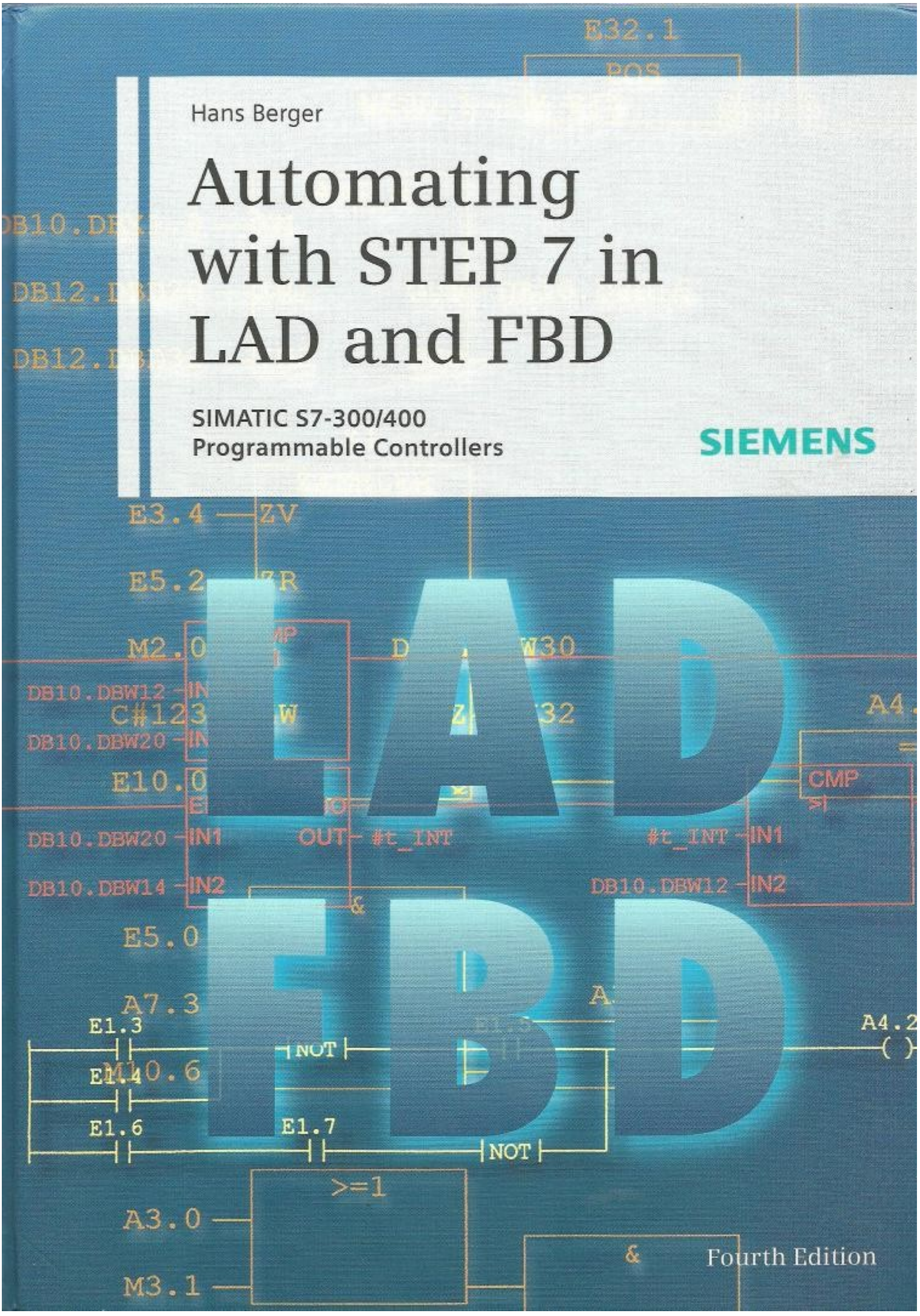
Hans Berger

Automating with STEP 7 in LAD and FBD

SIMATIC S7-300/400
Programmable Controllers

SIEMENS



Fourth Edition

Hans Berger

Automating with STEP 7 in LAD and FBD

Ladder diagram (LAD) and function block diagram (FBD) are the graphic-oriented programming languages in the programming software STEP 7. Now in its fourth edition, this book introduces in the latest version of STEP 7 with new functions. It describes elements and applications for use with both SIMATIC S7-300 and SIMATIC S7-400 including the applications with PROFINET. Special functions like PROFINET IO, SFC 109 Protect and function blocks for fieldbus systems are also described.

It is aimed at all users of SIMATIC S7 controllers. First time users are introduced to the field of programmable controllers, while advanced users learn about specific applications of the SIMATIC S7 automation system.

SIMATIC is the worldwide established automation system for implementing industrial control systems for machines, manufacturing plants and industrial processes. Relevant open-loop and closed-loop control tasks are formulated in various programming languages with the programming software STEP 7.

All programming examples found in the book – and even a few extra examples – are available over the publisher's website under Downloads.

Contents

Order No. A19100-L531-B951-X-7600
4th edition, 2008

Publicis Corporate Publishing
www.publicis.de/books

ISBN 978-3-89578-297-8

Automating with STEP7 in LAD and FBD

Programmable Controllers
SIMATIC S7-300/400

by Hans Berger

4th revised and extended edition, 2008

Publicis Corporate Publishing

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

The programming examples concentrate on describing the LAD and FBD functions and providing SIMATIC S7 users with programming tips for solving specific tasks with this controller.

The programming examples given in the book do not pretend to be complete solutions or to be executable on future STEP 7 releases or S7-300/400 versions. Additional care must be taken in order to comply with the relevant safety regulations.

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

The author and publisher are always grateful to hear your responses to the contents of the book.
Publicis Corporate Publishing
P.O. Box 3240
D-91050 Erlangen
E-mail: publishing-books@publicis.de

Internet: www.publicis.de/books

ISBN 978-3-89578-297-8

4th edition, 2008

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

Printed in Germany

Preface

The SIMATIC automation system unites all the subsystems of an automation solution under uniform system architecture into a homogeneous whole from the field level right up to process control. This Totally Integrated Automation (TIA) concept permits integrated configuring, programming, data management and communications within the complete automation system. Fine-tuned communications mechanisms permit harmonious interaction between programmable controllers, visualization systems and distributed I/Os.

As the basic tool for SIMATIC, STEP 7 handles the parenthesis function for Totally Integrated Automation. STEP 7 is used to carry out the configuration and programming of the SIMATIC S7, SIMATIC C7 and SIMATIC WinAC automation systems. Microsoft Windows has been selected as the operating system, thus opening up the world of standard PCs with the user desktop widely used in the office environment.

For block programming STEP 7 provides programming languages that comply with DIN EN 6.1131-3: STL (statement list; an Assembler-like language), LAD (ladder logic; a representation similar to relay logic diagrams), FBD (function block diagram) and the S7-SCL optional package (structured control language, a Pascal-like high-level language). Several optional packages supplement these languages: S7-GRAPH (sequential control), S7-HiGraph (programming with state-transition diagrams) and CFC (connecting blocks; similar to function block diagram). The various methods of representation allow every user to select the suitable control function description. This broad adaptability in representing the control task to be solved significantly simplifies working with STEP 7.

This book describes the LAD and FBD programming languages for S7-300/400. As a valuable supplement to the language description, and following an introduction to the S7-300/400 automation system, it provides valuable and practice-oriented information on the basic handling of STEP 7 for the configuration of SIMATIC PLCs, their networking and programming. The description of the "basic functions" of a binary control, such as e.g. logic operations or storage functions, is particularly useful for beginners or those converting from contactor controls to STEP 7. The digital functions explain how digital values are combined; for example, basic calculations, comparisons or data type conversion.

The book shows how you can control the program processing (program flow) with LAD and FBD and design structured programs. In addition to the cyclically processed main program, you can also incorporate event-driven program sections as well as influence the behavior of the controller at startup and in the event of errors/faults. The book concludes with a general overview of the system functions and the function set for LAD and FBD. The contents of this book describe Version 5.4 Service Pack 3 of the STEP 7 programming software.

Erlangen, May 2008

Hans Berger

The Contents of the Book at a Glance

Overview of the S7-300/400 programmable logic controller

PLC functions comparable to a contactor control system

Handling numbers and digital operands

Introduction

1 SIMATIC S7-300/400 Programmable Controller

Structure of the Programmable Controller (Hardware Components of S7-300/400);
Memory Areas;
Distributed I/O (PROFIBUS DP);
Communications (Subnets);
Module Addresses;
Addresses Areas

2 STEP 7 Programming Software

Editing Projects;
Configuring Stations;
Configuring the Network;
Symbol Editor;
LAD/FBD Program Editor;
Online Mode; Testing LAD and FBD Programs

3 SIMATIC S7 Program

Program Processing;
Block Types;
Programming Code Blocks and Data Blocks;
Addressing Variables, Constant Representation, Data Types Description

Basic functions

4 Binary Logic Operations

AND, OR and Exclusive OR Functions;
Nesting Functions

5 Memory Functions

Assign, Set and Reset; Midline Outputs;
Edge Evaluation;
Example of a Conveyor Belt Control System

6 Move Functions

Load and Transfer Functions;
System Functions for Data Transfer

7 Timers

Start SIMATIC Timers with Five Different Characteristics, Resetting and Scanning;
IEC Timer Functions

8 Counters

SIMATIC Counters; Count up, Count down, Set, Reset and Scan Counters;
IEC Counter Functions

Digital functions

9 Comparison Functions

Comparison According to Data Types INT, DINT and REAL

10 Arithmetic Functions

Four-function Math with INT, DINT and REAL numbers;

11 Mathematical Functions

Trigonometric Functions;
Arc Functions;
Squaring, Square-root Extraction, Exponentiation, Logarithms

12 Conversion Functions

Data Type Conversion;
Complement Formation

13 Shift Functions

Shifting and Rotating

14 Word Logic

Processing a AND, OR and Exclusive OR Word Logic Operation

Controlling program execution, block functions

Processing the user program

Supplements to LAD and FBD; block libraries, Function overviews

Program Flow Control

15 Status Bits

Binary Flags,
Digital Flags;

Setting and Evaluating the Status Bits;

EN/ENO Mechanism

16 Jump Functions

Unconditional Jump;
Jump if RLO = "1"
Jump if RLO = "0"

17 Master Control Relay

MCR Dependency,
MCR Area,
MCR Zone

18 Block Functions

Block Call,
Block End;

Temporary and Static Local Data, Local Instances;

Accessing Data Operands
Opening a Data Block

19 Block Parameters

Formal Parameters,
Actual Parameters;

Declarations and Assignments, "Parameter Passing"

Program Processing

20 Main Program

Program Structure;

Scan Cycle Control (Response Time, Start Information, Background Scanning);

Program Functions;

Communications with PROFIBUS and PROFINET;

GD Communications;

S7 and S7 Basic Communications

21 Interrupt Handling

Hardware Interrupts;
Watchdog Interrupts;
Time-of-Day Interrupts;
Time-Delay Interrupts;
DPV1 Interrupts
Multiprocessor Interrupt;

Handling Interrupt Events

22 Restart Characteristics

Cold Restart, Warm Restart, Hot Restart;

STOP, HOLD, Memory Reset;

Parameterizing Modules

23 Error Handling

Synchronous Errors;
Asynchronous Errors;

System Diagnostics

Appendix

24 Supplements to Graphic Programming

Block Protection KNOW_HOW_PROTECT;

Indirect Addressing, Pointers: General Remarks;

Brief Description of the "Message Frame Example"

25 Block Libraries

Organization Blocks;

System Function Blocks;

IEC Function Blocks;

S5-S7 Converting Blocks;

TI/S7 Converting Blocks;

PID Control Blocks;

Communication Blocks

26 Function Set LAD

Basic Functions;

Digital Functions;

Program Flow Control

27 Function Set FBD

Basic Functions;

Digital Functions;

Program Flow Control

The Programming Examples at a Glance

The present book provides many figures representing the use of the LAD and FBD programming languages. All programming examples can be downloaded from the publisher's website www.publicis.de/books. There are two libraries LAD_Book and FBD_Book.

The libraries LAD_Book and FBD_Book contain eight programs that are essentially illustrations of the graphical representation. Two extensive examples show the programming of functions, function blocks and local instances (Conveyor Example) and the handling of data (Message Frame Example). All the examples contain symbols and comments.

Library LAD_Book

Data Types Examples of Definition and Application	**Program Processing** Examples of SFC Calls
FB 101 Elementary Data Types FB 102 Complex Data Types FB 103 Parameter Types	FB 120 Chapter 20: Main Program FB 121 Chapter 21: Interrupt Processing FB 122 Chapter 22: Start-up Characteristics FB 123 Chapter 23: Error Handling
Basic Functions LAD Representation Examples	**Conveyor Example** Examples of Basic Functions and Local Instances
FB 104 Chapter4: Series and Parallel Circuits FB 105 Chapter5: Memory Functions FB 106 Chapter6: Move Functions FB 107 Chapter7: Timer Functions FB 108 Chapter8: Counter Functions	FC 11 Belt Control FC 12 Counter Control FB 20 Feed FB 21 Conveyor Belt FB 22 Parts Counter
Digital Functions LAD Representation Examples	**Message Frame Example** Data Handling Examples
FB 109 Chapter 9: Comparison Functions FB 110 Chapter 10: Arithmetic Functions FB 111 Chapter 11: Math Functions FB 112 Chapter 12: Conversion Functions FB 113 Chapter 13: Shift Functions FB 114 Chapter 14: Word Logic	UDT 51 Data Structure for the Frame Header UDT 52 Data Structure for a Message FB 51 Generate Message Frame FB 52 Store Message Frame FC 51 Time-of-day Check FC 52 Copy Data Area with indirect Addressing
Program Flow Control LAD Representation Examples	**General Examples**
FB 115 Chapter 15: Status Bits FB 116 Chapter 16: Jump Functions FB 117 Chapter 17: Master Control Relay FB 118 Chapter 18: Block Functions FB 119 Chapter 19: Block Parameters	FC 41 Range Monitor FC 42 Limit Value Detection FC 43 Compound Interest Calculation FC 44 Doubleword-wise Edge Evaluation

The libraries are supplied in archived form. Before you can start working with them, you must dearchive the libraries. Select the FILE → DEARCHIVE menu item in the SIMATIC Manager and follow the instructions (see also the README.TXT within the download files).

To try the programs out, set up a project corresponding to your hardware configuration and then copy the program, including the symbol table from the library to the project. Now you can call the example programs, adapt them for your own purposes and test them online.

Library FBD_Book

Data Types Examples of Definition and Application	**Program Processing** Examples of SFC Calls
FB 101 Elementary Data Types FB 102 Complex Data Types FB 103 Parameter Types	FB 120 Chapter 20: Main Program FB 121 Chapter 21: Interrupt Processing FB 122 Chapter 22: Start-up Characteristics FB 123 Chapter 23: Error Handling
Basic Functions FBD Representation Examples	**Conveyor Example** Examples of Basic Functions and Local Instances
FB 104 Chapter4: Series and Parallel Circuits FB 105 Chapter5: Memory Functions FB 106 Chapter6: Move Functions FB 107 Chapter7: Timer Functions FB 108 Chapter8: Counter Functions	FC 11 Belt Control FC 12 Counter Control FB 20 Feed FB 21 Conveyor Belt FB 22 Parts Counter
Digital Functions FBD Representation Examples	**Message Frame Example** Data Handling Examples
FB 109 Chapter 9: Comparison Functions FB 110 Chapter 10: Arithmetic Functions FB 111 Chapter 11: Math Functions FB 112 Chapter 12: Conversion Functions FB 113 Chapter 13: Shift Functions FB 114 Chapter 14: Word Logic	UDT 51 Data Structure for the Frame Header UDT 52 Data Structure for a Message FB 51 Generate Message Frame FB 52 Store Message Frame FC 51 Time-of-day Check FC 52 Copy Data Area with indirect Addressing
Program Flow Control FBD Representation Examples	**General Examples**
FB 115 Chapter 15: Status Bits FB 116 Chapter 16: Jump Functions FB 117 Chapter 17: Master Control Relay FB 118 Chapter 18: Block Functions FB 119 Chapter 19: Block Parameters	FC 41 Range Monitor FC 42 Limit Value Detection FC 43 Compound Interest Calculation FC 44 Doubleword-wise Edge Evaluation

Automating with STEP 7

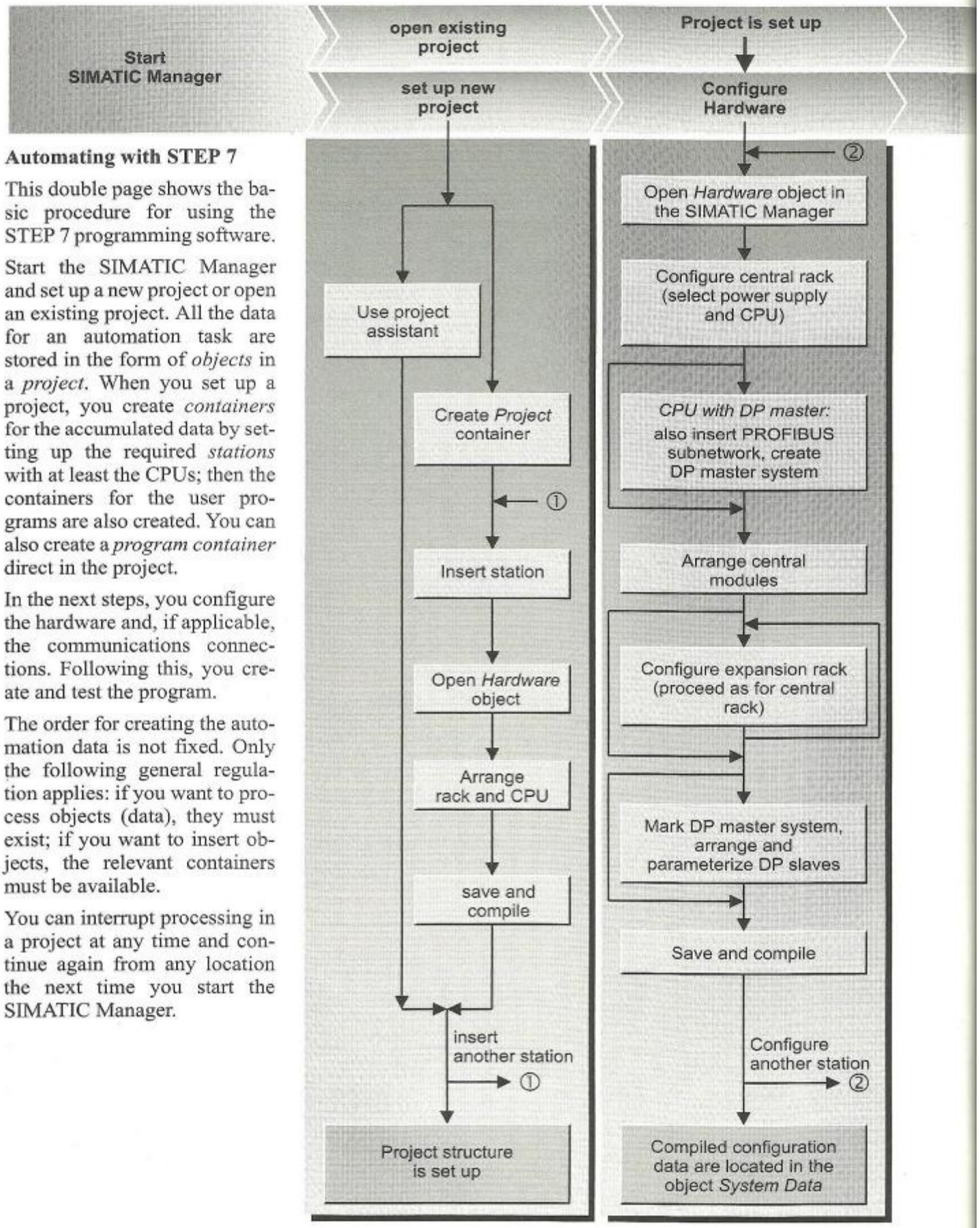
This double page shows the basic procedure for using the STEP 7 programming software.

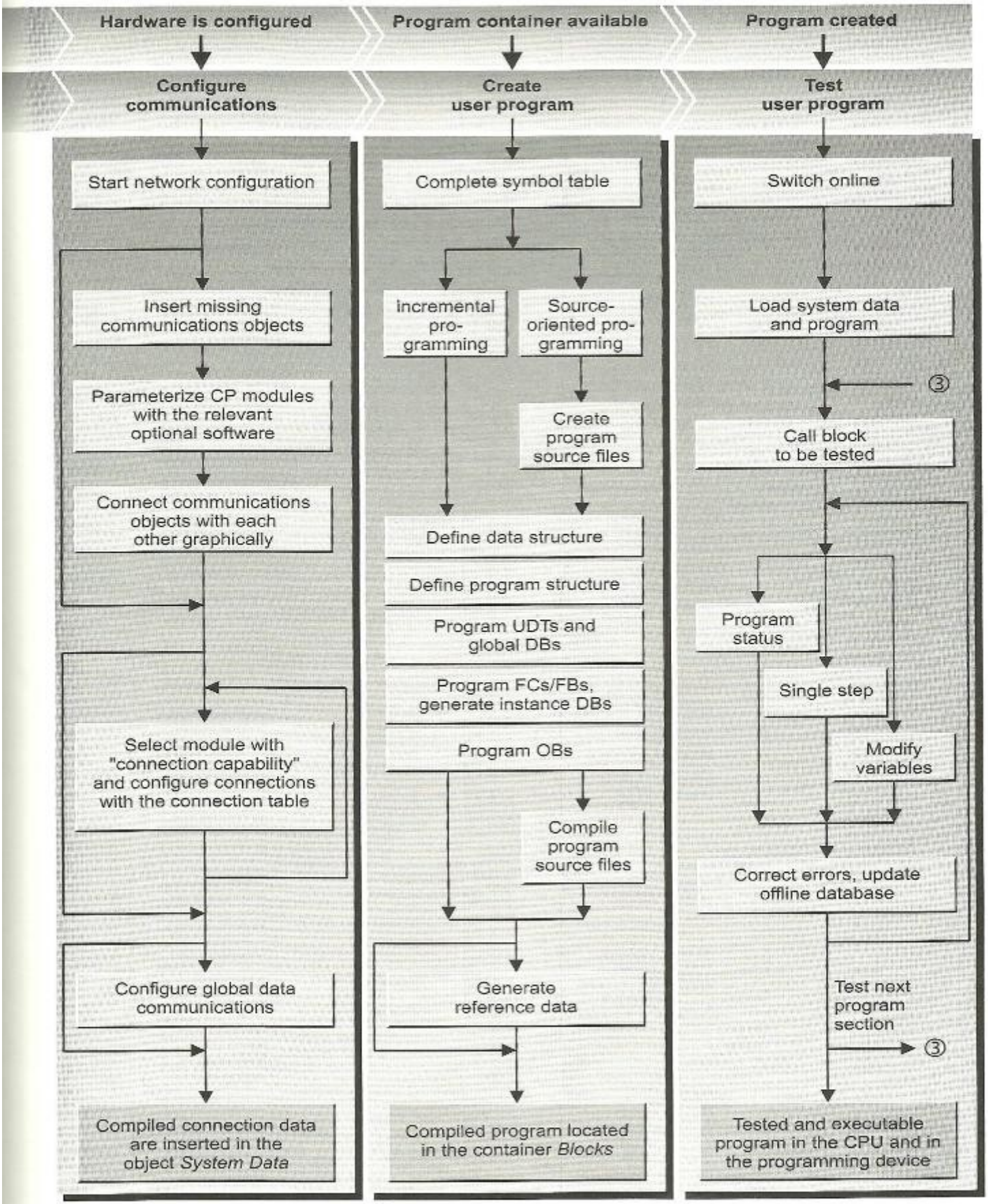
Start the SIMATIC Manager and set up a new project or open an existing project. All the data for an automation task are stored in the form of *objects* in a *project*. When you set up a project, you create *containers* for the accumulated data by setting up the required *stations* with at least the CPUs; then the containers for the user programs are also created. You can also create a *program container* direct in the project.

In the next steps, you configure the hardware and, if applicable, the communications connections. Following this, you create and test the program.

The order for creating the automation data is not fixed. Only the following general regulation applies: if you want to process objects (data), they must exist; if you want to insert objects, the relevant containers must be available.

You can interrupt processing in a project at any time and continue again from any location the next time you start the SIMATIC Manager.




Hardware is configured
Program container available
Program created
Configure communications
Create user program
Test user program
Start network configuration
Insert missing communications objects
Parameterize CP modules with the relevant optional software
Connect communications objects with each other graphically
Select module with "connection capability" and configure connections with the connection table
Configure global data communications
Compiled connection data are inserted in the object System Data
Complete symbol table
incremental pro-gramming
Source-oriented pro-gramming
Create program source files
Define data structure
Define program structure
Program UDTs and global DBs
Program FCs/FBs, generate instance DBs
Program OBs
Compile program source files
Generate reference data
Compiled program located in the container Blocks
Switch online
Load system data and program
③
Call block to be tested
Program status
Single step
Modify variables
Correct errors, update offline database
Test next program section
③
Tested and executable program in the CPU and in the programming device

Contents

Introduction

This portion of the book provides an overview of the SIMATIC S7-300/400.

The **S7-300/400 programmable controller** is of modular design. The modules with which it is configured can be central (in the vicinity of the CPU) or distributed without any special settings or parameter assignments having to be made. In SIMATIC S7 systems, distributed I/O is an integral part of the system. The CPU, with its various memory areas, forms the hardware basis for processing of the user programs. A load memory contains the complete user program: the parts of the program relevant to its execution at any given time are in a work memory whose short access times are the prerequisite for fast program processing.

STEP 7 is the programming software for S7-300/400 and the automation tool is the SIMATIC Manager. The SIMATIC Manager is an application for the Windows operating systems from Microsoft and contains all functions needed to set up a project. When necessary, the SIMATIC Manager starts additional tools, for example to configure stations, initialize modules, and to write and test programs.

You formulate your automation solution in the STEP 7 programming languages. The **SIMATIC S7 program** is structured, that is to say, it consists of blocks with defined functions that are composed of networks or rungs. Different priority classes allow a graduated interruptibility of the user program currently executing. STEP 7 works with variables of various data types starting with binary variables (data type BOOL) through digital variables (e.g. data type INT or REAL for computing tasks) up to complex data types such as arrays or structures (combinations of variables of different types to form a single variable).

The first chapter contains an overview of the hardware in an S7-300/400 programmable controller, and the second chapter contains an overview of the STEP 7 programming software. The basis for the description is the function scope for STEP 7 Version 5.4 SP3.

Chapter 3 "SIMATIC S7 Program" serves as an introduction to the most important elements of an S7 program and shows the programming of individual blocks in the programming languages LAD and FBD. The functions and operations of LAD and FBD are then described in the subsequent chapters of the book. All the descriptions are explained using brief examples.

1 **SIMATIC S7-300/400 Programmable Controller**
Structure of the programmable controller; distributed I/O; communications; module addresses; operand areas

2 **STEP 7 Programming Software**
SIMATIC Manager; processing a project; configuring a station; configuring a network; writing programs (symbol table, program editor); switching online; testing programs

3 **SIMATIC S7 Program**
Program processing with priority classes; program blocks; addressing variables; programming blocks with LAD and FBD; variables and constants; data types (overview)

1 SIMATIC S7-300/400 Programmable Controller

1.1 Structure of the Programmable Controller

1.1.1 Components

The SIMATIC S7-300/400 is a modular programmable controller comprising the following components:

- Racks
  Accommodate the modules and connect them to each other
- Power supply (PS);
  Provides the internal supply voltages
- Central processing unit (CPU)
  Stores and processes the user program
- Interface modules (IMs);
  Connect the racks to one another
- Signal modules (SMs);
  Adapt the signals from the system to the internal signal level or control actuators via digital and analog signals
- Function modules (FMs);
  Execute complex or time-critical processes independently of the CPU
- Communications processors (CPs)
  Establish the connection to subsidiary networks (subnets)
- Subnets
  Connect programmable controllers to each other or to other devices

A programmable controller (or station) may consist of several racks, which are linked to one another via bus cables. The power supply, CPU and I/O modules (SMs, FMs and CPs) are plugged into the central rack. If there is not enough room in the central rack for the I/O modules or if you want some or all I/O modules to be separate from the central rack, expansion racks are available which are connected to the central rack via interface modules (Figure 1.1).

It is also possible to connect distributed I/O to a station (see Chapter 1.2.1 "PROFIBUS DP").

The racks connect the modules with two buses: the I/O bus (or P bus) and the communication bus (or K bus). The I/O bus is designed for high-speed exchange of input and output signals, the communication bus for the exchange of large amounts of data. The communication bus connects the CPU and the programming device interface (MPI) with function modules and communications processors.

1.1.2 S7-300 Station

Centralized configuration

In an S7-300 controller, as many as 8 I/O modules can be plugged into the central rack. Should this single-tier configuration prove insufficient, you have two options for controllers equipped with a CPU 313 or higher:

- A two-tier configuration (with IM 365 up to 1 meter between racks) or
- A configuration of up to four tiers (with IM 360 and IM 361 up to 10 meters between racks)

You can operate a maximum of 8 modules in a rack. The number of modules may be limited by the maximum permissible current per rack, which is 1.2 A.

