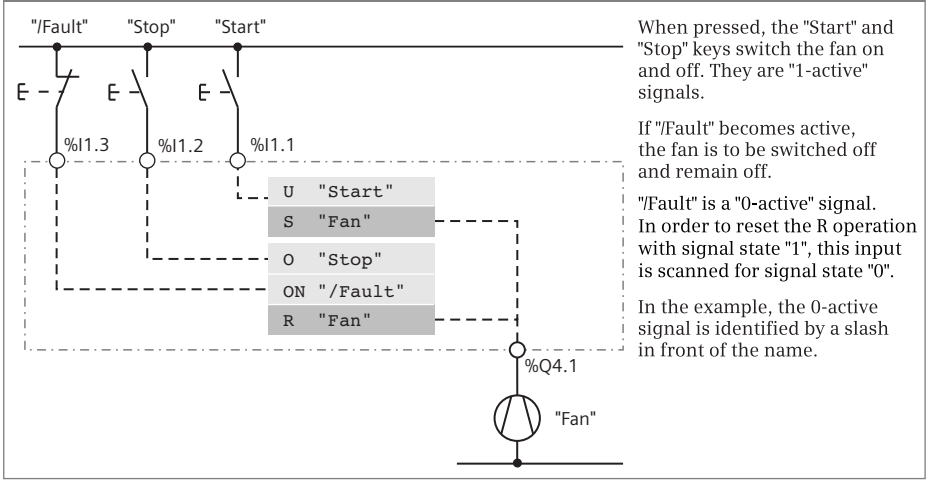


Fig. 9.4 Scanning for signal states “1” and “0”

You can start the next logic operation directly in the next line, or you can leave an empty line for clarity reasons. You can also commence a new network.

9.2.4 AND function

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 431.

Fig. 9.5 shows an example of an AND function. The *#Fan1.running* and *#Fan2.running* 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.

```
A     #Fan1.running
A     #Fan2.running
=     #Display.twoFans          //Two fans are running

O     #Fan1.running
O     #Fan2.running
=     #Display.MinOneFan        //At least one fan is running

X     #Fan1.running
X     #Fan2.running
=     #Display.oneFan           //Only one fan is running
```

Fig. 9.5 Example of binary logic operations with STL

9.2.5 OR function

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 432.

Fig. 9.5 shows an example of an OR function. The *#Fan1.running* and *#Fan2.running* 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.

9.2.6 Exclusive OR function

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 432.

Fig. 9.5 shows an example of an exclusive OR function. The *#Fan1.running* and *#Fan2.running* 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.

9.2.7 Combined binary logic operations

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. 9.6 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).

```
A     #Fan1.running
AN    #Fan2.running
O                          //ORing of AND functions
AN    #Fan1.running
A     #Fan2.running
=     #Display.oneFan_1    //Only one fan is running
```

Fig. 9.6 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. 9.7 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.

Parenthesized expressions in binary logic operations

The example in Fig. 9.7 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. 9.8 is a general representation of this schema.

```
A(
O     #Fan1.running
O     #Fan2.running
)
A(                              //ANDing of OR functions
ON    #Fan1.running
ON    #Fan2.running
)
=     #Display.oneFan_2         //Only one fan is running
```

Fig. 9.7 Example of ANDing of OR functions

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 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 be-

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.
Logic operation 1		
...	Delivers RLO 1	
Parenthesis function (		
...		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.
Logic operation 2		
...	Delivers RLO 2	
)	Following the parenthesis: RLO 3	
...		The result of the logic operation following the parenthesis is therefore:
Further logic operation		
...		

Fig. 9.8 Generally applicable schema for the processing of binary parenthesized expressions

ing 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. 9.9, 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.

```
A(
O     #Enable_manual
O     #Enable_auto
)                          //OR function is first AND input
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. 9.9 Example with conditional operations in a parenthesized expression

9.2.8 Control of result of logic operation

Negate RLO

The NOT operation negates the result of logic operation at any position in an operation. 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”.

The response of the negation is described in Chapter 12.1.6 “Negate result of logic operation, NOT contact” on page 433.

Fig. 9.10 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.

```
O     #Fan1.running
O     #Fan2.running
NOT                          //Negate RLO
=     #Display.noFan         //No fan is running
```

Fig. 9.10 Example of negation of result of logic operation

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. 9.11).

```
SET                          //Set RLO to "1"
S     #Fan1.drive            //Fan 1 is switched on
R     #Fan2.drive            //Fan 2 is switched off

CLR                          //Set RLO to "0"
CD    "Parts_counter"        //The internal edge trigger flag for
                             //counting down is reset
```

Fig. 9.11 Example of setting and resetting the result of logic operation

9.3 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 9.2).

To program a memory function, enter the operation in a line followed by a space and then the operand; in the case of a memory function, enter a binary tag from the operand areas inputs, outputs, bit memories, data, and temporary local data. A binary tag from the bit memory and data areas is ideal for the edge trigger flag.

Table 9.2 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

The program editor supports you during input of operands by displaying all suitable, previously programmed tags following input of the first character.

In the same line you can enter a comment, separated by two slashes, up to the end of the line.

9.3.1 Assignment

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 "Standard coil, assignment" on page 435.

In Fig. 9.12, 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. The result of logic operation is negated by NOT and, together with a further statement, controls the *#Display.noFan* tag.

```
O     #Fan1.running
O     #Fan2.running
=     #Display.MinOneFan        //At least one fan is running
NOT                             //Negate RLO
=     #Display.noFan            //No fan is running
```

Fig. 9.12 Example of assignment of result of logic operation

9.3.2 Setting and resetting

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 436.

In Fig. 9.13, an AND function comprising *#Fan1.enable* and *#Fan1.start* controls the set *#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 in-

put. *#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.

```
A     #Fan1.enable
A     #Fan1.start
S     #Fan1.drive              //Switch on fan 1
A     #Fan1.enable
A     #Fan1.stop
ON    #Fan1.fault
R     #Fan1.drive              //Switch off fan 1
```

Fig. 9.13 Example of set and reset operations

9.3.3 Edge evaluation

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 12.2.5 “Edge evaluation” on page 438).

```
A     #Alarm_bit
FP    #Alarm_bit_Edge_trigger_flag     //Evaluation for positive edge
S     #Alarm_memory
A     #Acknowledge
R     #Alarm_memory
```

Fig. 9.14 Example of edge evaluation with STL

Fig. 9.14 shows an application of edge evaluation. Let us assume that an 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”).

9.4 Programming timer and counter functions with STL

9.4.1 SIMATIC timer functions

Timer functions are used to implement dynamic processes in the user program. The SIMATIC timer functions are an operand area in the CPU's system memory and their number is limited. Table 9.3 shows the operations possible in conjunction with a timer operand. How the SIMATIC timer functions respond is described in detail in Chapter 12.3 “SIMATIC timer functions” on page 443.

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 time operand (T) to which you can assign a symbolic name in the PLC tag table.

Table 9.3 Operations for SIMATIC timer operands

Operation	Operand	Function
SP	Timer operand	Start a SIMATIC timer function as pulse
SE	Timer operand	Start a SIMATIC timer function as extended pulse
SD	Timer operand	Start a SIMATIC timer function as ON delay
SS	Timer operand	Start a SIMATIC timer function as retentive ON delay
SF	Timer operand	Start a SIMATIC timer function as OFF delay
FR	Timer operand	Enabling a SIMATIC timer function
R	Timer operand	Resetting a SIMATIC timer function
L	Timer operand	Direct loading of a time value
LC	Timer operand	Coded loading of a time value
A, AN	Timer operand	Status scan of a SIMATIC timer function and linking according to AND
O, ON	Timer operand	Status scan of a SIMATIC timer function and linking according to OR
X, XN	Timer operand	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 13.2.3 “Loading and transferring with STL” on page 478.

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. 9.15, the time *"Fan5.on_delay"* is started as an ON delay by the positive edge of *#Fan5.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 *"Fan5.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 5 is switched on following the ON delay. Once the start signal *#Fan5.start* has signal state "0", fan 5 continues to run for the follow-up time and is then switched off.

```
A     #Fan5.start
L     S5T#3S
SD    "Fan5.on_delay"          //Start as ON delay

A     "Fan5.on_delay"
L     #Follow_up_time
SF    "Fan5.off_delay"         //Start as OFF delay

A     "Fan5.off_delay"         //Scan status
=     #Fan5.drive
```

Fig. 9.15 Example of application of SIMATIC timer functions with STL

Example of clock generator

The somewhat more complex example in Fig. 9.16 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".

```
      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:   ...   //Further program
```

#Start_input starts the clock generator.

If the timer *"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 timer and thus also determines the duration – *#Pulse_duration* or *#Pause_duration* – with which the timer is started.

Fig. 9.16 Example of clock generator with different pulse-to-pause ratio

9.4.2 SIMATIC counter functions

Counter functions are used to implement counting tasks in the user program. The SIMATIC counter functions are an operand area in the CPU's system memory and their number is limited. Table 9.4 shows the counter operations possible in conjunction with a counter operand. How a SIMATIC counter function responds is described in detail in Chapter 12.5 “SIMATIC counter functions” on page 462.

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.

Table 9.4 Operations for SIMATIC counter operands

Operation	Operand	Function
CU	Counter operand	Increment a SIMATIC counter function by one unit
CD	Counter operand	Decrement a SIMATIC counter function by one unit
S	Counter operand	Set a SIMATIC counter function to a start value
FR	Counter operand	Enabling a SIMATIC counter function
R	Counter operand	Resetting a SIMATIC counter function
L	Counter operand	Direct loading of a count value
LC	Counter operand	Coded loading of a count value
A, AN	Counter operand	Status scan of a SIMATIC counter function and linking according to an AND logic operation
O, ON	Counter operand	Status scan of a SIMATIC counter function and linking according to OR
X, XN	Counter operand	Status scan of a SIMATIC counter function and linking according to an exclusive OR logic operation

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 13.2.3 “Loading and transferring with STL” on page 478.

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. 9.17 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

```
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. 9.17 Example of application of a SIMATIC counter function with STL

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.

9.4.3 IEC timer functions

Timer functions are used to implement dynamic processes in the user program. With a CPU 300, an IEC timer function is a system function block (SFB) in the operating system and is called in the user program like a function block. A detailed description of the IEC timer functions is provided in Chapter 12.4 "IEC timer functions" on page 459.

For programming, drag the corresponding symbol (TP, TON, or TOF) 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. 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.

Fig. 9.18 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.

```
CALL #MessageDelay               //Start as ON delay
     Time
     IN := #Measurement_too_high
     PT := T#10S                 //10 s duration
     Q  := #Message_too_high
     ET :=                       //ET is not required
```

Fig. 9.18 Example of IEC timer function with STL

9.4.4 IEC counter functions

A counter function implements counting processes in the user program. With a CPU 300, an IEC counter function is a system function block (SFB) in the operating system and is called in the user program like a function block. A detailed description of the IEC counter functions is provided in Chapter 12.6 “IEC counter functions” on page 470.

For programming, drag the corresponding symbol (CTUD, CTU or CTD) 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. 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. 9.19 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 globally 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.

```
A     "Light_barrier2"
FP    "Light_barrier2_Edge_trigger_flag"
A     "Light_barrier1"
=     #temp_bool1                          //Count up
A     "Light_barrier1"
FP    "Light_barrier1_Edge_trigger_flag"
A     "Light_barrier2"
=     #temp_bool2                          //Count down
      CALL #Lock_counter                   //Start as
         Int                               //Up/down counter
         CU   := #temp_bool1
         CD   := #temp_bool2
         R    := #Acknowledge
         LOAD :=
         PV   := 0
         QU   :=
         QD   :=
         CV   :=
```

Fig. 9.19 Example of IEC counter function with STL

9.5 Programming digital functions with STL

The digital functions process digital values mainly with the data types INT, DINT, and REAL. With a CPU 300, processing takes place in two registers of the control processor, the so-called accumulators. These are 32-bit wide memory locations which are addressed by the digital functions and special statements.

A digital function which manipulates a single value does this in accumulator 1 (Fig. 9.21). If two input values are required for a digital function, for example to add them, both accumulators are used. In this case the value for accumulator 2 is loaded first and then the value for accumulator 1. When loading for the second time, the value present in accumulator 1 is shifted into accumulator 2. The two values can then be linked. The result of the digital operation is subsequently present in accumulator 1.

9.5.1 Transfer functions

The transfer functions copy the value of a tag.

The *Load* operation transfers a digital value from the CPU's system memory or from a data block into accumulator 1. The *Transfer* operation transfers a digital value from accumulator 1 to the system memory or to a data block. A detailed description of the transfer functions is provided in Chapter 13.2.3 “Loading and transferring with STL” on page 478. Table 9.5 shows the transfer functions available with STL.

The program elements catalog contains the transfer functions under *Basic instructions* > *Basic instructions* > *Load and transfer*.

Fig. 9.20 shows an example of loading and transferring: The *#Messages* tag is transferred from the data block *“Data.STL”* to the *“Message_bits”* tag in the bit memory address area.

```
L     “Data.STL”.Messages          //Load value into accumulator 1
T     “Message_bits”               //Fetch value from accumulator 1
```

Fig. 9.20 Example of load and transfer statements

9.5.2 Comparison functions

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 comparison function is described in Chapter 13.3 “Comparison functions” on page 487. Table 9.6 shows the comparison functions available with STL.

The program elements catalog contains the comparison functions under *Basic instructions* > *Basic instructions* > *Comparator operations*.

Accumulator assignment with digital functions

Digital function with one parameter

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*	
	Accumulator 1	Accumulator 2
L #Input value	Input value	<Accumulator 1>
Function	Result	<Accumulator 1>
T #Result	Result	<Accumulator 1>

Digital function with two parameters

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*	
	Accumulator 1	Accumulator 2
L #Input value1	Input value 1	<Accumulator 1>
L #Input value2	Input value 2	Input value 1
Function	Result	Input value 1
T #Result	Result	Input value 1

Digital function with several parameters

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*	
	Accumulator 1	Accumulator 2
L #Input value1	Input value 1	<Accumulator 1>
L #Input value2	Input value 2	Input value1
Function	Intermediate result	Input value1
L #Input value3	Input value 3	Intermediate result
Function	End result	Intermediate result
T #End result	End result	Intermediate result

Fig. 9.21 Accumulator assignment with digital operations

Table 9.5 Transfer functions with STL

Operation	Operand	Function
L	Digital tag from the operand areas: peripheral inputs, peripheral outputs, inputs, outputs, bit memories, data, and temporary local data	Transfer to accumulator 1
L LC	SIMATIC timer function, SIMATIC counter function 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

Table 9.6 Comparison functions with STL

Operation	Operand	Function	Operation	Operand	Function
		Comparison for			Comparison for
==I	–	Equal to according to INT	==D	–	Equal to according to DINT
<>I		Not equal to according to INT	<>D		Not equal to according to DINT
>I		Greater than according to INT	>D		Greater than according to DINT
>=I		Greater than or equal to according to INT	>=D		Greater than or equal to according to DINT
<I		Less than according to INT	<D		Less than according to DINT
<=I		Less than or equal to according to INT	<=D		Less than or equal to according to DINT
		Comparison for			Comparison for
==R	–	Equal to according to REAL	>=R	–	Greater than or equal to according to REAL
<>R		Not equal to according to REAL			
>R		Greater than according to REAL	<R		Less than according to REAL
			<=R		Less than or equal to according to REAL

General execution of a comparison function

You program a comparison function according to the following general schema:

```
L     Tag1
L     Tag2
Comparison function
=     Result of comparison
```

The tags are compared according to the schema *Tag1 (compare) Tag2*.

A comparison function does not change the accumulator contents. It is always carried out independent of conditions. A comparison function sets the status bits.

Comparison function in a logic operation

The comparison function delivers a binary result of logic operation and can therefore be used together with other binary functions. The comparison function sets the status bit /FC, i.e. an operation step commences with the comparison function.

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
A     Input1
=     Output1
```

Output1 is set in the example if the comparison is fulfilled and *Input1* has signal state “1”.

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).

```
O     Input2
O(
L     Tag1
L     Tag2
Comparison function
)
O     Input3
=     Output2
```

Output2 is set in the example if *Input2* or *Input3* has signal state “1” or if the comparison is fulfilled.

Since a comparison function does not change the accumulator contents, it is possible in STL to repeatedly carry out successive comparisons.

```
L     Tag1
L     Tag2
>I
JC    Greater_than
==I
JC    Equal_to
```

In the example, two comparison functions are applied to the same accumulator contents. The first comparison generates RLO = “1” if *Tag1* is greater than *Tag2* 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.

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. The example shown above can also be programmed as follows:

```
L     Tag1
L     Tag2
>I
JP    Greater_than
JZ    Equal_to
```

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.

```
L     #Measurement_temperature
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
```

Fig. 9.22 Example of comparison function with STL

Two comparison functions are programmed in Fig. 9.22. 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.

9.5.3 Arithmetic functions

The arithmetic functions for numerical values implement the basic arithmetical operations with the data formats INT, DINT, and REAL in the user program.

An arithmetic function calculates a result from the values present in the accumulators 1 and 2 and stores it in accumulator 1. A detailed description of these arithmetic functions is provided in Chapter 13.4 "Arithmetic functions" on page 491. Table 9.7 shows the arithmetic functions available with STL.

The program elements catalog contains the arithmetic functions under *Basic instructions* > *Basic instructions* > *Math functions*.

Table 9.7 Arithmetic functions with STL

Operation	Operand	Function	Operation	Operand	Function
+I –I *I /I	–	Addition according to INT Subtraction according to INT Multiplication according to INT Division according to INT	+D –D *D /D	–	Addition according to DINT Subtraction according to DINT Multiplication according to DINT Division according to DINT
+R –R *R /R	–	Addition according to REAL Subtraction according to REAL Multiplication according to REAL Division according to REAL	MOD	–	Division according to DINT with remainder as result

You program an arithmetic function for two digital tags according to the following general schema:

```
L     Tag1
L     Tag2
Arithmetic function
T     Result of calculation
```

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. The content of accumulator 2 remains unchanged.

An arithmetic function carries out the calculation according to the specified characteristic independent of the contents of the accumulators and independent of conditions.

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 (divi-

Arithmetic functions, integer division

Program	*Accumulator assignment **following** execution of instruction*			
	Accumulator 1		Accumulator 2	
	31 16	15 0	31 16	15 0
`L value1`	0	Value 1 (dividend)	<Accumulator 1>	
`L value2`	0	Value 2 (divisor)	0	Value 1 (dividend)
`/I`	Remainder	Quotient	0	Dividend
`T result`	Remainder	Quotient	0	Dividend

The contents of the right word in accumulator 2 (value 1) are divided by the contents of the right word in accumulator 1 (value 2). 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. 9.23 Result of arithmetic function /I

dend) by the value in accumulator 1 (divisor) and delivers two results: the quotient and the remainder, both numbers with data type INT (Fig. 9.23).

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 (/I, /D, MOD) delivers a value of zero in each case as quotient and remainder and sets the status bits CC0, CC1, OV, and OS to “1”.

Successive arithmetic functions

You can permit an arithmetic function to directly follow a previous arithmetic function (chain calculation). The result of the first function is then linked further by means of the next function and the accumulators serve as intermediate memories.

```
L     Value1
L     Value2
+I
L     Value3
-I
T     Result1
```

Result1 := Value1 + Value2 – Value3

With a CPU 300, 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.

```
L     Value6
L     Value5
+R
+R
T     Result2
```

Result2 := Value5 + 2 × Value6

```
L     Value8
L     Value7
*D
*D
T     Result3
```

Result3 := Value7 × (Value8)^2

Example

In Fig. 9.24, the upper limit of a measured value is monitored. A hysteresis is introduced to ensure that the *#Measurement_too_high* message does not "pulsate" when the measured value changes rapidly around the upper limit. The message *#Measurement_too_high* is only canceled when the measured value has dropped again below the upper limit by the magnitude of the hysteresis.

```
L     #Measurement_temperature
L     #Upper_limit
>=I
S     #Measurement_too_high

L     #Upper_limit
L     #Hysteresis
-I                                //Subtraction with the
L     #Measurement_temperature    //data format INT
>I
R     #Measurement_too_high
```

Fig. 9.24 Example of arithmetic function with STL

9.5.4 Math functions

You can use the mathematical functions, for example, to implement trigonometric functions, exponential functions, and logarithmic functions with tags in data format REAL.

A math function calculates a result from the value present in accumulator 1 and saves it in accumulator 1. A detailed description of these math functions is provided in Chapter 13.5 "Math functions" on page 496. Table 9.8 shows the math functions available with STL.

The program elements catalog contains the arithmetic functions under *Basic instructions* > *Basic instructions* > *Math functions*.

Table 9.8 Math functions with STL

Operation	Operand	Function	Operation	Operand	Function
SIN COS TAN	–	Calculate sine Calculate cosine Calculate tangent	ASIN ACOS ATAN	–	Calculate arcsine Calculate arccosine Calculate arctangent
SQR SQRT	–	Generate square Extract square root	EXP LN	–	Generate exponential function to base e Generate natural logarithm (to base e)

Fig. 9.25 shows the calculation of the reactive power 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.

```
L     #Voltage
L     #Current
*R                              //Multiplication according to REAL
L     #phi
SIN                             //Calculate sine
*R                              //Multiplication according to REAL
T     #Reactive_power           //Save result
```

Fig. 9.25 Example of math functions with STL

9.5.5 Conversion functions

The conversion functions convert the data formats of tags.

A conversion function converts the value present in accumulator 1 and saves the result in accumulator 1. A detailed description of the conversion functions is provided in Chapter 13.6 “Conversion functions” on page 500. Table 9.9 shows the conversion functions available with STL.

Table 9.9 Conversion functions with STL

Operation	Operand	Function
ITD	–	Data type conversion from INT to DINT
ITB		Data type conversion from INT to BCD
DTB		Data type conversion from DINT to BCD
DTR		Data type conversion from DINT to REAL
BTI	–	Data type conversion from BCD to INT
BTD		Data type conversion from BCD to DINT
		Data type conversion from REAL to DINT
RND+	–	With rounding to the next higher number
RND–		With rounding to the next lower number
RND		With rounding to the next integer
TRUNC		Without rounding
INVI	–	Generation of the one's complement for INT
INVD		Generation of the one's complement for DINT
NEGI		Generation of the two's complement (negation) of an INT number
NEGD		Generation of the two's complement (negation) of a DINT number
NEGR		Negation of a REAL number
ABS		Generation of magnitude of a REAL number

General processing of conversion functions

The conversion functions are 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. The conversion functions do not change the contents of the remaining accumulators. You program a conversion function according to the following general schema:

```
L     Tag
Conversion function
T     Result
```

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.

Successive conversion functions

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 (Fig. 9.26).

```
L     #Measurement_temperature
ITD                           //Conversion INT to DINT
DTB                           //Conversion DINT to BCD32
T     #Measurement_display
```

Fig. 9.26 Example of conversion functions with STL

The program elements catalog contains the arithmetic functions under *Basic instructions* > *Basic instructions* > *Conversion operations*.

Fig. 9.26 shows an example of the conversion functions. A measured value present in data format INT is first expanded to the data format DINT and then converted into the BCD format.

9.5.6 Shift functions

With the shift functions you can shift the content of tags bit-by-bit to the left or right.

A 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. A detailed description of the shift functions is provided in Chapter 13.7 “Shift functions” on page 514. Table 9.10 shows the shift functions available with STL.

Table 9.10 Shift functions with STL

Operation	Operand	Function
		Shift word-by-word
SLW	n	To left with shift number as parameter
SLW		To left with shift number in accumulator 2
SRW	n	To right with shift number as parameter
SRW		To right with shift number in accumulator 2
SSI	n	With sign to right with shift number as parameter
SSI		With sign to right with shift number in accumulator 2
		Shift doubleword-by-doubleword
SLD	n	To left with shift number as parameter
SLD		To left with shift number in accumulator 2
SRD	n	To right with shift number as parameter
SRD		To right with shift number in accumulator 2
SSD	n	With sign to right with shift number as parameter
SSD		With sign to right with shift number in accumulator 2
		Doubleword rotation
RLD	n	To left with shift number as parameter
RLD		To left with shift number in accumulator 2
RRD	n	To right with shift number as parameter
RRD		To right with shift number in accumulator 2
RLDA	–	Doubleword rotation by one position to left by the condition code bit CC1
RRDA		Doubleword rotation by one position to right by the condition code bit CC1

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. You can program a shift function in two different ways:

▷ With the shift number in accumulator 2

```
L     Shift number
L     Input tag
Shift function
T     Result
```

▷ With the shift number as parameter

```
L     Input tag
Shift function + shift number
T     Result
```

The shift functions set status bit CC0 to “0” and status bit CC1 to the signal state of the last bit shifted out.

Successive shift functions

Shift functions can be applied as often as desired to the content of the accumulator. Example:

```
L     Value
SSD   4
SLD   2
T     Result
```

In the example 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”.

```
L     16
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 to right
T     #Quantity_display
```

```
L     #Quantity_high         //Shorter program
SLD   12                     //Shift doubleword-by-doubleword to left
L     #Quantity_low
OD
T     #Quantity_display
```

Fig. 9.27 Example of shift functions with STL

The program elements catalog contains the shift functions under *Basic instructions* > *Basic instructions* > *Shift and rotate*.

In Fig. 9.27, 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.

Table 9.11 Word logic operations with STL

Operation	Operand	Function
AW	W#16#xxxx	Word-by-word AND logic operation with the parameter
AW		Word-by-word AND logic operation with the content of accumulator 2
AD	DW#16#xxxx_xxxx	Doubleword-by-doubleword AND logic operation with the parameter
AD		Doubleword-by-doubleword AND logic operation with the content of accumulator 2
OW	W#16#xxxx	Word-by-word OR logic operation with the parameter
OW		Word-by-word OR logic operation with the content of accumulator 2
OD	DW#16#xxxx_xxxx	Doubleword-by-doubleword OR logic operation with the parameter
OD		Doubleword-by-doubleword OR logic operation with the content of accumulator 2
XOW	W#16#xxxx	Word-by-word exclusive OR logic operation with the parameter
XOW		Word-by-word exclusive OR logic operation with the content of accumulator 2
XOD	DW#16#xxxx_xxxx	Doubleword-by-doubleword exclusive OR logic operation with the parameter
XOD		Doubleword-by-doubleword exclusive OR logic operation with the content of accumulator 2

9.5.7 Word logic operations

The word logic operations link the individual bits of two tags according to AND, OR, or exclusive-OR.

A word logic operation links the values present in accumulators 1 and 2 in stores the result in accumulator 1. A detailed description of the word logic operations is provided in Chapter 13.8.1 “Word logic operations” on page 519. Table 9.11 shows the word logic operations available with STL.

Processing of a word logic operation

The word logic operations are carried out independent of conditions. The result of logic operation (RLO) is not affected. You can program a word logic operation in two different ways:

- Linking with a value in accumulator 2

```
L     Tag1
L     Tag2
Word logic operation
T     Result
```

- Linking with the parameter (constant)

```
L     Tag
Word logic operation + constant
T     Result
```

Word-by-word operation

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. 9.28).

16-bit logical operation

Program	*Accumulator assignment **following** execution of instruction*			
	Accumulator 1		Accumulator 2	
	31 16	15 0	31 16	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 bit memory doubleword %MD160 is present in the left word of the result (%MD170), and the ANDing of %MD162 and %MD166 in the right word (%MD172).

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. 9.28 Execution of a 16-bit word logic operation

Successive word logic operations

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:

```
L     Value1
L     Value2
AW
L     Value3
OW
T     Result1
```

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.

The program elements catalog contains the word logic operations under *Basic instructions* > *Basic instructions* > *Word logic operations*.

Fig. 9.29 shows how you can program 32 edge evaluations simultaneously for rising and falling edges. The message bits are collected in a doubleword *Messages*, which is present in the data block *"Data.STL"*. 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 as-

```
L     "Data.STL".Messages
L     "Data.STL".Messages_EM
XOD                           //What bits have changed?#
T     #Messages_change

L     #Messages_change
L     "Data.STL".Messages
AD                            //What change was a positive edge?
T     "Data.STL".Messages_st

L     #Messages_change
L     "Data.STL".Messages
INVD                          //Invert message bits
AD                            //What change was a negative edge?
T     "Data.STL".Messages_fa

L     "Data.STL".Messages
T     "Data.STL".Messages_EM  //Update edge trigger flag
```

Fig. 9.29 Example of word logic operations with STL

signment of *Messages* and *Messages_EM*, in other words: the associated message bit has changed.

In order to obtain the positive signal edges, the changes are linked to the messages by an AND logic operation. The bit is set for a rising signal edge wherever the message 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 message bits – the message bits with signal state “0” are now “1” – you obtain the pulse flags for a falling edge.

At the end it is only necessary for the edge trigger flags to track the messages.

9.6 Controlling the program flow with STL

You can influence processing of the user program by means of the program flow control functions. Using status bits you can recognize errors during the execution of digital functions, with the jump functions you can implement program branches, the block functions enable calling and termination of blocks, and the Master Control Relay enables influencing of output functions in complete program components.

9.6.1 Working with status bits in the statement list

Scanning status bits

The control processor saves internal statuses in the status bits during program execution. These statuses can be scanned using scan operations and jump functions. A detailed description of the status bits is provided in Chapter 14.1 “Status bits” on page 531. Table 9.12 shows the available scan operations.

Table 9.12 Scan operations for status bits with STL

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
	>0	Result greater than zero
	>=0	Result greater than or equal to zero
	<0	Result less than zero
	<=0	Result less than or equal to zero
	<>0	Result not equal to zero
	==0	Result equal to zero
	UO	Result invalid (unordered)
	OV	Overflow
	OS	Retentive overflow
	BR	Binary result

The program elements catalog contains the scan operations under *Basic instructions > Basic instructions > Bit logic operations*.

Example: In Fig. 9.30, 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. The status bit UO (invalid) is set if the floating-point number is invalid. The intermediate memory *#Floating_point_number_invalid* is set when this status bit is scanned and a JC jump is made to the *Error* label.

```
L     #Floating_point_number
L     1.0
==R
A UO                                  Scan for "Result invalid"
=     #Floating_point_number_invalid
JC    Error
```

Fig. 9.30 Example of scanning a status bit with STL

The status bits can also be scanned when loading the status word and the status bits are set when transferring the status word.

References to evaluation of a numerical range overflow

If the result of a calculation is outside the numerical range defined for the data type, it sets the status bit OV (overflow) and parallel to this the status bit OS (stored overflow).

If the result of a subsequent function is within the permissible numerical range, e.g. with a chain calculation, the status bit OV is reset. However, the status bit OS remains set so that a result overflow within a chain calculation is also recognized at the end of the calculation. OS is only reset by a jump function JOS or a change in block (call or block end). An example is shown in Fig. 9.31.

Save binary result

You can use the SAVE statement to save the result of logic operation in the binary result. The jump functions JCB and JNB also influence the binary result. A detailed description of SAVE is provided in Chapter 14.1.4 “Controlling the binary result” on page 535.

The program elements catalog contains SAVE under *Basic instructions > Basic instructions > Bit logic operations*.

Note that SAVE does not set the status bit /FC to “0”. This means that a binary logic operation cannot be aborted by SAVE. SAVE is used to save the current result of logic operation without influencing the logic operation.

```
//Binary scans
  L    #Value1
  L    #Value2
  +I
  A OV
  =    #Status1
  L    #Value3
  +I
  A OV
  =    #Status2
  L    Value4
  +I
  A OS
  =    #Group status
  T    #Result
```

```
//Jump functions
  L    #Value1
  L    #Value2
  +I
  JO   Label_ST1

  L    #Value3
  +I
  JO   Label_ST2

  L    Value4
  +I
  JOS  Label_ST

  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 number overflow for the complete calculation.

Fig. 9.31 Example of overflow evaluation

The binary result can be used as a group error message, for example. Fig. 9.32 shows an example of the collection of error messages in the last network of a block. The network is accessed via the *Error* jump label. In the event of an error message (*#Floating_point_number_invalid* with "1" and *#Adder_error* with "0"), the binary result BR if set to "0" before the block is left. The signal state of BR is transferred to the ENO output of this block.

```
Error :  AN   #Floating_point_number_invalid   //Error with "1"
         A    #Adder_error                     //Error with "0"
         SAVE                                  //Reset BR with error
         BE                                    //Block is left
```

Fig. 9.32 Example of setting of BR with SAVE

9.6.2 EN/ENO mechanism with STL

The EN/ENO mechanism with LAD, FBD, and SCL uses the enable input EN and enable output ENO. These sequences of statements represented as implicitly defined block parameters are not present with STL. However, you can program a group error message via the binary result BR and link block calls to each other.

Using BR as error message

If you program a block with STL which you wish to call in LAD, FBD or SCL, you should pay attention to the "correct" connection of the ENO output: The block is to be left with BR = "1" if no errors have occurred, and with BR = "0" if the processing was faulty.

Example: BR is set at the start of the block to “1”. If an error then occurs during block processing, e.g. a result exceeds a defined range, and therefore further processing should be stopped, use JNB to set the binary result to “0” and jump e.g. to the block end (in the case of an error, the condition must deliver signal state “0” here).

```
SET
SAVE                //BR = "1"
...
...
L     10_000
L     #Result       //If result > 10_000
<=I
JNB Error           //Then BR = "0" and jump to error
...
```

The following sequence of statements sets BR to “0” in the event of a numerical range overflow:

```
L     #Number1
L     #Number2
+I
T     #Total
AN OV               //Scan for numerical range overflow
SAVE                //With overflow: BR = "0"
```

The following programming is suitable if processing is to be aborted when an error is detected and a jump should be made to a program section with error evaluation:

```
L     #Number1
L     #Number2
+I
T     #Total
AN OV               //Scan for numerical range overflow
JNB   Error         //With overflow: BR = "0" and jump to error
```

Note that the jump functions JCB and JNB update the binary result according to the result of logic operation, even if the jump condition is not fulfilled. If you wish to use BR as a group error message in the block, you may only reset BR to “0” in the event of an error; BR must not be influenced if no errors are present.

In the case of a block which uses the enable output ENO as an error display, the binary result is set corresponding to the enable output ENO subsequent to the call statement and can be used further.

```
CALL  "Adder.STL"
      Number1 := "Data.STL".Number[1]
      Number2 := "Data.STL".Number[2]
      Number3 := "Data.STL".Number[3]
      Total := "Data.STL".Result[1]
JNB   Error                    //Jump with error in block
```

Track enable input EN

You can track the enable input EN with STL 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:

```
        ON     #Call              //If #Call is not fulfilled
        ON BR                     //or BR signals an error with "0"
        JC     M001               //the block is not processed
        CALL   "Adder.STL"
           Number1 := "Data.STL".Number[1]
           Number2 := "Data.STL".Number[2]
           Number3 := "Data.STL".Number[3]
           Total := "Data.STL".Result[1]
M001:   NOP 0
...
```

9.6.3 Jump functions

You use jump functions to exit linear program execution and continue at a different point in the block.

A detailed description of the jump functions is provided in Chapter 14.2 "Jump functions" on page 539. Table 9.13 shows the jump functions available with STL.

Table 9.13 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"
JNBI	Label	Jump if BR = "0"
JZ	Label	Jump if result equal to zero
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
JO	Label	Jump if overflow
JOS	Label	Jump if stored overflow
JL	Label	Jump list
LOOP	Label	Loop jump

General information

With STL the jump function consists of the jump operation and the jump destination (jump label). The jump operation specifies the condition under which the jump is carried out, the jump destination specifies the statement at which program execution is to be continued following the jump. The jump label (at the entry) is positioned in front of the operation and separated by a colon (Fig. 9.33).

	L	#Result	If the result is greater than 10 000, program execution is continued at the *Error* jump label.
	L	10_000	
	>I		
	JC	Error	
	...		
Error:	L	#Number	Further program
	...		

Fig. 9.33 Example of a jump function

You can set the jump label prior to each statement in the block. An operation must always follow a jump statement. It is possible to jump within the block beyond network limits. If you use the Master Control Relay (MCR), the jump destination must be located in the same MCR zone or in the same MCR area as the jump function.

The program elements catalog contains the jump functions under *Basic instructions > Basic instructions > Program control operations*.

9.6.4 Jump list

The jump list **JL** enables specific (calculated) jumping to a program section in the block independent of a jump number.

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

	L	#Jump_number	The *#Jump_number* tag loaded in accumulator 1 defines the JU jump function to be executed in the list following JL. The jump label with the jump list JL defines the end of the list of JU statements.
	JL	End	
	JU	Label...	
	JU	Label...	
	...		
	JU	Label...	
End:	...	//Further program	
	...		

Fig. 9.34 General schema for programming the jump list

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. 9.34.

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.

9.6.5 Loop jump

The loop jump **LOOP** permits simplified programming of loops.

LOOP interprets the right word of accumulator 1 as an unsigned 16-bit number in the range from 0 to 65535.

During processing, LOOP initially decrements the content of accumulator 1 by one. If the value is not yet zero, the jump is carried out to the specified jump label.

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.

You program a loop jump in accordance with the general schema shown in Fig. 9.35.

`      L    #Number` `Next: T    #Counter`	The *#Number* tag contains the total number of executed loops.

`      ...       //Program`
`      ...       //in the`
`      ...       //loop`
	The *#Counter* tag contains the number of loops still to be executed.

`      L    #Counter` `      LOOP Next`	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. 9.35 General schema for programming the loop jump

During the first cycle, the default setting for *#Counter* is the number of loops to be executed. The content of *#Counter* is loaded into the accumulator at the end of the

program loop and decremented by the LOOP statement. If the battery content is not zero afterwards, the jump to the specified jump label – here *Next* – is performed and the *#Counter* tag is updated.

The loop jump does not change the status bits.

9.6.6 Block functions

The block functions are used to call and terminate code blocks and to open data blocks.

A detailed description of the block functions is provided in Chapters 14.3 “Block end functions” on page 545, 14.4 “Calling of code blocks” on page 547 and 14.5 “Data block functions” on page 553. Table 9.14 shows the block functions available with STL.

Table 9.14 Block functions with STL

Operation	Operand	Function
BEC BEU BE	–	Conditional block end Absolute block end Block end
CALL CALL CALL	Code block #Instance Code block, data block	Calling a function (FC) or a system function (SFC) Calling a function block (FB) or a system function block (SFB) as local instance Calling a function block (FB) or a system function block (SFB) as single instance
UC CC	Code block Code block	Absolute change to a block without parameter Conditional change to a block without parameter
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

The program elements catalog contains the block functions under *Basic instructions > Basic instructions > Program control operations*.

Calling of code blocks

Fig. 9.36 shows an example of calling a function and calling a function block.

The binary result is scanned directly following calling of the function (FC), and the #Adder_error tag is set to “0” in the event of an error. A jump is then made to an error program section.

The binary result is scanned directly following calling of the function block. The block is left if no errors are present – BR is then “1”.

```
//FC call and jump in event of error
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                                  //If BR = “0” then error
=     #Adder_error                    //Set error tag
JCN   Error                           //and abort processing
//FB call as single instance
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                                  //If BR = “1” then no error
BEC                                   //and block end
```

Fig. 9.36 Examples of block calls in STL

Open data block

The **OPN** statement opens the specified data block via the DB register, the **OPNDI** statement via the DI register. The data block can be addressed absolutely or symbolically or be a tag with parameter type BLOCK_DB.

```
OPN   “Motor_DB” //Symbolic addressing
OPN   %DB101     //Absolute addressing
OPN   #Motor     //Addressing via a block parameter
```

Opening of a data block is carried out independent of any conditions. It does not influence the result of logic operation or the accumulator contents; the nesting depth of the block calls is not changed.

The opened data block must be present in the work memory.

You need only open a data block if you wish to address a data operand individually (on the left in Fig. 9.37). Complete addressing together with the data block is recommended (right), and the program editor then takes over opening of the data block. Refer to Chapter 4.2.2 „Absolute addressing of tags“, section “Partial addressing of data operands” on page 99 for information on what you must observe with partial addressing.

Partial addressing	Complete addressing
OPN %DB12 L %DBW14 OPN %DB13 L %DBW18 +I OPN %DB10 T %DBW16	L %DB12.DBW14 L %DB13.DBW18 +I T %DB10.DBW16

Fig. 9.37 Example of opening a data block

Opening a data block using block parameters

A block parameter with parameter type BLOCK_DB allows the transfer of a data block (or more precisely: a data block number) to the called block. Example: If an input parameter *#Data block* has the parameter type BLOCK_DB, you can open a data block transferred as parameter:

```
OPN   #Data block  //Open via the DB register
OPNDI #Data block  //Open via the DI register
```

When calling a function block, you can also use a data block as instance data block; the data block is transferred as a block parameter with parameter type BLOCK_DB. Since the program editor is not able to check the data type of the data block used during runtime, you must make sure that the transferred data block also matches the called function block as an instance data block.

Example: A block parameter with parameter type BLOCK_DB and name *#Motor data* can be specified as an instance data block when calling a function block:

```
CALL "Motor control", #Motor data
     parameter1 := ...
     parameter2 := ...
     ...
```

9.6.7 Master Control Relay (MCR)

The Master Control Relay controls write operations to the user memory. A detailed description of the MCR functions is provided in Chapter 14.6 “Master control relay” on page 560.

You use the “MCRA” statement to activate an MCR area and the “MCRD” statement to deactivate the MCR area again. Within an MCR area, you can open an MCR zone using “MCR(” and close it again using “)MCR”. The program elements catalog contains the MCR functions under *Basic instructions* > *Basic instructions* > *Additional instructions*.

The “MCRA” and “MCRD” statements are executed independent of conditions and do not change any status bits.

The “MCR(” statement depends on the result of logic operation: If the RLO = “0”, the MCR dependency is activated in the following MCR zone and deactivated again with RLO = “1”. “MCR(” terminates a binary logic operation.

The “)MCR” statement is executed independent of conditions and terminates a binary logic operation.

The example in Fig. 9.38 shows an MCR area with two nested MCR zones. The MCR dependency in the first MCR zone and in the subordinate MCR zone is controlled by the *#Z1_control* tag. If *#Z1_control* has signal state “0”, neither *#Output1* nor *#Output2* can be set to “1”. In the nested MCR zone, the MCR dependency is controlled by *#Z2_control*. If *#Z1_control* has signal state “1”, *#Z2_control* with signal state “0” can prevent setting of *#Output2*. Table 9.15 provides a summary of the dependencies.

Table 9.15 MCR dependency with nested MCR zones (example in Fig. 9.38)

The #Z1_control tag is	The #Z2_control tag is	In Zone 1, the MCR dependency is	With the condition fulfilled, #Output1 is	In Zone 2, the MCR dependency is	With the condition fulfilled, #Output2 is
"1"	"1"	deactivated	set to "1"	deactivated	set to "1"
"1"	"0"	deactivated	set to "1"	activated	reset
"0"	"1" or "0"	activated	reset	activated	reset

```
MCRA              //Activate MCR
...
```

```
A    #Z1_control
MCR(              //Open MCR zone 1
A    #Input1
A    #Input2
=    #Output1
...
```

The *#Z1_control* tag is used to activate the MCR dependency in all MCR zones.

```
A    #Z2_control
MCR(              //Open MCR zone 2
A    #Input1
A    #Input2
=    #Output2
...
```

The *#Z2_control* tag is used to activate the MCR dependency in the nested MCR zone if the MCR dependency is deactivated in the “outer” MCR zone.

```
)MCR              //Close MCR zone 1
...
)MCR              //Close MCR zone 2
...
MCRD              //deactivate MCR
```

Fig. 9.38 Example of nested MCR zones

9.7 Further STL functions

This chapter describes the operations which have a direct effect on the contents of the accumulators and the nil operations. You can find the statements for indirect addressing in Chapter 4.3 “Indirect addressing” on page 103.

9.7.1 Accumulator functions

The accumulator functions transfer values between the accumulators or swap bytes in 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 9.16 shows the accumulator functions of a CPU 300 available in the STL programming language.

Table 9.16 Accumulator functions of a CPU 300 (STL)

Operation	Operand	Function
+	Constant	Add a constant value to the content of accumulator 1
DEC INC	Constant Constant	Decrement the content of accumulator 1 by a constant value Increment the content of accumulator 1 by a constant value
POP PUSH TAK CAW CAD	– – – – –	Shift the content of accumulator 1 to accumulator 2 Fetch back the content of accumulator 2 to accumulator 1 Swap the contents of accumulators 1 and 2 Swap the bytes in the right word of accumulator 1 Swap the bytes in accumulator 1

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* (+, DEC, INC), and under *Basic instructions* > *Basic instructions* > *Conversion operations* (CAW, CAD).

Direct transfer between the accumulators

The principle of operation of the PUSH, POP, and TAK accumulator functions is shown in Fig. 9.39.

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.

The **TAK** statement swaps the contents of accumulators 1 and 2.

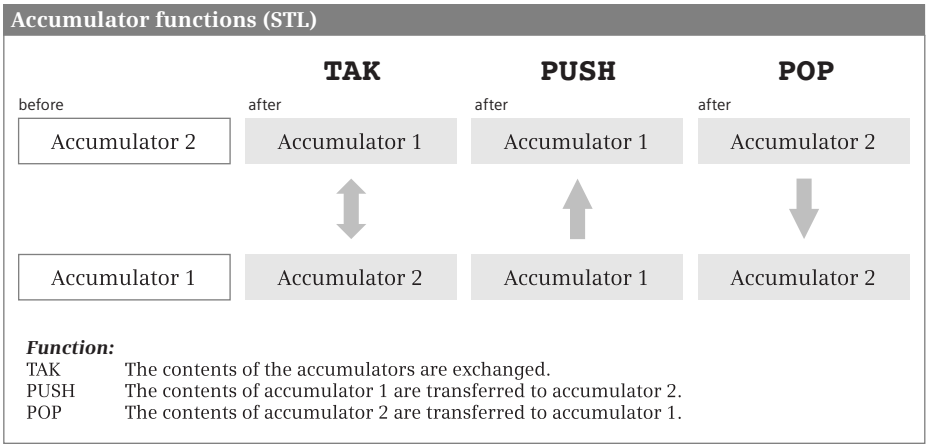


Fig. 9.39 Direct transfer between the accumulators

Swap bytes in accumulator 1

CAW Swap bytes in the right word in accumulator 1

CAD Swap bytes in the complete accumulator 1

The **CAW** statement swaps the two right bytes in accumulator 1 (Fig. 9.40). 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.

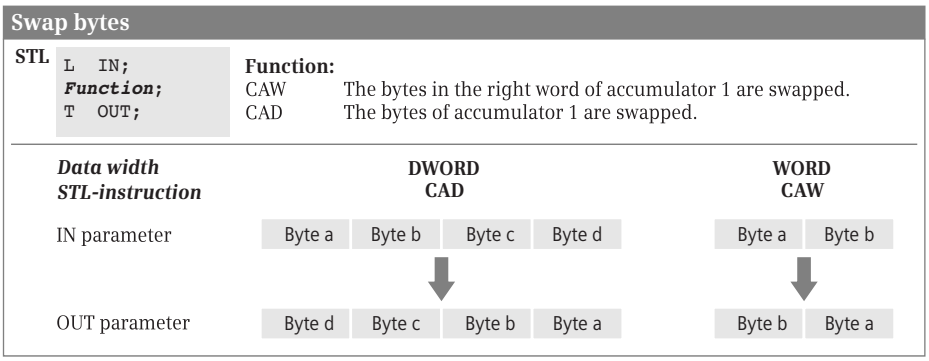


Fig. 9.40 Swapping bytes in accumulator 1

9.7.2 Adding of constants to accumulator 1

The addition of constants to accumulator 1 is used in the programming language STL (Fig. 9.41). In the other programming languages, the arithmetic addition is used for this.

```
+   16#bb     //Adding of a byte constant
+   ±w        //Adding of a word constant
+   L#±d      //Adding of a doubleword constant
```

Decrementing, incrementing

STL

```
L  #Value
DEC  <Decrement>
T  #Result

L  #Value
INC <Increment>
T  #Result
```

Function:

DEC reduces the value in accumulator 1 by the specified parameter; INC increases it by the specified parameter. The parameter can be within the range 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.

Addition of constant to accumulator 1 (STL)

STL

```
L  #Value
+  <±Constant>
T  #Result
```

Function:

The specified constant - which can also be negative - is added to the contents of accumulator 1 without changing the assignment of accumulator 2. If the constant has a byte or word width, only the right word in accumulator 1 is changed. You can identify a constant of doubleword width by means of an "L#" in front of the constant, e.g. L#-123.

Fig. 9.41 Decrementing, incrementing, and addition of constant

You program the addition of a constant according to the following general schema:

```
L     Tag
Addition of constant
T     Result
```

The addition of a constant is preferably used for calculating addresses since – unlike an arithmetic function – it influences neither the contents of the remaining accumulators nor the status bits.

The “Add constant” statement adds the constant present in the operation to the content of accumulator 1. You can specify this constant as a byte constant in hexadecimal form or as a word and doubleword constant in decimal form. If you wish to carry out the addition of a word constant as a DINT calculation, write L# in front of the constant. If the specified decimal constant is greater than the INT numerical range, a DINT calculation is automatically carried out.

You can specify a minus sign for a decimal number in order to also carry out the subtraction of constants. Prior to the addition of a byte constant, this is extended to an INT number with the correct sign.

The addition of a byte or word constant only influences the right word in accumulator 1, as with a calculation with data type INT; transfer to the left word does not take place.

Bit 15 (the sign bit) is overwritten if the INT range of values is exceeded. The addition of a doubleword constant influences all 32 bits of accumulator 1 in accordance with a DINT calculation.

Execution of these statements is independent of conditions.

9.7.3 Decrementing, incrementing

Decrementing and incrementing are used in the programming language STL. These functions can be emulated in other programming languages (Fig. 9.41).

DEC n Decrement

INC n Increment

You program decrementing and incrementing according to the following general schema:

```
L     Tag
DEC   n            //Reduce content of tags by n
T     Result

L     Tag
INC   n            //Increase content of tags by n
T     Result
```

The decrement and increment 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 accumulator content is only changed in the right byte. Transfer to the bytes on the left is not carried out. The calculation is carried out “modulo 256”, i.e. when increasing above a value of 255, counting restarts at the beginning, or when decreasing below a value of 0, counting restarts at 255.

Execution of the decrement and increment statements is independent of the result of logic operation. They are always executed and influence neither the result of logic operation nor the status bits.

9.7.4 Null instructions

Null instructions result in no response whatsoever by the control processor during execution. Table 9.17 shows the null instructions available with STL.

The program elements catalog contains the null instructions under *Basic instructions > Basic instructions > Additional instructions*.

Table 9.17 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

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).

BLD instructions

The program editor uses BLD *nnn* display construction statements to integrate information for decompilation into the program.

10 Structured Control Language SCL

10.1 Introduction to programming with SCL

This chapter describes programming with Structured Control Language (SCL); it uses examples to show how the program functions are represented in SCL. You can find a description of the individual functions, e.g. comparison functions, in Chapters 12 “Basic functions” on page 427, 13 “Digital functions” on page 475, and 14 “Program flow control” on page 530.

Use of the program and symbol editor, which generally applies to all programming languages, is described in Chapter 6 “Program editor” on page 218.

SCL is used to program the contents of blocks (the user program). What blocks are and how they are created is described in Chapters 5.2.3 “Block types” on page 156 and 6.3 “Programming a code block” on page 223.

10.1.1 Programming with SCL in general

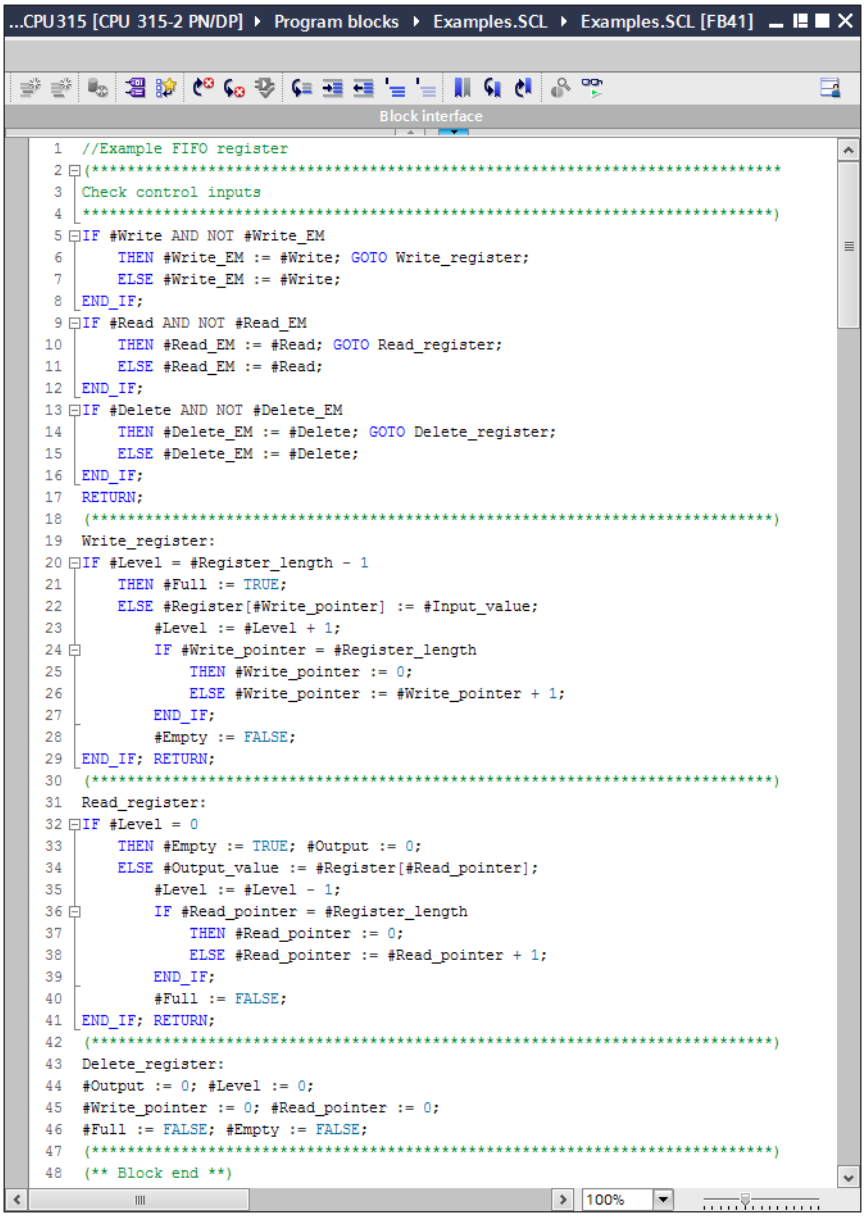
You use SCL to program the control function of the programmable controller – the user program (control program). The user program is organized in different types of blocks.

Fig. 10.1 shows the SCL program for a FIFO register. With a rising edge at *#Write*, this block writes the value present at the *#Input* parameter into a FIFO register. With a rising edge at *#Read*, the value at *#Output* is output again. The values are read out in the order in which they were written into the register (FIFO, first in first out). The register can be emptied using *#Delete*. The two displays *#Full* and *#Empty* show the status of the register (*#Full* and *#Empty* are each set following writing or reading). The block works with a write pointer and a read pointer.

The program editor constructs an SCL program line by line. You commence with the first statement in the first line. Each SCL statement is concluded by a semicolon. You can write several statements in one line or one statement can occupy several lines.

You can make the SCL program clearer and easier to read by using comments and empty lines. Comments and empty lines have no influence on the function of the SCL program.

Line comments commence with two slashes and terminate at the end of the line. Block comments commence with left parenthesis and asterisk, can extend over several lines, and terminate with asterisk and right parenthesis.




```
1  //Example FIFO register
2 (*****************************************************************************
3  Check control inputs
4  *****************************************************************************)
5 IF #Write AND NOT #Write_EM
6      THEN #Write_EM := #Write; GOTO Write_register;
7      ELSE #Write_EM := #Write;
8  END_IF;
9 IF #Read AND NOT #Read_EM
10     THEN #Read_EM := #Read; GOTO Read_register;
11     ELSE #Read_EM := #Read;
12 END_IF;
13 IF #Delete AND NOT #Delete_EM
14     THEN #Delete_EM := #Delete; GOTO Delete_register;
15     ELSE #Delete_EM := #Delete;
16 END_IF;
17  RETURN;
18  (*****************************************************************************)
19  Write_register:
20 IF #Level = #Register_length - 1
21     THEN #Full := TRUE;
22     ELSE #Register[#Write_pointer] := #Input_value;
23         #Level := #Level + 1;
24         IF #Write_pointer = #Register_length
25             THEN #Write_pointer := 0;
26             ELSE #Write_pointer := #Write_pointer + 1;
27         END_IF;
28         #Empty := FALSE;
29 END_IF; RETURN;
30  (*****************************************************************************)
31  Read_register:
32 IF #Level = 0
33     THEN #Empty := TRUE; #Output := 0;
34     ELSE #Output_value := #Register[#Read_pointer];
35         #Level := #Level - 1;
36         IF #Read_pointer = #Register_length
37             THEN #Read_pointer := 0;
38             ELSE #Read_pointer := #Read_pointer + 1;
39         END_IF;
40         #Full := FALSE;
41 END_IF; RETURN;
42  (*****************************************************************************)
43  Delete_register:
44  #Output := 0; #Level := 0;
45  #Write_pointer := 0; #Read_pointer := 0;
46  #Full := FALSE; #Empty := FALSE;
47  (*****************************************************************************)
48  (** Block end **)
```

Fig. 10.1 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. Dragging the statement with the mouse from the program elements catalog is of advantage with SCL if you import functions with a parameter list into your program. To call self-created blocks, drag them from the *Program blocks* folder into a line.

10.1.2 SCL statements and operators

The SCL program consists of a sequence of individual statements. Fig. 10.2 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 10.1. Operators of equal priority are processed from the left to the right.

Table 10.1 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	*, /, DIV, MOD	4
	Addition, subtraction	+, –	5
Comparison	Less than, less than-equal to, greater than, greater than-equal to	<, <=, >, >=	6
	Equal to, not equal to	=, <>	7
Binary logic operation	Negation (unary)	NOT	3
	AND logic operation	AND, &	8
	Exclusive OR	XOR	9
	OR logic operation	OR	10
Assignment	Assignment	:=	11

“Unary” means that this operator has a fixed assignment to an operand

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.

SCL instructions

SCL instruction

An SCL instruction consists of a jump label with subsequent colon and the actual instruction, which is terminated by a semicolon. The instruction can extend over several lines. The instruction 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 and the line comment can be omitted.

General SCL instruction

Label	:	SCL instruction	;	//	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 linking, 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 can be processed repeatedly. A control statement commences with a keyword (xxx) and is terminated by END_xxx.

Control statement

Label	:	**xxx**	Instruction sequence	**END_xxx**	;	//	Comment
		IF	Instruction sequence	**END_IF**	;		IF branch
		CASE	Instruction sequence	**END_CASE**	;		CASE branch
		FOR	Instruction sequence	**END_FOR**	;		FOR loop
		WHILE	Instruction sequence	**END_WHILE**	;		WHILE loop
		REPEAT	Instruction 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 element catalog are calls of system blocks with return value.

Block call

Label	:	Block name (parameter list)	;	//	Comment
		Tag := block name (parameter list)	;		

Fig. 10.2 Types of SCL statements

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.

10.2 Programming binary logic operations with SCL

The binary logic operations are executed in SCL with logic expressions in conjunction with binary tags or expressions which deliver a binary result. The binary operations can be "nested" using parentheses and thus influence the processing sequence (Table 10.2).

Table 10.2 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

10.2.1 Scanning for signal states "1" and "0"

The scanning of a binary operand in SCL is always the direct scanning of the status of the binary operand. This corresponds to scanning for signal state "1". If scanning for signal state "0" is required for the program function, one uses the NOT operator in order to negate the result of scan. NOT can also be used to negate the result of binary expressions.

The example in Fig. 10.3 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".

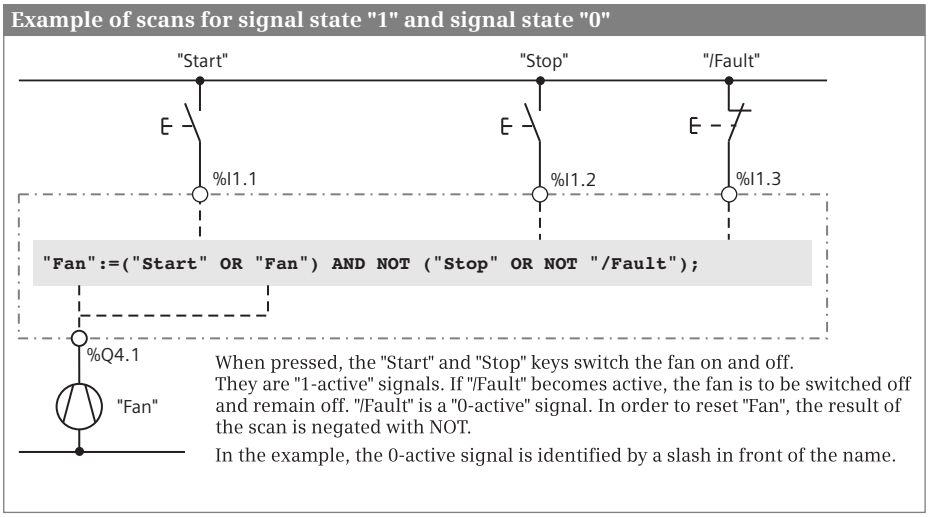


Fig. 10.3 Scanning for signal states “1” and “0”

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";
```

10.2.2 AND function

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 431.

SCL implements the AND logic operation using a logic expression with the operators & or AND, which link binary tags or binary expressions.

Fig. 10.4 shows an example of an AND logic operation. The *#Fan1.running* and *#Fan2.running* 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.running AND #Fan2.running;
//OR function
#Display.Min_oneFan := #Fan1.running OR #Fan2.running;
//Exclusive OR function
#Display.oneFan := #Fan1.running XOR #Fan2.running;
```

Fig. 10.4 Examples of binary logic operations with SCL

10.2.3 OR function

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 432.

SCL implements the OR logic operation using a logic expression with the operator OR, which links binary tags or binary expressions.

Fig. 10.4 shows an example of an OR logic operation. The *#Fan1.running* and *#Fan2.running* 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.

10.2.4 Exclusive OR function

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 432.

SCL implements the exclusive OR logic operation using a logic expression with the operator XOR, which links binary tags or binary expressions.

Fig. 10.4 shows an example of an exclusive OR logic operation. The *#Fan1.running* and *#Fan2.running* 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.

10.2.5 Combined binary logic operations

The AND, OR, and exclusive OR functions can be freely combined with one another. With SCL the operators have the following priority regarding execution: AND or & are executed before XOR, followed by OR. NOT is executed before the logic operation operators.

Logic operations such as the ORing of AND functions do not require parentheses, as shown in the top example in Fig. 10.5. 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 or if the second AND function is fulfilled (or if both are fulfilled, but this is not the case in this example).

```
//ORing of AND functions – does not require parentheses
#Display.oneFan_1 := #Fan1.running AND NOT #Fan2.running
     OR NOT #Fan1.running AND #Fan2.running;
//ANDing of OR functions – parentheses required
#Display.oneFan_2 := (#Fan1.running OR #Fan2.running)
     AND (NOT #Fan1.running OR NOT #Fan2.running);
```

Fig. 10.5 Examples of combined binary logic operations with SCL

This logic operation does not require parentheses since the AND function is processed “before” the OR function because of its higher priority. 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. 10.5. 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.

10.2.6 Negating result of logic operation

The NOT operator negates the result of logic operation at any position in an logic operation. Using the NOT operator it is possible in a simple manner to obtain:

- a NAND function (negated AND function, is fulfilled if at least one input has the result of scan “0”),
- a NOR function (negated OR function, is fulfilled if all inputs have the result of scan “0”), and
- an inclusive OR function (equivalence function, negated exclusive OR function, is fulfilled if an even number of inputs has the result of scan “1”).

Fig. 10.6 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.running AND #Fan2.running);

//NOR function – no fan is running
#Display.nor := NOT (#Fan1.running OR #Fan2.running);

//Inclusive OR function – neither of the fans or both fans are running
#Display.nxor := NOT (#Fan1.running XOR #Fan2.running);
```

Fig. 10.6 Examples of the negation of binary functions

10.3 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. The set and reset statements standard with LAD, FBD, and STL can be emulated.

10.3.1 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 “Standard coil, assignment” on page 435.

An example of a (binary) value assignment is shown in Fig. 10.7. 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.

10.3.2 Setting and resetting

The set and reset operations present with LAD, FBD, and STL (see Chapter 12.2 “Memory functions” on page 435) can be emulated with SCL, for example, using a simple IF branch.

In Fig. 10.7, 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. 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
#Fan1.drive := (#Fan1.start OR #Fan1.drive)
     AND NOT #Fan1.stop AND #Fan1.fault;
```

```
//Set tag with RLO = "1"
IF #Fan2.start THEN #Fan2.drive := TRUE; END_IF;
```

```
// Reset tag with RLO = "1"
IF #Fan2.stop OR NOT #Fan2.fault
     THEN #Fan2.drive := FALSE; END_IF;
```

Fig. 10.7 Assigning, setting, and resetting with SCL

10.3.3 Edge evaluation

Edge evaluation detects a change in a binary signal state.

With SCL, a change in signal state can be detected by comparing the current signal state with the previous one. The previous signal state is saved in a so-called edge trigger flag. This is, for example, a bit from the bit memories or data operand area.

Fig. 10.8 shows one example each with a rising (positive) edge and a falling (negative) edge.

With the first edge evaluation, a pulse flag (*#Message.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 *#Message.memory* tag is set to TRUE. Following pulse generation, the edge trigger flag must be updated.

The second edge evaluation is implemented using an IF statement. If a negative edge is detected, *#Message.memory* is reset to FALSE. This is followed by updating of the edge trigger flag.

```
//Set message memory with positive signal edge
#Message.pulse_pos := #Message.bit AND NOT #Message.edge_pos;
#Message.edge_pos := #Message.bit;
IF #Message.pulse_pos THEN #Message.memory := TRUE; END_IF;
//Reset message memory with negative signal edge
IF NOT #Message.ack AND #Message.edge_neg
       THEN #Message.memory := FALSE; END_IF;
#Message.edge_neg := #Message.ack;
```

Fig. 10.8 Examples of edge evaluation with SCL

10.4 Programming timer and counter functions with SCL

10.4.1 SIMATIC timer functions

Timer functions are used to implement dynamic processes in the user program. 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 10.3 shows the parameters possible in association with a function call of a SIMATIC timer. How the SIMATIC timer functions respond is described in detail in Chapter 12.3 “SIMATIC timer functions” on page 443.

Table 10.3 Call of SIMATIC timer functions with SCL

Timer function	Parameter	Data type	Description
	Function value	S5TIME, TIME	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	Timer operand (T) Start input Default time value Reset input Current integer-coded time value Binary status of timer function

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. 10.9, the time "*Fan5.on_delay"* is started as an ON delay by the positive edge of *#Fan5.start*. Following expiry of the duration, the timer function "*Fan5.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 5 is switched on following the ON delay. Once the start signal *#Fan5.start* has signal state "0", fan 5 continues to run for the follow-up time and is then switched off.

```
//Switch fan on and off with delay
#temp_S5Time := S_ODT(T_NO := "Fan5.on_delay", S := #Fan5.start,
     TV := S5T#3s, Q => #temp_bool);
#temp_S5Time := S_OFFDT(T_NO := "Fan5.off_delay", S := #temp_bool,
     TV := #Follow_up_time, Q => #Fan5.drive);
```

Fig. 10.9 Example of application of SIMATIC timer functions

10.4.2 SIMATIC counter functions

Counter functions are used to implement counting tasks in the user program. 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 10.4 shows the parameters possible in association with a function call of a SIMATIC counter. How a SIMATIC counter function responds is described in detail in Chapter 12.5 "SIMATIC counter functions" on page 462.

Table 10.4 Call of SIMATIC counter functions with SCL

Timer 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

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. 10.10 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.

```
//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. 10.10 Example of application of a SIMATIC counter function in SCL

10.4.3 IEC timer functions

Timer functions are used to implement dynamic processes in the user program. With a CPU 300, an IEC timer function is a system function block (SFB) in the operating system and is called in the user program like a function block. A detailed description of the IEC timer functions is provided in Chapter 12.4 "IEC timer functions" on page 459.

For programming, drag the corresponding symbol (TP, TON, or TOF) 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. The instance data block generated automatically when selecting as a single instance is saved in the project tree under *Program blocks* > *System blocks* > *Program resources*.

With the IEC timer functions, a binary tag must be connected to the IN input, and a duration to the PT input. You can also directly access the output parameters using the instance data, for example with *"DB_name".Q* for a single instance or *#Instance_name.Q* for a local instance.

Fig. 10.11 shows the IEC timer function *#Message_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, *#Message_too_high* is set.

```
//Message delay
#Message_delay(IN := #Measurement_too_high, PT := T#10s,
     Q => #Message_too_high);
```

Fig. 10.11 Example of IEC timer function with SCL

10.4.4 IEC counter functions

A counter function implements counting processes in the user program. With a CPU 300, an IEC counter function is a system function block (SFB) in the operating system and is called in the user program like a function block. A detailed description of the IEC counter functions is provided in Chapter 12.6 “IEC counter functions” on page 470.

For programming, drag the corresponding symbol (CTU, CTD, or CTUD) 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. 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. 10.12 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 globally 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. 10.12 Example of IEC counter function with SCL

10.5 Programming digital functions with SCL

The digital functions process digital values mainly with the data types INT, DINT, and REAL.

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.

10.5.1 Transfer function, value assignment of a digital tag

The “simple” transfer function corresponds with SCL to the value assignment.

A detailed description of the transfer functions is provided in Chapter 13.2 “Transfer functions” on page 476. Example of a value assignment: The value of the *#Messages* tag in data block *“Data.SCL”* is assigned to the *“Message_bits”* tag in the bit memory area.

```
"Message_bits" := "Data.SCL".Messages;
```

10.5.2 Comparison functions

A comparison function compares the values of two tags.

SCL implements the comparison function using a comparison operator. The comparison result has the data type BOOL and can be linked further like a Boolean tag. The comparison result has signal state TRUE if the comparison is fulfilled, otherwise FALSE. The comparison function is described in Chapter 13.3 “Comparison functions” on page 487. Table 10.5 shows the comparison operators available with SCL.

Table 10.5 Comparison functions with SCL

Operator	Description	Operand data types
= <> < <= > >=	Compare for equal Compare for unequal Compare for greater than Compare for greater than-equal Compare for less than Compare for less than-equal	INT, DINT, REAL, CHAR, STRING, TIME, DATE, TIME_OF_DAY
= <>	Compare for equal Compare for unequal	BOOL, BYTE, WORD, DWORD

Two comparison functions are programmed in Fig. 10.13. 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.