The modules are linked to one another via a backplane bus, which combines the functions of the P and K buses.

Local bus segment

A special feature regarding configuration is the use of the FM 356 application module. An FM 356 is able to "split" a module's backplane bus and to take over control of the remaining modules in the split-off "local bus segment" itself. The limitations mentioned above regarding the

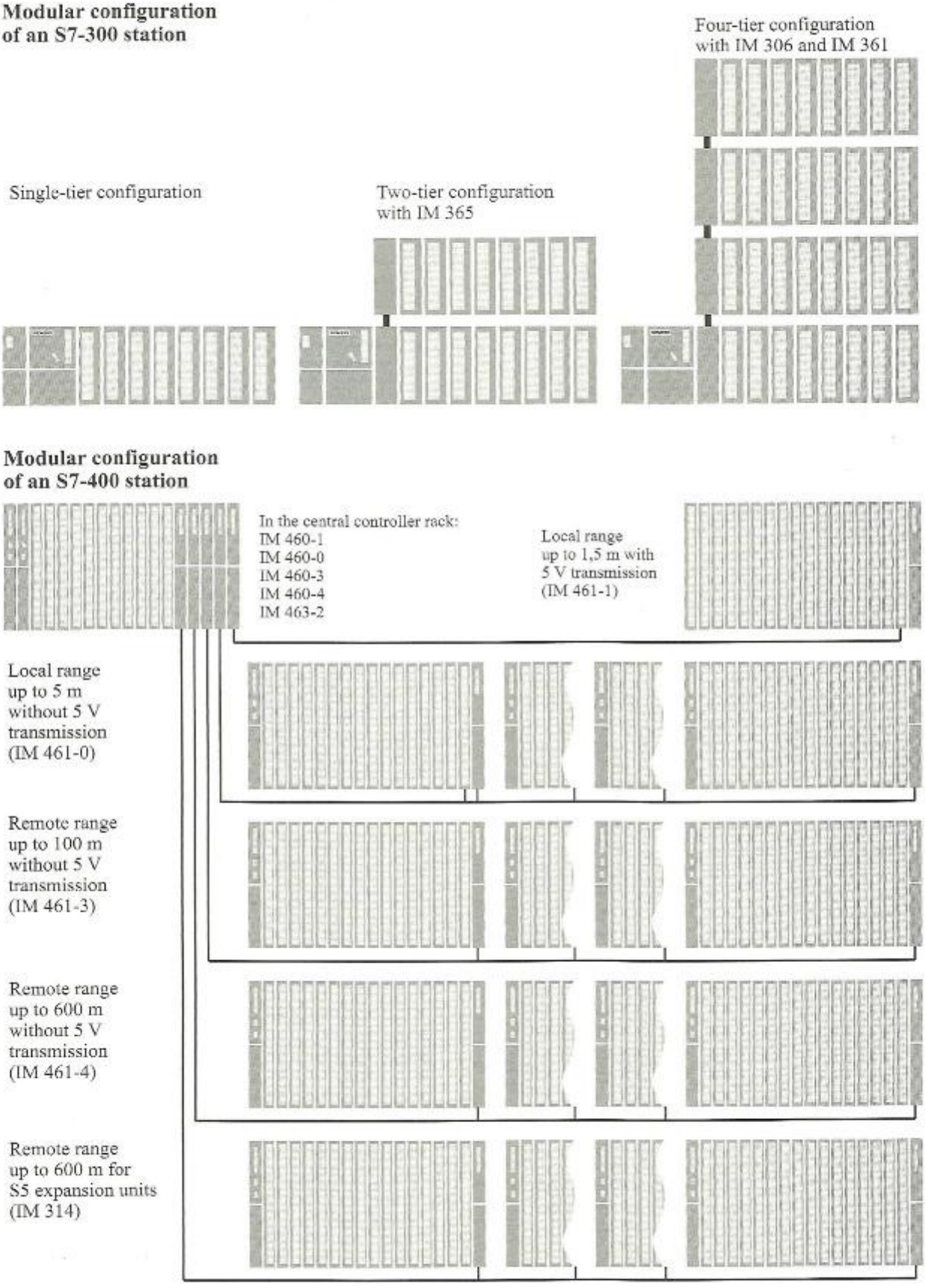


Figure 1.1 Hardware Configuration for S7-300/400

number of modules and the power consumption also apply in this case.

Standard CPUs

The standard CPUs are available in versions that differ with regard to memory capacity and processing speed. They 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. A micro memory card (MMC) is required for operating the standard CPUs – as with all innovated S7-300-CPUs. This medium opens up new application possibilities compared to the previously used memory card (see Chapter 1.1.6 "CPU Memory Areas").

The now discontinued CPU 318 can be replaced by the CPUs 317 and 319.

Compact CPUs

The 3xxC CPUs permit construction of compact mini programmable controllers. Depending on the version, they already contain:

- Integral I/Os
  Digital and analog inputs/outputs
- Integral technology functions
  Counting, measurement, control, positioning
- Integral communications interfaces
  PROFIBUS DP master or slave, point-to-point coupling (PtP)

The technological functions are system blocks which use the onboard I/O of the CPU.

Technology CPUs

The CPUs 3xxT combine open-loop control functions with simple motion control functions. The open-loop control component is designed as in a standard CPU. It is configured, parameterized and programmed using STEP 7. The technology objects and the motion control component require the optional S7-Technology package that is integrated in the SIMATIC Manager after installation.

The Technology CPUs have a PROFIBUS DP interface that allows operation as DP master or DP slave. The CPUs are used for cross-sector automation tasks in series mechanical equipment manufacture, special mechanical equipment manufacture, and plant building.

Failsafe CPUs

The CPUs 3xxF 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 (see "S7 Distributed Safety" under 1.1.5 "Safety-related SIMATIC"). Standard modules for normal applications can be used parallel to safety-related operation..

SIPLUS S7-300

The SIPLUS product family offers CPUs and modules based on the S7-300 that can be used in harsh environments. With horizontal installation, you have an extended temperature range of –25°C to +60°C. They have increased immunity to vibration and shock, and they meet the increased requirements for humidity, condensation and freezing. Please note the technical data for the module concerned. Selected types of the SIPLUS product range are available on request for use on rolling stock in accordance with EN 50155 "Railway applications – Electronic equipment used on rolling stock".

1.1.3 S7-400 Station

Centralized configuration

The controller rack for the S7-400 is available in the UR1 (18 slots), UR2 (9 slots) and CR3 (4 slots) versions. UR1 and UR2 can also be used as expansion racks. The power supply and the CPU also occupy slots in the racks, possibly even two or more per module. If necessary, the number of slots available can be increased using expansion racks: UR1 and ER1 have 18 slots each, UR2 and ER2 have 9 each.

The IM 460-1 and IM 461-1 interface modules make it possible to have one expansion rack per

interface up to 1.5 meters from the central rack, including the 5 V supply voltage. In addition, as many as four expansion racks can be operated up to 5 meters away using IM 460-0 and IM 461-0 interface modules. And finally, IM 460-3 and IM 461-3 or IM 460-4 and 461-4 interface modules can be used to operate as many as four expansion racks at a distance of up to 100 or 600 meters away.

A maximum of 21 expansion racks can be connected to a central rack. To distinguish between racks, you set the number of the rack on the coding switch of the receiving IM.

The backplane bus consists of a parallel P bus and a serial K bus. Expansion racks ER1 and ER2 are designed for "simple" signal modules which generate no hardware interrupts, do not have to be supplied with 24 V voltage via the P bus, require no back-up voltage, and have no K bus connection. The K bus is in racks UR1, UR2 and CR2 either when these racks are used as central racks or expansion racks with the numbers 1 to 6.

Segmented rack

A special feature is the segmented rack CR2. The rack can accommodate two CPUs with a shared power supply while keeping them functionally separate. The two CPUs can exchange data with one another via the K bus, but have completely separate P buses for their own signal modules.

Multiprocessor mode

In an S7-400, as many as four specially designed CPUs in a suitable rack can take part in multiprocessor mode. Each module in this station is assigned to only one CPU, both with its address and its interrupts. See Chapters 20.3.6 "Multiprocessing Mode" and 21.7 "Multiprocessor Interrupt" for more details.

Connecting SIMATIC S5 modules

The IM 463-2 interface module allows you to connect S5 expansion units (EG 183U, EG 185U, EG 186U as well as ER 701-2 and ER 701-3) to an S7-400, and also allows centralized expansion of the expansion units. An IM 314 in the S5 expansion unit handles the link. You can operate all analog and digital modules allowed in these expansion units. An S7-400 can accommodate as many as four IM 463-2 interface modules; as many as four S5 expansion units can be connected in a distributed configuration to each of an IM 463-2's two interfaces.

1.1.4 Fault-tolerant SIMATIC

Two designs of SIMATIC S7 fault-tolerant automation systems are available for applications with high fault tolerance demands for machines and processes: software redundancy and S7-400H/FH.

Software redundancy

Using SIMATIC S7-300/400 standard components, you can establish a software-based redundant system with a master station controlling the process and a standby station assuming control in the event of the master failing.

Fault tolerance through software redundancy is suitable for slow processes because transfer to the standby station can require several seconds depending on the configuration of the programmable controllers. The process signals are "frozen" during this time. The standby station then continues operation with the data last valid in the master station.

Redundancy of the input/output modules is implemented with distributed I/O (ET200M with IM 153-3 interface module for redundant PROFIBUS DP). The optional "Software Redundancy" software is available for configuring.

Fault-tolerant SIMATIC S7-400H

The SIMATIC S7-400H is a fault-tolerant programmable controller with redundant configuration comprising two central racks, each with an H CPU and a synchronization module for data comparison via fiber optic cable. Both controllers operate in "hot standby" mode; in the event of a fault, the intact controller assumes operation alone via automatic bumpless transfer. The UR2-H mounting rack with two times nine slots makes it possible to establish a fault-tolerant system in a single mounting rack.

The I/O can have normal availability (single-channel, single-sided configuration) or enhanced availability (single-channel switched configuration with ET200M). Communication is carried out over a simple or a redundant bus.

The user program is the same as that for a non-redundant controller; the redundancy function is handled exclusively by the hardware and is invisible to the user. The software package required for configuration is included in STEP 7 V5.3 and later. The standard library *Redundant IO* already supplied contains blocks for supporting the redundant I/O.

1.1.5 Safety-related SIMATIC

Failsafe automation systems control processes in which the safe state can be achieved by direct switching off. They are used in plants with increased safety requirements. The safety requirements achievable with SIMATIC S7 are: Safety Integrity Level SIL 1 to SIL 3 according to IEC 61508, Requirement Category AK 1 to AK 6 according to DIN V 19250 (DIN V VDE 0801) and Category 1 to 4 according to EN 954-1.

The safety functions are located as appropriate in the safety-related user program in a correspondingly designed CPU (F-CPU) and in the failsafe inputs and outputs (F-modules and F-submodules, see "Failsafe I/O").

The safety technology with F-CPU and F-modules (for the safety-related plant components) can be integrated in an S7-300/400 PLC in addition to the standard applications.

Safety-related communication using PROFIBUS DP applies the specially developed PROFIBUS profile PROFIsafe. This permits transmission of the user data of safety functions within the standard data telegram.

SIMATIC S7 provides two systems for implementation of "Safety Integrated": S7 Distributed Safety and S7 F/FH Systems.

S7 Distributed Safety

S7 Distributed Safety is a failsafe automation system mainly for applications with machine controls (for protection of machines and personnel) and in the process industry.

F-CPUs currently available are the CPUs 315F-2DP and 317F-2DP for S7-300, the CPU 416F-2 for S7-400, and the basic module IM151-F/CPU for ET200S. The F-modules and F-submodules are connected to S7-400 using PROFIBUS DP with the safety-relevant profile PROFISafe. Use of F-modules in the controller rack is additionally possible with S7-300.

STEP 7 with the optional S7 Distributed Safety package is necessary for configuration and programming of the failsafe system. The safety-relevant section of the program is programmed using F-LAD or F-FBD with a limited operation set and fewer data types compared to the basic languages. If a fault is detected in the safety program, the F-CPU enters the operating state STOP.

The option package also contains a block library for the safety program with failsafe blocks and templates.

S7 F/FH Systems

S7 F/FH Systems is a failsafe automation system based on S7-400 with main applications in the process industry.

S7 F/FH Systems is based on the S7-400 automation system. The F-modules and F-submodules are connected to S7-400 using PROFIBUS DP and the safety-relevant profile PROFISafe.

With S7-400F, a failsafe user program can be incorporated into the standard user program. In addition to failsafety, the S7-400FH also provides increased availability. If a detected fault results in a STOP of the master CPU, a switch is made without feedback to the CPU running in hot standby mode (see 1.1.4 "Fault-tolerant SIMATIC").

Configuration is carried out using the standard applications of STEP 7, V5.1 and later. The option package "S7 F Systems" is required for parameterization of the failsafe signal modules and for programming of the safety-relevant program components, plus the option package "CFC", V5.0 SP3 and later, the option package "S7-SCL" V5.0 and later and – for the fault-tolerant functions – the option package "S7 H Systems" V5.1 and later.

Special function blocks from the supplied F-library can be called and interconnected using

CFC (Continuous Function Chart). In addition to functions for programming safety functions, they also contain functions for error detection and response. In the event of faults and failures, this guarantees that the failsafe system is held in a safe state or is transferred to a safe state. If a fault is detected in the safety program, the failsafe component of the plant is switched off, whereas the remaining part continues to operate.

An F-runtime license must be present on each CPU to permit operation of an S7-400F/FH. Failsafe signal modules (F-modules) are required for safety operation, and are operated in the ET200M distributed I/O station.

Failsafe I/O

The failsafe signal modules (F-modules) are required for safety operation in the failsafe systems. Failsafety is achieved with the integral safety functions and appropriate wiring of the sensors and actuators.

The F-modules can also be used in standard applications with increased diagnostics requirements. The F-modules can be operated in redundant mode to increase the availability both in standard and safety operation with S7 F/FH systems. The failsafe I/O is available in various versions:

- The failsafe signal modules of S7-300 design are used in the ET200M distributed I/O station or centrally together with the CPU 315F-2DP.
- The failsafe power and electronics modules are used in the ET200S distributed I/O station, for which the IM 151-F/CPU basic module is also available as a failsafe CPU.
- Failsafe PROFIBUS DP standard slaves can also be used with S7 Distributed Safety.

1.1.6 CPU Memory Areas

Figure 1.2 shows the memory areas in the programming device, the CPU and the signal modules which are important for your program.

The programming device contains the *offline data*. These consist of the user program (program code and user data), the system data (e.g. hardware, network and interconnection configurations), and further project-specific data such as symbol tables and comments.

The *online data* consist of the user program and the system data on the CPU, and are accommodated in two areas, namely load memory and work memory. Also present in addition are the system memory and possibly a backup memory.

The I/O modules contain memories for the signal state of the inputs and outputs.

The CPUs have a slot for a plug-in *memory submodule*. The load memory, or parts thereof, is located here (see "Physical design of CPU memory" later). The memory submodule is designed as a memory card (S7-400 CPUs) or as a micro memory card (S7-300 CPUs and ET200 CPUs derived from these). The firmware of the CPU operating system can also be updated using the memory submodule.

Memory card

The memory module for the S7-400 CPUs is the memory card (MC). There are two types of memory card: RAM cards and flash EPROM cards.

If you want to expand load memory only, use a RAM card. A RAM card allows you to modify the entire user program online. This is necessary, for example, when testing and starting up larger programs. RAM memory cards lose their contents when unplugged.

If you want to protect your user program, including configuration data and module parameters, against power failure following testing and starting up even without a backup battery, use a flash EPROM card. In this case, load the entire program offline onto the flash EPROM card with the card plugged into the programming device. With the relevant CPUs, you can also load the program online with the memory card plugged into the CPU.

Micro memory card

The memory submodule for the newer S7-300 CPUs is a micro memory card (MMC). The data on the MMC are saved non-volatile, but can be read, written and deleted as with a RAM. This response permits data backup without a battery.

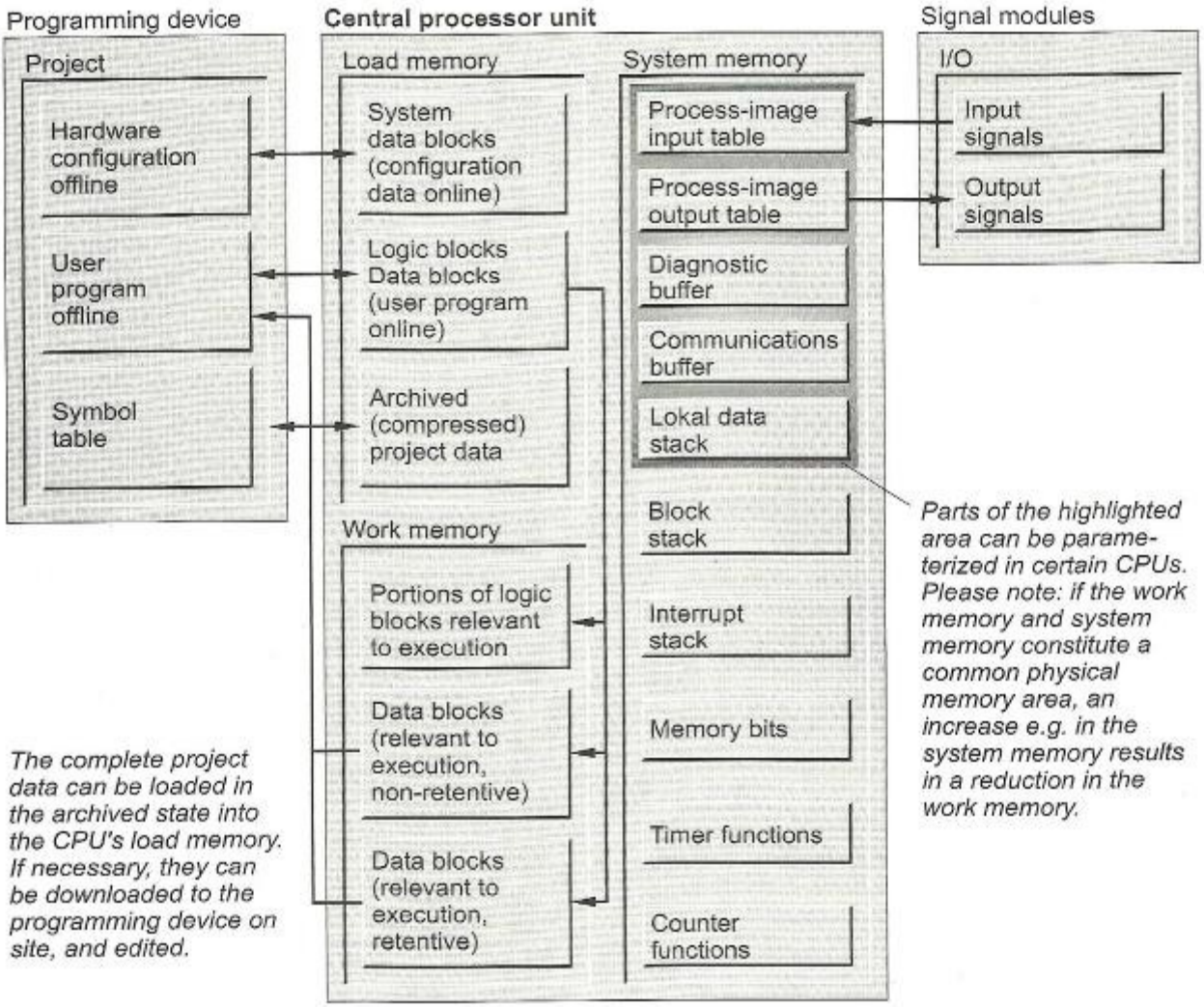


Figure 1.2 CPU Memory Areas

The complete load memory is present on the MMC, meaning that an MMC is always required for operation. The MMC can be used as a portable memory medium for user programs or firmware updates. Using special system functions you can read or write data blocks on the MMC from the user program, for example to read recipes from the MMC or to create a measured-value archive on the MMC and to provide it with data.

Load memory

The entire user program, including configuration data (system data), is in the load memory. The user program is always initially transferred from the programming device to the load memory, and from there to the work memory. The program in the load memory is not processed as the control program.

If a CPU does not have a micro memory card, the load memory is designed as a memory integrated in the CPU or as a plug-in memory card. It can be designed as a RAM or ROM.

If load memory consists of an integrated RAM or a RAM memory card, a backup battery is required in order to keep the user program retentive. Where load memory is implemented as integrated EEPROM, as a plug-in flash EPROM memory card or as a micro memory card, the CPU can be operated without battery backup.

From STEP 7 V5.1 onwards, and with appropriately designed CPUs, you can save the com-

plete project data as a compressed archive file in the load memory (see Chapter 2.2.2 "Managing, Reorganizing and Archiving").

Work memory

Work memory is designed in the form of high-speed RAM fully integrated in the CPU. The operating system of the CPU copies the program code "relevant to execution" and the user data into the work memory. "Relevant" is a characteristic of the existing objects and does not mean that a particular code block will necessarily be called and executed. The "actual" control program is executed in the work memory.

Depending on the product, the work memory can be designed either as a correlated area or divided according to program and data memories, where the latter can also be divided into retentive and non-retentive memories.

When uploading the user program into the programming device, the blocks are fetched from the load memory, supplemented by the actual values of the data operands from the work memory (further information can be found in Sections 2.6.4 "Loading the User Program into the CPU" and 2.6.5 "Block Handling").

System memory

System memory contains the addresses (variables) that you access in your program. The addresses are combined into areas (address areas) containing a CPU-specific number of addresses. Addresses may be, for example, inputs used to scan the signal states of momentary-contact switches and limit switches, and outputs that you can use to control contactors and lamps.

The system memory on a CPU contains the following address areas:

- Inputs (I)
  Inputs are an image ("process image") of the digital input modules.
- Outputs (Q)
  Outputs are an image ("process image") of the digital output modules.
- Bit memories (M)
  Stores of information accessible throughout the whole program from any point.
- Timers (T)
  Timers are locations used to implement waiting and monitoring times.
- Counters (Z)
  Counters are software-level locations, which can be used for up and down counting.
- Temporary local data (L)
  Locations used as dynamic intermediate buffers during block processing. The temporary local data are located in the L stack, which the CPU occupies dynamically during program execution.

The letters enclosed in parentheses represent the abbreviations to be used for the different addresses when writing programs. You may also assign a symbol to each variable and then use the symbol in place of the address identifier.

The system memory also contains buffers for communication jobs and system messages (diagnostics buffer). The size of these data buffers, as well as the size of the process image and the L stack, are parameterizable on certain CPUs.

Physical design of CPU memory

The physical design of the load memory is different for the various types of CPU (Figure 1.3).

The CPUs with micro memory card do not have an integrated load memory. A micro memory card must always be inserted to permit operation. The load memory on the micro memory card reacts as if it contains RAM and ROM components. The program is transmitted and tested completely normally in the RAM, and then written into the ROM by means of a menu command following the test. Retentivity of the user program is achieved with the micro memory card.

With the CPU 317, part of the work memory can be used for retentive data blocks. The rest of the work memory (complete memory less retentive data quantity) can be used for the program code and the non-retentive data.

The integrated RAM load memory with the S7-400 CPUs is designed for small programs and for modification of individual blocks if the load memory is a flash EPROM memory card. If the complete control program is larger than the

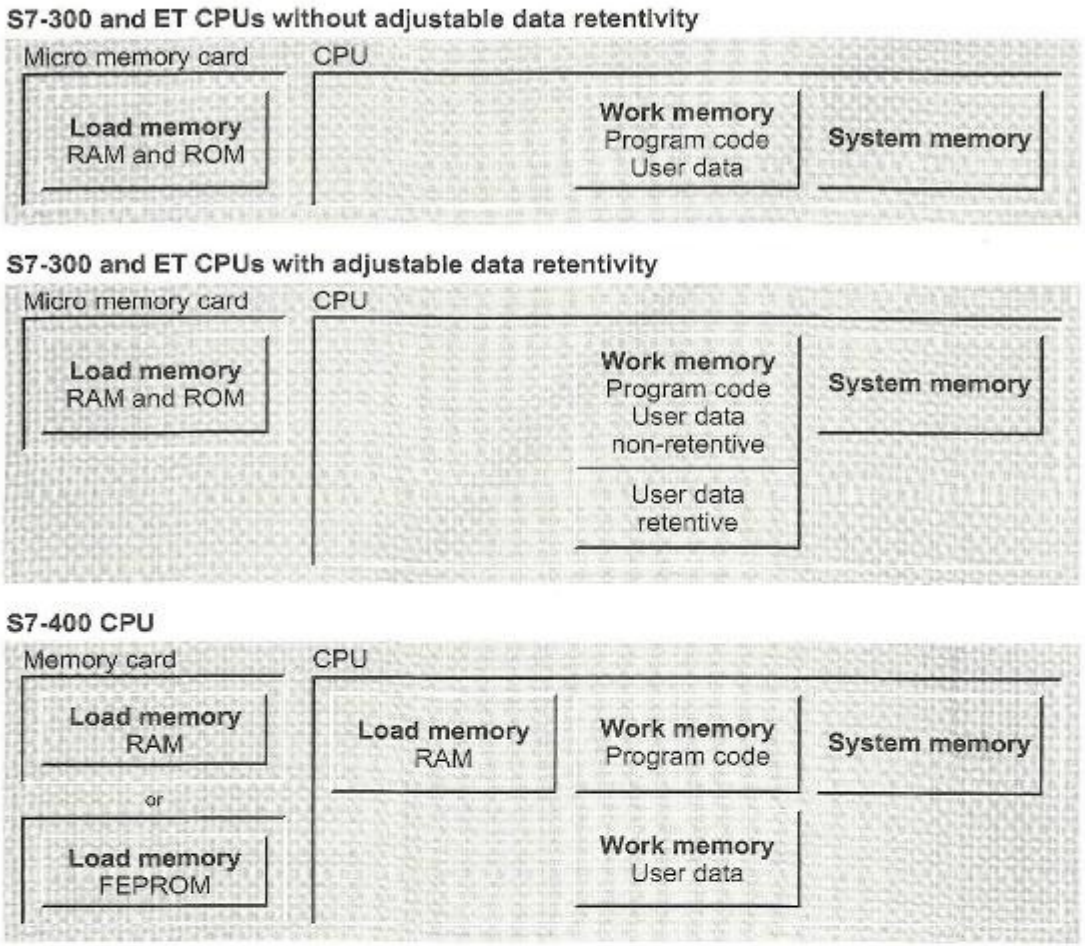


Figure 1.3 Physical Design of CPU Memory

integrated load memory, you require a RAM memory card for testing. The tested program is then transmitted by the programming device to a flash EPROM memory card which you insert into the CPU for operation.

The work memory of S7-400 CPUs is divided into two parts: one part saves the program code, the other the user data. The system and work memories in the S7-400 CPUs constitute one (physical) unit. If, for example, the size of the process image changes, this has effects on the size of the work memory.

1.2 Distributed I/O

Distributed I/O refers to modules connected via PROFIBUS DP or PROFINET IO. PROFIBUS DP uses the PROFIBUS subnetwork for data transmission, PROFINET IO the Industrial Ethernet subnetwork (for further information, see Chapter 1.3.2 "Subnets").

1.2.1 PROFIBUS DP

PROFIBUS DP provides a standardized interface for transferring predominantly binary process data between an "interface module" in the (central) programmable controller and the field devices. This "interface module" is called the DP master and the field devices are the DP slaves.

The DP master and all the slaves it controls form a DP master system. There can be up to 32 stations in one segment and up to 127 stations in the entire network. A DP master can control a number of DP slaves specific to itself. You

can also connect programming devices to the PROFIBUS DP network as well as, for example, devices for human machine interface, ET200 devices or SIMATIC S5 DP slaves.

DP master system

PROFIBUS DP is usually operated as a "mono master system", that is, one DP master controls several DP slaves. The DP master is the only master on the bus, with the exception of a temporarily available programming device (diagnostics and service device). The DP master and the DP slaves assigned to it form a DP master system (Figure 1.4).

You can also install several DP master systems on one PROFIBUS subnet (multi master system). However, this increases the response time in individual cases because when a DP master has initialized "its" DP slaves, the access rights fall to the next DP master that in turn initializes "its" DP slaves, etc.

You can reduce the response time if a DP master system contains only a few DP slaves. Since it is possible to operate several DP masters in one S7 station, you can distribute the DP slaves of a station over several DP master systems. In multiprocessor mode, every CPU has its own DP master systems.

DP master

The DP master is the active node on the PROFIBUS network. It exchanges cyclic data with "its" DP slaves. A DP master can be

- A CPU with integral DP master interface or plug-in interface submodule (e.g. CPU 315-2DP, CPU 417)
- An interface module in conjunction with a CPU (e.g. IM 467)
- A CP in conjunction with a CPU (e.g. CP 342-5, CP 443-5)

There are "Class 1 masters" for data exchange in process operation and "Class 2 masters" for service and diagnostics (e.g. a programming device).

DP slaves

The DP slaves are the passive nodes on PROFIBUS. In SIMATIC S7, a distinction is made between

- Compact DP slaves
  They behave like a single module towards the DP master
- Modular DP slaves
  They comprise several modules (submodules)
- Intelligent DP slaves
  They contain a control program that controls the lower-level (own) modules

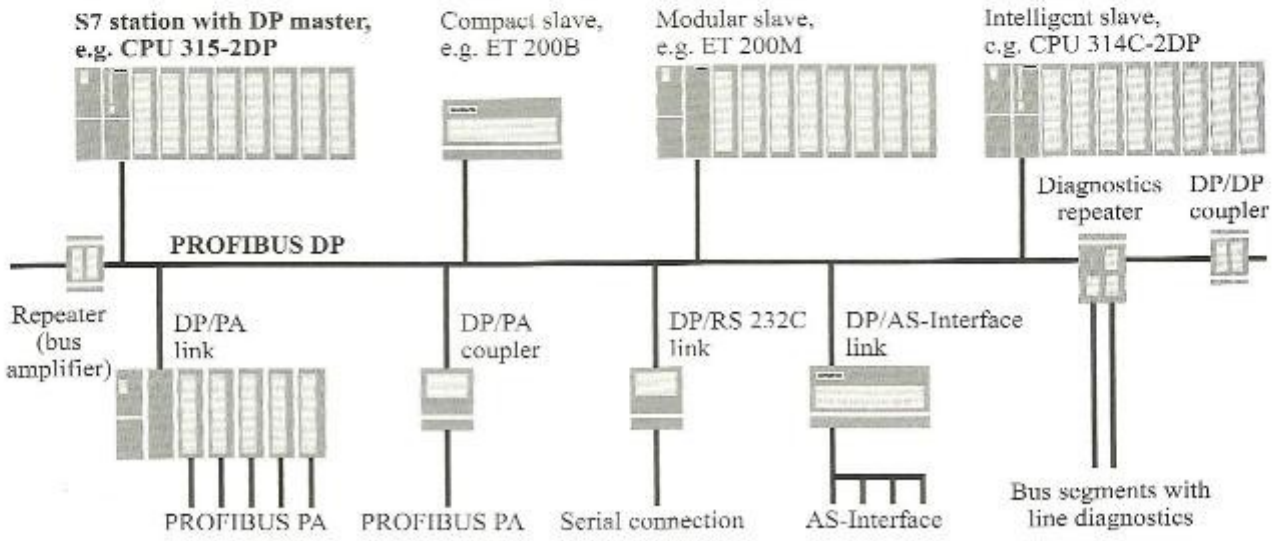


Figure 1.4 Components of a PROFIBUS DP Master System in an RS485 Segment.

Compact PROFIBUS DP slaves

Examples of DP slaves include the ET200B (version with digital input/output modules or analog input/output modules; degree of protection IP 20; max. data transfer rate 12 Mbit/s), the ET200C (rugged construction IP 66/67; different variants with digital inputs/outputs and analog inputs/outputs; data transfer rate 1.5 Mbit/s or 12 Mbit/s) and the ET200L-SC (discrete modularity with freely combinable digital input/output modules and analog input/output modules; degree of protection IP 20; data transfer rate 1.5 Mbit/s). The bus gateways such as DP/AS-i link behave like a compact slave on PROFIBUS DP.

Modular PROFIBUS DP slaves

The ET200M is an example of a modular DP slave. The design corresponds to an S7-300 station with DIN rail, power supply, IM 153 interface module instead of the CPU and with up to 8 signal modules (SMs) or function modules (FMs). The data transfer rate is 9.6 kbit/s to 12 Mbit/s.

The ET200M can also be designed with *active bus modules* if the DP master is an S7-400 station. This means that the S7-300 input/output modules can be plugged in and removed during operation under power. Operation of the remaining modules continues. The modules no longer have to be plugged in without gaps.

The ET200M can be used with the IM 153-3 interface module as a slave in a *redundant bus*. The IM 153-3 has two connections, one for the DP master in the master station and one for the DP master in the standby station.

Intelligent PROFIBUS DP slaves

Examples of intelligent DP slaves are CPUs with an integral DP (slave) interface, or an S7-300 station with the CP 342-5 communications processor. Equally, an ET200pro station with the IM 154-8 PN/DP CPU interface or an ET200S station with the IM 151-7 CPU interface module can be operated as intelligent DP slaves.

RS 485 repeater

The RS 485 repeater combines two bus segments in a PROFIBUS subnetwork. As a result, the number of stations and the expansion of the subnetwork can be increased.

The repeater provides signal regeneration and electrical isolation. It can be operated at transmission rates up to 12 Mbit/s, including 45.45 kbit/s for PROFIBUS PA.

The RS 485 is not configured; it need only be considered when calculating the bus parameters.

Diagnostics repeater

Using a diagnostics repeater, you can determine the topology and carry outline diagnostics in a PROFIBUS segment (RS 485 copper cable) during runtime. The diagnostics repeater provides signal regeneration and electrical isolation of the connected segments. The maximum segment length is 100 m in each case; the transmission rate can be between 9.6 kbit/s and 12 Mbit/s.

The diagnostics repeater has connections for three bus segments. The cable from the DP master is connected to the infeed terminals of bus segment DP1. The two other connections DP2 and DP3 contain the test circuits for determination of the topology and line diagnostics on the connected bus segments. Up to 9 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 diagnostics data to the DP master. These are the topology of the bus segment (stations and cable lengths), the contents of the segment diagnostics buffers (last ten events with fault information, location and cause) and the statistics data (statement on quality of bus system). In addition, the diagnostics repeater provides monitoring functions for isochrone mode.

The diagnostics data can be fetched and also graphically displayed by a programming device with STEP 7 V5.2 or later. Line diagnostics is triggered from the user program by the system function SFC 103 DP_TOPOL, and read using SFC 59 RD_REC or SFB 52 RDREC. In order to set the clock on the diagnostics repeater, you read the CPU time using the system function

SFC 1 READ_CLK and transmit it using SFC 58 WR_REC or SFB 53 WRREC.

The diagnostics repeater is configured and parameterized using STEP 7. A GSD file is available for operation on non-SIMATIC masters.

1.2.2 PROFINET IO

PROFINET IO offers a standardized interface for transmission of mainly binary process data between an "interface module" in the (central) programmable controller and the field devices using Industrial Ethernet. This "interface module" is referred to as the IO controller and the field devices as IO devices. The IO controller with all the IO devices controlled by it constitute a PROFINET IO system.

PROFINET IO system

A PROFINET IO system comprises the IO controller in the central station and the IO devices (field devices) assigned to it. The Industrial Ethernet subnet connecting them can also be shared by other stations and applications (Figure 1.5).

IO controller

The IO controller is the active station on the PROFINET. It exchanges data cyclically with "its" IO devices. An IO controller can be:

- A CPU with integral PROFINET interface (e.g. CPU 317-2PN/DP)
- A CP module in conjunction with a CPU (e.g. CP 343-1)

IO device

The IO devices are the passive stations on the PROFINET. In the case of SIMATIC S7, these can be the modular I/O devices such as ET200M, ET200S and ET200pro. The gateways PN/PN coupler, IE/PB link and IE/AS-i link are also IO devices.

IO supervisor

IO supervisors are devices for parameterization, startup, diagnostics, and human machine interfacing, e.g. programming devices or HMI devices.

1.2.3 Actuator/Sensor Interface

The Actuator/Sensor interface (AS-i) is a networking system for the lowest process level in automation plants in accordance with the international standard EN 50295. An AS-i master controls up to 62 AS-i slaves via a 2-wire AS-i cable that carries both the control signals and the supply voltage. (Figure 1.6).

One AS-i segment can be up to 100 m in length; in combination with repeaters and extension plugs, a maximum expansion of 600 m can be achieved.

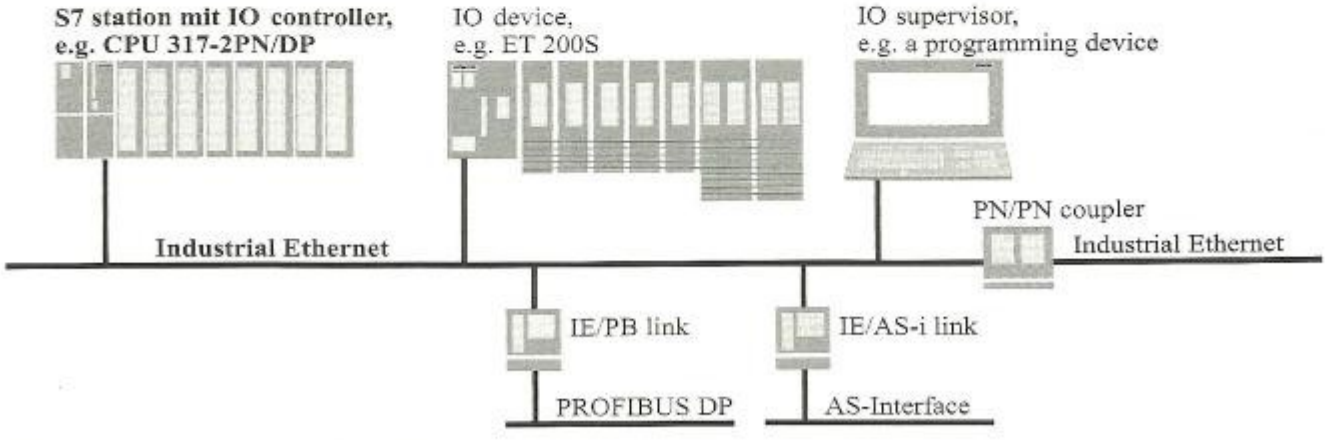


Figure 1.5 Components of a PROFINET IO system

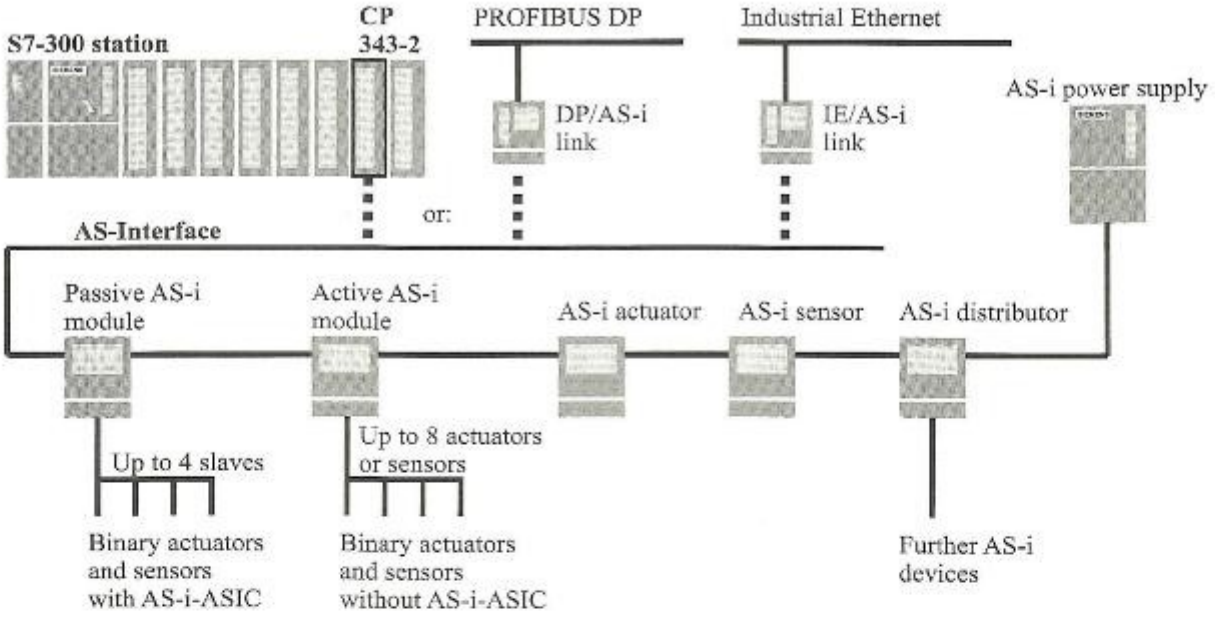


Figure 1.6 Connecting the AS-i bus system to SIMATIC S7

With "AS-Interface Safety at Work", you can connect safety sensors such as emergency-off switches, door contact switches, or safety light arrays direct to the AS-i network up to Category 4 in accordance with EN 954-1 or SIL3 in accordance with IEC 61508. This requires safe AS-i slaves for connecting the safety sensors and a safety monitor that combines the safe inputs with parameterizable logic and ensures safe shutdown.

AS-i master

Standard AS-i masters can control up to 31 standard AS-i slaves with a maximum cycle time of 5 ms. In the case of extended AS-i masters, the quantity structure increases to a maximum of 62 AS-i slaves with an extended address area with a maximum cycle time of 10 ms. Slaves with an extended address area occupy one address in pairs; if standard slaves are operated on an extended master, they each occupy one address.

The **AS-i master CP 343-2** is used in an S7-300 station or in an ET200M station. It supports the following AS-i slaves:

- Standard slaves
- Slaves with extended addressing mode (A/B slaves)
- Analog slaves to slave profile 7.3 or 7.4

In *standard mode*, the CP 343-2 behaves like an I/O module: It occupies 16 input bytes and 16 output bytes in the analog address area (from 128 upwards). Up to 31 standard slaves or 62 A/B slaves (slaves with extended address area) can be operated on the CP 343-2. The AS-i slaves are parameterized with default values stored in the CP.

In *extended mode*, the full range of functions in accordance with the AS-i master specification is available. If you use the FC block supplied, master calls can be made from the user program in addition to standard mode (transfer of parameters during operation, checking of the desired/actual configuration, test and diagnostics).

AS-i slaves

AS-i slaves can be bus-enabled sensors and actuators with AS-i ASICs, or they can be AS-i modules. You connect sensors and actuators with AS-i ASICs to a passive module. Conventional sensors and actuators can be connected to an active module.

AS-i slaves are available in the standard version with one standard slave occupying one of the maximum of 31 possible addresses. The user program handles the standard slaves like binary inputs and outputs.

AS-i slaves with extended addressing mode (A/B slaves) occupy an address in pairs so that up to 62 slaves can be operated on one master.

"A slaves" are treated like standard slaves, and "B slaves" are addressed via data records. AS-i A/B slaves can also acquire and transfer analog values.

1.2.4 Gateways

Gateways allow data exchange between devices on different subnets, and the forwarding of configuration and parameterization information beyond subnet boundaries (Figure 1.7).

Connecting two PROFIBUS subnets

The **DP/DP coupler** (Version 2) connects two PROFIBUS subnets to each other, allowing you to exchange data between the DP masters. Both subnets are 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 node 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 another. 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 and parameterized with STEP 7. A GSD file is available for operation on non-Siemens masters.

Connecting PROFIBUS DP to PROFIBUS PA

PROFIBUS PA (Process Automation) is a bus system for process engineering, both for intrinsically-safe areas (Ex area Zone 1), e.g. in the chemical industry, as well as for non-intrinsically-safe areas such as in the food and beverages industry.

The protocol for PROFIBUS PA is based on the standard EN 50170, Volume 2 (PROFIBUS DP), and the transmission technology is based on IEC 1158-2.

There are two methods of linking PROFIBUS DP and PROFIBUS PA:

- DP/PA coupler, when PROFIBUS DP can be operated at 45.45 kbit/s
- DP/PA link which converts the data transfer rates of PROFIBUS DP to the data transfer rate of PROFIBUS PA

The **DP/PA coupler** enables connection of PA field devices to PROFIBUS DP. On PROFIBUS DP, the DP/PA coupler is a DP slave that is operated at 45.45 kbit/s. Up to 31 PA field devices can be connected to one DP/PA coupler. The field devices form a PROFIBUS PA segment with a data transfer rate of 31.25 kbit/s. All PROFIBUS PA segments together form a shared PROFIBUS PA bus system.

The DP/PA coupler is available in two versions: a non-Ex version with up to 400 mA output current and an Ex version with up to 100 mA output current.

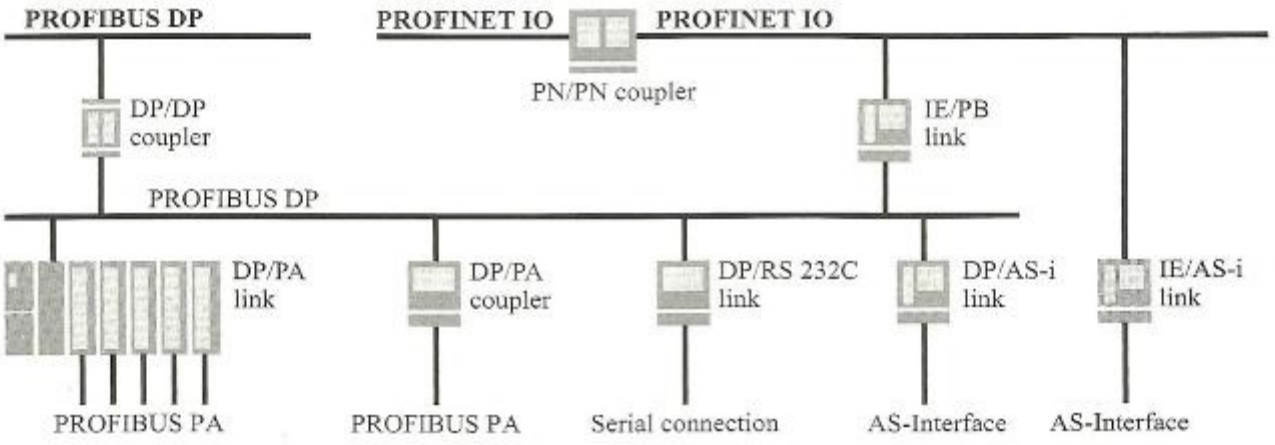


Figure 1.7 Gateways

The **DP/PA link** enables the connection of PA field devices to PROFIBUS DP with data transfer rates between 9.6 kbit/s and 12 Mbit/s. A DP/PA link comprises an IM 157 interface module and up to 5 DP/PA couplers that are connected to each other via SIMATIC S7 bus connectors. It maps the bus system consisting of all PROFIBUS PA segments to a PROFIBUS DP slave. A maximum of 31 PA field devices can be connected per DP/PA link.

SIMATIC PDM (Process Device Manager, previously SIPROM) is a cross-vendor tool for parameterization, startup and diagnostics of intelligent field devices with PROFIBUS PA or HART functionality. The DDL (Device Description Language) is available for parameterizing HART transducers (Highway Addressable Remote Transducers).

From STEP 7 V5.1 SP3, the control technology modules are parameterized with the Hardware Configuration; you must then no longer use SIMATIC PDM.

Connecting PROFIBUS DP to the AS-Interface

A **DP/AS-Interface link** enables the connection of PROFIBUS DP to the AS-Interface. On PROFIBUS DP, the link is a modular DP slave with a data transfer rate of up to 12 Mbit/s in degree of protection IP 20. On the AS-Interface, it is an AS-i master that controls the AS-i slaves. The link is available in the versions *DP/AS-i Link 20E* and *DP/AS-i Link Advanced*. The following AS-i slaves can be controlled:

- Standard slaves, AS-i analog slaves
- Slaves with extended addressing mode (A/B slaves)
- Slaves with data transfer mechanisms in accordance with AS-i specification V3.0 (DP/AS-i Link Advanced)

Connection of PROFIBUS DP to a serial interface

The **PROFIBUS DP/RS 232C link** is a converter between an RS 232C (V.24) interface and PROFIBUS DP. Devices with an RS 232C interface can be connected to PROFIBUS DP with the DP/RS 232C link. The DP/RS 232C link supports the procedures 3964R and free ASCII protocol.

The PROFIBUS DP/RS 232C link is connected to the device via a point-to-point connection. Conversion to the PROFIBUS DP protocol takes place in the PROFIBUS DP/RS 232C link. The data is transferred consistently in both directions. Up to 224 bytes of user data can be transferred per message frame.

The data transfer rate on PROFIBUS DP can be up to 12 Mbit/s; RS 232C can be operated at up to 38.4 kbit/s with no parity, even or odd parity, 8 data bits, and 1 stop bit.

Connecting two PROFINET subnets

With the **PN/PN Coupler**, you connect two Ethernet subnets to each other in order to exchange data between the IO controllers of both subnets. There is galvanic isolation between the subnets.

The PN/PN Coupler is a 120-mm-wide module that is installed on a DIN rail. The subnets are connected using RJ45 connectors. Two connections with internal switch function are available for each subnet.

From the viewpoint of the relevant IO controller, the PN/PN Coupler is an IO device in its own PROFINET IO system. Both IO devices are linked by a data transfer area with 256 input bytes and 256 output bytes, divisible into a maximum of 16 areas. Input areas in one subnet must correspond to output areas in another.

The PN/DP Coupler is configured and parameterized with STEP 7. A GSDML file is available for other configuring tools.

Connection of PROFINET IO to PROFIBUS DP

You can connect the Industrial Ethernet subnetworks and PROFIBUS using the **IE/PB link PNIO**. If you use PROFINET IO, the IE/PB link PNIO takes over the role of a proxy for the DP slaves on the PROFIBUS. An IO controller can access DP slaves just like IO devices using the IE/PB link. In standard mode, the IE/PB link is transparent for PG/OP communications and S7 routing between subnetworks.

The IE/PB link PNIO is a double-width module of S7-300 design. The IE/PB link is connected to Industrial Ethernet via an 8-contact RJ45 socket, and to PROFIBUS via a 9-contact SUB-D socket.

The IE/PB link is configured using STEP 7 as an IO device to which a DP master system is connected. When switching on, the subordinate DP slaves are also provided with the configuration data from the IO controller.

Please note that limitations exist on the PROFIBUS DP following an IE/PB link. For example, you cannot connect a DP/PA link, the DP segment does not have CIR capability, and isochrone mode cannot be configured.

Connecting PROFINET IO to the AS-Interface

An **IE/AS-i link** enables the connection of PROFINET IO to the AS-Interface. On PROFINET IO, the link is an IO device. On the AS-Interface, it is an AS-i master that controls the AS-i slaves. The IO controller can access the individual binary and analog values of the AS-i slaves directly.

Connection to PROFINET is made via two RJ45 connectors with internal switch function. The AS-Interface bus is connected to 4-pin plug-in screw-type contacts.

The link is available in the versions single master and double master (in accordance with AS-Interface specification V3.0) for the connection of up to 62 AS-i slaves in each case and integral analog value transfer. The following AS-i slaves can be controlled:

- Standard slaves, AS-i analog slaves
- Slaves with extended addressing mode (A/B slaves)
- Slaves with data transfer mechanisms in accordance with AS-i specification V3.0

The IE/AS-i link is configured and parameterized with STEP 7. A GSDML file is available for other configuring tools.

1.3 Communications

Communications – data exchange between programmable modules – is an integral component of SIMATIC S7. Almost all communications functions are handled via the operating system. You can exchange data without any additional hardware and with just one connecting cable between the two CPUs. If you use CP modules, you can achieve powerful network links and the facility of linking to non-Siemens systems.

SIMATIC NET is the umbrella term for SIMATIC communications. It represents information exchange between programmable controllers and between programmable controllers and human machine interface devices. There are various communications paths available depending on performance requirements.

1.3.1 Introduction

The most significant communications objects are initially SIMATIC stations or non-Siemens devices between which you want to exchange data. You require modules with communications capability here. With SIMATIC S7, all CPUs have an MPI interface over which they can handle communications.

In addition, there are communications processors (CPs) available that enable data exchange at higher throughput rates and with different protocols. You must link these modules via networks. A network is the hardware connection between communication nodes.

Data is exchanged via a "connection" in accordance with a specific execution plan ("communications service") which is based, among other things, on a specific coordination procedure ("protocol"). S7 connection is the standard between S7 modules with communications capability, for example.

Using an S7 connection, Figure 1.8 shows the objects involved in communication between two stations. The user program of the left-hand station contains the data to be transmitted in a data block (DB). The communications function in the example is a system function block (SFB). Assign the parameter RD with a pointer to the data to be sent, and trigger the transmission from the program. The communications

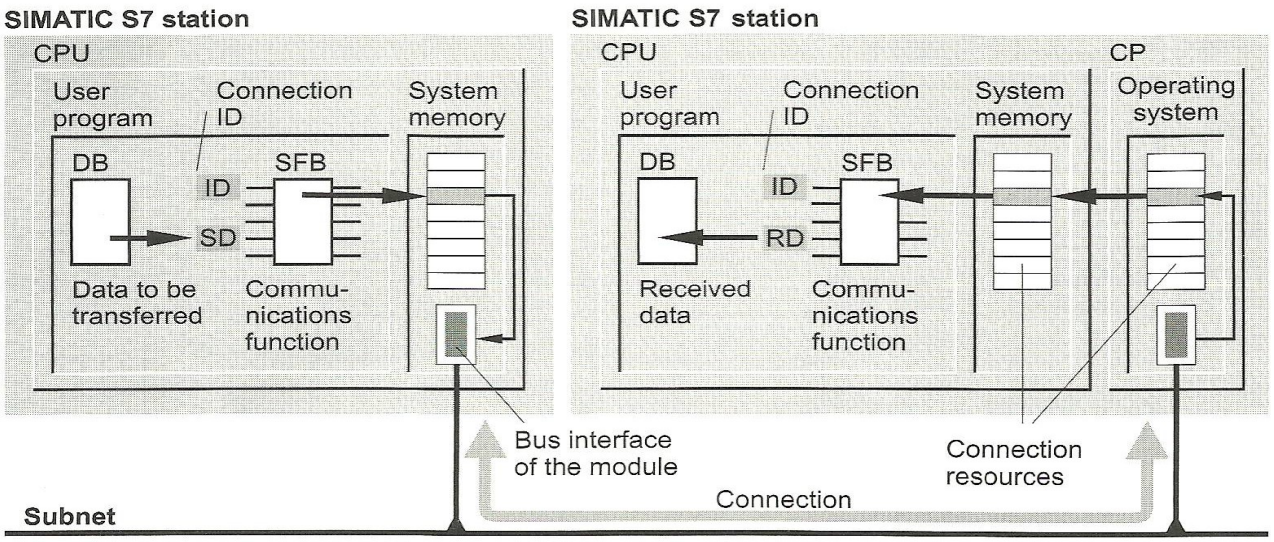


Figure 1.8 Data Exchange Between Two SIMATIC S7 Stations

function is additionally assigned a connection ID with which the used connection is specified. The connection occupies a connection resource in the CPU's system memory. The data are transmitted e.g. to a CP module in another station via the module's bus interface. Connection resources are used in both the CP module and CPU. Because of the connection ID (and the configured connection path) the communications function in the receiver station "recognizes" the data addressed to it, and writes them into the data block of the user program by means of the pointer in parameter RD.

Network

A network is a connection between several devices for the purpose of communication. It comprises one or more identical or different subnets linked together.

Subnet

In a subnet, all the communications nodes are linked via a hardware connection with uniform physical characteristics and transmission parameters, such as the data transfer rate, and they exchange data via a shared transmission procedure. SIMATIC recognizes MPI, PROFIBUS, Industrial Ethernet and point-to-point connection (PTP) as subnets.

Communications service

A communications service determines how the data are exchanged between communications nodes and how the data are to be handled. It is based on a protocol that describes, amongst other things, the coordination procedure between the communications nodes.

The services mostly used with SIMATIC are: PG communications, OP communications, S7 basic communications, S7 communications, global data communications, PtP communications, S5-compatible communications (SEND/RECEIVE interface).

Connection

A connection defines the communications relationships between two communications nodes. It is the logical assignment of two nodes for executing a specific communications service and also contains special properties such as the connection type (dynamic, static) and how it is established.

SIMATIC recognizes the following connection types: S7 connection, S7 connection (fault-tolerant), point-to-point connection, FMS and FDL connections, ISO transport connection, ISO-on-TCP and TCP connections, UDP connection and e-mail connection.

Communications functions

The communications functions are the user program's interface to the communications service. For SIMATIC S7-internal communications, the communications functions are integrated in the operating system of the CPU and they are called via system blocks. Loadable blocks are available for communication with non-Siemens devices via communications processors.

Overview of communications objects

Table 1.1 shows the relationships between subnets, modules with communications capability and communications services. In addition to the communications services shown, PG/OP communications is also possible via MPI, PROFIBUS and Industrial Ethernet subnets.

1.3.2 Subnets

Subnets are communications paths with the same physical characteristics and the same communications procedure. Subnets are the central objects for communication in the SIMATIC Manager.

The subnets differ in their performance capability:

- MPI
  Low-cost method of networking a few SIMATIC devices with small data volumes.
- PROFIBUS
  High-speed exchange of small and mid-range volumes of data, used primarily with distributed I/O.
- Industrial Ethernet
  Communications between computers and programmable controllers for high-speed exchange of large volumes of data, also used with distributed I/Os (PROFINET IO).
- Point-to-point (PTP)
  Serial link between two communications partners with special protocols.

From STEP 7 V5, you can use a programming device to reach SIMATIC S7 stations via subnets, for the purposes of, say, programming or parameterizing. The gateways between the subnets must be located in an S7 station with "routing capability".

MPI

Every CPU with SIMATIC S7 has an "interface with multipoint capability" (multipoint interface, or MPI). It enables establishment of subnets in which CPUs, human machine interface devices and programming devices can exchange data with each other. Data exchange is handled via a Siemens proprietary protocol.

The maximum number of nodes on the MPI network is 32. Each node has access to the bus for a specific length of time and may send data frames. After this time, it passes the access rights to the next node ("token passing" access procedure).