```
#Measurement_in_range := (#Measurement_temperature >= "Lower_limit") AND
     (Measurement_temperature <= "Upper_limit");
```

Fig. 10.13 Example of comparison expressions with SCL

10.5.3 Arithmetic functions

The arithmetic functions for numerical values implement the basic arithmetical operations with the data formats INT, DINT, and REAL. SCL uses an arithmetic operator for this.

The program elements catalog contains the arithmetic functions under *Basic instructions > Math functions*.

A detailed description of these arithmetic functions is provided in Chapter 13.4 “Arithmetic functions” on page 491. Table 10.6 shows the arithmetic operators available with SCL.

Table 10.6 Arithmetic functions with SCL

Operator	Description	Data type		
		1st operand	2nd operand	Result
**	Power	INT, DINT, REAL	INT	REAL
*	Multiplication	INT, DINT, REAL TIME	INT, DINT, REAL INT, DINT	INT, DINT, REAL TIME
/	Division	INT, DINT, REAL	INT, DINT, REAL	INT, DINT, REAL
DIV	Integer division	INT, DINT TIME	INT, DINT INT, DINT	INT, DINT TIME
MOD	Division with remainder as result	INT, DINT	INT, DINT	INT, DINT
+	Addition	INT, DINT, REAL TIME TOD DT	INT, DINT, REAL TIME TIME TIME	INT, DINT, REAL TIME TOD DT
–	Subtraction	INT, DINT, REAL TIME TOD DATE TOD DT	INT, DINT, REAL TIME TIME DATE TOD TIME	INT, DINT, REAL TIME TOD TIME TIME DT

The data type of the first and second operands can be INT, DINT, or REAL. If you link INT and DINT operands together, the result is of data type DINT; if you link an INT or DINT operand with a REAL operand, the result is of data type REAL. The program editor carries out a corresponding data type conversion (not visible to the user) prior to the arithmetic operation (see also Chapter 13.6.1 “Implicit data type conversion” on page 501).

In the case of a division, the second operand must not be zero.

In Fig. 10.14, the upper limit of a measured value is monitored. A hysteresis is introduced to ensure that the *#Measurement_too_high* and *#Measurement_too_low* messages do not “pulsate” when the measurement changes rapidly around the upper or lower limit. The messages are only canceled when the measured value has

dropped again below the upper limit or risen again above the upper limit by the magnitude of the hysteresis.

```
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. 10.14 Example of arithmetic expressions with SCL

10.5.4 Math functions

The mathematical functions comprise the trigonometric functions, exponential functions, and logarithmic functions, and deliver a result in data format REAL.

A detailed description of these math functions is provided in Chapter 13.5 “Math functions” on page 496. Table 10.7 shows the math functions available with SCL.

Table 10.7 Math functions with SCL

Operation	Function	Operation	Function
SIN COS TAN	Calculate sine Calculate cosine Calculate tangent	ASIN ACOS ATAN	Calculate arcsine Calculate arccosine Calculate arctangent
SQR SQRT	Generate square Extract square root	EXP LN	Generate exponential function to base e Generate natural logarithm (to base e)

The program elements catalog contains the arithmetic functions under *Basic instructions* > *Math functions*.

Fig. 10.15 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. 10.15 Example of math functions with SCL

10.5.5 Conversion functions

The conversion functions convert the data formats of tags and expressions.

A detailed description of the conversion functions – including a description of implicit conversion – is provided in Chapter 13.6 “Conversion functions” on page 500. Table 10.8 shows the (explicit) conversion functions available with SCL.

Table 10.8 Conversion functions with SCL

Operation	Operand	Function
CONVERT		Data type conversion
ROUND TRUNC		Data type conversion from REAL to DINT with rounding to the next integer Data type conversion from REAL to DINT without rounding

The program elements catalog contains the conversion functions under *Basic instructions* > *Conversion operations*.

When inserting the CONVERT function, the data types involved in the conversion are selected in a dialog box (Fig. 10.16).

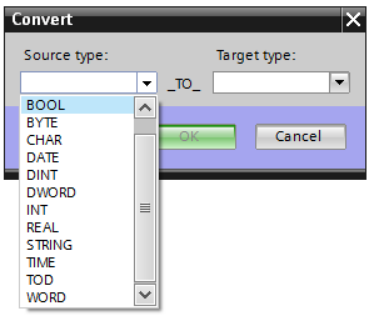


Fig. 10.16 Selection of data types with the CONVERT function

Table 10.9 shows the data type conversions possible with SCL. There are also the conversions WORD_TO_BLOCK_DB and BLOCK_DB_TO_WORD. If the permissible numerical range is left during a conversion, the ENO tag is set to FALSE and the result of the conversion is invalid.

```
#Measurement_display :=
      DINT_TO_BCD_DWORD(INT_TO_DINT(#Measurement_temperature));
```

Fig. 10.17 Example of conversion functions with SCL

Table 10.9 Data type conversion with SCL

from \ to	BOOL	BYTE	WORD	DWORD	INT	DINT	REAL	TIME	S5TIME	DT	TOD	DATE	CHAR	STRING	BCD16	BCD32
BOOL		X	X	X	K	K										
BYTE	X		X	X	K	K							X			
WORD	X	X		X	X	K										
DWORD	X	X	X		K	X	XX									
INT	K	K	X	K		X	X						X	S	B	
DINT	K	K	K	X	X		X	X			X	X		S		B
REAL				XX	X	X R								S		
TIME						X			T							
S5TIME								T								
DT					T1)						T	T				
TOD						X										
DATE						X										
CHAR		X			X									X		
STRING					S	S	S						X			
BCD16					B											
BCD32						B										

Data type conversion is possible:

- X With CONVERT
- S With S_CONV (integrated in CONVERT)
- T With T_CONV (integrated in CONVERT) 1) Conversion of DT to day of week
- B The BCD data types are handled like WORD or DWORD (example: BCD16 = WORD_BCD)
- R ROUND, TRUNC
- K By combination of two conversions (example: BOOL_TO_INT = WORD_TO_INT(BOOL_TO_WORD)
- XX The content of the tag (the value) is not converted!

Fig. 10.17 shows an example of the conversion functions. A measured value present in data format INT is first expanded to the data format DINT and then converted into the 7-decade BCD format.

10.5.6 Shift functions

A shift function shifts the content of a tag bit-by-bit to the left or right.

A detailed description of the shift functions is provided in Chapter 13.7 “Shift functions” on page 514. Table 10.10 shows the shift functions available with SCL.

The program elements catalog contains the shift functions under *Basic instructions > Shift and rotate*.

Table 10.10 Shift functions with SCL

Operation	Data types IN	Data type N	Function
SHR (IN, N) SHL (IN, N)	BYTE, WORD, DWORD BYTE, WORD, DWORD	INT, DINT INT, DINT	Shift to right Shift to left
ROR (IN, N) ROL (IN, N)	BYTE, WORD, DWORD BYTE, WORD, DWORD	INT, DINT INT, DINT	Rotate to right Rotate to left

```
#Quantity_display :=
      SHL(IN := #Quantity_high, N := 12) OR #Quantity_low;
```

Fig. 10.18 Example of shift functions with SCL

In Fig. 10.18, 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.

10.5.7 Word logic operations, logic expression

The word logic operations apply the binary operations AND, OR, and XOR to each bit of a word or doubleword.

A detailed description of the word logic operations is provided in Chapter 13.8.1 “Word logic operations” on page 519. Table 10.11 shows the word logic operations available with SCL.

Table 10.11 Word logic operations with SCL

Operation	Operand data types	Function
AND, & OR XOR	BOOL, BYTE, WORD, DWORD BOOL, BYTE, WORD, DWORD BOOL, BYTE, WORD, DWORD	AND logic operation OR logic operation Exclusive OR logic operation
NOT	BOOL, BYTE, WORD, DWORD	Negation

Fig. 10.19 shows how you can program 32 edge evaluations simultaneously for rising and falling edges. The message bits are collected in a doubleword *Messages*, which is present in data block *“Data.SCL”*. The edge trigger flags *Messages_EM* are also present in this data block. If the two doublewords are linked by an XOR logic operation, the result is a doubleword in which each set bit represents a different assignment of *Messages* and *Messages_EM*, in other words: the associated message bit has changed. In order to obtain the positive signal edges, the changes are linked to

the messages by an AND logic operation. The bit for a rising signal edge is set wherever the message 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 message bits – the message bits with signal state “0” are now “1” – you obtain the pulse flags for a falling edge. At the end it is only necessary for the edge trigger flags to track the messages.

```
#Message_changes :=
     "Data.SCL".Messages XOR "Data.SCL".Messages_EM;
"Data.SCL".Messages_pos :=
     #Message_changes AND "Data.SCL".Messages;
"Data.SCL".Messages_neg :=
     #Message_changes AND NOT "Data.SCL".Messages;
"Data.SCL".Messages_EM := "Data.SCL".Messages;
```

Fig. 10.19 Example of word logic operations with SCL

10.6 Controlling the program flow with SCL

You can influence processing of the user program by means of the program flow control functions. You can recognize errors in program execution by using the ENO tag, the control statements permit you to implement program branches, and the block functions allow you to call and terminate blocks.

10.6.1 Working with the ENO tag

The programming language SCL offers a pre-defined tag named ENO with data type BOOL, i.e. ENO is not declared by the user but is always present. This block-local tag shows FALSE to indicate an error in process execution in an SCL block.

In order to use automatic error detection with the ENO tag, the block attribute *Set ENO automatically* must be activated. When compiling the block, additional code is generated for controlling ENO. You activate the block attribute *Set ENO automatically* in the properties of the SCL block under *Attributes*.

Error analysis with ENO

At the block start, the ENO tag is always TRUE. ENO is set to FALSE if a called block signals an error or following faulty execution of an arithmetic expression or conversion function. Every error in the further block program also sets ENO to FALSE: ENO is used as a group error message for program execution in a block.

You can scan the ENO tag at any time:

```
#Total := #Total + #New_value;
IF NOT ENO                                  //Scan ENO
  THEN (* faulty addition *);
END_IF;
```

In this program, the THEN branch is even executed if faulty program execution took place prior to the addition which ENO also set to FALSE.

You can assign a value to the ENO tag at any time. If you only wish to check the correct execution of the addition (always assuming that 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).

10.6.2 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) and system function blocks (SFB), ENO for all block types which can be called. EN and ENO are not displayed by the program editor in the offered template.

EN is the first parameter in the parameter list, ENO the last. Use of these parameters is optional. If you do not require these parameters, simply omit them.

The EN/ENO mechanism is only supported in SCL if the block attribute *Set ENO automatically* is activated.

Enable input EN

You can control the calling of a block using the enable input EN. If EN is TRUE or not used, the called block is processed. If EN is FALSE, the called block is not processed. You use the enable input EN in the parameter list like an input parameter:

```
"Block"(EN := #Enable, In1 := ..., In2 := ...);
(* "Block" is only processed if #Enable = TRUE *)
```

You can use the enable input to implement a conditional block call, which depends on the value of a binary tag or binary expression.

Enable output ENO

You can scan the error status of the block using the enable output ENO. If ENO is TRUE, processing has been carried out correctly. If FALSE, the ENO output signals that an error is present in the block. You can scan the state of the ENO output in the parameter list using a tag:

```
"Block" (In1 := ..., In2 := ..., ENO => #OK);
(* With error-free processing, #OK has the value TRUE *)
```

If the called block signals an error, this is transferred to the “block-local” ENO tag:

```
"Block" (In1 := ..., In2 := ..., ENO => #OK);
#No_error := ENO;
IF NOT #OK THEN (* error in block *); END_IF;
IF NOT #No_error THEN (* group error message *); END_IF;
```

The *#OK* tag is FALSE if block processing was faulty. The *#No_error* tag is FALSE if block processing was faulty or if an error was already present prior to the block call.

If a function block with EN = FALSE is not processed, this has no influence on the “block-local” ENO tag. However, the ENO output is set to FALSE.

```
"Block" (EN := #Enable, ..., ENO => #OK);
#No_error := ENO;
```

If the *#Enable* tag is FALSE, the *#OK* tag is FALSE and the *#No_error* tag remains uninfluenced at its “old” value.

If you wish to use the EN/ENO mechanism 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. 10.20 provides a summary of how the enable output ENO and the ENO tag are controlled with a block call.

<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="2">Is EN = TRUE?</td><td rowspan="2">NO</td><td colspan="2" rowspan="2">Block/function being processed</td></tr>
<tr><td colspan="2">YES</td></tr>
<tr><td colspan="2">Block/function being processed</td><td rowspan="3">Block/function not being processed</td><td colspan="2" rowspan="2"></td></tr>
<tr><td colspan="2">Has an error occurred?</td></tr>
<tr><td>YES</td><td>NO</td><td>Has an error occurred?</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>

Fig. 10.20 Schematic for setting of enable output ENO and the ENO tag

10.6.3 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. 10.21).

Example: 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

Control statement IF

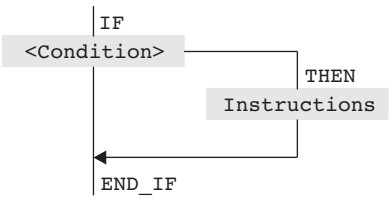
The control statement IF processes a program section <Instructions> depending on a Boolean value <Condition>. <Condition> can be a binary tag or an expression with a Boolean result.

Simple IF branch

```
IF <Condition>
   THEN <Instructions>;
END_IF;
```

If <Condition> has the value TRUE, the instruction block following THEN is then processed.

If <Condition> has the value FALSE, processing of the program is continued following END_IF.

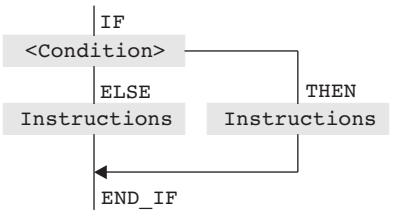


IF branch with ELSE

```
IF <Condition>
   THEN <Instructions>;
   ELSE <Instructions>;
END_IF;
```

If <Condition> has the value TRUE, the instruction block following THEN is then processed.

If <Condition> has the value FALSE, the instruction block following ELSE is then processed.



Cascaded IF branch

```
IF <Condition1>
   THEN <Instructions>;
   ELSIF <Condition2>
      THEN <Instructions>;
   ELSE <Instructions>;
END_IF;
```

A further condition is scanned by ELSIF ... THEN if the preceding condition is not fulfilled.

The ELSIF ... THEN instruction can be inserted cascaded: An ELSIF scan can again follow ELSIF ... THEN.

ELSE and the subsequent instructions can also be omitted.

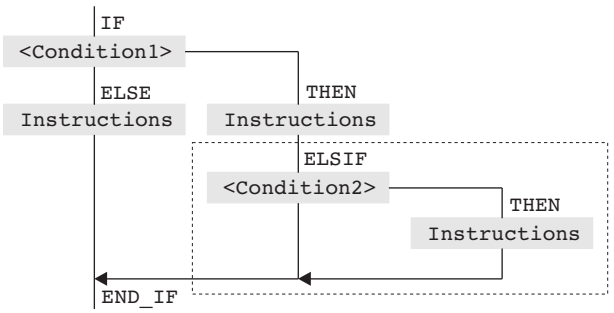


Fig. 10.21 Principle of operation of the IF branch

ELSIF is carried out. If neither of the two comparisons is fulfilled, the statements following ELSE are processed.

```
#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;
```

CASE statement

You can use the CASE statement to process one or more sequences of statements depending on an INT value (Fig. 10.22).

Control statement CASE

The control statement CASE processes a program section <Instructions> 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> : <Instructions1>;
   <Range2> : <Instructions2>;
   ...
   ELSE     : <Instructions0>;
END_CASE;
```

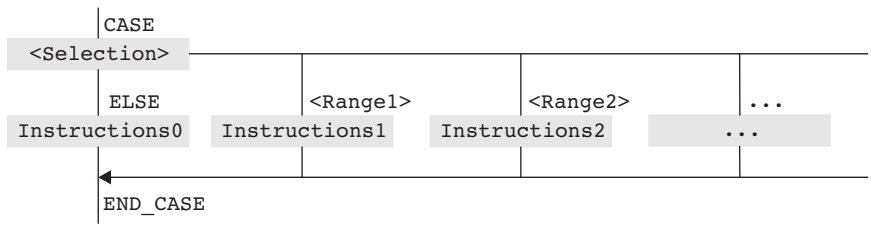


Fig. 10.22 Principle of operation of the CASE branch

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: 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;
```

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. 10.23).

Control statement FOR

The control statement FOR processes a program section <Instructions> for as long as a #Loop-control variable 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 <Instructions>;
END_FOR;
```

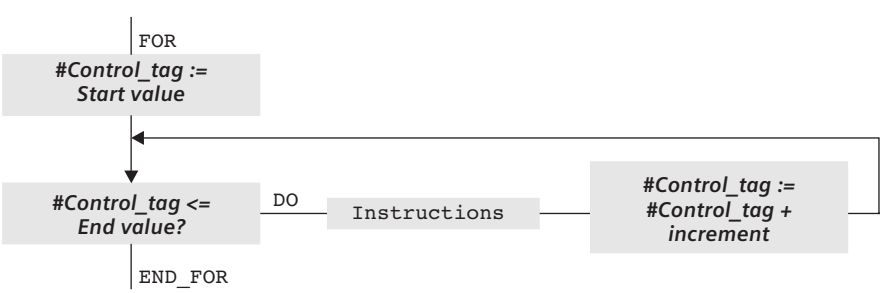


Fig. 10.23 Principle of operation of the FOR loop

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.

Control statement WHILE

The control statement WHILE processes a program section <Instructions> for as long as a <Condition> is fulfilled. <Condition> is a binary tag or an expression with binary result.

```
WHILE <Condition>
   DO <Instructions>;
END_WHILE;
```

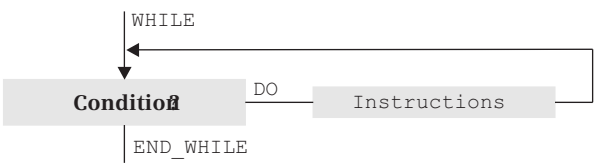


Fig. 10.24 Principle of operation of the WHILE loop

Example: The values of the I/O words %IW128:P to %IW142:P are transferred to the memory words %MW160 to %MW174.

```
FOR #i := 128 TO 142 BY 2 DO
    MW(#i + 32) := IW(#i):P;
END_FOR;
```

WHILE statement

The WHILE statement is used to repeatedly process a program loop for as long as a feasibility condition is fulfilled (Fig. 10.24).

<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 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: A search is carried out for the bit pattern 16#FFFF in the data block %DB10: Data word %DW0 contains either the value 16#FFFF or the interval to the next data word in which 16#FFFF or again the interval to the next data word could be present. The #k tag indicates how often the scan has been executed.

```
#i := 0; #k := 0;
WHILE NOT(DB10.DW(#i) = 16#FFFF) DO
  #i := #i + WORD_TO_INT(DB10.DW(#i)); #k := #k + 1;
END_WHILE;
```

REPEAT statement

The REPEAT statement is used to repeatedly process a program loop for as long as an abort condition is not fulfilled (Fig. 10.25).

<Condition> is an operand or expression with data type BOOL. The statements following REPEAT are repeatedly processed for as long as <Condition> is FALSE. <Condition> is scanned after each loop processing. If the value is TRUE, program execution is continued following END_REPEAT. The program loop is executed at least once, even if the abort condition is fulfilled right from the start.

REPEAT loops can be nested: Further REPEAT loops can be programmed within a REPEAT loop.

The current program execution can be aborted in the REPEAT loop using CONTINUE; EXIT terminates the complete REPEAT loop processing.

Example: The system block CREA_DBL is repeatedly called in the startup program until the data block has been created in the load memory.

```
REPEAT
  SFC_ERROR := CREA_DBL(REQ := TRUE, ..., BUSY := #t_Bool, ... );
UNTIL NOT #t_Bool
END_REPEAT;
```

Control statement REPEAT

The control statement REPEAT processes a program section <Instructions> for as long as a <Condition> is not fulfilled. <Condition> is a binary tag or an expression with binary result.

```
REPEAT
   <Instructions>;
   UNTIL <Condition>;
END_REPEAT;
```

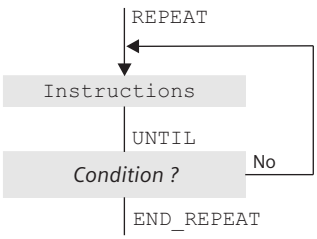


Fig. 10.25 Principle of operation of the REPEAT control statement

CONTINUE statement

CONTINUE finishes the current program execution in a FOR, WHILE or REPEAT loop (Fig. 10.26).

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: Memory bits are set by two nested FOR loops. If the byte address (#i) is equal to zero and if the bit address (#k) is less than 2, the subsequent statements of the inner FOR loop are not processed (setting commences with the memory bit %M0.2).

```
FOR #i := 0 TO 2 DO
   FOR #k := 0 TO 7 DO
      IF (#i = 0 & #k < 2) THEN CONTINUE; END_IF;
      MX(#i,#k) := TRUE;
   END_FOR;
END_FOR;
```

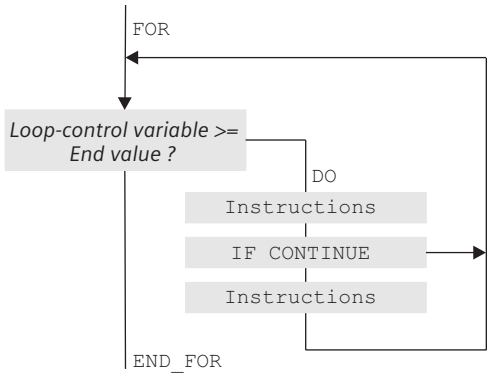
EXIT statement

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. 10.27).

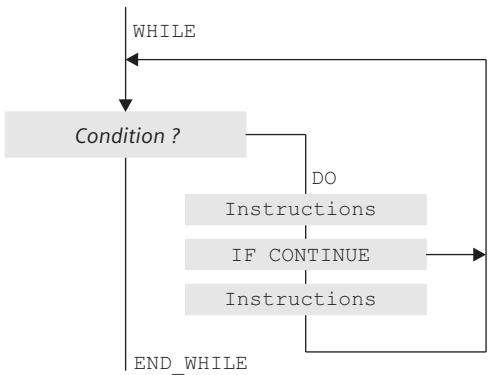
Control statement CONTINUE

The control statement CONTINUE finishes the current execution of a FOR, WHILE or REPEAT program loop. CONTINUE can be positioned anywhere in the instruction part of the loop.

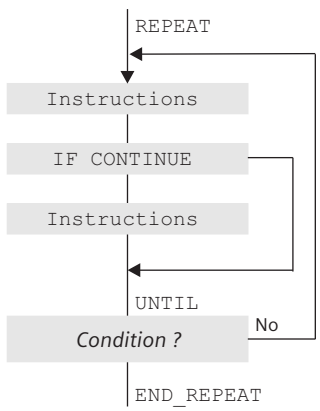
Finish execution of a FOR loop



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 instruction sequence:

```
IF <Condition>
   THEN CONTINUE;
END_IF;
```

the following is present in the representations:

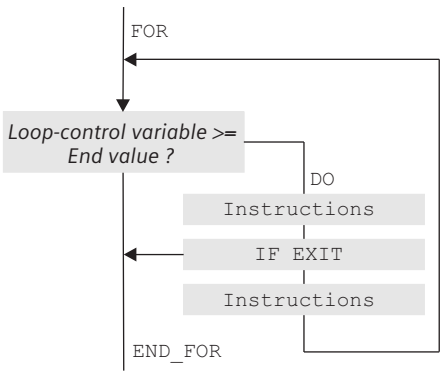
```
   IF CONTINUE
```

Fig. 10.26 Principle of operation of the CONTINUE control statement

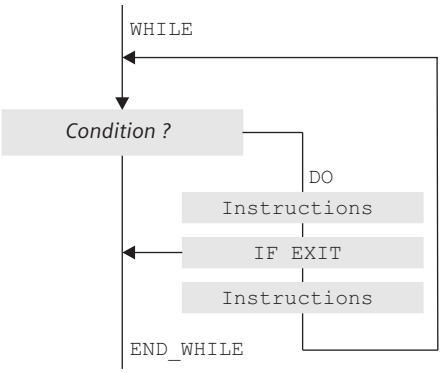
Control statement EXIT

The control statement EXIT finishes a FOR, WHILE or REPEAT program loop.
EXIT can be positioned anywhere in the instruction part of the loop.

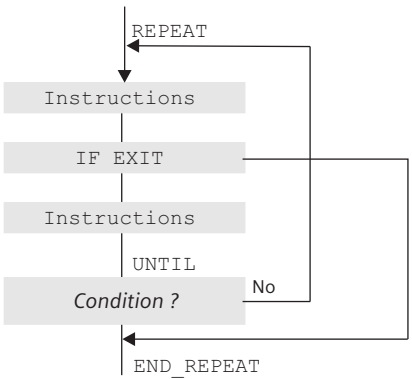
Cancel a FOR loop



Cancel a WHILE loop



Cancel a REPEAT loop



EXIT usually depends on a condition. This condition has the data type BOOL and can be a tag or an expression.

For the instruction sequence:

```
IF <Condition>
   THEN EXIT;
END_IF;
```

the following is present in the representations:

```
IF EXIT
```

Fig. 10.27 Principle of operation of the EXIT control statement

EXIT results in leaving of the loop which directly surrounds the EXIT statement.

Example: Memory bits are set by two nested FOR loops. If the byte address (#i) is equal to 2 and if the bit address (#k) is greater than 5, processing of the inner FOR loop is aborted (setting ends with the memory bit %M2.5).

```
FOR #i := 0 TO 2 DO
 FOR #k := 0 TO 7 DO
  IF (#i = 2 & #k > 5) THEN EXIT; END_IF;
  MX(#i,#k) := TRUE;
 END_FOR;
END_FOR;
```

In the example, processing of the FOR loop with the loop-control tag #k is aborted upon EXIT. Processing of the outer FOR loop with the loop-control tag #i is not influenced by this. However, the example is designed such that the EXIT statement becomes effective in the last execution of the “i-loop”.

10.6.4 Block functions

The block functions call and terminate blocks. A detailed description of the block functions is provided in Chapter 14.4 “Calling of code blocks” on page 547. Fig. 10.28 shows an example of the block functions with SCL.

```
//Block call FC without function value
"Adder.SCL"(Number_1 := #Measurements[1],
           Number_2 := #Measurements[2],
           Number_3 := #Measurements[3],
           Total => #Results[1]);
```

```
//Block call FC with function value
#Results[2] := "Adder2.SCL"(Number_1 := #Measurements[1],
                           Number_2 := #Measurements[2],
                           Number_3 := #Measurements[3]);
```

```
//Block call FB as single instance
"DB_Adder"(Value1 := #Interval[1],
          Value2 := #Interval[2],
          Value3 := #Interval[3],
          Result => #Position[1]);
```

```
//Block call FB as local instance
#Result(Value1 := #Interval[4],
        Value2 := #Interval[5],
        Value3 := #Interval[6],
        Result => #Position[2]);
```

```
//Example of supplying parameters with values
#Results[4] := Results[3] +
               LIMIT(MN := #Lower_limit + #Hysteresis,
                     IN := REAL_TO_INT(#Result.Result),
                     MX := #Upper_limit);
```

Fig. 10.28 Examples of block functions with SCL

When calling, an SFC system function is handled like an FC function, and an SFB system function block like an FB function block.

Terminate block with RETURN

The RETURN statement terminates processing in the current block.

The program elements catalog contains RETURN under *Basic instructions* > *Program control operations*.

Example: 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 (first example in Fig. 10.28).

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 second example in Fig. 10.28, the function value of the *"Adder2.SCL"* function is assigned to the *#Results[2]* 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.

The function block *"Adder"* is called in the third example in Fig. 10.28. 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.

The function block *"Adder"* is called in the penultimate example in Fig. 10.28. 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 last example in Fig. 10.28, 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.

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.

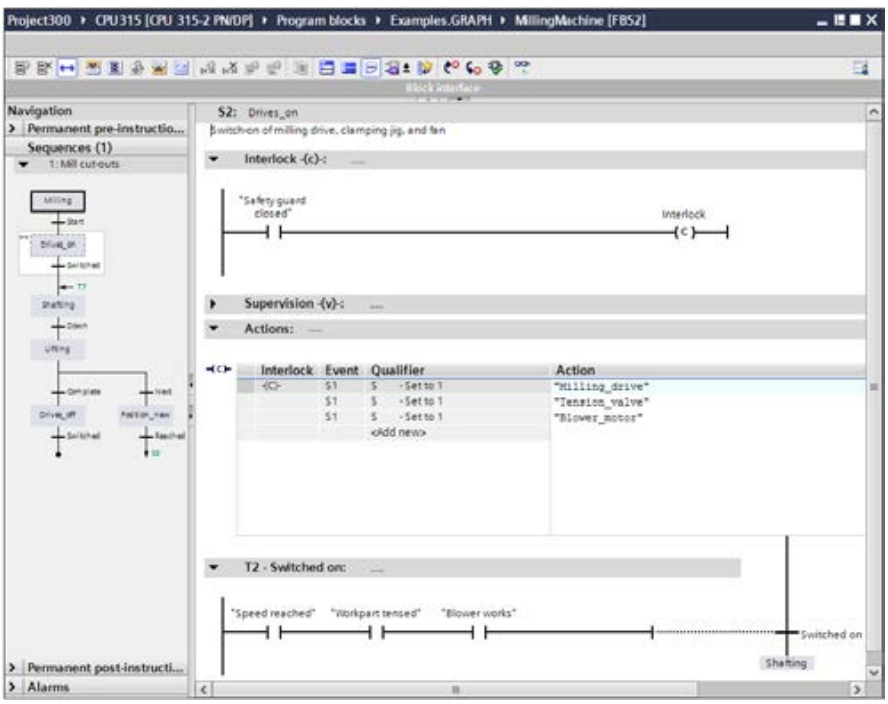


Fig. 11.1 Example of representation of a step in a sequencer

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 there can be several) and the following transitions are subsequently processed. The series-connected permanent instructions are processed following the sequencers.

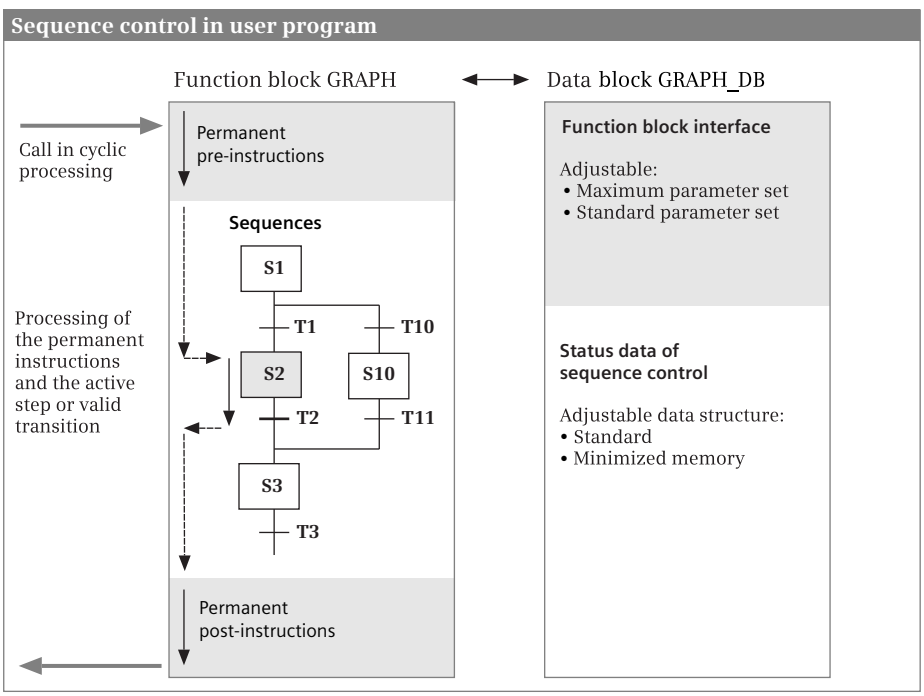


Fig. 11.2 Components and properties of a sequential control

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* option 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.

You set the structure of the instance data block in the properties of the function block by using the block attribute *Create minimized DB*. If the attribute is not activated, the data is stored successively in the data block ("classical" version, high memory requirements!). This provides the advantage that all block-local tags – unless they control the internal sequence – can be addressed. When using a data block with minimized memory requirements, access to block-local tags is limited.

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.

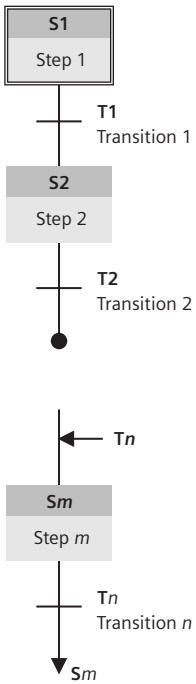
11.2 Elements of a sequential control

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** (S*n*) contains the actions to be executed in the form of instructions. The actions of a step are only carried out 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** (T*n*) 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 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 one 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).

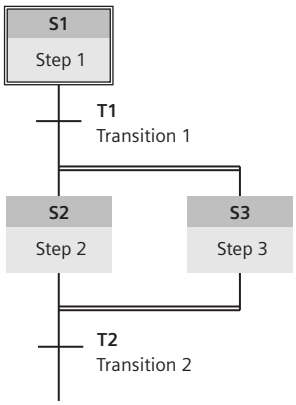
If a simultaneous branch is finished by a sequence end, this has no influence on the rest of the sequencer.

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



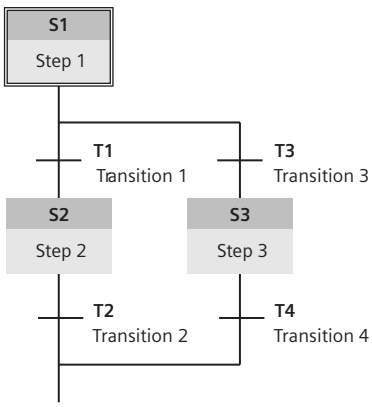
A **sequencer** can include branching; it is then divided into two or more branches. The branches can be combined together again or finish separately.

With **simultaneous branching**, the further branches are processed simultaneously. A simultaneous alternative branch starts and finishes with a step.

If the previous transition is fulfilled, the first steps of all simultaneous branches are activated simultaneously.

If simultaneous branches are combined 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 that alternative branch is processed whose start transition is fulfilled first. 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 fault-free 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, fault-free activation duration is greater than the configured fault-free 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.

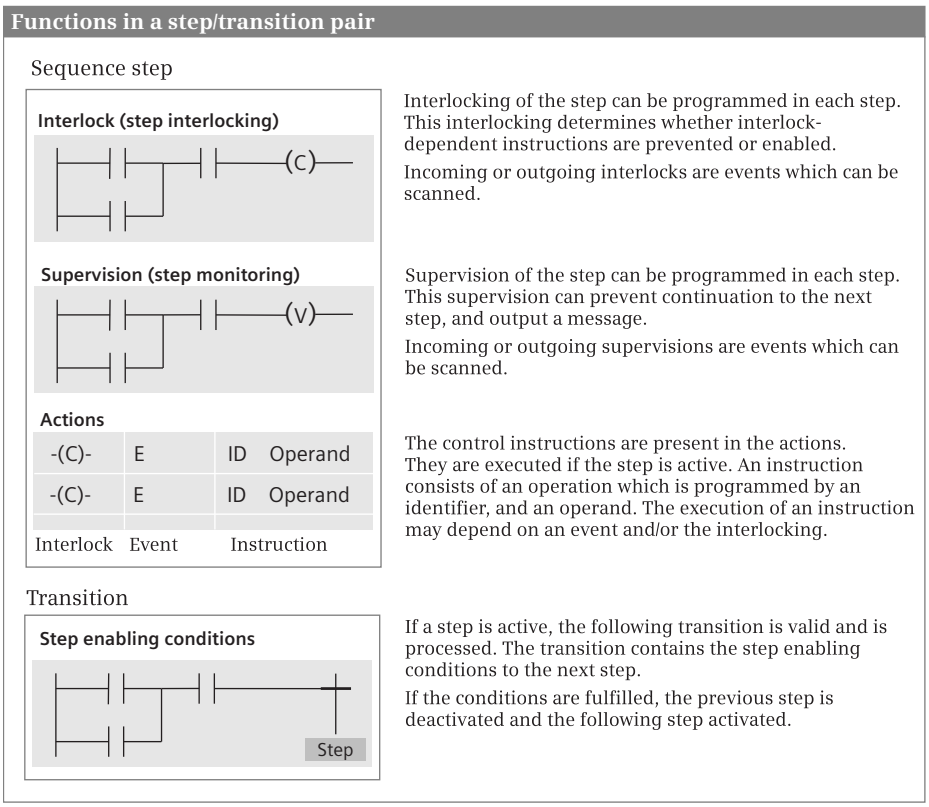


Fig. 11.5 Components of a sequence step/transition pair

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 fault-free 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 bit memory using the S operation. The instruction is S %M12.0 and means: As long as the step is active, the bit memory %M12.0 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 %M12.0 and means: As long as the step is active and as long as the condition is fulfilled, the bit memory %M12.0 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 %M12.0 and means: If the step is active and if the event – in this case the step activation – occurs, the bit memory %M12.0 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 %M12.0 and means: The bit memory 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 condition is satisfied Supervision condition is no longer satisfied	the supervision error (fault) occurs the supervision error is cleared
L1 L0	Interlock condition is not satisfied Interlock condition is satisfied	the interlock condition changes to “not fulfilled” (fault) the interlock condition changes to “fulfilled”
A1	An alarm is acknowledged	an alarm message is acknowledged
R1	Incoming registration	a registration is coming in (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 following operand areas can be controlled in a step: inputs I, outputs Q, bit memories M, and data DB.DBX. The tags have the data type BOOL and can be addressed absolutely or symbolically.

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.

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

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)

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 12.3.4 “Timer response as extended pulse” on page 451.

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 12.3.6 “Timer response as retentive ON delay” on page 455.

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 12.3.7 "Timer response as OFF delay" on page 457.

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 12.3.4 "Timer response as extended pulse" on page 451, 12.3.6 "Timer response as retentive ON delay" on page 455, and 12.3.7 "Timer response as OFF delay" on page 457.

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 12.5 "SIMATIC counter functions" on page 462.

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 a statement identified by "-(C)-", the interlock condition is optional and can also be omitted.

Table 11.5 Actions for 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 operation
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 a statement identified by "-(C)-", the interlock condition is optional and can also be omitted.

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.

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 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, or creating a “normal” or memory-optimized data block.

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).

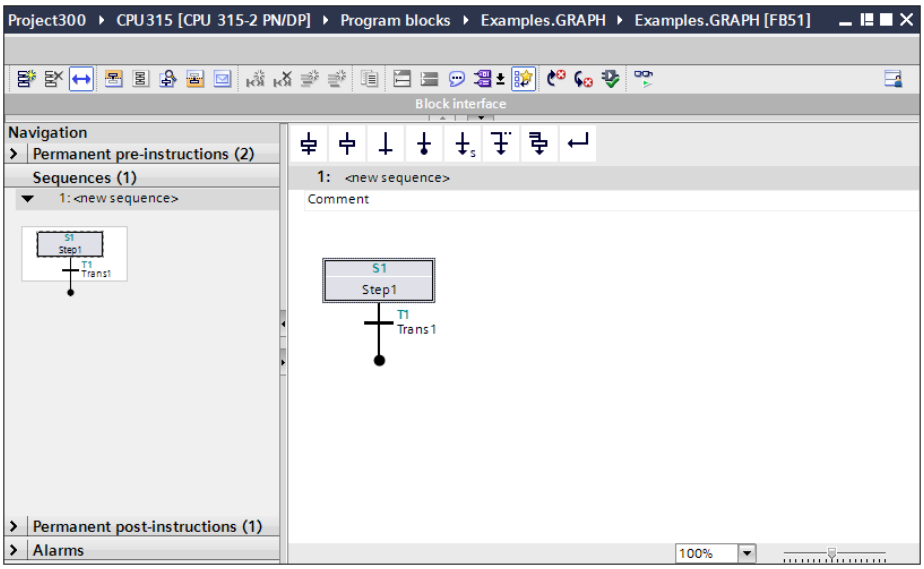


Fig. 11.6 Working window of the GRAPH editor

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 218.

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 program 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 program 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 then the *Renumber...* command 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 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 statements to be executed. You enter the statements in the *Qualifier* column. Click on *<Add new>* and then select the desired statement from the drop-down list. Specify the associated tag or operand in the *Action* column. You can control the display using the *Absolute/symbolic operands* symbol. In the *Event* column, select the event from a drop-down list for which the statement is to be executed. There is a mandatory entry for

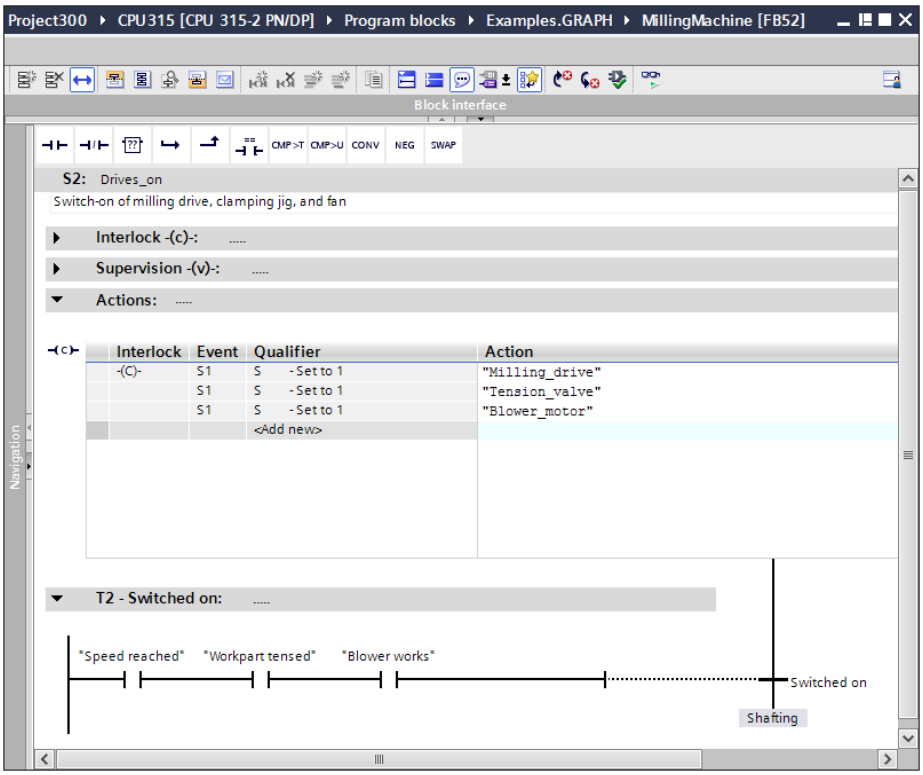


Fig. 11.7 Example of single step view

an edge-controlled statement; this is indicated by the <???> string highlighted in red. You specify in the *Interlock* column whether the statement specified in the *Qualifier* column depends on the step interlock.

Some statements 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.

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. 11.8).

The *IEC check* attribute indicates how strict the data type test is to be in the code block. With the attribute not activated, it is usually sufficient if the tags used have the data width required for execution of the function or instruction; with the attribute activated, the data types of the tags must correspond to the required data types.

If the *Create minimized DB* attribute is activated, the instance data block for the GRAPH block is created in minimized format. Despite the advantage of the reduced memory requirements, there are a number of limitations. For example, certain elements of the step and transition structures, messages, and the “Skip steps” option are not available, and the step and transition numbers are automatically numbered consecutively when compiling.

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.

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.

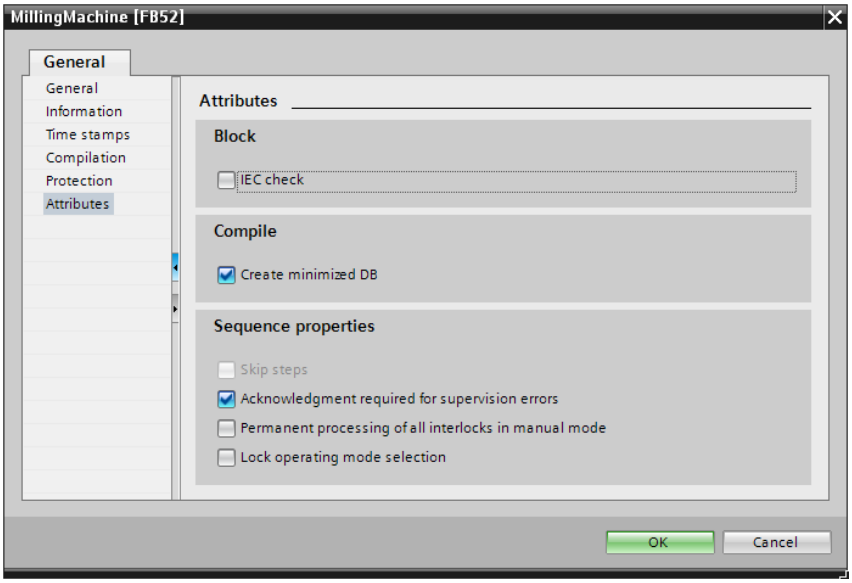


Fig. 11.8 Block attributes for the sequential control

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 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. Exception: GRAPH-specific parameters which are not available in the minimized data block cannot be added manually.

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 generating the instance data block, the GRAPH editor creates the PLC data types *GraphStep* or *GraphStepMin* and *GraphTransition* or *GraphTransitionMin* de-

pending on the *Create minimized DB* attribute. *GraphStep* and *GraphTransition* contain the GRAPH-specific tags for each step or transition (see Chapter 11.2.4 “GRAPH-specific tags” on page 403). 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 fault-free 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 239).

For optimum use, the GRAPH function block requires an additional system block. The system function SFC 73 G7_STD_4 is used if the *Create minimized DB* attribute is activated, or the system function SFC 72 G7_STD_3 with the standard memory space model. When compiling, the system block is stored 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 588.

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. 11.9 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 419.

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. 11.9 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.

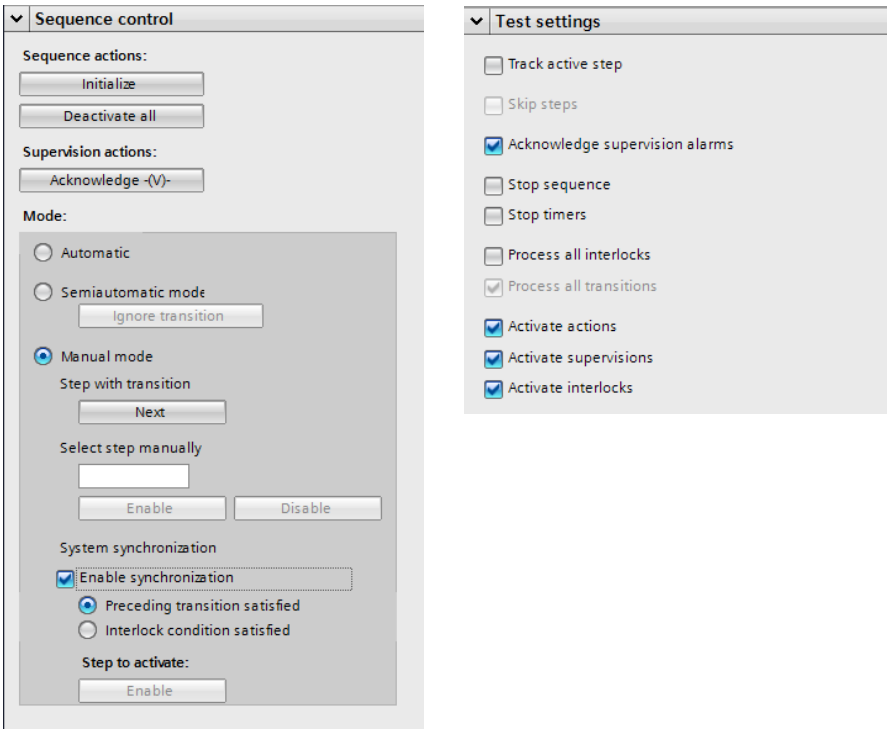


Fig. 11.9 Test aids for GRAPH in the *Testing* task card

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.

11.4.4 Synchronization of 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 589.

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.

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. 11.10). 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.

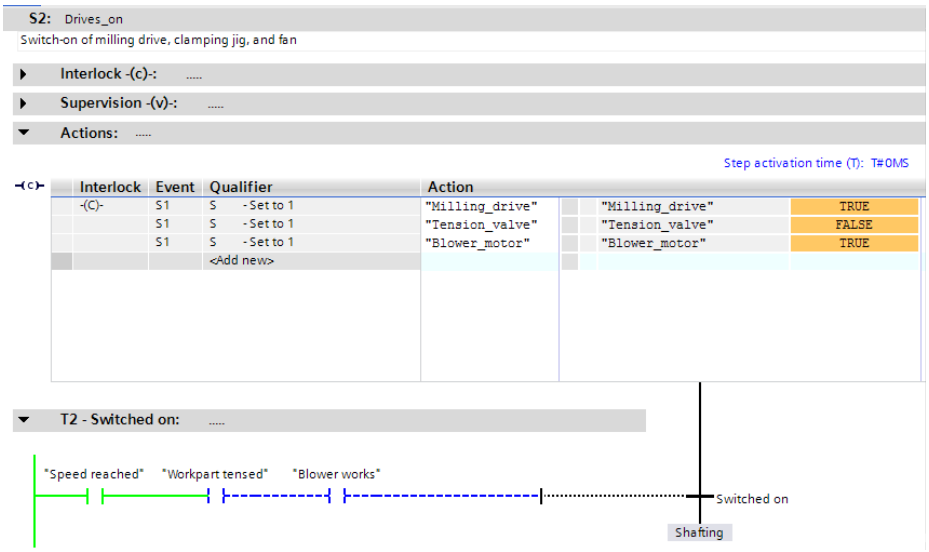


Fig. 11.10 Example of the program status in the single step view in LAD

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 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.

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 249, 8 “Function block diagram FBD” on page 282, 9 “Statement list STL” on page 315, and 10 “Structured Control Language SCL” on page 363 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.
- In the statement list (STL), the binary logic operations are scans positioned underneath each other.
- With the Structured Control Language (SCL), expressions with binary tags form the binary logic operations.

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, STL, SCL).
- 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 253, for FBD in Chapter 8.2.6 “Combined binary logic operations, negating result of logic opera-

tion" on page 288, for STL in Chapter 9.2.7 "Combined binary logic operations" on page 322, and for SCL in Chapter 10.2.5 "Combined binary logic operations" on page 369.

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).

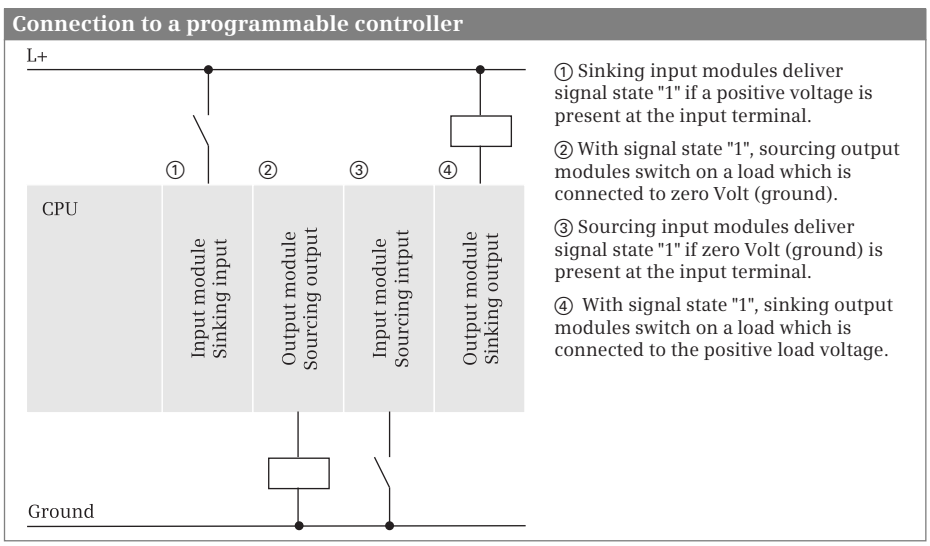


Fig. 12.1 Connection of binary signals to a programmable controller

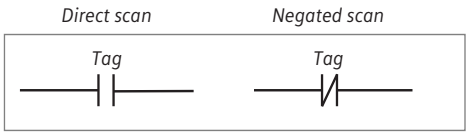
Types of sensors, NO contacts, and NC contacts

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 control 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).

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.

```
U   tag          UN  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).

```
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.

Fig. 12.2 Direct and negated scanning of a binary operand

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.

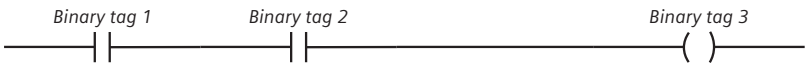
The result of logic operation is output in order to control an actuator (relay, contactor, lamp, etc.). The memory functions, primarily the assignment, are available for this purpose. An assignment takes over the original signal state of the result of logic operation. If it is necessary to output the negated signal state (the opposite result of logic operation), a negation is programmed prior to the assignment.

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).

AND function, series connection

LAD
Series connection of contacts



FBD
AND box

STL
AND function

```
U  Binary tag 1
U  Binary tag 2
=  Binary tag 3
```

SCL
AND function

```
Binary tag 3 := Binary tag 1 & Binary tag 2;
Binary tag 3 := Binary tag 1 AND Binary tag 2;
```

Tags	Signal state			
Binary tag 1	"0"	"1"	"0"	**"1"**
Binary tag 2	"0"	"0"	"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

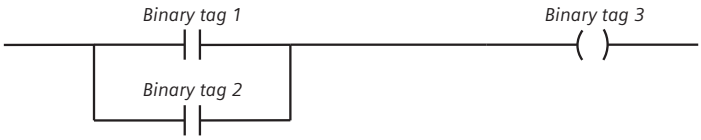
LAD implements the AND function using a series connection of contacts. FBD uses the AND box with two or more inputs. In STL, the AND function is represented by the AND (A) operation. SCL uses the logic operator AND or &.

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).

OR function, parallel connection

LAD
Parallel connection of contacts



FBD
OR box

STL
OR function

```
O  Binary tag 1
O  Binary tag 2
=  Binary tag 3
```

SCL
OR function

```
Binary tag 3 := Binary tag 1 OR Binary tag 2;
```

Tags	Signal state			
Binary tag 1	**"0"**	"1"	"0"	"1"
Binary tag 2	**"0"**	"0"	"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

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.

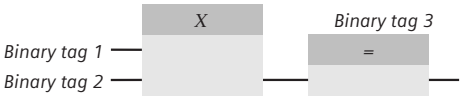
12.1.5 Exclusive OR function, non-equivalence function

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 spe-

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



STL
Exclusive OR function

```
X  Binary tag 1
X  Binary tag 2
=  Binary tag 3
```

SCL
XOR function

```
Binary tag 3 := Binary tag 1 XOR Binary tag 2;
```

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

cial 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).

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”).

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.

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.

Negation of result of logic operation

LAD
NOT-contact

FBD
Negation symbol

STL
NOT statement

```
... (RLO1)
NOT
... (RLO2)
```

SCL
NOT operator

```
... (RLO1) NOT (RLO2)
```

Tags	Signal state	
RLO1	"0"	"1"
RLO2	"1"	"0"

The negation reverses the result of the logic operation:
Signal state "1" become signal state "0", and "0" becomes "1".

Fig. 12.6 Representation and principle of operation of the negation

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
- Dominant setting and resetting
- Edge evaluations

The memory functions can be used together with all binary tags. There are limitations when using temporary local data bits as edge trigger flags.

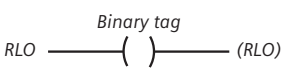
A result of logic operation can be used to influence several memory functions simultaneously. The result of logic operation does not change during execution of a memory function.

12.2.2 Standard 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” (Fig. 12.7).

Standard coil, assignment

LAD
Standard coil



FBD
Assign-box

STL
Assignment

```
...          //(RLO)
=  Binary tag
```

SCL
Assignment

```
Binary tag:= (... RLO ...);
```

RLO, tags	Signal state	
Result of logic operation	"0"	"1"
Binary tag	"0"	"1"

The standard coil or the assignment transfers the result of the logic operation to the binary tag.

The result of the logic operation is not changed by the assignment.

Fig. 12.7 Representation and principle of operation of the assignment

With LAD, the assignment is represented by the standard coil, with FBD by the assign box. With STL, the “=” operation stands for the assignment and with SCL the assignment operator “:=”.

With the MCR dependency switched on, the binary operand present with the standard coil or assignment is set to signal state “0”.

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 do not influence the result of logic operation.

With the MCR dependency switched on, the binary operand present with the single set or reset is no longer influenced (the signal state is “frozen”).

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. With STL, an “S” stands for setting a binary operand, and an

Single setting and resetting

LAD
S-coil, R-coil

FBD
S-box, R-box

STL
Set-operation, reset-operation

```
...          //(RLO)
S  Binary tag 1
```

```
...          //(RLO)
R  Binary tag 2
```

SCL
Emulation with the IF-statement

```
IF (RLO) THEN
   Binary tag 1 := TRUE;
END_IF;
```

```
IF (RLO) THEN
   Binary tag 2 := FALSE;
END_IF;
```

RLO, tags	Signal state	
Result of logic operation	"0"	**"1"**
Binary tag 1 (S)	Unchanged	**"1"**
Binary tag 2 (R)	Unchanged	**"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 the logic operation is not changed by the set or reset.

Fig. 12.8 Representation and principle of operation of the set and reset functions

“R” for resetting. 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.

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 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.

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. 12.9).

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.

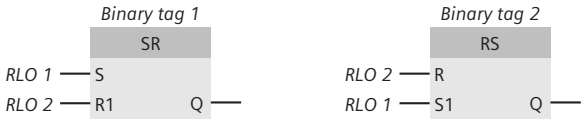
With the MCR dependency switched on, the binary operand at the RS box or SR box is no longer influenced (the signal state is “frozen”).

With STL, the memory function is implemented by single setting and resetting. The sequence of statements defines whether setting or resetting is dominant. With SCL, the memory function can be emulated, for example, by an assignment together with an IF statement.

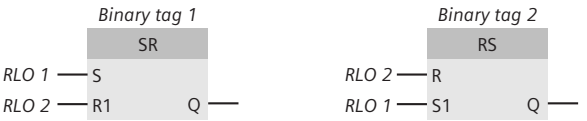
Note that the binary tag used for a memory function may be reset during startup by the CPU's operating system. In certain cases, the signal state of a memory box is retained: This depends on the operand area used (e.g. static local data) and on the settings in the CPU (e.g. retentive behavior).

Dominant setting and resetting

LAD
SR-box, RS-box



FBD
SR-box, RS-box



STL
Emulation by the sequence of set and reset statements

```
...        //(RLO1)
S  Binary tag 1

...        //(RLO2)
R  Binary tag 1
```

```
...        //(RLO2)
R  Binary tag 2

...        //(RLO1)
S  Binary tag 2
```

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;
```

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. 12.9 Dominant setting and resetting, memory boxes

12.2.5 Edge evaluation

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 CPU compares the current result of logic operation, e.g. the result of scan of an input, with a saved result of logic operation. If the two signal states are different, a signal edge is present (Fig. 12.10).

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 CPU processed the edge evaluation last. If a signal edge is now present, i.e. if the current RLO is different from the signal state of the edge trigger flag, the CPU 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 CPU does not detect an edge any more.

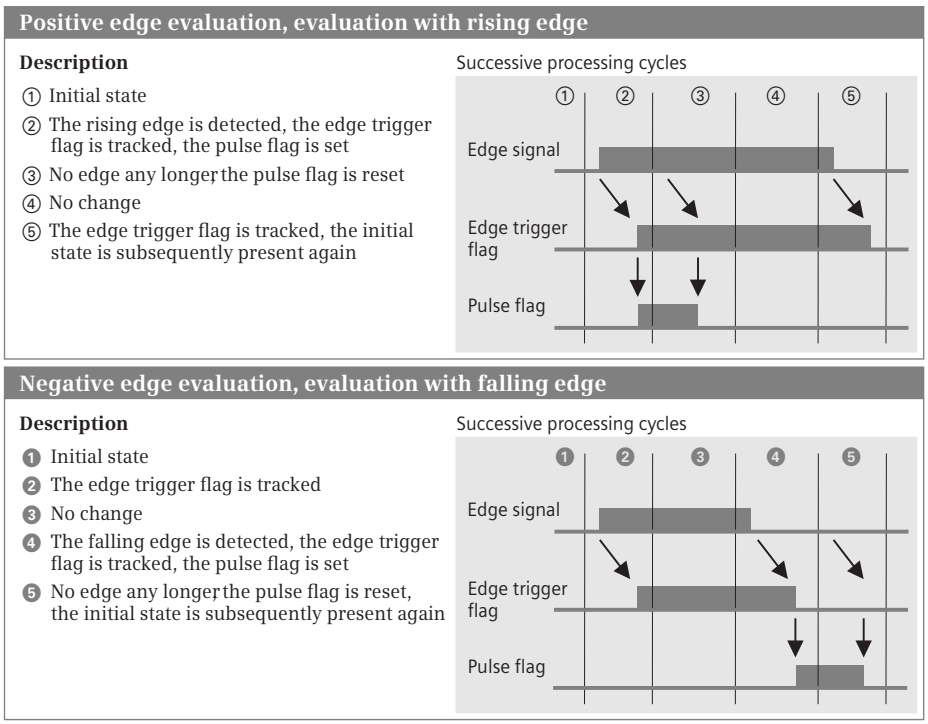


Fig. 12.10 Principle of operation of an edge evaluation in successive cycles

A detected edge is indicated by the RLO following edge evaluation. If the CPU 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 CPU does not detect an edge during the next processing of the edge evaluation (if the “input RLO” of the edge evaluation does not change), it 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 edge signal then changes its state from “0” to “1”. The signal state of the edge trigger flag is initially still “0” so that a rising edge is detected and the pulse flag is set to “1”. The edge trigger flag is updated to signal state “1”. ③ The next processing cycle does not show a change in the signal state or edge signal (comparison with signal state of edge trigger flag). The pulse flag is reset to “0”. ④ No changes take place in the next processing cycles. ⑤ If the edge signal is reset to state “0” again, the edge trigger flag is also updated and the initial state is reached. The pulse flag was set to “1” for one processing cycle only.

Principle of operation of negative edge: ❶ In the initial state, the edge signal, the edge trigger flag, and the pulse flag have signal state “0”. ❷ The edge 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 edge 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.

Edge evaluation of result of logic operation

This edge evaluation generates a pulse when the result of logic operation changes (with ladder logic: change in “current flow”). The P_TRIG box is available for evaluation of a positive edge and the N_TRIG box for evaluation of a negative edge.

Edge evaluation of result of logic operation

The **P_TRIG box** detects a positive edge at the CLK input and then sets the Q output to the signal state "1" for the duration of one cycle (in processing cycle ② in the table).

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).

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 the signal state "1" for the duration of one cycle (in processing cycle ❸ in the table).

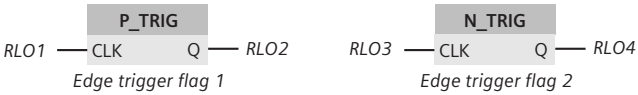
The **FN statement** detects a negative edge of the result of the logic operation prior to the statement and then sets the result of the logic operation following the statement to the signal state "1" for the duration of one cycle (in processing cycle ❸ in the table).

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



STL
FP statement
FN statement

```
...          //RLO1
FP    Edge trigger flag 1
...          //RLO2
```

```
...          //RLO3
FN    Edge trigger flag 2
...          //RLO4
```

SCL
Emulation with
the IF statement

```
//positive edge
IF (RLO1) AND NOT edge trigger flag1
   THEN (* statements *);
     // Program section is executed if there is a pos. edge
     // corresponds to (RLO2)
END_IF;

//Update of edge trigger flag
Edge trigger flag 1 := (RLO1);
```

```
//negative edge
IF NOT (RLO3) AND edge trigger flag 2
   THEN (* statements *);
     // Program section is executed if there is a neg. edge
  // corresponds to (RLO4)
END_IF;

//Update of edge trigger flag
Edge trigger flag 2 := (RLO3);
```

Fig. 12.11 Edge evaluation of result of logic operation (of the “current flow”)

Edge evaluation of a binary tag

The **P contact** or the **P box** detects a positive edge of the binary tag above it and then sets the RLO to the signal state "1" for the duration of one cycle (in processing cycle ② in the table).

The P contact links the RLO in front of it to the pulse flag according to AND (series connection).

The FP statement is used in STL (see above).

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 the signal state "1" for the duration of one cycle (in processing cycle ❸ in the table).

The N contact links the RLO in front of it to the pulse flag according to AND (series connection).

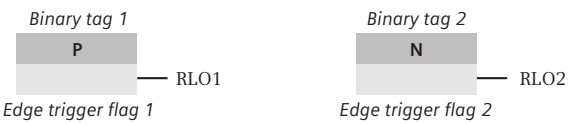
The FN statement is used in STL (see above).

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



STL
FP statement
FN statement

```
U     Binary tag 1
FP    Edge trigger flag 1
...          //= RLO1
```

```
U     Binary tag 2
FN    Edge trigger flag 2
...          //= RLO2
```

SCL
Emulation with pulse flag

```
//positive edge

//Modify pulse flag
Pulse flag 1 := binary tag 1 AND NOT edge trigger flag 1;

//Update edge trigger flag
Edge trigger flag 1 :=  binary tag 1;

//RLO1 is mapped in pulse flag 1
```

```
//negative edge

//Modify pulse flag
Pulse flag 2 := NOT binary tag 2 AND edge trigger flag 2;

//Update edge trigger flag
Edge trigger flag 2 :=  binary tag 2;

//RLO2 is mapped in pulse flag 2
```

Fig. 12.12 Edge evaluation of a binary tag

The positive edge evaluation generates the pulse with a change in signal state from “0” to “1” (rising edge) at the CLK input and the negative edge evaluation with a change in signal state from “1” to “0” (falling edge). The pulse is available at the Q output of the edge evaluation.

Fig. 12.11 shows the representation and signal states of edge evaluation. The principle of operation of edge evaluation is described in detail in Fig. 12.10: The edge signal corresponds to the result of logic operation present at the CLK input, the edge trigger flag corresponds to the binary tag named underneath the function, and the pulse flag corresponds to the result of logic operation following the edge evaluation.

Edge evaluation of a binary tag

The edge evaluation of a binary tag is represented in the ladder logic as a contact, above which the scanned binary tag and below which the edge trigger flag are named. The pulse of the edge evaluation (quasi the signal state of the pulse flag) is connected in series with the result of logic operation of the preceding logic operation. A positive, rising edge is detected by the P contact and a negative, falling edge by the N contact.

In the function block diagram, the binary tag is named above the edge evaluation box and the edge 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 by the N box. Fig. 12.12 shows the representation and signal states of edge evaluation. The principle of operation of edge evaluation is described in detail in Fig. 12.10.

12.3 SIMATIC timer functions

12.3.1 Overview

You can use the SIMATIC 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 responses are available for a SIMATIC timer function:

- Pulse generation
- Extended pulse
- ON delay
- Retentive ON delay
- OFF delay

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. 12.13). 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 STL, the individual statements must be programmed in the indicated sequence. With SCL, the complete function call corresponds to the overall function.

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 timer function. If the enabling statement is used with STL, it must be programmed prior to the start statement (Fig. 12.14).

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.

12.3.2 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.

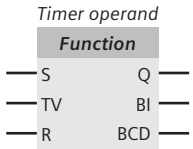
Specification of duration

When starting, the timer function is loaded with a default value of data type S5-TIME. In the figures this default value is named *Duration*. The duration can be specified as a constant or as a tag.

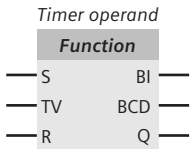
The duration is calculated internally from the time value and time scale: Duration = Time value × Time scale. The duration is the time during which a timer function is active (“time running”). The time value represents the number of time periods for which the timer function runs. The time scale specifies the interval at which the CPU's operating system changes the time value (Fig. 12.15).

SIMATIC timer functions, representation as overall 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_BI
BCD	OUTPUT	S5TIME	Time value_BCD

SCL

```
Time value_BCD := Function
      T_NO     := Timer operand,
      S        := Start input,
      TV       := Duration,
      R        := Reset input,
      Q        := Timer status,
      BI       := Time value_BI);
```

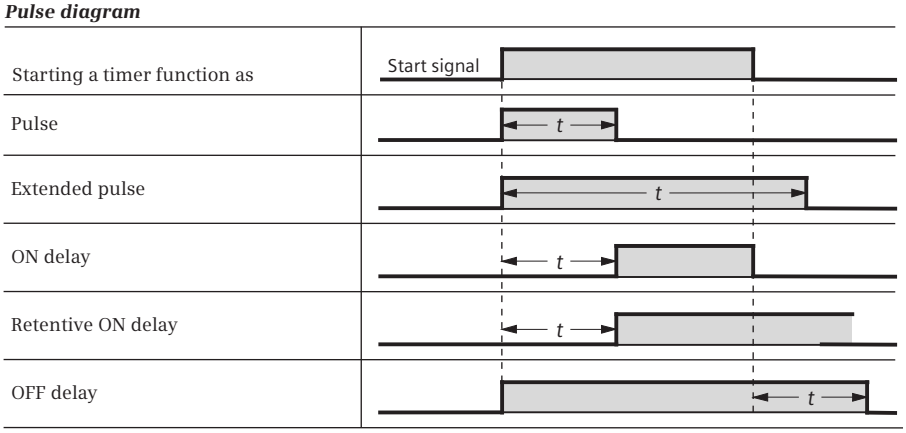
STL

```
A   Enable input
FR  Timer operand

A   Start input
L   Duration
FKT Timer operand

A   Reset input
R   Timer operand

L   Timer operand
T   Time value_BI

LC  Timer operand
T   Time value_BCD

A   Timer operand
=   Timer status
```

Function designation

Start of timer function as	With LAD and FBD as Box	With LAD and FBD as 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



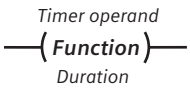
t = set duration

Fig. 12.13 SIMATIC timer functions as overall function

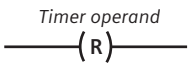
Representation of a SIMATIC timer function with single elements

LAD

Start timer function



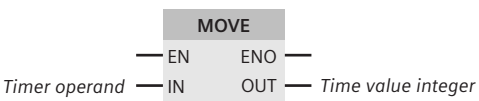
Reset timer function



Scan timer status for "1"

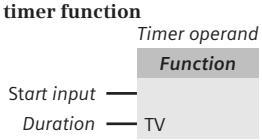
Scan timer status for "0"

Load time value integer-coded

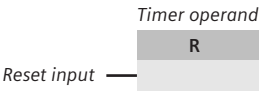


FBD

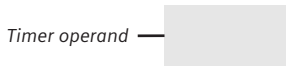
Start timer function



Reset timer function

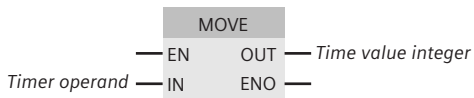


Scan timer status for "1"



Scan timer status for "0"

Load time value integer-coded



STL

```
A    Enable input
FR   Timer operand
A    Start input
L    Duration
FKT  Timer operand
A    Reset input
R    Timer operand
L    Timer operand
T    Time value_BI
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.

SCL

```
Time value_BCD  := Function(
       T_NO     := Timer operand,
       S        := Start input,
       TV       := Duration,
       R        := Reset input,
       BI       => Time value_BI,
       Q        => Time 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 be handled in pairs.

Fig. 12.14 SIMATIC timer functions, representation as single elements

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.

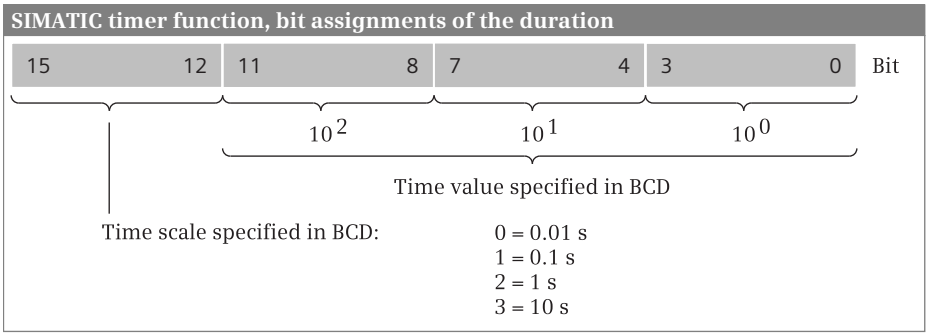


Fig. 12.15 Bit assignment of the duration with a SIMATIC timer function

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. 12.13 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.