As transmission medium, MPI uses either a shielded twisted-pair cable or a glass or plastic fiber-optic cable. The maximum cable length in a bus segment with non-electrically-isolated interfaces is up to 50 m depending on the transmission rate, and up to 1000 m with electrically isolated interfaces. This can be increased by inserting RS485 repeaters (up to 1100 m) or optical link modules (up to > 100 km). The data transfer rate is usually 187.5 kbit/s.

Over an MPI subnet, you can exchange data between CPUs with global data communications, station-external S7 basic communications or S7 communications. No additional modules are required.

PROFIBUS

PROFIBUS stands for "Process Field Bus" and is a vendor-independent standard complying with IEC 61158/EN 50170 for universal automation (PROFIBUS DP and PROFIBUS FMS) and for process automation according to IEC 61158-2 (PROFIBUS PA).

The maximum number of nodes in a PROFIBUS network is 127, where the network is divided into segments with up to 32 nodes. A distinction is made between active and passive nodes. An active node receives access rights to the bus for a specific length of time and may send data frames. After this time, it passes the access rights to the next node ("token passing" access procedure). If passive nodes (slaves) are assigned to an active node (master), the master executes data exchange with the slaves assigned to it while it is in possession of the

Table 1.1 Communications Objects

Subnet	Modules	Communications Service, Connection	Configuring, Interface
MPI	All CPUs	Global data communications	GD table
		Station-internal S7 basic communications	SFC calls
		S7 communications	Connection table, FB/SFB calls
PROFI-BUS	CPUs with DP interface	PROFIBUS DP (DP master or DP slave)	Hardware configuration, SFB/SFC calls, inputs/outputs
		Station-internal S7 basic communications	SFC calls
	IM 467	PROFIBUS DP (DP master)	Hardware configuration, SFB/SFC calls, inputs/outputs
		Station-internal S7 basic communications	SFC calls
	CP 342-5 CP 443-5 Extended	CP 342-5: PROFIBUS DP V0 CP 433-5 Ext.: PROFIBUS DP V1 (DP master or DP slave)	Hardware configuration, SFB/SFC calls, inputs/outputs
		Station-internal S7 basic communications	SFC calls
		S7 communications	Connection table, FB/SFB calls
		S5-compatible communications	NCM, connection table, SEND/RECEIVE
	CP 343-5 CP 443-5 Basic	Station-internal S7 basic communications	SFC calls
		S7 communications	Connection table, FB/SFB calls
		S5-compatible communications	NCM, connection table, SEND/RECEIVE
		PROFIBUS FMS	NCM, connection table, FMS interface
Industrial Ethernet	CPUs with PN interface	PROFINET IO (IO controller)	Hardware configuration, SFB/SFC calls, inputs/outputs
		IE communications	FB calls
	CP 343-1 Lean CP 343-1 CP 443-1	S7 communications	Connection table, FB/SFB calls
		S5-compatible communications Transport protocols TCP/IP and UDP, also ISO with CP 443-1	NCM, connection table, SEND/RECEIVE
	CP 343-1 IT CP 443-1 Advanced CP 443-1 IT	S7 communications	Connection table, FB/SFB calls
		S5-compatible communications Transport protocols TCP/IP and UDP, also ISO with CP 443-1	NCM, connection table, SEND/RECEIVE
		IT communications (HTTP, FTP, E-mail)	NCM, connection table, SEND/RECEIVE
	CP 343-1 PN	S7 communications	Connection table, FB/SFB calls
		S5-compatible communications Transport protocols TCP and UDP	NCM, connection table, SEND/RECEIVE

NCM is the configuring software for CP modules (integrated in STEP 7 V5.2 and later)

access rights. A passive node does not receive access rights.

The PROFIBUS network can also be physically designed as an electrical network, optical network or wireless coupling with various transmission rates. The length of a segment depends on the transmission rate. The electrical network can be configured with a linear or tree topology. It uses a shielded, twisted two-wire cable (RS485 interface). The transmission rate can be adjusted in steps from 9.6 kbit/s to 12 Mbit/s (31.25 kbit/s with PROFIBUS PA).

The optical network uses either plastic, PCF or glass fiber-optic cables. It is suitable for large distances, provides electrical isolation, and is insensitive to electromagnetic interferences. The transmission rate can be adjusted in steps from 9.6 kbit/s to 12 Mbit/s. When using optical link modules (OLMs), designs are possible with linear, ring or star topologies. 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 structure with integral interface and optical bus terminal (OBT).

Using the PROFIBUS infrared link module (ILM), single or several PROFIBUS slaves or segments can be provided with a wireless connection to PROFIBUS slaves. The maximum transmission rate of 1.5 Mbit/s and the maximum range of 15 m means that communication is possible with moving parts.

You implement connection of distributed I/O via a PROFIBUS subnetwork; the relevant PROFIBUS DP communications service is implicit. You can use either CPUs with integral or plug-in DP master, or the relevant CPs. You can also operate station-internal S7 basic communications or S7 communications via this network.

You can transfer data with PROFIBUS FMS and PROFIBUS FDL using the relevant CPs. There are loadable blocks (FMS interface or SEND/RECEIVE interface) available as the interface to the user program).

Industrial Ethernet

Industrial Ethernet is the subnet for connecting computers and programmable controllers, with the focus on the industrial area, defined by the international standard IEEE 802.3/802.3u. The standard IEEE 802.11 a/b/g/h defines the connection to wireless local area networks (WLANs) and Industrial Wireless LANs (IWLANs).

The number of stations networked using Industrial Ethernet is unlimited; up to 1024 stations are permissible per segment. Before accessing, each node checks to see if another node is currently transmitting. If this is the case, the node waits for a random time before attempting another access (CSMA/CD access procedure). All nodes have equal access rights.

The physical connections on Industrial Ethernet consist of point-to-point connections between communication nodes. Each node is connected with precisely one partner. To enable several nodes to communicate with each other, they are connected to a "distributor" (switch or hub) that has several connections.

A *switch* is an active bus element that regenerates signals, prioritizes them, and distributes them only to those devices that are connected to it. A *hub* adjusts to the lowest data transfer rate at the connections, and forwards all signals unprioritized to all connected devices.

The network can be configured as a linear, star, tree or ring topology. The data transfer rates are 10 Mbit/s, 100 Mbit/s (Fast Ethernet) or 1000 Mbit/s (Gigabit Ethernet, not on PROFINET).

Industrial Ethernet can be physically designed as an electrical network, optical network 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).

You can exchange data with S7 and IE communications via Industrial Ethernet and utilize the S7 functions. With appropriately designed modules, you can also establish ISO transport connections, ISO-on-TCP connections, TCP, UDP and e-mail connections.

PROFINET

PROFINET is the open Industrial Ethernet standard of PROFIBUS International (PNO). PROFINET uses the Industrial Ethernet subnet as the physical medium for data transmission, taking into account the requirements of industrial automation. For example, PROFINET offers a real-time (RT) response for communications with field devices, and isochronous real-time (IRT) transmission for motion control applications. Compatibility with TCP/IP and the IT standards of Industrial Ethernet are retained.

Siemens applies PROFINET to two automation concepts:

- *Component Based Automation* (CBA) uses PROFINET for communication between control units as components in distributed systems. The configuration tool is SIMATIC iMap.
- *PROFINET IO* uses PROFINET to transmit data to and from field devices (distributed I/O). The configuration tool is SIMATIC STEP 7.

Point-to-point connection

A point-to-point connection (PTP) enables data exchange via a serial link. A point-to-point connection is handled by the SIMATIC Manager as a subnet and configured similarly.

The transmission medium is an electrical cable with interface-dependent assignment. RS 232C (V.24), 20 mA (TTY) and RS 422/485 are available as interfaces. The data transfer rate is in the range 300 bits/s to 19.2 kbit/s with a 20 mA interface or 76.8 kbit/s with RS 232C and RS 422/485. The cable length depends on the physical interface and the data transfer rate; it is 10 m with RS 232C, 1000 m with a 20 mA interface at 9.6 kbit/s and 1200 with RS 422/485 at 19.2 kbit/s.

3964 (R), RK 512, printer drivers and an ASCII driver are available as protocols (procedures), the latter enabling definition of a user-specific procedure.

AS-Interface

The AS-Interface (actuator/sensor interface, AS-i) networks the appropriately designed binary sensors and actuators in accordance with the AS-Interface specification IEC TG 178. The AS-Interface does not appear in the SIMATIC Manager as a subnet; only the AS-I master is configured with the hardware configuration or with the network configuration.

The transmission medium is an unshielded twisted-pair cable that supplies the actuators and sensors with both data and power (power supply required). Network range can be up to 600 m with repeaters and extension plugs. The data transfer rate is set at 167 kbit/s.

A master controls up to 62 slaves through cyclic scanning and so guarantees a defined response time.

1.3.3 Communications Services

Data exchange over the subnets is controlled by different communications services – depending on the connection selected. The services are provided by the CPU or CP modules. In addition to communications with field devices (PROFIBUS DP, PROFIBUS PA and PROFINET IO, see Chapter 1.2.1 "PROFIBUS DP" and 1.2.2 "PROFINET IO"), the services listed below are available depending on the module used.

PG communications

PG communications is used to exchange data between an engineering station and a SIMATIC station. It is used, for example, by a programming device in online mode to execute the functions "Monitor variables" or "Read diagnostics buffer" or to load user programs. The communications functions required for PG communications are integrated in the operating system of the SIMATIC modules. PG communications can be executed over the MPI, PROFIBUS and Industrial Ethernet subnets. By applying S7 routing, the PG communications can also be used beyond subnets.

OP communications

OP communications is used to exchange data between an operator station and a SIMATIC station. For example, it is used by an HMI device for operation and monitoring, or to read and write variables. The communications functions required for OP communications are integrated in the operating system of the SIMATIC modules. OP communications can be executed over the MPI, PROFIBUS and Industrial Ethernet subnets.

S7 basic communications

S7 basic communications is an event-controlled service for exchanging small volumes of data between a CPU and a module in the same SIMATIC station ("station-internal") or between a CPU and a module in a different SIMATIC station ("station-external"). The connections are established dynamically when required. The communications functions required for the S7 basic communications are integrated in the operating system of the CPU, e.g. you trigger the data transfer in the user program by means of system functions SFC. Station-internal S7 basic communications is executed over PROFIBUS, station-external over MPI.

S7 communications

S7 communications is an event-controlled service for exchanging larger volumes of data between CPU modules with control and monitoring functions. The connections are static, and are programmed using STEP 7. The communications functions required for S7 communications are either integrated in the operating system of the CPU (system function blocks SFB) or they are loadable function blocks (FB). S7 communications can be executed over the MPI, PROFIBUS and Industrial Ethernet subnets.

IE communication

With "Open communication via Industrial Ethernet" (IE communication for short), you transfer data between two devices connected to the Ethernet subnet. Communication can be implemented using the protocols TCP native in accordance with RFC 793, ISO-on-TCP in accordance with RFC 1006, or UDP in accordance with RFC 768. The communication functions are loadable function blocks (FBs) contained in the *Standard Library* of STEP 7 under *Communication Blocks*. The function blocks are called in the main program and they control connection buildup and cleardown as well as data transfer.

Global data communications

Global data communications enables exchange of small volumes of data between several CPUs without additional programming overhead in the user program. Transfer can be cyclic or event-driven. The communications functions required are integrated in the operating system of the CPU. Global data communications is only possible over the MPI bus or the K bus.

PtP communications

PtP communications (point-to-point connection) transfers data over a serial interface, e.g. between a SIMATIC station and a printer. The communications functions required are integrated in the operating system, e.g. as system function blocks SFB. Data exchange is possible using various transfer procedures.

S5-compatible communications

S5-compatible communications is an event-controlled service for data transfer between SIMATIC stations and non-SIMATIC stations. The connections are static, and are programmed using STEP 7. The communications functions are usually loadable functions FC with which you can control the transfer from the user program. Data are sent and received over the SEND/RECEIVE interface, and data can be fetched and written over the FETCH/WRITE interface (S7 is a passive partner). S5-compatible communications can be implemented with Industrial Ethernet using the TCP, ISO on TCP, ISO transport and UDP connections, and with PROFIBUS using FDL.

Standard communications

Standard communications uses standardized, vendor-independent protocols for data transfer.

PROFIBUS FMS (Fieldbus Message Specification) provides services for the program-based, device-independent transfer of structured variables (FMS variables) in accordance with EN 50170 Volume 2. Data exchange is carried out using static FMS connections over a PROFIBUS subnet. The communications functions are loadable function blocks FB with which you control the transfer from the user program.

Using an IT communications processor, a SIMATIC station is linked to the **IT communications**. Transfer over Industrial Ethernet comprises PG/OP/S7 communications and S5-compatible communications (SEND/RECEIVE) with the ISO, TCP/IP and UPD transport protocols. It is additionally possible to use SMTP (Simple Mail Transfer Protocol) for e-mails, HTTP (Hyper Text Transfer Protocol) for access using Web browsers, and FTP (File Transfer Protocol) for program-controlled data exchange with devices from different operating systems.

1.3.4 Connections

A connection is either dynamic or static depending on the communications service selected. Dynamic connections are not configured; their buildup or cleardown is event-driven ("Communications via non-configured connections"). There can only ever be one non-configured connection to a communications partner.

Static connections are configured in the connection table; they are built up at startup and remain throughout the entire program execution ("communications via configured connections"). Several connections can be established in parallel to one communications partner. You use a "Connection type" to select the desired communications service in the network configuration (see Chapter 2.4 "Configuring the Network").

You do not need to configure connections with the network configuration for global data communications and PROFIBUS DP or for S7 basic communications in the case of S7 functions. You define the communications partners for global data communications in the global data table; in the case of PROFIBUS DP and S7 basic communications, you define the partners via the node addresses.

Connection resources

Each connection requires connection resources on the participating communications partner for the end point of the connection or the transition point in a CP module. If, for example, S7 functions are executed via a bus interface of the CPU, a connection is assigned in the CPU; the same functions via the MPI interface of the CP occupy one connection in the CP and one connection in the CPU.

Each CPU has a specific number of possible connections. Limitations and rules exist regarding the use of connection resources. For example, not every connection resource can be used for every type of connection. One connection is reserved for a programming device and one connection for an OP (these cannot be used for any other purpose).

Connection resources are also required temporarily for the "non-configured connections" in S7 basic communications.

1.4 Module Addresses

1.4.1 Signal Path

When you wire your machine or plant, you determine which signals are connected where on the programmable controller (Figure 1.9).

An input signal, for example the signal from momentary-contact switch +HP01-S10, the one for "Switch motor on", is run to an input module, where it is connected to a specific terminal. This terminal has an "address" called the I/O address (for instance byte 5, bit 2).

Before every program execution start, the CPU then automatically copies the signal to the process input image, where it is then accessed as an "input" address (I 5.2, for example). The expression "I 5.2" is the absolute address.

You can now give this input a name by assigning an alphanumeric symbol corresponding to this input signal (such as "Switch motor on") to the absolute address in the symbol table. The

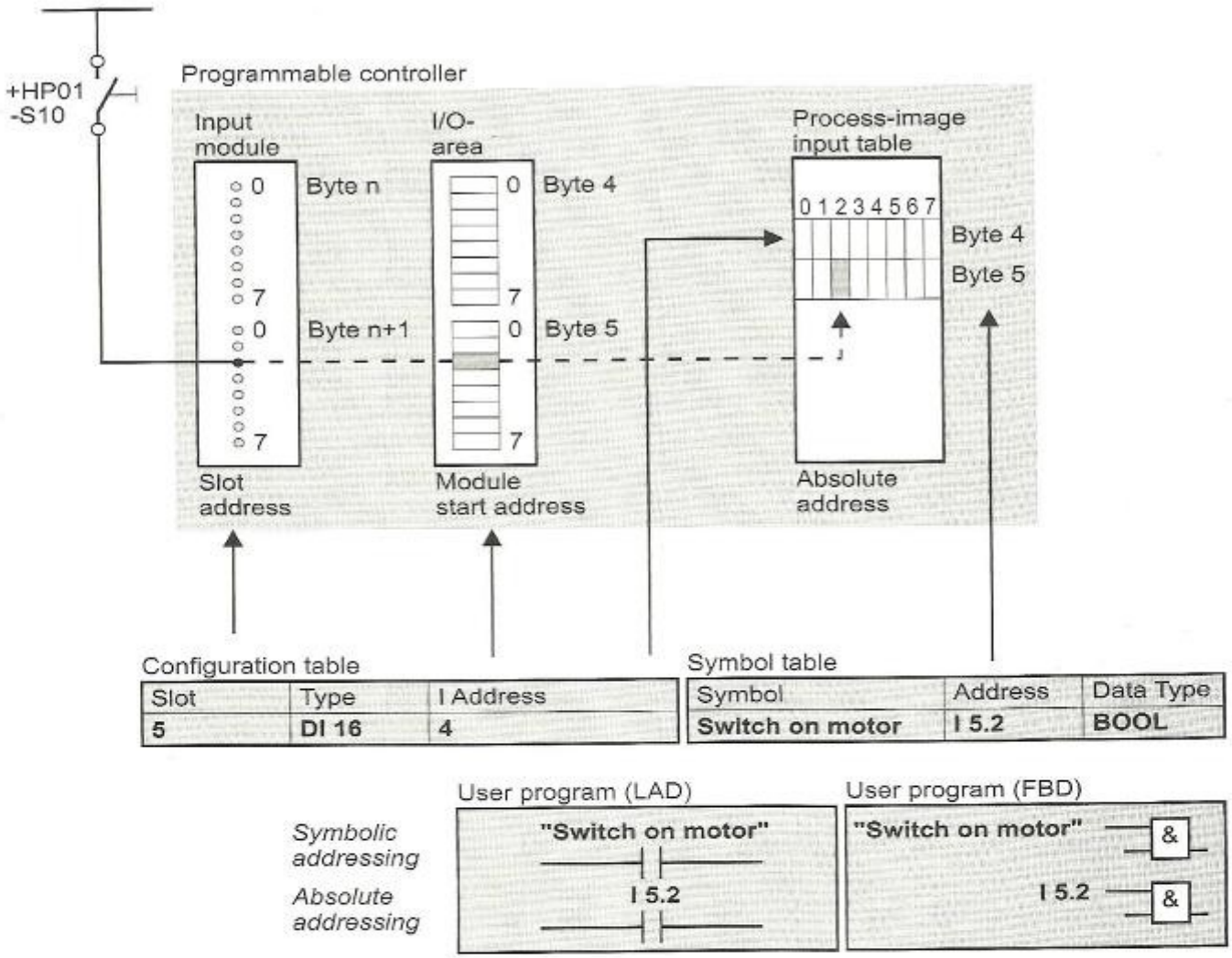


Figure 1.9
Correlation between Module Address, Absolute Address and Symbolic Address (Path of a Signal from Sensor to Scanning in the Program)

expression "Switch motor on" is the symbolic address.

1.4.2 Slot Address

Every slot has a fixed address in the programmable controller (an S7 station). This slot address consists of the number of the mounting rack and the number of the slot. A module is uniquely described using the slot address ("geographical address").

If the module contains interface cards, each of these cards is also assigned a submodule address. In this way, each binary and analog signal and each serial connection in the system has its own unique address.

Correspondingly, distributed I/O modules also have a "geographical address". In this case, the number of the DP master system or the PROFINET IO system and the station number replace the rack number.

You use STEP 7's "Hardware Configuration" tool to plan the hardware configuration of an S7 station as per the physical location of the modules. This tool also makes it possible to set the module start addresses and parameterize the modules (see Chapter 2.3 "Configuring Stations").

1.4.3 Logical Address

The logical address corresponds to the absolute address. It is also referred to as the user data address, for you 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 PI and peripheral outputs PQ). The range of logical addresses commences at zero and ends at an upper limit specific to the CPU.

With digital modules, the individual signals (the individual bits) are combined into bunches of 8, referred to as bytes. Modules exist with one, two or four bytes. These bytes have the relative addresses 0, 1, 2 and 3; addressing of the bytes commences at the modules start address. Example: with a digital module with four bytes and the start address 8, the individual bytes with addresses 8, 9, 10 and 11 are addressed. With analog modules, the individual analog signals (voltages, currents) are referred to as "channels", each of which requires two bytes. Depending on their design, analog modules exist with 2, 4, 8 or 16 channels corresponding to an address range of 4, 8, 16 or 32 bytes.

Using the hardware configuration, you assign a logical addresses to each byte of a used module. The addresses are assigned as standard starting at zero, but you can change the proposed address. The logical addresses of the individual modules must not overlap. The logical addresses are defined separately for the input and output modules, so that an input byte can have the same number as an output byte.

The user data of the distributed I/O are also addressed in bytes using a logical address. In order to guarantee an unambiguous assignment of all user data of a CPU (or more exactly: all user data on a P bus), the logical addresses of the distributed I/O must not overlap with the logical addresses of the central modules.

It is usually the case that the digital modules are assigned according to addresses in the process image so that their signal statuses are automatically updated and they can be addressed using the address areas "Input" and "Output". Analog modules, FM modules and CP modules are assigned an address which is not within the process image.

1.4.4 Module Start Address

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

In the case of mixed modules containing input and output ranges, the lower range start address is defined as the module start address. If the input and output ranges have the same start address, use the input address.

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

The modules start address is used in many cases to identify a module. Other than this, it has no special meaning.

1.4.5 Diagnostics Address

Appropriately equipped modules can supply diagnostics data that you can evaluate in your program. If centralized modules have a user data address (module start address), you access the module via this address when reading the diagnostics data. If the modules have no user data address (e.g. power supplies), or if they are part of the distributed I/O, there is a diagnostics address for this purpose.

The diagnostics address is always an address in the I/O input area and occupies one byte. The user data length of this address is zero; if it is located in the process image, as is permitted, it is not taken into account by the CPU when updating the process image.

STEP 7 automatically assigns the diagnostics address counting down from the highest possible I/O address. You can change the diagnostics address with the Hardware Configuration function.

The diagnostics data can only be read with special system functions; accessing this address with load statements has no effect (see also Chapter 20.4 "Communication via Distributed I/O").

1.4.6 Addresses for Bus Nodes

MPI

Modules that are nodes on an MPI network (CPUs, FMs and CPs) also have an **MPI address**. This address is decisive for the link to programming devices, human machine interface devices and for global data communications.

Please note that with older revision levels of the S7-300 CPUs, the FM and CP modules operated in the same station receive an MPI address derived from the MPI address of the CPU. In the case of newer S7-300 CPUs, the MPI addresses of FM and CP modules in the same station can be determined independently of the MPI address of the CPU.

PROFIBUS DP

Each DP station (e.g. DP master, DP slave, programming device) on the PROFIBUS also has a **node address** (station number) with which it can be unambiguously addressed on the bus.

PROFINET IO

Stations on the Industrial Ethernet have a factory-set **MAC address** which is unique worldwide. An **IP address** is additionally required for identification on the bus, and is configured for the IO controller. The IP addresses for the IO devices are derived from the IP address of the IO controller. The IO controller (the interface) and each IO device is additionally assigned a **device name.** The IO device is addressed by the user program by means of a **device number** (station number).

1.5 Address Areas

The address areas available in every programmable controller are

- the peripheral inputs and outputs
- the process input image and the process output image
- the bit memory area
- the timer and counter functions (see Chapters 7 "Timers" and 8 "Counters")
- the L stack (see Chapter 18.1.5 "Temporary Local Data")

To this are added the code and data blocks with the block-local variables, depending on the user program.

1.5.1 User Data Area

In SIMATIC S7, each module can have two address areas: a user data area, which can be directly addressed with Load and Transfer statements, and a system data area for transferring data records.

When modules are accessed, it makes no difference whether they are in racks with centralized configuration or used as distributed I/O. All modules occupy the same (logical) address space.

A module's user data properties depend on the module type. In the case of signal modules, they are either digital or analog input/output signals, and in the case of function modules and communications processors, they might, for example, be control or status information. The volume of user data is module-specific. There are modules that occupy one, two, four or more bytes in this area. Addressing always begins at relative byte 0. The address of byte 0 is the module start address; it is stipulated in the configuration table.

The user data represent the I/O address area, divided, depending on the direction of transfer, into peripheral inputs (PIs) and peripheral outputs (PQs). If the user data are in the area of the process images, the CPU automatically handles the transfers when updating the process images.

Peripheral inputs

You use the peripheral input (PI) address area when you read from the user data area on input modules. Part of the PI address area leads to the process image. This part always begins at I/O address 0; the length of the area is CPU-specific.

With a Direct I/O Read operation, you can access the modules whose interfaces do not lead to the process input image (for instance analog input modules). The signal states of modules that lead to the process input image can also be read with a Direct Read operation.

The momentary signal states of the input bits are then scanned. Please note that this signal state may differ from the relevant inputs in the process image since the process input image is updated at the beginning of the program scan.

Peripheral inputs may occupy the same absolute addresses as peripheral outputs.

Peripheral outputs

You use the peripheral output (PQ) address area when you write values to the user data area on an output module. Part of the PQ address area leads to the process image. This part always begins at I/O address 0; the length of the area is CPU-specific.

With a Direct I/O Write operation, you can access modules whose interfaces do not lead to the process output image (such as analog output modules). The signal states of modules controlled by the process output image can also be directly affected. The signal states of the output bits then change immediately. Please note that a Direct I/O Write operation also updates the signal states of the relevant modules in the process output image! Thus, there is no difference between the process output image and the signal states on the output modules.

Peripheral outputs can reserve the same absolute addresses as peripheral inputs.

1.5.2 Process Image

The process image contains the image of the digital input and digital output modules, and is thus divided into process input image and process output image. The process input image is accessed via the address area for inputs (I), the process output image via the address area for outputs (Q). As a rule, the machine or process is controlled via the inputs and outputs.

The process image can be divided into subsidiary process images that can be updated either automatically or via the user program. Please refer to Chapter 20.2.1 "Process Image Updating" for more details.

On the S7-300 CPUs and, from 10/98, also on S7-400 CPUs, you can use the addresses of the process image not occupied by modules as additional memory area similar to the bit memory area. This applies both for the process input image and the process output image.

On suitably equipped CPUs, say, the CPU 417, the size of the process image can be parameterized. If you enlarge the process image, you reduce the size of the work memory accordingly. Following a change to the size of the process image, the CPU executes initialization of the work memory, with the same effect as a cold restart.

Inputs

An input is an image of the corresponding bit on a digital input module. Scanning an input is the same as scanning the bit on the module itself. Prior to program execution in every program cycle, the CPU's operating system copies the signal state from the module to the process input image.

The use of a process input image has many advantages:

- Inputs can be scanned and linked bit by bit (I/O bits cannot be directly addressed).
- Scanning an input is much faster than accessing an input module (for example, you avoid the transient recovery time on the I/O bus, and the system memory response times are shorter than the module's response times). The program is therefore executed that much more quickly.
- The signal state of an input is the same throughout the entire program cycle (there is data consistency throughout a program cycle). When a bit on an input module changes, the change in the signal state is transferred to the input at the start of the next program cycle.
- Inputs can also be set and reset because they are located in random access memory. Digital input modules can only be read. Inputs can be set during debugging or startup to simulate sensor states, thus simplifying program testing.

These advantages are offset by an increased program response time (please also refer to Chapter 20.2.4 "Response Time").

Outputs

An output is an image of the corresponding bit on a digital output module. Setting an output is the same as setting the bit on the output module itself. The CPU's operating system copies the signal state from the process output image to the module.

The use of a process output image has many advantages:

- Outputs can be set and reset bit by bit (direct addressing of I/O bits is not possible).
- Setting an output is much faster than accessing an output module (for example, you avoid the transient recovery time on the I/O bus, and the system memory response times are shorter than the module response times). The program is therefore executed that much more quickly.
- A multiple signal state change at an output during a program cycle does not affect the bit on the output module. It is the signal state of the output at the end of the program cycle that is transferred to the module.
- Outputs can also be scanned because they are located in random access memory. While it is possible to write to digital output modules, it is not possible to read them. The scanning and linking of the outputs makes additional storage of the output bit to be scanned unnecessary.

These advantages are offset by an increased program response time. Chapter 20.2.4 "Response Time" describes how a programmable controller's response time comes about.

1.5.3 Consistent User Data

Data consistency means that data can be handled in a block. Transfer of a data block must not be interrupted, and it is not permissible for the data source or target to be changed from the other end during a transmission either. For example, if you transfer four bytes individually, the transmitting program can be interrupted by a program of higher priority between each byte, and this program could change the data in the source or target area.

In the case of direct access to user data (loading and transferring), the data are read and written as byte, word or doubleword. The load and transfer instructions, upon which the MOVE box with LAD/FBD and the assignment of variables with elementary data types with SCL are based, are designed as interruptible. If you wish to transfer a data block with more than four bytes without interruption between system memory and work memory, use the system function SFC 81 UBLKMOV.

Data transfer between a DP slave and DP master is consistent for a complete slave even if e.g. the transfer area for an intelligent DP slave is divided into several consistent blocks. Data consistency with internode communication is the same as with direct access (1-, 2- and 4-byte consistency). This similarly applies to data transfer between IO controller and IO devices on the PROFINET IO.

When configuring stations of the distributed I/O with three or more than four bytes of user data, you can specify the consistent user data areas. These areas are transferred consistent to the parameterized target area (e.g. data area in work memory or process image) using the system functions SFC 14 DPRD_DAT and SFC 15 DPWR_DAT.

Please note that the "normal" updating of process images can be interrupted following each transmitted doubleword. An exception with newer CPUs is the transfer of user data blocks for distributed I/O using a partial process image if the user data blocks can be configured as consistent using the hardware configuration. You can also influence these data blocks in the process image using a direct access, but you could also possibly destroy the data consistency.

CPU-specific data apply to the maximum size of a consistent area for data transfer with global data communications, S7 basic communications and S7 communications through the operating system (see Technical specifications in the CPU manual).

Diagnostics data and parameters are always transferred consistently in data records (e.g. diagnostics data with the SFC 13 DPMRM_DG or SFB 54 RALRM, or parameter data transferred to and from modules with the SFB 52 RDREC and SFB 53 WRREC).

1.5.4 Bit Memories

The area called bit memories holds what could be regarded as the controller's "auxiliary contactors". Bit memories are used primarily for storing binary signal states. The bits in this area can be treated as outputs, but are not "externalized". Bit memories are located in the CPU's system memory area, and is therefore available at all times. The number of bits in bit memories is CPU-specific.

Bit memories are used to store intermediate results that are valid beyond block boundaries and are processed in more than one block. Besides the data in global data blocks, the following are also available for storing intermediate results

- Temporary local data, which are available in all blocks but valid for the current block call only, and
- Static local data, which are available only in function blocks but valid over multiple block calls.

Retentive bit memories

Part of bit memories may be designated "retentive", which means that the bits in that part of bit memories retain their signal states even under off-circuit conditions. Retentivity always begins with memory byte 0 and ends at the designated location. Retentivity is set when the CPU is parameterized. Please refer to Chapter 22.2.4 "Retentivity" for additional information.

Clock memories

Many procedures in the controller require a periodic signal. Such a signal can be implemented using timers (clock pulse generator), watchdog interrupts (time-controlled program execution), or simply by using clock memories.

Clock memories consist of bits whose signal states change periodically with a mark-to-space ratio of 1:1. The bits are combined into a byte, and correspond to fixed frequencies (Figure 1.10). You specify the number of clock memory bits when you parameterize the CPU. Please note that the updating of clock memories is asynchronous to execution of the main program.

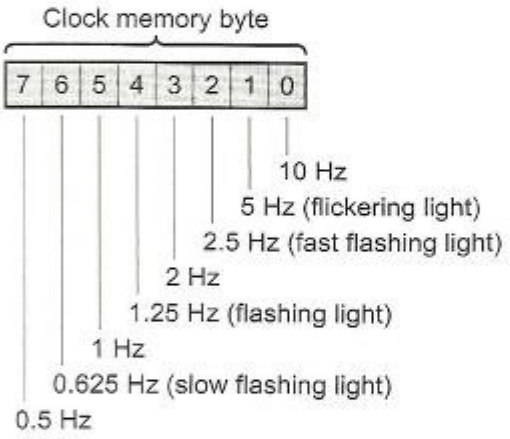


Figure 1.10 Contents of the Clock Memory Byte

2 STEP 7 Programming Software

2.1 STEP 7 Basis Package

This chapter describes the STEP 7 basic package, Version 5.4 SP3. While the first chapter presented an overview of the properties of the programmable controller, this chapter tells you how to set these properties.

The basic package contains the statement list (STL), ladder logic (LAD) and function block diagram (FBD) programming languages. In addition to the basic package, option packages such as S7-SCL (Structured Control Language), S7-GRAPH (sequence planning) and S7-HiGraph (state-transition diagram) are also available.

2.1.1 Installation

STEP 7 V5.4 is a 32-bit application which executes with MS Windows 2000 Professional with SP4, MS Windows XP Professional with SP2, MS Windows Server 2003 SP2 standard edition as workstation computer or MS Windows Vista 32-Bit Ultimate and Business. MS Internet Explorer V6.0 or higher is required under all operating systems. You require administrator privileges in order to install STEP 7, and you must be registered at least as a main user in order to work with STEP 7.

If you wish to work rapidly with STEP 7 or process large projects, e.g. with several hundred modules, you should use a programming device or PC with up-to-date processing power.

STEP 7 V5.4 occupies approximately 650 to 900 MB on the hard disk depending on the scope of installation and the number of installed languages. A swap-out file is also needed, whose size must be at least twice that of the main memory.

You should ensure there is sufficient memory on the drive containing your project data. The memory requirements may increase for certain operations, such as copying a project. If there is insufficient space for the swap-out file, errors such as program crashes may occur. You are recommended not to store the project data on the drive containing the Windows swap-out file.

The SETUP program on the CD is used for installation, or STEP 7 is already factory-installed on the programming device. In addition to STEP 7, the CD also includes, *inter alia*, the Automation License Manager (see Chapter 2.1.2 "Automation License Manager") and the STEP 7 electronic manuals with Acrobat Reader.

An MPI interface is needed for the online connection to a programmable controller. The programming devices have the multipoint interface already built in, but PCs must be retrofitted with an MPI module. If you want to use PC memory cards or micro memory cards, you will need a prommer.

STEP 7 V5 has multi-user capability, that is, a project that is stored, say, on a central server can be edited simultaneously from several workstations. You make the necessary settings in the Windows Control Panel with the "SIMATIC Workstation" program. In the dialog box that appears, you can parameterize the workstation as a single-user system or a multi-user system with the protocols used.

Deinstallation of STEP 7 is carried out with the setup program or in the usual manner for MS Windows using the "Software" program in the Windows Control Panel.

2.1.2 Automation License Manager

A license (right of use) is required to operate STEP 7. This consists of the certificate of license and the electronic license key. The license key is provided on the license key disk or a USB stick.

A license key can be present on the license key disk, on a USB stick and on local or networked hard disks. A license key will only function if it is present on a hard disk with write access. You use the *Automation License Manger* to transfer and administer the license keys. Installation of the Automatic License Manager is a requirement for operating STEP 7. You can install the Automation License Manager together with STEP 7 or on its own.

The type of license key is defined in the certificate of license:

- Single License
  This license is applicable for an unlimited time, and permissible on any one computer.
- Floating License
  This license is applicable for an unlimited time, and provided for procurement via a network.
- Trial License
  This license is limited to 14 days, or to a certain number of days starting with its initial use. It can be used for testing and validation.
- Upgrade License
  This license permits upgrading of an authorization/license key from a previous version to the current version.

During installation of STEP 7, licensing will be requested if an appropriate license key is not yet present on the hard disk. You can also carry out licensing at a later point in time.

The license key is saved on the hard disk in specially identified blocks. To prevent unintentional destruction of the license key, please observe the information on the handling of license keys provided in the help function of the Automation License Manager.

2.1.3 SIMATIC Manager

The SIMATIC Manager is the main tool in STEP 7; you will find its icon in Windows.

The SIMATIC Manager is started by double-clicking on its icon.

When first started, the project wizard is displayed. This can be used for simple creation of new projects. You can deactivate it with the check box "Display Wizard on starting the SIMATIC Manager" since it can also be called, if required, via the menu command FILE → "NEW PROJECT" WIZARD.

Programming begins with opening or creating a "project". The example projects supplied are a good basis for familiarization.

When you open example project ZEn01_09_STEP7_Zebra with FILE → OPEN, you will see the split project window: on the left is the structure of the open object (the object hierarchy), and on the right is the selected object. Clicking on the box containing a plus sign in the left window displays additional levels of the structure; selecting an object in the left half of the window displays its contents in the right half of the window (Figure 2.1).

Under the SIMATIC Manager, you work with the objects in the STEP 7 world. These "logical" objects correspond to "real" objects in your plant. A project contains the entire plant, a station corresponds to a programmable controller. A project may contain several stations connected to one another, for example, via an MPI subnet. A station contains a CPU, and the CPU contains a program, in our case an S7 program. This program, in turn, is a "container" for other objects, such as the object *Blocks*, which contains, among other things, the compiled blocks.

The STEP 7 objects are connected to one another via a tree structure. Figure 2.2 shows the most important parts of the tree structure (the "main branch", as it were) when you are working with the STEP 7 basic package for S7

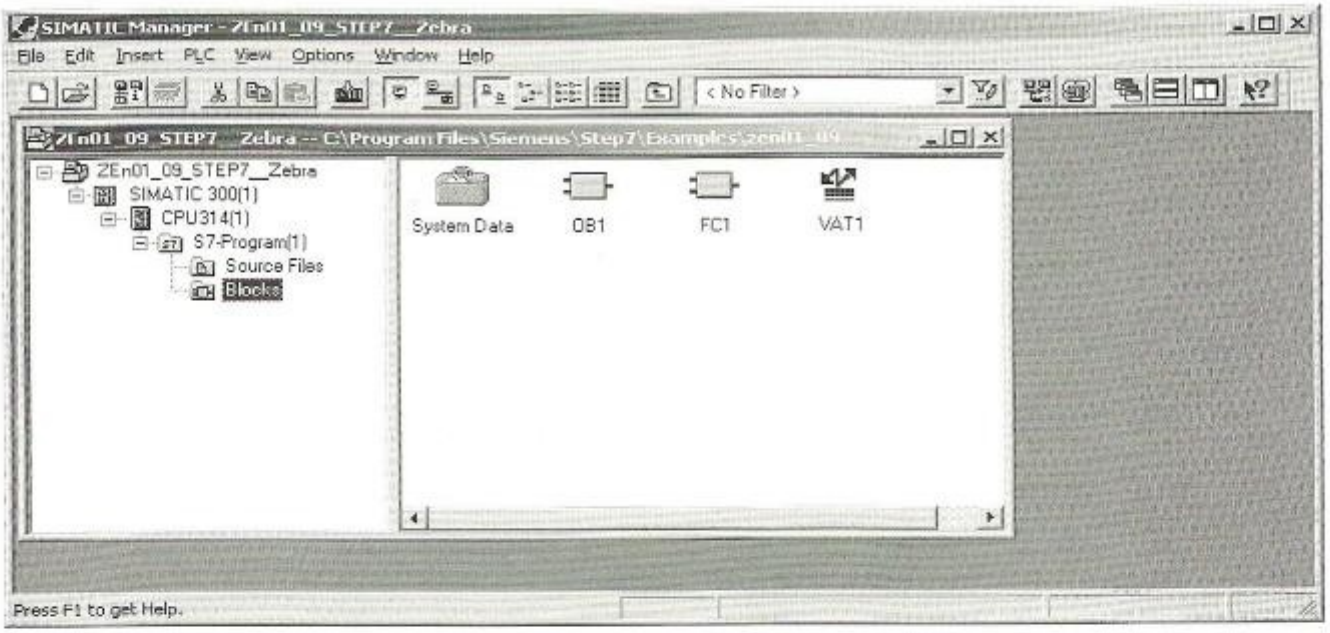


Figure 2.1 SIMATIC Manager Example

applications in offline view. The objects shown in bold type are containers for other objects.

All objects in the Figure are available to you in the offline view. These are the objects that are on the programming device's hard disk. If your programming device is online on a CPU (normally a PLC target system), you can switch to the online view by selecting VIEW → ONLINE. This option displays yet another project window containing the objects on the destination device; the objects shown in italics in the Figure are then no longer included.

You can see from the title bar of the active project window whether you are working offline or online. For clearer differentiation, the title bar and the window title can be set to a different color than the offline window. For this purpose, select OPTIONS → CUSTOMIZE and modify the entries in the "View" tab.

Select OPTIONS → CUSTOMIZE to change the SIMATIC Manager's basic settings, such as the session language, the archive program and the storage location for projects and libraries, and configuring the archive program.

Editing sequences

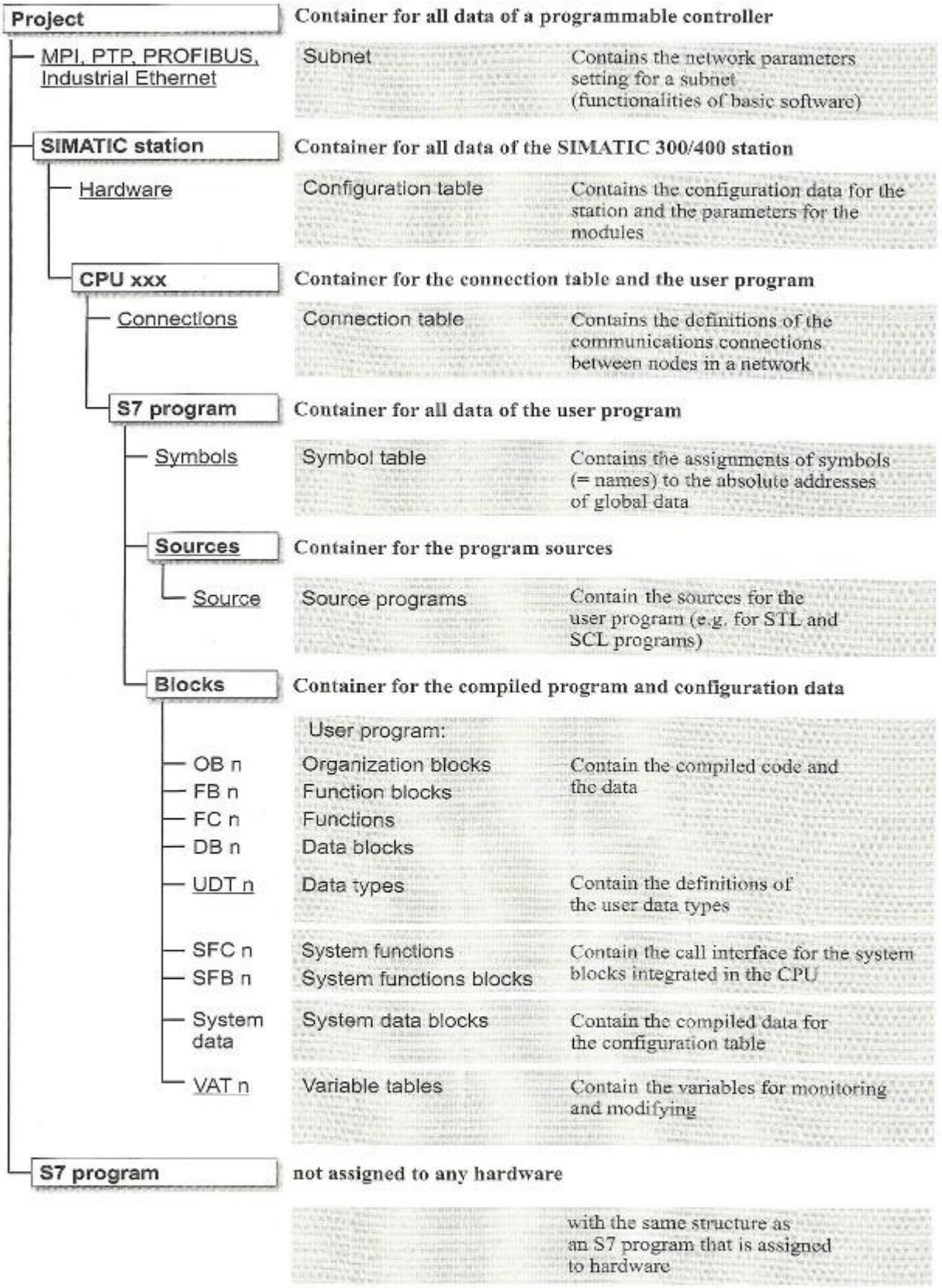
The following applies for the general editing of objects:

To *select an object* means to click on it once with the mouse so that it is highlighted (this is possible in both halves of the project window).

To *name an object* means to click on the name of the selected object (a frame will appear around the name and you can change the name in the window) or select the menu item EDIT → OBJECT PROPERTIES and change the name in the dialog box. With some objects such as a CPU, you can only change the name with the relevant tool (application), in this case with the Hardware Configuration.

To *open an object*, double-click on that object. If the object is a container for other objects, the SIMATIC Manager displays the contents of the object in the right half of the window. If the object is on the lowest hierarchical level, the SIMATIC Manager starts the appropriate tool for editing the object (for instance, double-clicking on a block starts the editor, allowing the block to be edited).

In this book, the menu items in the standard menu bar at the top of the window are described



(The underlinedt objects are only present in the offline data management.)

Figure 2.2 Object Hierarchy in a STEP 7 Project

as *operator sequences*. Programmers experienced in the use of the operator interface use the icons from the toolbar. The use of the *right mouse button* is very effective. Clicking on an object once with the right mouse button screens a menu showing the current editing options.

2.1.4 Projects and Libraries

In STEP 7, the "main objects" at the top of the object hierarchy are projects and libraries. Starting with STEP 7 V5.2, you can combine projects and libraries into multiprojects (see Chapter 2.1.5 "Multiprojects").

Projects are used for the systematic storing of data and programs needed for solving an automation task. Essentially, these are

- the hardware configuration data,
- the parameterization data for the modules,
- the configuring data for communication via networks,
- the programs (code and data, symbols, sources).

The objects in a project are arranged hierarchically. The opening of a project is the first step in editing all (subordinate) objects which that object contains. The following sections discuss how to edit these objects.

Libraries are used for storing reusable program components. Libraries are organized hierarchically. They may contain STEP 7 programs which in turn may contain a user program (a container for compiled blocks), a container for source programs, and a symbol table. With the exception of online connections (no debugging possible), the creation of a program or program section in a library provides the same functionality as in an object.

As supplied, STEP 7 V5 provides the *Standard Library* containing the following programs:

- System Function Blocks
  Contains the call interfaces of the system blocks for offline programming integrated in the CPU
- S5-S7 Converting Blocks
  Contains loadable functions for the S5/S7 converter (replacement of S5 standard function blocks in conjunction with program conversion)
- T1-S7 Converting Blocks
  Contains additional loadable functions and function blocks for the T1-S7 converter
- IEC Function Blocks
  Contains loadable functions for editing variables of the complex data types DATE_AND_TIME and STRING
- Communication Blocks
  Contains loadable functions for controlling CP modules
- Miscellaneous Blocks
  Contains blocks for time stamping and time synchronization
- PID Control Blocks
  Contains loadable function blocks for closed-loop control
- Organization Blocks
  Contains the templates for the organization blocks (essentially the variable declaration for the start information)

You will find an overview of the contents of these libraries in Chapter 25 "Block Libraries". Should you, for example, purchase an S7 module with standard blocks, the associated installation program installs the standard blocks as a library on the hard disk. You can then copy these blocks from the library to your project. A library is opened with FILE → OPEN, and can then be edited in the same way as a project. You can also create your own libraries.

The menu item FILE → NEW generates a new object at the top of the object hierarchy (project, library). The location in the directory structure where the SIMATIC Manager is to create a project or library must be specified under the menu item OPTIONS → CUSTOMIZE or in the "New dialog" box.

The INSERT menu is used to add new objects to existing ones (such as adding a new block to a program). Before doing so, however, you must first select the object container in which you want to insert the new object from the left half of the SIMATIC Manager window.

You copy object containers and objects with EDIT → COPY and EDIT → PASTE or, as is usual with Windows, by dragging the selected object

with the mouse from one window and dropping it in another. Please note that you cannot undo deletion of an object or an object container in the SIMATIC Manager.

2.1.5 Multiprojects

In a multiproject, projects and libraries are combined in an entity. The multiproject allows processing of communications connections such as S7 connections between the projects. A multiproject can then be handled almost like a single project. Limitations: stations connected together by means of direct data exchange ("internode communication") or through global data communication must be present in the same project.

In a multiproject, it is possible to carry out parallel processing of individual projects by various employees without problem. The individual projects can be present in different directories in a networked environment. The cross-project functions, such as the matching of subnetworks and connections, are then carried out centrally when processing the multiproject. In the case of central storage on a server, only the operating systems MS Windows 2000 Server and MS Windows Server 2003 are permitted.

It is also advantageous to create a multiproject if you wish to make the individual projects smaller and clearer.

2.1.6 Online Help

The SIMATIC Manager's online help provides information you need during your programming session without the need to refer to hard-copy manuals. You can select the topics you need information on by selecting the HELP menu. The online help option GETTING STARTED, for instance, provides a brief summary on how to use the SIMATIC Manager.

HELP → CONTENTS starts the central STEP 7 Help function from any application. This contains all the basic knowledge. If you click on the "Home" symbol in the menu bar (start page), you will be provided with an introduction to the central topics of STEP 7: Starting with STEP 7, Configuring & programming, Tests & troubleshooting, as well as SIMATIC on the Internet.

HELP → CONTEXT-SENSITIVE HELP F1 provides context-sensitive help, i.e. if you press F1, you get information concerning an object selected by the mouse or concerning the current error message.

In the symbol bar, there is a button with an arrow and a question mark. If you click on this button, a question mark is added to the mouse pointer. With this "Help" mouse pointer, you can now click on an object on the screen, e.g. a symbol or a menu command, and you will get the associated online help.

2.2 Editing Projects

When you set up a project, you create "containers" for the resulting data, then you generate the data and fill these containers. Normally, you create a project with the relevant hardware, configure the hardware, or at least the CPU, and receive in return containers for the user program. However, you can also put an S7 program directly into the project container without involving any hardware at all. Note that initializing of the modules (address modifications, CPU settings, configuring connections) is possible only with the Hardware Configuration tool.

We strongly recommend that the entire project editing process be carried out using the SIMATIC Manager. Creating, copying or deleting directories or files as well as changing names (!) with the Windows Explorer within the structure of a project can cause problems with the SIMATIC Manager.

2.2.1 Creating Projects

Project wizard

The STEP 7 Wizard helps you in creating a new project. You specify the CPU used and the wizard creates for you a project with an S7 station and the selected CPU as well as an S7 program container, a source container and a block container with the selected organization blocks.

You start the project wizard using FILE → "NEW PROJECT" WIZARD.

Creating a project with the S7 station

If you want to create a project "manually", this section outlines the necessary actions for you. You will find general information on operator entries for object editing in Chapter 2.1.3 "SIMATIC Manager".

Creating a new project

Select FILE → NEW, enter a name in the dialog box, change the type and storage location if necessary, and confirm with "OK" or RETURN.

Inserting a new station in the project

Select the project and insert a station with INSERT → STATION → SIMATIC 300 STATION (in this case an S7-300).

Configuring a station

Click on the plus box next to the project in the left half of the project window and select the station; the SIMATIC Manager displays the Hardware object in the right half of the window. Double-clicking on *Hardware* starts the Hardware Configuration tool, with which you edit the configuration tables.

If the module catalog is not on the screen, call it up with VIEW → CATALOG.

You begin configuring by selecting the rail with the mouse, for instance under "SIMATIC 300" and "RACK 300", "holding" it, dragging it to the free portion in the upper half of the station window, and "letting it go" (drag & drop). You then see a table representing the slots on the rail.

Next, select the required modules from the module catalog and, using the procedure described above, drag and drop them in the appropriate slots. To enable further editing of the project structure, a station requires at least one CPU, for instance the CPU 314 in slot 2. You can add all other modules later. Editing of the hardware configuration is discussed in detail in Chapter 2.3 "Configuring Stations".

Store and compile the station, then close and return to the SIMATIC Manager. In addition to the hardware configuration, the open station now also shows the CPU.

When it configures the CPU, the SIMATIC Manager also creates an S7 program with all objects. The project structure is now complete.

Viewing the contents of the S7 program

Open the CPU; in the right half of the project window you will see the symbols for the *S7 program* and for the *connection table*.

Open the S7 program; the SIMATIC Manager displays the symbols for the compiled user program (the compiled blocks), the container for the source programs, and the symbol table in the right half of the window.

Open the user program (*Blocks*); the SIMATIC Manager displays the symbols for the compiled configuration data *(System data)* and an empty organization block for the main program (OB 1) in the right half of the window.

Editing user program objects

We have now arrived at the lowest level of the object hierarchy. The first time OB 1 is opened, the window with the object properties is displayed and the editor needed to edit the program in the organization block is opened. You add another empty block for incremental editing by opening INSERT → S7 BLOCK → ... (*Blocks* must be highlighted) and selecting the required block type from the list provided.

When opened, the *System data* object shows a list of available system data blocks. You receive the compiled configuration data. These system data blocks are edited via the *Hardware* object in the container *Station*. You can transfer *System data* to the CPU with PLC → DOWNLOAD and parameterize the CPU in this way.

The object container *Sources* is empty. With *Sources* selected, you can select INSERT → S7 SOFTWARE → STL SOURCE to insert an empty source text file or you can select INSERT → EXTERNAL SOURCE to transfer a source text file created, say, with another editor in ASCII format to the *Sources* container.

Creating a project without an S7 station

If you wish, you can create a program without first having to configure a station. To do so, generate the container for your program your-

self. Select the project and generate an S7 program with INSERT → PROGRAM → S7-PROGRAM. Under this S7 program, the SIMATIC Manager creates the object containers *Sources* and *Blocks*. *Blocks* contains an empty OB 1.

Creating a library

You can also create a program under a library, for instance if you want to use it more than once. In this way, the standard program is always available and you can copy it entire or in part into your current program.

Please note that you cannot establish online connections in a library, which means that you can debug a STEP 7 program only within a project.

2.2.2 Managing, Reorganizing and Archiving

The SIMATIC Manager maintains a list of all known "main objects", arranged according to user projects, libraries, example projects and multiprojects. You install the example projects and the standard libraries in conjunction with STEP 7 and you install the user projects, the multiprojects and your own libraries yourself.

When you execute FILE → MANAGE, the SIMATIC Manager shows you a list of all known projects and libraries with name and path. You can then delete from the list projects or libraries you no longer want to display ("Hide") or include in the list new projects and libraries ("Display").

When it executes FILE → REORGANIZE, the SIMATIC Manager eliminates the gaps created by deletions and optimizes data memory similarly to the way a defragmentation program optimizes the data memory on the hard disk. The reorganization can take some time, depending on the data movements involved.

You can also archive a project or library (FILE → ARCHIVE). In this case, the SIMATIC Manager stores the selected object (the project or library directory with all subdirectories and files) in compressed form in an archive file.

From STEP 7 V5.4 SP3, the archive program PKZip V8.6 CLI is supplied for archiving and dearchiving projects and libraries (the archive program ARJ.exe is not suitable for MS Windows Vista). You can also open ARJ archives with PKZip V8.6 CLI.

Projects and libraries cannot be edited in the archived (compressed) state. You can unpack an archived object with FILE → RETRIEVE and then you can edit it further. The retrieved objects are automatically accepted into the project or library management system.

You make the settings for archiving and retrieving on the "Archive" tab under OPTIONS → CUSTOMIZE; e.g. setting the target directory for archiving and retrieving or "Generate archive path automatically" (then no additional specifications are required when archiving because the name of the archive file is generated from the project name).

Archiving a project in the CPU

With the appropriately designed CPUs, you can store a project in archived (compressed) form in the load memory of the CPU, that is, on the memory card. In this way, you can save all project data required for full execution of the user program, such as symbols or source files, direct at the machine or plant. If it becomes necessary to modify or supplement the program, you load the locally stored data onto the hard disk, correct the user program, and save the up-to-date project data again to the CPU.

When loading the project data onto a memory card or micro memory card plugged into the CPU, open the project, mark the CPU and select PLC → SAVE TO MEMORY CARD. In the reverse direction, transfer the stored data back to the programming device with PLC → RETRIEVE FROM MEMORY CARD. Please note that when you write to a memory card plugged into the CPU, the entire contents of the load memory are written to the CPU, including the system data and the user programs.

If you want to fetch back the project data stored on the CPU without creating a project on the hard disk, select the relevant CPU with PLC → DISPLAY ACCESSIBLE NODES. If the memory card is plugged into the module receptacle of the programming device, select the memory card with FILE → S7 MEMORY CARD → OPEN before transferring.

2.2.3 Project Versions

Since STEP 7 V5 has become available, there are three different versions of SIMATIC projects. STEP 7 V1 creates version 1 projects, STEP 7 V2 creates version 2 projects, and STEP 7 V3/V4/V5.0 can be used to create and edit both version 2 and version 3 projects. With STEP 7 from version V5.1, you can create and edit V3 projects and V3 libraries.

If you have a version 1 project, you can convert it into a version 2 project with FILE → OPEN VERSION 1 PROJECT. The project structure with the programs, the compiled version 1 blocks, the STL source programs, the symbol table and the hardware configuration remain unchanged.

You can create and edit version 2 projects with STEP 7 versions V2, V3, V4 and V5.0 (Figure 2.3). STEP 7 V5.1 works only with version 3 projects.

Up to STEP 7 Version 5.3 you can convert a V1 project to a V2 project with FILE → OPEN VERSION 1 PROJECT. With FILE → OPEN , you can open a V2 project and convert it to a V3 project. It is not possible to create a V2 project or save a project as a V2 project.

2.2.4 Creating and editing multiprojects

Using FILE → NEW you can create a new multiproject in the SIMATIC Manager in which you select "Multiproject" as the type in the dialog box. With the multiproject selected, you can then generate a new project or a new library in the multiproject using FILE → MULTIPROJECT → CREATE IN MULTIPROJECT. You can process the newly created project or library as described in the previous chapters. Using FILE → MULTIPROJECT → INSERT INTO MULTIPROJECT you can incorporate existing projects and libraries into the multiproject.