12.3.3 Timer response as pulse

Starting a pulse time

The diagram in Fig. 12.16 describes the response of the timer function following starting as pulse and when resetting.

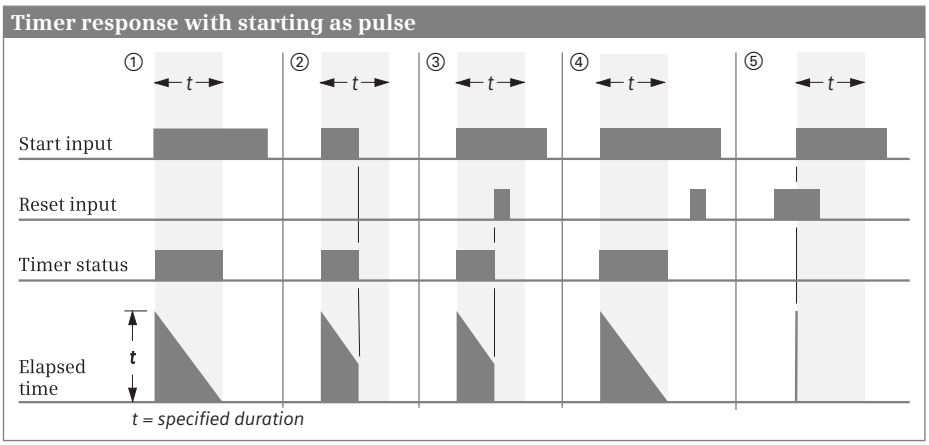


Fig. 12.16 Timer response with starting and resetting as pulse

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.

Resetting a pulse time

Resetting a pulse time has a static effect and has priority over starting of the timer function (Fig. 12.16).

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. 12.17 shows enabling of a timer function started as a pulse.

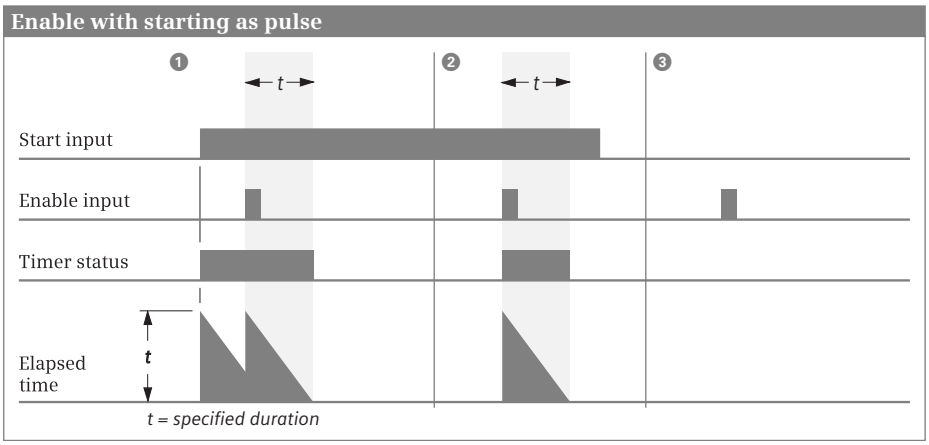


Fig. 12.17 Enabling with a pulse time

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.

12.3.4 Timer response as extended pulse

Starting as extended pulse

The diagram in Fig. 12.18 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.

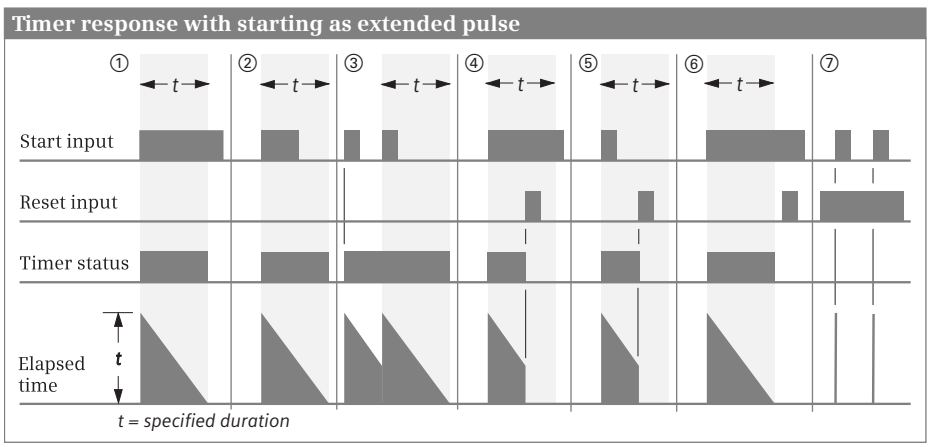


Fig. 12.18 Timer response as extended pulse

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. 12.18).

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.

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. 12.19 shows enabling of a timer function started as a extended pulse.

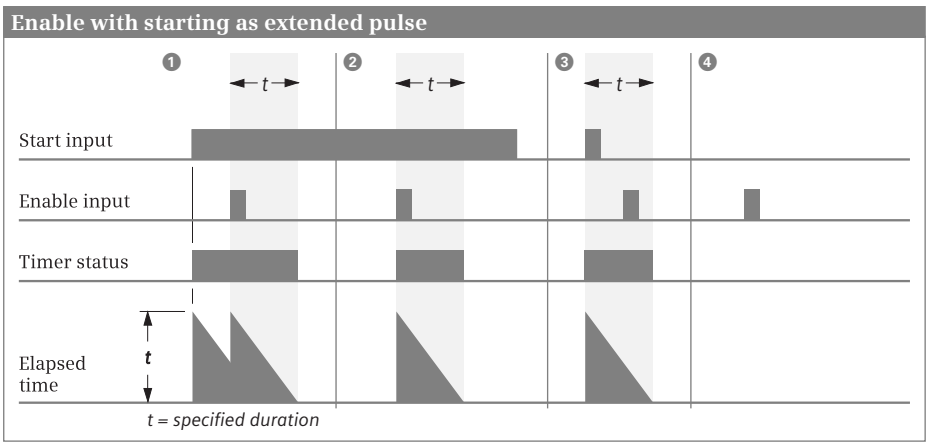


Fig. 12.19 Enabling with extended pulse

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.

12.3.5 Timer response as ON delay

Starting as ON delay

The diagram in Fig. 12.20 describes the response of the timer function following starting as ON delay and when resetting.

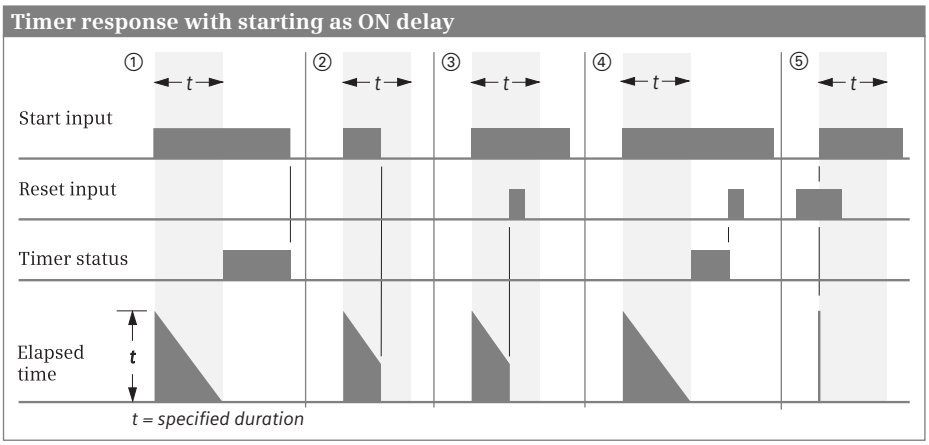


Fig. 12.20 Timer response as ON delay

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.

Resetting as ON delay

Resetting an ON delay has a static effect and has priority over starting of the timer function (Fig. 12.20).

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. 12.21 shows enabling of a timer function as ON delay.

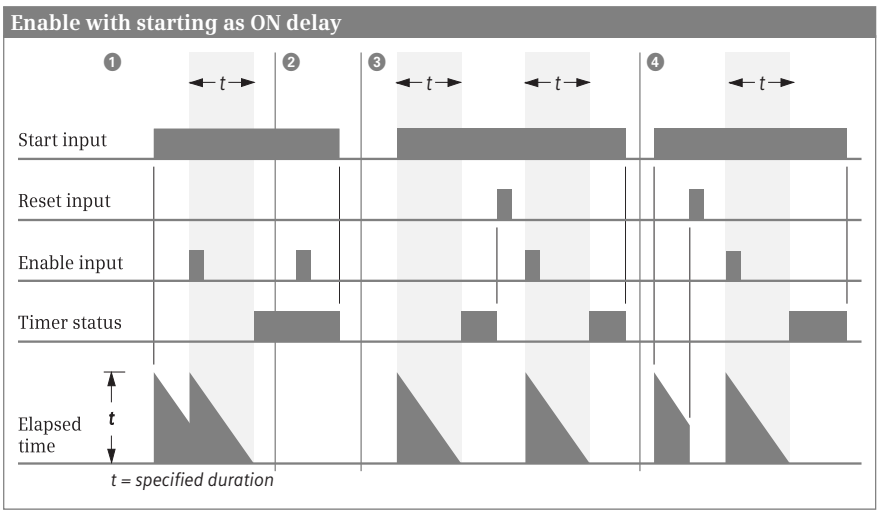


Fig. 12.21 Enabling with ON delay

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.

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.

12.3.6 Timer response as retentive ON delay

Starting as retentive ON delay

The diagram in Fig. 12.22 describes the response of the timer function following starting and when resetting.

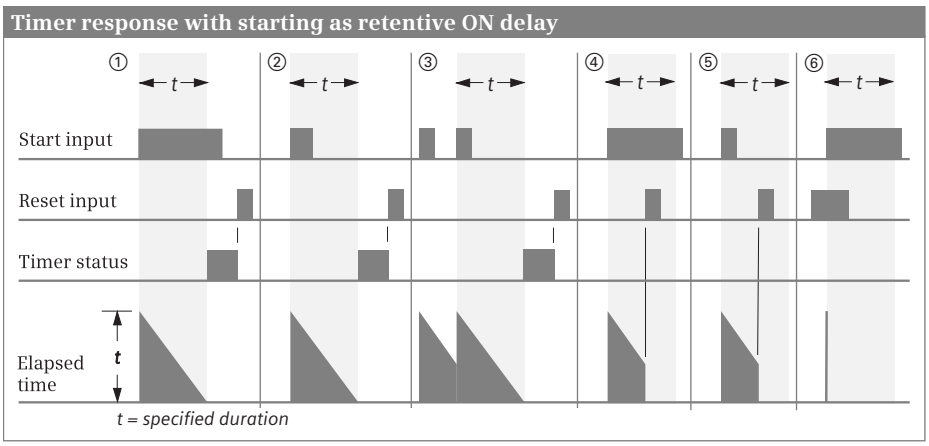


Fig. 12.22 Timer response as retentive ON delay

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.

Resetting as retentive ON delay

Resetting a retentive ON delay has a static effect and has priority over starting of the timer function (Fig. 12.22).

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. 12.23 shows enabling of a timer function started as a retentive ON delay.

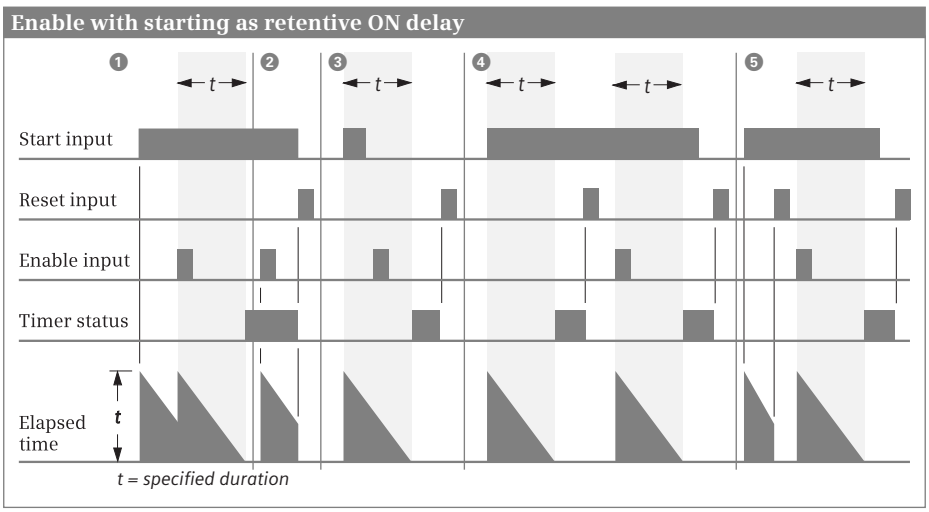


Fig. 12.23 Enabling with retentive ON delay

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.

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.

12.3.7 Timer response as OFF delay

Starting as OFF delay

The diagram in Fig. 12.24 describes the response of the timer function following starting as OFF delay and when resetting.

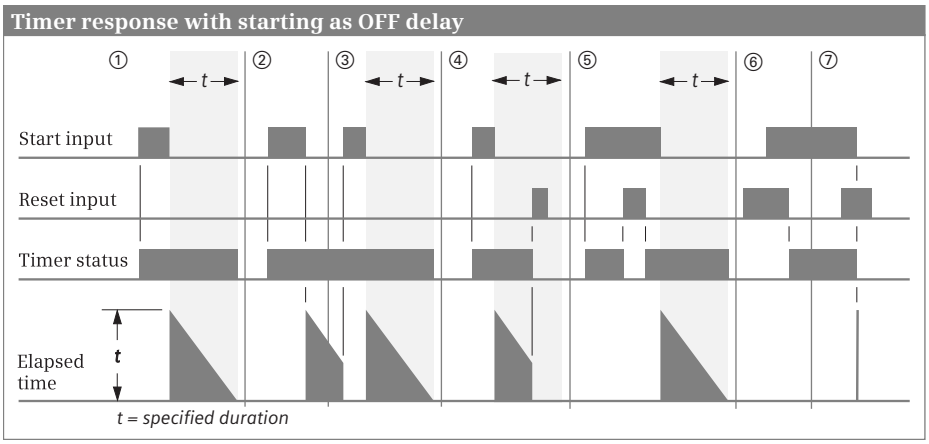


Fig. 12.24 Timer response as OFF delay

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. 12.24).

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.

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. 12.25 shows enabling of a timer function started as an OFF delay.

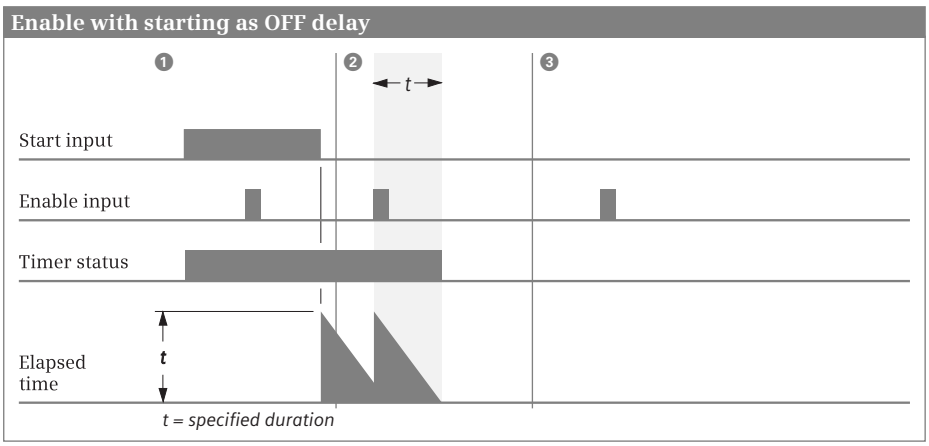


Fig. 12.25 Enabling with OFF delay

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.

12.4 IEC timer functions

12.4.1 Introduction

The timer functions described below are referred to as “IEC timer functions” in order to distinguish them from the “SIMATIC timer functions” additionally present with SIMATIC S7-300/400.

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 (SFB 3)
- TON ON delay (SFB 4)
- TOF OFF delay (SFB 5)

With a CPU 300, an IEC timer function is implemented as a system function block (SFB). 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.

When calling a 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.

The timer functions run in STARTUP and RUN modes.

12.4.2 Pulse generation TP

The pulse generation shortens or extends an input signal to the programmed duration (Fig. 12.26).

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.

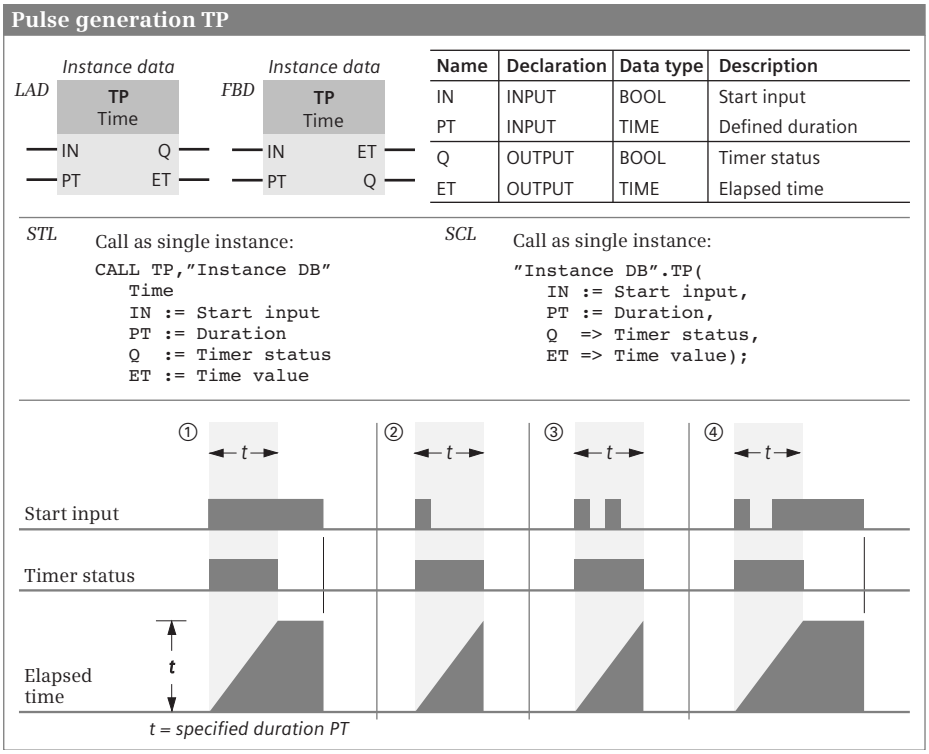


Fig. 12.26 Pulse generation TP, representation and function

12.4.3 ON delay TON

The ON delay delays an input signal by the programmed duration (Fig. 12.27).

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.

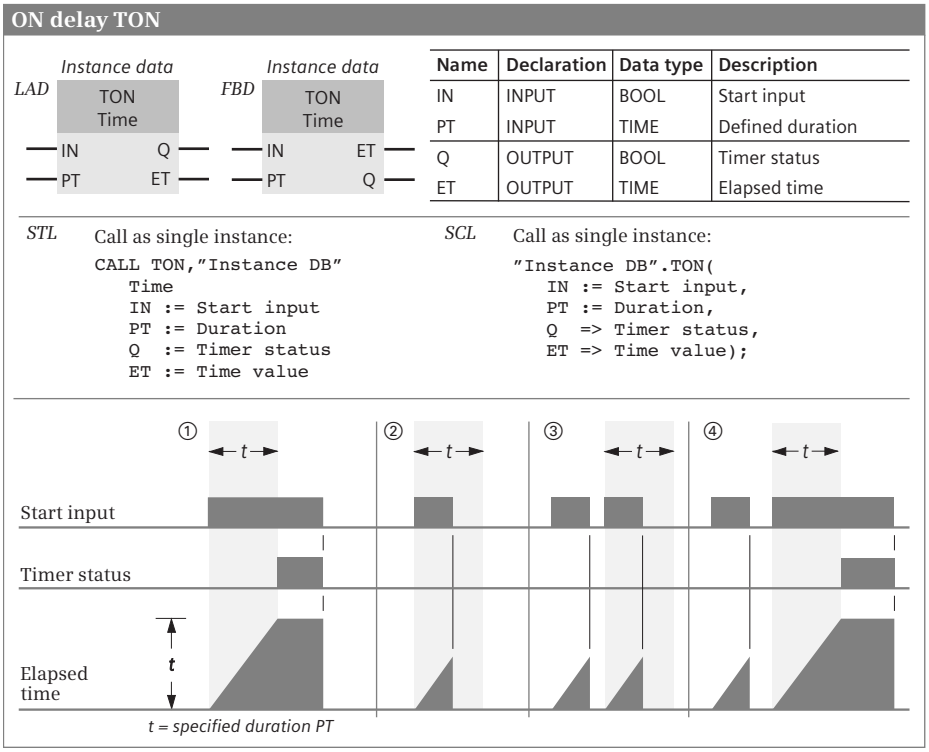


Fig. 12.27 ON delay TON, representation and function

12.4.4 OFF delay TOF

The OFF delay delays the switching off of an input signal by the programmed duration (Fig. 12.28).

① ③ 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.

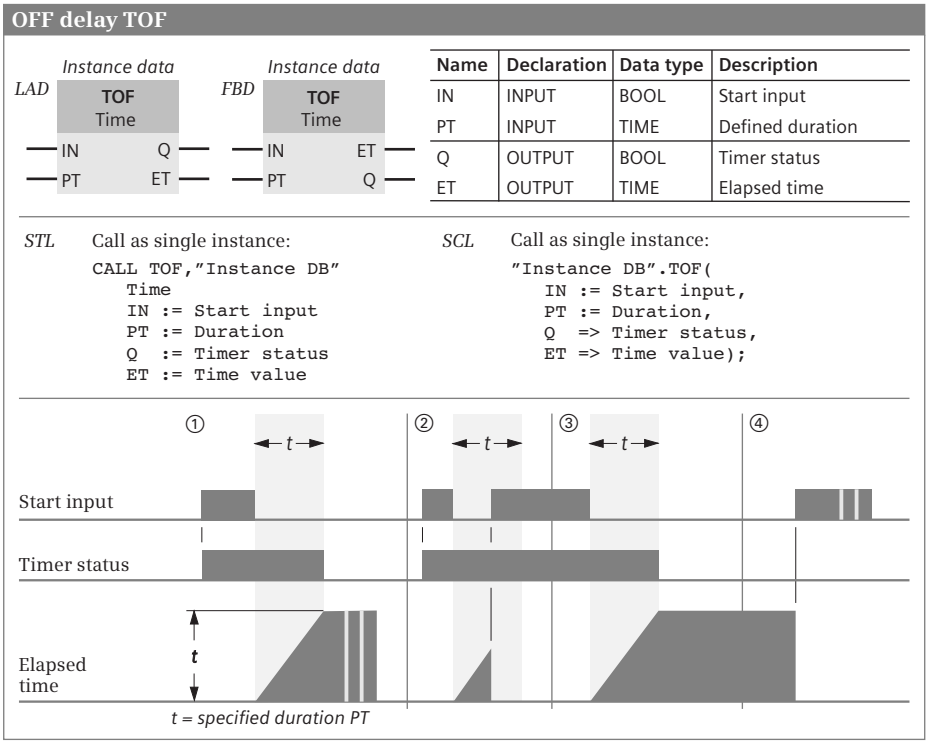


Fig. 12.28 OFF delay TOF, representation and function

12.5 SIMATIC counter functions

12.5.1 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
- Down counter
- Up/down counter

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.

Note: The integrated functions of the compact CPUs with S7-300 (CPU 3xxC) also contain counter functions which can count at up to 10 or 30 kHz or also 60 kHz via a special counter input.

SIMATIC counters as overall function

The overall function is represented in the programming languages LAD and FBD as a box (Fig. 12.29). 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 STL, the individual statements must be programmed in the indicated sequence. With SCL, the complete function call corresponds to the overall function.

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 statements, 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. 12.30).

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.

12.5.2 Programming a counter function

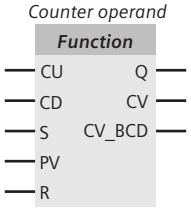
Count up

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.

SIMATIC counter functions, representation as total function

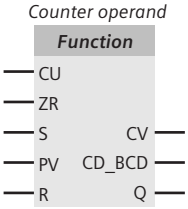
LAD *Counter operand*



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

FBD *Counter operand*



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 Box	With LAD and FBD as Single element	With SCL with	With STL 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

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.

SCL

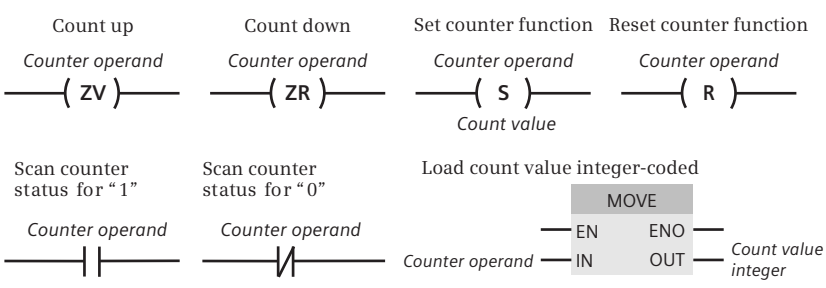
```
Count value BCD := Function(
      C_NO      := Counter operand,
      CU        := Count up,
      CD        := Count down,
      S         := Set input,
      PV        := Count value,
      R         := Reset input,
      Q         => Counter status,
      CV        => Count value integer);
```

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.

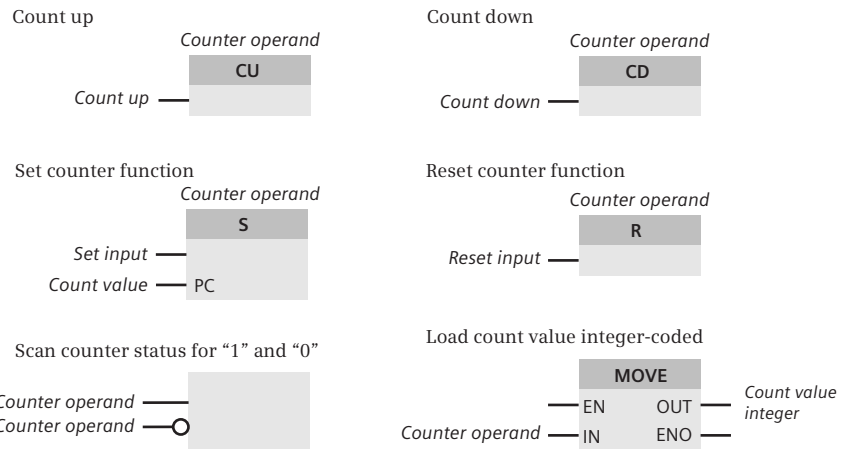
Fig. 12.29 SIMATIC counter functions as overall function

Representation of a SIMATIC counter function with single elements

LAD



FBD



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.

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 which you require.

You must always specify the counter operand (C_NO).

Set input (S) and count value (PV) must be handled in pairs.

Fig. 12.30 SIMATIC counter functions, representation as single elements

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. 12.31 shows the bit assignment of the count value.

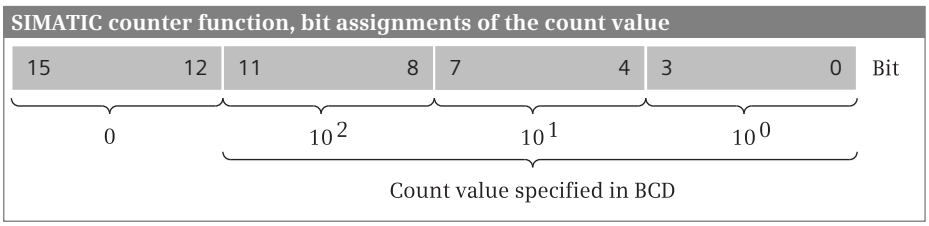


Fig. 12.31 Bit assignment of the count value of a SIMATIC counter

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.

12.5.3 Principle of operation of a counter function

Fig. 12.32 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.

❾ 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”.

❿ Counting down with a count value of 0 has no effect.

⓫ With a positive edge at the set input, the count value is set to 4. The counter status has signal state “1”.

⓬ 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. 12.30 on Page 465.

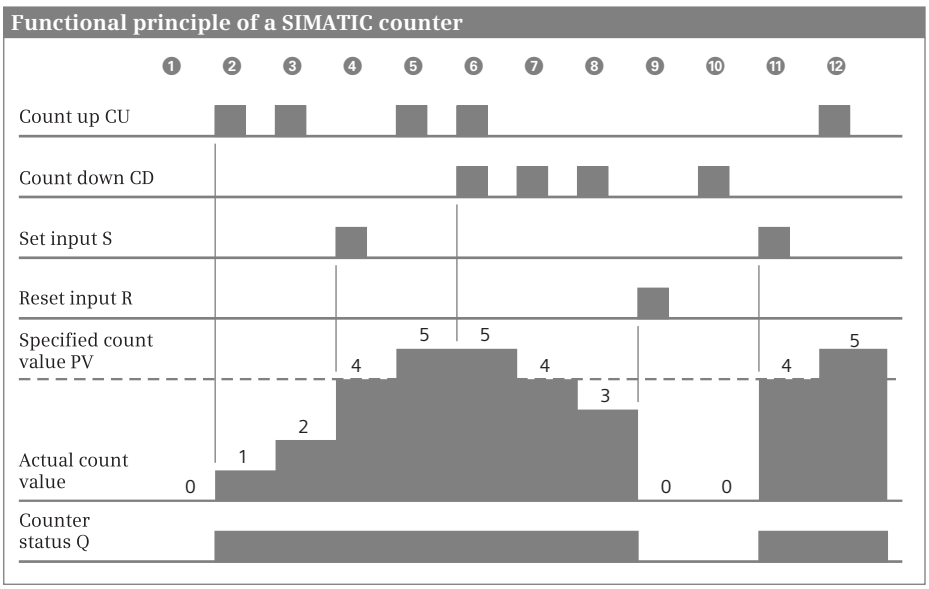


Fig. 12.32 Principle of operation of a SIMATIC counter function

12.5.4 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. 12.33):

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. 12.30 on Page 465.

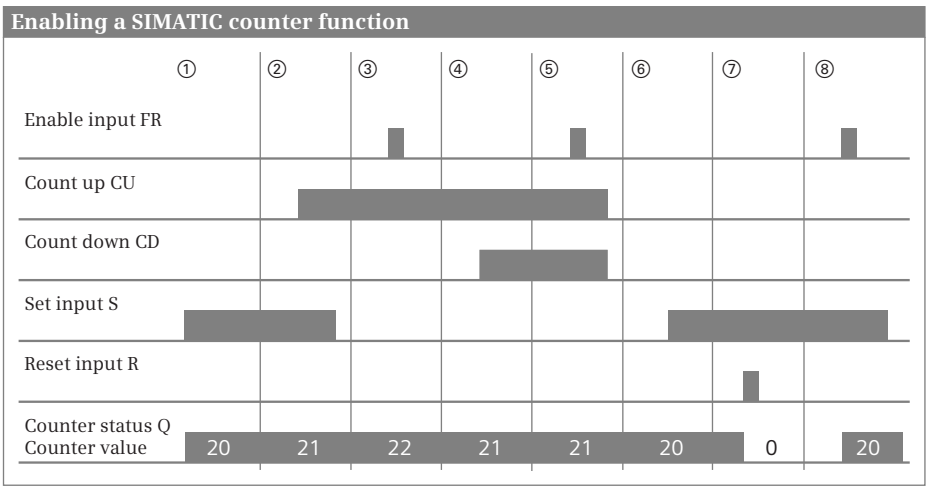


Fig. 12.33 Enabling a SIMATIC counter function

12.6 IEC counter functions

12.6.1 Introduction

The counter functions described below are referred to as “IEC counter functions” in order to distinguish them from the “SIMATIC counter functions” additionally present with SIMATIC S7-300/400.

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 (SFB 0)
- CTD Down counter (SFB 1)
- CTUD Up/down counter (SFB 2)

With a CPU 300, an IEC counter function is implemented as a system function block (SFB). 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.

When calling a counter function you must supply a start input and the defined count value PV (preset value) with tags. Supplying of the counter status Q and the current count value CV is optional.

The counter functions run in STARTUP and RUN modes.

The counting range corresponds to the INT data format and ranges from -32 768 to 32 767.

12.6.2 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. 12.34).

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”.

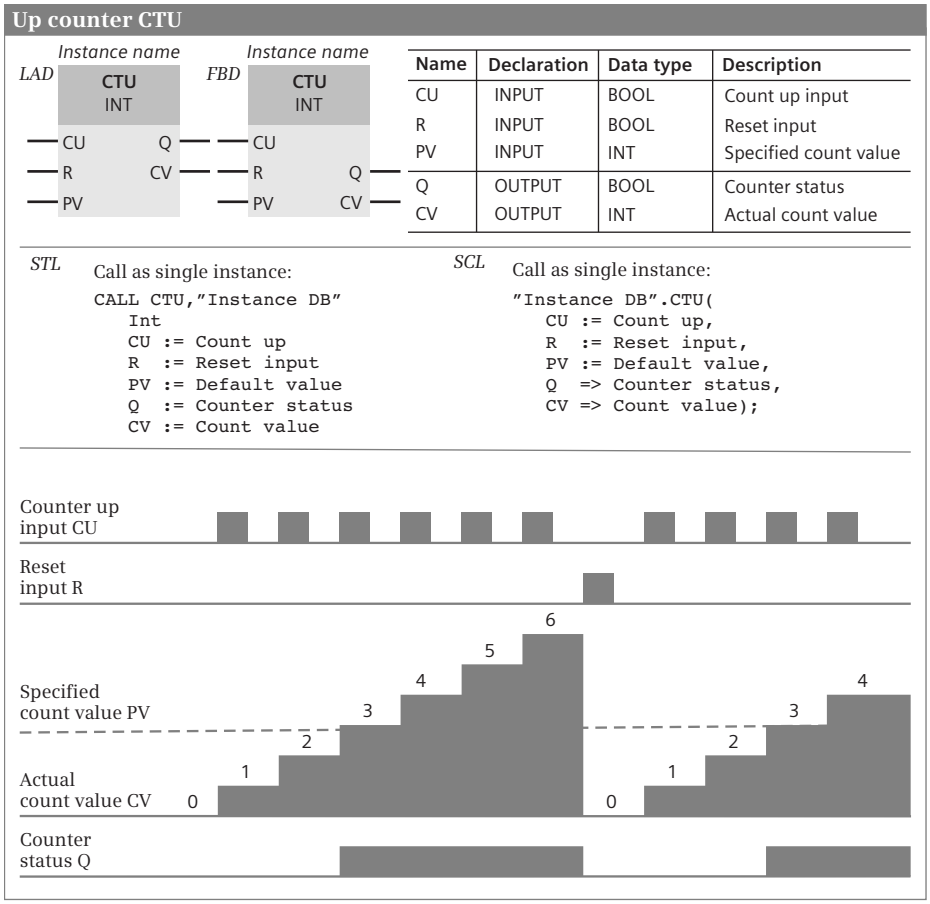


Fig. 12.34 Up counter CTU, representation and function

The Q output has signal state "1" if the current count value is greater than or equal to the specified count value (CV ≥ PV).

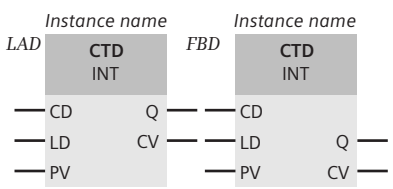
12.6.3 Down counter CTD

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. 12.35).

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).

Down counter CTD



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

STL

Call as single instance:

```
CALL CTD,"Instance DB"
   Int
   CD := Count down
   LD := Load input
   PV := Default value
   Q  := Counter status
   CV := Count value
```

SCL

Call as single instance:

```
"Instance DB".CTD(
   CD := Count down,
   LD := Load input,
   PV := Default value,
   Q  => Counter status,
   CV => Count value);
```

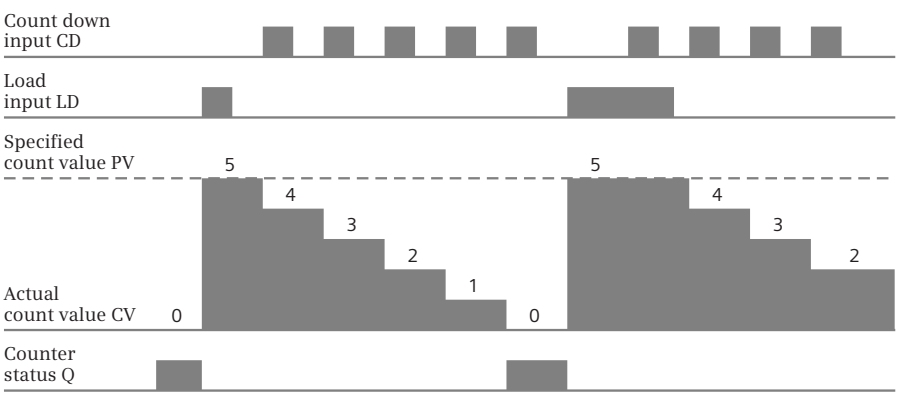


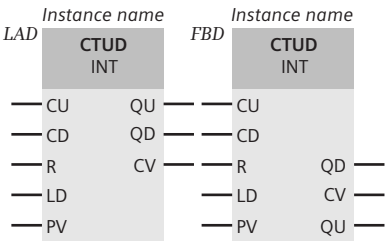
Fig. 12.35 Down counter CTD, representation and function

12.6.4 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. 12.36).

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.

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	Status down status
CV	OUTPUT	INT	Actual count value

STL Call as 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
```

SCL Call as 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);
```

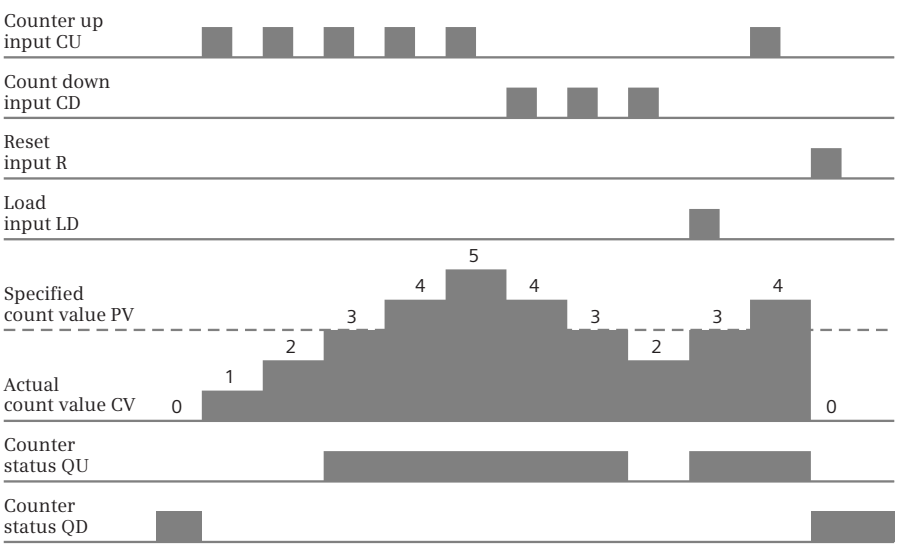


Fig. 12.36 Up/down counter CTUD, representation and function

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 program elements catalog under *Basic instructions* and *Extended instructions*.

The Chapters 7 “Ladder logic LAD” on page 249, 8 “Function block diagram FBD” on page 282, 9 “Statement list STL” on page 315, and 10 “Structured Control Language SCL” on page 363 describe how you can program the functions using the individual programming languages and what special features exist.

The following digital functions are available with a CPU 300:

- The transfer functions transfer the value of a (digital) tag or memory area.
- The comparison functions generate a binary result by comparing two tags. Tags with data types INT, DINT, REAL, STRING, and with time data types can be compared.
- The arithmetic functions link two tags with data types INT, DINT, and REAL in accordance with the basic arithmetic operations.
- The math functions convert the value of a tag with data type REAL 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.

The “simple” digital functions are boxes in the case of LAD and FBD (with LAD, the comparison is a contact), operations for linking the contents of accumulators in the case of STL, and arithmetic, logic and comparison expressions in the case of SCL.

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 box in the case of LAD and FBD, with load and transfer functions in the case of STL, and with the value assignment function in the case of SCL
- System blocks for transferring a memory area in the work memory
- System blocks for transferring a memory area between work and load memories
- System blocks for controlling a memory area with MCR dependency

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.

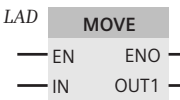
As a result of the different language elements, the transfer function is represented differently in the various programming languages: by the MOVE box in the case of LAD and FBD, by load and transfer functions in the case of STL, and by the value assignment function in the case of SCL.

13.2.2 MOVE box with LAD and FBD

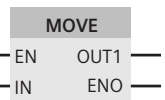
The MOVE box transfers the value at the IN parameter to the tag at the OUT1 parameter (Fig. 13.1). You can use EN to control execution of the MOVE box depending on the result of logic operation. If EN = "1" or not connected, the transfer function is executed and ENO has signal state "1". If EN = "0", ENO is also = "0". The MOVE box does not report any errors.

MOVE box

LAD



FBD



Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
IN	INPUT	Data type *)	Input
OUT1	OUTPUT	Data type *)	First output

*) See text

Function:
The value present in the IN parameter is transferred to the OUT1 parameter.

Data type:
All elementary data types (except BOOL).

Dependencies:
Execution of the MOVE box depends on the MCR function.

Fig. 13.1 Representation and function of the MOVE box with LAD and FBD

Different data types

Tags with different data types can be connected to the input and output of the MOVE box. Conversion of a data type is therefore possible. Table 13.1 shows the combinations with the block attribute *IEC check* deactivated. If *IEC check* is activated, the data types at the input and output must be the same, with the following exceptions: With BYTE at the input, WORD and DWORD are permissible at the output; with WORD at the input, DWORD is permissible at the output.

If the input tag is "smaller" than the output tag, it is transferred right-justified to the output tag and the space to the left is occupied by zero. Example: If input byte %IB4 is transferred to the memory word %MW48, zero is present in memory byte %MB48 and the value of input byte %IB4 is present in memory byte %MB49.

If the input tag is "larger" than the output tag, only the right part of the input tag which fits in the output tag is transferred. Example: If memory word %MW50 is transferred to output byte %QB6, the value of memory byte %MB51 is present in output byte %QB6.

MCR dependency of the MOVE box

The transfer to a memory area depends on the master control relay, and therefore the MOVE box only transfers the input value to the output tag if MCR dependency is switched off. Zero is transferred if MCR dependency is switched on.

Table 13.1 Different data types on the MOVE box with the IEC check block attribute deactivated

IN parameter	OUT1 parameter			
BYTE	BYTE, WORD, DWORD	INT, DINT	TIME, DATE, TOD	CHAR
WORD	BYTE, WORD, DWORD	INT, DINT	TIME, DATE, TOD, S5TIME	CHAR
DWORD	BYTE, WORD, DWORD	INT, DINT, REAL	TIME, DATE, TOD	CHAR
INT	BYTE, WORD, DWORD	INT, DINT	TIME, DATE, TOD	
DINT	BYTE, WORD, DWORD	INT, DINT	TIME, DATE, TOD	
REAL	DWORD	REAL		
TIME	BYTE, WORD, DWORD	INT, DINT	TIME	
S5TIME	WORD		S5TIME	
DATE	BYTE, WORD, DWORD	INT, DINT	DATE	
TOD	BYTE, WORD, DWORD	INT, DINT	TOD	
CHAR	BYTE, WORD, DWORD			CHAR

13.2.3 Loading and transferring with STL

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. 13.2).

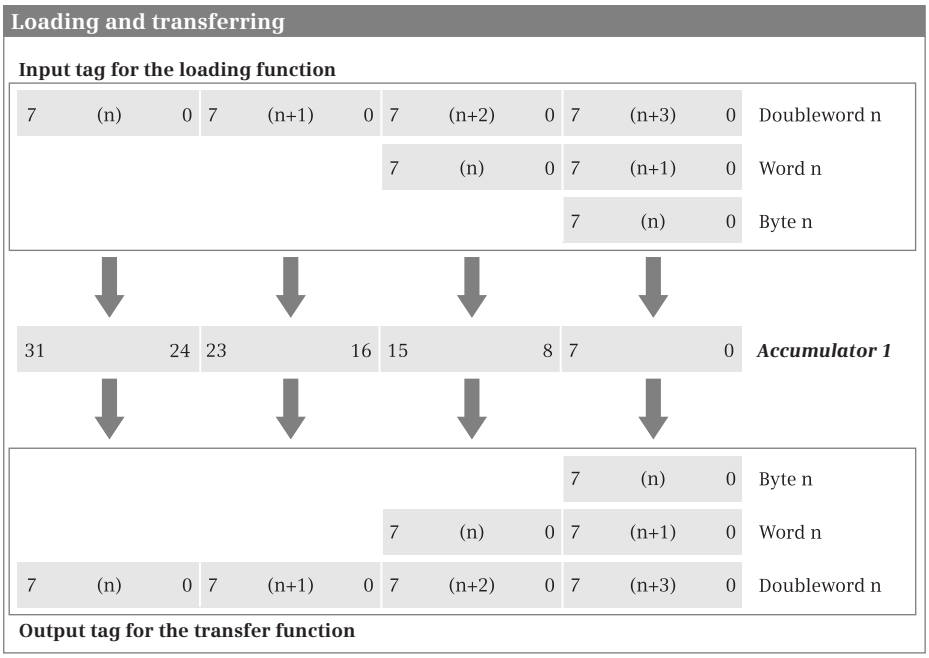


Fig. 13.2 Transfer of tags with a width of 8, 16, or 32 bits

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. In the case of a CPU 300, the previous contents of accumulator 2 are lost in the process (see also Chapter 9.5.1 “Transfer functions” on page 333).

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 9.7.1 “Accumulator functions” on page 358).

MCR dependency of transfer statement

The transfer to a memory area depends on the master control relay and therefore the transfer statement only transfers the output parameter if MCR dependency is switched off. Zero is transferred if MCR dependency is switched on.

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 single tag, a combination of tag values, or a function whose value is assigned to the output parameter.

```
#Output_tag := #Input_tag; //Assignment of tag value
```

The data type of the value assignment is determined by the output tag. The data types on both sides of the assignment operator must be the same. Exception: With the *IEC check* block attribute deactivated, the “implicit data type conversion” is applicable, see Chapter 13.6.1 “Implicit data type conversion” on page 501.

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 DT and STRING tags

Every DT tag can be assigned another DT tag or a DT constant.

Every STRING tag can be assigned another STRING tag or a STRING constant. If the assigned string is longer than the tag present on the left of the assignment opera-

tor, a warning is output during compilation. 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`	Declaration
`Array2  ARRAY [1..16] OF INT`	
`var_int  INT`	
`i  INT`	
`k  INT`	
`#var_int := #Array1[#i,#k];`	Assignment of a component
`#Array2  := #Array1[#i];`	Assignment of a partial array

13.2.5 Copying and filling a data area in the work memory

The following system blocks are available for copying and filling a data area:

- BLKMOV Copy data area (SFC 20)
- UBLKMOV Copy data area without interruption (SFC 81)
- FILL Fill data area (SFC 21)

The representation is made with EN/ENO boxes in the case of LAD and FBD, and as a function call with error information as the function value in the case of STL and SCL. Fig. 13.3 shows the general graphic representation.

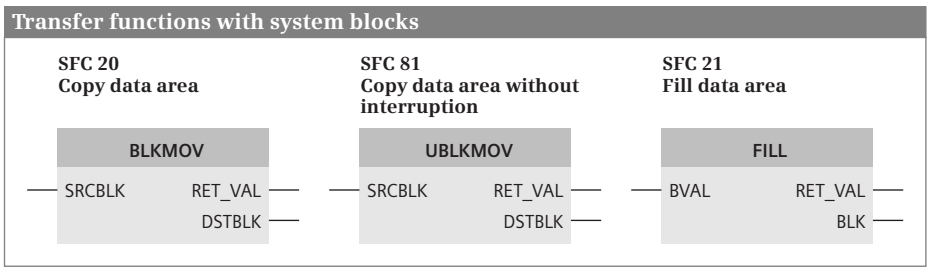


Fig. 13.3 Graphic representation of the transfer functions

Block parameters with data type ANY

The system blocks BLKMOV, UBLKMOV, and FILL each have two parameters with data type ANY. You can basically connect any operand, any tag, or any absolutely addressed area to these parameters.

If you use a tag with complex data type, it can only be a “complete” tag; components of a tag, e.g. individual array or structure components, are not permissible. You can use the ANY pointer for specifying an absolutely addressed area; its structure is described in Chapter 4.6.2 “Pointer” on page 136. 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.

“Variable ANY pointer”

If you connect an actual parameter with data type ANY, which is present in the temporary local data, to a block parameter of data type ANY, the program editor takes on the actual parameter as ANY pointer. In this manner you can set up an ANY pointer in the temporary local data which you can change during runtime, i.e. a variable (dynamic) design of the area is possible. Further details can be found in Chapters 4.6.3 ““Variable” ANY pointer with STL” on page 139 and 4.6.4 ““Variable” ANY pointer with SCL” on page 140.

Copying of STRING tags

If a STRING tag is only present at the SRCBLK 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 and DSTBLK parameters, the two length bytes are also transferred to the destination tag.

BLKMOV Copy data area

BLKMOV copies the contents of a source area at the SRCBLK parameter in the direction of increasing addresses (incrementing) into a destination area at the DSTBLK parameter. With BLKMOV, the copying process can be interrupted by a program of higher priority following each doubleword.

The following actual parameters can be connected to the SRCBLK and DSTBLK parameters:

- Any tags from the data blocks in the work memory and from the following operand areas: inputs I, outputs Q, bit memories M
- Tags from the temporary local data (with special handling for data type ANY)
- Absolutely addressed data areas with specification of an ANY pointer

You cannot use BLKMOV to access the peripheral inputs and outputs, data blocks in the load memory, or the SIMATIC timers and counters operand areas. Exception: If the system block READ_DBL is not present in the CPU, BLKMOV can read from the load memory. In this case the transfer must not be interrupted by a program of higher priority.

The specified area is copied in the case of inputs and outputs independent of the actual assignment with input and output modules.

The source and destination areas must not overlap. If the source and destination areas are of different length, transfer is only performed up to the length of the smaller area. No data transfer takes place if the limits of operand areas are violated and an error message is output.

UBLKMOV Copy data area without interruption

UBLKMOV copies the contents of a source area at the SRCBLK parameter in the direction of increasing addresses (incrementing) into a destination area at the DSTBLK parameter. With UBLKMOV, copying cannot be interrupted and therefore the time required for responding to interrupts may increase under certain circumstances. The maximum number of transmitted bytes is therefore limited to 512.

The following actual parameters can be connected to the SRCBLK and DSTBLK parameters:

- Any tags from the data blocks in the work memory and from the following operand areas: inputs I, outputs Q, bit memories M
- Tags from the temporary local data (with special handling for data type ANY)
- Absolutely addressed data areas with specification of an ANY pointer

You cannot use UBLKMOV to access the peripheral inputs and outputs, data blocks in the load memory, or the SIMATIC timers and counters operand areas.

The specified area is copied in the case of inputs and outputs independent of the actual assignment with input and output modules.

The source and destination areas must not overlap. If the source and destination areas are of different length, transfer is only performed up to the length of the smaller area. No data transfer takes place if the limits of operand areas are violated and an error message is output.

FILL Fill data area

FILL copies a defined value (source area) into a memory area (destination area) as often as necessary until the latter is written completely. The transfer is carried out in the direction of increasing addresses (incrementing).

The following actual parameters can be connected to the BVAL and BLK parameters:

- Any tag from the data blocks in the work memory and from the following operand areas: inputs I, outputs Q, bit memories M
- Absolutely addressed data areas with specification of an ANY pointer
- Tag in the temporary local data of data type ANY (special handling)

You cannot use FILL to access the peripheral inputs and outputs, data blocks in the load memory, or the SIMATIC timers and counters operand areas.

The specified area is filled in the case of inputs and outputs independent of the actual assignment with input and output modules.

The source and destination areas must not overlap. 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. No data transfer takes place if the limits of operand areas are violated and an error message is output.

13.2.6 Transfer data area from and to 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.

One can newly create a data block in the load memory using the program and then transfer data from a data block in the work memory to the data block in the load memory using the WRIT_DBL function, for example for archiving.

The opposite direction is possible using the READ_DBL function. This transfers data from a data block in the load memory – for example recipe data – to a data block in the work memory.

Fig. 13.4 shows the graphic representation of these functions.

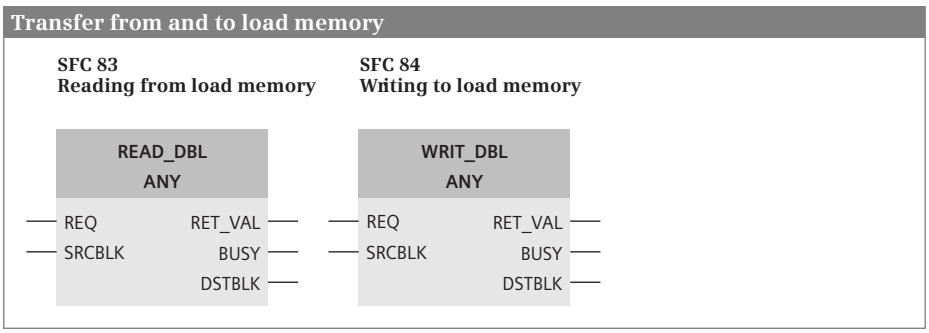


Fig. 13.4 Graphic representation of functions for transferring from and to the load memory

ANY parameter with READ_DBL and WRIT_DBL

Complete data blocks or parts of data blocks are permissible as actual parameters at the SRCBLK and DSTBLK block parameters.

The SRCBLK and DSTBLK parameters can be supplied with:

- Complete data blocks, e.g. %DB100 or “Recipe 1”
- A data area addressed absolutely as ANY pointer, e.g. P#DB100.DBX16.0 BYTE 64. With an ANY pointer of type BOOL, e.g. for an array with binary components, the length must be divisible by 8.
- Tags from the data blocks. With symbolic addressing, only “complete” tags are accepted which are present in a data block; individual array or structure components are not permissible.

Processing of system functions

The READ_DBL and WRIT_DBL functions work asynchronously: They trigger the transfer process by signal state “1” at the REQ parameter. You may only access the read and written data areas again when the parameter BUSY has signal state “0” again. Please also observe the CPU's system resources when using asynchronous system functions.

READ_DBL Reading from load memory

READ_DBL reads data from a data block present in the load memory and writes it into a data block present in the work memory. The contents of the read data block are not changed.

READ_DBL only reads values from the load memory. The start values of the data operands – which may differ from the actual values in the work memory – are present here following loading of the user program.

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.

WRIT_DBL Writing to load memory

WRIT_DBL reads data from a data block in the work memory and writes it into a data block in the load memory. The contents of the read data block are not changed.

WRIT_DBL only reads values from the work memory. The actual values of the data operands are present here which can be changed during program execution.

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.

Please note that the load memory usually only permits a limited number of write operations as a result of the physical design. Too frequent writing, e.g. cyclic, reduces the service life of the load memory.

13.2.7 Control memory area with MCR dependency

The following system and standard blocks are available for setting and resetting a memory area with MCR dependency switched on:

- SETP Set I/O area bit by bit (SFC 79)
- RESETP Reset I/O area bit by bit (SFC 80)
- SET Set process image and memory area bit by bit (FC 83)
- RESET Reset process image and memory area bit by bit (FC 82)
- SETI Set process image and memory area byte by byte (FC 101)
- RESETI Reset process image and memory area byte by byte (FC 100)

The representation is made with EN/ENO boxes in the case of LAD and FBD, and as a function call with or without function value in the case of STL. The function of the master control relay is not implemented with SCL.

Fig. 13.5 shows the general graphic representation of these functions.

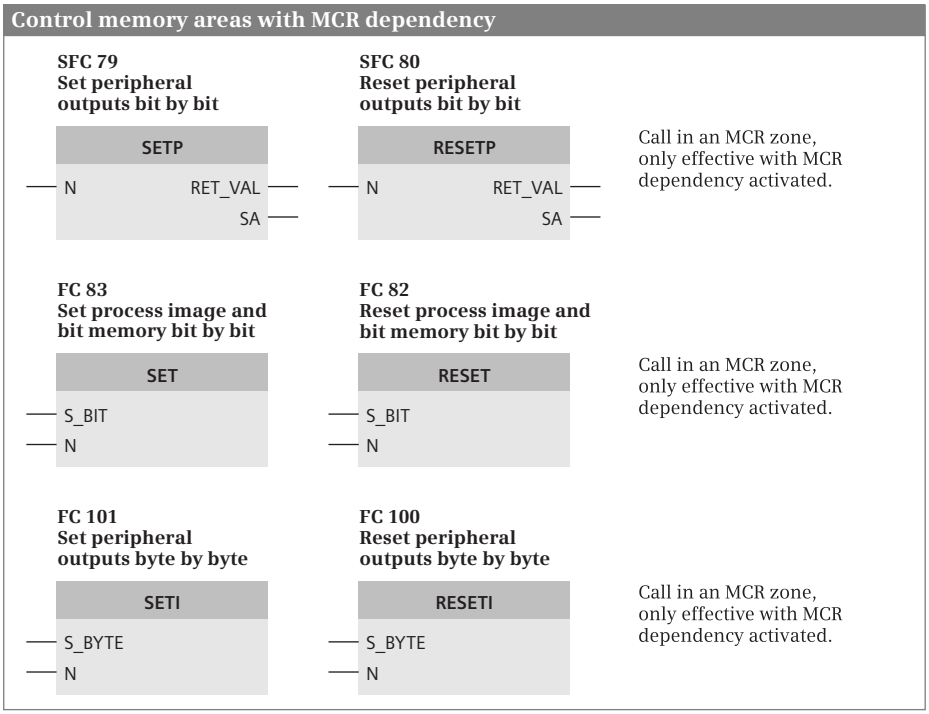


Fig. 13.5 Graphic representation of the set and reset functions with MCR dependency

SETP Set peripheral output area bit by bit
RESETP Reset peripheral output area bit by bit

SETP sets the bits in a continuous area of the peripheral outputs to signal state “1”.

RESETP resets the bits in a continuous area of the peripheral outputs to signal state “0”.

You call SETP and RESETP in an MCR zone. The system blocks only work with MCR dependency switched on; if MCR dependency is switched off, calling of SETP or RESETP has no effect.

Setting and resetting of the I/O bits results in simultaneous updating of the process image output. The I/O is influenced byte by byte. The bits in the first and last bytes which have not been selected by the system blocks are assigned the signal states currently present in the process image output.

You use the N parameter to define the number of bits to be controlled and the SA parameter to define the first bit (data type POINTER). RET_VAL is used by the system blocks to signal any errors.

SET Set process image and memory area bit by bit
RESET Reset process image and memory area bit by bit

SET sets the bits in a continuous memory area or in the process image inputs and outputs to signal state “1”.

RESET resets the bits in a continuous memory area or in the process image inputs and outputs to signal state “0”.

You call SET and RESET in an MCR zone. The system blocks only work with the MCR dependency switched on; if the MCR dependency is switched off, calling of SET or RESET has no effect.

The bits in the first and last bytes which have not been selected by the system blocks retain the current signal states.

You use the N parameter to define the number of bits to be controlled and the S_BIT parameter to define the first bit (data type POINTER with cross-area, register indirect addressing). If the S_BIT parameter does not point to the memory area or process image, the bits are not influenced and the ENO output has signal state “0”.

SETI Set peripheral output bits byte by byte
RESETI Reset peripheral output bits byte by byte

SETI sets the bits in a continuous area of the peripheral outputs to signal state “1”.

RESETI resets the bits in a continuous area of the peripheral outputs to signal state “0”.

You call SETI and RESETI in an MCR zone. The system blocks only work with the MCR dependency switched on; if the MCR dependency is switched off, calling of SETI or RESETI has no effect.

You use the N parameter to define the number of bits to be controlled, which must be a multiple of eight. You use the S_BYTE parameter to define the first byte (data type POINTER with cross-area, register-indirect addressing and bit address = 0). If the S_BYTE parameter does not point to the peripheral output area, if the bit address is not zero, and if the number N is not a multiple of eight, the bits are not influenced and the ENO output has signal state “0”.

The corresponding bits are not updated in the process image output.

13.3 Comparison functions

The following are available for comparing two tags:

- The comparison function implemented using “basic instructions” for comparing numerical values
- The comparison function implemented using system blocks for comparing two tags with data types DT (T_COMP) and STRING (S_COMP)

The “simple” comparison functions are the comparison contact in the case of LAD, the comparison box in the case of FBD, the comparison operation in the case of STL, and the comparison expression in the case of SCL.

13.3.1 Execution of “simple” comparison function

A comparison function compares the values of two digital tags and delivers the binary comparison result “1” or TRUE in the case of “comparison fulfilled” or “0” or FALSE in the case of “comparison not fulfilled”. Table 13.2 shows setting of the comparison result depending on the magnitude of the compared values.

Table 13.2 Result following a comparison function

The relation between the compared values	delivers the following comparison result with					
	==	<>	>	>=	<	<=
Input value 1 > Input value 2 (Accu 2 > Accu 1)	"0"	"1"	"1"	"1"	"0"	"0"
Input value 1 = Input value 2 (Accu 2 = Accu 1)	"1"	"0"	"0"	"1"	"0"	"1"
Input value 1 < Input value 2 (Accu 2 < Accu 1)	"0"	"1"	"0"	"0"	"1"	"1"

Fig. 13.6 shows how the comparison function is implemented in the various programming languages.

With LAD and FBD, the two digital tags to be compared must have the data type set for the comparison function.

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.

With SCL, the comparison is implemented by a comparison expression. The compared tags must belong to the same class of data type, i.e. BYTE, WORD, and DWORD can be compared with each other, as can INT, DINT, and REAL.

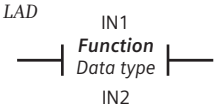
A prerequisite for a fulfilled comparison with floating-point numbers is that they are valid. If an invalid floating-point number is compared, the comparison is never fulfilled. The comparison function is carried out independent of conditions. Table 13.3 shows how a comparison function influences the status bits.

13.3.2 Comparison function T_COMP

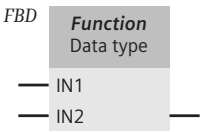
T_COMP compares two tags with data type DATE_AND_TIME (DT). The graphic representation is shown in Fig. 13.7. The comparison function T_COMP is implemented with loadable standard functions (FC), which are represented as an EN/ENO box

“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	Result of comparison

*) The two data types must be the same

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	INT, DINT, REAL, TIME, BYTE, WORD, DWORD
<>	not equal to	INT, DINT, REAL, TIME, BYTE, WORD, DWORD
>	greater than	INT, DINT, REAL, TIME
<	less than	INT, DINT, REAL, TIME
>=	greater than or equal to	INT, DINT, REAL, TIME
<=	less than or equal to	INT, DINT, REAL, TIME

STL

```
L   IN1
L   IN2
Comparison operation
=   Result of comparison
```

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.

SCL

```
Result of comparison := IN1 Comparison function IN2;
```

Comparison function:	
==	equal to
<>	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Approved data types:

Comparison of:	*Comparison with:*
INT, DINT, REAL	INT, DINT, REAL
CHAR, STRING	CHAR, STRING *)
TIME	TIME
DATE	DATE
TIME_OF_DAY	TIME_OF_DAY

In addition with *equal to* and *not equal to*:

BYTE, WORD, DWORD	BYTE, WORD, DWORD

*) With a comparison CHAR and STRING, only one character of the string can be compared:

```
#result := #var_char <= #var_string[#i];
```

Fig. 13.6 Function and representation of “simple” comparison functions

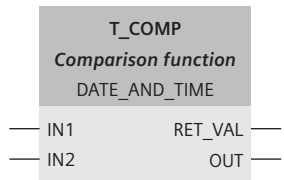
Table 13.3 Status bits with a comparison function

Status bit		BR	CC1	CC0	OV	OS	OR	STA	RLO	/FC
The result is	equal to	–	0	0	0	–	0	x	x	1
	greater than	–	1	0	0	–	0	x	x	1
	less than	–	0	1	0	–	0	x	x	1
Invalid REAL number		–	1	1	1	1	0	x	x	1
The comparison is	fulfilled	–	x	x	0	–	0	1	1	1
	not fulfilled	–	x	x	0	–	0	0	0	1

Comparison function for tags of data type DT

The comparison function T_COMP compares two tags of data type DATE_AND_TIME (DT). T_COMP is implemented using loadable standard functions (FC).

Compare tag of data type DT



Comparison function:

	Comparison for	Block	
EQ	equal to	FC 9	EQ_DT
NE	not equal to	FC 28	NE_DT
GT	greater than	FC 14	GT_DT
GE	greater than or equal to	FC 12	GE_DT
LT	less than	FC 23	LT_DT
LE	less than or equal to	FC 18	LE_DT

Fig. 13.7 Representation of comparison function for time data types

in LAD and FBD and as a block call in STL and SCL with the comparison result as a function value.

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.

The function value is TRUE if the comparison is fulfilled, otherwise FALSE. A tag with data type DT is considered as “less than” if the time is older.

T_COMP does not signal an error.

13.3.3 Comparison function S_COMP

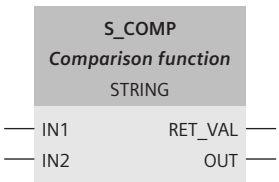
S_COMP compares two tags with data type STRING. The graphic representation is shown in Fig. 13.8. The comparison function S_COMP is implemented with loadable standard functions (FC), which are represented as an EN/ENO box in LAD and FBD and as a block call in STL and SCL with the comparison result as a function value.

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) charac-

Comparison function for tags of data type STRING

The comparison function S_COMP compares two tags of the data type STRING.
S_COMP is implemented using loadable standard functions (FC).

Compare tag of data type STRING



Comparison function:

	Comparison for	Block	
EQ	equal to	FC 10	EQ_STRNG
NE	not equal to	FC 29	NE_STRNG
GT	greater than	FC 15	GT_STRNG
GE	greater than or equal to	FC 13	GE_STRNG
LT	less than	FC 24	LT_STRNG
LE	less than or equal to	FC 19	LE_STRNG

Fig. 13.8 Representation of comparison function for strings

ters 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 function value is TRUE if the comparison is fulfilled, otherwise FALSE.

S_COMP does not signal an error.

13.4 Arithmetic functions

Two types of arithmetic functions are available:

- Arithmetic functions implemented using "basic instructions" for basic arithmetic operations in association with numbers of data types INT, DINT, and REAL
- Arithmetic functions for date and time implemented using system blocks (T_ADD, T_SUB, T_DIFF, T_COMBINE).

With SCL, the arithmetic functions implemented with system blocks are included in the arithmetic expression.

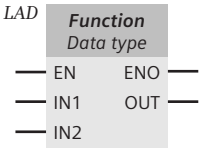
13.4.1 General function description

A "simple" arithmetic function links two digital values in accordance with the basic arithmetic operations. These are two digital tags which are linked in the case of LAD, FBD, and STL; expressions are also possible in the case of SCL. The result of an arithmetic function is transferred with LAD and FBD to a digital tag, it is present in accumulator 1 with STL, and with SCL the result can be assigned to a digital tag or linked further.

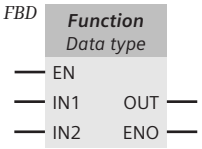
Fig. 13.9 shows the general representation of an arithmetic function in the various programming languages.

Arithmetic functions

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

*) All parameters of the same data type

An arithmetic function links the contents of two tags IN1 and IN2 according to the following scheme: Result := IN1 <logic operation> IN2.

Function:		***Data types:***
ADD	Addition	INT, DINT, REAL
SUB	Subtraction	INT, DINT, REAL
MUL	Multiplication	INT, DINT, REAL
DIV	Division	INT, DINT, REAL
MOD	Division with remainder as result	DINT

STL

```
L   IN1
L   IN2
Arithmetic operation
T   OUT
```

The contents of accumulator 2 (IN1) are linked with the contents of accumulator 1 (IN2); the result is stored in accumulator 1. The contents of accumulator 2 remain unchanged.

Arithmetic operation:

+I, +D, +R	Addition	The arithmetic operation interprets the contents of the accumulators in accordance with the data type during the operation:
-I, -D, -R	Subtraction	
*I, *D, *R	Multiplication	
/I, /D, /R	Division	I = INT, D = DINT, R = REAL.
MOD	Division with remainder as result	

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.

SCL

```
OUT := IN1 Function IN2;
```

Function:

+	Addition
-	Subtraction
*	Multiplication
/	Division
MOD	Division with remainder as result

Approved data types:

Operand 1:	*Operand 2:*	*Result*
With *addition* and *subtraction*:		
INT, DINT, REAL	INT, DINT, REAL	INT, DINT, REAL *)
TIME	TIME	TIME
TOD	TIME	TOD
DT	TIME	DT
Additionally with *subtraction*:		
DATE	DATE	TIME
TOD	TOD	TIME
DT	DT	TIME
With *multiplication* and *division*:		
INT, DINT, REAL	INT, DINT, REAL	INT, DINT, REAL *)
TIME	INT, DINT	TIME
With *division with remainder as result*:		
INT, DINT	INT, DINT	INT, DINT *)

*) The result assumes the "more significant" data type:
REAL before DINT before INT.

Fig. 13.9 Description of arithmetic functions for numerical values

13.4.2 Data types and status bits for an arithmetic function

Operand width, data types

The tags involved in the calculation must be of the same data type unless implicit data type conversion is involved (see Chapter 13.6.1 “Implicit data type conversion” on page 501).

Calculating with data type REAL

REAL numbers are mapped internally as floating-point numbers with two numerical ranges: one range with full accuracy (“normalized” floating-point numbers) and one range with limited accuracy (“denormalized” floating-point numbers, see also Chapter 4.4.5 “Floating-point data type REAL” on page 125). A CPU 400 calculates in both ranges, a CPU 300 only in the range with full accuracy. If the result of a calculation with a CPU 300 falls in the range with limited accuracy, zero is output as result and a downward violation of the numerical range is signaled.

Setting of status bits

An arithmetic function sets all digital condition codes. CC0 and CC1 signal whether the result is negative, zero, or positive. A numerical range overflow sets OV and OS (note the different significance of CC0 and CC1 in the case of an overflow). A division by zero is indicated by “1” in all digital condition codes (Table 13.4).

Table 13.4 Status bits with an arithmetic function for fixed-point calculation

Status bits with	INT	DINT	CC0	CC1	OV	OS
The result is	< –32 768 (+I, –I)	< –2 147 483 648 (+D, –D)	0	1	1	1
	< –32 768 (*I)	< –2 147 483 648 (*D)	1	0	1	1
	–32 768 to –1	–2 147 483 649 to –1	1	0	0	–
	0	0	0	0	0	–
	+1 to +32 767	+1 to +2 147 483 647	0	1	0	–
	>+32 767 (+I, –I)	>+2 147 483 647 (+D, –D)	1	0	1	1
	>+32 767 (*I)	>+2 147 483 647 (*D)	0	1	1	1
	+32 768 (/I)	+2 147 483 648 (/D)	0	1	1	1
	(–)65 536	(–)4 294 968 296	0	0	1	1
Division by zero	(/I)	(/D, MOD)	1	1	1	1

The result of an arithmetic function with data type REAL is in the permissible numerical range if it is “normalized”. A invalid REAL number is indicated by “1” in all digital condition codes (Table 13.5).