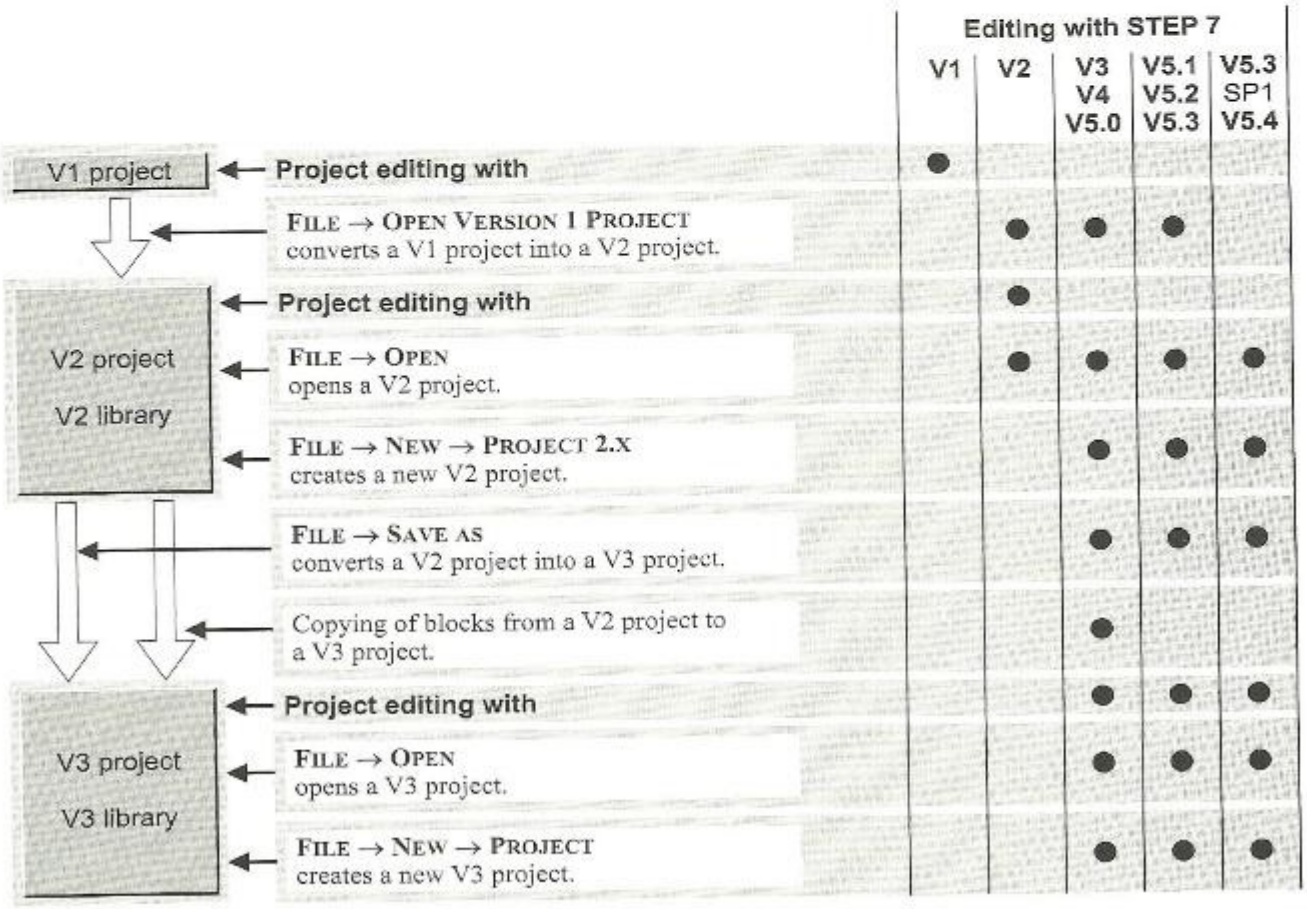


Figure 2.3 Editing Projects with Different Versions

You can also eliminate projects and libraries again from the multiproject: mark the projects: library, and select FILE → MULTIPROJECT → REMOVE FROM MULTIPROJECT. The project or library is not deleted in this process.

Using FILE → MULTIPROJECT → ADJUST PROJECTS you can start a wizard which supports you in the matching of cross-project connections and when combining subnets (Chapter 2.4.6 "Matching Projects in a Multiproject").

You can identify one of the libraries in a multiproject as the "master data library" using FILE → MULTIPROJECT → DEFINE AS MASTER DATA LIBRARY. This contains, for example, the common blocks of the projects in this multiproject. This library must then only contain one single S7 program.

The menu commands FILE → SAVE AS, FILE → REORGANIZE, FILE → MANAGE and FILE → ARCHIVE can also be used on a multiproject, and function as with a single project (see Chapter 2.2.2 "Managing, Reorganizing and Archiving"). In the same manner, archived multiprojects can be transferred to the load memory of a correspondingly designed CPU. There are limitations when archiving a multiproject whose components are distributed among network drives.

2.3 Configuring Stations

You use the Hardware Configuration tool to plan your programmable controller's configuration. Configuring is carried out offline without connection to the CPU. You can also use this tool to address and parameterize the modules. You can create the hardware configuration at the planning stage or you can wait until the hardware has already been installed.

You start the hardware configuration by selecting the station and then EDIT → OPEN OBJECT or by double-clicking on the *Hardware* object in the opened container *SIMATIC 300/400 Station*. You make the basic settings of the hardware configuration with OPTIONS → CUSTOMIZE.

When configuring has been completed, STATION → CONSISTENCY CHECK will show you whether your entries were free of errors. STATION → SAVE stores the configuration tables with all parameter assignment data in your project on the hard disk.

STATION → SAVE AND COMPILE not only saves but also compiles the configuration tables and stores the compiled data in the *System data* object in the offline container *Blocks*. After compiling, you can transfer the configuration data to a CPU with PLC → DOWNLOAD. The object *System data* in the online container *Blocks* represents the current configuration data on the CPU. You can "return" these data to the hard disk with PLC → UPLOAD.

You export the data of the hardware configuration with STATION → EXPORT. STEP 7 then creates a file in ASCII format that contains the configuration data and parameterization data of the modules. You can choose between a text format that contains the data in "readable" English characters, or a compact format with hexadecimal data. You can also import a correspondingly structured ASCII file.

Checksum

The Hardware Configuration generates a checksum via a correctly compiled station and stores it in the system data. Identical system configurations have the same checksum so that you can, for example, easily compare an online configuration with an offline configuration.

The checksum is a property of the *System data* object. To read the checksum, open the *Blocks* container in the S7 program, select the *System data* object and open it with EDIT → OPEN OBJECT.

The user program also has an appropriate checksum. You can find this along with the checksum of the system data in the properties of *Blocks*: select the *Blocks* container and then EDIT → OBJECT PROPERTIES on the "Checksums" tab.

Station window

When opened, the Hardware Configuration displays the station window and the hardware catalog (Figure 2.4). Enlarge or maximize the station window to facilitate editing. In the upper section, it displays the S7 stations in the

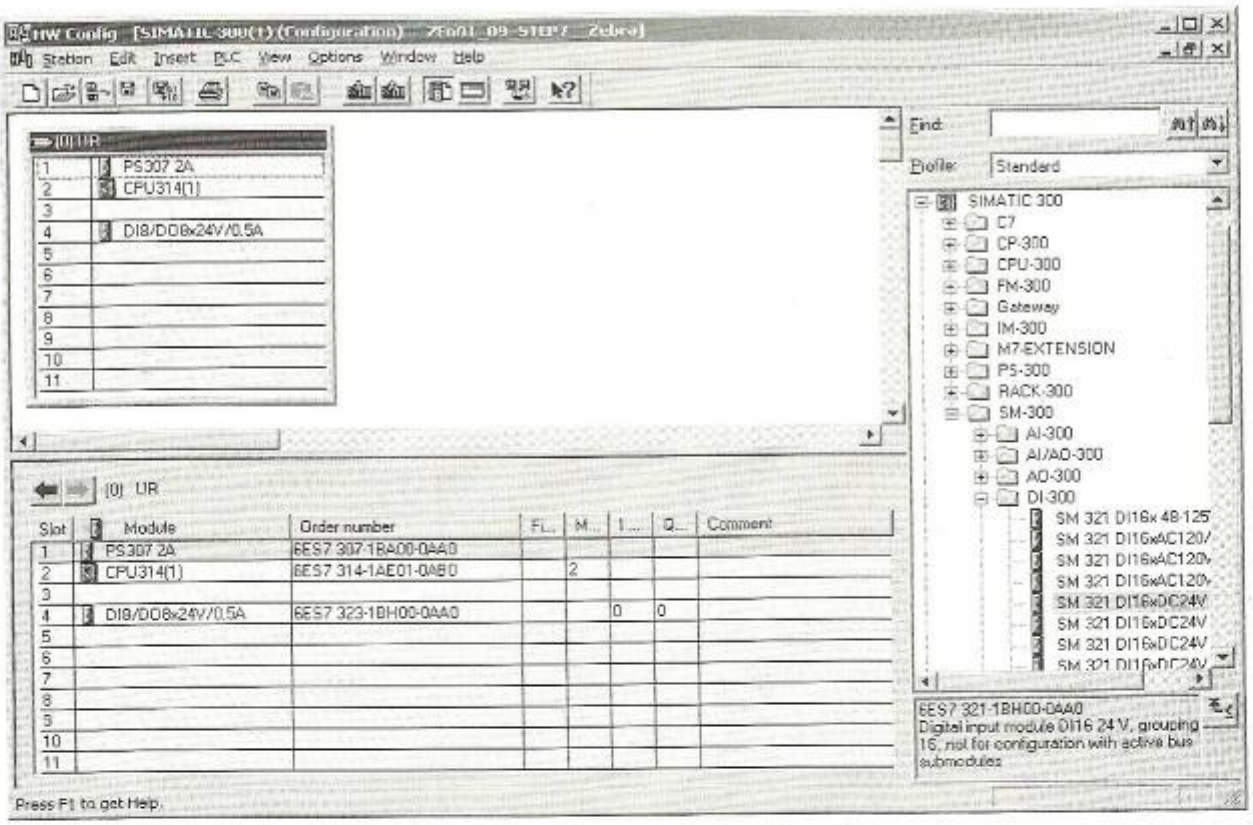


Figure 2.4 Example of a Station Window in the Hardware Configuration

form of tables (one per mounting rack) that are connected together via interface modules when several mounting racks are used. When distributed I/O is connected, the configuration of the DP master system or the PROFINET IO system is specified, with the DP stations and IO devices represented in the form of symbols. The lower section of the station window shows the configuration table that gives a detailed view of the rack or DP slave selected in the upper section.

Hardware catalog

You can fade the hardware catalog in and out with VIEW → CATALOG. It contains all available mounting racks, modules and interface submodules known to STEP 7. With OPTIONS → EDIT CATALOG PROFILE, you can compile your own hardware catalog that shows only the modules you want to work with – in the structure you select. By double-clicking on the title bar, you can "dock" the hardware catalog onto the right edge of the station window or release it again.

Installing hardware updates

With OPTIONS → INSTALL HW UPDATES you can update components for the hardware catalog. In the following dialogs, select whether you wish to download the update from the Internet or copy it from a CD. Enter the Internet address and the save path. By clicking the "Install" button, the Hardware Configuration transfers the data into the hardware catalog.

Product support information

With HELP → PRODUCT SUPPORT INFORMATION you can display information from the Internet for the selected module. You must first enable this function with OPTIONS → CUSTOMIZE and set a valid Internet address. The selected module can be in the hardware catalog or already in the configured rack.

Configuration table

The Hardware Configuration tool works with tables that each represent an S7 station (a mounting rack), a DP station or an IO device. A

configuration table shows the slots with the modules arranged in the slots or the properties of the modules such as their addresses and order numbers. A double-click on a module line opens the properties window of the module and allows parameterization of the module.

2.3.1 Arranging Modules

You begin configuring by selecting and "holding" the rail from the module catalog, for instance under "SIMATIC 300" and "RACK 300", with the mouse, dragging it to the upper half of the station window, and dropping it anywhere in that window (drag & drop). An empty configuration table is screened for the central rack. Next, select the required modules from the module catalog and, in the manner described above, drag and drop them in the appropriate slots. The permissible slots have a green background. A "No Parking" symbol tells you cannot drop the selected module at the intended slot.

You can also mark the slot to be equipped, and select INSERT → INSERT OBJECT. In a popup window, the Hardware Configuration then shows you all modules permissible for this slot, from which you can select one.

In the case of single-tier S7-300 stations, slot 3 remains empty; it is reserved for the interface module to the expansion rack.

You can generate the configuration table for another rack by dragging the selected rack from the catalog and dropping it in the station window. In S7-400 systems, a non-interconnected rack (or more precisely: the relevant receive interface module) is assigned an interface via the "Link" tab in the Properties window of a Send IM (select module and EDIT → OBJECT PROPERTIES).

The arrangement of distributed I/O stations is described in Chapter 20.4 "Communication via Distributed I/O".

2.3.2 Addressing Modules

When arranging modules, the Hardware Configuration tool automatically assigns a module start address. You can view this address in the lower half of the station window in the object properties for the relevant modules in the "Addresses" tab. If you deselect the option "System default" in this tab for S7-300 modules, you can change the module addresses. When doing so, please observe the addressing rules for S7-300 and S7-400 systems as well as the addressing capacity of the individual modules.

There are modules that have both inputs and outputs for which you can (theoretically) reserve different start addresses. However, please note carefully the special information provided in the product manuals; the large majority of function and communications modules require the same start address for inputs and outputs.

When assigning the module start address on the S7-400, you can also make the assignment to a subsidiary process image. If there is more than one CPU in the central rack, multiprocessor mode is automatically set and you must assign the module to a CPU.

With VIEW → ADDRESS OVERVIEW, you get a window containing all the module addresses currently in use for the CPU selected.

Modules on the MPI bus or communications bus have an MPI address. You may also change this address. Note, however, that the new MPI address becomes effective as soon as the configuration data are transferred to the CPU.

Symbols for user data addresses

In the Hardware Configuration tool, you can assign to the inputs and outputs symbols (names) that are transferred to the Symbol Table.

After you have arranged and addressed the digital and analog modules, you save the station data. Then you select the module (line) and EDIT → SYMBOLS. In the window that then opens, you can assign a symbol, a data type and a comment to the absolute address for each channel (bit-by-bit for digital modules and word-by-word for analog modules).

The "Add Symbol" button enters the absolute addresses as symbols in place of the absolute addresses without symbols. The "Apply" button transfers the symbols into the Symbol Table. "OK" also closes the dialog box.

2.3.3 Parameterizing Modules

When you parameterize a module, you define its properties. It is necessary to parameterize a module only when you want to change the default parameters. A requirement for parameterization is that the module is located in a configuration table.

Double-click on the module in the configuration table or select the module and then EDIT → OBJECT PROPERTIES. Several tabs with the specifiable parameters for this module are displayed in the dialog box. When you use this method to parameterize a CPU, you are specifying the run characteristics of your user program.

Some modules allow you to set their parameters at runtime via the user program with the system functions (see Chapter 22.5.2 "System Blocks for Module Parameterization").

2.3.4 Networking Modules with MPI

You define the nodes for the MPI subsidiary (subnet) with the Module Properties. Select the CPU, or the MPI interface card if the CPU is equipped with one, in the configuration table and open it with EDIT → OBJECT PROPERTIES. The dialog box that then appears contains the "Properties" button in the "Interface" box of the "General" tab. If you click on this button you are taken to another dialog box with a "Parameter" tab where you can find the suitable subnet.

This is also an opportunity to set the MPI address that you have provided for this CPU. Please note that on older S7-300 CPUs, FMs or CPs with MPI connection automatically receive an MPI address derived from the CPU.

The highest MPI address must be greater than or equal to the highest MPI address assigned in the subnet (take account of automatic assignment of FMs and CPs!). It must have the same value for all nodes in the subnet.

Tip: if you have several stations with the same type of CPUs, assign different names (identifiers) to the CPUs in the different stations. They all have the name "CPUxxx(1)" as default so in the subnet they can only be differentiated by their MPI addresses. If you do not want to assign a name yourself, you can, for example, change the default identifier from "CPU xxx(1)" to "CPUxxx(n)" where "n" is equal to the MPI address.

When assigning the MPI address, please also take into account the possibility of connecting a programming device or operator panel (OP) to the MPI network at a later date for service or maintenance purposes. You should connect permanently installed programming devices or OPs direct to the MPI network; for plug-in devices via a spur line, there is an MPI connector with a heavy-gauge threaded-joint socket. Tip: reserve address 0 for a service programming device, address 1 for a service OP and address 2 for a replacement CPU (corresponds to the default addresses).

2.3.5 Monitoring and Modifying Modules

With the Hardware Configuration, you can carry out a wiring check of the machine or plant without the user program. A requirement for this is that the programming device is connected to a station (online) and the configuration has been saved, compiled and loaded into the CPU. Now you can address every digital and analog module. Select a module and then PLC → MONITOR/MODIFY, and set the Monitor and Modify operating modes and the trigger conditions.

With the "Status Value" button, the Hardware Configuration shows you the signal states or the values of the module channels. The "Modify Value" button writes the value specified in the Modify Value column to the module.

If the "I/O Display" checkbox is active, the peripheral inputs/outputs (module memory) are displayed instead of the inputs/outputs (process image). The "Enable Periph. Outputs" checkbox revokes the output disable of the output modules if the CPU is in STOP mode (see Chapter 2.7.5 "Enabling Peripheral Outputs").

You can find other methods of monitoring and modifying inputs and outputs in Chapters 2.7.3 "Monitoring and Modifying Variables" and 2.7.4 "Forcing Variables".

2.4 Configuring the Network

The basis for communications with SIMATIC is the networking of the S7 stations. The required objects are the subnets and the modules with communications capability in the stations. You can create new subnets and stations with the SIMATIC Manager within the project hierarchy. You then add the modules with communications capability (CPUs and CPs) using the Hardware Configuration tool; at the same time, you assign the communications interfaces of these modules to a subnet. You then define the communications relationships between these modules – the connections – with the Network Configuration tool in the connection table.

The Network Configuration tool allows graphical representation and documentation of the configured networks and their nodes. You can also create all necessary subnets and stations with the Network Configuration tool; then you assign the stations to the subnets and parameterize the node properties of the modules with communications capability.

You can proceed as follows to define the communications relationships via the networking configuration tool:

- ▷ Open the MPI subnet created as standard in the project container (if it is no longer available, simply create a new subnet with INSERT → SUBNET).
- ▷ Use the Network Configuration tool to create the necessary stations and – if required – further subnets.
- ▷ Open the stations and provide them with the modules with communications capability.
- ▷ Connect the modules with the relevant subnets.
- ▷ Adapt the network parameters, if necessary.
- ▷ Define the communication connections in the connection table, if required.

You can also configure global data communications within the Network Configuration: select the MPI subnet and then select OPTIONS → DEFINE GLOBAL DATA (see Chapter 20.5 "Global Data Communication").

NETWORK → SAVE saves an incomplete Network Configuration. You can check the consistency of a Network Configuration with NETWORK → CONSISTENCY CHECK. You close the Network Configuration with NETWORK → SAVE AND COMPILE.

Network window

To start the Network Configuration, you must have created a project. Together with the project, the SIMATIC Manager automatically creates an MPI subnet.

A double-click on this or any other subnet starts the Network Configuration. You can also reach the Network Configuration if you open the Connections object in the CPU container.

In the upper section, the Network Configuration window shows all previously created subnets and stations (nodes) in the project with the configured connections (Figure 2.5).

The connection table is displayed in the lower section of the window if a module with "communications capability", e.g. an S7-400 CPU, is selected in the upper section of the window.

A second window displays the network object catalog with a selection of the available SIMATIC stations, subnets and DP stations. You can fade the catalog in and out with VIEW → CATALOG and you can "dock" it onto the right edge of the network window (double-click on the title bar). With VIEW → ZOOM IN, VIEW → ZOOM OUT and VIEW → ZOOM FACTOR..., you can adjust the clarity of the graphical representation.

2.4.1 Configuring the Network View

Selecting and arranging the components

You begin the Network Configuration by selecting a subnet that you select in the catalog with the mouse, hold and drag to the network window. The subnet is represented in the window as a horizontal line. Impermissible positions are indicated with a "prohibited" sign on the mouse pointer.

You proceed in the same way for the desired stations, at first without connection to the sub-

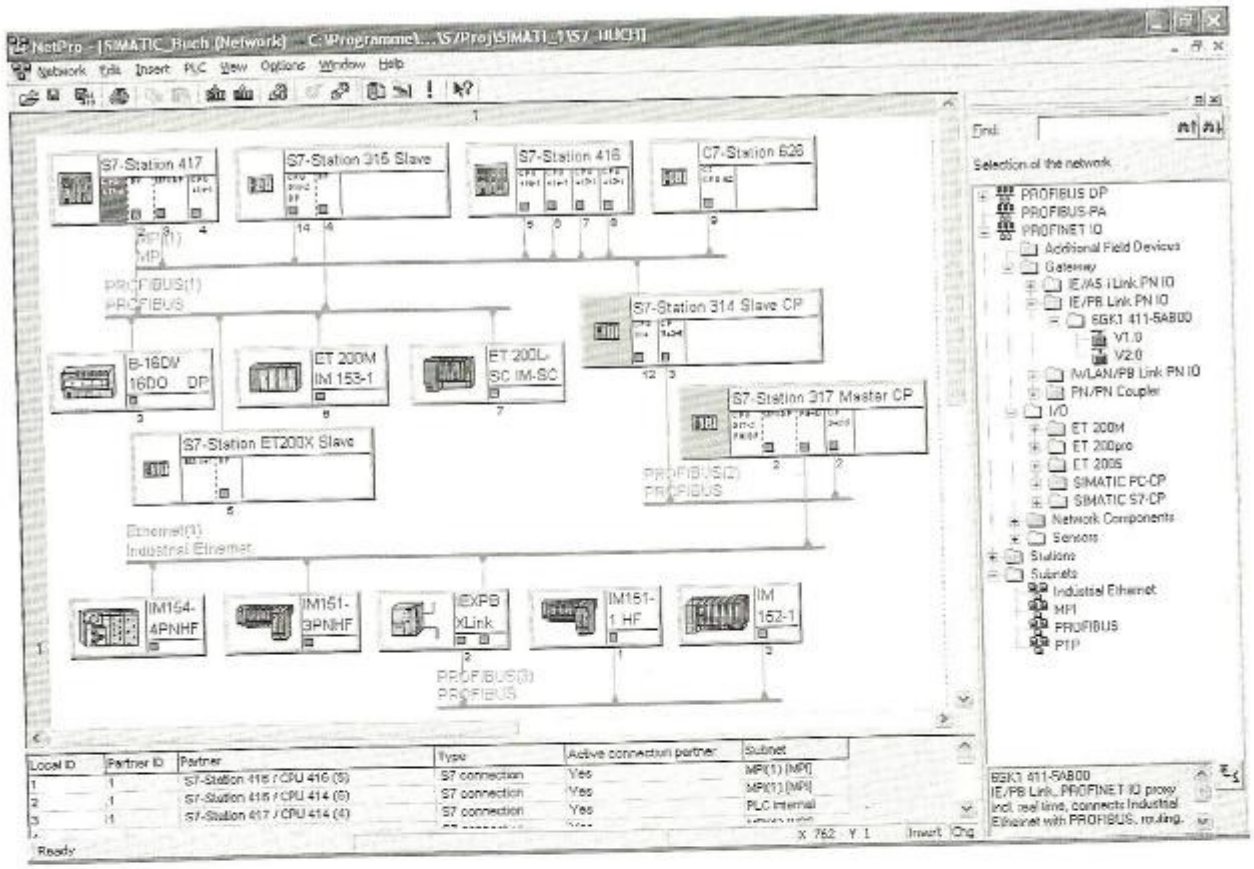


Figure 2.5 Network Configuration Example

net. The stations are still "empty". A double-click on a station opens the Hardware Configuration tool allowing you to configure the station or at least the module(s) with network connection. Save the station and return to the Network Configuration.

The interface of a module with communications capability is represented in the Network Configuration as a small box under the module view. Click on this box, hold and drag it to the relevant subnet. The connection to the subnet is represented as a vertical line.

Proceed in exactly the same way with all other nodes.

You can move created subnets and stations in the network window. In this way, you can also represent your hardware configuration visually. Under certain circumstances, you get a clearer and more compact arrangement if you reduce represented subnet lengths with VIEW → REDUCED SUBNET LENGTHS.

Setting communications properties

After creating the graphical view, you parameterize the subnets: select the subnets and then EDIT → OBJECT PROPERTIES. The properties window that then appears includes the S7 subnet ID in the "General" tab. The ID consists of two hexadecimal numbers, the project number and the subnet number. You require this S7 subnet ID if you want to go online with the programming device without a suitable project in order to reach other nodes via the subnet. You set the network properties in the "Network Settings" tab, e.g. the data transfer rate or the highest node address.

When you select the network connection of a node, you can define the network properties of the node with EDIT → OBJECT PROPERTIES, e.g., the node address and the subnet it is connected to, or you can create a new subnet.

On the "Interfaces" tab of the station properties, you can see an overview of all modules with

communications capability, with the node addresses and the subnet types used.

You define the module properties of the nodes in a similar way (with the same operator inputs as in the Hardware Configuration tool).

2.4.2 Configuring a Distributed I/O with the Network Configuration

You can also use the Network Configuration to configure the distributed I/O with PROFIBUS DP or PROFINET IO. Select VIEW → WITH DP SLAVES/IO DEVICES to display or fade out DP slaves and IO devices in the network view.

PROFIBUS DP

You require the following in order to configure a DP master system:

- A PROFIBUS subnet (if not already available, drag the PROFIBUS subnet from the network object catalog to the network window),
- A DP master in a station (if not already available, drag the station from the network object catalog to the network window, open the station and select a DP master with the Hardware Configuration tool, either integrated in the CPU or as an autonomous module),
- The connection from the DP master to the PROFIBUS subnet (either select the subnet in the Hardware Configuration tool or click on the network connection to the DP master in the Network Configuration, "hold" and drag to the PROFIBUS network).

In the network window, select the DP master to which the slave is to be assigned. Find the DP slave in the network object catalog under "PROFIBUS DP" and the relevant sub-catalog, drag it to the network window and fill out the properties window that appears.

You parameterize the DP slave by selecting it and then selecting EDIT → OPEN OBJECT. The Hardware Configuration is started. Now you can set the user data addresses or, in the case of modular slaves, select the I/O modules (see Chapter 2.3 "Configuring Stations").

You can only connect an intelligent DP slave to a subnet if you have previously created it (see Chapter 20.4.2 "Configuring PROFIBUS DP"). In the network object catalog, you can find the type of intelligent DP slave under "Already created stations"; drag it, with the DP master selected, to the network window and fill out the properties window that then appears (as in the Hardware Configuration tool).

With VIEW → HIGHLIGHT → MASTER SYSTEM, you emphasize the assignment of the nodes of a DP master system; first, you select the master or a slave of this master system. With VIEW → REARRANGE, the DP slaves are assigned optically to their DP master.

PROFINET IO

In order to configure a PROFINET IO system, you require:

- An Industrial Ethernet subnet (if not already available, drag the Industrial Ethernet subnet from the network object catalog to the network window)
- An IO controller in a station (if not already available, drag the station from the network object catalog to the network window, open the station, and select an IO controller with the Hardware Configuration tool, either integrated in the CPU or as an autonomous module)
- The connection from the IO controller to the Industrial Ethernet subnet (either already select the subnet in the Hardware Configuration tool, or click on the network connection to the IO controller in the Network Configuration, "hold" and drag to the Industrial Ethernet network

In the network window, select the IO controller to which the IO device is to be assigned. Find the IO device in the network object catalog under "PROFINET IO" and the relevant sub-catalog, drag it to the network window and fill out the properties window that appears.

You parameterize the IO device by selecting it and then selecting EDIT → OPEN OBJECT. The Hardware Configuration is started. Now you can set the user data addresses or the I/O modules (see Chapter 2.3 "Configuring Stations").

With VIEW → HIGHLIGHT → PROFINET IO SYSTEM, you emphasize the assignment of the nodes of a PROFINET IO system; first, you select the IO controller or an IO device. With VIEW → REARRANGE, the IO devices are assigned optically to their IO controller.

2.4.3 Configuring Connections

Connections describe the communications relationships between two devices. Connections must be configured if

- you want to establish S7 communications between two SIMATIC S7 devices ("Communication via configured connections") or
- the communications partner is not a SIMATIC S7 device.

Note: you do not require a configured connection for direct online connection of a programming device to the MPI network for programming or debugging. If you want to reach other nodes arranged in other connected subnets with the programming device, you must configure the connection of the programming device: in the Network Object Catalog, select the *PG/PC* object under *Stations* by double-clicking, open *PG/PC* in the network window by double-clicking, and select the interface and assign it to a subnet.

Connection table

The communications connections are configured in the connection table. Requirement: you have created a project with all stations that are to exchange data with each other, and you have assigned the modules with communications capability to a subnet.

The object *Connections* in the *CPU* container represents the connection table. A double-click on Connections starts the Network Configuration in the same way as a double-click on a subnet in the project container.

To configure the connections, select e.g. an S7-400 CPU in the Network Configuration. In the lower section of the network window, you get the connection table (Table 2.1; if it is not visible, place the mouse pointer on the lower edge of the window until it changes shape and then drag the window edge up). You enter a new communication connection with INSERT → NEW CONNECTION or by double-clicking on an empty line.