Table 13.5 Status bits with an arithmetic function for floating-point calculation

Status bits with	REAL	CC0	CC1	OV	OS
The result is	+ 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

If an error occurs during execution of the arithmetic function, the binary result BR and the ENO output are set to signal state “0” in the case of LAD and FBD, and the ENO tag and the ENO output to FALSE in the case of SCL.

Master control relay (MCR) dependency

The (actual) arithmetic function which links the contents of the two accumulators is independent of the MCR function. Saving of the result of a calculation with an EN/ENO box (LAD, FBD) or with a transfer statement (STL) only takes place with the MCR functionality switched off. Zero is saved if the MCR function is switched on.

13.4.3 Execution of the arithmetic function

Addition ADD

ADD adds the two input tags IN1 and IN2 and saves the total in the result OUT. An error occurs if the permissible numerical range is left, if one of the input tags is an invalid floating-point number, or if an attempt is made to add the floating-point numbers +∞ and -∞.

Subtraction SUB

SUB subtracts the input tag IN2 from the input tag IN1 and saves the difference in the result OUT. An error occurs if the permissible numerical range is left, if one of the input tags is an invalid floating-point number, or if an attempt is made to subtract the floating-point numbers +∞ from +∞ or -∞ from -∞.

Multiplication MUL

MUL multiplies the two input tags IN1 and IN2 and saves the product in the result OUT. An error occurs if the permissible numerical range is left, if one of the input tags is an invalid floating-point number, or if an attempt is made to multiply the floating-point numbers ∞ and 0.

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 the quotient and signals an error.

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

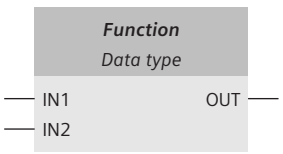
MOD divides the input tag IN1 (dividend) by the input tag IN2 (divisor) and delivers the remainder of the division in the result OUT. The remainder is the leftover part of the division; this is not the decimal places. With a negative dividend, the remainder is also negative. A division by zero delivers a value of zero as the remainder and signals an error.

13.4.4 Arithmetic functions for date and time

The arithmetic functions for date and time link tags with data types TIME, TOD, DATE, and DT. Fig. 13.10 shows the graphic representation of these functions. The date/timer functions are loadable standard functions (FC) which are represented as EN/ENO boxes in LAD and FBD and as block call with function value in STL and SCL.

Arithmetic functions for date and time

Arithmetic functions for date and time



Function:

Function	Block
T_ADD	FC 1 AD_DT_TM
T_SUB	FC 35 SB_DT_TM
T_DIFF	FC 34 SB_DT_ST
T_COMBINE	FC 3 D_TOD_DT

Function description	***Data types:*** ***IN1***		***IN2***		***OUT***
T_ADD adds a duration to a point in time	DT	+	TIME	=	DT
T_SUB subtracts a duration from a point in time	DT	-	TIME	=	DT
T_DIFF generates the difference between two points in time	DT	-	DT	=	TIME
T_COMBINE combines the date and time into one point in time	DATE	+	TOD	=	DT

Fig. 13.10 Graphic representation of the arithmetic functions for date and time

If the result of the arithmetic function is not within the permissible range, the result is limited to the corresponding value. The binary result BR and the ENO output are then set to “0” in the case of LAD and FBD, and the ENO tag and ENO output to FALSE in the case of SCL.

Some date/timer functions set the binary result BR and the ENO output to “0” if an error has occurred during processing of the function.

T_ADD Add duration to a time

T_ADD adds a duration in TIME format to a time in DATE_AND_TIME format and delivers a new time in DATE_AND_TIME format as the result. The result must be in the range from DT#1990-01-01-00:00:00.000 to DT#2089-12-31-59:59:59.999.

T_SUB Subtract duration from a time

T_SUB subtracts a duration in TIME format from a time in DATE_AND_TIME format and delivers a new time in DATE_AND_TIME format as the result. The result must be in the range from DT#1990-01-01-00:00:00.000 to DT#2089-12-31-59:59:59.999.

T_DIFF Subtract two times

T_DIFF subtracts two times in DATE_AND_TIME format and delivers a duration in TIME format as the result. The times must be in the range from DT#1990-01-01-00:00:00.000 to DT#2089-12-31-59:59:59.999.

If the first time IN1 is greater (more recent) than the second time IN2, the result is positive; if the first time is smaller (less recent) than the second time, the result is negative.

T_COMBINE Combine DATE and TIME_OF_DAY into DT

T_COMBINE combines the DATE and TIME_OF_DAY formats and converts these formats into the DATE_AND_TIME format. The input value IN1 must be between the limits DATE#1990-01-01 and DATE#2089-12-31.

T_COMBINE does not signal errors.

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. 13.11 shows the general representation of a math function in the various programming languages.

“Math functions” include the following:

- Sign, cosine, tangent
- Arcsine, arccosine, arctangent

Mathematical functions

LAD

Function *Data type*	
EN	ENO
IN	OUT

FBD

Function *Data type*	
EN	OUT
IN	ENO

Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
IN	INPUT	REAL	Digital tag
OUT	OUTPUT	REAL	Result

Function:

SIN	Sine		**ASIN**	Arcsine
COS	Cosine		**ACOS**	Arccosine
TAN	Tangent		**ATAN**	Arc tangent
SQR	Generate square		**EXP**	Exponential function to base e
SQRT	Extract square root		**LN**	Natural logarithm

The value present in the IN parameter is calculated according to the mathematical function, and output in the OUT parameter.

STL

```
L  IN
Mathematical operation
T  OUT
```

The value in accumulator 1 is calculated according to the mathematical operation, and the result stored in accumulator 1. The contents of accumulator 2 remain unchanged.
A mathematical operation influences the status bits.

Operation:

SIN	Sine		**ASIN**	Arcsine
COS	Cosine		**ACOS**	Arccosine
TAN	Tangent		**ATAN**	Arc tangent
SQR	Generate square		**EXP**	Exponential function to base e
SQRT	Extract square root		**LN**	Natural logarithm

SCL

```
OUT := Function (IN);
```

Function:

SIN	Sine		**ASIN**	Arcsine
COS	Cosine		**ACOS**	Arccosine
TAN	Tangent		**ATAN**	Arc tangent
SQR	Generate square		**EXP**	Exponential function to base e
SQRT	Extract square root		**LN**	Natural logarithm
			10**	Exponential function to base 10
			LOG	Common logarithm

Data type:
INT, DINT, and REAL are permitted as data types for the input tag. The result is present in the data type REAL.

Fig. 13.11 Overview of math functions

- ▷ Generate square, extract square root
- ▷ Exponential function to base e, natural logarithm

13.5.2 General execution of a math function

All math functions process numbers in data format REAL. With SCL, the input parameters can also be of data type INT or DINT; the result is always delivered with data type REAL.

Setting of status bits

A math function sets all digital condition codes (Table 13.6).

An error occurs if the permissible numerical range is left or if the input tag is an invalid floating-point number. The binary result BR and the ENO output are then set to "0" in the case of LAD and FBD, and the ENO tag and ENO output to FALSE in the case of SCL.

Master control relay (MCR) dependency

The (actual) math function which changes the content of accumulator 1 is independent of the MCR function. Saving of the result of a calculation with an EN/ENO box (LAD, FBD) or with a transfer statement (STL) only takes place with the MCR functionality switched off. Zero is saved if the MCR function is switched on.

13.5.3 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.

Table 13.6 Status bits with a math function

Status bits with	REAL	CC0	CC1	OV	OS
The result is	+ 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

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.

13.5.4 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. 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 13.7).

Table 13.7 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.

13.5.5 Additional math functions

The following additional math functions are available:

SQR Generate square

SQRT Extract square root

EXP Generate exponential function to base e

LN Generate natural logarithm (logarithm to base e)

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.

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.

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

$$\lg a = \frac{\ln a}{\ln 10} = 0.4342945 \cdot \ln a$$

13.6 Conversion functions

If you link tags together, they must have the same 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, the data type must be converted. The conversion functions are available for this.

The following conversion functions are available:

- Conversion of fixed-point numbers
- Conversion of floating-point numbers

- Generation of absolute value and negation
- Conversion of time data types (T_CONV) and character data types (S_CONV) implemented using system blocks
- Conversion of hexadecimal numbers (ATH, HTA) as well as scaling and unscaling (SCALE, UNSCALE) implemented using system blocks

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.

13.6.1 Implicit data type conversion

“Implicit” conversion functions also exist which convert a data type without changing the bit assignments and do not signal an error, for example the conversion from BYTE to WORD. These conversions are carried out “automatically”.

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.

It always applies during implicit data type conversion that the bit length of the source data type must not exceed that of the destination data type. For example, a tag with data format DWORD (source data type) cannot be applied to a block parameter which expects data type WORD (destination data type).

The scope of implicit data type conversion depends on the block attribute *IEC check* (see Table 13.8).

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);
```

13.6.2 Data type conversion of fixed-point numbers

The conversion function CONV converts the data types of fixed-point numbers. Fig. 13.12 shows the general representation of the conversion function in the various programming languages. CONV is represented as an EN/ENO box in the case of LAD and FBD, as operations in the case of STL which convert the value in accumulator 1, and in the case of SCL there are functions with the notation *Source data type_TO_Destination data type*.

The conversion options additionally offered by SCL are described in Chapter 10.5.5 “Conversion functions” on page 379.

Table 13.8 Implicit data type conversion

from \ to	BOOL	BYTE	WORD	DWORD	INT	DINT	REAL	TIME	S5TIME	DT	TOD	DATE	CHAR	STRING
BOOL		XS	XS	XS										
BYTE			X	X									O	
WORD				X					O					
DWORD								O						
INT						XS	XS							
DINT							XS	O			OS			
REAL														
TIME				O		O								
S5TIME			O											
DT														
TOD														
DATE														
CHAR		O	OS	OS										XS

Implicit data type conversion is possible:

- **X** Independent of attribute *IEC check*
- **O** With deactivated attribute *IEC check*
- **XS** Additionally with SCL and independent of attribute *IEC check*
- **OS** Additionally with SCL and deactivated attribute *IEC check*

Setting of status bits and of the ENO output

If an error occurs when converting a fixed-point number to a BCD number, the status bits OV *Exception Bit Overflow* and OS *Exception Bit Overflow Stored* are set to "1", the ENO output is set to "0", and with SCL the ENO tag is set to FALSE.

Master control relay (MCR) dependency

The (actual) conversion function which changes the content of accumulator 1 is independent of the MCR function. Saving of the result of a conversion with an EN/ENO box (LAD, FBD) or with a transfer statement (STL) only takes place with the MCR functionality switched off. Zero is saved if the MCR function is switched on.

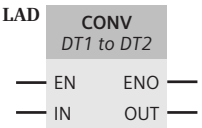
Conversion of INT to DINT

The function interprets the input value as a number with data type INT and transfers it to the right word of the output value. The signal state of bit 15 (the sign) of the input is transferred to bits 16 to 31 of the left word of the output value.

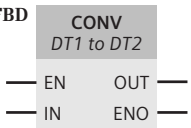
The conversion of INT to DINT does not report any errors.

Conversion of fixed-point numbers (CONV)

LAD



FBD



Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
IN	INPUT	Data type 1	Input tag
OUT	OUTPUT	Data type 2	Output tag

Function: The conversion function **CONV** is used to convert data types. The contents of the tag with data type **DT1** present in the IN parameter are converted and transferred to the tag with data type **DT2** present in the OUT parameter.

OUT := CONV(IN)

The following combinations of data types are possible:

Data type conversion for	from *DT1*	to *DT2*	*STL operation*	*SCL instruction*
Fixed-point numbers	INT	DINT	ITD	INT_TO_DINT
		BCD16	ITB	INT_TO_BCD
	DINT	REAL	DTR	DINT_TO_REAL
		BCD32	DTB	DINT_TO_BCD
BCD numbers	BCD16	INT	BTI	BCD_TO_INT
	BCD32	DINT	BTD	BCD_TO_DINT

STL

```
L   #var_dint      //Example of a conversion from DINT to REAL
DTR
T   #var_real
```

SCL

```
//Example of a conversion from DINT to REAL
#var_real := DINT_TO_REAL (#var_dint);
```

Fig. 13.12 Function and representation of the conversion function CONV

Conversion of INT to BCD

The function interprets the input value as a number with data type INT and converts it into a BCD-coded number with 3 decades. The 3 right-justified decades represent the absolute value of the decimal number. The sign is present in bits 12 to 15. If all bits are set to signal state “0”, the sign is positive; all bits set to signal state “1” means a negative sign.

If the INT number is too large for conversion into a BCD number (> 999), a conversion does not take place and the function sets the status bits OV and OS to “1”. The binary result and the ENO output are then set to signal state “0” in the case of LAD and FBD and the ENO tag and the ENO output to FALSE in the case of SCL.

Conversion of DINT to BCD

The function interprets the input value as a number with data type DINT and converts it into a BCD-coded number with 7 decades as the output value. The 7 right-justified decades represent the absolute value of the decimal number. The sign is

present in bits 28 to 31. If all bits are set to signal state “0”, the sign is positive; all bits set to signal state “1” means a negative sign.

If the DINT number is too large for conversion into a BCD number (> 9 999 999), a conversion does not take place and the function sets the status bits OV and OS to “1”. The binary result and the ENO output are then set to signal state “0” in the case of LAD and FBD and the ENO tag and the ENO output to FALSE in the case of SCL.

Conversion of DINT to REAL

The function interprets the input value as a number in DINT format and converts it into a number in floating-point format REAL.

Since a number in DINT format has a greater accuracy than a number in REAL format, rounding off may take place during conversion. It is rounded to the next integer (corresponding to the ROUND function).

The function does not report any errors.

Conversion of BCD to INT

The function interprets the input value as a BCD-coded number with 3 decades and converts it into an INT number. The 3 right-justified decades represent the absolute value of the decimal number. The sign is present in bits 12 to 15. Signal state “0” of these bits means “positive”, signal state “1” means “negative”. Only the signal state of bit 15 is considered during conversion.

If a pseudo tetrad (numerical value 10 to 15 or A to F in hexadecimal representation) is present in the BCD-coded number, the CPU signals a programming error and calls the organization block OB 121 *Programming error*. If this is not present, the CPU switches to the stop status.

The function does not set any status bits.

Conversion of BCD to DINT

The function interprets the input value as a BCD-coded number with 7 decades and converts it into an DINT number. The 7 right-justified decades represent the absolute value of the decimal number. The sign is present in bits 28 to 31. Signal state “0” of these bits means “positive”, signal state “1” means “negative”. Only the signal state of bit 31 is considered during conversion.

If a pseudo tetrad (numerical value 10 to 15 or A to F in hexadecimal representation) is present in the BCD-coded number, the CPU signals a programming error and calls the organization block OB 121 *Programming error*. If this is not present, the CPU switches to the stop status.

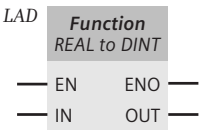
The function does not set any status bits.

13.6.3 Data type conversion of floating-point numbers

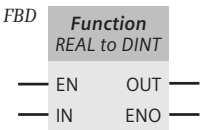
The conversion functions convert data types of floating-point numbers into fixed-point numbers. Fig. 13.13 shows the general representation of the conversion functions in the various programming languages. They are represented as an EN/ENO box in the case of LAD and FBD, as operations in the case of STL which convert the value in accumulator 1, and in the case of SCL there are functions with an input value.

Conversion of floating-point numbers

LAD



FBD



Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
IN	INPUT	REAL	Input tag
OUT	OUTPUT	DINT	Output tag

Function: The contents of the tag with data type **REAL** present in the IN parameter are converted and transferred to the tag with data type **DINT** present in the OUT parameter.

OUT := Function (IN)

The following data type conversions are possible for a floating-point number:

Conversion from REAL to DINT	*LAD/FBD function*	*STL operation*	*SCL instruction*
With rounding to the next integer	ROUND	RND	ROUND
With rounding to the next higher integer	CEIL	RND+	–
With rounding to the next smaller integer	FLOOR	RND–	–
Without rounding	TRUNC	TRUNC	TRUNC
Conversion from REAL with rounding to DINT	–	–	REAL_TO_DINT
Conversion from REAL with rounding to INT	–	–	REAL_TO_INT

STL

```
L   #var_real          //Example of a conversion from REAL to DINT
TRUNC
T   #var_dint
```

SCL

```
//Example of a conversion from REAL to DINT
#var_real := TRUNC(#var_dint);
```

Fig. 13.13
Function and representation of the conversion functions for floating-point numbers

The conversion options additionally offered by SCL are described in Chapter 10.5.5 “Conversion functions” on page 379.

Setting of status bits

If an error occurs during conversion of a floating-point number to a fixed-point number, the status bits OV *Exception Bit Overflow* and OS *Exception Bit Overflow Stored* are set to “1”. The binary result BR and the ENO output are set to signal

state “0” in the case of LAD and FBD and the ENO tag and the ENO output to FALSE in the case of SCL.

Master control relay (MCR) dependency

The (actual) conversion function which changes the content of accumulator 1 is independent of the MCR function. Saving of the result of a conversion with an EN/ENO box (LAD, FBD) or with a transfer statement (STL) only takes place with the MCR functionality switched off. Zero is saved if the MCR function is switched on.

Rounding to the next integer (ROUND, RND)

The function interprets the input value as a number in REAL format and converts it into a number in DINT format. The function 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.

If the input value is greater than or less than the range permissible for a number in DINT format, or does not correspond to any number in REAL format, the function sets the status bits OV and OS to “1” and the ENO output to “0”. A conversion is not carried out in this case.

Rounding to the next higher integer (CEIL, RND+)

The function interprets the input value as a number in REAL format and converts it into a number in DINT format. The function returns an integer which is greater than or equal to the number to be converted.

An error occurs if the input value is outside the DINT numerical range or is an invalid floating-point number. A conversion is not carried out in this case.

Rounding to the next lower integer (FLOOR, RND–)

The function interprets the input value as a number in REAL format and converts it into a number in DINT format. The function returns an integer which is less than or equal to the number to be converted.

An error occurs if the input value is outside the DINT numerical range or is an invalid floating-point number. A conversion is not carried out in this case.

Without rounding (TRUNC)

The function interprets the input value as a number in REAL format and converts it into a number in DINT format. The function returns the whole number part of the number to be converted; the fractional part is truncated.

An error occurs if the input value is outside the DINT numerical range or is an invalid floating-point number. A conversion is not carried out in this case.

Summary of conversion from REAL to DINT

Table 13.9 shows the different effects of the conversion functions REAL to DINT. The range between –1 and +1 has been selected as an example.

Table 13.9 Rounding modes when converting REAL numbers

Input value		Result			
REAL	DW#16#	ROUND/RND	CEIL/RND+	FLOOR/RND–	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

13.6.4 Data type conversion for date/time with T_CONV

The conversion function for date and time converts tags with data types TIME, TOD, DATE, and DT. Fig. 13.14 shows the graphic representation of these functions. The date/timer functions are loadable standard functions (FC) which are represented as EN/ENO boxes in LAD and FBD and as block call with function value in STL and SCL.

Data type conversion DT to DATE

The data type conversion DT to DATE extracts the data format DATE (D#) from the format DATE_AND_TIME (DT#). DATE is between the limits DATE#1990-1-1 and DATE#2089-12-31.

The function does not report any errors.

Data type conversion DT to TOD

The data type conversion DT to TOD extracts the data format TIME_OF_DAY (TOD#) from the format DATE_AND_TIME (DT#). TOD is between the limits TOD#00:00:00.000 and TOD#23:59:59.999.

The function does not report any errors.

Data type conversion DT to INT

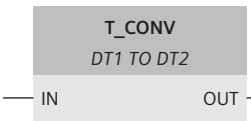
The data type conversion DT to INT extracts the day of the week from the format DATE_AND_TIME (DT#). The day of the week is present in data format INT:

1 = Sunday
2 = Monday
3 = Tuesday
4 = Wednesday
5 = Thursday
6 = Friday
7 = Saturday

The function does not report any errors.

Conversion function T_CONV

Conversion of time data types



The following conversions are possible:

Data type DT1	Data type DT2	SCL function	Block
DT	DATE	DT_TO_DATE	FC 6 DT_DATE
	TOD	DT_TO_TOD	FC 8 DT_TOD
	INT	DT_TO_INT	FC 7 DT_DAY
S5TIME	TIME	S5TIME_TO_TIME	FC 33 S5TI_TIM
TIME	S5TIME	TIME_TO_S5TIME	FC 40 TIM_S5TI

Fig. 13.14 Graphic representation of the conversion function T_CONV

Data type conversion S5TIME to TIME

The data type conversion S5TIME to TIME converts the data format S5TIME into the format TIME.

The function does not report any errors.

Data type conversion TIME to S5TIME

The data type conversion TIME to S5TIME converts the data format TIME into the format S5TIME. The value is rounded down during conversion.

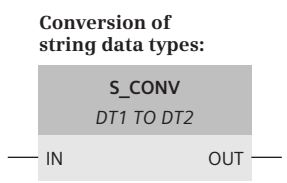
If the input parameter is greater than the S5TIME format which can be represented (greater than TIME# 02:46:30.000), S5TIME# 999.3 is output as result. The binary

result BR and the ENO output are then set to “0” in the case of LAD and FBD and the ENO tag and the ENO output to FALSE in the case of SCL.

13.6.5 Data type conversion for data type STRING with S_CONV

The conversion function for data type STRING converts tags with data type STRING into numerical data types and vice versa. The graphic representation of this function is shown in Fig. 13.15. The conversions are loadable standard functions (FC) which are represented as EN/ENO boxes in LAD and FBD and as block call with function value in STL and SCL.

Conversion function S_CONV



The following conversions are possible:

Data type DT1	Data type DT2	SCL function	Block
STRING	INT	STRING_TO_INT	FC 38 STRNG_I
	DINT	STRING_TO_DINT	FC 37 STRNG_DI
	REAL	STRING_TO_REAL	FC 39 STRNG_R
INT	STRING	INT_TO_STRING	FC 16 I_STRNG
DINT	STRING	DINT_TO_STRING	FC 5 DI_STRNG
REAL	STRING	REAL_TO_STRING	FC 30 R_STRNG

Fig. 13.15 Graphic representation of the conversion function S_CONV

If an error occurs during conversion, the binary result BR and the ENO output are set to “0” in the case of LAD and FBD and the ENO tag and the ENO output to FALSE in the case of SCL. A conversion is not carried out in this case.

STRING tag in the temporary local data

If you assign a STRING function value to a STRING tag which is located in the temporary local data, you must assign a defined value with the required length to this tag in the program (a default setting per declaration is not possible in the temporary local data).

Data type conversion INT to STRING

The function converts a tag in INT format into a string. The string is shown preceded by a sign (number of digits plus sign).

An error occurs if the destination tag is too short.

Data type conversion DINT to STRING

The function converts a tag in DINT format into a string. The string is shown preceded by a sign (number of digits plus sign).

An error occurs if the destination tag is too short.

Data type conversion REAL to STRING

The function converts a tag in REAL format into a string. The string is shown with 14 digits:

±v.nnnnnnnE±xx	±	Sign
	v	1 integer digit position
	n	7 decimal places
	x	2 exponent digits

An error occurs if the destination tag is too short or if the input tag is an invalid floating-point number.

Data type conversion STRING to INT

The function converts a string into a tag in INT format. The first character in the string may be a sign or a digit, the characters which then follow must be digits.

An error occurs if the length of the string is zero or greater than 6, if the string contains illegal characters, or if the converted value leaves the numerical range of INT.

Data type conversion STRING to DINT

The function converts a string into a tag in DINT format. The first character in the string may be a sign or a digit, the characters which then follow must be digits.

An error occurs if the length of the string is zero or greater than 11, if the string contains illegal characters, or if the converted value leaves the numerical range of DINT.

Data type conversion STRING to REAL

The function converts a string into a tag in REAL format. The string must have the following format:

±v.nnnnnnnE±xx	±	Sign
	v	1 integer digit position
	n	7 decimal places
	x	2 exponent digits

An error occurs if the length of the string is less than 14, if the string is not formatted as shown above, or if the converted value leaves the numerical range of REAL.

13.6.6 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. The conversion functions are loadable standard functions (FC). They are represented as EN/ENO boxes in the case of LAD and FBD and as block calls with the conversion result as the function value in the case of STL

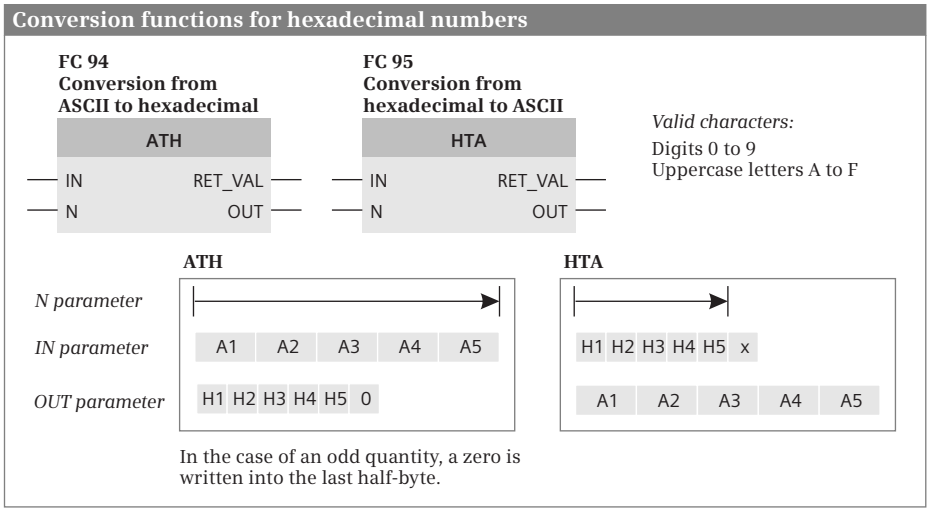


Fig. 13.16 Graphic representation of the ATH and HTA conversion functions

and SCL. The graphic representation of the conversion functions is shown in Fig. 13.16.

Area pointers of data type POINTER which point to the first byte of the input or output data area are expected at the IN and OUT parameters. Example: P#DB10.DBX12.0. The N parameter with data type INT specifies the number of characters to be converted.

ATH 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 and the uppercase letters A to F are permissible. An illegal character is converted into zeroes and an error message is output at the RET_VAL parameter.

HTA Conversion from hexadecimal to ASCII

HTA converts a string present in hexadecimal code into an ASCII-coded string. HTA does not report any errors.

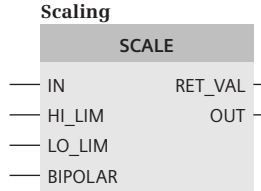
13.6.7 Scaling and unscaling

SCALE converts a fixed-point number into a floating-point number and scales it between two limits in the process. UNSCALE unscales a floating-point number between two limits and then converts it into a fixed-point number.

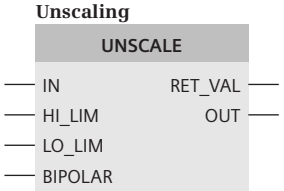
SCALE and UNSCALE are loadable standard functions (FC). They are represented as EN/ENO boxes in the case of LAD and FBD and as block calls with RET_VAL as the function value in the case of STL and SCL. The graphic representation of the functions is shown in Fig. 13.17.

Conversion functions for scaling and unscaling

FC 105
Scaling



FC 106
Unscaling



Data types:

Parameter	SCALE	UNSCALE
IN	INT	REAL
HI_LIM	REAL	REAL
LO_LIM	REAL	REAL
BIPOLAR	BOOL	BOOL
OUT	REAL	INT

SCALE

Conversion formula:

$$OUT = [\frac{IN - K1}{K2 - K1} \times (HI\_LIM - LO\_LIM)\ ] + LO\_LIM \qquad K1 < IN < K2$$

Two cases are distinguished:

1) The value of IN is unipolar in the range from 0 to +27 468; in this case, the BIPOLAR parameter is “0” and the constants have the values K1 = 0 and K2 = +27 468.
2) The value of IN is bipolar in the range from -27 468 to +27 468; in this case, the BIPOLAR parameter is “1” and the constants have the values K1 = -27 468 and K2 = +27 468.

UNSCALE

Conversion formula:

$$OUT = [\frac{IN - LO\_LIM}{HI\_LIM - LO\_LIM} \times (K2 - K1)\ ] + K1 \qquad K1 < OUT < K2$$

Two cases are distinguished:

1) The value of OUT is unipolar in the range from 0 to +27 468; in this case, the BIPOLAR parameter is “0” and the constants have the values K1 = 0 and K2 = +27 468.
2) The value of OUT is bipolar in the range from -27 468 to +27 468; in this case, the BIPOLAR parameter is “1” and the constants have the values K1 = -27 468 and K2 = +27 468.

Fig. 13.17 Graphic representation of the SCALE and UNSCALE functions

SCALE 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. Example of the application: conversion of an analog value from an analog input module into physical units.

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. 13.17), 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.

UNSCALE 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. Example of the application: conversion of physical units into an analog value for an analog output module.

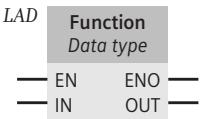
If the value at IN is greater than the value of the upper limit HI_LIM, the value of the constant K2 is output and an error signaled. If the value at IN is less than the lower limit LO_LIM, the value K1 (see Fig. 13.17) is output and an error signaled.

13.6.8 Further conversion functions

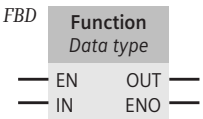
Further conversion functions include generation of an absolute value and negation (Fig. 13.18).

Generation of absolute value, negation

LAD



FBD



Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
IN	INPUT	Data type *)	Input tag
OUT	OUTPUT	Data type *)	Result

*) Same data types at IN and OUT

Function:		***Data type:***
ABS	Generation of absolute value	REAL
NEG	Negation	INT, DINT, REAL

STL

```
L  IN
Operation
T  OUT
```

Operation:		***Data type:***
ABS	Generation of absolute value	REAL
NEGI	Negation	INT
NEGD	Negation	DINT
NEGR	Negation	REAL

SCL

```
OUT := ABS(IN); //Absolute value
OUT := -IN;     //Negation
```

SCL *function:*		***Data type:***
ABS	Generation of absolute value	INT, DINT, REAL
–	Negation (multiplication by -1)	INT, DINT, REAL

Fig. 13.18 Generation of absolute value and negation

ABS Generation of absolute value

The ABS function generates the absolute value of the input value and writes the result in the output value. With the data type REAL, ABS sets the sign of the mantissa to “0”, even with an invalid REAL number.

The function does not report any errors and does not set any status bits.

Negation, generation of two's complement

The negation reverses the sign of the input value and outputs the result as output value. Execution of the function is equivalent to multiplication by –1.

If the result is outside the permissible numerical range with the data types INT and DINT, the overflow condition code is set. With the data types INT and DINT, the function sets the status bits CC0, CC1, OV, and OS (Table 13.10).

Table 13.10 Status bits with the negation of a fixed-point number

Status bits with	INT	DINT	CC0	CC1	OV	OS
The result is	(–)32 768	(–)2 147 483 648	1	0	1	1
	–32 768 to –1	–2 147 483 649 to –1	1	0	0	–
	0	0	0	0	0	–
	+1 to +32 767	+1 to +2 147 483 647	0	1	0	–

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. 13.19 shows the general representation of a shift function in the various programming languages.

13.7.2 General execution of a shift function

Data types of the tags with LAD and FBD

The tag *N* has data type BYTE, WORD, or INT. The tags *IN* and *OUT* have data types WORD or DWORD for SHL; SHR has data types WORD, DWORD, INT, or DINT; and ROL and ROR have data type DWORD.

Data types of tags for SCL

The tag *N* has data type INT or DINT. The tags *IN* and *OUT* have data types BYTE, WORD, or DWORD. The data type on OUT has the same or greater width than the data type on IN.

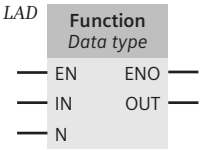
Shift number

If the shift number = 0, the function is not executed and the input value is then also the output value. If the shift number is greater than 15 (with word by word shifting) or greater than 31 (with doubleword by doubleword shifting), shifting is carried out by the available digits.

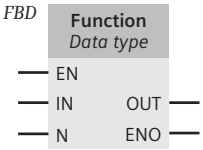
LAD, FBD, SCL: The shift number at the N input specifies the number of bit positions by which shifting is carried out. This can be a constant or variable.

Shift functions

LAD



FBD



Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
IN	INPUT	Data type *)	Input tag
N	INPUT	WORD, INT	Quantity
OUT	OUTPUT	Data type *)	Result

*) Same data type at IN and OUT

Function: **SHR** Shift to right
SHL Shift to left

Data type: WORD, DWORD, INT, DINT

SHR with WORD and DWORD: The bit positions that become free are filled with zeros.
SHR with INT and DINT: The bit positions that become free are filled with bit 15 or bit 31.
SHL: The positions that become free are filled with zeros.

Function: **ROR** Rotate to right
ROL Rotate to left

Data type: DWORD

ROR, ROL: The bit positions that become free are filled with the "shifted out" bit positions.

STL

```
L  IN
Operation with N parameter
T  OUT
```

```
L  N
L  IN
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):
Rotate to right	RRD	–
Rotate to left	RLD	–
Shift to right	SRD	SRW
Shift with sign	SSD	SSI
Shift to left	SLD	SLW
Rotate to right through CC1	RRDA	–
Rotate to left through CC1	RLDA	–

SCL

```
OUT := Function (IN := input tag, N := quantity);
```

A shift function shifts the contents of the IN input tag by N bit positions. When shifting, the bit positions that become free are filled with zeros; when rotating, they are filled with the shifted out bit positions.

Function:		
	Rotate to right	ROR
	Rotate to left	ROL
	Shift to right	SHR
	Shift to left	SHL

Data type: BYTE, WORD, DWORD

Fig. 13.19 Shift functions, representation, and principle of operation

STL: The shift number can be present as a parameter in the shift statement or in accumulator 2. The content of accumulator 1 is shifted. In the case of word by word shifting, only the right word (bits 0 to 15) is shifted, the left word in accumulator 1 remains uninfluenced.

Setting of status bits

The shift functions set the digital condition codes CC0, CC1, and OV (Table 13.11). With a shift quantity equal to zero, the condition code bits remain uninfluenced.

Table 13.11 Status bits with a shift function

Status bits		CC0	CC1	OV	OS
The "shifted-out" bit has	the signal state "0"	0	0	0	–
	the signal state "1"	0	1	0	–
The shift number is zero		No change			

Master control relay (MCR) dependency

The (actual) shift function which changes the content of accumulator 1 is independent of the MCR function. Saving of the result with an EN/ENO box (LAD, FBD) or with a transfer statement (STL) only takes place with the MCR functionality switched off. Zero is saved if the MCR function is switched on.

13.7.3 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 input tag has data type WORD or DWORD, the bit positions that become free when shifting are padded with zeros. If the input tag has data type INT or DINT, the bit positions that become free when shifting are filled with the sign of the fixed-point number.

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 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.

Shift to right with sign 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 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 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.

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. The bit positions that become free when shifting are padded with zeroes.

13.7.4 Shift to left

Shift to left with LAD and FBD

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. 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.

Shift to left with 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. The bit positions that become free when shifting are padded with zeroes.

13.7.5 Rotate to right

Rotate to right with LAD and FBD

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. 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.

Rotate to right with 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. The bit positions that become free when shifting are filled with the signal state of the shifted-out positions.

13.7.6 Rotate to left

Rotate to left with LAD and FBD

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. 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.

Rotate to left with 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. The bit positions that become free when shifting are filled with the signal state of the shifted-out positions.

13.7.7 Rotating by the condition code bit CC1 (STL)

The RLDA shift function shifts the entire contents of accumulator 1 by 1 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”.

The RRDA shift function shifts the entire contents of accumulator 1 by 1 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, MIN, MAX, LIMIT)

Word logic operations and the invert function are implemented as sequences of statements, the other logic functions are loadable functions (FC) which are represented as EN/ENO boxes in LAD and FBD and as a function with function value in STL and SCL.

13.8.1 Word logic operations

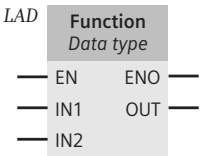
General processing of a word logic operation

A word logic operation links the values of two digital tags bit by bit according to AND, OR, or exclusive OR. Fig. 13.20 shows the general representation of a digital logic operation in the various programming languages.

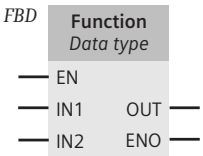
FBD and LAD use boxes with EN/ENO for the word logic operation. STL links the contents of accumulators 1 and 2 or the content of accumulator 1 to a constant

Logical operations

LAD



FBD



Name	Declaration	Data type	Description
EN	-	BOOL	Enabling input
ENO	-	BOOL	Enabling output
IN1	INPUT	Data type *)	Input tag 1
IN2	INPUT	Data type *)	Input tag 2
OUT	OUTPUT	Data type *)	Result

*) Same data type at IN1, IN2, and OUT

Function:
AND Bit by bit AND operation
OR Bit by bit OR operation
XOR Bit by bit exclusive OR operation

Data type:
WORD, DWORD

Gating 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  IN1
L  IN2
Operation without parameter
T  OUT
```

```
L  IN1
Operation with parameter
T  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). If the operation has no parameters, the contents of accumulators 1 and 2 are gated. If the operation has a parameter, the contents of accumulator 1 are gated with the parameter, and the result written into accumulator 1.

Operation:
UW, UD AND operation
OW, OD OR operation
XOW, XOD Exclusive OR operation

Data types:
WORD, DWORD

SCL

```
OUT := IN1 Function IN2;
```

Function:
AND AND operation
OR OR operation
XOR Exclusive OR operation

Data types:
BYTE, WORD, DWORD

Fig. 13.20 Representation and function of word logic operations

(see Chapter 9.5.7 “Word logic operations” on page 345). With SCL, the word logic operation is a logic expression.

All tags involved in the word logic operation have the same data type; with LAD, FBD, and STL, these are tags of word and doubleword width; with SCL, these are the data types BYTE, WORD, and DWORD.

The word logic operation generates the result bit by bit. Bit 0 of the IN1 input is linked to bit 0 of the IN2 input; the result is saved in bit 0 of the OUT output. The same logic operation takes place with bit 1, bit 2, etc. up to bit 15 or bit 31.

Setting of status bits

The word logic operations set the status bits CC0 and OV to signal state “0”. If the result of the word logic operation is zero, status bit CC1 is set to “0”, otherwise to “1” (Table 13.12).

Table 13.12 Status bits with a word logic operation

Status bits		CC0	CC1	OV	OS
The result of the word logic operation	is zero	0	0	0	–
	is not zero	0	1	0	–

Master control relay (MCR) dependency

The (actual) word logic operation which changes the content of accumulator 1 is independent of the MCR function. Saving of the result with an EN/ENO box (LAD, FBD) or with a transfer statement (STL) only takes place with the MCR functionality switched off. Zero is saved if the MCR function is switched on.

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.

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.

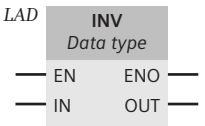
13.8.2 Invert

Inverting negates the value of a tag bit by bit; signal state “1” becomes signal state “0” and vice versa. Fig. 13.21 shows the representation of the function in the various programming languages.

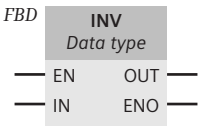
The function does not report any errors and does not set any status bits.

Invert

LAD



FBD



Name	Declaration	Data type	Description
EN	-	BOOL	Enable input
ENO	-	BOOL	Enable output
IN	INPUT	Data type *)	Input tag
OUT	OUTPUT	Data type *)	Result

*) Same data types at IN and OUT

Function:		***Data types:***
INV	Invert	INT, DINT

STL

```
L   IN
Operation
T   OUT
```

Operation:		***Data types:***
INVI	Invert	INT
INVD	Invert	DINT

SCL

```
OUT := NOT IN; //Invert
```

Function:		***Data types:***
NOT	Invert (negate)	BOOL, WORD, DWORD

Fig. 13.21 Representation of inverting

Inverting with STL

The INV_I function negates the content of the right word in accumulator 1 (bits 0 to 15). The left word is not affected.

The INV_DI function negates the content of the complete accumulator 1 (bits 0 to 31).

Inverting with SCL

The NOT operator (Boolean negation) negates the bits of the following operand or the result of the following expression.

13.8.3 Code bit and set bit number

The functions process individual bits in a tag. They are loadable standard functions (FC). They are represented as EN/ENO box in the case of LAD and FBD and as a block call without function value in the case of STL and SCL. The graphic representation of the function is shown in Fig. 13.22.

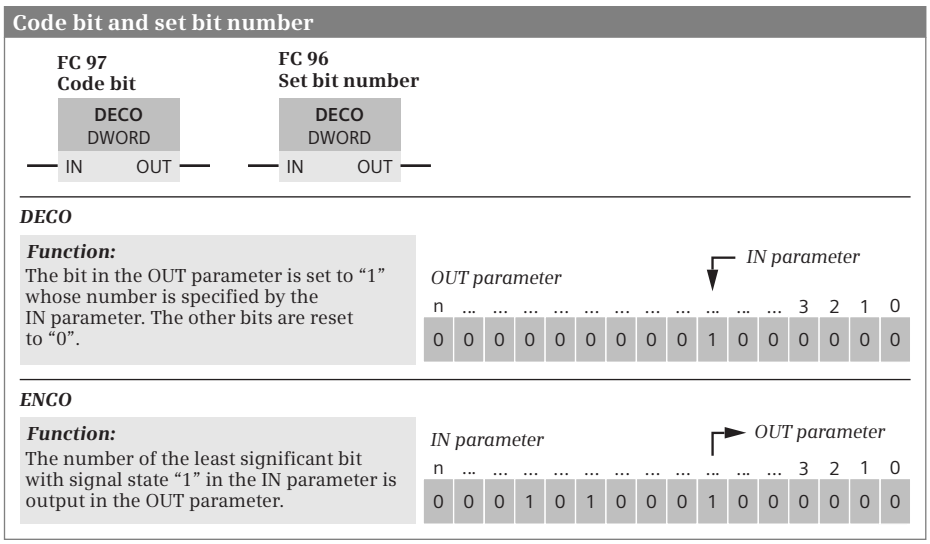


Fig. 13.22 Representation and description of the DECO and ENCO functions

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 type at the OUT parameter, only some of the bits are selected in the IN parameter: 3 bits (range 0 to 7) for BYTE, 4 bits (range 0 to 15) for WORD, and 5 bits (range 0 to 31) for DWORD.

DECO does not report any errors.

ENCO Set bit number

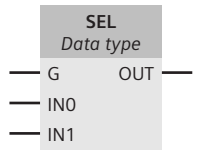
ENCO searches for the first bit set to signal state "1" in the bit sequence tag at the IN parameter starting from the right (starting with bit number 0) and outputs its number at the OUT parameter. If no bit is set, the number 0 is output at the OUT parameter and signal state "0" at the ENO output.

13.8.4 Selection and limiting functions

The selection and limiting functions select a tag or limit its value. They are implemented with loadable standard functions (FC) with LAD and FBD and with sequences of statements with SCL. They are represented as EN/ENO box in the case of LAD and FBD and as a block call with the OUT parameter as the function value in the case of STL and SCL. The graphic representation of the functions is shown in Fig. 13.23.

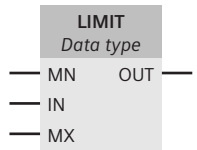
If one of the following errors occurs, the selection and limiting functions leave the function value unchanged and set the binary result BR and the ENO output to signal state "0" in the case of LAD and FBD, and the ENO tag and the ENO output to FALSE in the case of SCL:

Selection and limiting functions



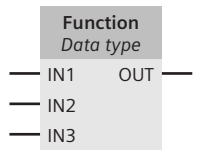
SEL Selecting (FC 36)

The G parameter selects the IN0 (with signal state "0") or IN1 (with signal state "1") input and transfers its value to the OUT parameter. All elementary data types (except BOOL) are permissible.



LIMIT Limit value (FC 22)

LIMIT limits the value in the IN parameter to the limits MN (lower limit) and MX (upper limit) and outputs it in the OUT parameter. All parameters have the same data type (INT, DINT, REAL; with SCL additionally TIME, DATE, TOD).



MAX Determine maximum (FC 25)

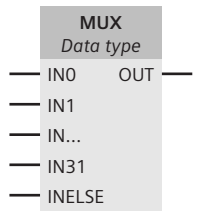
MAX transfers the largest of the three input values IN1, IN2 and IN3 to the OUT parameter. All parameters have the same data type (INT, DINT, REAL).

With SCL, the data types TIME, DATE and TOD are additionally permitted and MAX can have up to 32 inputs.

MIN Determine minimum (FC 27)

MIN transfers the smallest of the three input values IN1, IN2 and IN3 to the OUT parameter. All parameters have the same data type (INT, DINT, REAL).

With SCL, the data types TIME, DATE and TOD are additionally permitted and MIN can have up to 32 inputs.



MUX Multiple selection

Only available with SCL:

Depending on the assignment of the K parameter, MUX outputs a tag present at the INx and INELSE inputs to the OUT output. Up to 32 input parameters IN0 to IN31 can be used. If the value in the K parameter is larger than the number of inputs Inx, the value in the INELSE parameter is output.

The following data types are permissible: BYTE, WORD, DWORD, INT, DINT, REAL, TIME, DATE, TOD, DT, STRING, POINTER, and ANY.

Fig. 13.23 Representation and description of the selection and limiting functions

- A parameterized tag has an invalid data type
- All parameterized tags do not have the same data type
- A REAL tag does not represent a valid floating-point number

SEL Binary selection

SEL selects one of two tag values (IN0 and IN1) depending on a switch (parameter G). Tags with elementary data types are permissible as input values at the IN0 and IN1 parameters. Both input tags (actual parameters) and the output tag must have the same data type.

With SCL, the data types at IN0, IN1, and OUT must belong to the same class of data type (BYTE, WORD, DWORD or INT, DINT, REAL) and the output tag must have the "most significant" data type.

LIMIT Limiter

LIMIT limits the numerical value of the IN tag to the limits specified in the MN and MX parameters. Tags of data types INT, DINT, REAL are permissible as input values and additionally the data types TIME, DATE, and TOD with SCL. All input tags and the output tag must have the same data type or – with SCL – belong to the same class of data type (INT, DINT, REAL) and the output tag must have the "most significant" data type. The lower limit (MN parameter) must be smaller than the upper limit (MX parameter).

The function signals an error if – in addition to the errors mentioned above – the lower limit MN is not smaller than the upper limit MX.

MAX Maximum selection

MAX selects the largest of three numerical tag values in the case of LAD, FBD, and STL. With SCL, up to 32 input parameters can be programmed. Tags of data types INT, DINT, REAL are permissible as input values and additionally the data types TIME, DATE, and TOD with SCL. All input tags and the output tag must have the same data type or – with SCL – belong to the same class of data type (INT, DINT, REAL) and the output tag must have the "most significant" data type.

MIN Minimum selection

MIN selects the smallest of three numerical tag values in the case of LAD, FBD, and STL. With SCL, up to 32 input parameters can be programmed. Tags of data types INT, DINT, REAL are permissible as input values and additionally the data types TIME, DATE, and TOD with SCL. All input tags and the output tag must have the same data type or – with SCL – belong to the same class of data type (INT, DINT, REAL) and the output tag must have the "most significant" data type.

MUX (only for SCL)

Depending on the value of the K parameter, MUX selects a tag present at the INx or INELSE inputs and outputs it at the OUT parameter. Up to 32 input parameters (IN0 to IN31) can be programmed. The selection starts with K = 0 (corresponding to IN0) and ends with K = 31 (corresponding to IN31). If the value of K is outside the programmed inputs, the value at INELSE is transferred to the OUT output. If INELSE is not programmed in this case, the ENO tag and the ENO output are set to FALSE.

Tags of all elementary data types (except BOOL) and tags with data types DT, STRING, POINTER, and ANY are permissible as input values. All input tags and the output tag must have the same data type or belong to the same class of data type (BYTE, WORD, DWORD, or INT, DINT, REAL) and the output tag must have the "most significant" data type.

13.9 Functions for strings

A string can be processed with the following functions:

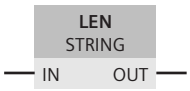
- LEN Outputs the length of a string (FC 21)
- CONCAT Combines two strings together (FC 2)
- LEFT Outputs the left part of a string (FC 20)
- RIGHT Outputs the right part of a string (FC 32)
- MID Outputs the middle part of a string (FC 26)
- DELETE Deletes part of a string (FC 4)
- INSERT Inserts characters into a string (FC 17)
- REPLACE Replaces characters in a string (FC 31)
- FIND Outputs the position of a searched character (FC 11)

The string functions are implemented using system functions. They are represented as EN/ENO box in the case of LAD and FBD and as a block call with the OUT parameter as the function value in the case of STL and SCL. The graphic representation of the functions is shown in Fig. 13.24.

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).

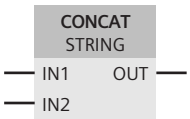
Please note that strings which you declare in the temporary local data cannot be assigned default values. In this case you must assign a defined value (can also be an empty string) to a STRING tag in the program before you use the STRING tag together with a function or block.

Processing of strings



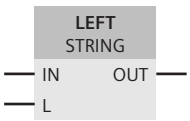
LEN Output length of a character string (FC 21)

LEN determines the length of the string in the IN parameter and outputs it in the OUT parameter.



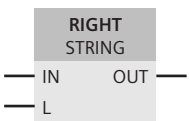
CONCAT Combine character stings together (FC 2)

CONCAT combines the strings present in the IN1 and IN2 parameters and outputs them as a single string in the OUT parameter.



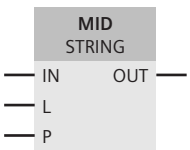
LEFT Output left part of character string (FC 20)

LEFT extracts the first characters (the number is specified in the L parameter) from the string present in the IN parameter and outputs them in the OUT parameter.



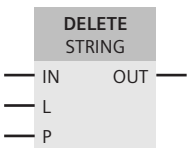
RIGHT Output right part of character string (FC 32)

RIGHT extracts the last characters (the number is specified in the L parameter) from the string present in the IN parameter and outputs them in the OUT parameter.



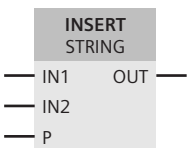
MID Output middle part of character string (FC 26)

MID removes a part of the string present in the IN parameter whose start position is specified in the P parameter and length in the L parameter and outputs it in the OUT parameter.



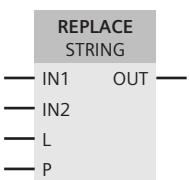
DELETE Delete part of character string (FC 4)

DELETE deletes a part of the string present in the IN parameter whose start position is specified in the P parameter and length in the L parameter and outputs the remainder "shifted together" in the OUT parameter.



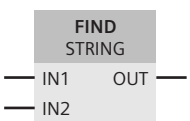
INSERT Insert character string (FC 17)

INSERT inserts the string present in the IN2 parameter into the string present in the IN1 parameter at the position specified in the P parameter and outputs the result in the OUT parameter.



REPLACE Replace part of character string (FC 31)

REPLACE replaces part of the string present in the string in the IN1 parameter by the string present in the IN2 parameter and outputs the result in the OUT parameter. The replaced part of the string commences at the position specified in the P parameter and has as many characters as specified in the L parameter.



FIND Find part of character string (FC 11)

FIND determines the position of the string present in the IN2 parameter in the string present in the IN1 parameter and outputs it in the OUT parameter.

Fig. 13.24 Description of the functions for processing of strings

LEN Length of a STRING tag

LEN outputs the current length of a string (number of valid characters) as a function value. An empty string has a length of zero. The maximum length is 254.

Except for incorrect parameter assignment, the function does not report any errors.

FIND Search in a STRING tag

FIND delivers the position of the second string (IN2) within the first string (IN1). The search starts on the left; the first occurrence of the string is reported. If the second string is not found in the first, zero is returned.

Except for incorrect parameter assignment, the function does not report any errors.

LEFT Left part of a STRING tag

LEFT delivers the first L characters of a string. If L is greater than the current length of the STRING tag, the input value is returned. With L = 0 and with an empty string as the input value, an empty string is returned.

If L is negative, an empty string is returned and the binary result BR and the ENO output are set to “0”.

RIGHT Right part of a STRING tag

RIGHT delivers the last L characters of a string. If L is greater than the current length of the STRING tag, the input value is returned. With L = 0 and with an empty string as the input value, an empty string is returned.

If L is negative, an empty string is returned and the binary result BR and the ENO output are set to “0”.

MID Middle part of a STRING tag

MID delivers the middle part of a string (L characters starting at the P character). If the sum of L and P exceeds the current length of the STRING tag, a string is returned from the P character up to the end of the input value.

In all other cases (P is outside the current length, P and/or L are equal to zero or negative), an empty string is output and the binary result BR and the ENO output are set to “0”.

CONCAT Combination of two STRING tags

CONCAT combines two STRING tags into one string.

If the result string is longer than the tag at the output parameter, it is limited to the maximum set length and the binary result BR and the ENO output are set to “0”.

INSERT Insert in a STRING tag

INSERT inserts the string at the IN2 parameter into the string at the IN1 parameter after the character at position P. If P is equal to zero, the second string is inserted before the first string. If P is greater than the current length of the first string, the second string is appended to the first one.

If P is negative, an empty string is output and the binary result BR and the ENO output are set to “0”. The binary result and the ENO output are also set to “0” if the result string is longer than the tag at the output parameter; in this case the result string is limited to the maximum set length.

DELETE Delete in a STRING tag

DELETE deletes L characters in a string starting at the P character. If L and/or P are equal to zero or if P is greater than the current length of the input string, the input string is returned. If the sum of L and P is greater than the input string, the string is deleted up to the end.

If L and/or P is negative, an empty string is output and the binary result BR and the ENO output are set to “0”.

REPLACE Replace in a STRING tag

REPLACE replaces L characters in the first string (IN1) starting at the character at position P by the second string (IN2). If L is equal to zero, the first string is returned. If P is equal to zero or one, the string is replaced from the first character (inclusive). If P is outside the first string, the second string is appended to the first string.

If L and/or P is negative, an empty string is output and the binary result BR and the ENO output are set to “0”. The binary result BR and the ENO output are also set to “0” if the result string is longer than the tag at the output parameter; in this case the result string is limited to the maximum set length.

14 Program flow control

This chapter describes the functions for controlling program execution, independent of the programming language as far as possible. The Chapters 7 “Ladder logic LAD” on page 249, 8 “Function block diagram FBD” on page 282, 9 “Statement list STL” on page 315, and 10 “Structured Control Language SCL” on page 363 describe how you can program the functions using the individual programming languages and what special features exist.

The status bits (LAD, FBD, STL) and the ENO tag (SCL) provide information on the result of an arithmetic function or on any errors, for example exceeding a numerical range. You can directly integrate the signal state of the status bits or the ENO tag into your program using binary logic operations or program branches.

The jump functions permit program branching depending on the status bits, the result of logic operation, or the binary result. With STL, you can execute the jumps with a calculated jump width or you can easily implement program loops. The control statements which are only present with SCL for controlling program execution are described in Chapter 10.6.3 “Control statements” on page 385.

The block functions are used to structure the user program. The organization blocks are the interface to the CPU's operating system. The function blocks and functions represent individual programs sections for a complete function. Function blocks and functions can be parameterized and thus used repeatedly with different tags.

A further possibility for influencing program execution is provided by the master control relay (MCR). Originally developed for contactor controls, LAD, FBD, and STL provide software emulation of this program control possibility.

Table 14.1 Status bits with LAD, FBD, and STL; assignment of status word

Bit	Binary flags		Bit	Digital flags	
0	/FC	First scan	4	OS	Stored overflow
1	RLO	Result of logic operation	5	OV	Overflow
2	STA	Status	6	CC0	Condition code bit CC0
3	OR	Status bit OR	7	CC1	Condition code bit CC1
8	BR	Binary result			

14.1 Status bits

The status bits are binary flags (condition code bits) which the CPU uses to control the binary logic operations and sets during digital processing. You can also scan these status bits or specifically influence them with the programming languages LAD, FBD, and STL.

SCL uses the ENO tag for the error scan (see Chapter 10.6.1 “Working with the ENO tag” on page 382).

14.1.1 Description of the status bits

Table 14.1 shows the available status bits. The first column shows the bit number in the status word. The CPU uses the binary flags for controlling the binary logic operations. The digital flags mainly indicate the results of arithmetic and math functions.

The programming significance of the status bits becomes important above all with the programming language STL. How the processing of binary statements influences the status bits is described in Chapter 14.1.2 “Controlling the status bits” on page 533.

First scan /FC

The /FC status bit steers the binary logic operation in a logic control. An operation step always starts with /FC = “0” and a binary scan statement, the first input bit scan. The first input bit scan with LAD corresponds to the first contact in a network, with FBD to the first binary function input, and with STL to the first scan operation following a conditional operation. The first input bit scan sets /FC = “1”.

An operation step ends with a conditional operation dependent on the result of logic operation, for example a binary value assignment, a jump depending on the result of logic operation, or a block change. These set /FC = “0”. The next binary scan is then the start of a new logic operation.

Result of logic operation RLO

The status bit RLO is the intermediate memory for binary logic operations. During the first input bit scan, the CPU transfers the scan result to the result of logic operation, which is mapped in the status bit RLO. With each subsequent scan, the CPU links the scan result to the saved result of logic operation, and again saves the result in the status bit RLO.

The result of logic operation is used to control memory, timer, and counter functions and to execute certain jump functions. The result of logic operation corresponds with LAD to the current flowing in the current path (RLO = “1” is equivalent to “current flowing”).

Status STA

The status bit STA corresponds to the signal state of the addressed binary operand or – with STL – to the scan result with the binary logic operations.

With the memory functions (set, reset, assign), the value of STA is the same as the written value or – if no write operation takes place, e.g. with RLO = “0” or MCR active – corresponds to the value of the addressed (and unmodified) binary operand.

With edge evaluations FP or FN, the value of the RLO prior to the edge evaluation is stored in STA. All other binary statements/functions set STA = “1”; and also the binary flag-dependent jumps JC, JCN, JBI, JNBI with STL (exception: CLR sets STA = “0”).

The status bit STA does not affect program execution. It is displayed with the test functions of the programming device, e.g. with the program status, so that you can use it to trace logic sequences or for troubleshooting.

Status bit OR

The status bit OR saves the result of a fulfilled AND logic operation and indicates to a subsequent OR logic operation that the result is already available (in association with the O statement in an AND before OR logic operation). All other bit-processing statements (binary functions) reset the status bit OR. An example is shown in Chapter 14.1.2 “Controlling the status bits” on page 533.

Overflow OV

The status bit OV indicates a numerical range overflow or the use of invalid floating-point numbers. The following functions influence the status bit OV: arithmetic functions, math functions, some conversion functions, and the comparison functions with data type REAL.

You can evaluate the status bit OV using scan statements or – with STL – using the jump statement JO.

Stored overflow OS

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 are therefore provided with the opportunity to subsequently evaluate a numerical range overflow or an operation with an invalid floating-point number in your program.

You can evaluate the status bit OS using scan statements or – with STL – using the jump statement JOS. JOS or a block change resets the status bit OS.

Status bits CC0 and CC1

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 jump functions and scan statements (see later in this chapter).

Binary result BR

The binary result BR is an additional memory for the result of logic operation. In the case of LAD, FBD, and SCL, it supports implementation of the EN/ENO mechanism with block calls.

You can also set or reset the status bit BR yourself and evaluate it with binary scans or – with STL – with jump statements. In the case of SCL, you scan the binary result using the ENO tag.

Status word STW

The status word contains all status bits. With STL, you can load it into accumulator 1 or write it with a value from accumulator 1.

```
L STW       //Load status word
            //...
T STW       //Transfer to status word
```

The assignment of the status word with the status bits can be found in Table 14.1 on page 530.

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.

Note that a CPU 300 does not load the status bits /FC, STA, and OR into the accumulator; the accumulator contains “0” at these locations. You cannot influence these bits using the status word either.

14.1.2 Controlling the status bits

The top part of Table 14.2 shows controlling of the binary flags using the example of an operation step, and the bottom part shows setting of the digital flags for arithmetic functions.

The following digital functions influence the status bits CC0, CC1, OV, and OS:

- Comparison functions (see Chapter 13.3 “Comparison functions” on page 487)
- Arithmetic functions (see Chapter 13.4 “Arithmetic functions” on page 491)

Table 14.2 Example of influencing the status bits with STL

STL statements		Binary flags: /FC	RLO	STA	OR	Remark	
...							
=	%M10.0	0	x	x	0		
A	%I4.0	1	1	1	0	%I4.0 is "1"	The part shaded in gray is an operation step
AN	%I4.1	1	1	0	0	%I4.1 is "0"	
O		1	1	1	1		
O	%I4.2	1	1	0	0	%I4.2 is "0"	
ON	%I4.3	1	1	0	0	%I4.3 is "0"	
=	%Q8.0	0	1	1	0	%Q8.0 to "1"	
R	%Q8.1	0	1	0	0	%Q8.1 to "0"	
S	%Q8.2	0	1	1	0	%Q8.2 to "1"	
A	%I5.0	1	x	x	0		
...							

STL statements		Digital flags: CC0	CC1	OV	OS	Remark	
...							
T	%MW10	x	x	x	x		
L	+12	x	x	x	x		
L	+15	x	x	x	x		
-I		1	0	0	0	Result negative	
L	+20000	1	0	0	0		
*I		1	0	1	1	Overflow	OV and OS to "1"
L	+20	1	1	1	1		
+I		0	1	0	1	OV becomes "0"	OS remains "1"
T	%MW20	0	1	0	1		
L	%MW40	0	1	0	1		
...							

- Math functions (see Chapter 13.5 "Math functions" on page 496)
- Conversion functions (see Chapter 13.6 "Conversion functions" on page 500)
- Shift functions (see Chapter 13.7 "Shift functions" on page 514)
- Word logic operation (see Chapter 13.8.1 "Word logic operations" on page 519)

The specified chapters describe how these functions influence the status bits.

14.1.3 Setting and resetting the result of logic operation

With STL, the **SET** statement sets the result of logic operation to signal state "1" and also the status bit STA to signal state "1".

With STL, the **CLR** statement sets the result of logic operation to signal state "0" and also the status bit STA to signal state "0".

Setting and resetting the result of logic operation (STL)

With STL, SET and CLR control the result of the logic operation without conditions.

STL

```
    (RLO)
SET
    (RLO = "1")
```

STL uses the SET operation to set the result of the logic operation to signal state "1". SET starts a new operation step.

```
    (RLO)
CLR
    (RLO = "0")
```

STL uses the CLR operation to reset the result of the logic operation to signal state "0". CLR starts a new operation step.

Fig. 14.1 Setting and resetting the result of logic operation

Both statements are executed independent of conditions. SET and CLR also reset the status bits OR and /FC so that a new logic operation starts with the next scan following SET or CLR (Fig. 14.1).

You can use SET to program absolute setting or resetting of a binary operand, and CLR to reset e.g. edge trigger flags:

```
SET
S    %M8.0  //Bit memory is set
R    %M8.1  //Bit memory is reset
CLR
CU   %C1    //Reset edge memory bit for "Count up"
```

Direct setting and resetting of the result of logic operation is also useful in conjunction with SIMATIC timer and counter functions. To start a timer or counter function, you require a change in the RLO from "0" to "1" (note that you also require a positive signal edge for enabling). In program sections with predominantly digital logic operations, the RLO is generally not defined, e.g. following the jump functions for evaluating the digital flags. Here you can use SET and CLR for defined setting or resetting of the RLO or for programming a change in RLO.

14.1.4 Controlling the binary result

The binary result BR is controlled using the following statements or functions:

- Control binary result with SAVE
  With SAVE you can save the result of logic operation (RLO) in the binary result (BR). SAVE transfers the signal state from status bit RLO to status bit BR. SAVE works independent of conditions and does not influence any further status bits (Fig. 14.2).

- Control binary result with JCB and JNB
  The jump functions JCB and JNB with STL also influence the binary result. JCB sets the BR to signal state “1”, JNB to “0”.
- Control binary result at block end
  The RET coil with LAD or the RET box with FBD save the result of logic operation in the binary result.
- Influencing of binary result through program execution
  A block call with LAD and FBD influences the binary result through the program execution in order to control the enable output ENO (Fig. 14.3).

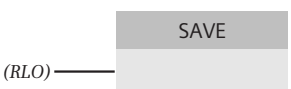
Assign binary result

In the programming languages LAD, FBD and STL, SAVE saves the result of logic operation (RLO) in the binary result (BR).

LAD

LAD uses the SAVE coil for assignment of the result of logic operation (of the "current flow") to the binary result.

FBD



In FBD, the assignment of the result of logic operation to the binary result is represented by the SAVE box.

STL

```
    (RLO)
SAVE
    (RLO)
```

The SAVE statement saves the result of logic operation in the binary result.

Fig. 14.2 Assign binary result with SAVE

<table>
<tr><td colspan="6">Is ENO connected?</td></tr>
<tr><td colspan="3">YES</td><td colspan="3">NO</td></tr>
<tr><td colspan="2">Is EN connected?</td><td rowspan="2"></td><td colspan="2">Is EN connected?</td><td rowspan="2"></td></tr>
<tr><td colspan="2">YES</td><td colspan="2">YES</td></tr>
<tr><td colspan="2">Is EN = “1”?</td><td rowspan="2"></td><td colspan="2">Is EN = “1”?</td><td rowspan="2"></td></tr>
<tr><td>YES</td><td>NO</td><td>YES</td><td>NO</td></tr>
<tr><td>BR is set corresponding to the function</td><td>BR = “0”</td><td>BR is set corresponding to the function</td><td>BR = “1”</td><td>BR = “0”</td><td>BR is not affected</td></tr>
</table>

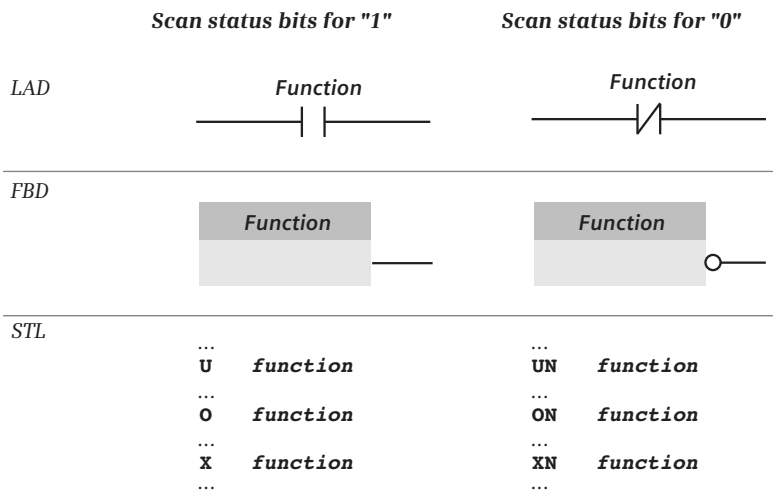
Fig. 14.3 General schema for setting the binary result

If the enable output ENO is connected, its signal state corresponds to that of the binary result. In certain cases (“BR is set corresponding to the function”), the executed function sets the binary result as follows:

- BR := “1”
  With MOVE, with the shift functions, and with the word logic operations
- BR := OV
  With the arithmetic and math functions
- BR := OV or “1”
  With the conversion functions
- BR := BR of the called block
  With block calls

Scan status bits A0, A1, OV, OS and BR

In the programming languages LAD, FBD and STL, the status bits A0, A1, OV, OS and BR can be scanned and the result of scan gated further.



Function:
The scan for signal state "1" delivers the result of scan "1" with:

>0	Result is greater than zero	**>=0**	Result is greater than or equal to zero
<0	Result is less than zero	**<=0**	Result is less than or equal to zero
==0	Result is equal to zero	**<>0**	Result is not equal to zero
UO	Result is invalid (unordered)	**OV**	Range overflow
OS	Retentive overflow	**BR**	Binary result

Fig. 14.4 Scan status bits

14.1.5 Evaluating the status bits

Evaluation with binary scans

Fig. 14.4 shows the scan functions for the status bits with LAD, FBD, and STL. The combination is scanned in the case of the status bits CC0 and CC1, and this delivers the relationships: equal to, not equal to, greater than, less than, and invalid (see also Table 14.3). A specific combination can fulfill several relationships. Example: The assignment of the status bits CC0 = “0” and CC1 = “0” delivers the relationships “Equal to zero”, “Greater than or equal to zero”, and “Less than or equal to zero”.

The overflow flags and the binary result are scanned directly.

The scans can be executed for signal state “1” or “0”. Example with STL: If CC0 = “0” and CC1 = “1” (with the meaning “Result is positive”), the A >0 statement delivers the scan result “1” and the AN >0 statement the scan result “0”.

Evaluation with jump functions

With the programming language STL you can evaluate the status bits RLO and BR, all combinations of CC0 and CC1, and the status bits OV and OS with corresponding jump functions (Table 14.3). A detailed description of the jump functions is provided in the next chapter.

Table 14.3 Evaluation of status bits with STL by scanning and with jump functions

RLO	BR	CC0	CC1	OV	OS	Scan function	Jump function	Result is
"1"	–	–	–	–	–	–	JC, JCB	–
"0"	–	–	–	–	–	–	JCN, JNB	–
–	"1"	–	–	–	–	BR to “1”	JBI	–
–	"0"	–	–	–	–	BR to “0”	JNBI	–
–	–	0	0	–	–	==0 to “1” >=0 to “1” <=0 to “1”	JZ JPZ JMZ	Equal to zero Greater than or equal to zero Less than or equal to zero
–	–	0	1	–	–	<>0 to “1” >0 to “1” >=0 to “1”	JN JP JPZ	Not equal to zero Greater than zero Greater than or equal to zero
–	–	1	0	–	–	<>0 to “1” <0 to “1” <=0 to “1”	JN JM JMZ	Not equal to zero Less than zero Less than or equal to zero
–	–	1	1	–	–	UO to “1”	JUO	Invalid
–	–	–	–	1	–	OV to “1”	JO	Overflow
–	–	–	–	–	1	OS to “1”	JOS	Retentive overflow

14.2 Jump functions

14.2.1 Introduction

You can use jump functions to 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 function as the jump destination. A jump label can consist of up to 128 letters, numbers, and underscore characters.

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 14.4 shows an overview of the types of jump functions.

Table 14.4 Types of jump functions

Jump functions	Present with			
	LAD	FBD	STL	SCL
Absolute jump	X	X	X	X
Jump depending on RLO	X	X	X	–
Jump depending on status bits	–	–	X	–
Jump list	–	–	X	1)
Loop jump	–	–	X	1)

1) See Chapter 10.6.3 “Control statements” on page 385

14.2.2 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. 14.5 shows the implementation of the jump function in the various programming languages.

Absolute jump function JMP (LAD and FBD)

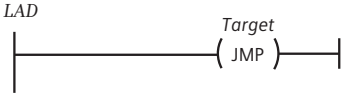
The jump functions consist of the jump statement (coil or box) and a jump label. The jump label identifies the entry point in the block at which program execution is continued when the jump function has been processed.

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.

Absolute jump

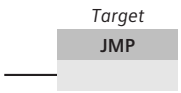
The absolute jump is executed independent of conditions during processing.

LAD



If the JMP coil (jump with RLO = "1") is connected to the left busbar, the jump function is always executed during processing.

FBD



If the input is not connected on the JMP box (jump with RLO = "1"), the jump function is always executed during processing.

STL

```
SPA   target
```

The SPA absolute jump is always executed during processing.

SCL

```
GOTO  target;
```

The GOTO statement is always executed during processing.