You create a connection for each "active" CPU. Please note that you cannot create a connection table for an S7-300 CPU; S7-300 CPUs can only be "passive" partners in an S7 connection.

In the "New Connection" window, you select the communications partner in the "Station" and "Module" dialog boxes (Figure 2.6); the station and the module must already exist. You also determine the connection type in this window.

If you want to set more connection properties, activate the check box "Before inserting: display properties".

The connection table contains all data of the configured connections. To be able to display this clearly, use VIEW → OPTIMIZE COLUMN WIDTH and VIEW → DISPLAY COLUMNS and select the information you are interested in.

Connection ID

The number of possible connections is CPU-specific. STEP 7 defines a connection ID for every connection and for every partner. You require this specification when you use communications blocks in your program.

You can modify the **local ID** (the connection ID of the currently opened module). This is neces-

Table 2.1 Connection Table Example

Local ID	Partner ID	Partners	Type	Active connection partner	Send operating mode messages
1	1	Station 416 / CPU416(5)	S7 connection	yes	no
2	2	Station 416 / CPU416(5)	S7 connection	yes	no
3		Station 315 / CPU315(7)	S7 connection	yes	no
4	1	Station 417 / CPU414(4)	S7 connection	yes	no

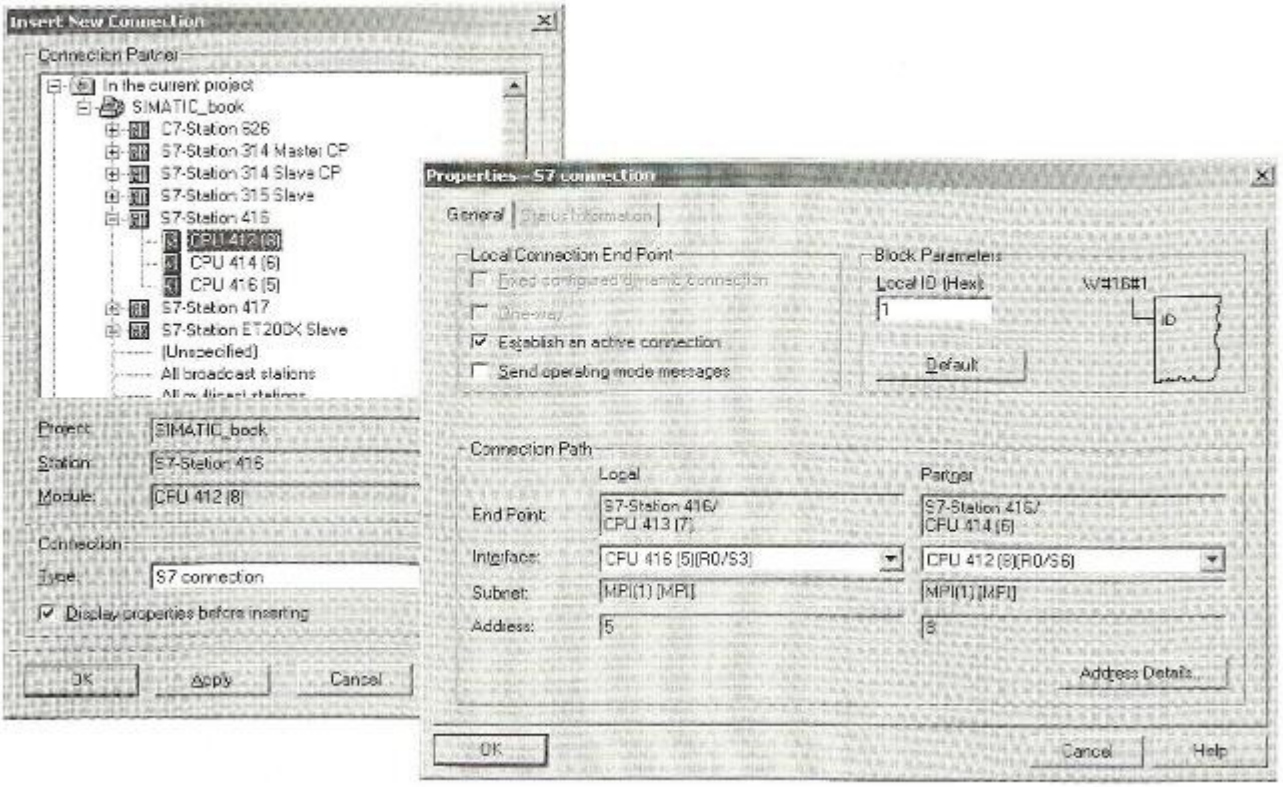


Figure 2.6 Configuring Communications Connections

sary if you have already programmed communications blocks and you want to use the local ID specified there for the connection.

You enter the new local ID as a hexadecimal number. It must be within the following value ranges, depending on the connection type, and must not already be assigned:

- Value range for S7 connections: 0001_{hex} to $0FFF_{hex}$
- Value range for S7 connections with loadable S7 communications (S7-300): 0001_{hex} to $008F_{hex}$
- Value range for PtP connections: 1000_{hex} to 1400_{hex}

You change the **partner ID** by going to the connection table of the partner CPU and changing (what is then) the local ID: select the connection line and then EDIT → OBJECT PROPERTIES. If STEP 7 does not enter a partner ID, it is a one-way connection (see below).

Partners

This column displays the connection partner. If you want to reserve a connection resource without naming a partner device, enter "unspecified" in the dialog box under Station.

In a **one-way connection**, communication can only be initiated from one partner; example: S7 communications between an S7-400 and S7-300 CPU. Even without S7 communications functions in the S7-300-CPU, data can be exchanged by an S7-400 CPU with SFB 14 GET and SFB 15 PUT. In the S7-300, no user program runs for this communication but the data exchange is handled by the operating system.

A one-way connection is configured in the connection table of the "active" CPU. Only then does STEP 7 assign a "Local ID". You also load this connection only in the local station.

With a **two-way connection**, both partners can assume communication actively; e.g. two S7-400 CPUs with the communications functions SFB 8 SEND and SFB 9 BRCV.

You configure a two-way connection only once for one of the two partners. STEP 7 then assigns a "Local ID" and a "Partner ID" and generates the connection data for both stations. You must load each partner with its own connection table.

Connection type

The STEP 7 Basic Package provides you with the following connection types in the Network Configuration:

PtP connection, approved for the subnet PTP (3964(R) and RK 512 procedures) with S7 communications. A PtP (point-to-point) connection is a serial connection between two partners. These can be two SIMATIC S7 devices with the relevant interfaces or CPs, or a SIMATIC S7 device and a non-Siemens device, e.g. a printer or a barcode reader.

S7 connection, approved for the subnets MPI, PROFIBUS and Industrial Ethernet with S7 communication. An S7 connection is the connection between SIMATIC S7 devices and can include programming devices and human machine interface devices. Data are exchanged via the S7 connection, or programming and control functions are executed.

Fault-tolerant S7 connection, approved for the subnets PROFIBUS and Industrial Ethernet with S7 communications. A fault-tolerant S7 connection is made between fault-tolerant SIMATIC S7 devices and it can also be established to an appropriately equipped PC.

The software component "SIMATIC NCM", which is part of STEP 7, is available for **parameterizing CPs**. You have connection types available for selection: FMS connection, FDL connection, ISO transport connection, ISO-on-TCP connection, TCP connection, UDP connection and e-mail connection.

Active connection buildup

Prior to the actual data transfer, the connection must be built up (initialized). If the connection partners have this capability, you specify here which device is to establish the connection. You do this with the check box "Establish an active connection" in the properties window of the connection (select the connection and then EDIT → OBJECT PROPERTIES).

Sending operating state messages

Connection partners with a configured two-way connection can exchange operating state messages. If the local node is to send its operating state messages, activate the relevant check box in the properties window of the connection. In the user program of the partner CPU, these messages can be received with SFB 23 USTATUS.

Connection path

As the connection path, the properties window of the connection displays the end points of the connection and the subnets over which the connection runs. If there are several subnets for selection, STEP 7 selects them in the order Industrial Ethernet before Industrial Ethernet/ TCP-IP before MPI before PROFIBUS.

The station and the CPU over which the connection runs are displayed as the end points of the connection. The modules with communications capability are listed under "Interface", specifying the rack number and the slot. If both CPUs are located in the same rack (e.g. S7-400 CPUs in multiprocessor mode), the display box shows "PLC-internal".

You will then see the MPI address or PROFIBUS address of the node under "Subnet" and "Address".

Connections between projects

For data exchange between two S7 modules belonging to different SIMATIC projects, you enter "unspecified" for connection partner in the connection table (in the local station in both projects).

Please ensure that the connection data agree in both projects (STEP 7 does not check this). After saving and compiling, you load the connection data into the local station in each project.

If a project is to subsequently become part of a multiproject, and if the connection partner is also within a project of the multiproject, select "In unknown project" as the connection partner, and enter an unambiguous connection name (reference) in the properties window.

Connection to non-S7 stations

Within a project, you can also specify stations other than S7 stations as connection partners:

- Other stations (non-Siemens devices and also S7 stations in another project)
- Programming devices/PCs
- SIMATIC S5 stations

A requirement for configuring the connection is that the non-S7 station exists as an object in the project container and you have connected the non-S7 station to the relevant subnet in the station properties (e.g. select the station in the Network Configuration, select EDIT → OBJECT PROPERTIES and connect the station with the desired subnet on the "Interfaces" tab).

2.4.4 Gateways

If the programming device is connected to a subnet, it can reach all other nodes on this subnet. For example, from one connection point, you can program and debug all S7 stations connected to an MPI network. If another subnet such as a PROFIBUS subnet is connected to an S7 station, the programming device can also reach the stations on the other subnet. The requirement for this is that the station with the subnet transition has routing capability, that is, it will channel the transferred message frames.

When the network configuration is compiled, routing tables containing all the necessary information are automatically generated for the stations with subnet transitions. All accessible communications partners must be configured in a plant network within an S7 project and must be supplied with the "knowledge" of which stations can be reached via which subnets and subnet transitions.

If you want to reach all nodes in a subnet with a programming device from one connection point, you must configure the connection point. You enter a "placeholder", a PG/PC station from the Network Object Catalog in the network configuration at the relevant subnet. You configure a PG/PC station on every subnet to which you want to connect a programming device.

During operation, you connect the programming device to the subnet and select PLC → ASSIGN PG/PC. This adapts the interfaces of the programming device to the configured settings for the subnet. Before disconnecting the programming device again from the subnet, select PLC → CANCEL PG/PC ASSIGNMENT.

If you go online with a programming device that does not contain the right project, you require the S7 subnet ID for network access. The S7 subnet ID comprises two numbers: the project number and the subnet number. You can obtain the subnet ID in the network configuration by selecting the subnet and then EDIT → OBJECT PROPERTIES on the "General" tab.

2.4.5 Loading the Connection Data

To activate the connections, you must load the connection table into the PLC following saving and compiling (all connection tables into all "active" CPUs).

Requirement: You are in the network window and the connection table is visible. The programming device is a node of the subnet over which the connection data are to be loaded into the modules with communications capability. All subnet nodes have been assigned unique node addresses. The modules to which connection data are to be transferred are in the STOP mode.

With PLC → DOWNLOAD TO CURRENT PROJECT → ..., you transfer the connection and configuration data to the accessible modules. Depending on which object is selected and which menu command is selected, you can choose between the following

→ SELECTED STATIONS

→ SELECTED AND PARTNER STATIONS

→ STATIONS ON THE SUBNET

→ SELECTED CONNECTIONS

→ CONNECTIONS AND GATEWAYS

In order to delete all connections of a programmable module, load an empty connection table into the associated module.

The compiled connection data are also a component part of the *System data* in the *Blocks* container. Transfer of the system data and the subsequent startup of the CPUs effectively also

transfers the connection data to the modules with communications capability.

For online operation via MPI, a programming device requires no additional hardware. If you connect a PC to a network or if you connect a programming device to an Ethernet or PROFIBUS network, you require the relevant interface module. You parameterize the module with the application "Setting the PG/PC Interface" in the Windows Control Panel.

2.4.6 Matching Projects in a Multiproject

When opening a multiproject with the Network Configuration tool, a window is displayed with the projects present in the multiproject. You also obtain this window if you open a project included in a multiproject and select VIEW → MULTIPROJECT. The window displays the projects present in the multiproject and the cross-project subnets which have already been combined. Select a project for further processing by double clicking (Figure 2.7).

Projects usually contain communications connections between the individual stations. If projects are combined into a multiproject, or if an existing project is included into the multiproject, these connections can be combined and matched.

If you select VIEW → CROSS-PROJECT NETWORK VIEW in an opened project that belongs to a multiproject, you will see an overview of all stations of the multiproject and the current connections. In the cross-project network view, you cannot make any changes to the projects. Selecting VIEW → CROSS-PROJECT NETWORK VIEW again exits the multiproject view.

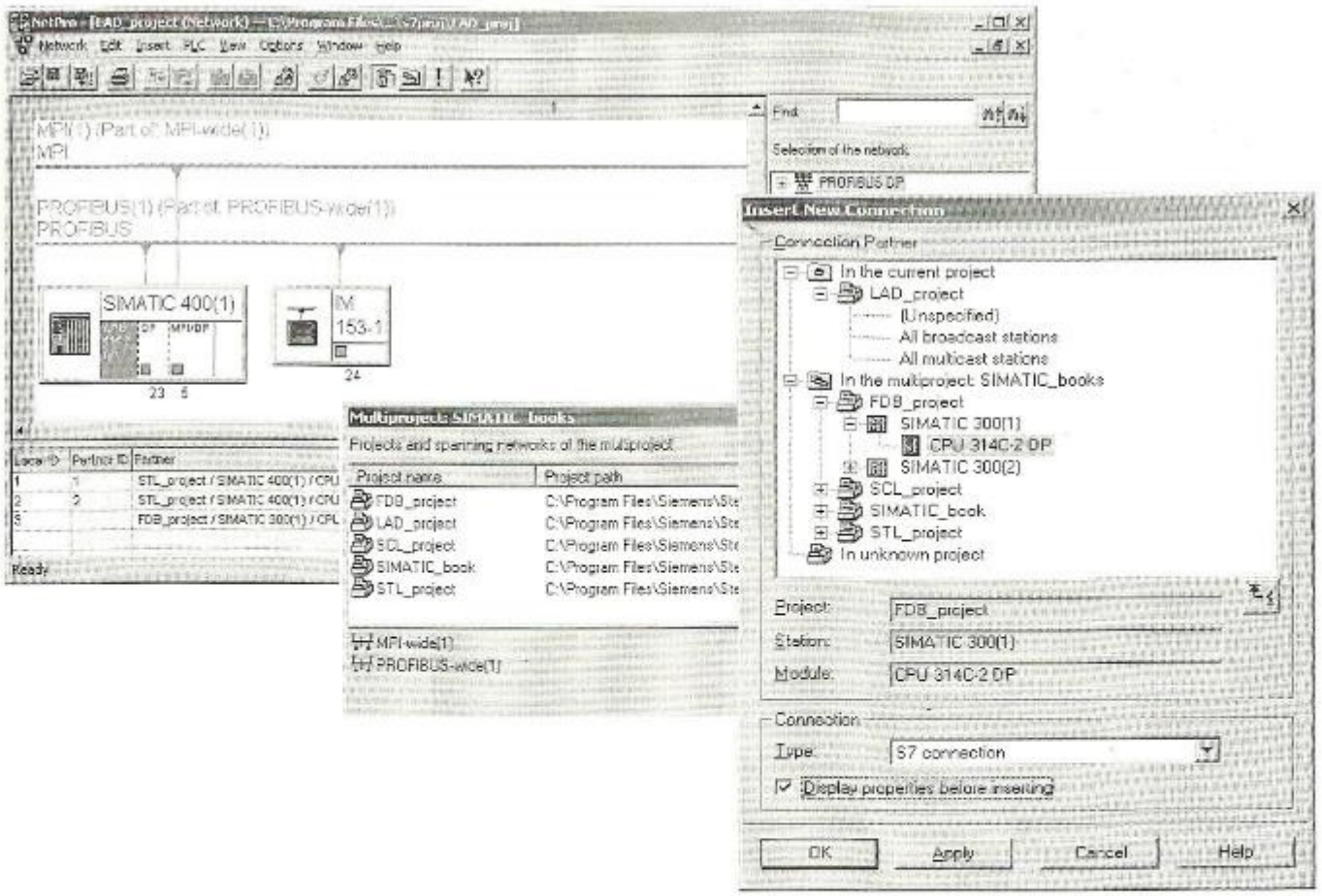


Figure 2.7 Multiproject in the Network Configuration tool

Combining subnets

The MPI, PROFIBUS and Industrial Ethernet subnets are combined together first. A prerequisite is that the subnets to be combined have the same subnet ID. With the subnet selected, you can set these in the Network Configuration tool using EDIT → OBJECT PROPERTIES.

With FILE → MULTIPROJECT → ADJUST PROJECTS you can call a wizard for an open multiproject in the SIMATIC Manager which supports you when matching. In the Network Configuration tool, you obtain the dialog window with EDIT → MERGE/UNMERGE SUBNETWORKS → ...

You select the type of subnet, click the "Execute" button, and obtain the subnets of the selected type present in the multiproject. You can now select individual subnets of the projects, and combine in a cross-project subnet. You can use the same dialog to eliminate subnets from the cross-project subnet.

Several cross-project subnets of the same type can be created in a multiproject. The properties of the cross-project subnet are determined by the first subnet added or by the subnet selected with the "Select" button. Use "OK" or "Apply" to acknowledge the settings. Subnets which are part of a cross-project subnet are identified by a different symbol in the SIMATIC Manager.

Combining connections

The connections configured in single projects which lead to a partner in another project can be combined in a multiproject. If you select the partner "In unknown project" when configuring connections in a single project in the window "Insert new connection", you can subsequently enter a connection name (reference) in the window "Properties - S7 connection". Connections in different projects with the same connection names can be combined automatically.

In the SIMATIC Manager, this is carried out by the wizard for project matching if you click "Combine connections" and "Execute". Connections are then combined which have identical connection names (reference).

In the Network Configuration tool, you can also combine connections with "unspecified" partners. Select EDIT → MERGE CONNECTIONS to obtain a dialog box with all configured connections. Select one connection in each of the Windows "Connections without connection partner" and "Possible connection partners" and click "Assign". The assigned connections are listed in the bottom window "Assigned connections". Use "Merge" to then combine the connections. The connections are assigned the properties of the local module of the currently opened project. You can modify the connection properties when combining.

Configuring cross-project connections

Following the combination of subnets, cross-project connections can be configured. The procedure is the same as for project-internal connections, extended by specification of the project at the connection partner.

You can test a network configuration in the multiproject for contradictions with NETWORK → CHECK CROSS-PROJECT CONSISTENCY.

2.5 Creating the S7 Program

2.5.1 Introduction

The user program is created under the object *S7 Program*. You can assign this object in the project hierarchy of a CPU or you can create it independently of a CPU. It contains the object *Symbols* and the containers *Sources* and *Blocks* (Figure 2.8).

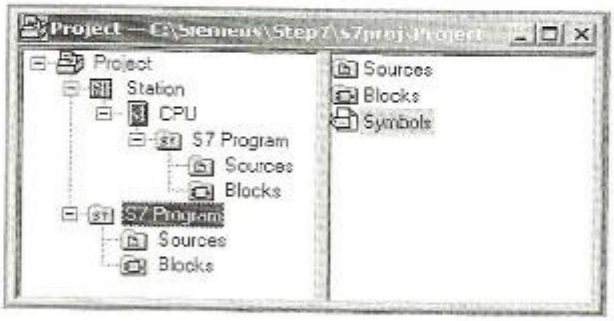


Figure 2.8 Objects for Programming

With **incremental** program creation, you enter the program direct block-by-block. Entries are checked immediately for syntax. At the same time, the block is compiled as it is saved and then stored in the container *Blocks*. With incremental programming, you can also edit blocks online in the CPU, even during operation. Incremental programming is possible in all the basic languages.

In the case of **source-oriented** program creation, you write one or more program sources and store these in the container *Sources*. Program sources are ASCII text files that contain the program statements for one or more blocks, possibly even for the entire program. You compile these sources and you get the compiled blocks in the container *Blocks*. Source-oriented program creation is used in STL and SCL; you cannot use source-oriented programming with LAD or FBD, but programs created with LAD or FBD can be stored as source files.

The signal states or the values of addresses are processed in the program. An address is, for example, the input I1.0 (*absolute addressing*). With the help of the **Symbol Table** under the object *Symbols*, you can assign a symbol (an alphanumeric name, e.g. "Switch motor on") to an address and then access it with this name (*symbolic addressing*). In the properties of the offline object container *Blocks*, you specify whether in the event of a change in the Symbol Table the absolute address or the symbol is to be definitive for the already compiled blocks when next saved (*address priority*).

2.5.2 Symbol Table

In the control program, you work with addresses; these are inputs, outputs, timers, blocks. You can assign absolute addresses (e.g. I1.0) or symbolic addresses (e.g. Start signal). Symbolic addressing uses names instead of the absolute address. You can make your program easier to read by using meaningful names.

In symbolic addressing, a distinction is made between *local* symbols and *global* symbols. A local symbol is known only in the block in which it has been defined. You can use the same local symbols in different blocks for different purposes. A global symbol is known throughout the entire program and has the same meaning in all blocks. You define global symbols in the symbol table (object *Symbols* in the container *S7 Program*).

A global symbol starts with an alpha character and can be up to 24 characters long. A global symbol can also contain spaces, special characters and national characters such as the umlaut. Exceptions to this are the characters 00_{hex}, FF_{hex} and the inverted commas ("). You must enclose symbols with special characters in inverted commas when programming. In the compiled block, the program editor displays all global symbols in inverted commas. The symbol comment can be up to 80 characters long.

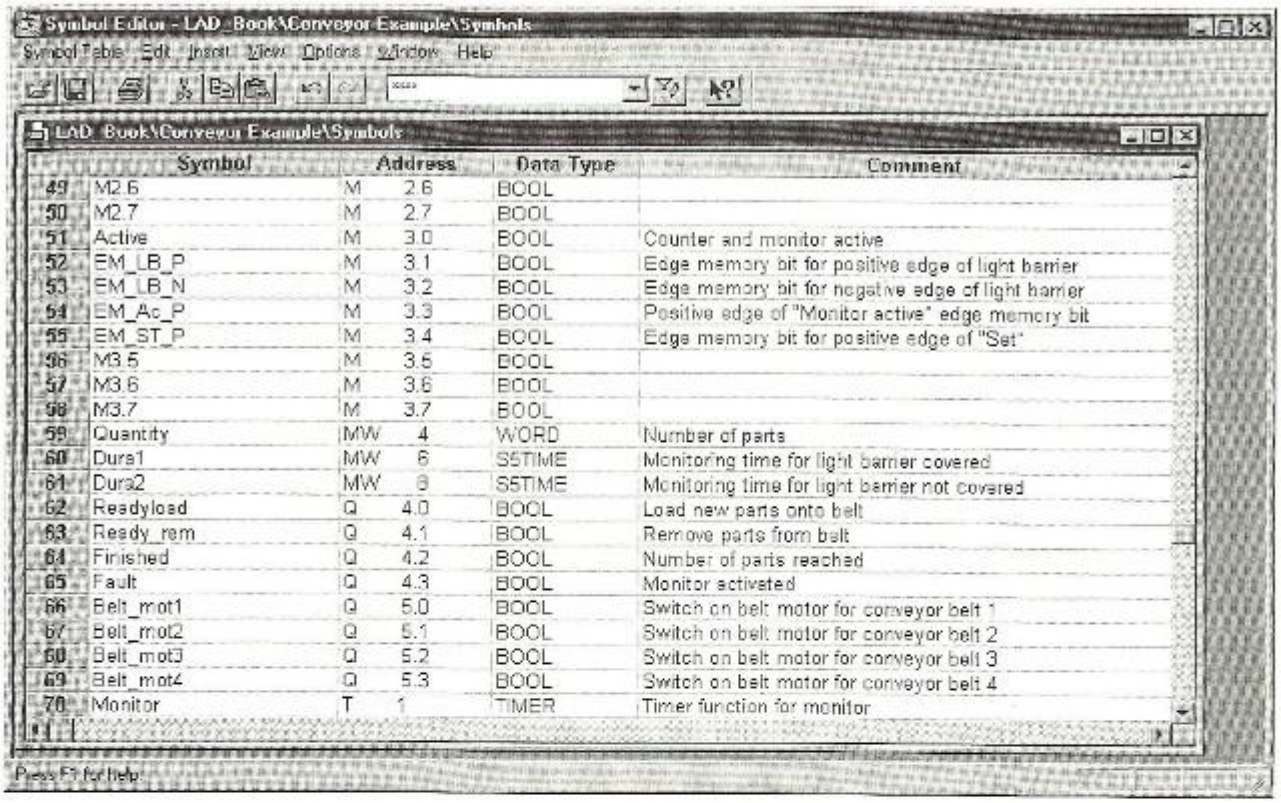
In the symbol table you can assign names to the following addresses and objects:

- Inputs I, outputs Q, peripheral inputs PI and peripheral outputs PQ
- Memory bits M, timer functions T and counter functions C
- Code blocks OBs, FBs, FCs, SFCs, SFBs and data blocks DBs
- User data types UDTs
- Variable table VAT

Data addresses in the data blocks are included among the local addresses; the associated symbols are defined in the declaration section of the data block in the case of global data blocks and in the declaration section of the function block in the case of instance data blocks.

When creating an S7 program, the SIMATIC Manager also creates an empty symbol table *Symbols*. You open this and can then define the global symbols and assign them to absolute addresses (Figure 2.9). There can be only one single symbol table in an S7 program.

The data type is part of the definition of a symbol. It defines specific properties of the data behind the symbol, essentially the representation of the data contents. For example, the data type BOOL identifies a binary variable and the data type INT designates a digital variable whose contents represent a 16-bit integer. Please refer to Chapter 3.5 "Variables, Constants and Data Types", it contains a detailed description of the data types.



	Symbol	Address	Data Type	Comment
49	M2.6	M 2.6	BOOL	
50	M2.7	M 2.7	BOOL	
51	Active	M 3.0	BOOL	Counter and monitor active
52	EM_LB_P	M 3.1	BOOL	Edge memory bit for positive edge of light barrier
53	EM_LB_N	M 3.2	BOOL	Edge memory bit for negative edge of light barrier
54	EM_Ac_P	M 3.3	BOOL	Positive edge of "Monitor active" edge memory bit
55	EM_ST_P	M 3.4	BOOL	Edge memory bit for positive edge of "Set"
56	M3.5	M 3.5	BOOL	
57	M3.6	M 3.6	BOOL	
58	M3.7	M 3.7	BOOL	
59	Quantity	MW 4	WORD	Number of parts
60	Dura1	MW 6	S5TIME	Monitoring time for light barrier covered
61	Dura2	MW 8	S5TIME	Monitoring time for light barrier not covered
62	Readyload	Q 4.0	BOOL	Load new parts onto belt
63	Ready_rem	Q 4.1	BOOL	Remove parts from belt
64	Finished	Q 4.2	BOOL	Number of parts reached
65	Fault	Q 4.3	BOOL	Monitor activated
66	Belt_mot1	Q 5.0	BOOL	Switch on belt motor for conveyor belt 1
67	Belt_mot2	Q 5.1	BOOL	Switch on belt motor for conveyor belt 2
68	Belt_mot3	Q 5.2	BOOL	Switch on belt motor for conveyor belt 3
69	Belt_mot4	Q 5.3	BOOL	Switch on belt motor for conveyor belt 4
70	Monitor	T 1	TIMER	Timer function for monitor

Figure 2.9 Symbol Table Example

With incremental programming, you create the symbol table before entering the program; you can also add or correct individual symbols during program input. In the case of source-oriented programming, the complete symbol table must be available when the program source is compiled.

Importing, exporting

Symbol tables can be imported and exported. "Exported" means a file is created with the contents of your symbol table. You can select here either the entire symbol table, a subset limited by filters or only selected lines. For the data format you can choose between pure ASCII text (extension *.asc), sequential assignment list (*.seq), System Data Format (*.sdf for Microsoft Access) and Data Interchange Format (*.dif for Microsoft Excel). You can edit the exported file with a suitable editor. You can also import a symbol table available in one of the formats named above.

Special object properties

With EDIT → SPECIAL OBJECT PROPERTIES → ..., you set attributes for each symbol in the symbol table. These attributes or properties are used in the following:

- ▷ Process monitoring with S7-PDIAG
- ▷ Human machine interface functions for monitoring with WinCC
- ▷ Configuring messages
- ▷ Configuring communications using the NCM software
- ▷ Control at contact with inputs and bit memories in the program editor

VIEW → COLUMNS R, O, M, C, CC makes the settings visible. With OPTIONS → CUSTOMIZE, you can specify whether or not the special object properties are to be copied and you can define behavior when importing symbols.

2.5.3 Program Editor

For creating the user program, the STEP 7 Basic Package contains a program editor for the LAD, FBD and STL programming languages. You program incrementally with LAD and FBD, that is, you enter an executable block direct; Figure 2.10 shows the possible actions for this.

If you use symbolic addressing for global addresses, the symbols must already be assigned to an absolute address in the case of incremental programming; however, you can enter new symbols or change symbols during program input.