Fig. 14.5 Absolute jump independent of conditions

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
```

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”
```

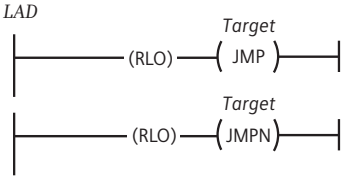
14.2.3 Conditional jump functions

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. 14.6 shows the implementation of the conditional jump function in the various programming languages.

Jump depending on result of logic operation

The conditional jump is executed depending on the result of logic operation (RLO).

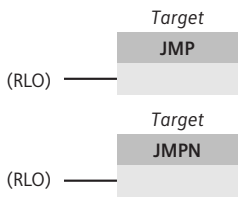
LAD



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.

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.

STL

```
(RLO)
SPB   target
```

The SPB conditional jump is executed with RLO = "1". With RLO = "0", the next statement is processed.

```
(RLO)
SPBN  target
```

The SPBN conditional jump is executed with RLO = "0". With RLO = "1", the next statement is processed.

```
(RLO)
SPBB  target
```

The SPBB conditional jump is executed with RLO = "1". With RLO = "0", the next statement is processed. The RLO is transferred to the binary result in both cases.

```
(RLO)
SPBNB target
```

The SPBNB conditional jump is executed with RLO = "0". With RLO = "1", the next statement is processed. The RLO is transferred to the binary result in both cases.

Fig. 14.6 Conditional jump depending on result of logic operation

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 instruction (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.

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.

14.2.4 Jump functions depending on status bits

STL provides jump functions with which the assignment of the status bits (BR, CC0, CC1, OV, OS) can be evaluated. An overview of the combinations which can be scanned and the jump functions is provided in Table 14.3 on Page 538.

Scanning of the status bits is possible in LAD and FBD, and it is possible to program a conditional jump function JMP or JMPN dependent on these (see Chapter 14.1.5 “Evaluating the status bits” on page 538).

Jump functions JBI and JNBI dependent on binary result (STL)

The jump function JBI is only executed if the binary result is “1” when this function is processed. The jump is not executed if the binary result is “0” and execution of the program is continued with the following statement.

The jump function JNBI is only executed if the binary result is “0” when this function is processed. The jump is not executed if the binary result is “1” and execution of the program is continued with the following statement.

JBI and JNBI terminate a binary logic operation; a new logic operation starts following the jump function or at the jump destination. The RLO is retained and can be evaluated using a memory function following the jump function.

Jump functions JZ, JN, JP, JPZ, JM, JMZ, and JUO (STL)

The jump functions dependent on the status bits CC0 and CC1 do not change any status bits. The result of logic operation is “carried over” with the jump and can be linked further (no change in /FC).

The jump function JZ is only executed if the status bits are CC0 = “0” and CC1 = “0”. This is the case if:

- The content of accumulator 1 is zero following an arithmetic or math function
- The content of accumulator 2 is equal to the content of accumulator 1 with a comparison function
- The content of accumulator 1 is zero following a word logic operation
- The value of the last shifted-out bit is “0” following a shift function

The jump function JN is only executed if the status bits CC0 and CC1 have different signal states. This is the case if:

- The content of accumulator 1 is not zero following an arithmetic or math function
- The content of accumulator 2 is not equal to the content of accumulator 1 with a comparison function
- The content of accumulator 1 is not zero following a word logic operation
- The value of the last shifted-out bit is “1” following a shift function

The jump function JP is only executed if the status bits are CC0 = “0” and CC1 = “1”. This is the case if:

- The content of accumulator 1 is within the permissible positive numerical range following an arithmetic or math function (you scan for violation of the numerical range using JO or JOS)
- The content of accumulator 2 is greater than the content of accumulator 1 with a comparison function
- The content of accumulator 1 is not zero following a word logic operation
- The value of the last shifted-out bit is “1” following a shift function

The jump function JPZ is only executed if the status bit CC0 = “0”. This is the case if:

- The content of accumulator 1 is within the permissible positive numerical range or zero following an arithmetic or math function (you scan for violation of the numerical range using JO or JOS)
- The content of accumulator 2 is greater than or equal to the content of accumulator 1 with a comparison function
- Following every word logic operation
- Following every shift function

The jump function JM is only executed if the status bits are CC0 = “1” and CC1 = “0”. This is the case if:

- The content of accumulator 1 is within the permissible negative numerical range following an arithmetic or math function (you scan for violation of the numerical range using JO or JOS)
- The content of accumulator 2 is less than the content of accumulator 1 with a comparison function

The jump function JMZ is only executed if the status bit CC1 = “0”. This is the case if:

- The content of accumulator 1 is within the permissible negative numerical range or zero following an arithmetic or math function (you scan for violation of the numerical range using JO or JOS)
- The content of accumulator 2 is less than or equal to the content of accumulator 1 with a comparison function

The jump function JUO is only executed if the status bits are CC0 = “1” and CC1 = “1”. This is the case if:

- Division is by zero with an arithmetic function
- An invalid REAL number has been specified as input value or is produced as result

Jump functions JO and JOS (STL)

A program branch can be executed depending on the status bits OV and OS. In this case you scan whether the result of a calculation is still within the permissible numerical range.

The jump function JO (jump if overflow) is only executed if the status bit OV is set to “1”. This is the case if the permissible numerical range has been left following execution of an operation. The following functions can set the status bit OV:

- Arithmetic functions
- Math functions
- Generation of two's complement
- Comparison functions with REAL numbers
- Conversion functions INT or DINT to BCD and REAL to DINT

If the status bit OV = “0”, JO continues execution of the program with the next statement.

In the case of a chain calculation with several 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 in order to evaluate a possible numerical range overflow at the end of the chain calculation.

The jump function JOS (jump if overflow stored) is only executed if the status bit OS is set to “1”. This is always the case is a numerical range overflow sets the status bit OV (see above). In contrast to OV, OS remains set if the result is subsequently in the permissible numerical range.

The following functions reset OS again:

- Block call and block end
- Jump if stored overflow JOS

If the status bit OS = “0”, JOS continues execution of the program with the next statement.

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. If an organization block is terminated, a branch is made to the operating system.

Fig. 14.7 shows the representation of the block end functions in the various programming languages.

14.3.1 Block end function RET (LAD and FBD)

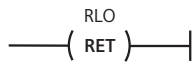
You can use the block end function RET to prematurely terminate processing in a block depending on the result of logic operation. The block end function is represented as an RET coil or RET box which requires a preceding logic operation. The block is left if the preceding logic operation is fulfilled, or processing is continued in the next network if the preceding logic operation is not fulfilled. The RET coil/box must only terminate a current path or logic operation on its own.

The RET coil/box simultaneously saves the result of logic operation in the binary result BR, irrespective of whether the preceding logic operation was fulfilled or not. The binary result is decisive for controlling the enable output ENO on the call box. If a block is left prematurely with the RET coil/box, the enable output ENO of the call is always occupied by signal state “1”. If the RET coil/box is the last statement in the block, the RLO currently present when the block is left is transferred to the enable output ENO.

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.

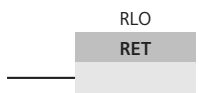
LAD



The RET coil is present alone in a network as termination of a preceding logic operation. The block is left if the preceding logic operation is fulfilled.

The RET coil saves the RLO in the binary result.

FBD



The RET box is present alone in a network as termination of a preceding logic operation. The block is left if the preceding logic operation is fulfilled.

The RET box saves the RLO in the binary result.

STL

```
BEC   //Conditional block end
BEU   //Unconditional block end
BE    //Block end
```

BEC terminates block processing when RLO = "1".

BEU and BE terminate block processing independent of conditions. BE is the last statement in a block, and can also be omitted.

SCL

```
RETURN; //Block end
```

RETURN terminates block processing independent of conditions.

Fig. 14.7 Block end functions

14.3.2 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 control processor sets the RLO to “1” and processes the statement following BEC. 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 always the last statement of a block. Programming of BE is optional.

14.3.3 RETURN statement (SCL)

RETURN leaves the currently processed block without conditions.

RETURN transfers the signal state of the ENO tag to the enable output ENO of the left block. Programming of RETURN at the end of the block is optional.

14.4 Calling of code blocks

14.4.1 General information on block calls

If a function block (FB) or a function (FC) is to be processed, it must be “called”.

With LAD and FBD, the block call consists of the call box which contains the address or name of the called block, the enable input EN, the enable output ENO, and the parameter list. With STL and SCL, the address or name of the block is specified in the call operation, followed by the parameter list.

Following processing of the call function, the CPU continues program execution in the called block. The block is processed up to a block end function or up to its end. The CPU then returns to the calling block and continues processing of this block after the call function.

An organization block (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.

The block parameters are the data interface to the called block. You should avoid data transfer using internal registers (e.g. accumulators, address registers, RLO) since the contents of these registers can be changed when changing the block (by “hidden” statements sent by the program editor). You – and also the program editor – can only use the binary result BR in order to control the enable output ENO of the call function.

Chapter 5.2 “Creating a user program” on page 151 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.

The calls of code blocks are represented by the block call box in the case of LAD and FBD, by the CALL statement in the case of STL, and by the block name in the case of SCL. Functions for changing to another block without parameters are additionally possible with LAD, FBD, and STL for special applications.

14.4.2 Calling a function (FC)

The calls of a function (FC) and a system function (SFC) are handled in the same manner. Fig. 14.8 shows the block call for a function and for a system function. All block parameters must be supplied with actual parameters.

FC and SFC calls with LAD and FBD

You use the call box with LAD and FBD to call a function or a system function. 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.

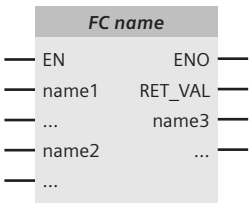
Calling a function (FC) or a system function (SFC)

A **function FC** represents a program section (a block) without own data. A **system function SFC** is an FC block whose program code is present in the operating system and whose call is present in the user program.

A function (FC) has a function value (return value) with the preset name RET_VAL (this name can be modified, 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.

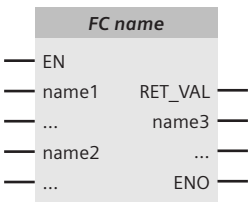
All block parameters must be supplied with actual parameters when called.

LAD



An FC or SFC block is called in LAD by an EN/ENO box. The input and in/out parameters are present on the left-hand side of the call box in the order of their declaration, and the output parameters on the right-hand side. If the function value is "activated", it is represented as the first output parameter.

FBD



An FC or SFC block is called in FBD by an EN/ENO box. The input and in/out parameters are present on the left-hand side of the call box in the order of their declaration, and the output parameters on the right-hand side. If the function value is "activated", it is represented as the first output parameter.

STL

```
CALL  "FC name"
  In1   := …
  In2   := …
  Out1  := …
  Out2  := …

CALL  "FC name"
  In1     := …
  In2     := …
  RET_VAL := …
  Out1    := …
  Out2    := …
```

An FC or SFC block is called in STL by the CALL operation. 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.

SCL

```
"FC name"(
  In1 := … ,
  In2 := … ,
  Out1 => … ,
  Out2 => … );

#Tag     := "FC name"(
   In1   := … ,
   In2   := … ,
   Out1  => … ,
   Out2  => … );
```

An FC or SFC block is called in SCL by its name. This is followed by the parameter list in parentheses. The parameters are specified in the order of their declaration, each separated by a comma.

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 , for example, to a tag or be used in an expression.

Fig. 14.8 Call functions for a function (FC) and a system function (SFC)

FC and SFC calls with STL

You use the CALL operation with STL to call a function or a system function. CALL is an absolute call, i.e. the specified block is always called and processed independent of conditions.

Note that the CALL statement can change the contents of the data block registers DB and DI, the contents of the address registers AR1 and AR2, and the contents of the accumulators 1 and 2. The status bits /FC, OS, and OR are set to “0”.

FC and SFC calls with SCL

You call a function (FC) or a system function (SFC) which does not have a function value using its absolute address or 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) or system function (SFC) 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 consists of the absolute address or the name of the block followed by the parameter list in round brackets. The parameter sequence is defined by the declaration. All parameters must be supplied with values.

The implicitly defined enable input EN cannot be used with a function with function value since the output parameters may have an undefined assignment under certain circumstances if EN is FALSE.

If you wish to use the implicitly defined enable output ENO, add it to the parameter list as the last parameter.

14.4.3 Calling a function block (FB)

The calls of a function block (FB) and a system function block (SFB) are handled in the same manner. Both block types can be called as single instance or local instance. Fig. 14.9 shows the block call for a function block as single instance and as local instance.

With function blocks, not all block parameters have to be supplied during the call. In/out parameters with complex data type and block parameters with parameter type should be assigned default values with the correct syntax or at least written during the first call. Writing of the other block parameters is optional.

FB and SFB calls with LAD and FBD

You use the call box with LAD and FBD to call a function block or a system 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.

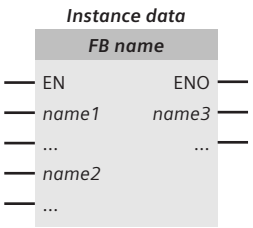
Calling a function block (FB) and a system function block (SFB)

A **function block FB** represents a program section (a block) with its own data which is present in an instance data block. A **system function block (SFB)** is a function block whose program code is present in the operating system and whose call and instance data are present in the user program.

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 “multi-instance”), one refers to a “local instance”.

In/out parameters with complex data type and block parameters with parameter type should be assigned default values with the correct syntax or at least written during the first call. Writing of the other block parameters is optional.

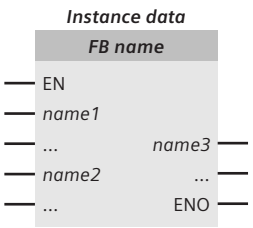
LAD



An FB or SFB block is called in LAD by the EN/ENO box. The input and in/out parameters are present on the left-hand side of the call box and the output parameters on the right-hand side, in the order of their declaration in each case.

The name of the call instance is shown above the call box. In the case of a single instance, this is the instance data block. In the case of a local instance, this is the instance name in the static local data of the calling function block.

FBD



An FB or SFB block is called in FBD by the EN/ENO box. The input and in/out parameters are present on the left-hand side of the call box and the output parameters on the right-hand 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.

STL

```
CALL "FB name","DB name"
  In1   := …
  In2   := …
  Out1  := …
  Out2  := …

CALL  #Instance name
  In1    := …
  In2    := …
  Out1    := …
  Out2    := …
```

The block name is specified when calling as a single instance, followed by a comma and the name of the instance data block. The instance name is specified when calling as local instance.

The next lines contain list of block parameters, in each case in the order of their declaration. Only the block parameters which are written need to be listed.

SCL

```
"DB name"(
  In1  := … ,
  In2  := … ,
  Out1 => … ,
  Out2 => … );

#Instance name(
  In1  := … ,
  In2  := … ,
  Out1 => … ,
  Out2 => … );
```

The name of the instance data block is specified when calling as a single instance. The instance name is specified when calling as local instance.

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 which are written need to be listed.

Fig. 14.9 Call functions for a function block (FB) and a system function block (SFB)

CALL statement with STL

You use the CALL operation with STL to call a function block or a system function block. CALL is an absolute call, i.e. the specified block is always called and processed independent of conditions. When called as single instance, the block name is followed by the name of the instance data block, separated by a dot. Specify the instance name when calling as local instance. Only specify the block parameters in the parameter list which you supply.

Note that the CALL statement can change the contents of the data block registers DB and DI, the contents of the address registers AR1 and AR2, and the contents of the accumulators 1 and 2. The status bits /FC, OS, and OR are set to “0”.

FB and SFB calls with SCL

With SCL you call a function block or a system function block as single instance with the block name and the instance data block or as local instance with the instance name. Specify the name of the instance data block when calling as single instance. Specify the instance name when calling as local instance. Only specify the block parameters in the parameter list which you supply.

If you wish to use the implicitly defined enable input EN, add it to the parameter list as the first parameter.

If you wish to use the implicitly defined enable output ENO, add it to the parameter list as the last parameter.

14.4.4 Change to a block without block parameter

Program execution can be continued in another block by using the functions for block change (not to be confused with the functions for block call). This block change is carried out without the program extension required for supplying the block parameters and for saving the register contents. Therefore the block selected by the change cannot and must not have any block parameters. Use of these functions is therefore limited to special applications. (Fig. 14.10).

The functions for block change are present in LAD, FBD, and STL. They have the following properties:

- Influencing of condition codes

  The status bit OS is reset when changing the block, the status bits CC0, CC1, and OV are not influenced. The status bit /FC is reset, i.e. a new logic operation starts with the first scan statement in the new block and following the block change statement.

- Settings of the master control relay

  The MCR dependency is deactivated upon a block change. The MCR is deactivated in the new block independent of whether it was activated or deactivated prior to the block change. When the block is left, the MCR dependency is reset to its state prior to the block change.

Change to a block without parameter

When executing the CALL function (with LAD and FBD) or the UC and CC operations (with STL), the program execution changes to the specified block. This block must not have any block parameters or with function blocks an instance data block, and thus no static local data.

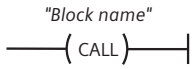
LAD		The CALL coil initiates the change to the block whose name is present above the coil. The change is carried out depending on a fulfilled preceding logic operation.
FBD		The CALL box initiates the change to the block whose name is present above the coil. The change is carried out depending on a fulfilled preceding logic operation.
STL	`UC "Block name"`	The UC operation initiates the change to the specified block independent of conditions.
	`CC "Block name"`	The CC operation initiates the change to the specified block depending on the result of logic operation.

Fig. 14.10 Change to a block without parameters with LAD, FBD, and STL

▷ Assignment of data block registers

The change to a block saves the data block registers. Following completion of block processing, their contents are restored. The global and instance data blocks present prior to the block change are also open following the change. If no data block was open prior to the block change (e.g. no instance data block in OB 1), no data block is open following the change either, irrespective of the data blocks open in the new block.

▷ Assignment of accumulators and address registers (STL)

The contents of accumulators and address registers are not changed upon a block change with UC or CC.

▷ Response of binary nesting stack (STL)

You can also change to a code block within a binary nesting expression. The current stack depth of the binary nesting stack is not changed upon a block change. The possible nesting stack depth in a block which is changed to within binary nesting is therefore the difference between the maximum possible nesting depth and the current nesting depth when changing the block.

Block change with LAD

With LAD, you use the CALL coil for a block change. If the CALL coil is connected directly to the left-hand power rail or if the preceding logic operation is fulfilled, a change is made to the block named above the CALL coil. Only one CALL coil is permissible per network.

You can use the CALL coil to change to a function (FC) or system function (SFC) that has no block parameters. You can also specify a block parameter of type BLOCK_FC as the operand on the CALL coil.

Block change with FBD

With FBD, you use the CALL box for a block change. If the CALL box does not have a preceding logic operation or if an existing preceding logic operation is fulfilled, a change is made to the block named above the CALL box. Only one single CALL box is permissible per network.

You can use the CALL box to change to a function (FC) or system function (SFC) that has no block parameters. You can also specify a block parameter of type BLOCK_FC as the operand on the CALL box.

Block change with STL

With STL, you use the UC operation for an absolute block change and the CC operation for a block change depending on the result of logic operation. The RLO is set to “1” if CC is processed with RLO = “0”, and the statement following CC is processed.

All blocks which can be called and which do not have their own parameters are permissible as operands for UC and CC.

You can also change to a block present as block parameter of type BLOCK_FC or BLOCK_FB. Memory-indirect addressing of the blocks is possible when changing using the UC and CC operations.

14.5 Data block functions

The data tags are saved in the data blocks. In order to access the data tags, a data block must first be “opened” (selected). During complete addressing with specification of the data block, the program editor generates the corresponding statement which is not visible to you as the user. During partial addressing where you only specify the data tag, you must ensure that the “correct” data block has first been opened.

Opening of a data block by the user is possible with LAD, FBD, and STL. SCL only supports complete addressing of data tags and can therefore relinquish statements for manipulation of data block registers by the user.

Every CPU 300 has two data block registers. These registers contain the numbers of the current data blocks. These are the data blocks whose operands are currently being used for processing. The program editor preferably uses the first data block register for access to global data blocks, and the second data block register for access to instance data blocks. Therefore these registers are also named “Global data block registers” (abbreviated to: DB registers) and “Instance data block registers” (abbreviated to: DI registers).

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). Data operands present in a data block opened using the DB register can be addressed – partially addressed – using “DB”, for example %DBW2. Data operands present in a data block opened using the DI register can be addressed using “DI”, for example %DIW2.

An opened data block remains “valid” until another data block is opened. This may take place using the program editor and is not visible to you as the user. This results in limitations in partial addressing of data tags. For more details on data addressing, refer to Chapter 4.2.2 “Absolute addressing of tags” on page 97.

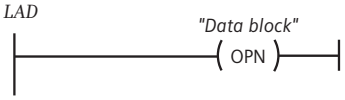
You can also generate data blocks during runtime and thus flexibly adapt the memory space for data to the data volume. It is additionally possible to save data blocks only in the load memory, which can be designed much larger than the work memory.

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.

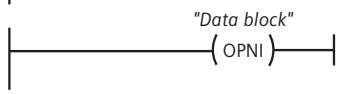
Open data block

Opening of a data block is required in order to address data operands. In the case of "complete addressing", the program editor is responsible for opening the data block by means of corresponding instructions "in the background". If one uses the "partial addressing" possible with the programming languages LAD, FBD and STL, one must open the "correct" data block oneself.

LAD

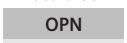


The OPN coil opens the data block via the DB register. It does not have a preceding logic operation.



The OPNI coil opens the data block via the DI register. It does not have a preceding logic operation.

FBD



The OPN box opens the data block via the DB register. It does not have a preceding logic operation.

The OPNI box opens the data block via the DI register. It does not have a preceding logic operation.

STL

```
OPN    "Data block"
```

The OPN operation opens the data block via the DB register.

```
OPNDI  "Data block"
```

The OPNDI operation opens the data block via the DI register.

Fig. 14.11 Opening a data block

14.5.1 Open data block

Opening of a data block is carried out independent of any conditions. It does not influence the result of logic operation or the accumulator contents; the nesting depth of the block calls is not changed.

The opened data block must be present in the work memory. The data block can be addressed absolutely or symbolically.

Fig. 14.11 shows the functions for opening a data block. A block parameter with parameter type BLOCK_DB can also be used instead of *"Data block"*.

14.5.2 Additional data block functions with STL

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
L  10
==I
JC Data10
```

If the data block %DB10 has been opened via the DB register, processing should continue at the jump 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).

14.5.3 Creating, deleting, and testing data blocks

The following system blocks are available for generating, deleting, and testing a data block and are represented graphically in Fig. 14.12.

- CREAT_DB Create data block in work memory (SFC 22)
- CREA_DBL Create data block in load memory (SFC 82)
- DEL_DB Delete data block (SFC 23)
- TEST_DB Test data block (SFC 24)

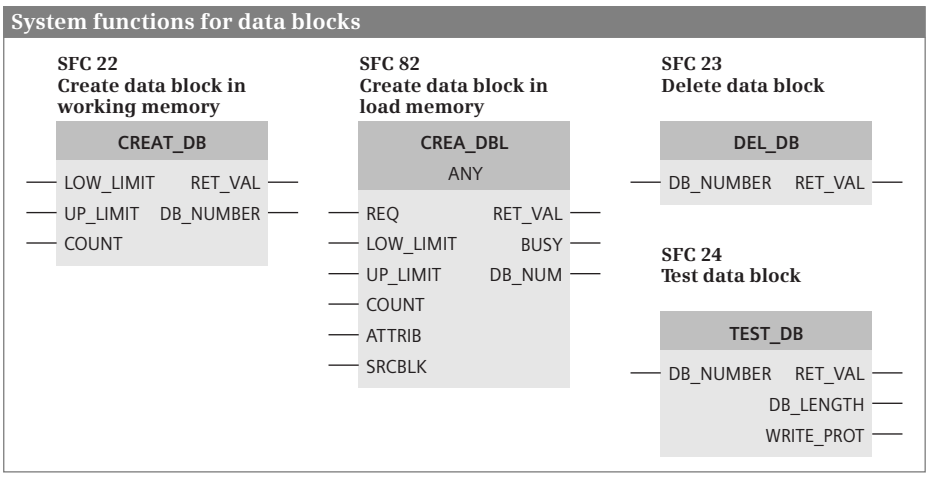


Fig. 14.12 Graphic representation of system functions for data blocks

Data blocks in the user memory

A data block is normally present twice in the user memory of a CPU: once in the load memory and – the part relevant to execution – in the work memory. If the *Only store in load memory* attribute is activated, the data block is only in the load memory (Fig. 14.13).

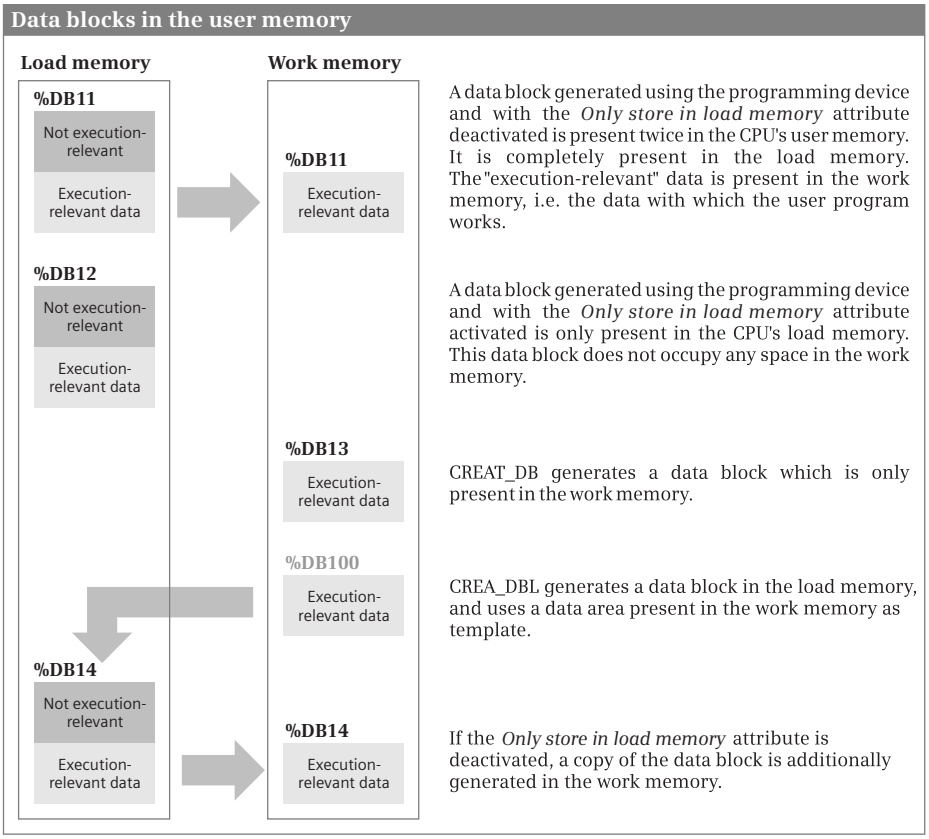


Fig. 14.13 Data blocks in the user memory

CREAT_DB Create data block in work memory

CREAT_DB generates a data block in the work memory. For the number of the data block, the system blocks use the lowest free number in the number range which is specified by the input parameters LOW_LIMIT and UP_LIMIT. The numbers specified 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, not even if the data block is only present in the load memory.

The output parameter DB_NUMBER 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.

The associated data block is not called when it is created. The current data block is still valid. Random data is present in a data block created using CREAT_DB. To use it meaningfully, it is first necessary to write into a data block created in this manner before data is read.

The data blocks created using CREAT_DB are only present in the work memory. If a CPU distinguishes between retentive and non-retentive work memories, CREAT_DB creates a retentive data block. “Retentive” data block means that its contents are retained during a warm restart.

A data block is not created in the event of an error. The DB_NUMBER parameter is then occupied by zero and an error number is output via RET_VAL.

CREA_DBL Create data block in load memory

CREA_DBL creates a data block in the load memory and as appropriate also in the work memory. For the number of the data block, the system function uses the lowest free number in the number range which is specified by the input parameters LOW_LIMIT and UP_LIMIT. The numbers specified 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, not even if the data block is only present in the work memory.

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.

The created data block is preassigned the data area specified at the input parameter SRCBLK. Here you can specify a complete data block, e.g. %DB160 or “Archive 1”, a tag from a data block, or a data area addressed absolute as ANY pointer, e.g. P#DB160.DBX16.0 BYTE 64. The source must be a data area in the work memory.

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 padded with zeros. 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.

You can assign the following properties to the created data block using the input parameter ATTRIB:

- Bit 0 = “1” The data block is only created in the load memory.
  Bit 0 = “0” The data block is created in the load and work memories.
- Bit 1 = “1” The data block is read-only.
  Bit 1 = “0” The data block is not read-only.
- Bit 2 = “1” The data block is not retentive.
  Bit 2 = “0” The data block is retentive.
- Bits 3 to 7 Reserved (assign “0”).

In the case of a data block only created in the load memory, the *Only store in load memory* attribute is activated. Following transfer to offline data management and uploading to the CPU, the data block remains only present in the load memory. Further information on the block properties can be found in Chapter 5.2.4 “Editing block properties” on page 158.

CREA_DBL works asynchronously: It triggers the creation process by signal state “1” at the input parameter REQ. You may only access the read and written data areas again when the output parameter BUSY has signal state “0” again. A maximum of three jobs can be executed simultaneously with CREA_DBL.

The associated data block is not called when it is created. The current data block is still valid.

The data block is not created in the event of an error. The output parameters are then unoccupied and an error message is output via the function value.

DEL_DB Delete data block

DEL_DB deletes the data block in the work and load memories whose number is specified in the input parameter DB_NUMBER. If the data block is currently called or if it was called further “above” in the call hierarchy, a CPU 300 switches to STOP.

Note: If the system function DEL_DB is called in the user program, additional checks are carried out when data blocks are accessed. These tests can increase the command runtime on the DB operand area. If a data block is accessed that was deleted by DEL_DB, OB 121 *Programming error* is called. Data blocks are deleted in the background and the process may take up to the end of the main program. Freeing up memory resources may claim several processing cycles.

A maximum of 21 jobs can be executed simultaneously with the system function DEL_DB. The data block is not deleted in the event of an error and an error number is signaled in the function value.

TEST_DB Test data block

TEST_DB delivers information on a data block whose number you specify in the input parameter DB_NUMBER. The number of existing bytes is present in the output parameter DB_LENGTH and the output parameter WRITE_PROT indicates whether the data block is read-only.

If the tested data block is only in the load memory, this is signaled as an error via RET_VAL; the DB_LENGTH and WRITE_PROT parameters are nevertheless occupied correctly.

If the specified data block is not in the CPU's user memory, RET_VAL = W#16#80B1 is returned.

14.6 Master control relay

14.6.1 Introduction

In the case of conventional controls, a master control relay activates or deactivates part of the controller which may consist of one or more current paths. A deactivated current path switches off all non-retentive contactors and retains the status of retentive contactors. You can only change the status of the contactors again when the master control relay (MCR) is activated.

Note that switching-off with the "software" master control relay does not replace the emergency stop or safety equipment! Consider switching with the master control relay to be the same as switching with a memory function!

The corresponding instructions are available in the programming languages LAD, FBD, and STL for implementation of the master control relay. SCL does not use statements for the MCR function.

Despite the MCR dependency being switched on, you can use the system blocks SET, RESET, SETI, RESETI, SETP, and RESETP to set or reset the bits of an I/O or memory area (see Chapter 13.2 "Transfer functions" on page 476).

14.6.2 MCR dependency

The master control relay (MCR) acts on all operations which write a value back into the memory. Fig. 14.14 lists these MCR-dependent operations.

Some program functions use transfer statements – not visible to you as the user – e.g. to write a value into a memory area. Since the transfer statement writes a value of zero with MCR dependency activated, the corresponding program function is then no longer guaranteed.

You must exclude the following program sections from MCR dependency, otherwise the CPU will switch to STOP or can have an undefined runtime response:

- Block calls with block parameters
- Access to block parameters which are parameter types (e.g. BLOCK_DB)
- Access to block parameters which are components of complex data types or PLC data types

If the MCR dependency is deactivated, the MCR-dependent operations respond "normally" as described in the corresponding chapters.

14.6.3 MCR area and MCR zone

In order to use the properties of the master control relay, define an MCR area using the *Activate MCR area* and *Deactivate MCR area* instructions. The MCR dependency is activated within an MCR area – but not yet switched on.

In order to switch on MCR dependency, define an MCR zone using the *Open MCR zone* and *Close MCR zone* instructions. If *Open MCR zone* is processed with result of

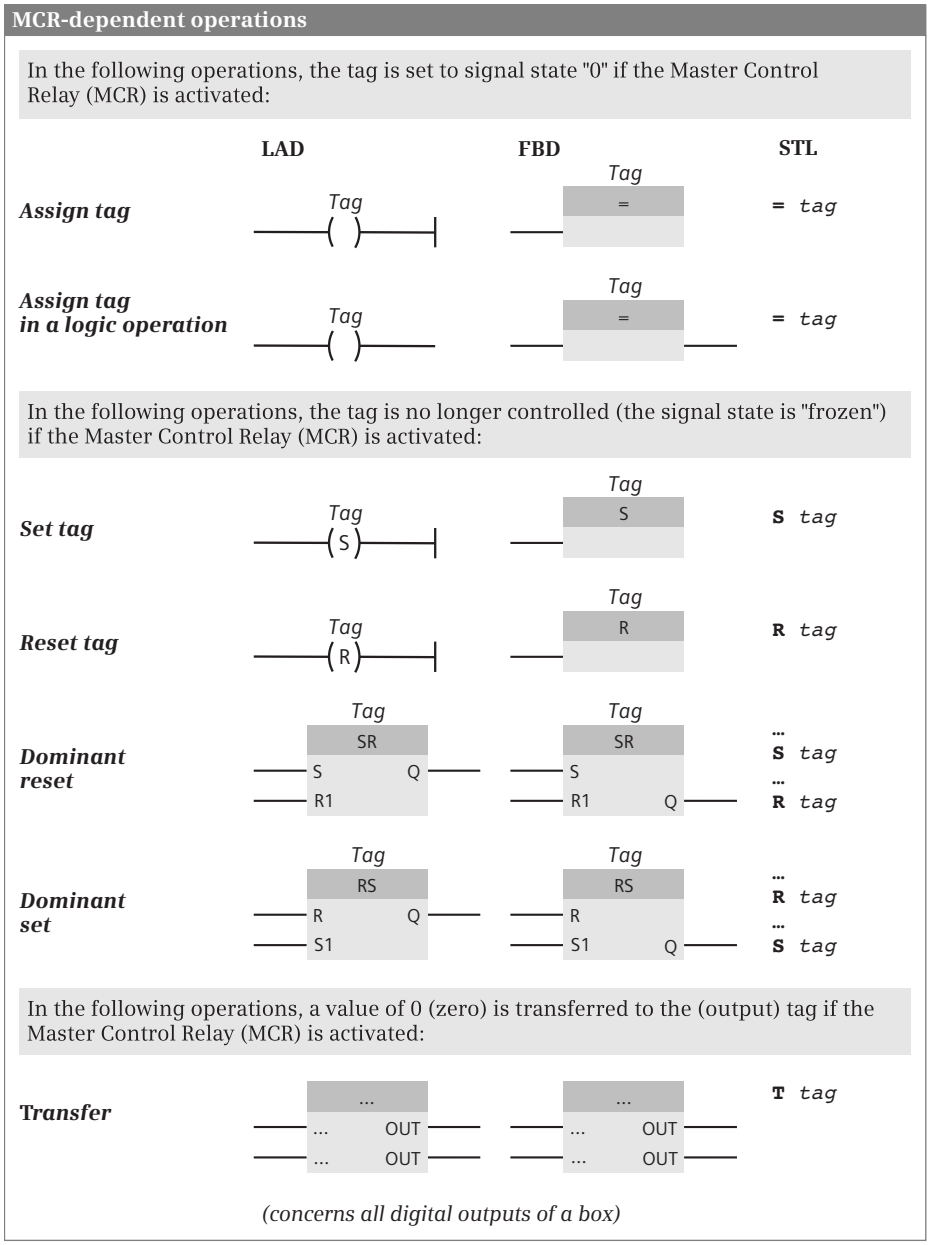


Fig. 14.14 Operations dependent on the MCR

logic operation “0”, the MCR dependency is switched on within the MCR zone (analogous to switching off the master control relay). If *Open MCR zone* is processed with RLO = “1”, MCR dependency is switched off and the operations in the MCR zone respond “normally” again (Fig. 14.15).

Master Control Relay MCR

The Master Control Relay (MCR) controls the write operations in the user memory. You can activate the Master Control Relay within an MCR area and control the MCR dependency within an MCR zone.

Block

Activate MCR area

MCR area

Open MCR zone

MCR zone

Close MCR zone

Deactivate MCR area

End of block

You can use the *Activate MCR area* instruction to open an MCR area. You can define MCR zones within this area.

You can use the *Open MCR zone* instruction to define the start of an MCR zone. Within this zone, you can use RLO = "0" on the *Open MCR zone* instruction to switch on the MCR dependency and RLO = "1" to switch it off again.

You can use the *Close MCR zone instruction* to terminate an MCR zone.

You can use the *Deactivate MCR area* instruction to close an MCR area.

MCR zones can be nested: You can open and close a further MCR zone within an MCR zone. You can nest up to eight MCR zones. The "outer" MCR zone in each case switches on the MCR dependency of all nested MCR zones. If the MCR dependency is switched off in the "outer" MCR zone, you can switch on the MCR dependency in the "inner" MCR zone.

Block

Activate MCR area

MCR area

Open MCR zone

MCR zone 1

Open MCR zone

MCR zone 2

Close MCR zone

Close MCR zone

Open MCR zone

MCR zone

Close MCR zone

Deactivate MCR area

End of block

You can use the *Open MCR zone* instruction of an "outer" MCR zone to switch on the MCR dependency of all nested MCR zones.

You can use the *Open MCR zone* instruction of an "inner" MCR zone to switch on the MCR dependency in this nested MCR zone, even if the MCR dependency in the "outer" MCR zone is switched off.

You can open and close any number of MCR zones in succession. This does not affect the nesting depth.

Fig. 14.15 Nesting of MCR area and MCR zones

MCR zones can be nested: You can open an MCR zone up to eight times before having to close an MCR zone again. The MCR dependency of an "outer" MCR zone switches on the MCR dependency of all nested MCR zones. If the MCR dependency is switched off in an "outer" MCR zone, you can switch on the MCR dependency in the "inner" MCR zone (and in all further nested MCR zones).

14.6.4 MCR area and MCR zone with a block change

If you call a block within an MCR area, MCR dependency is deactivated in the called block. An MCR area only begins again with the *Activate MCR area* instruction. When leaving a block, MCR dependency is set as it was prior to the block call, independent of the MCR dependency with which the called block was left.

A block call within an MCR zone does not change the nesting depth of an MCR zone. The program in the called block is still present in the MCR zone which was open during the block call and is controlled by this. However, you must activate the MCR dependency again in a called block by opening the MCR area.

14.6.5 Instructions for the master control relay

The instructions shown in Fig. 14.16 are available in the programming languages LAD, FBD, and STL for implementation of the master control relay (MCR).

Descriptions of how the master control relay is programmed in the various languages can be found for LAD in Chapter 7.6.5 "Master Control Relay (MCR)" on page 280, for FBD in Chapter 8.6.5 "Master Control Relay (MCR)" on page 313, and for STL in Chapter 9.6.7 "Master Control Relay (MCR)" on page 356.

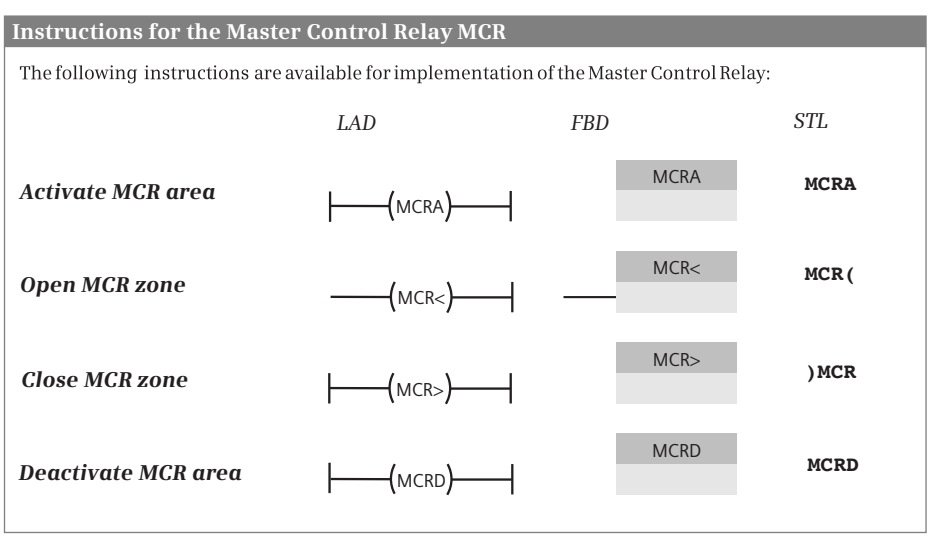


Fig. 14.16 Master control relay, instructions in LAD, FBD, and STL

15 Online operation 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 mechanical connection – the cabling – depends on the configuration of the programming device and CPU. Every CPU has an MPI (multi-point interface). CPUs with "DP" in the short designation have at least one PROFIBUS interface, where an MPI and a PROFIBUS interface can also be combined in one connection. CPUs with "PN" in the short designation have at least one PROFINET interface. A programming device can be connected via each of these interfaces.

The project tree shows under *Online access* which interfaces (interface modules) are available in the programming device. The mechanical connection (the connection to a subnet) and the logical connection (the definition of the transmission protocols) are not configured. Only the bus and network addresses of the two devices must be matched to each other.

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 diagnostic buffer. The online and diagnostics tools support you in troubleshooting during commissioning.

In online mode you transfer the program created offline to the CPU and test it. 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). In addition, the offline and online versions of a block can be compared.

You can also test a user program in the programming device without a connected CPU. The S7-PLCSIM simulation program is described in Chapter 18.3 "Simulation with the TIA Portal" on page 700.

15.1 Connection of a programming device to the PLC station

The connection between a programming device and a PLC station is not configured with the hardware configuration. If you connect the programming device directly to the MPI of the CPU, the default properties of the interface are sufficient for online mode. When connecting via the PN interface, the programming device must be addressed in the same subnet. Adaptation to the network configuration is carried out by STEP 7 by assigning a temporary IP address.

You parameterize the interfaces of the CPU using the hardware configuration (Chapter 3.3.1 “Parameterization of CPU properties” on page 70).

15.1.1 Settings on the programming device

Setting network addresses for Industrial Ethernet in the operating system

If your programming device is already working in a network, it has an IP address and a subnet mask which are matched to operation in the network. In most cases, integration into the network is made automatically through assignment of the network configuration by a DHCP server.

You can check the settings in the properties of the LAN connection, for example with *Start > Control Panel* and double-click on *Network and Internet connections*. Click on the used connection and click in the dialog window on the *Properties* button. Select *Internet Protocol Version 4 (TCP/IPv4)* in the selection box in the properties and click on the *Properties* button. The dialog window offers the options *Obtain an IP address automatically* (via a DHCP server) and *Use the following IP address* (manual settings). The *Advanced...* button leads to the advanced IP settings in which you can, for example, set additional temporary IP addresses and subnets masks.

Set access point

When installing STEP 7, the *Set PG/PC interface* tool is created in the Windows Control Panel. This can be used to check and set the interface to the network.

Open the *Set PG/PC interface* tool, for example from the Windows desktop using *Start > Control Panel*. The *Access Path* tab should show *S7ONLINE (STEP 7)* in the *Access Point of the Application* box. In order to change the interface module, select the LAN interface module used under *Interface Parameter Assignment Used* click on *OK*.

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 subnet with which the programming device is to be connected under *Assignment*.

15.1.2 Connecting the programming device to the PLC station

Connect the programming device to the CPU using a cable appropriate to the interface.

Switch on CPU

You require a Micro Memory Card (MMC) for operation of a CPU 300. It is inserted into the CPU when deenergized. Set the mode switch on the front panel of the module to STOP and switch on the power supply to the CPU. All LEDs light up when the CPU operating system ramps up. If the CPU does not detect any errors, the DC5V and STOP LEDs light up following ramping up. The red SF LED lights up in the event of an error and all LEDs flash if the CPU is faulty.

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 if the MMC is empty.

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 address. At the same time, the graphic is provided with an orange background (Fig. 15.1).

Select the line with the station. You can then use the *Flash LED* button in order to flash the FRCE LED on the front panel of the CPU. To process the selected station further in the project view, click on the *Show* button.

Temporary IP address on the programming device

An online connection over Ethernet can only be established if both devices are in the same subnet. 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.

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 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*.

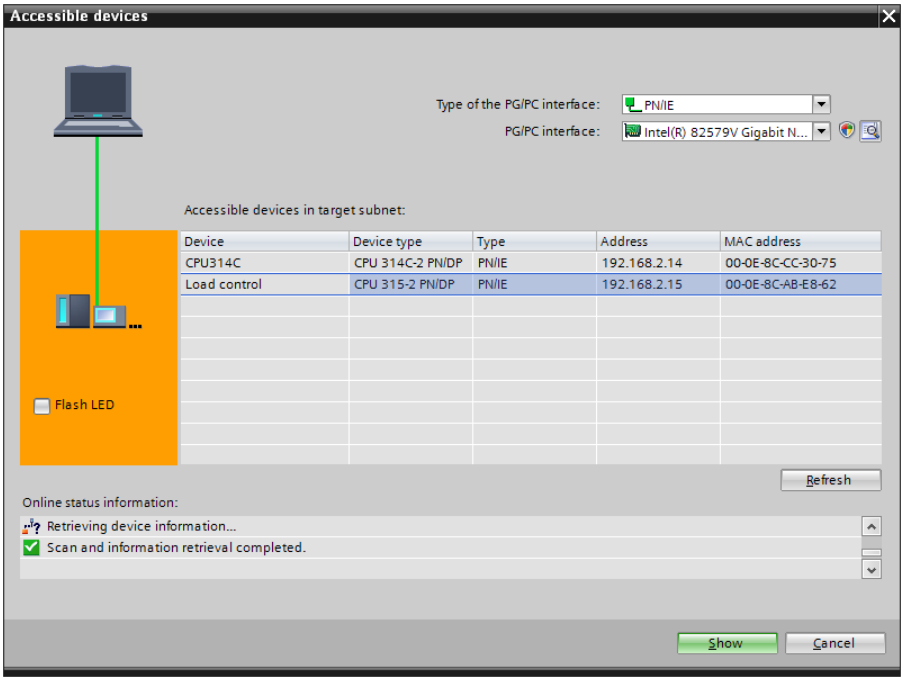


Fig. 15.1 Dialog window *Accessible devices*

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 583.

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 583 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 588 describes how you can test a user program.
- Chapter 15.2.4 “Editing of online project without offline project” on page 572 describes how you can access the online project data of the CPU without the user program.

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 Micro Memory Card as data medium.

If you transfer the user program to the CPU via an online connection, it is written into the CPU's load memory. In the case of a CPU 300, the load memory is on the Micro Memory Card.

You can also write to a Micro Memory Card in the programming device and use it as data medium. Transfer the project data from the offline data management to the Micro Memory Card inserted in the card reader and then insert the Micro Memory Card into the CPU in the deenergized state. Following a memory reset when switching on, the data relevant to execution is transferred from the Micro Memory Card to the CPU's work memory.

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 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 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 window. 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. The compilation process can be observed in the dialog window *Load preview*.

Following compilation without errors, set the further process in the dialog window *Load preview* for stopping the CPU, loading the device configuration, and loading the software. In the *Action* column you can use the drop-down list to change or deselect the suggested action. You can continue with loading by clicking on the *Load* button (Fig. 15.2).

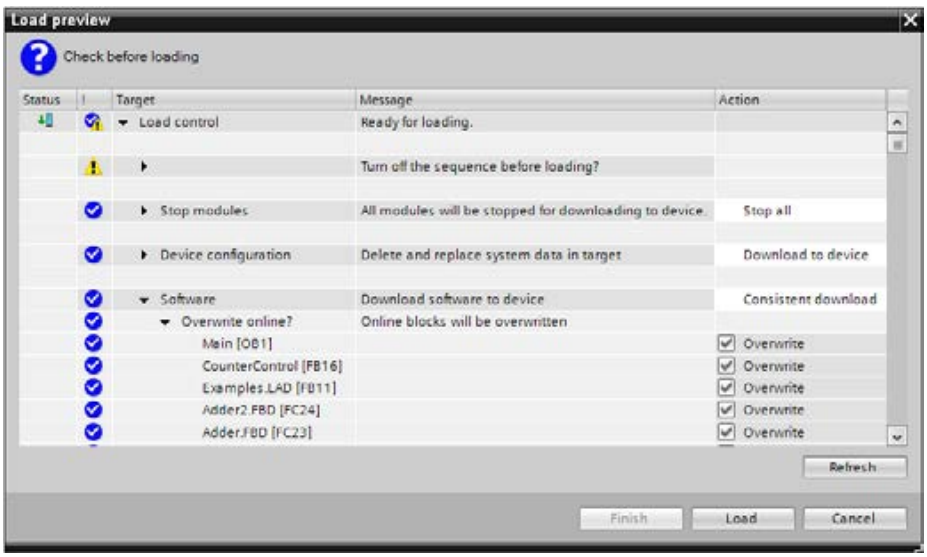


Fig. 15.2 Dialog window *Load preview*

The device configuration can only be loaded when the CPU is at STOP. You can select the following for loading the software: *Consistent download* (all previously selected blocks and the blocks with their call are loaded) and *Download selection* (only the selected blocks are loaded). If you select the *Program blocks* folder and load all blocks, blocks that are not available offline are deleted online.

Start CPU following loading

The results of loading are displayed in the dialog window *Load results*. Following loading without errors, you can start the CPU with the new user program.

Caution: Make sure when starting the CPU – 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!

If the CPU was in RUN mode prior to loading, the *Start all* checkbox is activated in the *Action* column. If it was at STOP, activate the checkbox in order to start the CPU. Click the *Finish* button.

With the *Start all* checkbox activated, the CPU is started following completion of loading. If no errors occur, the CPU is then in RUN mode. The green RUN LED lights up.

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
  (see Chapter 15.4.2 “Diagnostic information” on page 584 for information on online access and module status, reading the diagnostic buffer, updating the firmware, etc.)
- Use the online tools
  (see Chapter 15.4.5 “Online tools” on page 586 for control of CPU, display of current cycle time and memory configuration)
- Edit and compare blocks online and offline
  (see Chapter 15.3 “Working with blocks in online mode” on page 574)
- Test the user program
  (see Chapter 15.5 “Testing the user program” on page 588)

You can use the *Go offline connection* icon to switch online mode off again.

15.2.2 Reloading the project data

When reloading project data, only the changes compared to the online project data are loaded.

When using the command for loading, you define which project data is to be 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 574.

Changing configuration data of CPU

Changing configuration data is only possible in offline mode. Switch to offline mode, modify the configuration data offline, and transfer it to the CPU. If you only wish to transfer the configuration data, select the PLC station in the project tree and then the *Download to device* > *Hardware configuration* command from the shortcut menu.

The CPU is switched to STOP during loading.

Starting the CPU

In the *Load results* dialog window 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.

Loading an incorrectly compiled, inconsistent program

An error which occurs when compiling prior to loading is indicated in the dialog window *Load preview*. The *Target* column indicates under *Software* (click triangle on the left) the component for which the error has occurred. Continuation of loading is only possible when the error has been eliminated.

Error message following loading

If the CPU does not start following loading – the yellow STOP LED lights up – or if the red SF LED flashes, the diagnostic 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 “Diagnostic buffer” on page 585 describes how the diagnostic buffer supports you during troubleshooting.

15.2.3 Protection of the user program

With a CPU 300, access to the user program can be protected by a password. Anyone with knowledge of the password has unlimited access to the user program. You can define three protection levels for all those who do not know the password. You set the protection levels in the *Protection* tab with the hardware configuration when parameterizing the CPU.

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 following protection levels are possible:

- Protection level 1 (no protection) is the default setting. It means that there are no limitations to access to the user program. With the system function PROTECT, the write protection (protection level 2) can be switched on and off via the program in protection level 1 (see Chapter 5.5.4 “Hold, stop, and protect program” on page 181).
- In protection level 2, the user program can only be read.
- In protection level 3, it is neither possible to read nor write the user program. Exception: Reading the diagnostic buffer and monitoring tags using watch tables are possible in every protection level.

If you select either protection level 2 or 3, you will be requested to define a password. The password has a maximum length of 8 characters.

The protection is effective once the settings have been loaded to the CPU. If you access a CPU which is protected by a password, you will be requested to enter the password. Anyone in possession of the password has unlimited access to the CPU, independent of the protection level set.

Know-how protection with source files

In the case of source files for STL and SCL blocks it is possible to protect a block against undesired access by using the keyword KNOW_HOW_PROTECT. You can no longer cancel this protection, in contrast to block protection with password in the TIA Portal (see Chapter 18.1 “Working with source files” on page 687).

15.2.4 Editing of online project without offline project

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 desired PLC station in the *Accessible devices in target subnet* list 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 diagnostic 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. Only blocks which are present offline can be newly created, modified, deleted, or tested.

Uploading the 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 or the *Program blocks* folder 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, set any required actions and click the *Upload from device* button. The available online blocks are transferred to the offline project.

Note the following when uploading into an empty offline project: PLC tag tables and PLC data types are not available. The local data of the uploaded blocks have no symbolic addresses. Substitute identifiers are used instead, e.g. *Input_n* for an input block parameter or *Static_n* for static local data. Blocks programmed with SCL or GRAPH and their instance data blocks cannot be edited following the upload. Further details on uploading blocks are described in Chapter 15.3.5 "Uploading blocks from the CPU" on page 577.

15.2.5 Working with the Micro Memory Card

A SIMATIC Micro Memory Card (MMC) for a CPU 300 is an SD memory card (secure digital memory card) preformatted by Siemens. If you wish to delete the contents of the Micro Memory Card, you must only delete files or folders. Formatting the Micro Memory Card makes it unusable in a CPU 300.

The Micro Memory Card is essential for operation of a CPU 300. It is the load memory of the CPU. The Micro Memory Card can also be used as a storage medium for updating or saving the CPU operating system.

Writing project data onto the MMC

To transfer the user program, insert the MMC into the programming device's card reader. Copy the project data of the PLC station onto the MMC, for example with the PLC station selected using the *Copy* command and with the subsequently selected SD card using the *Paste* command from the shortcut menu, or by dragging the PLC station from the project tree onto the MMC with the mouse button pressed.

The project is compiled if necessary. Following error-free compilation, the *Load preview* window is displayed. Check the proposed actions and change if necessary, and then click on the *Load* button. Clicking on the *Finish* button following loading terminates the procedure.

Reading project data from the MMC

You can also transfer the user program or individual blocks of it from the MMC back to the offline project. The configuration data cannot be “transferred back”.

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. You can delete the online and/or offline version of a block. 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 Editing the online version of a block

Modifying the online version of a block

A block can only be modified 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 online version of the block, for example by adding a new scan of logic operation during program testing, the program editor automatically switches to the offline version. Following the modification, you transfer the modified offline version to the CPU, for example by right-clicking on a free space in the working window and the *Download to device* command from the shortcut menu.

Adding the online version of a block online

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 a block

You delete a block if you select it in the project tree and then the *Delete* command from the shortcut menu. If online mode is switched on and you wish to delete a block which is present both online and offline, you can choose the following in the dialog window that appears:

- *Delete blocks from device* deletes the selected blocks in the CPU.
- *Delete from project* deletes the offline version of the selected blocks.

The blocks are permanently 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.

15.3.3 Downloading a block to the CPU

To download into the user memory of the CPU, select one or more blocks in the project tree in the *Program blocks* folder and then the command *Download to 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.

Please note that a block with know-how protection is downloaded to the CPU without recovery information. When uploading the block, it cannot be opened anymore.

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, 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 block interface of a data block was changed, 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, the actual values of data tags are not changed in the work memory. If the block interface of a data block was changed, 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 15.3.6 “Working with setpoints” on page 579.

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.

15.3.4 Packing the work memory

If you wish to load a new or modified block into the CPU, the latter stores the block in the load memory and transfers the parts relevant to execution to the work memory. If a block with the same number already exists, this “old” block is declared as invalid in the memory management and the new block is added to the work memory “from behind”. A deleted block is also “only” declared as invalid and is not actually removed from the memory.

Gaps result in the work memory in this manner which reduce the memory space available for assignment more and more. The gaps can only be filled using the *Compress* function. If you carry out compression in RUN mode, the blocks currently being processed are not shifted; compression without gaps is only achieved when in the STOP state.

Online mode must be switched on to allow compression. Double-click in the project tree under the PLC station on the *Online & diagnostics* tool. You can then find the *Compress* button

- in the working window under *Diagnostics* > *Memory* and
- in the task window with the online tools on the *Memory* pallet.

The *Compress* button reorganizes the work memory in order to increase the largest continuous free memory area. No other online jobs such as the program status must be active.

15.3.5 Uploading blocks 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 for uploading, 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 individual blocks, select the blocks 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 *Preview for loading from device* dialog window, messages are displayed, which you can answer with actions, for example by inserting the block with a different name (but with the same number). As soon as uploading is possible, the *Load* button is activated. Clicking on the *Upload from device* button starts the upload.

Please note that the uploaded blocks with SCL or GRAPH program and their instance data blocks cannot be edited anymore. If the offline version of a data block differs from the online version, the data block can be uploaded into the offline project. The actual values of the online version are then used as start values in the offline version. The uploaded code and data blocks have no symbolic identifiers of the local data. Substitute identifiers are used instead, for example *Input_n* for an input parameter or *Static_n* for static local data.

Please note that a block with know-how protection is downloaded to the CPU without recovery information. When uploading the block, it cannot be opened anymore.

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 setpoint values, or only the retain 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”* icon. The actual values marked as setpoints from the *Snapshot* column are imported as start values (see Chapter 15.3.6 “Working with setpoints” on page 579).

- To import the actual values of retain 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* or *Apply snapshot values as start values > Only retain values* in the shortcut menu.

Saving actual values per program

You can write the actual values of a data block into the load memory per program (see Chapter 13.2.6 “Transfer data area from and to load memory” on page 483). The actual values then become start values.

You can then import this data block into the offline data management by copying directly from the Micro Memory Card and the result is a data block with the former actual values as start values.

15.3.6 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 setpoints* 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. An example is shown in Fig. 15.3. Note that the monitor values can come from different program cycles.

...10 ▸ Load control [CPU 315-2 PN/DP] ▸ Program blocks ▸ ConveyorBelt ▸ Belt_data [DB100]

Belt_data

	Name	Data type	Offset	Start value	Snapshot	Monitor value	Setpoint
1	Static						
2	Quantity	Array [1 .. 4] of Int	0.0				
3	Power	Array [1 .. 4] of Real	8.0				
4	Monitoring	Array [1..4] of Time	24.0				
5	Monitoring[1]	Time	0.0	T#5s	T#25S_340MS	T#25S_340MS	
6	Monitoring[2]	Time	4.0	T#5s	T#15_100MS	T#15_100MS	
7	Monitoring[3]	Time	8.0	T#2.8s	T#4S	T#4S	
8	Monitoring[4]	Time	12.0	T#3.3s	T#2S_500MS	T#2S_500MS	
9	Start signal	Bool	40.0	false	false	FALSE	
10	/Stop signal	Bool	40.1	false	false	FALSE	
11	Initial state	Bool	40.2	false	false	FALSE	
12	Duration	Time	42.0	T#10.5s	T#12S_770MS	T#12S_770MS	

Fig. 15.3 Monitor values and snapshots

To import the setpoints, click on the *Copy all setpoints from "Snapshot" column to the "Start value" column* icon. The actual values marked as setpoints 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*.

15.3.7 Comparing blocks

With the compare editor you can carry out an offline/online comparison and an offline/offline comparison. For an offline/online comparison, the offline blocks of a PLC station are compared with the blocks in the CPU. The offline/offline comparison compares the blocks, PLC tags, and PLC data types of two PLC stations.

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.

Comparing offline/online 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, objects which have different offline and online versions are displayed (Fig. 15.4). 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.

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.

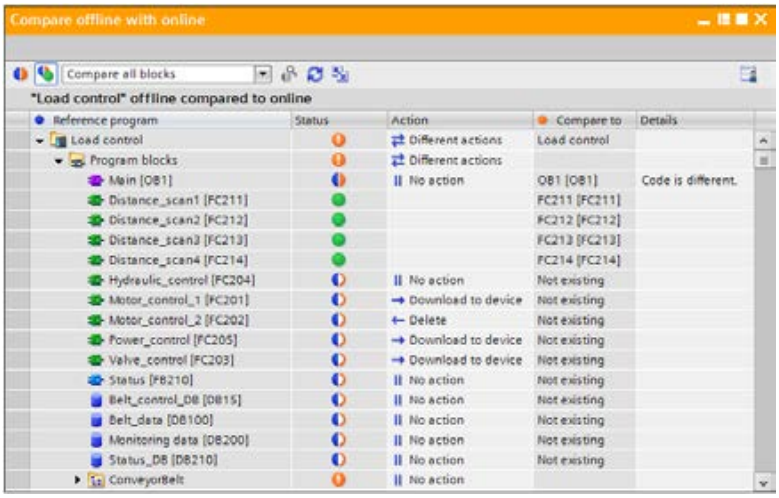


Fig. 15.4 Example of comparison of blocks

Offline/offline comparison of blocks

With 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 44). 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 “Move comparison object here”). 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 *A corresponding network was not 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 the diagnostics enable.

This chapter describes the diagnostics options offered by the programming device in online mode. Chapter 5.8 “Diagnostics” on page 211 describes how you can react to a diagnosis event in the program.

When a diagnosis event occurs:

- The SF LED (group error) 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 diagnostic buffer in the sequence of their occurrence with date and time. In addition to the diagnostic 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 a malfunction and can help to localize the fault. The CPU's operating system signals a malfunction in the following manner:

- The SF LED lights up permanently.
  A hardware fault or software error is present.

- The SF LED flashes.
  If none of the other LEDs lights up, the CPU does not have valid firmware. If the other LEDs (except MAINT) also flash, an internal system fault is present; the CPU is defective.
- The STOP LED lights up permanently.
  The CPU does not enter RUN mode when switched on or goes to the STOP state during RUN mode. Possible causes: Manual change in mode through the programming device, set startup type ("Startup – STOP"), STP function in user program, system response to an execution error in the program. The events triggering the STOP state are entered into the diagnostic buffer.
- The STOP LED flashes.
  The CPU requests a memory reset (flashing at 0.5 Hz) or carries out a memory reset (flashing at 2 Hz).

Every input/output channel of a digital module has a green status LED 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.

Digital and analog modules with diagnostic capability have an SF LED (group error), which lights up when a diagnosis event occurs. The SF LED goes out when all errors has been resolved.

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 manufacturer information.
- Diagnostics status: Status information of selected module, e.g. *Module exists*, differences between configured and existing modules.
- Diagnostic buffer: Display of diagnostic buffer content.
- Cycle time: Display of preset or configured cycle (monitoring) time and – in RUN mode – the cycle time diagram and the shortest, current, and longest cycle (processing) times.
- Memory: Display of memory utilization for the load, work and retentive memories.
- Communication: Display of connection resources and cycle load resulting from communication.
- MPI, DP and PN interfaces: Properties of the corresponding interface.
- Operating hours counter: Display of runtime meter used and count value.
- Performance data: Display of CPU performance data (quantity frameworks for operand areas, organization blocks, system blocks).

Cycle times and resources are displayed in parallel in the online tools.

15.4.3 Diagnostic buffer

The diagnostic 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 diagnostic 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. 15.5).

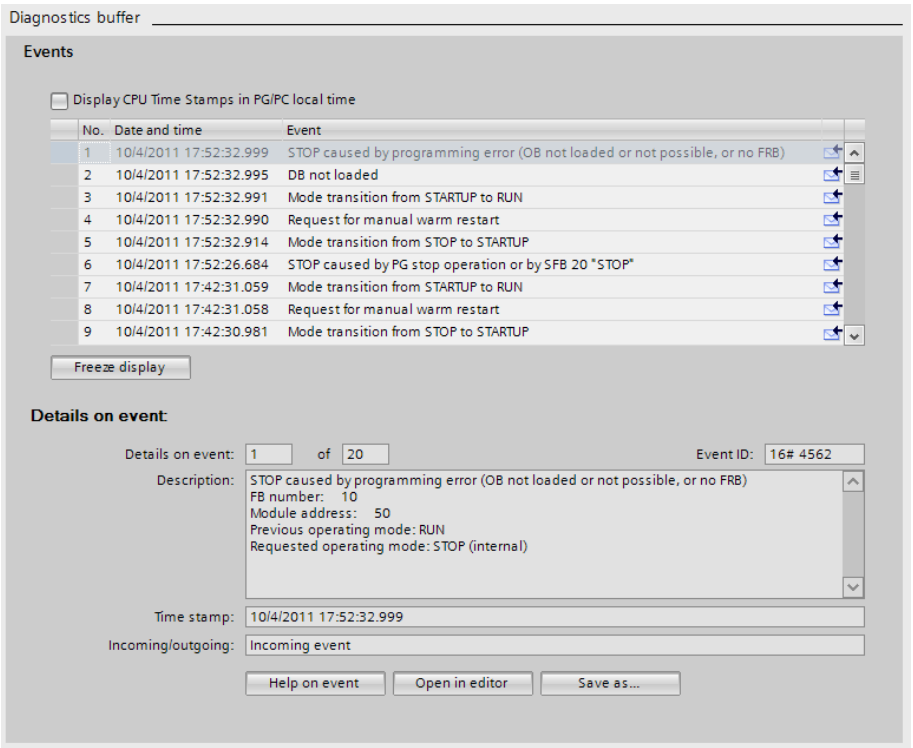


Fig. 15.5 Example of display of diagnostic buffer

The most recent event is present in the first line in the diagnostic buffer. A diagnostic 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.

Using the *Help on event* button, you can obtain additional information on the selected event. If the entry refers to a block, e.g. with an access error to the I/O, it is possible to switch to the position of the fault in the user program by using the *Open in editor* button.

The *Freeze display* button stops 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 diagnostic buffer as a text file.

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 diagnostic 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:

- Assign IP address: Setting of the IP address, subnet mask, router address.
- Set time: Display of programming device and module time, setting of real-time clock on CPU, time synchronization.
- Firmware update: Display of current firmware and preparations for updating the firmware.
- Assign name: Enter or change the PROFINET device name.
- Reset of PROFINET interface parameters

15.4.5 Online tools

You can use the *Online & diagnostics* command from the project tree 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 panel 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, retentive 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 diagnostic 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 retentive memories as bars. The *Compress* button can be used to minimize the utilization of the work memory.

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: Everything OK
- Blue/orange semicircles: The online version and offline versions of the object are different
- Blue/orange semicircles, right half (orange) filled: Only the online object is present
- Blue/orange semicircles, 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. 15.6). Via the link in the *Details* column you can access the *Go online* dialog or the online and diagnostics view of the faulty device.

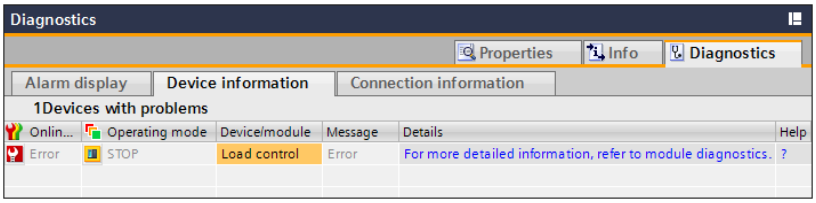


Fig. 15.6 *Diagnostics* tab in the inspector window

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 diagnostic 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
- Test in single step mode
  Step-by-step execution of an STL or SCL program with monitoring of the tag values
- Monitor PLC tags
  Monitor the values in a PLC tag table
- Monitor data tags
  Monitor the tag values in a data block
- Test with watch tables
  Monitor and control the tag values in watch tables
- “Enable peripheral outputs” and “Modify now”
  Control peripheral outputs with CPU at STOP
- Test with force table
  Monitor the tags in the force table and set to a fixed value (force)

A general prerequisite for testing the user program is an existing online connection. When testing with program status and in single step mode, the offline and online versions of the block must be identical.

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 18.3 “Simulation with the TIA Portal” on page 700).

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. In single step mode, you set a breakpoint at the position to be tested in the block program.

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 set condition applies, the program status is recorded or a jump is made to the breakpoint located in the specified call of the block. You can make the following settings when you click the *Change* button:

- No trigger applied
  If the call environment is not defined, the program status of an optional call is recorded in the case of a repeated call or a jump is made to the next breakpoint.
- Instance data block
  The program status of the function block is only recorded or program execution is only interrupted at a breakpoint in the function block if the function block is called with the specified instance data block.
- Call environment
  The program status of the code block is only recorded or program execution is only interrupted at a breakpoint in the code block if the code block is called from the specified block or from a specific path.

The setting *No condition defined* is the default setting for the call environment.

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.

In the case of a CPU 300 with firmware release ≥ V2.8, two blocks can be monitored simultaneously with the program status.

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. ... > *Decimal*, ... > *Hexadecimal*, and ... > *Floating-point* are possible.

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 *Modify* > *Modify to 0* command from the shortcut menu if the binary tag is to be set to signal state “0” or *Modify* > *Modify to 1* if the binary tag is to be set to signal state “1”. In the case of digital tags, select the *Modify* > *Modify operand*... command from the shortcut menu and specify the desired value.

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. 15.7).

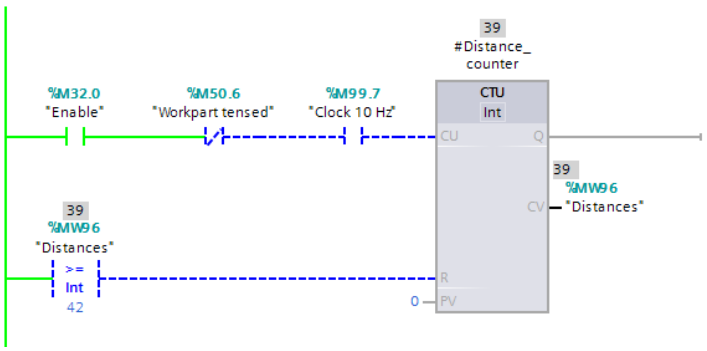


Fig. 15.7 Program status in LAD representation

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.

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. 15.8). 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

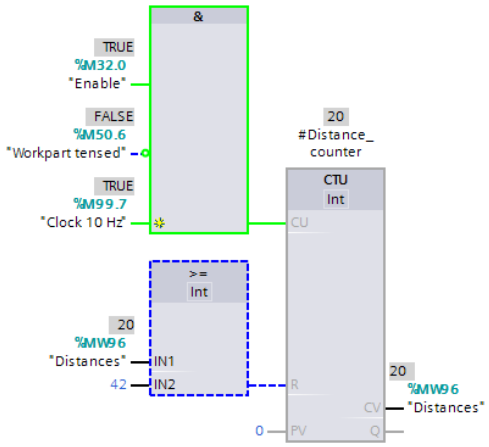


Fig. 15.8 Program status in FBD representation

from the shortcut menu. The *Modify* > *Monitor selection* command from the shortcut menu means that only the selected program element is monitored.

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. 15.9).

You can use the *Absolute/symbolic operands* icon to select the displayed type of addressing.

			Address	RLO	Value	Extra
1	A	"Enable"	%M32.0	1	1	
2	AN	"Workpart tensed"	%M50.6	1	0	
3	A	"Clock 10 Hz"	%M99.7	1	1	
4	=	#temp_bool_1		1	1	
5	L	"Distances"	%MW96	1	8	
6	L	42	42	1	42	
7	>=I			0		
8	=	#temp_bool_2		0	0	
9	CALL	#Distance_counter		In	Out	
10		Int				
11		CU :=#temp_bool_1		TRUE		
12		R :=#temp_bool_2		FALSE		
13		PV :=0	0	16#0		
14		Q :=				
15		CV :="Distances"	%MW96		16#8	

Fig. 15.9 Program status in STL 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 extended status information* attribute in the block properties and load the block again into the CPU (Fig. 15.10).

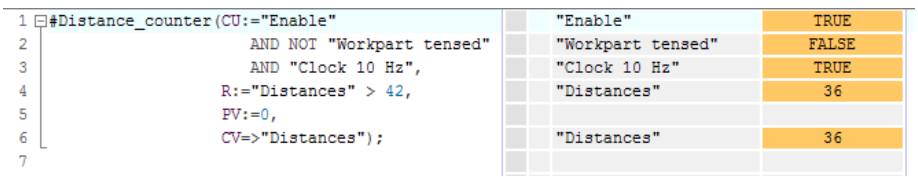


Fig. 15.10 Program status in SCL representation

15.5.3 Testing in single step mode

In the programming languages STL and SCL, you can test the program statement by statement in single step mode. The CPU is in the HOLD state in this case; the peripheral outputs should be switched off as a precaution. You can use breakpoints to stop the program at any desired position and test step-by-step.

Set breakpoints

You set the breakpoints for testing in single step mode in the program of the offline block. Click with the right mouse button at the beginning of the line in the gray area in front of the line number in which you wish to set a breakpoint and then select the *Set breakpoint* (STL) or *Breakpoints* > *Set* (SCL) command from the shortcut menu. Alternatively, double-click on the gray area. The breakpoint is then set and displayed on the *Testing* task card in the *Breakpoints* pallet.

You can set any number of breakpoints, but only activate one of the quantity which depends on the CPU for testing. Breakpoints are not saved. If the project is closed, the breakpoints are lost.

Moving to a breakpoint

In order to move to a breakpoint, double-click in the *Breakpoints* pallet on the breakpoint you wish to move to, or click on it with the right mouse button and select the *Go to breakpoint* command from the shortcut menu.

In order to move from one breakpoint to another, select (with the block open) the *Online* > *Breakpoints* > *Next* command in the main menu if the next breakpoint is to be jumped to, or *Online* > *Breakpoints* > *Previous* if the previous breakpoint is to be jumped to.

Activate breakpoints

The breakpoints must be activated to enable testing. In the case of a CPU 300 with firmware release ≥ V2.8, a maximum of 4 breakpoints can be active. Note that the single step functions “Step over”, “Step into” and “Run to cursor” also count as breakpoints. These functions can no longer be executed if the maximum number of breakpoints is activated.

When activated, the breakpoint is transferred to the CPU. Program execution is carried out in RUN mode up to the first active breakpoint and then stops in HOLD mode. The STOP LED lights up permanently and the RUN LED flashes.

Caution! When testing in single step mode, the CPU if switched to HOLD mode during ongoing operation on the process. Make sure that no damage to property or injury to persons can occur!

You activate a specific breakpoint by selecting it in the program code or in the *Breakpoints* pallet and then the *Enable breakpoint* command from the shortcut menu. Alternatively, you can activate all breakpoints together in the *Breakpoints* pallet with the drop-down list under the *Actions for all breakpoints* icon.

Once you have made sure that the activation of breakpoints does not have any negative impacts, acknowledge the displayed warning message.

Status display

In the task window on the *Testing* task card in the *PLC register* pallet, you can monitor the values of the status bits and the CPU registers for the statement line at which program execution has stopped. All status bits, the accumulators, the address registers, and the data block registers are displayed.

Testing in single step mode

You can select the functions for testing in single step mode in the shortcut menu which you call up in the gray area in front of the line number or by clicking the corresponding icons in the *Breakpoints* pallet (Fig. 15.11).

You can continue program testing in the following ways:

- Run
  Continue program execution at normal processing rate up to next active breakpoint.
- Run to cursor
  Continue program execution at normal processing rate up to the statement selected by the mouse pointer.
- Step over
  The currently selected statement is executed, a switch is made to the next statement and then stopped.
- Step into
  Call of a subordinate block and continuation of program execution in the called block.
- Step out
  Continuation of program execution in the called block if the program execution in the called block was interrupted at a breakpoint.

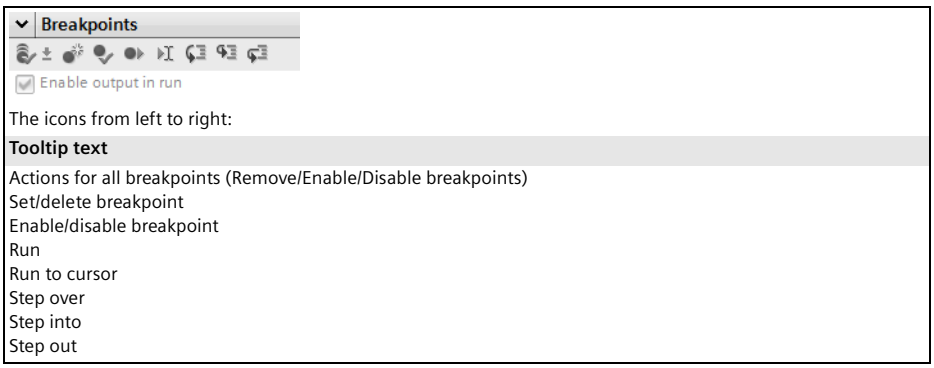


Fig. 15.11 Icons in the *Breakpoints* pallet

Upon each stop, the *PLC register* pallet shows the current assignment of the status contents and the CPU registers.

If the online connection is canceled, all breakpoints are deactivated. You will be informed in this case that the CPU then returns to RUN mode.

Exit single step mode

To exit single step mode, you disable and remove the breakpoints – for example in the *Breakpoints* pallet using the *Actions for all breakpoints* icon. The *Enable output in RUN* checkbox must be activated.

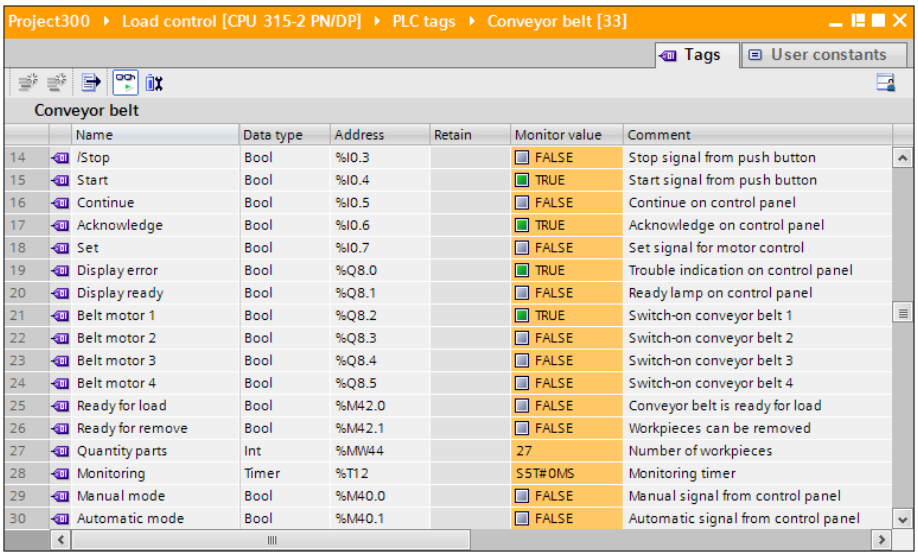
Subsequently click on the *Run* icon in the *Breakpoints* palette or right-click in the gray column in front of the line number and select the *Run* command from the shortcut menu. The CPU then goes to RUN mode.

15.5.4 Monitoring of PLC tags

To monitor using the tag table, double-click on the PLC tag table in the project tree. Click the *Monitor all* icon in the toolbar. The PLC tag table changes to online mode and the *Monitor value* column is displayed. You can now monitor the tag values.

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. 15.12 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.



	Name	Data type	Address	Retain	Monitor value	Comment
14	/Stop	Bool	%I0.3		FALSE	Stop signal from push button
15	Start	Bool	%I0.4		TRUE	Start signal from push button
16	Continue	Bool	%I0.5		FALSE	Continue on control panel
17	Acknowledge	Bool	%I0.6		TRUE	Acknowledge on control panel
18	Set	Bool	%I0.7		FALSE	Set signal for motor control
19	Display error	Bool	%Q8.0		TRUE	Trouble indication on control panel
20	Display ready	Bool	%Q8.1		FALSE	Ready lamp on control panel
21	Belt motor 1	Bool	%Q8.2		TRUE	Switch-on conveyor belt 1
22	Belt motor 2	Bool	%Q8.3		FALSE	Switch-on conveyor belt 2
23	Belt motor 3	Bool	%Q8.4		FALSE	Switch-on conveyor belt 3
24	Belt motor 4	Bool	%Q8.5		FALSE	Switch-on conveyor belt 4
25	Ready for load	Bool	%M42.0		FALSE	Conveyor belt is ready for load
26	Ready for remove	Bool	%M42.1		FALSE	Workpieces can be removed
27	Quantity parts	Int	%MW44		27	Number of workpieces
28	Monitoring	Timer	%T12		S5T#0MS	Monitoring timer
29	Manual mode	Bool	%M40.0		FALSE	Manual signal from control panel
30	Automatic mode	Bool	%M40.1		FALSE	Automatic signal from control panel

Fig. 15.12 Monitoring with the PLC tag table

15.5.5 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.

Fig. 15.13 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.

...00 ▸ Load control [CPU 315-2 PN/DP] ▸ Program blocks ▸ ConveyorBelt ▸ Belt_data [DB100]

Belt_data

	Name	Data type	Offset	Start value	Monitor value	Setpoint	Comment
1	Static						
2	Quantity	Array [1 .. 4] of Int	0.0				
3	Quantity[1]	Int	0.0	0	54		
4	Quantity[2]	Int	2.0	0	15		
5	Quantity[3]	Int	4.0	0	0		
6	Quantity[4]	Int	6.0	0	123		
7	Power	Array [1 .. 4] of Real	8.0				
8	Power[1]	Real	0.0	1.2	1.13		
9	Power[2]	Real	4.0	1.2	1.09		
10	Power[3]	Real	8.0	3.4	0.0		
11	Power[4]	Real	12.0	3.2	3.22		
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	TRUE		

Fig. 15.13 Example of monitoring of data tags

Please note that tag values displayed in monitoring mode can originate from different program cycles.

15.5.6 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. 15.15 on page 599.

Tags from the areas: peripheral inputs/outputs, inputs, outputs and bit memories, as well as tags from data blocks (global, instance and type data blocks) can be used in watch tables. The current time and count values are displayed with the SIMATIC timer functions (data type TIMER) and the SIMATIC counter functions (data type COUNTER).

In the case of a CPU 300 you can simultaneously monitor up to 30 tags, 14 of which you can control.

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.

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. 15.14 shows monitoring in expanded mode. In the *Monitor with trigger* column, the possible settings are “opened”.

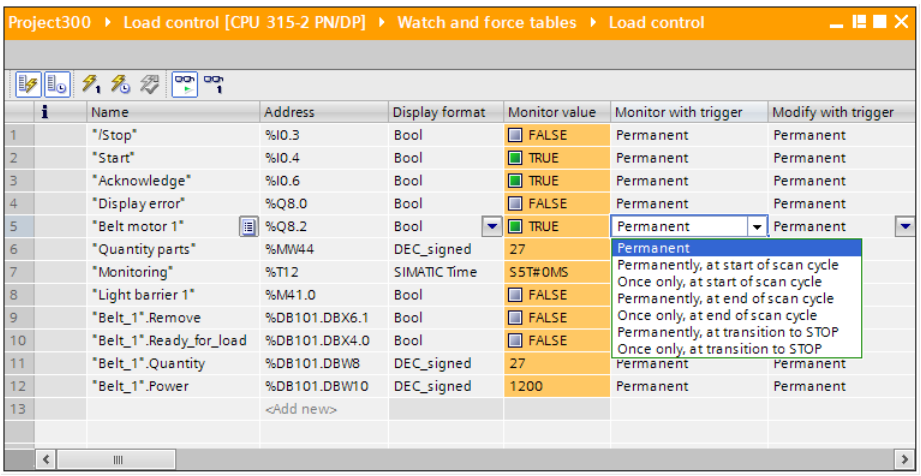


Fig. 15.14 Example of monitoring of tags in expanded mode

Monitoring and modifying with triggers

The watch tables permit specification of the monitoring and control time. The following can be selected:

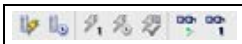
- Permanent
  In each program cycle, the inputs are monitored and controlled at the start of the cycle prior to processing of the main program and the outputs at the end of the cycle following processing of the main program.
- Permanently, at start of scan cycle
  In each program cycle, the tags are 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 additionally possible to control tags using the *Online > Modify > Modify now* command in the main menu bar or the *Modify > Modify now* command in the shortcut menu. The selected tags are then updated as rapidly as possible. Tags can 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 main menu under *Online*, in the shortcut menu, or using the icons in the toolbar of the working window shown in Fig. 15.15.



The icons from left to right:

Name in text	Tooltip text
Basic 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"
Enable PQ	The function "Enable peripheral outputs" disables the output disable (OD).
Monitor all	Monitor all
Monitor now	Monitor all values once and now

Fig. 15.15 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.

In basic mode, the *Name*, *Address*, *Display format*, *Monitor value*, *Modify value*, *Tag selection (represented by a lightning icon)*, 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 598). You can stop the current monitoring by clicking again on the *Monitor all* icon.

Monitoring of I/O

Monitoring of peripheral inputs is only possible in expanded mode. You can activate the expanded mode in the main menu by means of the *Online > Expanded mode* command, in the toolbar of the working window via the *Expanded mode* icon, or by means of the *Expanded mode* command in the shortcut menu.

Please note that the monitoring of peripheral inputs can lead to the cycle time being exceeded and that peripheral outputs can never be monitored.

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.

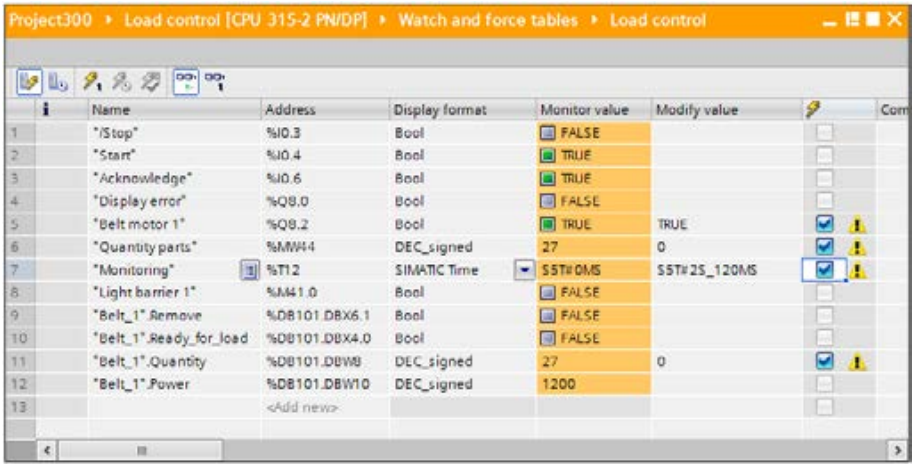
Modifying of tags with watch tables

You can call the test functions in the main menu under *Online*, in the shortcut menu, or using the icons in the toolbar of the working window shown in Fig. 15.15. Double-click to open the watch table. It is recommendable to switch on monitoring. An online connection to the CPU is then already established and the response to the modification can be monitored directly.

In basic mode, the *Name*, *Address*, *Display format*, *Monitor value*, *Modify value*, *Tag selection* (represented by a lightning icon), and *Comment* columns are 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. An example is shown in Fig. 15.16.

Caution! Make sure that no dangerous states can occur when modifying tags!

Please note that reloading of modified data blocks with an ongoing control job can result in unforeseen operating states. The control job continues to modify the specified address, while the address assignment in the data block may possibly have changed. Exit ongoing control jobs prior to the loading of data blocks.



	Name	Address	Display format	Monitor value	Modify value	
1	"/Stop"	%I0.3	Bool	FALSE		
2	"Start"	%I0.4	Bool	TRUE		
3	"Acknowledge"	%I0.6	Bool	TRUE		
4	"Display error"	%Q8.0	Bool	FALSE		
5	"Belt motor 1"	%Q8.2	Bool	TRUE	TRUE	
6	"Quantity parts"	%MW4	DEC_signed	27	0	
7	"Monitoring"	%T12	SIMATIC Time	S5T#0MS	S5T#2S_120MS	
8	"Light barrier 1"	%M41.0	Bool	FALSE		
9	"Belt_1".Remove	%DB101.DBX6.1	Bool	FALSE		
10	"Belt_1".Ready_for_load	%DB101.DBX4.0	Bool	FALSE		
11	"Belt_1".Quantity	%DB101.DBW8	DEC_signed	27	0	
12	"Belt_1".Power	%DB101.DBW10	DEC_signed	1200		
13		<Add new>				

Fig. 15.16 Example of controlling tags

Modify once and now

To modify the activated tags, click on the *Modify now* icon in the toolbar. Alternatively, select the *Online* > *Modify* > *Modify now* command in the main menu or the *Modify* > *Modify now* command in the shortcut menu.

The tags activated in the *Tag selection* column are immediately set (as fast as possible) to the modify 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 modify value again – the yellow triangle appears again in the *Tag selection* column.

Only the tags visible in the table are modified. Multiple modification (multiple input) of a tag in the watch table is not permissible.

Modify to 0 and modify to 1

You can set individual binary tags direct to signal state "0" or signal state "1". Select the tag and then the *Online* > *Modify* > *Modify to 0* or ... > *Modify to 1*command from the main menu or the commands *Modify* > *Modify to 0* and *Modify* > *Modify to 1* from the shortcut menu.

The values 16#0 or 16#1 are transferred for digital tags.

Modifying I/O

Modifying of peripheral outputs is only possible in expanded mode. You can activate the expanded mode in the main menu by means of the *Online* > *Expanded mode* command, in the toolbar of the working window via the *Expanded mode* icon, or by means of the *Expanded mode* command in the shortcut menu.

Please note that the modifying of peripheral outputs can lead to the cycle time being exceeded and that peripheral inputs can never be modified.

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* icon, 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.

15.5.7 Monitoring and modifying in the STOP operating state

In STOP operating state, the output modules are normally disabled by the OD signal (command output disable); the *Enable peripheral outputs* (Enable PQ) function can be used to switch off the OD signal so that you can also control the output modules with the CPU at STOP. Controlling is carried out using a watch table. You can also monitor the signal states of the peripheral inputs in the STOP operating mode. An application for this would be checking the wiring of the peripheral inputs/outputs in the STOP state and without a user program.

Caution! Make sure that no dangerous states can occur when modifying with "Enable PQ"!

Note: The process images are not updated in the STOP operating state. The signal states at the peripheral inputs (at the module terminals) are not transferred to the inputs. Modifying the outputs does not affect the peripheral outputs (the module terminals).

Prerequisites for the monitoring and modifying of I/O in the STOP operating state are:

- The CPU is in the STOP operating state.
- An online connection to the CPU exists.
- A watch table with the peripheral tags has been created and opened.
- The expanded mode has been activated.

Furthermore you must *exit all force jobs* (see Chapter 15.5.8 "Testing with the force table" on page 603)!

You can call the test functions in the main menu under *Online*, in the shortcut menu, or using the icons in the toolbar of the working window shown in Fig. 15.15 on page 599.

You activate monitoring by choosing *Monitor all* (depending on the trigger conditions) or *Monitor now* (once and now). Set the trigger condition "permanent".

For modifying, enter the modify value in the *Modify value* column and activate the selection checkbox. You switch off the command output disable (OD) for the output modules using *Enable PQ*. You can now modify the tags using *Modify now*. You can then control the peripheral outputs for as long as *Enable PQ* is switched on.

You deactivate the *Enable PQ* function – the OD signal is then switched on again – by selecting the *Enable PQ* function again, by changing the CPU mode, or by canceling the online connection.

In Fig. 15.17, the operand %QB8:P (peripheral output byte 8) has been selected for the *Enable PQ* function. "Bin" has been selected as the display format in order to be able to control each bit individually. The icon in the *Monitor value* column indicates that the signal state of peripheral outputs cannot be monitored.

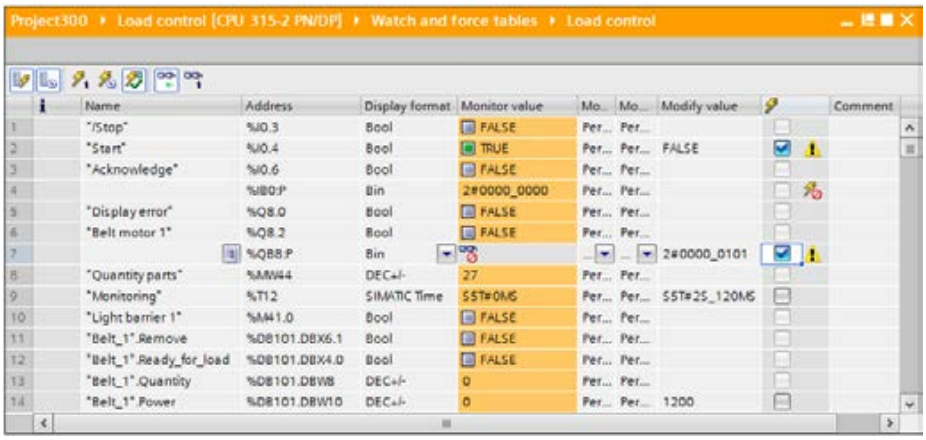


Fig. 15.17 Example of enabling peripheral outputs

15.5.8 Testing with the force table

Tags can be preassigned fixed values. This action is referred to as "forcing". A CPU 300 can force tags from the inputs and outputs area. Note the following special features:

- Forcing with a CPU 300 corresponds to cyclic control: Once the process image input has been updated, the CPU overwrites the inputs with the force value and before the process image output is output, the CPU overwrites the outputs with the force value (Fig. 15.18). The controlling of inputs (e.g. T %IW12) or outputs (e.g. = %Q4.7) changes a force value for the rest of the program cycle. The reading of a peripheral input area (e.g. L %IW14:P) can deliver a value which is different to the force value. When writing to the peripheral output area (e.g. T %QW18:P), the value at the output terminal – and the value in the process image output – can assume a value which is different to the force value for the rest of the program cycle.

- The channel LED on the input module displays the signal state of the input terminal and not the force value. The channel LED on the output module displays the signal state of the output terminal which corresponds to the force value.
- A force job can force a maximum of 10 tags with a CPU 300. Inputs and outputs that are present in a process image partition cannot be forced. Forcing is then only possible if the *Enable PQ* function is switched off.

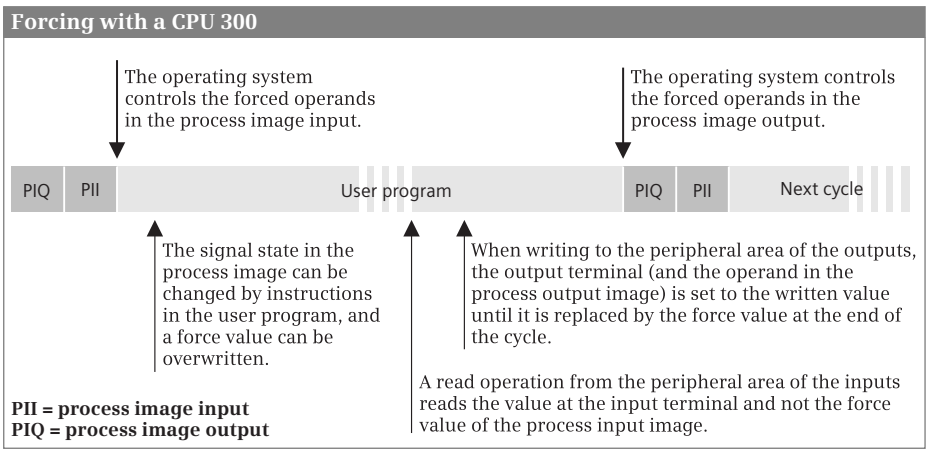


Fig. 15.18 Influence of the user program on forcing with a CPU 300

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. If a force jobs is present in the CPU, the yellow FRCE LED shows a continuous light.

In the force tables you can force peripheral inputs and outputs in data formats BOOL, BYTE, WORD, and DWORD. You can monitor tags from the inputs, outputs, bit memories, and data tags operand areas as well as peripheral inputs. The force table is present once for a CPU and cannot be copied or renamed.

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 the PLC tag table or in a data block. You can also enter the memory address (absolute address) in the *Address* column (Fig. 15.19).

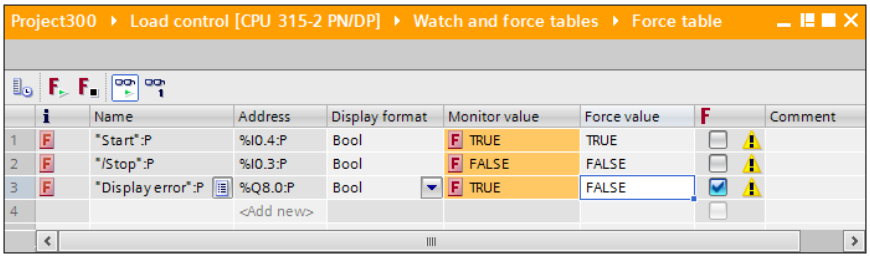


Fig. 15.19 Example of forcing of peripheral inputs and outputs

Monitoring tags in the force table

The entered tags can be monitored in basic mode. 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 15.5.6 “Testing with watch tables” on page 597 for details).

Forcing with the force table

You can call the test functions when forcing via the main menu by means of the *Online* > *Force* > ..., command, via the shortcut menu by means of the *Force* > ... command, or by means of the icons in the toolbar of the working window shown in Fig. 15.20.

To carry out forcing, enter a value in the *Force value* column and activate the checkbox in the *Force* column (tag selection). A yellow triangle with exclamation mark indicates that the selected tag has not yet been forced.

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. 15.20 Icons in the toolbar of the force table

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!

Start forcing

Start forcing sends a force job to the CPU which contains the tags selected for forcing. Forcing is effective immediately.

You can set an individual binary tag in the I/O area direct to signal state “0” or signal state “1”. Select the tag and choose the commands *Force* > *Force to 0* or ... > *Force to 1*. The values 16#0 or 16#1 are transferred for digital tags.

Stop forcing

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 that are visible in the force table using the *Stop forcing* icon. A new force job is sent to the CPU which terminates forcing for all forced tags.

Note that termination of forcing leave the tags in their last state! Only the force job is deleted. For example, an output of a digital module remains in signal state “1” after termination of forcing if it is not controlled otherwise by the program.

The FRCE LED on the CPU turns off when no more force jobs are present in the CPU.

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. SIMATIC S7 uses the PROFIBUS DP, PROFINET IO, and actuator/sensor interface (AS-i) bus systems.

The distributed I/O is handled like the central I/O. The distributed inputs/outputs are in the same address volume as the central inputs/outputs, and therefore the addresses of the distributed I/O must not overlap with those of the central I/O. The distributed I/Os can be addressed via the following operand areas: peripheral inputs (I:P) and peripheral outputs (Q:P) and – if they are present in the process image – also via the inputs (I) and outputs (Q).

Transfer between the distributed modules and the central CPU is carried out “automatically” and you need not take this into account when addressing.

Data transfer to and from the distributed I/O is controlled from a central point: With PROFIBUS DP it is the DP master, with PROFINET IO it is the IO controller, and with AS-Interface it is the AS-i master. The distributed stations – these are the DP slaves with PROFIBUS DP, the IO devices with PROFINET IO, and the AS-i slaves with AS-Interface – are the passive partners in the data transfer.

S7 stations and ET200 stations with a CPU can also be used as distributed I/O stations and these are then “intelligent” DP slaves or IO devices. While these stations are controlling their own modules (considered from their viewpoint as central modules), they also satisfy – when working at the same time as DP slaves or IO devices – the data requirements of the respective DP master or IO controller.

The distributed I/O is configured using the hardware configuration. PROFIBUS DP, PROFINET IO, and actuator/sensor interface (with the CP 343-2P module as AS-i master) are configured as a 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 PROFIBUS DP and PROFINET IO. 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 PROFIBUS DP and PROFINET IO. Depending on their use locally on the machine or in the process, the mechanical properties can be highly different, especially the degree 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 200M

ET 200M is a modular 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. 16.1 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.

16.2.2 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.

Fig. 16.2 ET 200MP

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.

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.

16.2.3 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.

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 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.

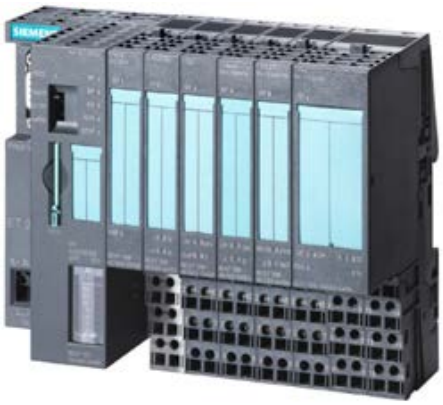


Fig. 16.3 ET 200S with IM 151 CPU

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).

16.2.4 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 I/O modules can be used on a mounting rail in a station; the length of the backplane bus must not exceed 1 m. 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”).

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 I/O modules can be connected to potential groups with individual rooting of the power supply.

Fig. 16.4 ET 200SP

With the interface module IM 155-6 PN ST, an ET 200SP 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 for PROFINET IO 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 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.

16.2.5 ET 200iSP

ET 200iSP is an intrinsically-safe I/O system with degree of protection IP 30 for use in hazardous gas and dust areas, i.e. in zones 1 and 2 as well as 21 and 22, with connection of intrinsically-safe signals from zones 0, 1 or 2 and 20, 21, or 22.

Fig. 16.5 ET 200iSP with redundant IM 152-1

ET 200iSP consists of a power supply module, an interface module, and up to 32 electronic modules for digital and analog inputs/outputs. The modules are snapped onto terminal modules and can be replaced during ongoing operation.

The bus line must also have an intrinsically-safe design in order to operate an ET200iSP intrinsically-safe. This is achieved using an RS 485-IS coupler as isolating transformer. The maximum data transfer rate is 1.5 Mbit/s. ET 200iSP can also be used in redundant mode in fault-tolerant systems.

16.2.6 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. 16.6 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 mod-

ule 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 the easy configuration of 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.

16.2.7 ET 200eco and ET 200eco 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.

Fig. 16.7 ET 200eco PN (left) and ET 200eco with ECOFAST connection

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.

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.

ET 200eco 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 on the PROFINET 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 field devices which are referred to as IO devices (Fig. 16.8).

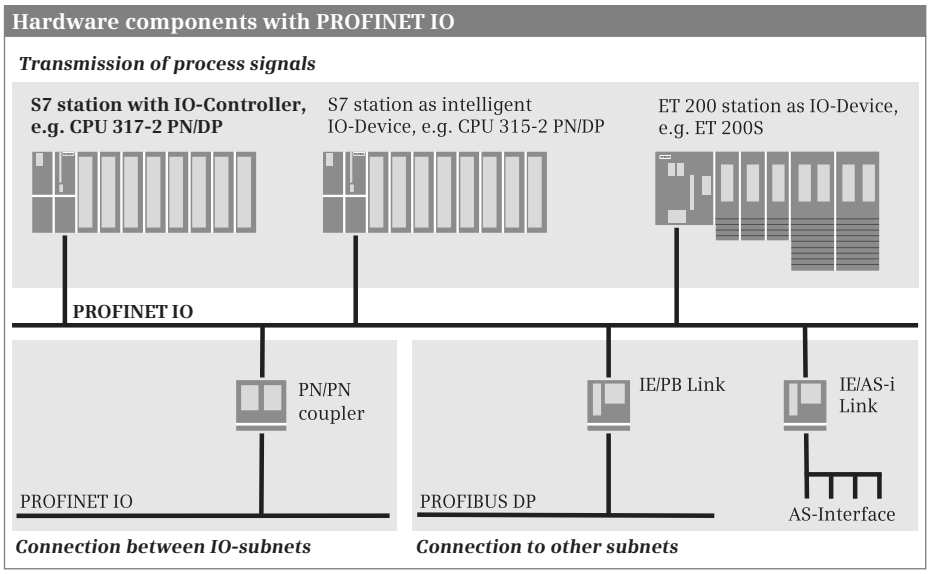


Fig. 16.8 Components of a PROFINET IO system

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. An IO controller can be:

- A CPU with integral PROFINET interface (with the letters “PN” in the short designation, e.g. CPU 317-2 PN/DP)
- A communication module in the PLC station (e.g. CP 343-1)

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 200eco, ET 200M, ET 200S, and ET 200pro.

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 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 PN/PN coupler and the IE/PB Link are described in more detail in Chapter 16.3.5 “Coupling modules for PROFINET IO” on page 623, the IE/AS-i Link in Chapter 16.7.1 “Components of actuator/sensor interface” on page 657.

PROFINET IO system

The IO controller and all IO devices controlled by it constitute a PROFINET IO system (Fig. 16.9). An IO device is supplied with data by its IO controller within an update time which is calculated by STEP 7 in specific intervals and in turn sends its data to the IO controller.

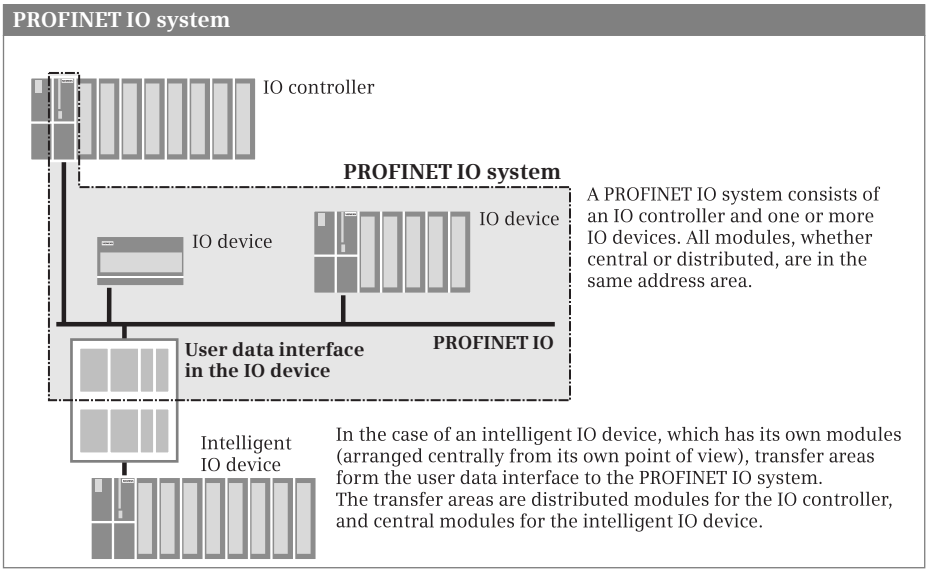


Fig. 16.9 Schematic representation of a PROFINET IO system

Several PROFINET IO systems can be operated in a PN/IE subnet.

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 of four decimal numbers, each in the range from 0 to 255, and is represented by four bytes separated by dots, for example 192.168.1.3. This address consists of the subnet number and the actual station address, which one can extract with the subnet mask from the IP address. Example: If the subnet mask has the value 255.255.255.0, the subnet number for the above-mentioned IP address is 192.168.1 and the station address 3.

Each station on the PROFINET is additionally assigned a device name and number. Further information on the station addresses in an Ethernet subnet can be found in Chapter 3.4.7 “Configuring a PROFINET subnet” on page 85.

Geographic addresses with PROFINET IO

The geographic address identifies the slot of a module. With an IO device, the geographic address comprises the ID of the PROFINET IO system, the device number, the number of the slot, and possibly also a submodule number.

The PROFINET IO system ID is assigned by STEP 7 and is in the range from 100 to 115. Within the station, the “virtual” slot 0 (not physically present) represents the IO device. The modules with the user data are arranged in an IO device starting at slot 1 (Fig. 16.10).

Addresses in a PROFINET IO system

IO controller *(e.g. CPU 315-2 PN/DP)*

Slot	*Module*	*Address/name*
1	(Power supply PS)	
2	CPU	PLC_1
2 X1	MPI/DP interface 1	2047*
2 X2	PROFINET interface_1	2046*
	IO controller	2043*
	IP address	192.168.0.1
	Device name	plc_1
	Device number	0
2 X2 P1	Port_1	2045*
2 X2 P2	Port_2	2044*
3	(Interface module IM)	2000
4	I/O module	log. address
...	...	...

In an S7-300 station, the IO controller is integrated in the CPU. This is inserted in slot 2.

In the example, the PROFINET interface is the second interface of the CPU ("slot" 2 X2). The interface has two ports with the "slots" 2 X2 P1 and 2 X2 P2.

PROFINET IO system

Subnet	Industrial Ethernet
Subnet name	PN/IE_1
Subnet mask	255.255.0.0
S7 subnet ID	B97D-1
IO system name	PLC_1.PROFINET IO system
IO system ID	100

IO device *(e.g. ET 200M)*

Slot	*Module*	*Address/name*
0	Interface module IM	2042*
0 X1	PROFINET interface	2041*
	IP address	192.168.0.2
	Device name	io-device_1
	Device number	1
0 X1 P1	Port_1	2040*
0 X1 P2	Port_2	2039*
1	I/O module	log. address
...	...	...

With an IO device – an ET 200 station in the example – the "virtual" slot 0 represents the station. The PROFINET interface is the only and there the first interface ("slot" 0 X1).

The interface has two ports with the "slots" 0 X1 P1 and 0 X1 P2.

The **geographic address** of a module corresponds to the slot number supplemented by the device number of the IO device and by the PROFINET IO system ID.

The **logical addresses** are used to address the user data of a module. These are input or output addresses depending on the transmission direction. The smallest logical address of a module is the module start address. The automatically assigned logical addresses can be changed. The module addresses must not overlap; they must be unambiguous.

The **diagnostics addresses** are assigned an asterisk in the address overview. STEP 7 assigns the diagnostic addresses automatically during the configuration, starting with the highest input address available. The next lower addresses are then assigned in the sequence of the configuration. The diagnostic addresses can be changed. Just like the input addresses, they must be unambiguous and must not overlap with the other input addresses.

Fig. 16.10 Addresses in a PROFINET IO system

Logical addresses with PROFINET IO

The user data of the IO devices shares the range of logical addresses with the user data of the central modules in the S7 station with the IO controller. The logical addresses of all modules are within the range of peripheral inputs or outputs. This means that the addresses of the central modules must not overlap with those of the IO devices.

You use the logical address to address the user data, in other words the signal states of the digital input/output channels or the values at the analog input/output channels. Each byte of user data is unequivocally defined by the logical address. The logical address corresponds to the absolute address. A symbol (name) can be assigned to it so that it is easier to read (symbolic addressing). Further details can be found in Chapter 4.2 “Addressing of operands and tags” on page 96.

Consistent user data transfer to and from IO devices

Data consistency means that a block of user data is handled together. With PROFINET IO, a block with up to 1024 bytes can be transferred consistently.

A data block addressed in the process image, for example the user data area of a digital input module in the IO device, is transferred consistently during automatic updating of the process image.

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 functions DPRD_DAT (read) and DPWR_DAT (write) for consistent data transfer.

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 93.

Diagnostic addresses with PROFINET IO

Modules and stations with diagnostic data which do not have their own user data address are assigned a diagnostic address. A diagnostic address is within the area of logical input addresses. A diagnostic address can only be used to address diagnostic data records.

During configuration, STEP 7 assigns the diagnostic addresses downwards starting at the highest input address. You can change the diagnostic address. The address overview in the hardware configuration identifies a diagnostic address by means of an asterisk.

Fig. 16.10 shows an example of the diagnostic addresses in an IO system. The PLC station with IO controller is a CPU 315-2 PN/DP in this case with a maximum input address of 2047. This address is assigned to the first interface as the diagnostic address. The next smallest address is then the diagnostic address of the second (PROFINET) interface. Since the ports of the PROFINET interface and the IO controller can also supply diagnostic data, they are assigned the subsequently following diagnostic addresses.

The same principle is applied to an IO device. Slot 0, which presents the IO device, is assigned the diagnostic address which is the highest unused input address at the time of configuration (address 2042 in the example). The PROFINET interface and the ports are then assigned the following addresses. The automatic assignment of the diagnostic addresses is based on the configuration sequence.

The diagnostic data is scanned in the user program by system blocks which use a user data address or diagnostic address for specification of the interrupt-triggering component. For a diagnostic interrupt, for example, the system block RALRM *Read additional interrupt information* can be used in the interrupt handler. You can use the system block RDREC *Read data record* to scan the diagnostic data record DS1, which contains the OB start information and additional component-specific diagnostic data.

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.

You address a transfer area like a peripheral input (I:P) or peripheral output (Q:P). You can address transfer areas with addresses in the area of the process image like inputs (I) or outputs (Q).

16.3.3 Special PROFINET configurations

In the properties of the PROFINET interface, activate the PROFINET functions described below when configuring an IO controller or IO device (see next chapter).

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 devices can participate per ring by means of the Media Redundancy Protocol (MRP) used with SIMATIC S7. The media redundancy must be configured in the interface properties of all participating stations under *Advanced options > Media redundancy*. IRT communication cannot be used if media redundancy is configured.

Changing IO devices during operation

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 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

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.

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).

16.3.4 Configuring PROFINET IO

General procedure

A prerequisite for configuration of the distributed I/O with PROFINET IO is a created project with a PLC station. To select the stations involved, start the hardware configuration in the Network view.

- The starting point for the configuration is the IO controller – either integrated in a CPU 300 with PN interface or in the CP 343-1 communication module. *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. 16.11).

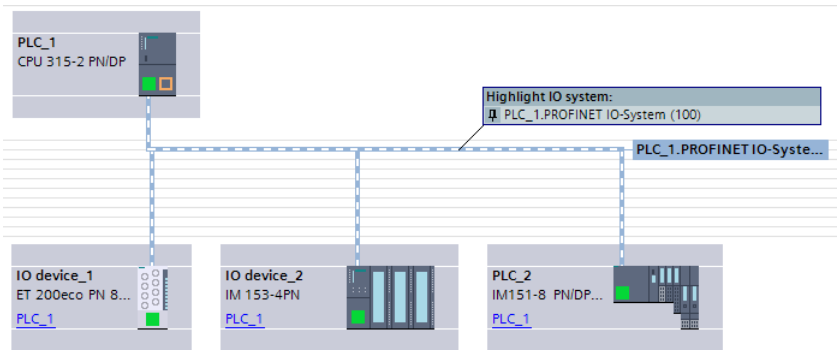


Fig. 16.11 Example of representation of a PROFINET IO system

You then make the parameter settings for the stations and the fitting with input/output modules in the Device view.

Configuring the IO controller in the Network view

Prerequisite: You have created a project and a PLC station, for example a CPU 300 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 3.4.7 “Configuring a PROFINET subnet” on page 85. 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. 16.11 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×24VDC / 1.3A 8×M12* and an ET 200M station from the object tree *Distributed I/O > ET 200M > Interface modules > PROFINET > IM 153-4 PN > ...* .

The interfaces of the IO devices are connected in the graphic with the green/white marking and are thus part of the PROFINET IO system.

The automatically assigned station name is applied as the PROFINET device name. You can change the name in the station properties and also the device number and IP address.

Configuring an IO device

With the IO device selected, you can set its properties in the inspector window in the Device view. You fit a modular IO device with the desired modules or submodules from the hardware catalog and then set their parameters.

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 wish to create an ET200S station as an I-device, drag the interface module with the left mouse button pressed from the object tree

Controllers > *SIMATIC ET 200 CPU* > *ET 200S CPU* > *IM 151-8 PN/DP CPU* > ... 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 PN interface of 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. Select the type (CD) from the drop-down list in the *Type* column and click in the *Data direction* (↔) column on the arrow in order to set the type of transfer area (arrow to right → means input area, arrow to 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.

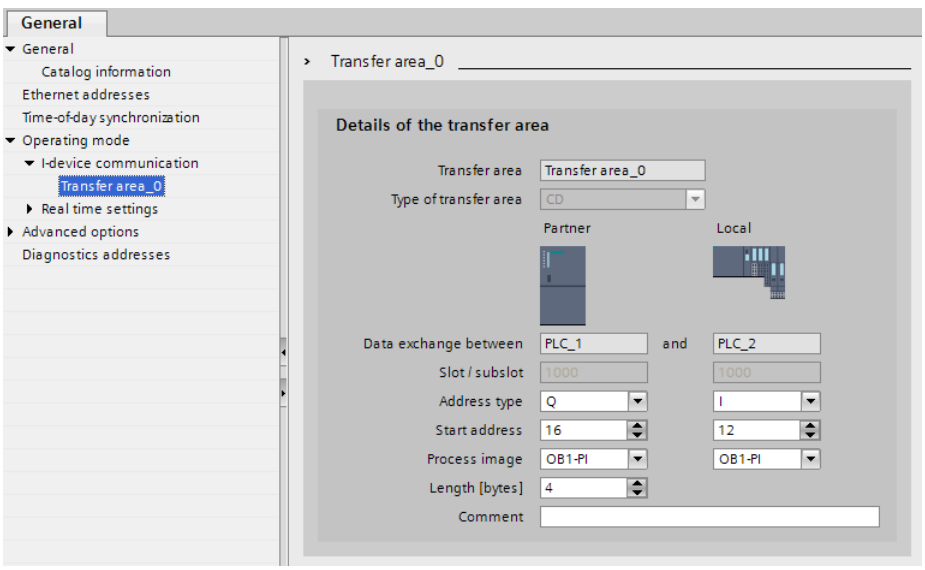


Fig. 16.12 Example of configuration of a transfer area

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. 16.12). If the set address is in the process image, you can select in this display whether the updating is to be carried out in the OB1 process image (OB1-PI) or in the process image partition (PIP 1).

16.3.5 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 coupler* > *PN/PN coupler Vx.0* > ... . 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 sys-

tem, 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 link properties under *Network gateway*. You configure the Ethernet addresses and the real-time settings in the *PROFINET interface* group.

You configure the setting of the PROFINET device number and the assignment to the PROFIBUS station number in the properties of the IE/PB Link. The table shown in the *PROFINET device number* group contains the PROFIBUS station number in the *PB address* column and the device number assigned by STEP 7 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 635.

16.3.6 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 in the PROFINET IO system

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 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. 16.13). 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.

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.

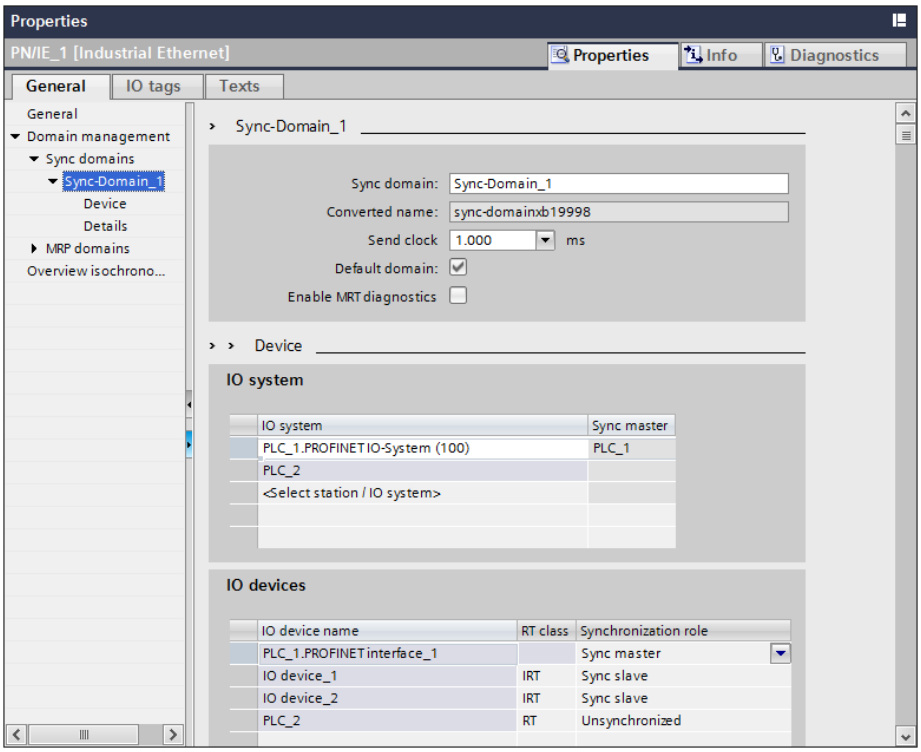


Fig. 16.13 Configuration of a SYNC domain

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 under *Port options* and the respective last station under Boundaries (limits).

In the *Topology view* tab of the hardware configuration, you can graphically configure the port interconnection (with graphs or tables) (Fig. 16.14).

The ports of the configured stations are displayed in the Topology view. 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 comparison* button.

The properties of the port connection are shown in the inspector window if you select a port in the graphic or in the table.

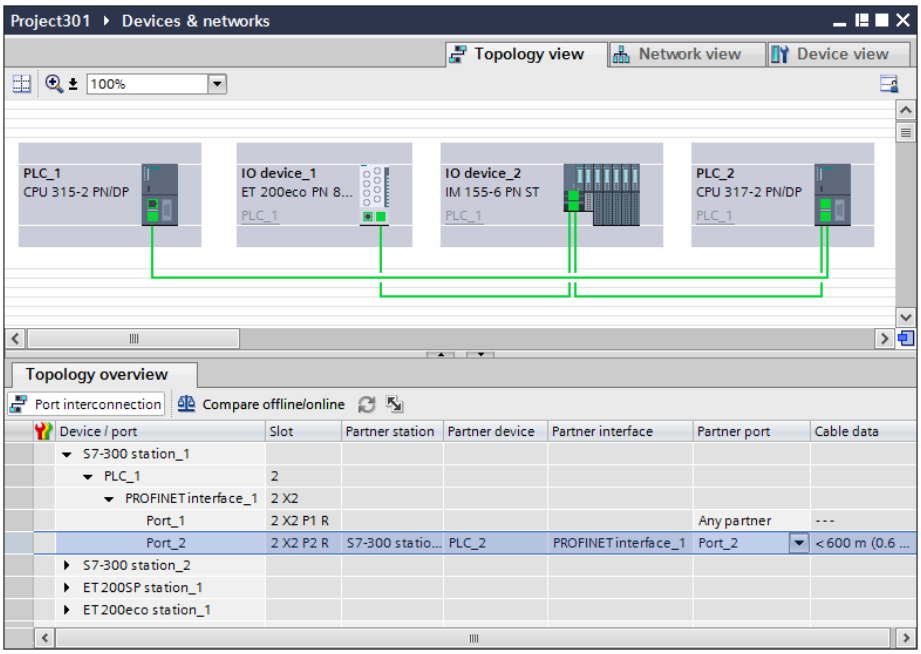


Fig. 16.14 Graphical and tabular view of the PROFINET topology

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. 16.15).

The PROFIBUS network can be designed physically as an electrical network, optical network, or wireless coupling with different data transfer rates. The length of a segment depends on the transfer rate and is adjustable in steps for an electrical or optical network from 9.6 Kbit/s to 12 Mbit/s. The electrical network can be configured as a bus or tree structure. It uses a shielded, twisted two-wire cable (RS485 interface).

The optical network uses either plastic, PCF or glass fiber-optic cables. It is suitable for long distances, offers galvanic isolation, and is impervious to electromagnetic influences. Using optical link modules (OLMs) it is possible to construct a linear, ring, or star topology. An OLM also provides the connection between electrical and optical networks with a mixed design. A cost-optimized version is the design as a linear topology with integral interface and optical bus terminal (OBT).

Using the PROFIBUS Infrared Link Module (ILM), a wireless connection can be provided for one or more PROFIBUS slaves or segments with PROFIBUS slaves. The

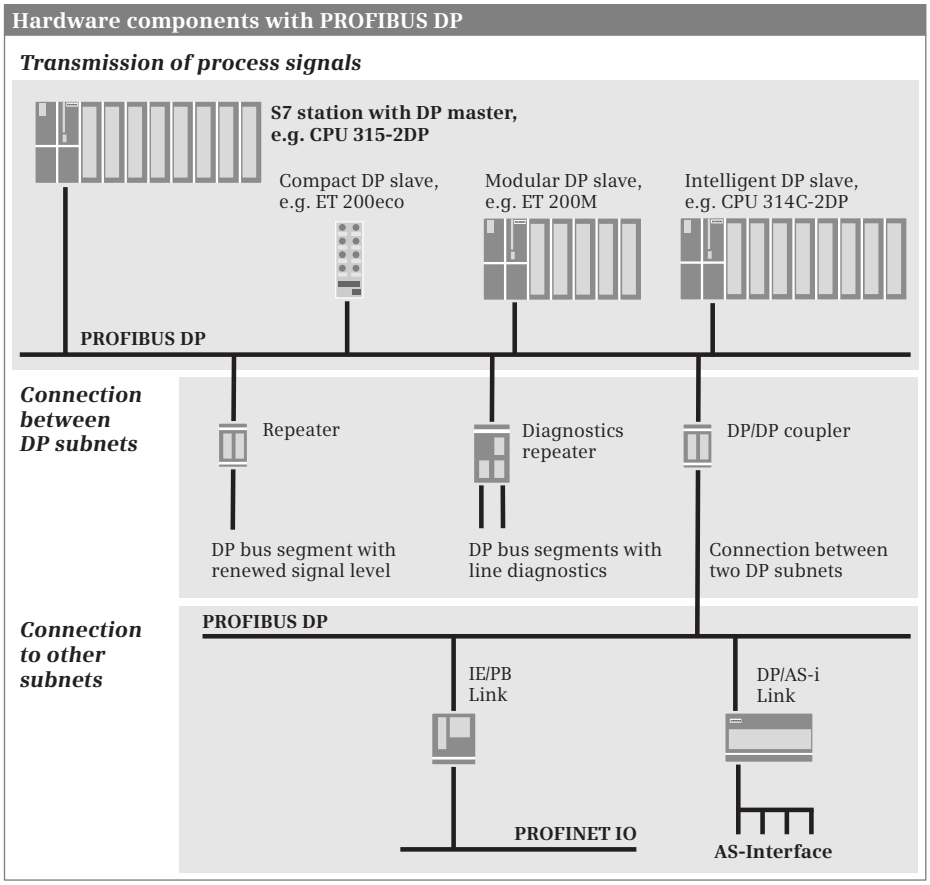


Fig. 16.15 Hardware components with PROFIBUS DP

maximum data transfer rate of 1.5 Mbit/s and the maximum range of 15 m mean that communication is possible with moving system components.

DP master

The DP master is the active station on the PROFIBUS. It exchanges data cyclically with "its" DP slaves. A DP master can be:

- A CPU with integral PROFIBUS interface (with the letters "DP" in the short designation, e.g. CPU 315-2 PN/DP)
- A communication module in the PLC station (e.g. CP 342-5)
- 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. Examples of intelligent DP slaves are S7 stations with CPUs having an integral DP slave functionality, as well as the ET 200S distributed I/O station with the IM 151-7 CPU interface and the ET 200pro distributed I/O station with the IM 154-8 PN/DP CPU interface.

A CPU which is configured as an intelligent DP slave cannot be a DP master at the same time. However, a CP 342-5 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 and the DP/DP coupler are described in more detail in Chapter 16.4.4 “Coupling modules for PROFIBUS DP” on page 638, the DP/AS-i Link in Chapter 16.7.1 “Components of actuator/sensor interface” on page 657, and the IE/PB Link PN IO in Chapter 16.3.5 “Coupling modules for PROFINET IO” on page 623.

PROFIBUS DP master system

The DP master and all DP slaves controlled by it form a PROFIBUS DP master system (Fig. 16.16). 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.

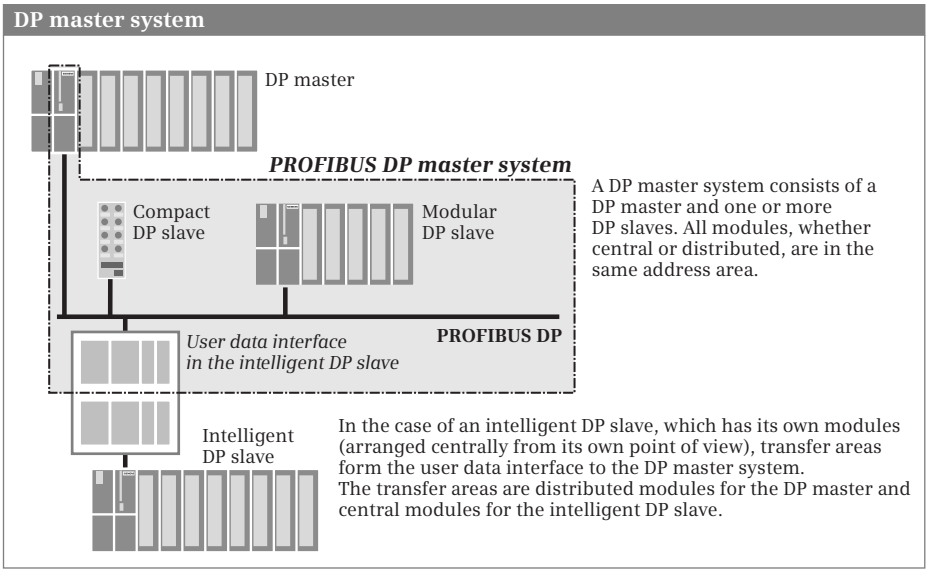


Fig. 16.16 Schematic representation of a PROFIBUS DP master system

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 new types of interrupt. PROFIBUS devices from Siemens ("DP S7 slaves"), which can handle further functions in addition to the cyclic data exchange, e.g. diagnostic interrupts, have the operating mode "S7-compatible".

The operating modes of DP master and DP slaves must be matched to each other. DP masters in operating mode "DPV0" control DPV0 slaves, those in operating mode "S7-compatible" control DPV0 and DP S7 slaves. DPV1 masters from Siemens can control DP slaves with all operating modes.

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.

STEP 7 assigns the station addresses automatically and you can change the addresses within the specified range. You set the highest station address in the properties of the subnet or DP master system under *Network settings*.

Geographic address with PROFIBUS DP

The geographic address identifies the slot of a module. With a DP slave, the geographic address comprises the ID of the DP master system, the station number, and the slot number.

The DP master system ID is assigned by STEP 7 and is in the range from 1 to 32 for a DP master integrated in the CPU and in the range from 180 to 195 for the CP 342-5 as DP master.

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 96.

Consistent user data transfer to and from DP slaves

Data consistency means that a block of user data is handled together without interruption. With PROFIBUS DP and a CPU 300, a block with a maximum of 32 bytes can be transferred consistently.

A data block addressed in the process image, for example the user data area of a digital output module in the DP slave, is transferred consistently during automatic updating of the process image.

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 functions DPRD_DAT (read) and DPWR_DAT (write) for consistent data transfer.

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 93.

Diagnostic addresses with PROFIBUS DP

Modules and stations with diagnostic data which do not have their own user data address are assigned a diagnostic address. A diagnostic address is within the area of logical input addresses. A diagnostic address can only be used to address diagnostic data records.

During configuration, STEP 7 assigns the diagnostic addresses downwards starting at the highest input address. You can change the diagnostic address. The address overview in the hardware configuration identifies a diagnostic address by means of an asterisk.

Fig. 16.17 shows an example of the diagnostic addresses in an DP master system. The PLC station with the integral DP master is a CPU 315-2 PN/DP in this case with a maximum input address of 2047. This address is assigned to the first interface as the diagnostic address. The next smallest address is then the diagnostic address of the second interface. Since the ports of this second interface can also deliver diagnostic data, they are assigned the subsequently following diagnostic addresses.

A compact DP slave has one diagnostic address for the complete station, as with a modular DPV1 slave. A DP S7 slave with the S7-300 slot model has one diagnostic address for the station and one for the interface module.

The diagnostic data is scanned in the user program by system blocks which use a user data address or diagnostic address for specification of the interrupt-triggering component. For a diagnostic interrupt, for example, the system block RALRM *Read additional interrupt information* can be used in the interrupt handler. You can use the system block RDREC *Read data record* to scan the diagnostic data record DS1, which contains the OB start information and additional component-specific diagnostic data.

Diagnostic addresses with intelligent DP slaves

In addition to the logical addresses of the transfer areas, the user data interface has one diagnostic address for device diagnostics and one for signaling mode transitions. You can find these diagnostic addresses in the properties of the DP interface under *Operating mode* > *I-slave communication* in the area *Diagnostics address of communication*. The DP master obtains information about the status of the

Addresses in a PROFIBUS DP master system

DP master *(e.g. CPU 315-2 PN/DP)*

Slot	*Module*	*Address/name*
1	(Power supply PS)	
2	CPU module	PLC_1
2 X1	MPI/DP interface 1	2047*
2 X2	PROFINET interface 1	2046*
2 X2 P1	Port_1	2045*
2 X2 P2	Port_2	2044*
3	(EU interface module IM)	2000
4	I/O module	log. address
...	...	...

In an S7-300 station, the DP master is integrated in the CPU module. This is inserted in slot 2.

In the example, the combined MPI/PROFIBUS interface is the first interface of the CPU module ("slot" 2 X1). It is assigned the highest input address as the diagnostics address.

The logical addresses for the signal modules commence with slot 4.

PROFIBUS DP master system

Subnet	PROFIBUS
Subnet name	PROFIBUS_1
S7 subnet ID	E265-1
DP master system name	PLC_1.DP master system
DP master system ID	1

DP slave_1 *(e.g. ET 200eco)*

Slot	*Module*	*Address*
0	Interface module IM	2043*
1	I/O module	log. Address

With a compact DP slave an ET 200eco station in the example the "virtual" slot 0 represents the station.

DP slave_2 *(e.g. ET 200M)*

Slot	*Module*	*Address*
0	ET 200M station	2041*
1	–	
2	Interface module IM	2042*
3	–	
4	I/O module	log. Address
...	...	...

With a modular DP slave an ET 200M station with the S7-300 slot model in the example the "virtual" slot 0 represents the station. The interface module is present in slot 2.

The logical addresses for the signal modules commence with slot 4.

The **geographic address** of a module corresponds to the slot number supplemented by the station number of the DP slave and by the DP master system ID. In the (modular) S7-300 slot model, the I/O modules are addressed commencing with slot 4, otherwise commencing with slot 1.

The **logical addresses** are used to address the user data of a module. These are input or output addresses depending on the transmission direction. The smallest logical address of a module is the module start address. The automatically assigned logical addresses can be changed. The module addresses must not overlap, they must be unambiguous.

The **diagnostics addresses** are assigned an asterisk in the address overview. STEP 7 assigns the diagnostics addresses automatically during the configuration, starting with the highest input address available. The next lower addresses are then assigned in the sequence of the configuration. The diagnostics addresses can be changed. Just like the input addresses, they must be unambiguous and must not overlap with the other input addresses.

Fig. 16.17 Addresses in a DP master system

DP slaves via the master address. The DP slave obtains information about the status of the DP master via the slave address.

User data interface with intelligent DP slaves

With the compact and modular DP slaves, the addresses of the inputs and outputs are together with the addresses of the central modules in the address volume of the DP master. With intelligent DP slaves (abbreviated to: I-Slaves), the input/output modules of the DP slaves are assigned to the slave CPU. Every intelligent DP slave therefore has a user data interface as common memory area with the DP master whose size depends on the slave CPU used.

The user data interface can be divided into several areas of different length and data consistency. The individual areas then respond like modules whose lowest address is the module start address. From the viewpoint of the DP master, the I-slave then appears like a compact or modular DP slave depending on the division.

A transfer area which is represented as an input module from the viewpoint of the DP master is an output module from the viewpoint of the DP slave and vice versa. The logical addresses on the master side are in the address volume of the DP master and the logical addresses on the slave side in the address volume of the DP slave. The addresses on the master side can be different from those on the slave side.

You address a transfer area like a peripheral input (I:P) or peripheral output (Q:P). You can address transfer areas with addresses in the area of the process image like inputs (I) or outputs (Q).

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 300 with DP interface or as the CP 342-5 communication module.
- If the interface is a combined MPI/DP interface, set the interface type to PROFIBUS in the interface properties.
- Activate the *DP master* mode of the DP interface.
- 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.
- ▷ You drag an intelligent DP slave as a stand-alone PLC station into the working window, set the interface type to PROFIBUS (with a combined MPI/DP interface), 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. 16.18).

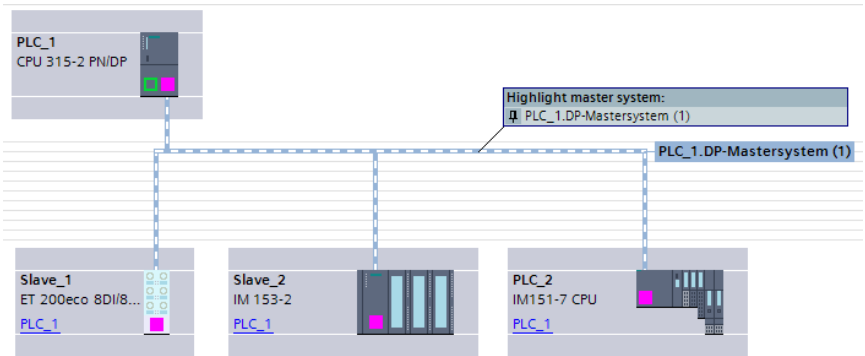


Fig. 16.18 Example of representation of a PROFIBUS DP master system

You then make the parameter settings for the stations and the fitting with input/output modules in the Device view.

Configuring the DP master in the Network view

Prerequisite: You have created a project and a PLC station, for example a CPU 300 with DP interface. Start the device configuration and select the *Network view* tab in the working window.

If the CPU has a combined MPI/DP interface, you must first change over the interface since this combined interface is set as standard to MPI mode. Select *PROFIBUS* as the interface type in the interface properties under *MPI address* in the *Parameters* box.

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-Mastersystem (<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. 16.18 shows two stations of the distributed I/O: An ET 200eco station from the object tree *Distributed I/O* > *ET 200eco* > *PROFIBUS* > *DI/DO* > *8DI/8DO* > ... and an ET 200M station from the object tree *Distributed I/O* > *ET 200M* > *Interface modules* > *PROFIBUS* > *IM 153-2* > ... .

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 *Startup 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 wish to create an ET200S station as an I-slave, drag the interface module with the left mouse button pressed from the object tree *Controllers* > *SIMATIC ET 200 CPU* > *ET 200S CPU* > *IM 151-7 CPU* > ... to the working window.

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 S7-300 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 DP interface of the CPU or the ET station in the working win-

dow 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. Select the type (MS) from the drop-down list in the *Type* column and click in the *Data direction* (↔) column on the arrow in order to set the type of transfer area (arrow to right → means input area, arrow to left ← means output area from the viewpoint of the I-slaves).

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. 16.19).

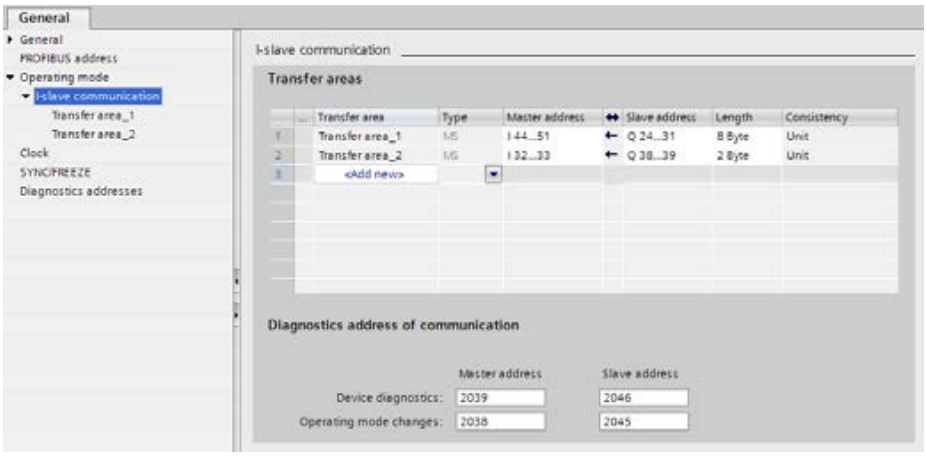


Fig. 16.19 Example of configuration of the transfer areas of an I-slave

In this manner you can configure further transfer areas. The configured transfer areas are displayed in the *I-slave communication* properties group. If you click a transfer area here, you obtain its details. If the set address is in the process image, you can select in this display whether the updating is to be carried out in the OB1 process image (OB1-PI) or in the process image partition (PIP 1).

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.

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 STEP 7. You can find it in the hardware catalog under *Network components* > *Diagnostics repeater* > ... .

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 STEP 7. 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.

16.4.5 Special functions for PROFIBUS DP

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

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. Exception: If the CP 342-5 communication module is the DP master, this limitation does not exist.

Using the system block DPSYC_FR in the user program, you can trigger the output of a command to a group (see Chapter 16.6 “System blocks for distributed I/O” on page 648). 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.

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 Isochronous mode

16.5.1 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.

16.5.2 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. 16.20).

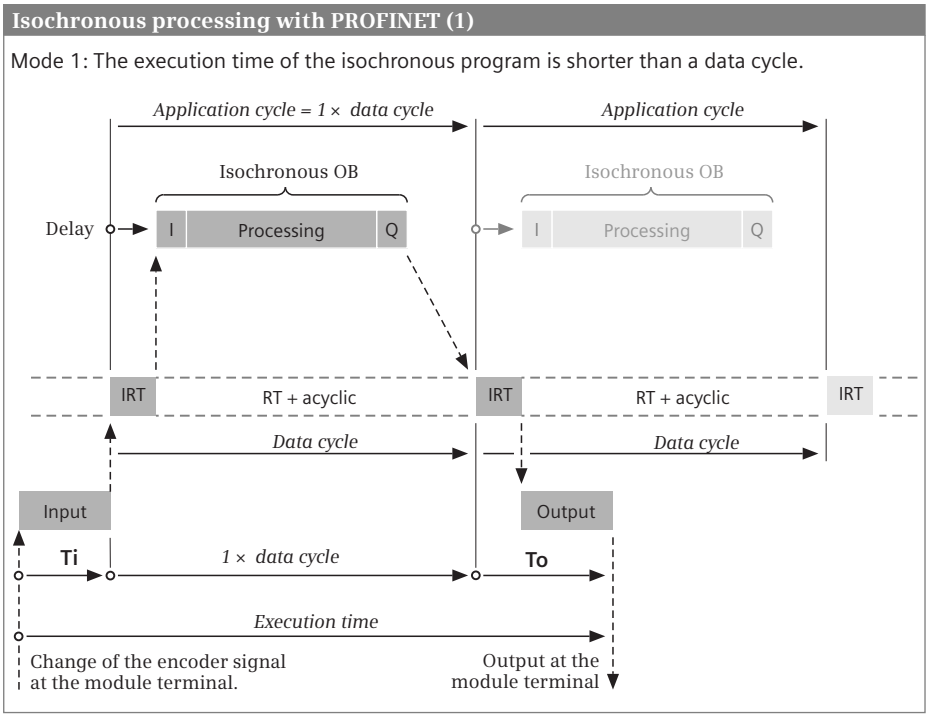


Fig. 16.20 Isochronous mode in the PROFINET IO system (1)

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 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. 16.20); 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. 16.21).

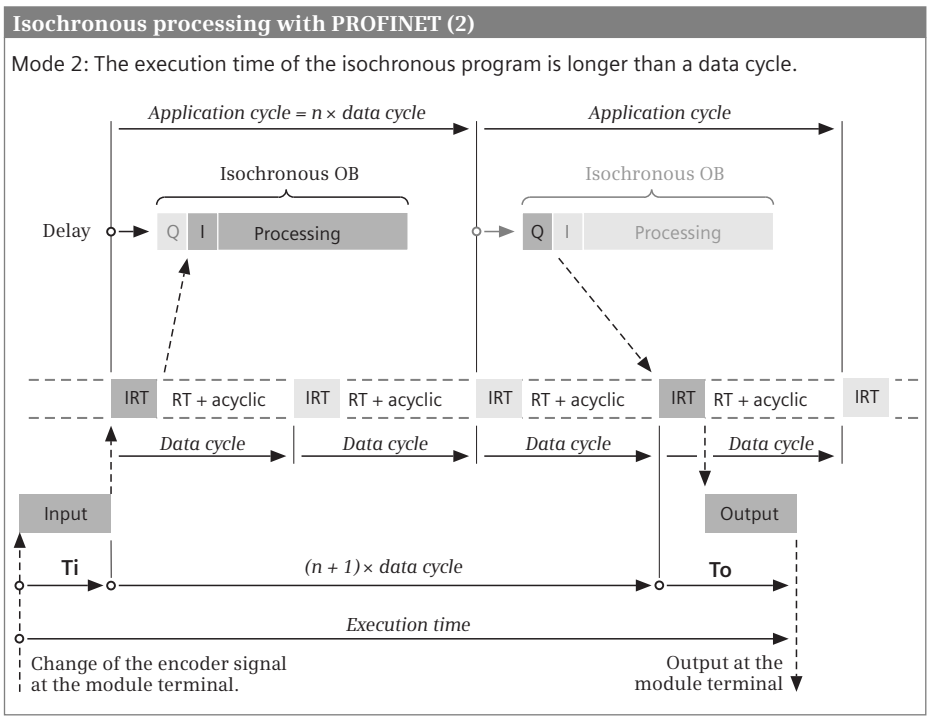


Fig. 16.21 Isochronous mode in the PROFINET IO system (2)

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.

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. 16.21), 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 the IO controller and the IO devices, for example with a CPU 315-2 PN/DP as IO controller, an IO device ET 200MP with IM 155-5 PN ST, and an IO device ET 200S with IM 151-3BA60 (see Chapter 16.3.4 “Configuring PROFINET IO” on page 619).

Configure a sync domain with IRT communication with the CPU 315 as sync master and the IO devices as sync slave (see Chapter 16.3.6 “Real-time communication in PROFINET” on page 624).

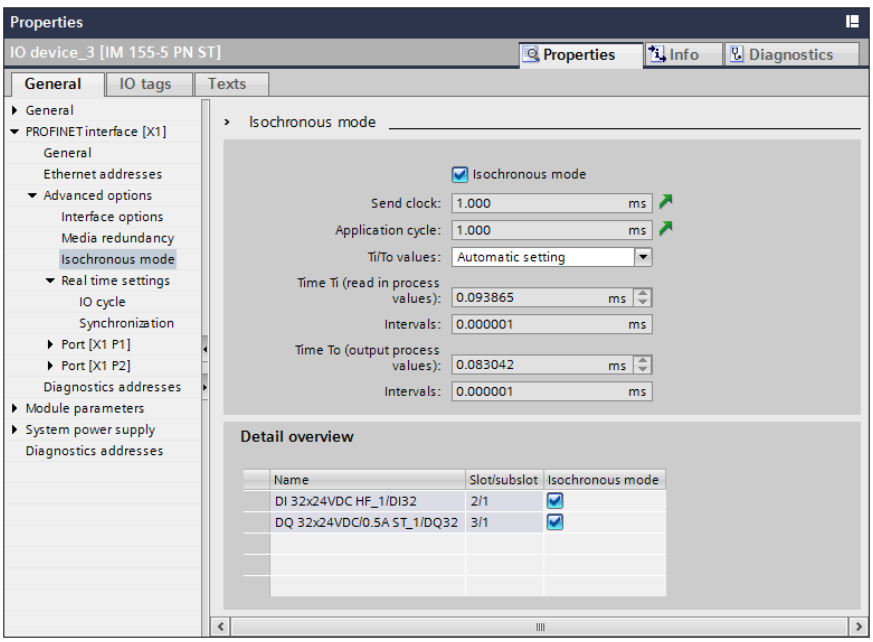


Fig. 16.22 Configuration of isochronous mode for ET 200MP

In the properties of the CPU, assign the isochronous mode organization block OB 61 to the PROFINET IO system and the process image partition PIP 1 (see Chapter 5.6.8 “Isochronous mode interrupt, organization block OB 61” on page 197).

In the properties of the interface module (IM) in the IO device, select the entry *PROFINET interface > Advanced options > Isochronous mode* (Fig. 16.22). Activate the *Isochronous mode* checkbox and set the method of determining the Ti/To values: *Automatic setting*, *From OB*, or *Manual*. In the detail overview you identify the I/O modules that participate in isochronous mode. In the module properties under *I/O addresses*, you assign the user data of this module to the process image partition PIP 1.

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*.

16.5.3 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 organization block OB 61. 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. 16.23 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

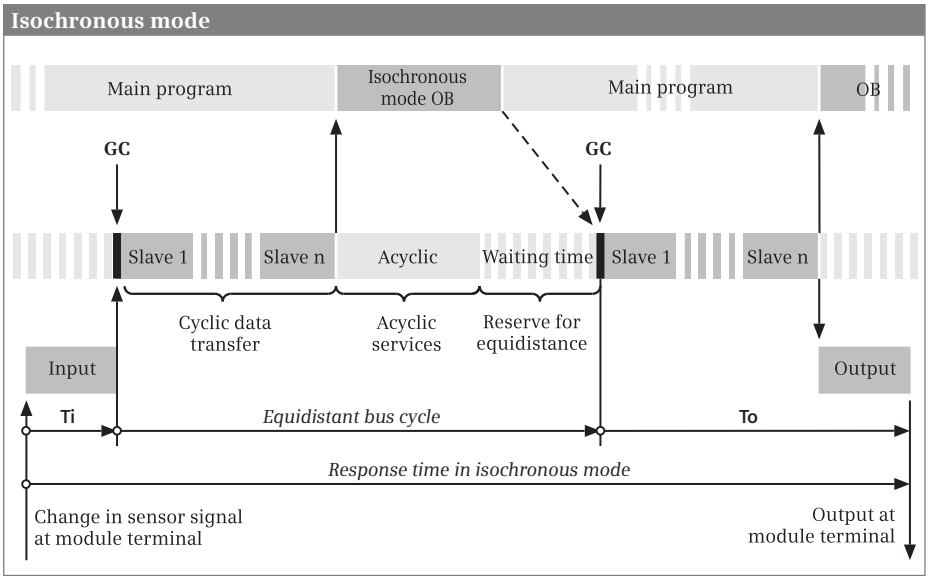


Fig. 16.23 Response time with constant bus cycle time and isochronous mode

transfer using the global control command (GC) is available. The equidistant bus 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 with the DP master and the DP slaves (see Chapter 16.4.3 “Configuring PROFIBUS DP” on page 635).

In the properties of the CPU, assign the isochronous mode organization block OB 61 to the DP master system and the process image partition PIP 1 (see Chapter 5.6.8 “Isochronous mode interrupt, organization block OB 61” on page 197).

In the properties of the interface module (IM) in the DP slave, select the entry *Isochronous mode* (Fig. 16.24). Activate the *Synchronize DP slave with constant DP bus cycle time* checkbox and set the method of determining the Ti/To values: *From subnet*, *Automatic minimum*, or *Manual*. In the detail overview you identify the I/O modules that participate in isochronous mode. In the module properties under *I/O addresses*, you assign the user data of this module to the process image partition PIP 1.

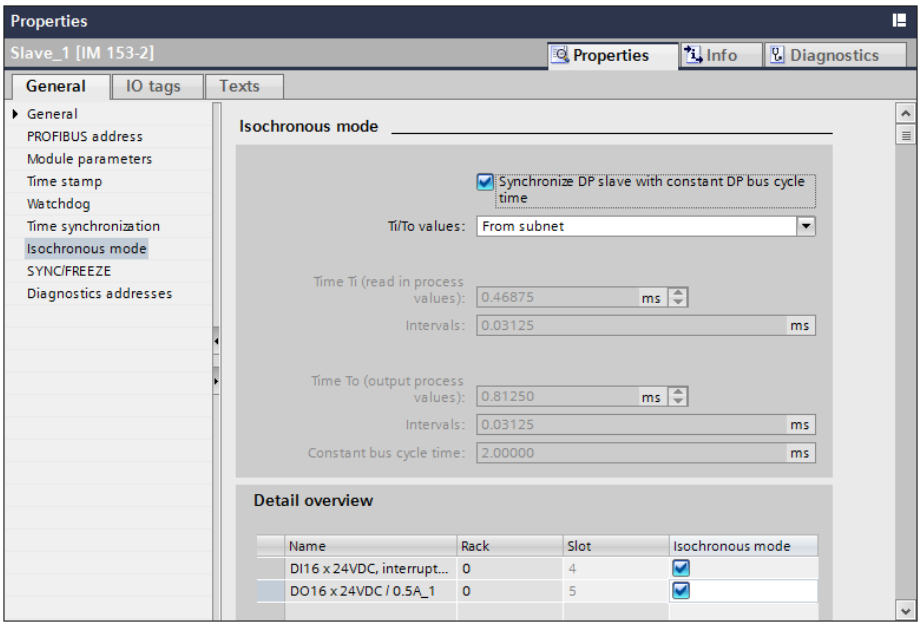


Fig. 16.24 Activation of isochronous mode in a DP slave

In the properties of the PROFIBUS subnet, you are given an overview of the set values and the list of the modules participating in the isochronous mode under *Constant bus cycle time*.

16.6 System blocks for distributed I/O

16.6.1 System blocks for PROFIBUS DP

The following system blocks can be used together with PROFIBUS DP:

- SALRM Trigger interrupt with DP master (SFB 75)
- DP_PRAL Trigger hardware interrupt with DP master (SFC 7)
- DPSYC_FR Send SYNC/FREEZE commands (SFC 11)
- DPNRM_DG Read diagnostic data from a DP standard slave (SFC 13)
- DP_TOPOL Determine bus topology (SFC 103)

Fig. 16.25 shows the graphic representation of the system block calls for PROFIBUS DP.

In addition to the blocks listed above, you can use the following blocks with PROFIBUS DP and also with PROFINET IO:

- GETIO Read all inputs of a station (FB 20)
- SETIO Write to all outputs of a station (FB 21)
- GETIO_PART Read some inputs of a station (FB 22)
- SETIO_PART Write some outputs of a station (FB 23)
- D_ACT_DP Activate/deactivate distributed station (SFC 12)
- DPRD_DAT Read user data (SFC 14)
- DPWR_DAT Write user data (SFC 15)

These blocks are described in Chapter 16.6.2 “System blocks for PROFIBUS DP and PROFINET IO” on page 652.

SALRM Trigger interrupt with DP master

SALRM triggers a diagnostic or hardware interrupt from the user program of an intelligent slave for the corresponding DP master. You define the type of interrupt using the ATYPE parameter. The system block can only be used in S7-compatible mode.

REQ = “1” triggers the interrupt request; the DONE, BUSY, ERROR, and STATUS parameters show the job status. The job is completed (BUSY = “0”) when processing of the interrupt OB has been finished in the DP master.

The user data interface between DP master and intelligent DP slave can be divided into individual transfer areas which represent individual modules from the viewpoint of the master CPU. You can trigger an interrupt in the DP master for each of these “virtual” slots. You specify the address range using the ID parameter, which you assign a user data address from the viewpoint of the slave CPU. Bit 15 contains the I/O ID: “0” corresponds to an input address, “1” to an output address. The start

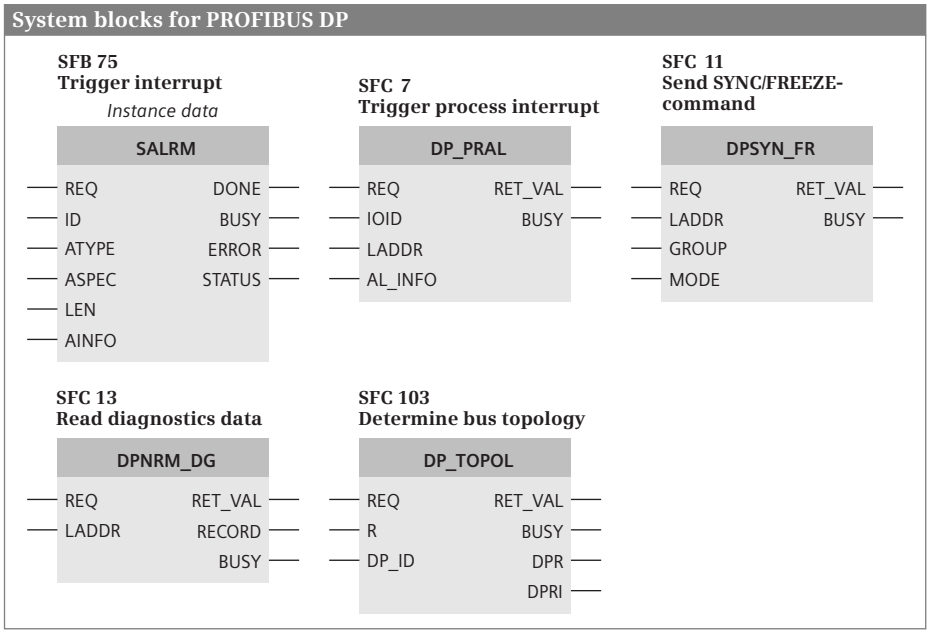


Fig. 16.25 Graphic representation of system blocks for PROFIBUS DP

information of the interrupt OB then contains the addresses of the interrupt-triggering "modules" from the viewpoint of the master CPU.

You can use the AINFO parameter to transfer self-defined additional interrupt information which can be evaluated in the interrupt OB of the master CPU. AINFO is specified as an ANY pointer to a data area. The length of the information sent is determined by the LEN parameter and by the area length of the ANY pointer (the shorter length is decisive). The first 4 bytes are displayed in the start information of the interrupt OB in the master CPU in bytes 8 to 11 (the OB40_POINT_ADDR tag for hardware interrupts, data record DS 0 for the diagnostic interrupt). You can use RALRM to read the complete additional interrupt information in the master CPU.

DP_PRAL Trigger hardware interrupt with DP master

DP_PRAL triggers a hardware interrupt from the user program of an intelligent slave for the corresponding DP master. This results in starting of organization block OB 40 in the program of the master CPU.

You can use the AL_INFO parameter for a self-defined interrupt ID which is transferred to the start information of the interrupt OB called in the DP master (OB40_POINT_ADDR tag). REQ = "1" triggers the interrupt request; the RET_VAL and BUSY parameters show the job status. The job is completed when processing of the interrupt OB has been finished in the master CPU.

The user data interface between DP master and intelligent DP slave can be divided into individual transfer areas which represent individual “modules” from the viewpoint of the master CPU. The lowest address of a transfer area is the module start address. You can trigger a hardware interrupt in the master CPU for each of these “virtual” slots.

You specify a transfer area using the IOID and LADDR parameters from the viewpoint of the slave CPU. The start information of the interrupt OB then contains the addresses of the interrupt-triggering “module” from the viewpoint of the master CPU.

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. 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). The commands in the MODE parameter are also organized in bits:

- UNFREEZE if bit 2 = “1”
- FREEZE if bit 3 = “1”
- UNSYNC if bit 4 = “1”
- SYNC if bit 5 = “1”

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 restart.

DPNRM_DG Read diagnostic data

DPNRM_DG reads the diagnostic data of a DP standard slave. 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. Tags with data types ARRAY and STRUCT, a PLC data type, or an ANY pointer with data type BYTE (e.g. P#DBzDBXy.x BYTE nnn) are permissible as actual parameters.

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.

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. 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).

The diagnostics repeater is described in Chapter 16.4.4 “Coupling modules for PROFIBUS DP” on page 638.

16.6.2 System blocks for PROFIBUS DP and PROFINET IO

The following system blocks can be used with PROFIBUS DP and PROFINET IO:

- GETIO Read all inputs of a station (FB 20)
- SETIO Write to all outputs of a station (FB 21)
- GETIO_PART Read some inputs of a station (FB 22)
- SETIO_PART Write some outputs of a station (FB 23)
- D_ACT_DP Activate/deactivate distributed station (SFC 12)
- DPRD_DAT Read user data consistently (SFC 14)
- DPWR_DAT Write user data consistently (SFC 15)

Fig. 16.26 shows the graphic representation of the system blocks for PROFIBUS DP and PROFINET IO.

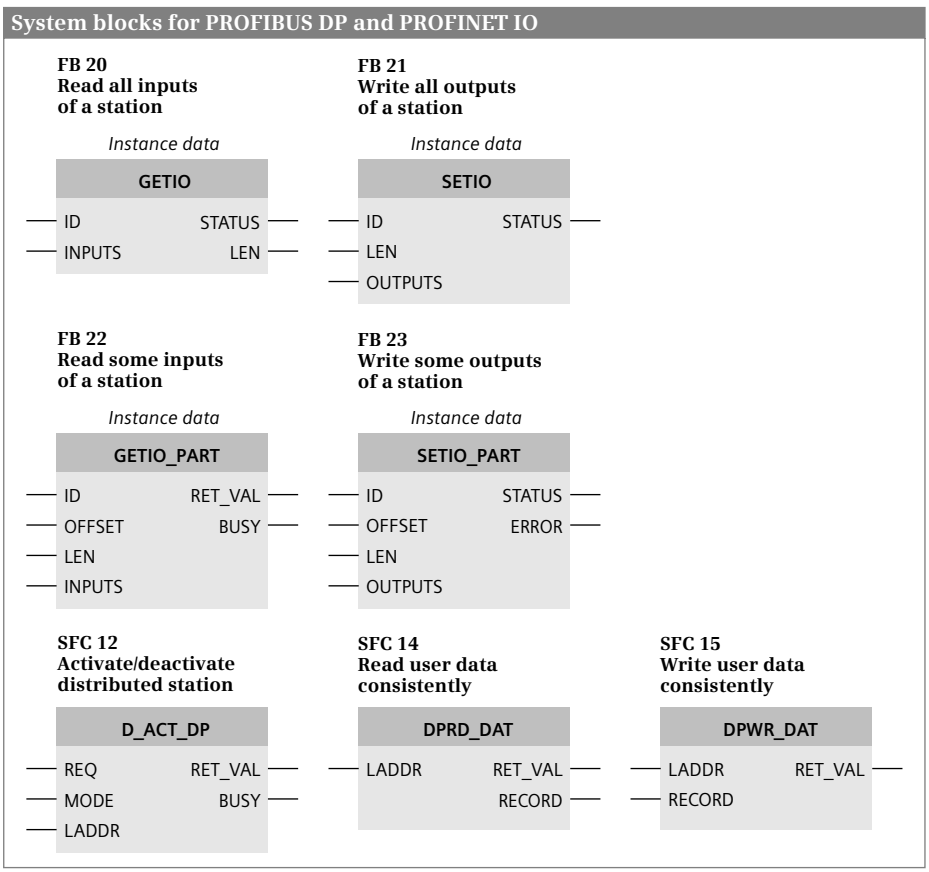


Fig. 16.26 Graphic representation of system blocks for PROFIBUS DP and PROFINET IO

GETIO Read all inputs of a station

GETIO uses DPRD_DAT to consistently read all input data of a DP standard slave or an IO device or all data of an input area in the case of modular stations. The right-hand word of the ID parameter contains the start address of the input area to be read.

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.

SETIO Write to all outputs of a station

SETIO uses DPWR_DAT to consistently write all output data to a DP standard slave or IO device or all data of an output area in the case of modular stations. The right-hand word of the ID parameter contains the start address of the output area to be written to.

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. This is why information in the LEN parameter is irrelevant.

GETIO_PART Read some inputs of a station

GETIO_PART uses UBLKMOV to consistently read some of the input data of a DP standard slave or IO device or some of the data of an input area in the case of modular stations. The right-hand word of the ID parameter contains the start address of the input area, the OFFSET parameter contains the number of the first byte to be read, and the LEN parameter contains the number of bytes.

The input bytes to be read must be addressed in the process image input in order to use GETIO_PART. Please ensure that the OFFSET and LEN parameters do not violate any boundaries with neighboring data of other stations.

If the destination area specified by the INPUTS parameter is smaller than the input area read, the function only transfers as many bytes as can be written to the destination area. If the destination area is larger, only the first LEN bytes of the area are written to. In both cases, no error is indicated on the ERROR parameter. ERROR only has signal state “1” if UBLKMOV signals an error.

SETIO_PART Write some outputs of a station

SETIO_PART uses UBLKMOV to consistently write some of the output data to a DP standard slave or IO device or some of the data of an output area in the case of modular stations. The right-hand word of the ID parameter contains the start address of the output area, the OFFSET parameter contains the number of the first byte to be written, and the LEN parameter contains the number of bytes.

The output bytes to be written must be addressed in the process image output in order to use SETIO_PART. Please ensure that the OFFSET and LEN parameters do not violate any boundaries with neighboring data of other stations.

If the source area specified by the OUTPUTS parameter is smaller than the output area to be written, the function only transfers as many bytes as the source area contains. If the source area is larger, only the first bytes specified at the LEN parameter are transferred. In both cases, no error is indicated on the ERROR parameter. ERROR only has signal state “1” if UBLKMOV signals an error.

D_ACT_DP Activate/deactivate distributed station

D_ACT_DP deactivates and activates stations of the distributed I/O and allows scanning of the deactivated or activated status. A distributed station can be a DP slave or an IO device.

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 calls of the asynchronous error organization blocks OB 85 (program execution error if the user data of the deactivated station is present in an automatically updated process image), and OB 86 (station failure) are omitted. You must not address the station from the program once it has been deactivated, since otherwise an I/O access error with calling of OB 122 will occur 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. The asynchronous error OBs 85 and 86 are not started when activating. If the BUSY parameter has signal state “0” following activation, the station can be addressed from the user program.

During a startup, the operating system of a CPU 300 automatically activates the deactivated station and waits until all stations have been activated.

DPRD_DAT Read user data consistently

DPRD_DAT reads consistent user data with a length of 3 bytes or greater than 4 bytes from a DP standard slave or IO device.

The LADDR parameter receives the module start address of the user data (input area). The RECORD parameter describes the area in which the read data is saved.

Tags with data types ARRAY and STRUCT, a PLC data type, or an ANY pointer with data type BYTE (e.g. P#DBzDBXy.x BYTE nnn) are permissible as actual parameters.

Note: If peripheral inputs (I:P) are addressed whose addresses are in the process image input (I), the process image is not updated.

DPWR_DAT Write user data consistently

DPWR_DAT writes consistent user data with a length of 3 bytes or greater than 4 bytes to a DP standard slave or IO device.

The LADDR parameter receives the module start address of the user data (output area). The RECORD parameter describes the area from which the transferred data is read. Tags with data types ARRAY and STRUCT, a PLC data type, or an ANY pointer with data type BYTE (e.g. P#DBzDBXy.x BYTE nnn) are permissible as actual parameters.

If peripheral outputs (Q:P) are addressed whose addresses are in the process image output (Q), the process image is not updated.

16.6.3 System blocks for PROFINET IO

The following system blocks can be used with PROFINET IO:

- RCVREC Receive data record from IO controller (SFB 73)
- PRVREC Provide data record for IO controller (SFB 74)
- IP_CONF Configure PROFINET interface (SFC 104)

Fig. 16.27 shows the graphic representation of the system block calls for PROFINET IO.

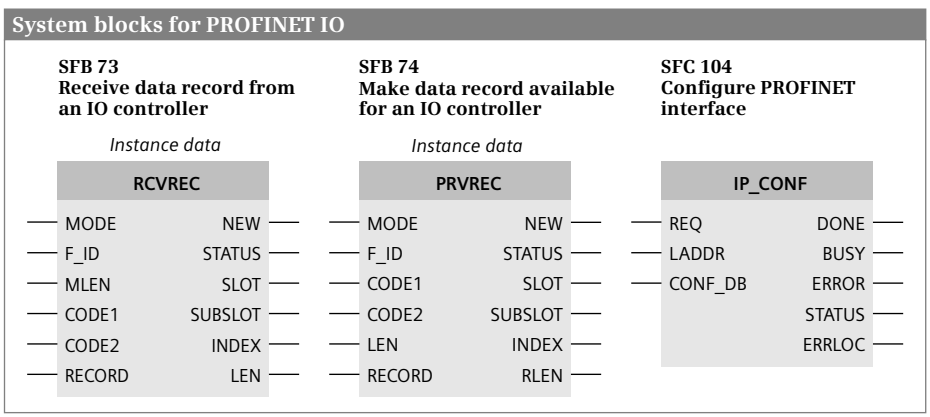


Fig. 16.27 Graphic representation of system blocks for PROFINET IO

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.

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.

IP_CONF Configure PROFINET interface

IP_CONF configures the integral PROFINET interface of the CPU. A prerequisite is that the *Obtain IP address in different manner* option was set during parameterization of the PROFINET interface with the hardware configuration when assigning the IP parameters.

IP_CONF works asynchronously, i.e. processing of a job can extend over several program cycles. The 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". 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 diagnostic address of the PROFINET interface at the LADDR parameter. The CONF_DB parameter is a pointer to the configuration data. This can be a data area addressed by an ANY pointer or a complete data block.

16.7 Actuator/sensor interface

16.7.1 Components of actuator/sensor interface

The actuator/sensor interface (AS-i) is an industrial fieldbus system for the lowest process level in automation plants in accordance with the open international standard EN 50295. An AS-i master controls up to 62 AS-i slaves via a 2-wire AS-i cable that transfers both the control signals and the supply voltage (Fig. 16.28).

An AS-i master can be configured in a PLC station or as a router. Configuration of an AS-i bus system is possible with the CP 343-2P communication module and the AS-i slaves included in the hardware catalog.

AS-i master

The AS-i master controls up to 62 AS-i slaves over the AS-i bus and provides the connection to higher-level controllers. Examples of modules with AS-i master are the CP 343-2 and 343-2P S7-300 communication modules, the DP/AS-i Link 20E and DP/AS-i Link Advanced modules for linking to PROFIBUS DP, and the IE/AS-i Link PN IO module for linking to PROFINET IO.

The **AS-i master CP 343-2** is a module in S7-300 design. It supports all AS-i master functions in accordance with the enhanced AS-i specification V3.0, such as double

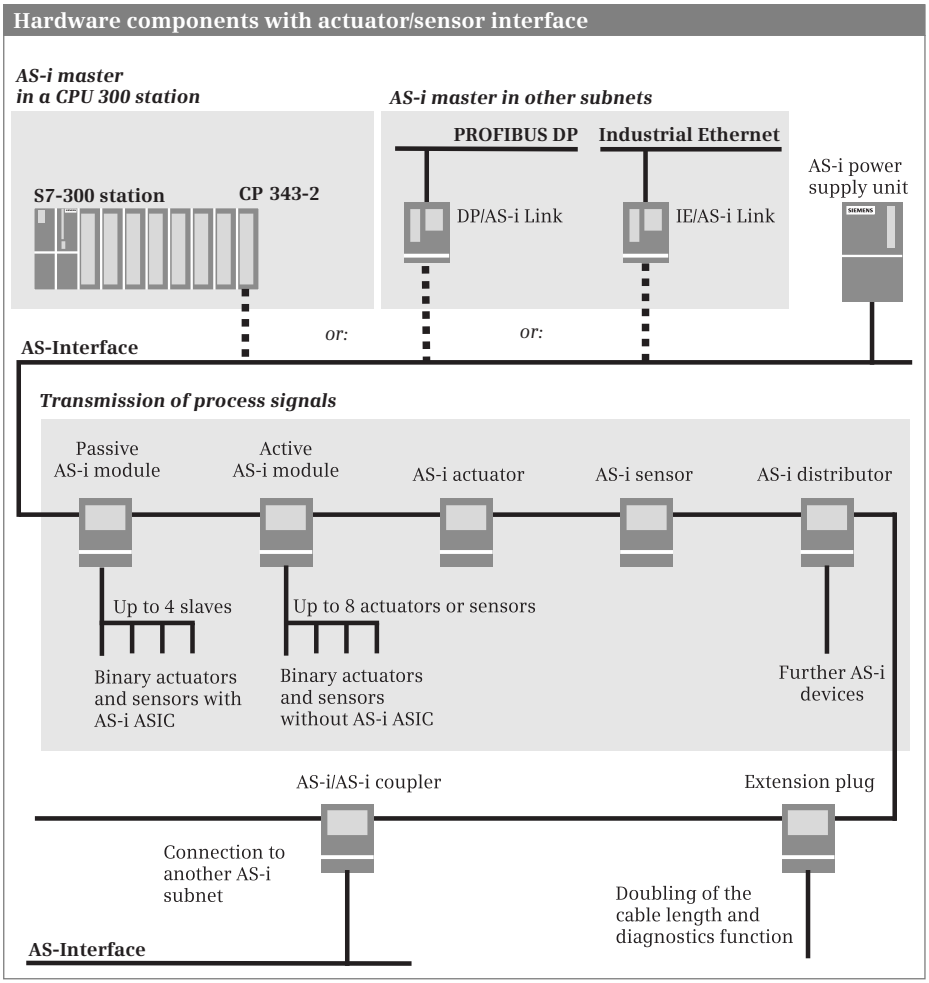


Fig. 16.28 Connecting the AS-i bus system to SIMATIC S7

address assignment (A/B slaves). It can thus control up to 62 digital slaves or up to 62 analog slaves.

The connection to the CPU is made via a user data interface with 16 bytes digital inputs and 16 bytes digital outputs (for the standard and A slaves). The data of the B slaves can be accessed by the system blocks RDREC *Read data record* and WRREC *Write data record*. The standard block AS-3422 is available for controlling the master interface.

The CP 343-2P communication module can be used to configure an AS-i bus system with AS-i slaves.

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.

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.

The **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.

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.

AS-i slaves

Actuators and sensors are connected to the AS-i bus using the AS-i slaves. Examples of units available as AS-i slaves:

- Compact modules with degree of protection IP 65/67 with up to 8 DI and 4 DO
- Analog modules with degree of protection IP 65/67 with 1, 2, or 4 AI/AO channels
- SlimLine and flat modules with degree of protection IP 20 with up to 16 DI
- Counter modules with degree of protection IP 20 for transmission of count values in the range from 0 to 15 for pulses up to 769 Hz
- Motor starters with degree of protection IP 65/67 for DC or AC motors
- LOGO! communication module CM AS-i for connection of the LOGO! logic module to the AS-i bus via 4 DI and 4 DO
- AS-i slaves integrated in operator panels, signaling columns, and contactors

In the standard version, up to 31 AS-i slaves can be connected to an AS-i bus, or up to 62 as A/B slaves.

Coupling modules

An **extension plug** can be used to double the length of the AS-i cable to 200 m. The extension plug contains an integral AS-i slave for diagnostics.

The **AS-i/AS-i data coupler** connects two AS-i bus systems bidirectionally via four virtual inputs and four virtual outputs. The data coupler is a standard slave in each of the two AS-i subnets. The two subnets are galvanically isolated.

16.7.2 Addresses on the actuator/sensor interface

Station addresses on the AS-i subnet

An AS-i subnet comprises an AS-i master and several AS-i slaves. The AS-i master does not have a station address. The station addresses of the AS-i slaves are assigned without gaps starting from 1 up to the maximum address (up to max. 62 depending on the properties of the AS-i slaves).

User data addresses on the AS-i subnet

The user data interface between CPU and AS-i master allows – depending on the design and operating mode – 16, 32, or 64 bytes in the input and output areas. The two operand areas start at the same address. Each AS-i slave occupies 4 bits in the address area. Fig. 16.29 shows the arrangement of the slave data in the user data interface.

User data addresses in the AS-i bus system

Arrangement of the slave data

Each AS-i slave occupies 4 bits of user data. One byte therefore contains the data for two AS-I slaves. During the configuration, STEP 7 stores the slave data alternately on the right and left in the bytes of the user data interface.

If only standard slaves are used, the user data interface is 16 bytes long, when using A/B slaves it is 32 bytes long in each case in the input and output areas.

Standard and A/B slaves can be used mixed together.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
n	= reserved =				Slave 1 / slave 1A			
n + 1	Slave 2 / slave 2A				Slave 3 / slave 3A			
n + ...	...				...			
n + 15	Slave 30 / slave 30A				Slave 31 / slave 31A			
n + 16	= reserved =				Slave 1B			
n + 17	Slave 2B				Slave 3B			
n + ...	...				...			
n + 31	Slave 30B				Slave 31B			

n = start address of the user data interface

Fig. 16.29 Arrangement of slave data in the user data interface

16.7.3 Configuring the actuator/sensor interface with CP 343-2P

General procedure

A prerequisite for configuration of the actuator/sensor interface is a created project with an S7-300 station. To configure the interface, start the hardware configuration in the Device view.

- ▷ The starting point for configuration is the AS-i master CP 343-2P.
- ▷ Select the AS-i interface in the Network view and connect it to an AS-i subnet.
- ▷ Then drag the desired AS-i slaves from the hardware catalog to the subnet and parameterize their properties.

The result is networking of the AS-i master with the AS-i slaves (Fig. 16.30).

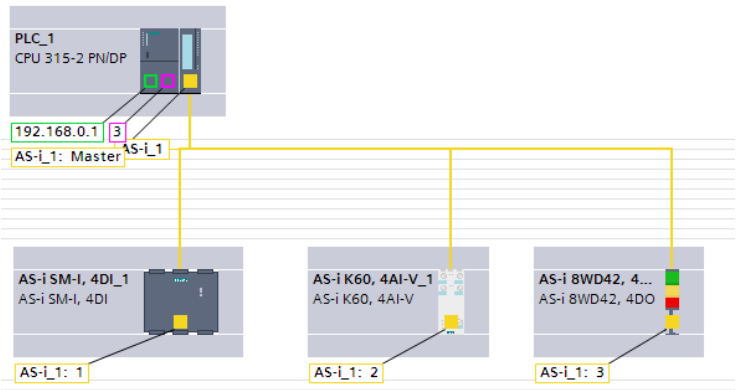


Fig. 16.30 Example of representation of an AS-i bus system

Configuration of AS-i master

You have created an S7-300 station in a project. The hardware configuration has been started and opened in the Network view. To add the CP 343-2P communication module, drag it with the mouse from the hardware catalog under *Controllers* > *SIMATIC S7-300* > *Communication modules* > *AS-Interface* > *CP 343-2P* > ... to the S7-300 station in the working window. Select the AS-i interface shown in yellow and then select the *Add subnet* command from the shortcut menu.

Set the start address of the user data interface in the properties of the CP 343-2P under *I/O addresses* in the *Input addresses* box. The output addresses start at the same address.

Configuration of AS-i slave

Drag the desired AS-i slave with the mouse from the hardware catalog under *Field devices* > *AS-Interface slaves* > ... to the AS-i subnet. STEP 7 assigns the station number and the position of the slave data in the user data interface in the sequence of configuration.

You can change the station number under *AS-Interface* in the properties of the AS-i slave. The position of the slave data in the user data interface changes accordingly.

16.7.4 System functions for AS-i

ASi_3422 Control AS-i master response (FC 7)

ASi_3422 controls a CP 343-2 or CP 343-2P communication module by means of command jobs (Fig. 16.31). You call the standard block once in the start-up routine using STARTUP = “1”. You start a job in the main program using ACT = “1”. The DONE parameter signals with signal state “1” that the job has been completed without errors. If the job is terminated with an error, ERROR = “1” and STATUS contains the error information.

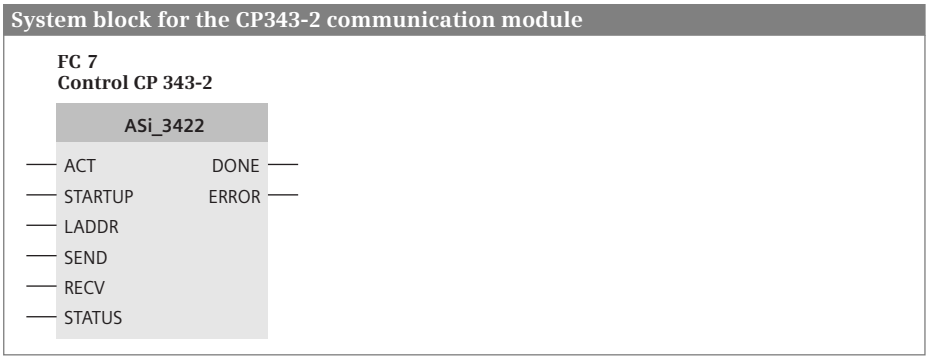


Fig. 16.31 Standard block for controlling the CP 343-2 communication module

The LADDR parameter contains the configured start address of the user data interface. The command sent to the CP module is taken from the data area specified by the SEND parameter and the reply of the CP module is saved in the receive area specified by RECV.

The program elements catalog contains ASi_3422 under *Extended instructions* > *Distributed I/O* > *Others*.

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 300 can exchange data with other stations over all subnets common to SIMATIC S7:

- With MPI (Multi Point Interface) by means of station-external S7 basic communication and S7 communication
- With PROFIBUS by means of station-internal S7 basic communication and S7 communication
- With Industrial Ethernet by means of station-internal S7 basic communication, S7 communication, and open user communication

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 CP 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 300 has a limited number of “connection resources”. The maximum number of connections which can be configured simultaneously is 6 for a CPU 312, 12 for a CPU 314, 16 for a CPU 315, and 32 for a CPU 317 and CPU 319.

Data exchange with the distributed I/O (PROFIBUS DP, PROFINET IO, and AS-i) is described in Chapter 16 “Distributed I/O” on page 607.

A prerequisite for configuration of communication is a created project with the PLC stations involved in the communication. Chapter 3 “Device configuration” on page 65 describes how to create a project and configure the PLC stations.

17.2 S7 basic communication

S7 basic communication is possible as station-internal S7 basic communication within an automation system and as station-external S7 basic communication between two automation systems.

The communication functions required for S7 basic communication are present in the system blocks integrated in the operating system. Connection configuration is not required for S7 basic communication.

17.2.1 Basics of station-internal S7 basic communication

Using station-internal S7 basic communication, you can exchange data within an automation system between central and distributed stations, for example between the DP master and an intelligent DP slave or between the IO controller and an intelligent IO device (Fig. 17.1).

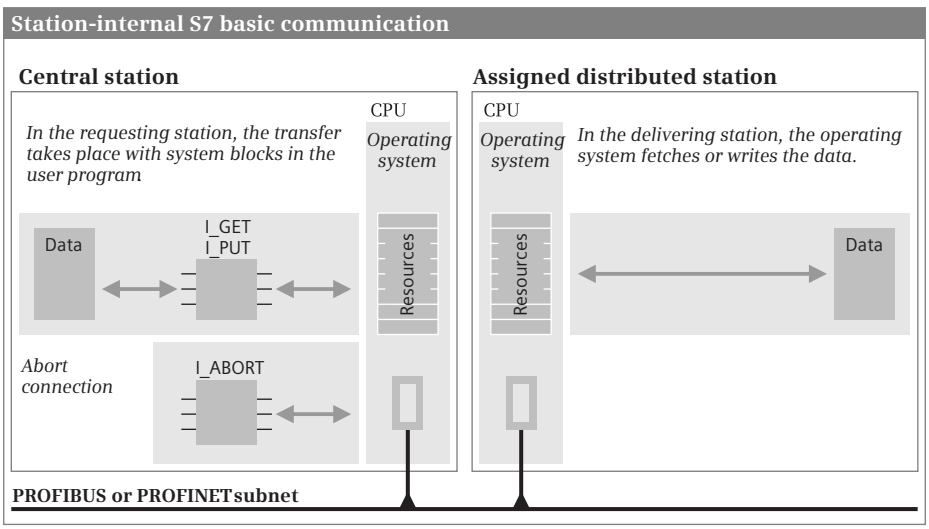


Fig. 17.1 Principle of station-internal S7 basic communication

Addressing of stations, connections

Identification of stations is derived from the I/O address: You specify the module start address of a transfer area at the LADDR parameter and whether this address is in the input or output area at the IOID parameter.

The communication functions establish the required connections in dynamic mode and – parameterizable – clear them again at the end of the job. If a connection cannot be established because resources are missing either in the sender or receiver, “Temporary shortage of resources” is signaled. Triggering of the transmission

must then be repeated. There can only be one connection in each direction between two communication partners.

By changing the block parameters during runtime, you can use a communication function for different connections. A communication function may not interrupt itself. You can only modify (edit) a program section in which a communication function is used when at STOP; a restart must be subsequently carried out.

User data, data consistency

The communication functions transfer a maximum of 76 bytes as user data. The CPU's operating system combines the user data into blocks independent of the transfer direction and these blocks are data-consistent. The length of the data transmitted consistently depends on the block size of the “passive” CPU.

17.2.2 Configuring of station-internal S7 basic communication

A prerequisite for station-internal S7 basic communication is a PROFIBUS or PROFINET subnet. Chapter 3.4.6 “Configuring a PROFIBUS subnet” on page 83 and Chapter 3.4.7 “Configuring a PROFINET subnet” on page 85 describe how to create such a subnet.

Particularly for station-internal S7 basic communication, configuration is unnecessary since the data transfer is handled via dynamic connections. Data transfer is triggered in the user program by system blocks.

17.2.3 System blocks for station-internal S7 basic communication

The following system blocks handle data transfer for station-internal S7 basic communication:

- I_GET Read data (SFC 72)
- I_PUT Write data (SFC 73)
- I_ABORT Abort connection (SFC 74)

The program elements catalog contains the system blocks under *Communication* > *Communication with iSlave/iDevice*. The graphic representation of the block calls is shown in Fig. 17.2.

I_GET Read data

I_GET reads data from a distributed station. A job is triggered by REQ = “1” and BUSY = “0” (“Initial call”). BUSY is set to “1” during execution of the job; changes to the REQ parameter now no longer have an effect. BUSY is returned to “0” at the end of the job. If REQ is still “1”, the job is started again immediately.

Following triggering of the read operation, the operating system in the partner device assembles the data requested by VAR_ADDR and sends it. If an SFC is called, the received data is entered completely into the destination area. RET_VAL then indicates the number of transferred bytes.

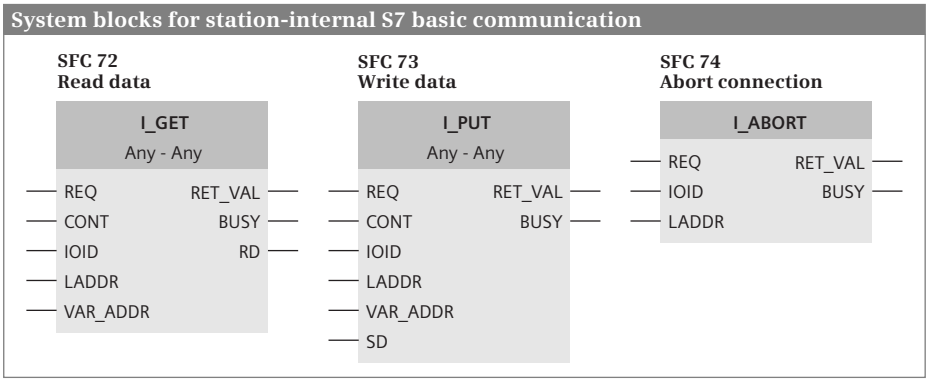


Fig. 17.2 Graphic representation of the SFCs for station-external S7 basic communication

If CONT = "0", the communication connection is cleared again. CONT = "1" retains the connection. The data is even read if the communication partner is at STOP.

The RD and VAR_ADDR parameters describe the area from which the data to be sent is read or into which the received data is to be written. Operands, tags, or any data areas addressed by an ANY pointer are permissible as actual parameters. A data type test between the send and receive data is not carried out.

The partner module is defined with the IOID and LADDR parameters.

I_PUT Write data

I_PUT writes data to a distributed station. A job is triggered by REQ = "1" and BUSY = "0" ("Initial call"). BUSY is set to "1" during execution of the job; changes to the REQ parameter now no longer have an effect. BUSY is returned to "0" at the end of the job. If REQ is still "1", the job is started again immediately.

Following triggering of the write operation, the operating system accepts all data from the source area into an internal buffer during the initial call and sends it to the partner device. The received data is written there by its operating system into the data area VAR_ADDR. BUSY is subsequently set to "0". The data is even written if the communication partner is at STOP.

The SD and VAR_ADDR parameters describe the area from which the data to be sent is read or into which the received data is to be written. Operands, tags, or any data areas addressed by an ANY pointer are permissible as actual parameters. A data type test between the send and receive data is not carried out.

The partner module is defined with the IOID and LADDR parameters.

I_ABORT Abort connection

I_ABORT aborts an existing connection. REQ = "1" triggers the job. I_ABORT can only be used to abort connections established using I_GET or I_PUT in the own station.

BUSY is set to “1” during execution of the job; changes to the REQ parameter now no longer have an effect. BUSY is returned to “0” at the end of the job. If REQ is still “1”, the job is started again immediately.

17.2.4 Basics of station-external S7 basic communication

Using station-external S7 basic communication, you can exchange data between PLC stations on the same MPI subnet depending on events (“MPI communication”). The communication functions required for this are system blocks in the CPU's operating system. The communication functions establish the connections themselves as required. Therefore the station-external connections are not configured in the connection table (Fig. 17.3).

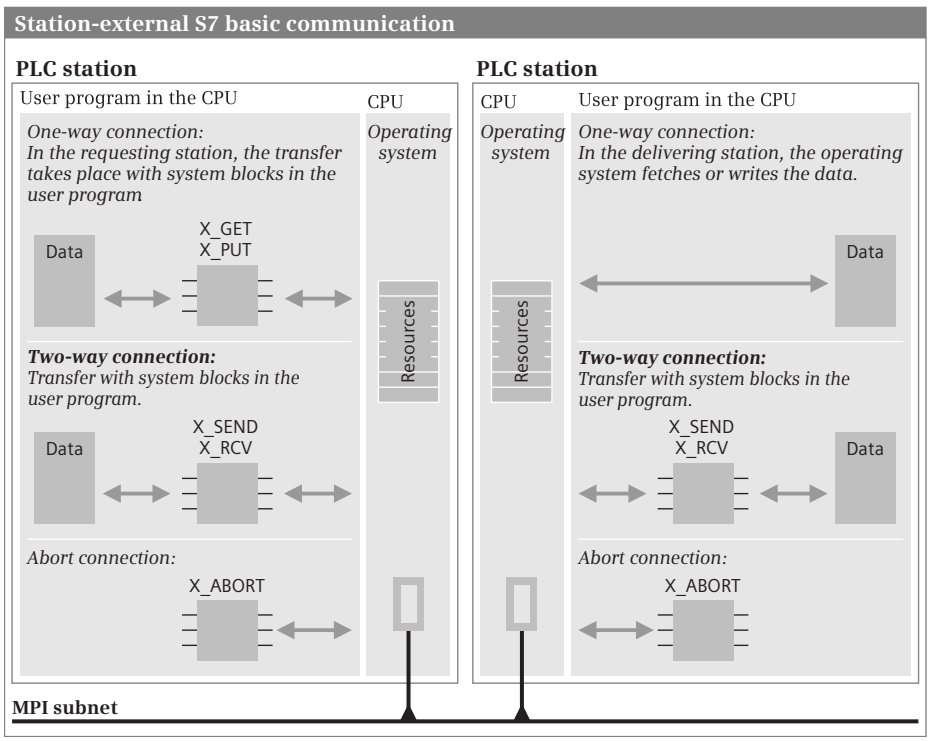


Fig. 17.3 Principle of station-external S7 basic communication

Addressing of stations, connections

Stations are derived from the MPI address at the DEST_ID parameter.

The communication functions establish the required connections in dynamic mode and – parameterizable – clear them again at the end of the job. If a connection cannot be established because resources are missing either in the sender or receiver,

“Temporary shortage of resources” is signaled. Triggering of the transmission must then be repeated. There can only be one connection in each direction between two communication partners.

All actively established connections are canceled upon a transition from RUN to STOP.

By changing the block parameters during runtime, you can use a communication function for different connections. A communication function may not interrupt itself. You can only modify (edit) a program section in which a communication function is used when at STOP; a restart must be subsequently carried out.

User data, data consistency

These communication functions transfer a maximum of 76 bytes as user data. The CPU's operating system combines the user data into blocks independent of the transfer direction and these blocks are data-consistent. The length of the data transmitted consistently is a CPU-specific variable.

If two CPUs exchange data by means of X_GET or X_PUT, the block size of the “passive” CPU is decisive for the data consistency of the transferred data. With a SEND/RCV connection, all data is transferred consistently.

17.2.5 Configuring of station-external S7 basic communication

A prerequisite for station-external S7 basic communication is an MPI subnet. Chapter 3.4.5 “Configuring an MPI subnet” on page 82 describes how to create such a subnet.

Particularly for station-external S7 basic communication, configuration is unnecessary since the data transfer is handled via dynamic connections. Data transfer is triggered in the user program by system blocks.

17.2.6 System blocks for station-external S7 basic communication

The following system blocks handle data transfer between different PLC stations on the MPI subnet:

- X_SEND Send data (SFC 65)
- X_RCV Receive data (SFC 66)
- X_GET Read data (SFC 67)
- X_PUT Write data (SFC 68)
- X_ABORT Abort connection (SFC 69)

The system blocks can be found in the program elements catalog under *Communication > MPI communication*. The graphic representation of the calls is shown in Fig. 17.4.

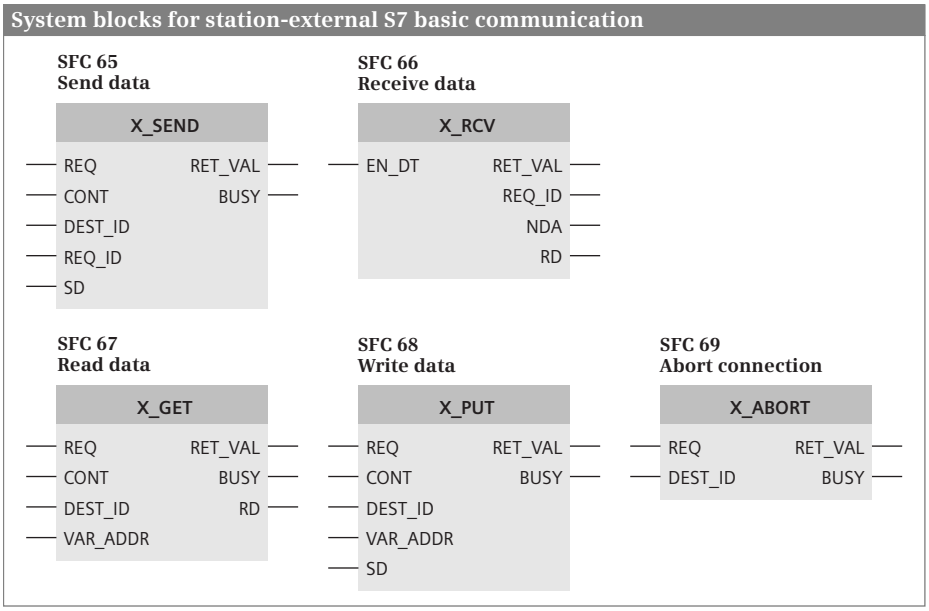


Fig. 17.4 Graphic representation of the SFCs for station-internal S7 basic communication

X_SEND Send data

X_SEND sends data which is received in the partner station by X_RCV. The partner device is specified with the MPI address at the DEST_ID parameter. A job is triggered by REQ = "1" and BUSY = "0" ("Initial call"). BUSY is set to "1" during execution of the job; changes to the REQ parameter now no longer have an effect. BUSY is returned to "0" at the end of the job. If REQ is still "1", the job is started again immediately.

The operating system reads all data from the source area into an internal buffer during the initial call and transfers it to the partner device.

BUSY is "1" for the duration of the send procedure. BUSY if set to "0" when the partner signals that the data has been fetched and the send job is finished.

With CONT = "0", the connection is canceled again and the corresponding CPU resources are then available for other communication connections. With CONT = "1", the connection is retained. By means of the REQ_ID parameter you can send an ID together with the send data which you can evaluate at X_RCV in the partner station.

The SD parameter describes the area from which the data to be sent is read. Operands, tags, or any data areas addressed by an ANY pointer are permissible as actual parameters. A data type test between the send and receive data is not carried out.

X_RCV Receive data

X_RCV receives data which was sent by the partner station by X_SEND. The received data is saved in an internal buffer. Several send operations can be saved in a queue in the chronological order of arrival.

You use EN_DT = “0” to check that data has been received; NDA is then “1”, RET_VAL shows the number of bytes of received data, and REQ_ID shows the same assignment as the corresponding parameter of X_SEND. With EN_DT = “1”, the system function transfers the initially entered (oldest) sent data completely into the destination area; NDA is then “1” and RET_VAL shows the number of transmitted bytes. If no data is present in the internal queue when EN_DT = “1”, NDA is then “0”.

Upon a restart, all data waiting for sending in the queue is rejected.

Upon cancellation of a connection, the oldest entry in the queue is retained if it has already been “scanned” with EN_DT = “0”, otherwise it is rejected like all other entries in the queue.

The RD parameter describes the area into which the received data is written. Operands, tags, or any data areas addressed by an ANY pointer are permissible as actual parameters.

A data type test between the send and receive data is not carried out. If the received data is irrelevant, an “empty” ANY pointer (NIL pointer) is permissible at the RD parameter.

X_GET Read data

X_GET reads data from the partner station. A job is triggered by REQ = “1” and BUSY = “0” (“Initial call”). BUSY is set to “1” during execution of the job; changes to the REQ parameter now no longer have an effect.

BUSY is returned to “0” at the end of the job. If REQ is still “1”, the job is started again immediately.

Following triggering of the read operation, the operating system in the partner device assembles the data requested by VAR_ADDR and sends it. When the communication function is called, the received data is entered completely into the destination area specified at the RD parameter. RET_VAL then indicates the number of transferred bytes.

If CONT = “0”, the communication connection is cleared again. CONT = “1” retains the connection. The data is even read if the communication partner is at STOP.

The RD and VAR_ADDR parameters describe the area from which the data to be sent is read or into which the received data is to be written. Operands, tags, or any data areas addressed by an ANY pointer are permissible as actual parameters. A data type test between the send and receive data is not carried out.

X_PUT Write data

X_PUT writes data to a partner station. A job is triggered by REQ = “1” and BUSY = “0” (“Initial call”). BUSY is set to “1” during execution of the job; changes to the REQ parameter now no longer have an effect.

BUSY is returned to “0” at the end of the job. If REQ is still “1”, the job is started again immediately.

Following triggering of the write operation, the operating system accepts all data from the source area specified at the SD parameter into an internal buffer during the initial call and sends it to the partner device. The received data is written there by its operating system into the data area specified at the VAR_ADDR parameter. BUSY is subsequently set to “0”.

The data is even written if the communication partner is at STOP.

The SD and VAR_ADDR parameters describe the area from which the data to be sent is read or into which the received data is to be written. Operands, tags, or any data areas addressed by an ANY pointer are permissible as actual parameters. A data type test between the send and receive data is not carried out.

X_ABORT Abort connection

X_ABORT aborts an existing connection. REQ = “1” starts the job. X_ABORT can only be used to abort connections established using X_SEND, X_GET, or X_PUT in the own station.

17.3 S7 communication

17.3.1 Basics

Using S7 communication you can transfer larger data quantities between PLC stations. The stations are connected to one another over an Ethernet subnet. The communication connections are static; they are configured in the connection table (Fig. 17.5).

S7 communication can be configured with a one-way or two-way connection.

The communication functions are standard function blocks for a CPU 300 which you copy from the program elements catalog into the user program. They can be called as a single instance with own data block or – in a function block – as a local instance. The standard function blocks used in the program and the instance data blocks derived from them are saved and displayed in the project tree under *Program blocks* > *System blocks* > *Program resources*.

17.3.2 Configuring S7 communication

A prerequisite for S7 communication is a PROFINET subnet. Chapter 3.4.7 “Configuring a PROFINET subnet” on page 85 describes how to create such a subnet.

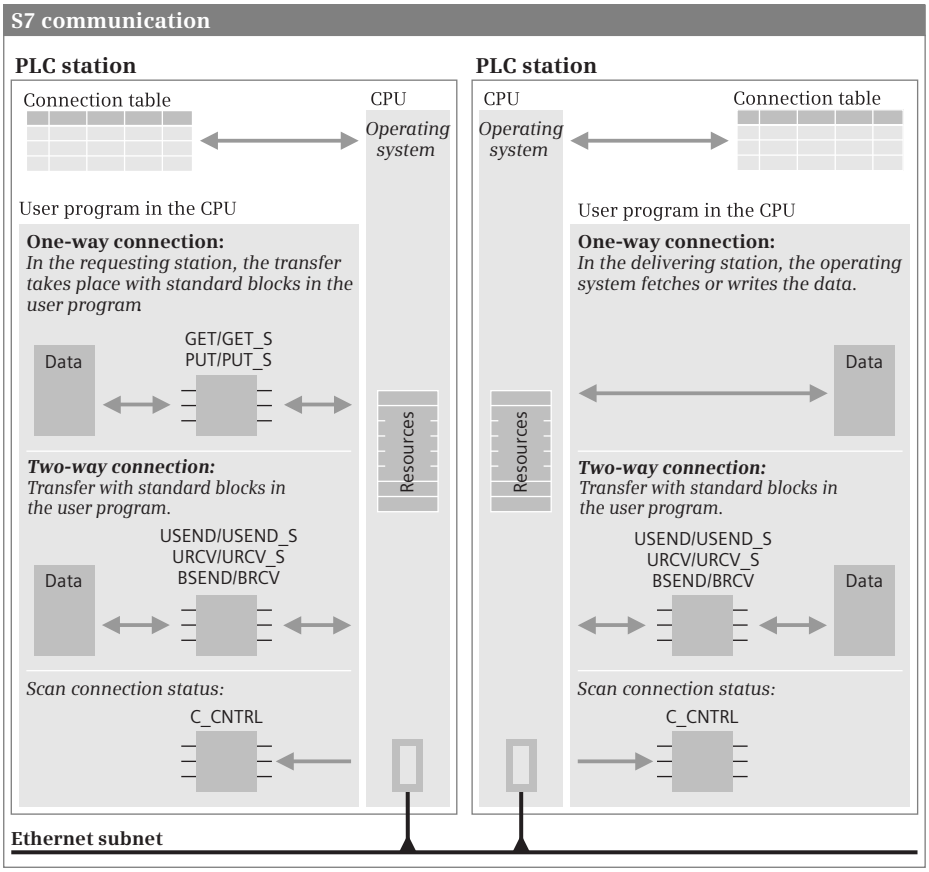


Fig. 17.5 Principle of S7 communication

Configure connection

You configure a connection between two PLC stations in the Network view of the hardware configuration. Click on the *Connections* button in the toolbar of the working window and set the connection type *S7 connection* in the adjacent drop-down list. Then select a bus interface of one PLC station and drag it with the right mouse button to a same type of bus interface in the partner station. The two interfaces are networked with the corresponding subnet and an S7 connection is created.

The connection table is present in the bottom part of the working window. The S7 connection is listed in the *Connections* tab. You can set the columns to be displayed: Right-click in a column title and then select the *Show/hide columns* command from the shortcut menu.

A communication connection is specified by a connection ID for every communication partner. STEP 7 assigns the connection ID when compiling the connection table. You use the “Local ID” for parameterization of the communication functions in

the module from which the connection is considered and the “Partner ID” for parameterization of the communication functions in the partner module.

It is possible to use the same logical connection for different send/receive jobs. To distinguish them, you must specify a job ID in addition to the connection ID in order to define the association between the send and receive blocks.

You can change the standard name assigned by STEP 7 to a connection: In the connection table, double-click in the *Local connection name* column on the cell with the name and enter a different name.

If you select a connection in the connection table, the inspector window shows the properties of this connection in the *Properties* tab. Among other options, you can use *Special connection properties* to set which communication partner is to initiate the active connection establishment.

Initialization

S7 communication must be initialized in the startup so that the connection can be established to the communication partner. Initialization takes place in the CPU for which the *Active connection establishment* attribute is activated. To do this, you call up the communication blocks used in cyclic mode in a startup OB and supply the parameters (if present) as follows:

- REQ = FALSE
- ID = Local connection ID from the connection table
  (data type WORD W#16#xxxx)
- R_ID = Job ID which you define for a “pair of blocks”
  (data type DWORD DW#16#xxxx xxxx)

The blocks must be repeatedly called up in a program loop until the DONE parameter has signal state “1”. The ERROR and STATUS parameters provide information on any errors and the job status.

You need not connect the data areas during the startup (concerns parameters ADDR_*n*, RD_*n* and SD_*n*).

Cyclic mode

In cyclic mode, you can call the communication blocks with their absolute name and control data transfer with the REQ and EN_R parameters. You must evaluate the results at the NDR, DONE, ERROR, and STATUS parameters immediately after each execution of a communication block since they only remain valid until the next call.

With S7-300, the parameters with data type ANY (SD_1, RD_1, ADDR_1) may only be supplied with bit memories and data as the operand areas.

17.3.3 One-way data exchange

In the case of one-way data exchange, the call of the communication block is only present in one CPU. In the partner CPU, the operating system handles the required communication functions.

The following blocks are available for one-way data exchange:

- GET_S (FB 14), GET (FB 34)
  Read data from a partner CPU
- PUT_S (FB 15), PUT (FB 35)
  Write data to a partner CPU

GET and PUT have four send and receive areas, but a CPU 300 can only use the first one in each case (ADDR_1, RD_1, and SD_1).

The standard blocks GET and PUT can be found in the program elements catalog under *Communication > S7 communication*, the standard blocks GET_S and PUT_S under *Communication > S7 communication > Others*. The graphic representation of these system blocks is shown in Fig. 17.6.

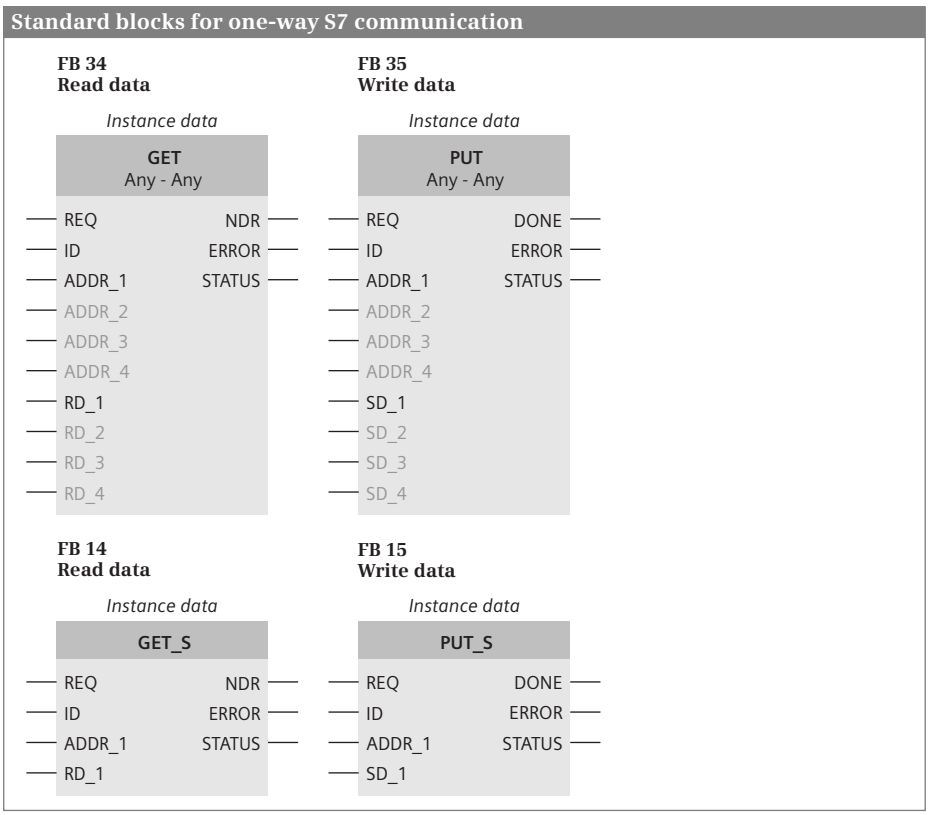


Fig. 17.6 Communication functions for one-way S7 communication

GET_S/GET Read data from a partner CPU
PUT_S/PUT Write data to a partner CPU

The data read using GET_S or GET is combined in the partner CPU by the operating system; the data written using PUT_S or 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 data blocks transmitted consistently depends on the (server) CPU used.

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.

By means of the ADDR_1 parameter, you specify the tag or area in the partner device from which you wish to fetch data or to which you wish to send data. The area at ADDR_1 must agree with the corresponding area at SD_1 or RD_1. You need not assign the other area parameters for a CPU 300 (not all parameters need be assigned on a function block).

17.3.4 Two-way data exchange

In the case of 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" which are then distinguished by the job ID.

The following blocks are available for two-way data exchange:

- USEND_S (FB 8), USEND (FB 28)
  Uncoordinated sending of a data packet with CPU-specific length
- URCV_S (FB 9), URCV (FB 29)
  Uncoordinated receipt of a data packet with CPU-specific length
- BSEND (FB 12)
  Send a data block with a length of up to 32 or 64 KB
- BRCV (FB 13)
  Receive a data block with a length of up to 32 or 64 KB

USEND and URCV have four send and receive areas, but a CPU 300 can only use the first one in each case (RD_1 and SD_1).

The standard blocks USEND, URCV, BSEND, and BRCV can be found in the hardware catalog under *Communication > S7 communication*, the standard blocks USEND_S and URCV_S under *Communication > S7 communication > Others*. The graphic representation of the block calls is shown in Fig. 17.7.

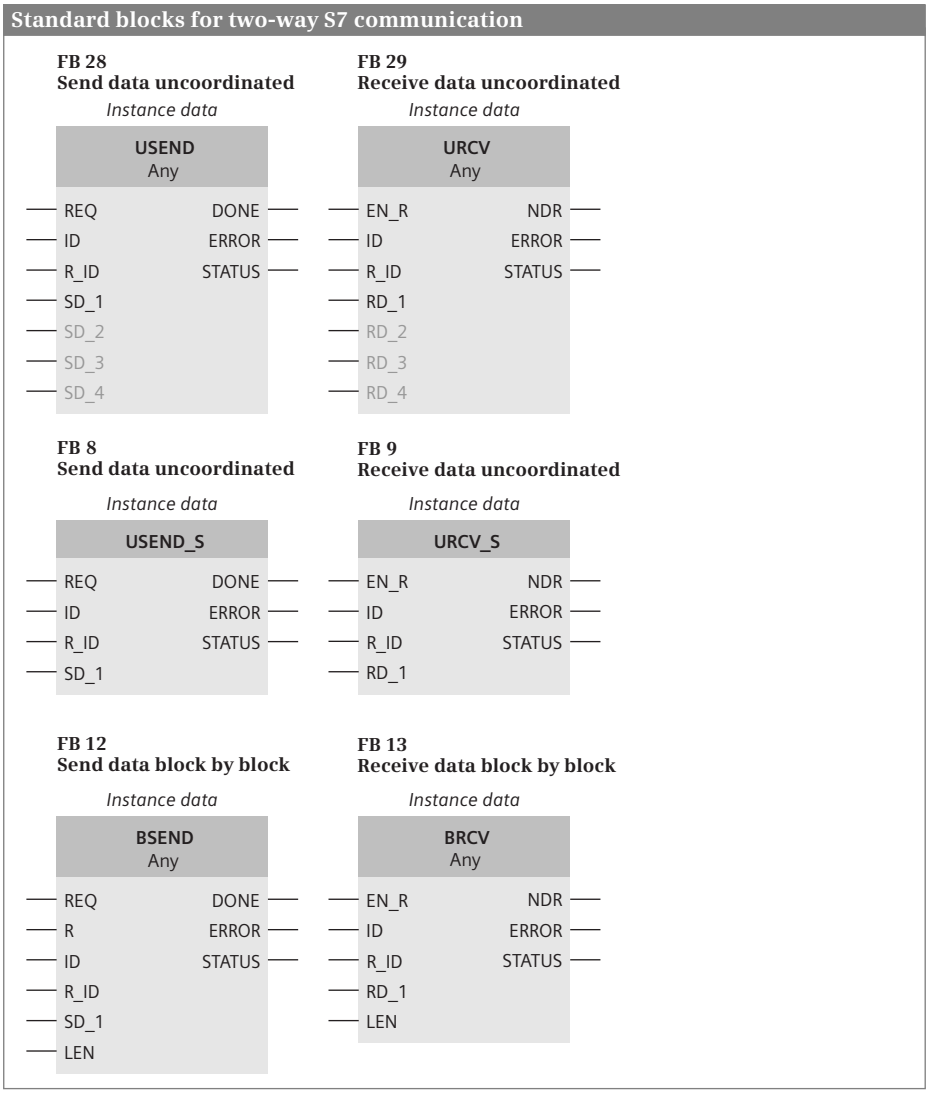


Fig. 17.7 Communication functions for two-way S7 communication

USEND_S/USEND Send data uncoordinated
URCV_S/URCV Receive data uncoordinated

You specify the tag or area you wish to transfer at the SD_1 and RD_1 parameters. The send area SD_1 must agree with the corresponding receive area RD_1. You need not assign the other area parameters for a CPU 300 (not all parameters need be assigned on a function block).

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”.

If the NDR parameter has assumed the value “1” following data transfer, call the block again, but with EN_R = “0” this time in order to prevent the receive area from being overwritten by new data during data evaluation.

You supply the ID parameter with the connection ID defined by STEP 7 in the connection table for both the local and partner devices (the two IDs can be different). 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 logical connection (specified by means of ID).

With a CPU 300, you can change the assignments of the ID and R_ID parameters following completion of each job.

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”).

BSEND Block-oriented sending
BRCV Block-oriented receiving

At the SD_1 or RD_1 parameter you specify a pointer to the first byte of the data area (when called for the first time, the length of this actual parameter determines the maximum size of the communication buffer, it is not evaluated with further calls); the number of bytes of the data to be currently sent or received is present at the LEN parameter.

The data volume transferred can be up to 32 KB or 64 KB for an S7-300 with integral interface; the transfer process is carried out in blocks asynchronous to execution of the user program. The LEN parameter is updated after every received block.

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”.

If the NDR parameter has assumed the value “1” following data transfer, call the block again, but with EN_R = “0” this time in order to prevent the receive area from being overwritten by new data during data evaluation.

You supply the ID parameter with the connection ID defined by STEP 7 in the connection table for both the local and partner devices (the two IDs can be different). 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 logical connection (specified by means of ID).

With a CPU 300, you can change the assignments of the ID and R_ID parameters following completion of each job.

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”).

17.3.5 Monitoring functions

The standard block

▷ C_CNTRL Scan status of a connection (FC 62)

is available for monitoring functions. The program elements catalog contains the system block under *Communication > S7 communication*. The graphic representation of the block call is shown in Fig. 17.8.

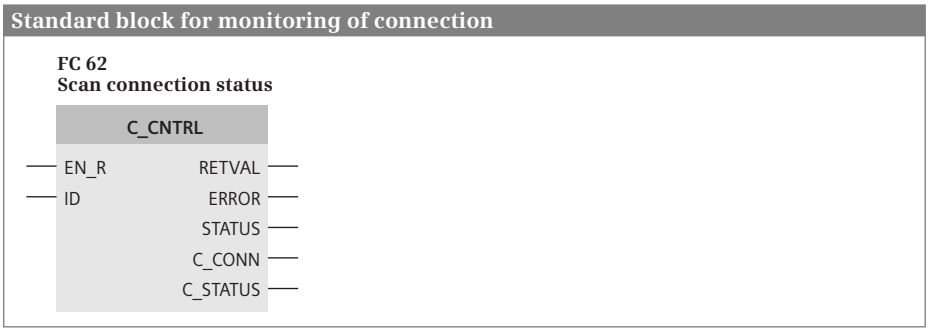


Fig. 17.8 Communication function for monitoring of connection

C_CNTRL Scan status of a connection

With a CPU 300, C_CNTRL determines the status of a connection in the local device. You supply the ID parameter with the connection ID defined by STEP 7 in the connection table for the local device.

The actual connection status is indicated by "1" in the EN_R parameter. You must evaluate the ERROR and STATUS parameters after every block call.

The C_CONN and C_STATUS parameters provide information on the actual connection status.

17.4 Open user communication

17.4.1 Basics

Open user communication ("Open communication via Industrial Ethernet") transfers data between two devices connected to the Ethernet subnet. Communication can be implemented using the TCP native protocol in accordance with RFC 793, the ISO-on-TCP protocol in accordance with RFC 1006, or the UDP protocol in accordance with RFC 768.

Configuring open user communication

The following are required before data can be transferred with open user communication:

- In the case of the TCP native and ISO-on-TCP protocols, a connection must be established to the communication partner (“connection-oriented protocols”).
- In the case of the UDP protocol, a connection must be established to the communication layer of the CPU operating system (“connectionless protocol”). The partner is then addressed when the relevant function block is called.

The connection is configured via a data area (not using the connection table). The necessary data structures are stored in the PLC data type TCON_PAR that the function blocks use for establishing and clearing the connection. The data contains the connection ID, which defines a specific connection and the associated function block calls, and the information on the protocol used.

Establishment of the connection to the partner or setting up of the communication access point is handled by the TCON communication function that you call in the main program of both partner devices. Data can be transferred in parallel in both directions over an established connection. Several connections can exist on one physical line. The TDISCON communication function cancels the connection again and thus releases the resources used (Fig. 17.9).

The TSEND and TRCV communication functions transfer data using the TCP native or ISO-on-TCP protocol. Data transfer with the UDP protocol requires the TUSEND and TURCV communication functions. When calling these communication functions, you specify the address of the partner device in a data area.

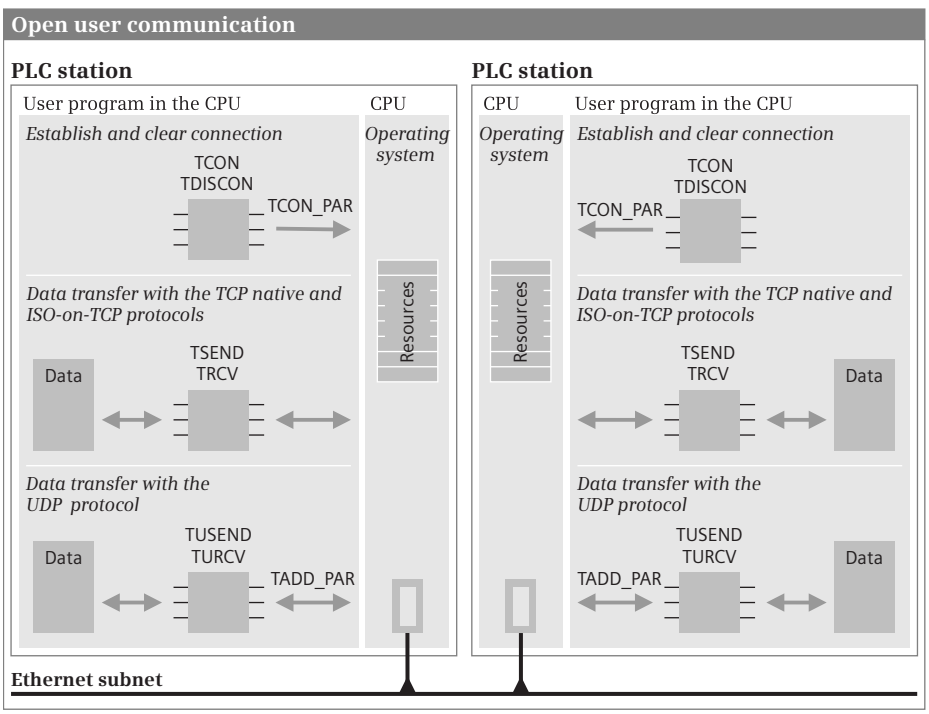


Fig. 17.9 Principle of open user communication

Calling the communication functions

The communication functions for open user communication work asynchronously, i.e. job execution can take several program cycles under certain circumstances. You can call the communication functions in the main program and control data transfer with the REQ and EN_R parameters. You must evaluate the results at the BUSY, NDR, DONE, ERROR, and STATUS parameters immediately after each execution since they only remain valid until the next call.

17.4.2 Establishing and clearing connections

Before data can be transferred with open user communication, a connection must be established to the partner device (in the case of TCP native and ISO-on-TCP) or to the communication layer of the operating system (in the case of UDP). You can use the following standard blocks for this purpose:

- TCON (FB 65)
  Establish connection to the communication partner or the communication layer of the operating system
- TDISCON (FB 66)
  Clear connection

The communication functions can be found in the program elements catalog under *Communication* > *Open user communication*. The graphic representation of the block calls is shown in Fig. 17.10.

TCON Establish connection

TCON creates the conditions for open user communication. The parameters required for this are located in a data area that has the structure of the data type TCON_PAR.

When using the TCP native and ISO-on-TCP protocols, a connection is established to the communication partner. The station for which “Active connection setup” is entered establishes the connection. The partner station must then be designated as “passive”. This designation is independent of the direction of transfer of the data.

The connection is monitored and maintained by the CPU's operating system. In the event of a break in connection, the active partner attempts to re-establish the connection without having to call TCON again. TDISCON clears the communication connection with the CPU at STOP or in the case of POWER OFF/ON.

If the UDP protocol is used, TCON sets up a local communication access point that represents the connection between the user program and the communication layer of the operating system. No connection is made to the connection partner.

Designate the communication connection by assigning the ID parameter. The specification must correspond to the variable *id* in the connection data. You specify the connection data with the pointer at the CONNECT parameter.

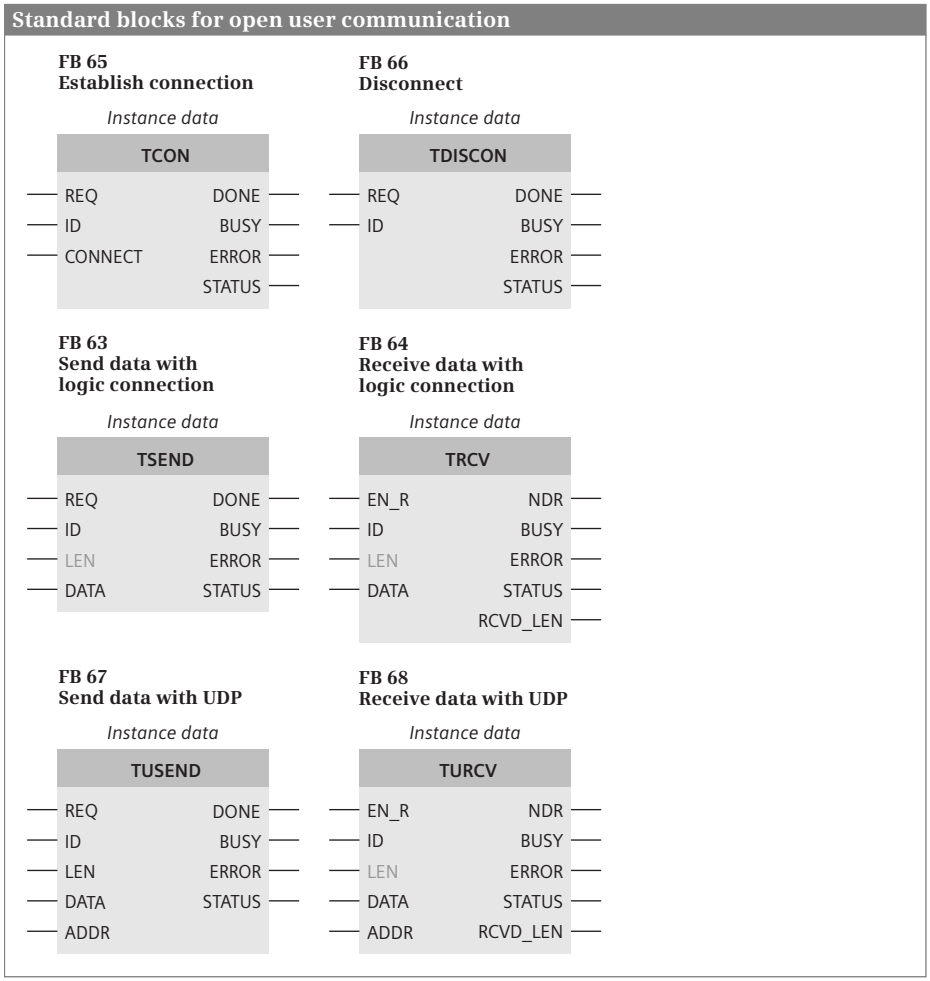


Fig. 17.10 Graphic representation of blocks for open user communication

In the initial state, the REQ, BUSY, DONE, and ERROR parameters have signal state “0”. A rising edge at the REQ parameter starts the connection establishment.

While the job is running, BUSY = “1”. The job has been successfully completed if BUSY = “0”, DONE = “1”, and ERROR = “0”. If the job contains errors, then BUSY = “0”, DONE = “0”, and ERROR = “1”. The error is then specified at the STATUS parameter. BUSY, DONE, and ERROR are reset to “0” if REQ is returned to “0”.

TDISCON Clear connection

TDISCON terminates the requirements for open user communication. The connection to the communication partner is cleared or the communication access point is closed.

Designate the communication connection by assigning the ID parameter. The specification must correspond to the variable *id* in the connection data.

In the initial state, the REQ, BUSY, DONE, and ERROR parameters have signal state “0”. A rising edge at the REQ parameter starts clearance of the connection.

While the job is running, BUSY = “1”. The job has been successfully completed if BUSY = “0”, DONE = “1”, and ERROR = “0”. If the job contains errors, then BUSY = “0”, DONE = “0”, and ERROR = “1”. The error is then specified at the STATUS parameter. BUSY, DONE, and ERROR are reset to “0” if REQ is returned to “0”.

TCON_PAR Structure of the connection data

The data type TCON_PAR (UDT 65) contains the structure of the connection data either for the communication connection to the partner device (TCP native and ISO-on-TCP protocols) or for the connection to the communication layer of the local operating system (UDP protocol).

You require a data block with this structure for each connection. For each connection, you can use your own global data block based on TCON_PAR, or you can combine the data blocks in a shared global data block (Table 17.1).

17.4.3 Data transfer with TCP native or ISO-on-TCP

The following standard blocks are available for data transfer with the TCP native and ISO-on-TCP connection-oriented protocols:

- TSEND Send data with logic connection (FB 63)
- TRCV Receive data with logic connection (FB 64)

The communication functions can be found in the program elements catalog under *Communication* > *Open user communication*. The graphic representation of the block calls is shown in Fig. 17.10.

You have to establish a connection to the partner station with TCON before transferring the data. Data can be exchanged simultaneously in both directions over the connection with TSEND and TRCV.

TSEND Send data with logic connection

TSEND sends data with the TCP native or ISO-on-TCP protocol via an existing communication connection.

Designate the communication connection by assigning the ID parameter. The specification must agree with the variable *id* in the connection data. Specify the send mailbox with the pointer at the DATA parameter.

In the initial state, the REQ, BUSY, DONE, and ERROR parameters have signal state “0”. Start data transfer with a rising edge at the REQ parameter. On the initial call with “1”, the data is fetched from the area specified by the DATA parameter. The number of bytes specified at the LEN parameter is sent.

Table 17.1 Structure of data type TCON_PAR

Byte	Parameter	Data type	Description
0 and 1	block_length	WORD	Length of TCON_PAR (fixed at 64 bytes)
2 and 3	id	WORD	Connection ID
4	connection_type	BYTE	Protocol variant B#16#11 : TCP B#16#12 : ISO-on-TCP B#16#13 : TCP (compatibility mode) With UDP : B#16#13
5	active_est	BOOL	Type of connection establishment FALSE: Passive connection establishment TRUE: Active connection establishment With UDP : FALSE
6	local_device_id	BYTE	Communication module ID
7	local_tsap_id_len	BYTE	Length of parameter local_tsap_id
8	rem_subnet_id_len	BYTE	B#16#00
9	rem_staddr_len	BYTE	Length of parameter rem_staddr With UDP : B#16#00
10	rem_tsap_id_len	BYTE	Length of parameter rem_tsap_id With UDP : B#16#00
11	next_staddr_len	BYTE	Length of parameter next_staddr With UDP : B#16#00
12 to 27	local_tsap_id	ARRAY [1..16] OF BYTE	Depending on connection: local port number or local TSAP ID With UDP : local port number
28 to 33	rem_subnet_id	ARRAY [1..6] OF BYTE	B#16#00
34 to 39	rem_staddr	ARRAY [1..6] OF BYTE	IP address of remote partner With UDP : B#16#00
40 to 55	rem_tsap_id	ARRAY [1..16] OF BYTE	Depending on connection: remote port number or remote TSAP ID With UDP : B#16#00
56 to 61	rem_staddr	ARRAY [1..6] OF BYTE	CP slot With UDP : B#16#00
62 and 63	spare	WORD	W#16#0000

While the job is running, BUSY = “1”. The job has been successfully completed if BUSY = “0”, DONE = “1”, and ERROR = “0”. If the job contains errors, then BUSY = “0”, DONE = “0”, and ERROR = “1”. The error is then specified at the STATUS parameter. BUSY, DONE, and ERROR are reset to “0” if REQ is returned to “0”.

The data in the send area can then be modified again when either DONE or ERROR has signal state “1”.

TRCV Receive data with logic connection

TRCV receives data with the TCP native or ISO-on-TCP protocol via an existing communication connection.

Designate the communication connection by assigning the ID parameter. The specification must agree with the variable *id* in the connection data. Specify the receive mailbox with the pointer at the DATA parameter.

If the LEN parameter has 0, the length specified in the DATA parameter is used. After a data block has been received, the number of bytes received is made available at the RCVD_LEN parameter and NDR is set to signal state “1”.

With the TCP native protocol, neither the length of the message frame nor its start or end is transferred. So that the number of bytes sent is correctly received, the LEN parameter at the receive block must be assigned the same value as the LEN parameter at the send block.

If a larger value has been selected for LEN at the receive block, part of the following message frame (from the next job) will also be received. NDR is only set to “1” if the parameterized length has been reached.

If a smaller value has been selected for LEN, NDR is set to “1” when the parameterized length is reached and the RCVD_LEN parameter is assigned the number of received bytes. With each subsequent call, another data block is received.

With the ISO-on-TCP protocol, information about the length and end of a message frame is sent. If LEN at the receive block is larger than at the send block, the sent data is received, NDR is set to “1” and the number of received bytes is written in RCVD_LEN. If LEN is smaller, an error message is issued: ERROR = “1”, STATUS = W#16#8088.

TRCV only receives data if the EN_R parameter has signal state “1”.

While the job is running, BUSY = “1”. The job has been successfully completed if BUSY = “0”, NDR = “1”, and ERROR = “0”. If the job contains errors, BUSY = “0”, NDR = “0”, and ERROR = “1”. The error is then specified at the STATUS parameter. BUSY, NDR, and ERROR are reset to “0” if EN_R is returned to “0”.

The data in the receive mailbox is consistent if NDR has signal state “1”.

17.4.4 Data transfer with UDP

The following standard blocks are available for data transfer with the connectionless UDP protocol:

- TUSEND Send data with UDP (FB 67)
- TURCV Receive data with UDP (FB 68)

The communication functions can be found in the program elements catalog under *Communication > Open user communication*. The graphic representation of the block calls is shown in Fig. 17.10.

You have to establish a connection to the communication layer of the operating system with TCON before transferring the data. The address of the communication partner is located in a data area that has the structure of the data type TADD_PAR.

TUSEND Send data with UDP

TUSEND sends data with the UDP protocol.

The assignment of the ID parameter designates the connection between the user program and the communication layer of the operating system. The value must agree with the *id* tag in the connection data. Specify the send mailbox with the pointer at the DATA parameter.

The information on the communication partner is located in a data area to which the pointer at the ADDR parameter points. With each new send job, the address and thus the partner can be changed without having to redefine the communication access point with TCON.

In the initial state, the REQ, BUSY, DONE, and ERROR parameters have signal state “0”. Start data transfer with a rising edge at the REQ parameter. On the initial call with “1”, the data is fetched from the area specified by the DATA parameter. The number of bytes specified at the LEN parameter is sent (1 to max. 1460).

While the job is running, BUSY = “1”. The job has been successfully completed if BUSY = “0”, DONE = “1”, and ERROR = “0”. If the job contains errors, then BUSY = “0”, DONE = “0”, and ERROR = “1”. The error is then specified at the STATUS parameter. BUSY, DONE, and ERROR are reset to “0” if REQ is returned to “0”.

The data in the send area can then be modified again when either DONE or ERROR has signal state “1”.

TURCV Receive data with UDP

TURCV receives data with the UDP protocol.

The assignment of the ID parameter designates the connection between the user program and the communication layer of the operating system. The value must agree with the *id* tag in the connection data. Specify the receive mailbox with the pointer at the DATA parameter.

The information on the communication partner is located in a data area to which the pointer at the ADDR parameter points.

The number of bytes to be received is set at the LEN parameter (1 to max. 1460). After a data block has been received, the number of bytes received is made available at the RCVD_LEN parameter and NDR is set to signal state “1”.

Data is only received if the EN_R parameter has signal state “1”.

While the job is running, BUSY = “1”. The job has been successfully completed if BUSY = “0”, NDR = “1”, and ERROR = “0”. If the job contains errors, BUSY = “0”, NDR = “0”, and ERROR = “1”. The error is then specified at the STATUS parameter. BUSY, NDR, and ERROR are reset to “0” if EN_R is returned to “0”.

The data in the receive area is consistent if NDR has signal state “1”.

UDP (User Data Protocol)

Establishment of a connection is not carried out with UDP. The communication partner is specified at the ADDR parameter of the send block (IP address and port number). The receive block then supplies the IP address and the port number of the sender at the ADDR parameter.

The data type TADD_PAR contains the structure of the address information. The pointer at ADDR points to a data area with this structure.

With UDP, information on the length and end of a message frame is transferred. If LEN at the receive block is larger, the sent data is copied to the receive mailbox, NDR is set to “1” and the number of received bytes is written in RCVD_LEN. If LEN is smaller, an error message is issued: ERROR = “1”, STATUS = W#16#8088.

TADDR_PAR Structure of the address information with UDP

The data type TADDR_PAR (UDT 66) contains the structure of the remote partner's address information when using the UDP protocol. With this data structure you configure a data area in a data block which contains the addresses of the receiver stations and parameterize this data area at the ADDR parameter of the send block TUSEND. At the receive block TURCV you parameterize a data area with this structure at the ADDR parameter which accommodates the addresses of the transmitting station (Table 17.2).

Table 17.2 Structure of data type TADDR_PAR

Byte	Parameter	Data type	Description
0 to 3	rem_ip_addr	ARRAY [1..4] OF BYTE	IP address of remote partner
4 and 5	rem_port_nr	ARRAY [1..2] OF BYTE	Port no. of remote partner
6 and 7	spare	ARRAY [1..2] OF BYTE	B#16#00

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. 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 STL or SCL in the TIA Portal can also be saved as text files.

18.1.1 General procedure

A source file can be generated in two different ways: You write the source file completely using a text editor, or you take a block as template and generate a source file by exporting the block. Following editing with the text editor, you import the external source file into the TIA Portal and generate the blocks contained in the source file by compiling. You can then edit these further using the program editor of STEP 7.

Generating a source file by exporting

In the project tree, select the block(s) from which you wish to generate a source file in the *Program blocks* folder and select the *Copy as text* command from the shortcut menu. The program editor writes the source text into the Windows clipboard.

Start the desired text editor and paste in the contents of the clipboard. Save an STL source file with the file extension .stl and an SCL source file with the file extension .scl. Files with these extensions can be imported as external source files into the TIA Portal.

Generating a source file with a text editor

In order to program a block, you must use keywords in a specific sequence in the source file. The program of each block consists of the block header with specification of the block type and properties. With code blocks, this is followed by the declaration of the interface and the actual program. With data blocks and PLC data types you subsequently specify the data operands or types.

If the source file contains blocks which are called in the source file or if data operands are accessed, you should observe a specific sequence in the source file. The blocks or data operands should be located before the position of use in the source file. You can also call blocks in the source file which are present in the *Program blocks* folder or use system blocks from the program elements catalog.

A source file can contain several blocks and these can be logic or data blocks as well as PLC data types. You export and import PLC tags separate from the source file (see Chapter 6.2.4 “Exporting and importing a PLC tag table” on page 222).

When working with source files, you must handle blocks generated using STL separate from those generated 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.

The following chapters describe how to program blocks in a source file.

Importing an external source file

To import an external source file, open the *External source files* folder in the project tree and double-click on *Add new external file*. In the dialog window, select the type of source (*STL sources* or *SCL sources*), navigate to the storage location, select the source file, and import it by clicking on the *Open* button.

The source file is saved in the *External source files* folder.

Editing an external source file in the TIA Portal

As preparation for editing an external source file in the TIA Portal, you must link the file extension .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. Under *Open with*, 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.

18.1.2 Programming a code block in the source file

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 title in the block properties Block comment in the block properties
	CODE_VERSION1 KNOW_HOW_PROTECT	Only with FB ("not capable of multi-instance"), only with STL Know-how protection (cannot be canceled)
	NAME : *Block name* FAMILY : *Block family* AUTHOR : *Created by* VERSION : *Version*	Block property: Block name Block property: Block family Block property: Created by Block property: Block version
Declaration	VAR_INPUT name : Data type := Default setting; *) END_VAR	Input parameter (not with OB)
	VAR_OUTPUT name : Data type := Default setting; *) END_VAR	Output parameter (not with OB)
	VAR_IN_OUT name : Data type := Default setting; *) END_VAR	In/out parameter (not with OB)
	VAR name : Data type := Default setting; *) END_VAR	Static local data (only with FB)
	VAR_TEMP name : Data type := Default setting; *) END_VAR	Temporary local data
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

*) Overlaying of data types with the keyword AT is additionally possible with SCL (see text)

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.

With symbolic addressing, an organization block should have the standard name. The correct block number is then assigned to it when importing from the external source. If the organization block has any name, it is assigned 0 (zero) as the number. The envisaged block number must then be manually assigned to this OB 0, for example using the *Properties* command in the shortcut menu of the organization block. The standard names of the organization blocks are listed in Table 5.4 "Organization blocks available with the standard controllers of S7-300" on page 187.

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.

The keyword KNOW_HOW_PROTECT protects the block from unwanted access. You can no longer cancel this protection, in contrast to block protection with password in the TIA Portal.

The keyword CODE_VERSION1 is permissible for function blocks with STL program. Following importing into STEP 7, the block attribute *Multiple instance capability* can be enabled or disabled for these function blocks.

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.

The declaration of a tag consists of the name, the data type, possibly a default setting, and an optional tag comment. Example:

```
Quantity : INT := +500; //Units per batch
```

Not all tags can have default values, e.g. default values are not possible for the temporary local data. Chapter 5.2.5 "Block interface" on page 161 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.

The programming language SCL permits the overlaying of data types with the keyword AT in the declaration section of code blocks. You program the overlaying

directly after the declaration of the tags to be overlaid. The schema is as follows: *var_new* AT *var_old* : new_data type. Example:

```
VAR_INPUT
    Date            : DT;
    Byte_array AT date : ARRAY [1..8] OF BYTE;
END_VAR
```

You can now address the total tag in the program of the block using *#Date* or individual components such as the day using *Byte_array[3]*.

Overlaying a data type is described in Chapter 4.4 “Elementary data types” in section “Overlaying tags (data type views with SCL)” on page 121.

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 9.1.2 “Structure of an STL statement” on page 316 for the syntax of an STL statement and to Chapter 10.1.2 “SCL statements and operators” on page 365 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.

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. No networks are possible with SCL.

If the source file contains blocks which are called in the source file or if data operands are accessed, you should observe a specific sequence in the source file. The blocks or data operands should be located before the position of use in the source file.

When calling a block, you enter the block parameters in round parentheses, each separated by a comma. Make sure that the transferred block parameters are listed in the same order as they have been declared in the called block.

Fig. 18.1 shows an example of an STL source file for a function block with the associated instance data block.

Fig. 18.2 shows an example of an SCL source file for a function block with the associated instance data block.

18.1.3 Programming a data block in the source file

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.

The keyword KNOW_HOW_PROTECT protects the block from unwanted access. You can no longer cancel this protection, in contrast to block protection with password in the TIA Portal.

Block interface

The block interface contains the declaration of the data operands. With a global data block, you program the data operand here, with a type data block, you specify the assigned PLC data type, and with an instance data block, you assign the associated function block.

The declaration of a tag in a global data block consists of the name, the data type, possibly a default setting, and an optional tag comment. Example:

```
Quantity : INT := +500;  //Units per batch
```

The tag order can be random. If you combine tags with data type BOOL and also combine byte-wide tags with data types BYTE and CHAR, you can minimize the memory requirements (see also Chapter 18.5.1 “Storage in global data blocks” on page 712).

The declaration in a type data block consists only of the specification of the assigned PLC data type.

The declaration in an instance data block consists only of the specification of the assigned function block.

```
FUNCTION_BLOCK "FIFO_STL"
TITLE = Intermediate memory for 4 values
//Example of a function block in STL
AUTHOR  : Berger
FAMILY  : Book_300
NAME    : Memory
VERSION : 01.00
VAR_INPUT
  Transfer     : 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 Import
      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  : Book_300
NAME    : FIFO_Dat
VERSION : 01.00
FIFO_STL                           //Instance for the FB "FIFO_STL"
BEGIN
  Value1 := 1.0;                   //Individual default setting
  Value2 := 1.0;                   //of selected values
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  : Book_300
NAME    : Memory
VERSION : 01.00

VAR_INPUT
  Transfer    : 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  : Book_300
NAME    : FIFO_Dat
VERSION : 01.00

FIFO_SCL                           //Instance for the FB "FIFO_SCL"

BEGIN
  Value1 := 1.0;                   //Individual default setting
  Value2 := 1.0;                   //of selected values
END_DATA_BLOCK
```

Fig. 18.2 Example of an SCL source file

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 comment	Block title Block comment
	KNOW_HOW_PROTECT UNLINKED READ_ONLY NON_RETAIN	Know-how protection (cannot be canceled) Block attribute: not executable Block attribute: read-only Block attribute: non-retentive
	NAME : *Block name* FAMILY : *Block family* AUTHOR : *Created by* VERSION : *Version*	Block property: Block name Block property: Block family Block property: Created by Block property: Block version
Declaration	STRUCT name : Data type := Default setting; END_STRUCT Data type_name FB_name	For a global data block Alternatively for a type data block 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

Default setting with start values

The initialization part starts with BEGIN and ends with END_DATA_BLOCK. You must specify these keywords even if you do not assign default values to the tags in the initialization part.

By assigning default values to the start values, it is possible to assign individual values to each application (each instance) in the case of type and instance data blocks.

If you do not specify a start value for a data operand, the value from the block interface is used: The default value is retained for a global data block and the default value in the PLC data type or in the function block then applies to a type or instance data block.

18.1.4 Programming a PLC data type in the source file

Table 18.3 shows which keywords you require for data type programming and the sequence in which the keywords are used.

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.

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; END_STRUCT	Declaration of data type components
Block end	END_TYPE	End of the PLC data type

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 component consists of the name, the data type, possibly a default setting, and an optional comment. Example:

```
Quantity : INT := +500;     //Units per batch
```

18.2 Migrating projects

Automation projects which have been created using STEP 7 Versions 5.4 SP5 and 5.5 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 V11 can be upgraded to Version V12.

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 components 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 instructions are converted into the representation which is standard for the TIA Portal and may therefore deviate from the previous representation. System blocks and standard blocks from a standard library are converted into instructions which can be found in the program elements catalog under *Extended instructions*.

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 data type UDT) is retained.

Migration of blocks with know-how protection

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.

If the program sources of SCL blocks are missing in the original project, the blocks are migrated into blocks with know-how protection. If the program sources are present, the blocks are migrated into non-protected blocks. If applicable, you must

then protect the associated blocks again using the *Edit* > *Know-how protection* command.

Migration of LAD and FBD blocks

Blocks with jump functions 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 to STL and subsequently back to LAD or FBD again.

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.

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, also with a different name if there is a name conflict), symbolic constants as limits for the ARRAY declaration (these are replaced by fixed values), nested ARRAY tags (these are replaced by multi-dimensional arrays), the DIV operator (this is replaced by the slash "/"), and the EXPD function (this is replaced by the notation "10**").

The program in SCL blocks is changed as follows: A floating-point number is always specified as a fractional number (for example, "12E2" becomes "12.0E2"), indirect addressing uses round parentheses (for example, MW[12] becomes MW(12)), and the absolute or indirect addressing of data operands is only possible with a data block addressed in absolute mode (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.

Upgrading projects to STEP 7 V12 SP1

A project created with STEP 7 V11 can also be edited with STEP V12. Backwards compatibility is retained, which means that you can then 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* > *Open* from the main menu. Then select the *Project* > *Upgrade* command in the main menu. After confirming the prompt, the original project is closed without changes and the new version of the project is saved with the name *<Project_name>_<Project_version>* and opened for editing. To conclude the upgrade, compile each station in the new project with the command *Edit* > *Compile* from the main menu.

Projects created with STEP 7 V10.5 or STEP 7 V12 (without SP1) are upgraded immediately upon opening after confirming the prompt.

18.3 Simulation with the TIA Portal

The TIA Portal contains the S7-PLCSIM simulation software which emulates a programmable controller. As a result, you can use the programming device to test the user program without additional hardware.

No connections to “real” programmable controllers must exist when using the simulation. Any existing online connections are cleared when starting the simulation.

The program of only one station can be simulated at a time.

18.3.1 Differences from a real CPU

The simulation software cannot emulate a real programmable controller to 100%, but can additionally execute a number of useful functions. Some differences from a “real” CPU 300 are listed below.

The user program in a simulation can be executed cyclically or automatically. Cyclically means that the program is executed completely once and only starts the next processing cycle when requested.

The output statuses of the simulated CPU are not changed when at STOP.

In the simulation subwindows you specify the tag values with which the program is to work. These changes are imported immediately and not only at the cycle control point as with a CPU 300. In the case of an input, the modified signal state is also copied immediately into the I/O area I:P so that the modified value is not overwritten during the next updating of the process image. A modified signal state of an output immediately updates the corresponding I/O output Q:P. Forcing is not supported by PLCSIM.

You can reset individual or all SIMATIC timers or assign them different time values using a menu command. Processing of error and interrupt organization blocks can be triggered manually.

Modules which are inserted during ongoing operation can be detected and configured by a CPU 300. PLCSIM does not support this automatic configuration. PLCSIM works with the modules which are present in the loaded hardware configuration.

PLCSIM does not support any FM modules. PLCSIM does not support all system blocks. Non-supported system blocks are ignored in the simulation.

PLCSIM simulates four accumulators

The simulation software works with four accumulators and not with two like a CPU 300. This can result in unintended effects in certain STL statement sequences which use arithmetic functions and the POP statement.

Example:

`L #var1`
`L #var2`
`+R`
`+R`
`T #var3`
	With an arithmetic operation, the contents of accumulator 3 are "fetched" into accumulator 2. The previous contents of accumulator 2 cannot therefore be used repeatedly as with a CPU 300. The example shown on the left delivers a result in the simulation which is different from that in a CPU 300.

The simulation uses the POP statement to shift the contents of accumulator 2 into accumulator 1 and the contents of accumulator 3 into accumulator 2. With a CPU 300, POP does not change the contents of accumulator 2.

18.3.2 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 the simulation software.

To start the simulation, select the station or the *Program blocks* folder in the project tree and then select the *Online* > *Simulation* > *Start* command from the main menu. Following the start, PLCSIM is presented in a separate window which contains the "subwindow" CPU (Fig. 18.3). The simulated user program is supplied with tag values via further subwindows – for example for inputs and outputs.

Using the *View* > *Always On Top* command you can ensure that the simulation window is not covered by other windows. To set the PG/PC interface, select the access mode to the simulated CPU from the drop-down list in the toolbar, for example MPI or TCP/IP.

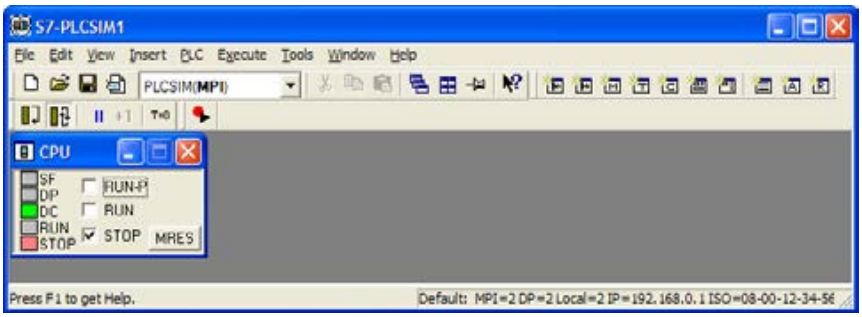


Fig. 18.3 Input window of the PLCSIM simulation software

The simulated CPU responds like a real CPU switched online. You establish a connection to the simulated CPU and can then use the online and diagnostics tools. Following starting, the simulated CPU is displayed in the project tree under *Online access > PLCSIM V5.x [access mode]*.

When downloading the user program, proceed as described in Chapter 15 “Online operation and program test” on page 564.

Saving and restoring the simulation

You can save the current state of a simulation and then load this state again later for continued processing.

To save the automation system for the first time, select the *File > Save PLC As ...* command and specify the storage location. You can save an intermediate state of the simulation using *File > Save PLC*. The <name>.plc file saves the operating state, the user program, the hardware configuration, and the current signal states of the simulated programmable controller.

You can save the current subwindows using *File > Save Layout as*. The <name>.lay file saves the current simulation windows together with their positions and contents. You can save the window arrangement in a previously created file using *File > Save Layout*.

You restore the simulation using *File > Open PLC* and *File > Open Layout*. Using *File > Recent Simulation > ...* and *File > Recent Layout > ...* you can select the files which have been saved last from a list.

18.3.3 Using the simulation

Controlling the CPU

The CPU subwindow emulates the control and display elements of a CPU. LED symbols indicate the current operating mode (RUN or STOP) and a group error (SF). You control the operating mode by activating checkboxes: STOP, RUN, and RUN-P. If the RUN checkbox is activated, the user program cannot be modified when in

RUN mode. With the RUN-P checkbox activated, the simulated CPU responds like a CPU 300 in RUN mode.

You can use the MRES button to trigger a memory reset for the simulated CPU.

Addressing operands

You use the bit, byte, word, or doubleword address when addressing operands, for example IB12 for the input byte 12. In the case of a byte address, you can individually monitor and control the bits of the byte. The peripheral operand area is addressed using PI (peripheral inputs) or PQ (peripheral outputs).

For a data operand you use the complete addressing with data block and data operand, for example DB10.DBW24.

The operand ID for a SIMATIC timer function is “T” and “C” for a SIMATIC counter function.

Subwindows for inputs, outputs, bit memories, and data

You can insert an empty subwindow into the working window of the simulation using *Insert* > *Input Variable*, ... > *Output Variable*, ... > *Bit Memory* or ... > *Generic*. Each subwindow accommodates one operand. You can insert any number of subwindows and arrange them as desired in the simulation window (Fig. 18.4).

You can use these subwindows to monitor and control operands from the following operand areas: inputs (I), peripheral inputs (PI), outputs (A), peripheral outputs (PQ), bit memories (M), and data (DB).

You enter the operands in the left box. With peripheral inputs and outputs it is possible to specify a byte (PIB, PQB), a word (PIW, PQW), or a doubleword (PID, PQD).

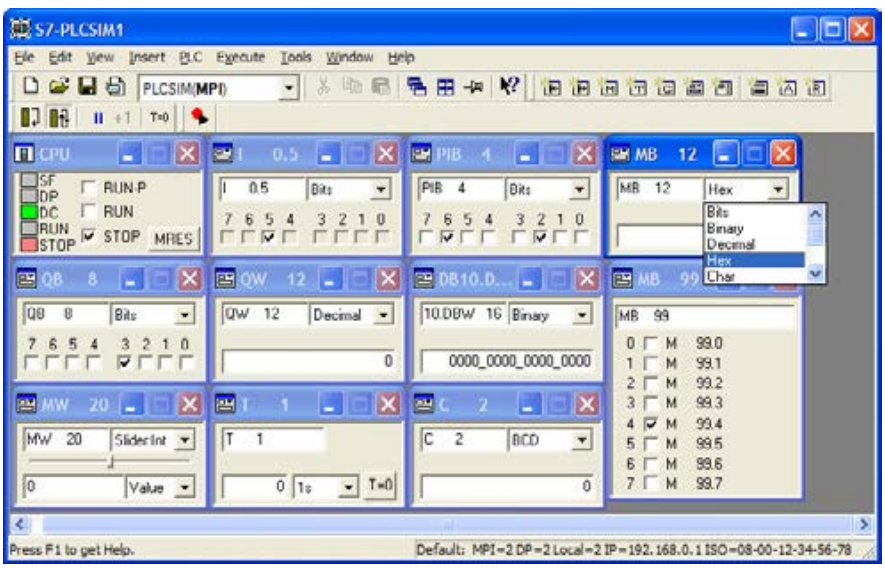


Fig. 18.4 Subwindows for monitoring and control

You address a data operand together with the data block (“complete addressing”). Depending on the operand width (bit, byte, word, or doubleword), select the data format from the drop-down list with which you wish to display and control the operand value.

If you select the *Slider* format from the drop-down list, a slider is displayed which you can adjust using the mouse pointer. INT and DEC are available as display formats. You set the limits for the adjustable range using *Min* and *Max* from the bottom drop-down list. The adjustable value is displayed using the *Value* setting from this drop-down list.

Subwindow “Vertical bits”

You can use *Insert* > *Vertical bits* to insert a subwindow into the simulation window, which arranges the bits among themselves and which can be labeled with the absolute address.

Subwindows for SIMATIC timers and counters

You insert a subwindow for a SIMATIC timer or counter function using *Insert* > *Timer*, *Insert* > *Counter*, or *Insert* > *Generic*. The subwindow for a timer function contains the duration divided according to time value (0 to 999) and time frame (10 ms, 100 ms, 1 s, 10 s). You can select binary, hexadecimal, BCD, and S7 format (W#16#xxxx) as the formats for a count value.

If you specify a timer or counter operand in a subwindow for inputs, outputs or bit memories, the current timer or counter value is displayed in selectable formats (also with slider).

Using the *Execute* > *Manual Timers* command you can stop time processing by the simulation and you can set time values in the subwindows. Using the *Execute* > *Automatic Timers* command, the simulation continues with processing of the timer functions.

You can reset all or individual timer functions. With the *Execute* > *Reset Timers* command you can select in the dialog window whether you wish to reset all timer functions or only a specific timer function. You can do the same using the *T=0* button (reset timers) in the *CPU mode* toolbar (all timers) or the *T=0* button in the bottom window of the timer.

Further subwindows

You can use the *View* > ... command from the main menu to insert subwindows for CPU-internal registers (Fig. 18.5).

The *View* > *Accumulators* command shows the *ACCUs & Status Word* subwindows. This shows the assignments of the accumulators (four with the simulation), the address registers, and the status word.

The *View* > *Block Registers* command shows the *Block Regs* subwindow. This shows the current assignment of the data block registers, the current and previous code blocks, and the value of the STEP address counter (SAC) an.

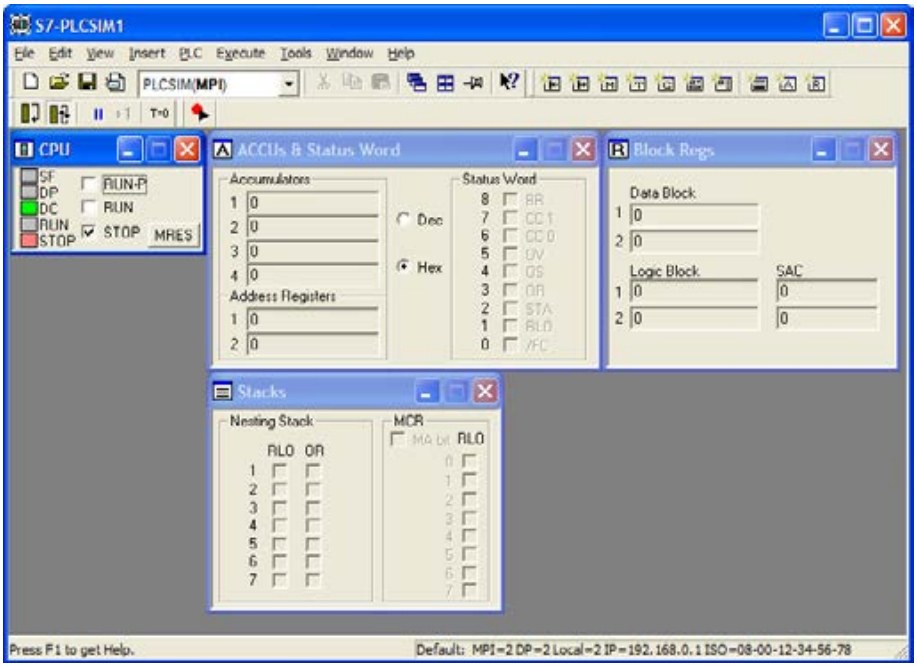


Fig. 18.5 Subwindows for the CPU-internal registers

The *View* > *Stacks* command shows the *Stacks* subwindow. This shows the used depth of the nesting stack and the assignments of the status bits RLO and OR. In addition, the current level of the master control relay (MCR) is shown with the status bit RLO and the MCR activity bit.

18.3.4 Testing the program with the simulation

The program has been loaded into the simulation and at least the subwindow for the CPU is present. You can use this subwindow to trigger the change between STOP state and RUN mode. The green RUN LED indicates a successful start and the red STOP LED and the red SF LED (group error) indicate a “serious” error in the user program.

You simulate switching on and off of the power supply using *PLC* > *Power on* and *PLC* > *Power off*. The green DC LED indicates that a supply voltage is present.

You can immediately stop the continuous cyclic program execution using the *Pause* button in the toolbar, or at the end of the cycle using the *Single Scan* button (alternatively using the *Execute* > *Scan Mode*> *Single Scan* command). Each click on the *Next Scan* button requests the simulation to execute one single further cycle. Clicking on the *Continuous Scan* button reestablishes continuous program execution.

The cycle time monitoring is switched on as standard and set to the maximum value 6000 ms. You can use *Execute* > *Scan Cycle Monitoring*... to deactivate it or to set a different value.

Online functions, watch tables, and program status

The programming device treats the simulated programmable controller like a real one. With the simulation switched on, you can connect the station online and then use the online functions such as reading the diagnostic buffer. The simulated CPU can be started and stopped using the CPU control panel in the online tools. The further possibilities offered by STEP 7 are described in Chapter 15.4 “Hardware diagnostics” on page 583.

You can also use the test functions of STEP 7 on the simulated CPU. Testing with watch tables and program status is described in Chapter 15.5 “Testing the user program” on page 588. Forcing with a force table is not possible with a simulated CPU.

Error and interrupt organization blocks

If a program error occurs, e.g. access to an operand which does not exist, PLCSIM responds by calling the synchronous error organization block OB 122. If this is not present in the program, the simulated CPU goes to STOP. The further procedure is as when testing with a real CPU.

For testing purposes, you can also manually trigger the starting of an error or interrupt organization block. You can use *Execute* > *Trigger error OB* > ... to start single processing of the selected organization block. You set the triggering conditions (with a real CPU), for example the errors which can trigger a diagnostic interrupt, in a dialog window (Fig. 18.6).

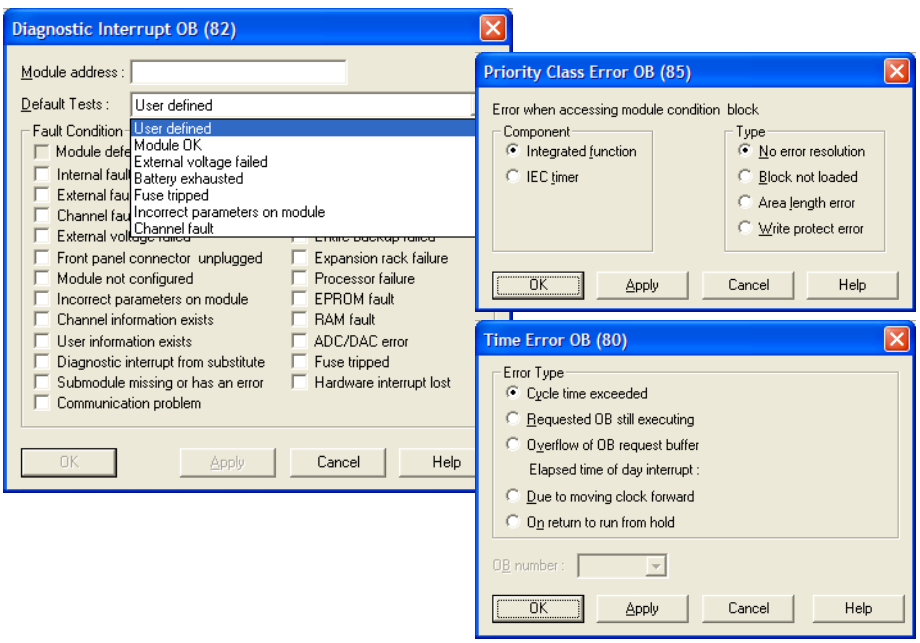


Fig. 18.6 Examples of trigger conditions with error and interrupt organization blocks

18.3.5 Additional functions of PLCSIM

Recording and playback

To achieve (simple) simulation of a connected machine or controlled process, change the operand values in the subwindows into a sequence meaningful for testing. You can record this sequence of data modifications and play it back again at different speeds in order to observe the response of the user program.

You can start recording and playback using the *Tools* > *Record/Playback* command. You are provided with a window with function symbols for controlling the recorder functions.

Initially create a file using the *New event file* symbol in which you wish to save your recorded inputs. Click on the *Record* symbol to start recording. The subsequent changes in values in the subwindows are saved in chronological order. The *Pause* symbol interrupts the recording, the *Stop* symbol terminates the recording. Save the recording using the *Save event file* symbol.

To play back, fetch the saved inputs using *Open event file*. You can initially use the *Delta* symbol to set the speed at which the recording is to be played back. The relative intervals between the individual inputs are retained. Start the playback using the *Play* symbol and terminate it using the *Stop* symbol.

S7ProSim

PLCSIM uses S7ProSim as a COM object or ActiveX control which can be used in applications which support Microsoft's OLE/COM technology. You can thus expand PLCSIM by process simulation, where software development knowledge in Visual Basic or Visual C++ is a requirement.

18.4 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 on the HTML pages.

18.4.1 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. You can use the dialog window *Enable system diagnostics* to permit the display of diagnostic information in the web server by means of *Report System Errors (RSE)* in order to obtain the complete functionality of the web pages on the module status, topology, and messages.

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 project language used, and the user management.

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.

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. 18.7).

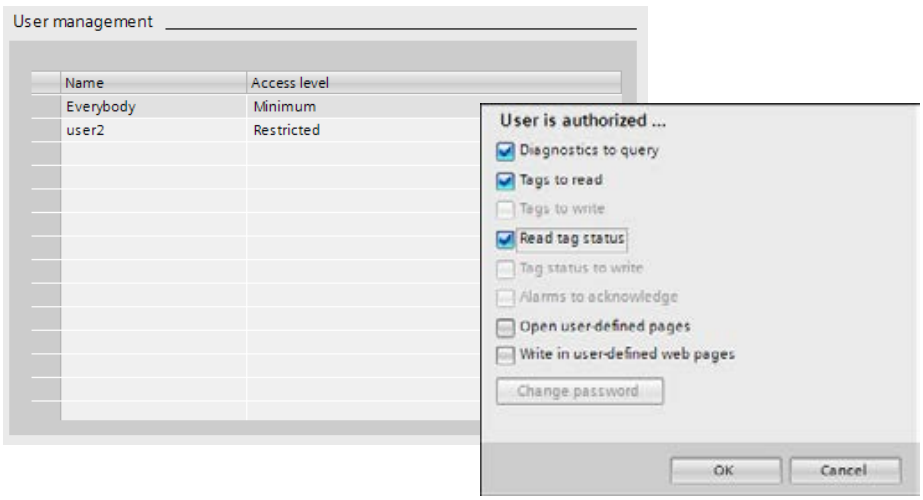


Fig. 18.7 User management in the web server

18.4.2 Reading out Web information

In order to access the CPU's Web server, the PC or PG must establish an Ethernet connection (TCP/IP) to the CPU. Start the Web browser and enter the CPU's IP address as URL in the form *http://aaa.bbb.ccc.ddd* or – for a secure connection – *https://aaa.bbb.ccc.ddd*.

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.

To enable logging on, two input boxes are provided for the user name and password on every page at the top left.

18.4.3 Standard Web pages

The first page displayed by the Web server is the Welcome page. From here, click on ENTER to reach the Start page. 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 state, diagnostics state, and position of the mode switch.

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 diagnostic buffer can save up to 500 messages. Select the group of 100 to be displayed from the drop-down list. Detailed information is displayed on the selected event (Fig. 18.8)

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.

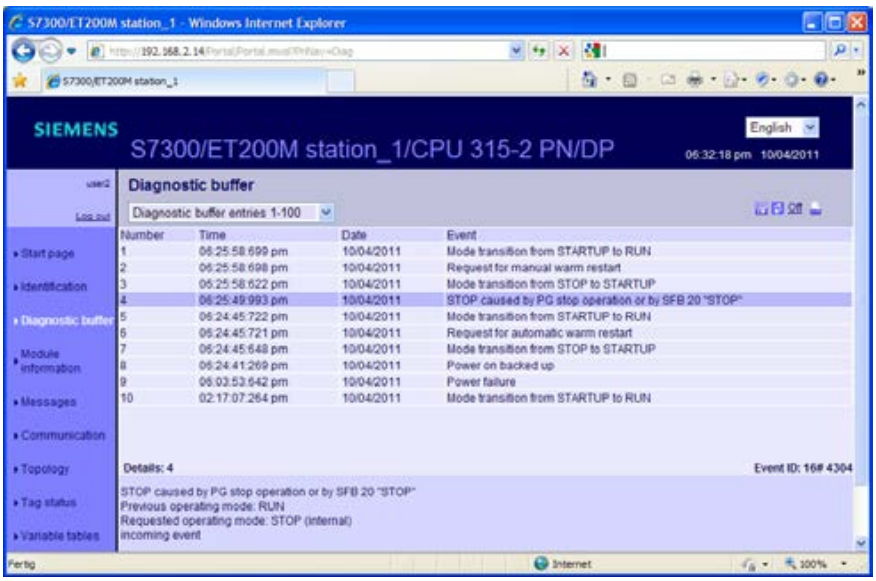


Fig. 18.8 Display of diagnostic buffer in the Web browser

Module information

Prerequisites for display on the *Module information* page are:

- Enabling of system diagnostics with *Report System Errors* with the hardware configuration in the CPU properties under *System diagnostics*
- Cyclic calling of the function block RSE_FB *Signal system errors* at least every 100 ms, with an excessively long cycle time in a cyclic interrupt OB with a time interval of at least 100 ms
- Enabling of diagnostics support in the CPU properties in the *Diagnostics support* group by means of the *Diagnostics status DB* checkbox

The *Module information* page shows the CPU's status. From here you can call up the status of individual modules. Use the link in the “Heading” to access a higher module level, the links in the table column *Name* to access lower levels.

The module information of a component is displayed by various symbols: *Component is OK, Disabled PROFIBUS slaves or PROFINET devices, Component cannot be accessed, Maintenance required, Maintenance request, Component failed or faulty,* and *Error in a lower module level*.

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.

The selected display classes of the messages are displayed during operation in plain text on the web page *Messages*, the messages of non-selected display classes in hexadecimal code. You can reduce the memory requirements of the configuration data present in the load memory by only selecting the display classes which are actually required.

You configure the display classes for *Report System Errors* in the CPU properties under *System diagnostics* and for block-related (PLC) messages with the message editor.

Communication

The *Communication* page contains the *Parameter*, *Statistics*, *Resources*, and *Open communication* tabs.

Information on the PROFINET interface can be found in the *Parameters* and *Statistics* tabs. The MAC address and the IP address are displayed, for example, as are statistical analyses of sent and received data packages.

The *Resources* tab shows the number of available, reserved and occupied connections. The status of the communication connections are shown in the *Open communication* tab.

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 the connection relationships and thus permits fast locating of the error location.

Tag status

On the *Tag status* page you can monitor the status of up to 50 tags. When you specify the address of the tag and the display format, you receive the value of the tag.

You can select the display language in the window at the top right. When specifying addresses, please note that the mnemonics for English (e.g. "I" for input) differs from those of other languages ("E" in German, for example). Syntax errors are indicated in red.

Variable tables (watch tables)

On the *Variable tables* page you have the opportunity to make your defined watch tables accessible to users via the web browser. You have previously selected an existing table by means from the drop-down list in the CPU properties under the *Web server > Watch tables* group, and specified by means of the drop-down list of the second column whether a read or write access is to be permissible.

The web server allows you to monitor up to 50 watch tables with up to 200 tags each. The memory space available in the CPU might not be sufficient to make use of all the possibilities. If watch tables are displayed incompletely, reduce the memory required by the messages and symbol comments. If possible, use only one language and keep the number of tags per table low.

To display a watch table, select one of the available tables (previously configured in the web server) from the drop-down list.

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.

WWW Initialize Web server and synchronize Web pages

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. 18.9).

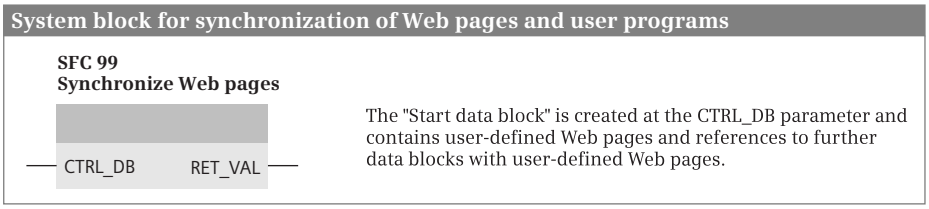


Fig. 18.9 Graphic representation of system function WWW

18.5 Storage of local tags

Block parameters are stored differently in the case of functions and function blocks. You as a user not need be bothered by this; you program the parameters for both types of block in the same manner. However, this difference is very important for direct access to the block parameters.

When programming a function block for which the *Multiple instance capability* attribute is activated, you need not be bothered in the case of symbolic addressing of the local data tags whether this function block is subsequently called as a single instance or local instance. However, direct access is then only possible indirectly via the address register AR2.

18.5.1 Storage in global data blocks

The program editor stores the individual tags in the data block in the sequence of their declaration. The following rules essentially apply:

- The first bit tag of an uninterrupted declaration sequence is in bit 0 of the next byte, and this is followed by the next bit tags.
- Byte tags are stored in the next byte.
- Word and doubleword tags always commence at a word limit, i.e. at a byte with even address.
- DT and STRING tags commence at a word limit.
- ARRAY tags commence at a word limit and are “filled” up to the next word limit. This also applies to bit and byte fields. Field components with elementary data

types are stored as described above. Field components with higher data types commence at word limits. Each dimension of a field is oriented like an independent field.

- STRUCT tags commence at a word limit and are “filled” up to the next word limit. This also applies to pure bit and byte structures. Structure components with elementary data types are stored as described above. Structure components with higher data types commence at word limits.

By combining bit tags and by arranging byte tags in pairs you can accommodate your data in a data block with optimum use of memory space.

Fig. 18.10 shows an example of non-optimized data storage. Note that the editor must always “fill” ARRAY and STRUCT tags up to the next word; i.e. no bit or byte tags can be placed in a resulting byte gap. However, you can arrange the tags optimally within the structure. You can achieve an optimum arrangement in the example if you declare the BYTE tag positioned following the REAL tag in front of the REAL tag, set the BYTE component in the STRUCT tag in front of the INT component, and declare the last declared BYTE tag in front of the DATE tag. The changed sequence in declaration then reduces the memory space requirements by five filler bytes.

18.5.2 Storage in instance data blocks

The program editor stores the tags in an instance data block in the following order:

- Input parameters
- Output parameters
- In/out parameters
- Local tags including local instances

Each tag is saved in the order of its declaration. Each declaration area commences at a word limit, i.e. at a byte with even address. The individual tags are arranged within the declaration areas as described in the previous chapter (as in a global data block). Fig. 18.10 shows an example of the occupation of an instance data block.

18.5.3 Storage in the temporary local data

Storage of the tags in the temporary local data (L stack) corresponds to storage in a global data block. The assignment always commences with the (relative) byte 0. Note with organization blocks that the first 20 bytes are occupied by the start information. The first 20 bytes must be declared even if you do not use the start information – even if you only declare an array with 20 bytes.

The editor itself also uses local data, for example to transfer parameters during a block call. The temporary local data declared symbolically as well as that used by the editor itself is stored by the editor in the sequence of declaration or use. The temporary local data addressed absolutely is not considered here so that overlapping could result if you do not know what local data is created by the editor.

Data storage in a data block

The tags in a data block are stored in the order of their declaration. All tags with a width of 16 bits or more start at a byte with even address. In certain cases, an address must therefore be bypassed in the assignment, and a "filler byte" is inserted. By using an adept sequence for the declaration, it is possible to minimize the number of filler bytes and thus the memory requirements.

All declaration sections for an instance data block commence at a byte with even address, and also the data of a local instance in this case.

The same assignment pattern of a global data block also applies to the temporary local data. With an organization block, the first 20 bytes (bytes 0 to 19) are occupied by the start information. The assignment with user tags then only commences with byte number 20.

The symbol ◁ indicates an even byte address

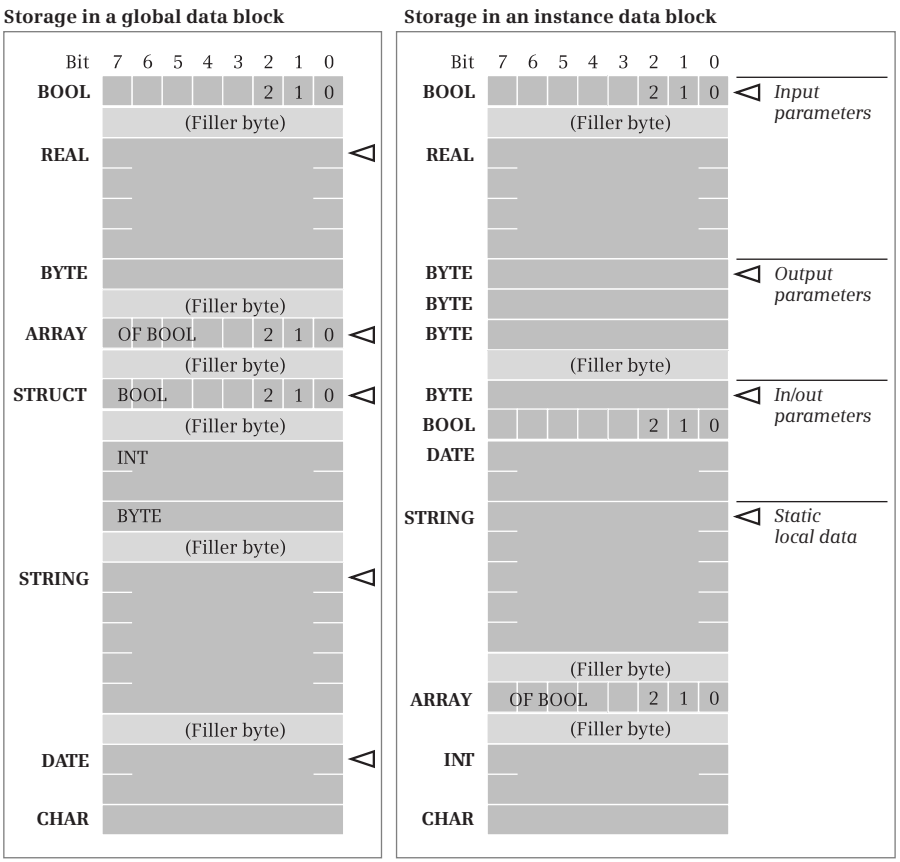


Fig. 18.10 Data storage in data blocks

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 (words, doublewords). 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.

18.5.4 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 18.4). Exception: With the parameter types TIMER, COUNTER, and BLOCK_xx, the pointer is a 16-bit number located in the left word of the block parameter.

Table 18.4 Parameter storage for functions

Data type	INPUT	IN_OUT	OUTPUT
	The parameter is an area pointer to a		
Elementary	Value	Value	Value
Complex	DB pointer	DB pointer	DB pointer
TIMER, COUNTER, BLOCK	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. 18.11). 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.

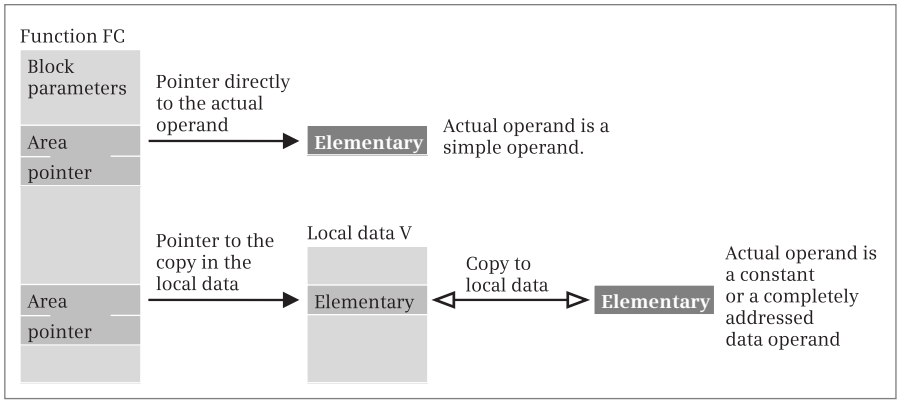
The same applies to loading a corresponding output parameter: Since copying from the "actual" tag from the data block into the V area does not take place, you load an (indefinite) value from the V area in this case.

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

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.



Pointer to a further pointer

If the block parameter has a complex 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.

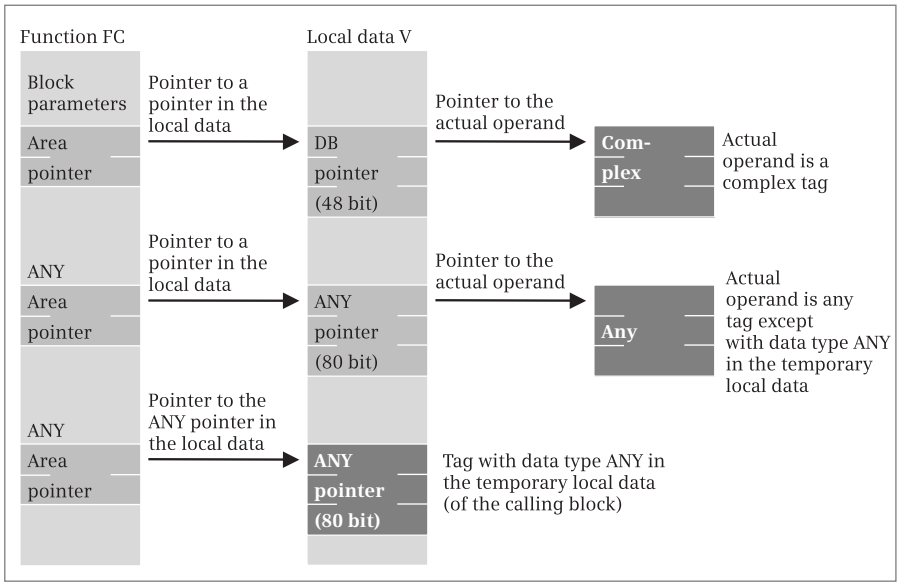


Fig. 18.11 Parameter transfer for functions (FC)

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 complex data types (DT, STRING, ARRAY, STRUCT, and UDT), the actual parameters are 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, and BLOCK_xx, 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 complex 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 complex 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 Chapters 4.6.3 ““Variable” ANY pointer with STL” on page 139 and 4.6.4 ““Variable” ANY pointer with SCL” on page 140.

18.5.5 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 18.5). When storing as a value, the memory space required depends on the data type of the block parameter; 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. 18.12. When copying actual parameters with complex data type into the instance data (input parameters) or back to the actual parameter

Table 18.5 Parameter storage for function blocks

Data type	INPUT	IN_OUT	OUTPUT
Elementary	Value	Value	Value
Complex	Value	DB pointer	Value
TIMER, COUNTER, BLOCK_xx	Number	–	–
POINTER	DB pointer	DB pointer	–
ANY	ANY pointer	ANY pointer	–

(output parameters), the program editor uses the system function BLKMOV whose parameters are formed in the temporary local data area of the calling block.

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.

18.5.6 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.

The data of a local instance is a subset of the static local data of the calling function block (Fig. 18.13). The local instance has a name which you define during program-

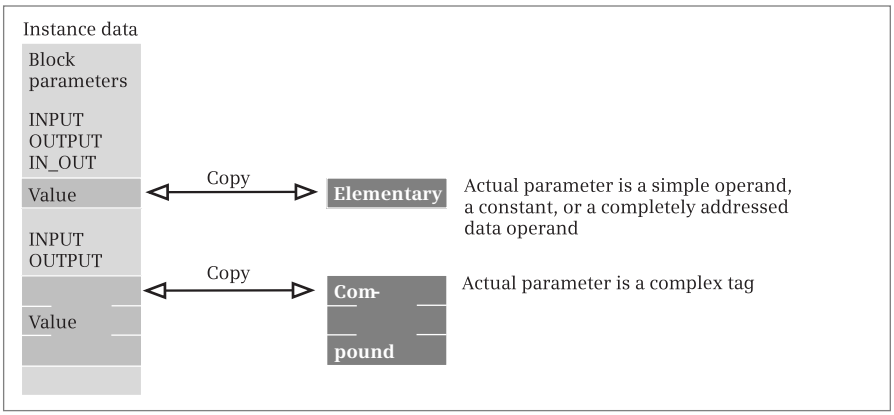
Parameter transfer with function blocks

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 compound data type.

If the block parameter has a data type TIMER, COUNTER or BLOCK_xx, the number of the timer or counter function or of the block is present in the instance data.



Pointer in the instance data

If an in/out parameter has a compound 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.

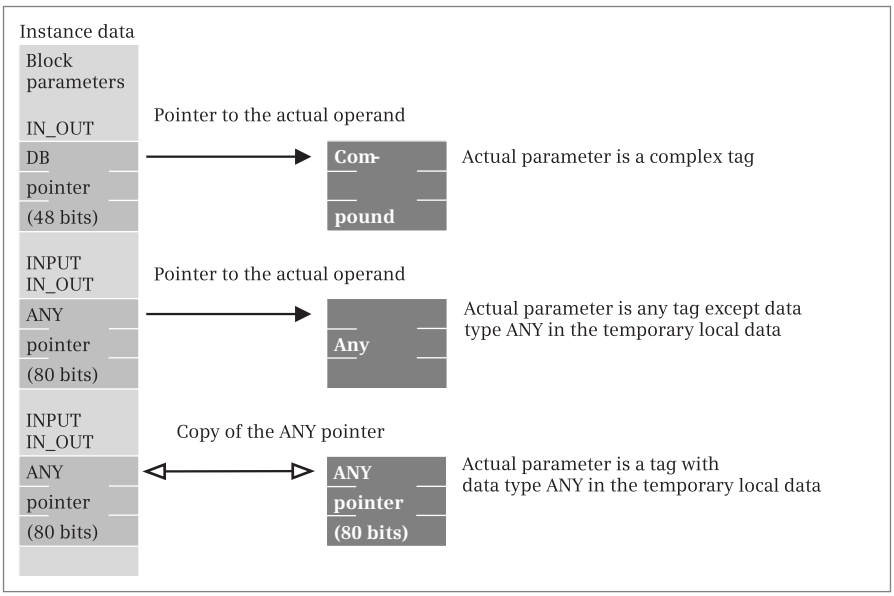


Fig. 18.12 Parameter transfer with function blocks (FB)

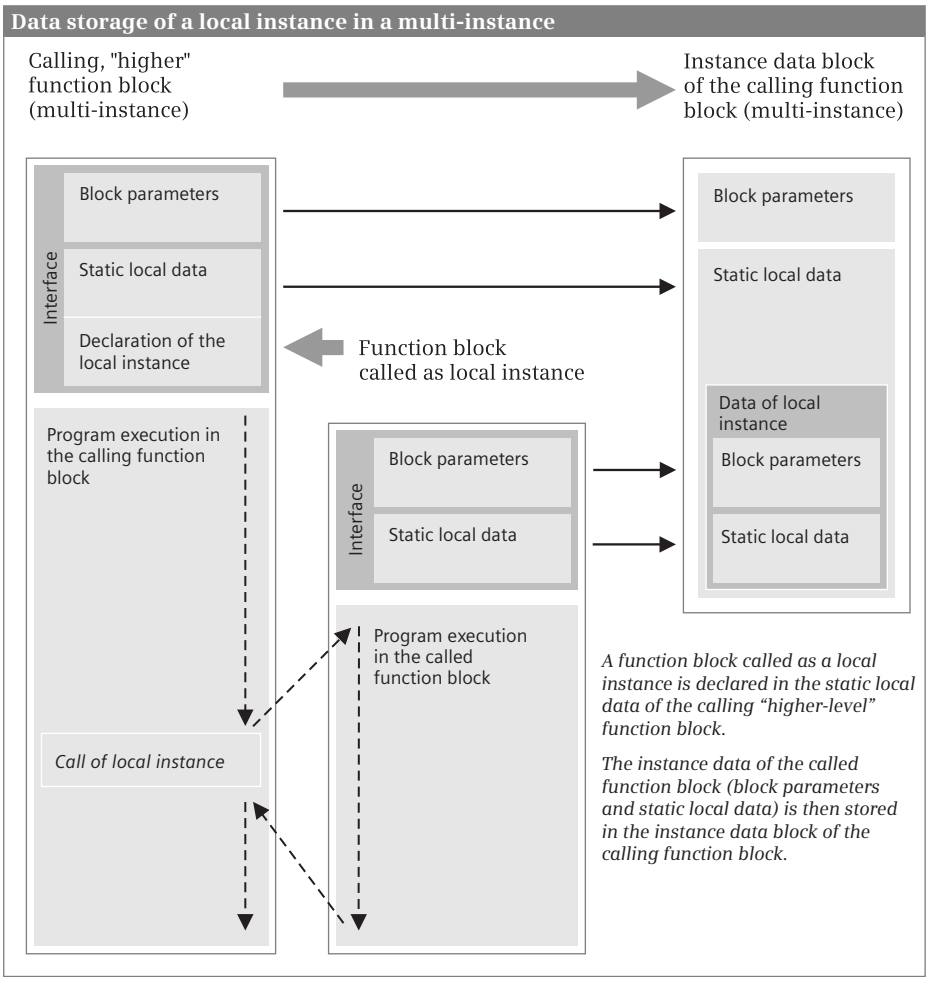


Fig. 18.13 Data storage of a local instance in a multi-instance

ming 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.

You handle the call of a system function block (SFB) just like the call of a function block. Whereas the program of the SFB is present in the CPU's operating system, the instance data of the SFB is stored in the user memory.

Index