LAD/FBD blocks can be "decompiled", i.e. a readable block can be created again from the MC7 code without an offline database (you can read any block from a CPU using a programming device without the associated project). In addition, an STL program source can be created from any compiled block.

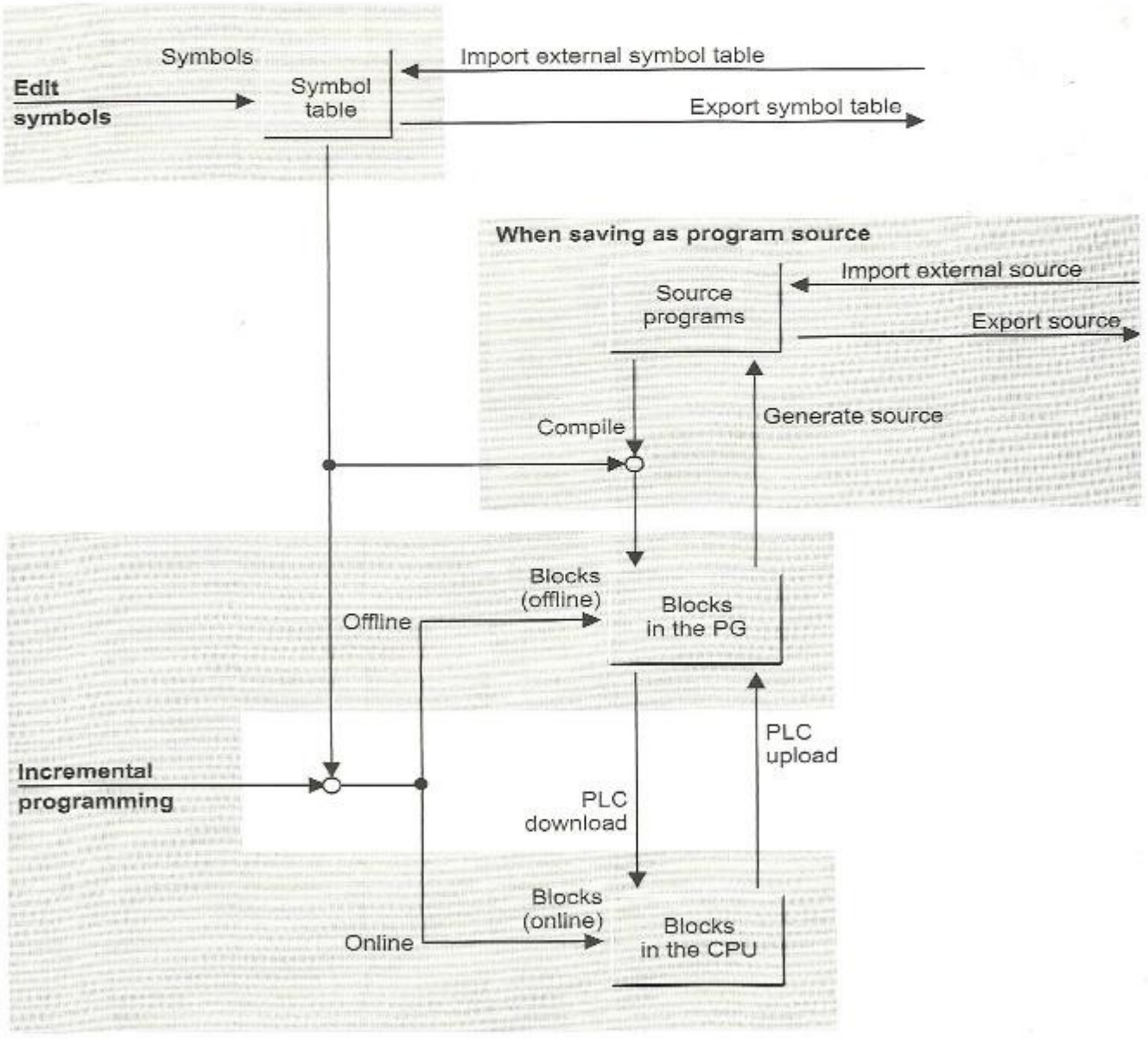


Figure 2.10 Writing Programs with the LAD/FBD Editor

Starting the program editor

You reach the program editor when you open a block in the SIMATIC Manager, e.g. by double-clicking on the automatically generated symbol of the organization block OB 1, or via the Windows taskbar with START → SIMATIC → STEP 7 → LAD, STL, FBD – PROGRAMMING S7 BLOCKS.

You can customize the properties of the program editor with OPTIONS → CUSTOMIZE. On the "Editor" tab, select the properties with which a new block is to be generated and displayed, such as the creation language, preselection for comments, and symbols.

Program editor window

Further windows can be displayed within the window of the program editor: the block window, the *Details* and *Overviews* windows, and the window with the AS registers (Figure 2.11).

The *block window* is automatically displayed when opening a block and contains the block interface at the top, i.e. the block parameters as well as the static and dynamic local data. You can program the block in the bottom program area. The block window and the contents are described in Chapter 3.3.2 "Block Window".

The *Overviews* window shows the program elements and the call structure. If it is not visible, display it on the screen using VIEW → OVERVIEWS.

The *Details* window can be displayed or suppressed using VIEW → DETAILS. It contains the following tabs:

- 1: Error
  Contains the errors found in the block by the program editor, e.g. following compilation. With OPTIONS → CUSTOMIZE in the "Sources" tab you can set whether warnings are also to be displayed.

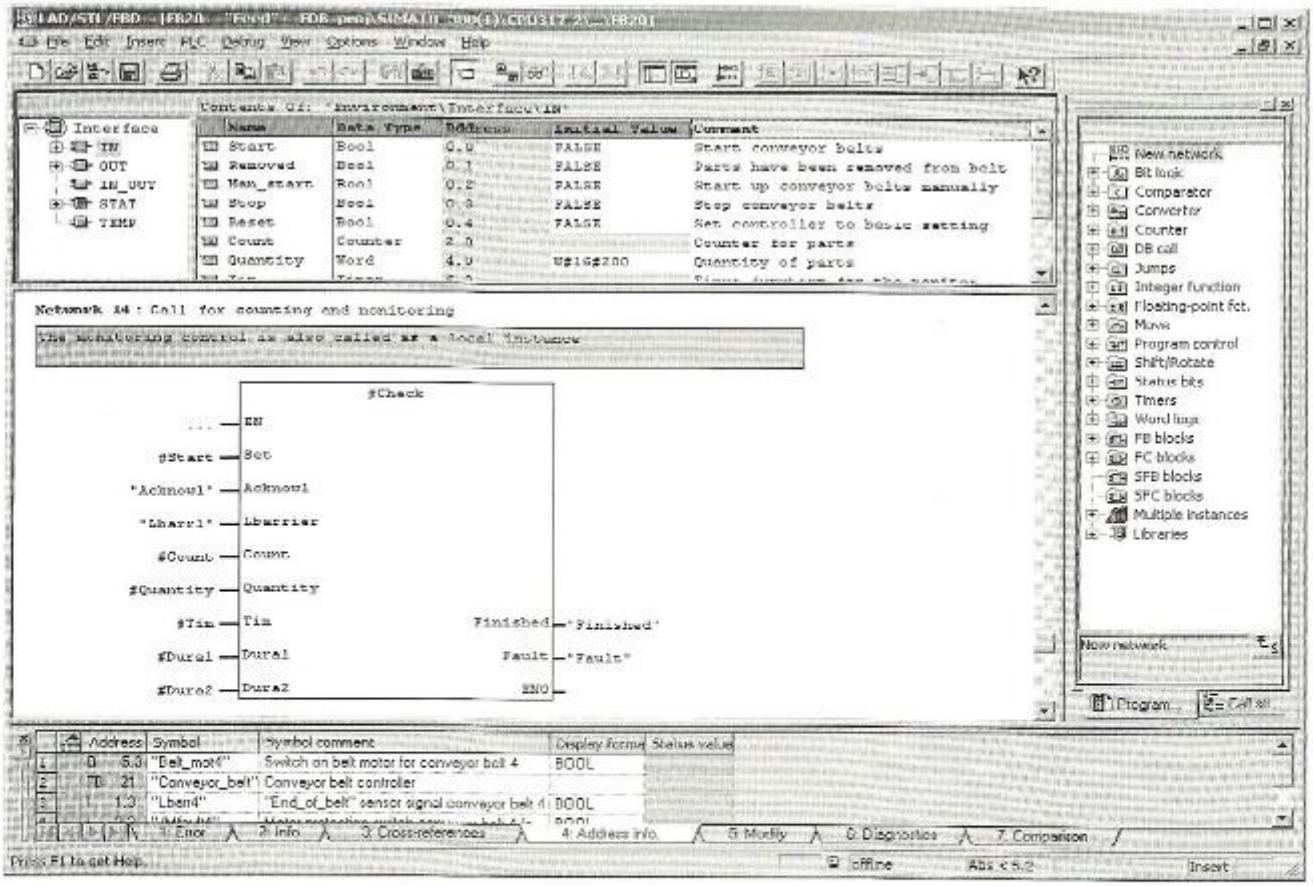


Figure 2.11 Example of editor window

- 2: Info
  Contains information on the currently selected address.
- 3: Cross-references
  Contains the references of addresses present in the current network (see Chapter 2.5.6 "Reference Data").
- 4: Address info
  Contains the symbol information of the addresses present in the current network (see Chapter 2.5.2 "Symbol Table"). You can edit existing symbols here, and new ones, and observe the address status.
- 5: Modify
  Contains an empty table of variables in which you can enter the addresses to be controlled (see Chapter 2.7.3 "Monitoring and Modifying Variables").
- 6: Diagnostics
  Contains a list with the existing monitoring functions for process diagnostics using the S7-PDIAG options package.
- 7: Comparison
  Contains the results of a previously implemented block comparison (see Chapter "Comparing blocks").

You can dock or undock the *Overviews* and *Details* windows at the edge of the editor window by double clicking the respective title bar.

The *PLC register contents* window shows the contents of the CPU registers (accumulators, address registers and DB registers).

Incremental programming

With incremental programming, you edit the blocks both in the offline and online *Blocks* container. The editor checks your entries in the incremental mode as soon as you have terminated a network. When the block is closed it is immediately compiled, so that only error-free blocks can be saved.

On the "Block" tab under OPTIONS → CUSTOMIZE, you set automatic updating of the reference data when saving a block.

The blocks can be edited both offline in the programming device's database and online in the CPU, generally referred to as the "programmable controller", or "PLC". For this purpose, the SIMATIC Manager provides an offline and an online window; the one is distinguished from the other by the labeling in the title bar.

In the offline window, you edit the blocks right in the PG database. If you are in the editor, you can store a modified block in the offline database with FILE → SAVE and transfer it to the CPU with PLC → DOWNLOAD. If you want to save the opened block under another number or in a different project, or if you want to transfer it to a library or to another CPU, use the menu command FILE → SAVE AS.

With the menu command FILE → STORE READ-ONLY... in the program editor, you can save a write-protected copy of the currently opened (and saved) block in another block container.

To edit a block in the CPU, open that block in the online window. This transfers the block from the CPU to the programming device so that it can be edited. You can write the edited block back to the CPU with PLC → DOWNLOAD. If the CPU is in RUN mode, the CPU will process the edited block in the next program scan cycle. If you want to save a block that you edited online in the offline database as well, you can do so with FILE → SAVE.

With incremental programming, you can execute all programming functions with one exception: if you want to provide block protection (KNOW_HOW_PROTECT), you can only do this via a program source file (see Chapter 24.1 "Block Protection" for more detailed information on this topic).

Chapter 2.6.4 "Loading the User Program into the CPU" and Chapter 2.6.5 "Block Handling" contain further information on online programming. Chapter 3.3 "Programming Code Blocks" and Chapter 3.4 "Programming Data Blocks" show you how to enter a LAD/FBD block.

Decompilation

When the program editor opens a compiled block, it carries out a "decompilation" into the LAD/FBD representation. It uses the program components not relevant to execution in the PG data management in order to display e.g. symbols, comments and jump labels. If the information from the PG data management is missing

during the decompilation, the editor uses replacement symbols.

Networks which cannot be decompiled in LAD or FBD are displayed in STL.

Updating or generating source files

On the "Sources" tab under OPTIONS → CUSTOMIZE, you can select the option "Generate source automatically" so that when you save an (incrementally created) block, the program source file is updated or created, if it does not already exist.

You can derive the name of a new source file from the absolute address or the symbolic address. The addresses can be transferred in absolute or symbolic form to the source file.

With the "Execute" button, you select, in the subsequent dialog box, the blocks from which you want to generate a program source file.

With FILE → GENERATE SOURCE you can produce ASCII source files from compiled blocks. First insert a *Sources* container under the object *S7 program*. When generating the source, first enter the storage location and the name of the source in the displayed dialog box, and subsequently select the blocks.

You can export source files from the project by selecting EDIT → EXPORT SOURCE in the SIMATIC Manager. You can then further process these ASCII files with another text editor, for example. Source files can also be imported back into the *Sources* container with INSERT → EXTERNAL SOURCE.

If you generate a source file from a block that you have created with LAD or FBD, you can generate a LAD or FBD block again from this source file. You compile the source file by opening it in the SIMATIC Manager with a double-click and by then selecting FILE → COMPILE in the program editor. An STL block is created in the *Blocks* container. You open this block and switch to your usual representation with VIEW → LAD or VIEW → FBD. After saving, the block retains this property.

If you selected the setting "Addresses – Symbolic" when creating the source file, you require a complete symbol table for compiling the source file. In this way, you can specify different absolute addresses in the symbol table and, after compilation, you end up with a program with, for example, different inputs and outputs. This allows you to adapt the program to a different hardware configuration. For this purpose, it is best to store these source files (which are independent of the hardware addressing) in a library, for example.

Comparing blocks

The block comparison enables you to find the differences between two blocks. The blocks can be present in different projects, in different target systems (CPUs), or in one project and one target system.

Use the program editor to compare the opened block with the same block in the CPU or in the project by using OPTIONS → COMPARE ON-/OFFLINE PARTNER. The result is displayed in the detail area of the editor window in the tab "7: Comparison".

Mark the *Blocks* object in the SIMATIC Manager, or only the blocks to be compared, and select OPTIONS → COMPARE BLOCKS. The comparison is carried out either between the online and offline data management (ONLINE/offline) or between two projects (Path1/Path2). When comparing the complete program – which can also include tables of variables and user data types (UDTs) – you can incorporate the system data. When using "Execute code comparison", the program code of the blocks is compared in addition, even of blocks with different generation languages.

The comparison includes all data of a block, even its time stamp for program code and interface. If you wish to know whether the program code is identical independent of the block properties, compare the checksum of the block. To do this, select the "Details" button in the results window of the block comparison.

2.5.4 Rewiring

The *Rewiring* function allows you to replace addresses in individually compiled blocks or in the entire user program. For example, you can replace input bits I 0.0 to I 0.7 with input bits I 16.0 to I 16.7. Permissible addresses are inputs, outputs, memory bits, timers and counters as well as functions FCs and function blocks FBs.

In the SIMATIC Manager, you select the objects in which you wish to carry out the rewiring; select a single block, a group of blocks by holding Ctrl and clicking with the mouse, or the entire *Blocks* user program. OPTIONS → REWIRE takes you to a table in which you can enter the old addresses to be replaced and the new addresses. When you confirm with "OK", the SIMATIC Manager then exchanges the addresses.

When "rewiring" blocks, change the numbers of the blocks first and then execute rewiring that changes the calls correspondingly. If you "rewire" a function block, its instance data block is automatically assigned to the rewired function block; the data block number is not changed.

Following rewiring, an info file shows you in which block changes were made, and how many.

The reference data are no longer up-to-date following rewiring, and must be regenerated.

Please note that "rewiring" only takes place in the compiled blocks; a program source, if present, is not modified.

Further possible methods of rewiring are:

- With compiled blocks, you can also use the *Address priority* function.
- If there is a program source file with symbolic addressing, you change the absolute addresses in the symbol table. Following the compilation, you get an "unwired" program.

2.5.5 Address Priority

In the properties window of the offline object container *Blocks* on the "Address priority" tab, you can set whether the absolute address or the symbol is to have priority for already saved blocks when they are displayed and saved again following a change to the symbol table or to the declaration or assignment of global data blocks.

The default is "Absolute value has priority" (the same behavior as in the previous STEP 7 versions). This default means that when a change is made in the symbol table, the absolute address is retained in the program and the symbol changes accordingly. If "Symbol has priority" is set, the absolute address changes and the symbol is retained.

Example:

The symbol table contains the following:

```
I 1.0 "Limit_switch_up"
I 1.1 "Limit_switch_down"
```

In the program of an already compiled block, input I 1.0 is scanned:

```
I 1.0 "Limit_switch_up"
```

If the assignments for inputs I 1.0 and I 1.1 are now changed in the symbol table to:

```
I 1.0 "Limit_switch_down"
I 1.1 "Limit_switch_up"
```

and the already compiled block is read out, then the program contains

```
I 1.1 "Limit_switch_up"
```

if "Symbol has priority" is set, and if "Absolute value has priority" is set, the program contains

```
I 1.0     "Limit_switch_down"
```

If, as a result of a change in the symbol table, there is no longer any assignment between an absolute address and a symbol, the statement will contain the absolute address if "Absolute value has priority" is set (even with symbolic display because the symbol would, of course, be missing); if "Symbol has priority" is set, the statement is rejected as errored (because the mandatory absolute address is missing).

If "Symbol has priority" is set, incrementally programmed blocks with symbolic addressing will retain their symbols in the event of a change to the symbol table. In this way, an already programmed block can be "rewired" by changing the address assignment.

Please note that this "rewiring" does not occur automatically because the already compiled blocks contain the executable MC7 code of the statements with absolute addresses. The change is only made in the relevant block – following the relevant message – after it has been opened and saved again.

In order to carry out the change in the complete block folder, select EDIT → CHECK BLOCK CONSISTENCY with the *Blocks* object marked.

2.5.6 Reference Data

As a supplement to the program itself, the SIMATIC Manager shows you the reference data, which you can use as the basis for corrections or tests. These reference data include the following:

- ▷ Cross references
- ▷ Assignment (Input, Output, Bit Memory, Timers, Counters)
- ▷ Program structure
- ▷ Unused symbols
- ▷ Addresses without symbols

To generate reference data, select the *Blocks* object and the menu command OPTIONS → REFERENCE DATA → DISPLAY. The representation of the reference data can be changed specifically for each work window with VIEW → FILTER; you can save the settings for later editing by selecting WINDOW → SAVE ARRANGEMENT. You can display and view several lists at the same time (Figure 2.12).

With OPTIONS → CUSTOMIZE in the program editor, specify on the "Blocks" tab whether or not the reference data are to be updated when compiling a program source file or when saving an incrementally written block.

Please note that the reference data are only available when the data are managed offline; the offline reference data are displayed even if the function is called in a block opened online.

Cross references

The cross-reference list shows the use of the addresses and blocks in the user program. It includes the absolute address, the symbol (if any), the block in which the address was used, how it was used (read or write) and the positions of use of the address. Click on a column header to sort the table by column contents.

EDIT → GO TO → LOCATION with the position marked or a double click on the position starts the program editor and displays the address in the programmed environment.

The cross-reference list shows the addresses you selected with VIEW → FILTER (for instance bit memory). STEP 7 then uses the filter saved as "Standard" every time it opens the cross-reference list.

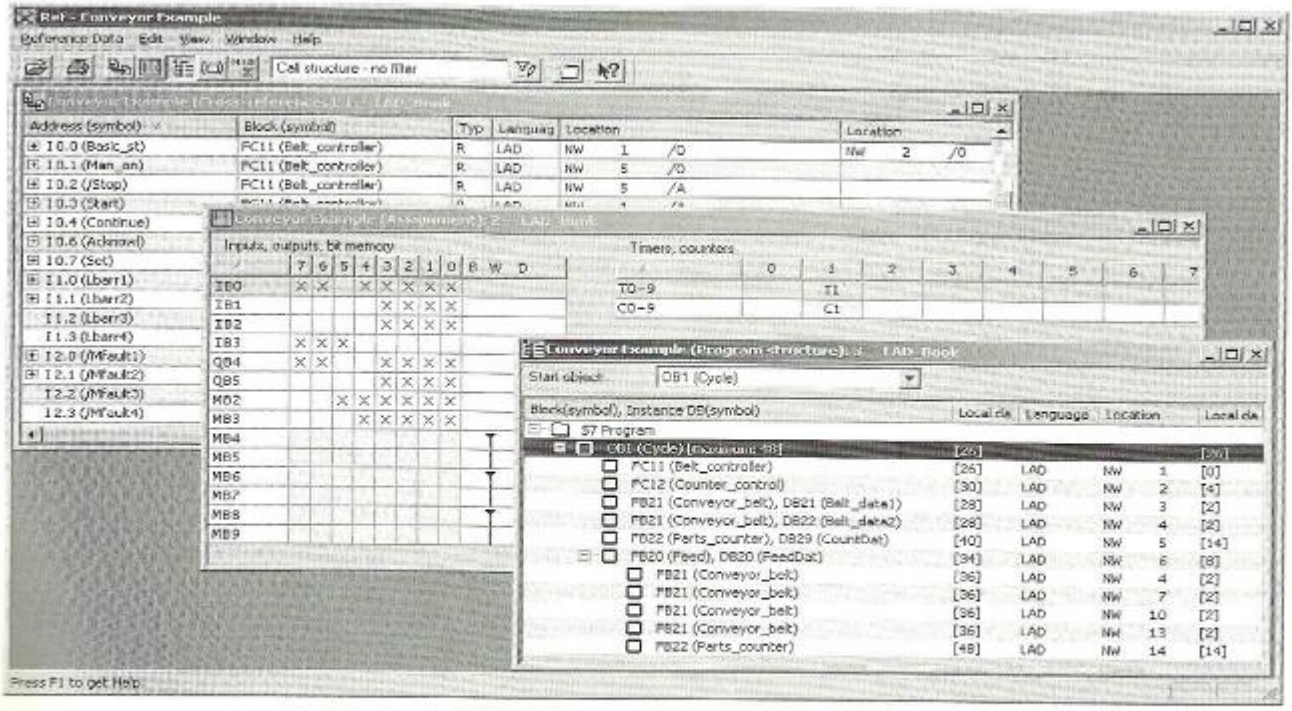


Figure 2.12 Examples of reference data (cross-references, assignment, program structure)

Advantage: the cross references show you whether the referenced addresses were also scanned or reset. They also show you in which blocks addresses are used (possibly more than once).

Assignments

The I/Q/M reference list shows which bits in address areas I, Q and M are assigned in the program. One byte, broken down into bits, appears on each line. Also shown is whether access is by byte, word, or doubleword. The T/C reference list shows the timers and counters used in the program. Ten timers or counters are displayed on a line.

Advantage: the list shows you whether certain address areas were (improperly) assigned or where there are still addresses available.

Program structure

The program structure shows the call hierarchy of the blocks in a user program. You can determine the start object for the call hierarchy from a selection list. With VIEW → FILTER you have a choice between two different views in the program structure:

The *Call structure* shows all nesting levels of the block calls. You control the display of nesting levels with the "+" and "–" boxes. The requirements for temporary local data are shown for one block or for the entire path up to the associated block. With the block selected, change using EDIT → GO TO → LOCATION to the block call, or open the block using EDIT → GO TO → BLOCK ADDRESS.

The display as *Dependency structure* shows two call levels. The blocks are shown (indented) in which the block position on the left is called.

Advantage: Which blocks were used? Were all programmed blocks called? What are the blocks' temporary local data requirements? Is the specified local data requirement per priority class (per organization block) sufficient?

Unused symbols

This list shows all addresses which have symbol table allocations but were not used in the program. The list shows the symbol, the address, the data type, and the comment from the symbol table.

Advantage: were the addresses in the list inadvertently forgotten when the program was being written? Or are they perhaps superfluous, and not really needed?

Addresses without symbol

This list shows all the addresses used in the program to which no symbols were allocated. The list shows these addresses and how often they were used.

Advantage: were addresses used inadvertently (by accident, or because of a typing error)?

2.5.7 Language Setting

STEP 7 offers several methods of working with different languages:

- The language of the operating system (character set)
- The STEP 7 language
- The language for comments and display text

The settings of the different languages are independent of each other.

Language settings in the operating system

You use the Windows control panel to select the character set with which you want to work under Windows. You can find the character sets tested with the multi-language version (MUI version) and the restrictions when operating with STEP 7 in the current Readme file or in the STEP 7 Help under "Setting up and editing the project".

Project language

The project language is the language that is set in the Windows control panel when the project is created. The SIMATIC Manager indicates the language in which the selected project or the selected library has been created in EDIT → OBJECT PROPERTIES. "Not yet defined" means you can use the project or the library language-neutrally, e.g. in multiprojects. These are always language-neutral. In language-neutral projects or libraries, only characters of the ASCII character set can be used ($2A_{hex}$ to $7F_{hex}$). You can

find additional information in the STEP 7 Help under "Setting up and editing the project".

STEP 7 language

The session language of the SIMATIC Manager that defines, for example, the menu names and the error messages, is called the STEP 7 language. You set this language in the SIMATIC Manager on the "Language" tab with OPTIONS → CUSTOMIZE. The languages installed with STEP 7 are offered for selection under "National language". On this tab you also set the programming mnemonics, that is, the language in which STEP 7 uses the operands and operations, e.g. "A I" (AND input for English), or "U E" (for German).

Multilingual Comments and Display Texts

Comments and display texts can be multilingual. You have entered the texts in the original language, such as English, and you want to generate a German version of your program. To do so, export the desired texts or text types. The export file is a *.csv file that you can edit with Microsoft Excel, for example. You can enter the translation for each text. You import the finished translation table back into your project. Now you can switch between the languages. You can do this with several languages.

With OPTIONS → LANGUAGE FOR DISPLAY DEVICES in the SIMATIC Manager, you select the languages available in your project, and you set the standard language for the display terminals.

Exporting and importing texts

Select the object in the SIMATIC Manager containing the comments you want to translate, e.g. the symbol table, the block container, several blocks or a single block. Select OPTIONS → MANAGE MULTILINGUAL TEXTS → EXPORT. In the dialog window that then appears, enter the storage location of the export file and the target language. Select the text types that you want to translate (Table 2.2).

A separate file is generated for every text type, e.g. the file SymbolComment.csv for the comments from the symbol table. Existing export files can be expanded. A log file provides information on the exported types of text, and any errors which have occurred.

Open the export file(s) with the FILE → OPEN dialog box in Microsoft Excel (not by double-clicking). The exported texts are displayed in the first column and you can translate the texts in the second column.

You can fetch the translated texts back to the project with OPTIONS → MANAGE MULTILINGUAL TEXTS → IMPORT. A log file provides information about the imported texts and any errors that may have occurred.

Please note that the name of the import file must not be changed since there is a direct relation between this and the text types contained in the file.

Selecting and deleting a language

You can change to all imported languages in the SIMATIC Manager with OPTIONS → MANAGE MULTILINGUAL TEXTS → CHANGE LANGUAGE. The language change is executed for the objects (blocks, symbol table) for which the relevant texts have been imported. This information is contained in the log file. You can make further settings, e.g. the "taking over" of multilingual comments when copying a block, using OPTIONS → MANAGE MULTILINGUAL TEXTS → SETTINGS FOR COMMENT MANAGEMENT. You can delete the imported language again with OPTIONS → MANAGE MULTILINGUAL TEXTS → DELETE LANGUAGE.

Table 2.2
Text types of the translated texts (selection)

Text type	Meaning
BlockTitle	Block title
BlockComment	Block comment
NetworkTitle	Network title
NetworkComment	Network comment
LineComment	STL line comment
InterfaceComment	Comment in ▷ the declaration table of code blocks ▷ data blocks ▷ user data types UDT
SymbolComment	Symbol comment

2.6 Online Mode

You create the hardware configuration and the user program on the programming device, generally referred to as the "engineering system" (ES). The S7 program is stored offline on the hard disk here, also in compiled form.

To transfer the program to the CPU, you must connect the programming device to the CPU. You establish an "online" connection. You can use this connection to determine the operating state of the CPU and the assigned modules, i.e., you can carry out diagnostics functions.

2.6.1 Connecting a PLC

The connection between the programming device's MPI interface and the CPU's MPI interface is the mechanical requirement for an online connection. The connection is unique when a CPU is the only programmable module connected. If there are several CPUs in the MPI subnet, each CPU must be assigned a unique node number (MPI address). You set the MPI address when you initialize the CPU. Before linking all the CPUs to one network, connect the programming device to only one CPU a time and transfer the *System Data* object from the offline user program *Blocks* or direct with the Hardware Configuration editor using the menu command PLC → DOWNLOAD. This assigns a CPU its own special MPI address ("naming") along with the other properties.

The MPI address of a CPU in the MPI network can be changed at any time by transferring a new parameter data record containing the new MPI address to the CPU. Note carefully: the new MPI address takes effect immediately. While the programming device adjusts immediately to the new address, you must adapt other applications, such as global data communications, to the new MPI address.

The MPI parameters are retained in the CPU even after a memory reset. The CPU can thus be addressed even after a memory reset.

A programming device can always be operated online on a CPU, even with a module-independent program and even though no project has been set up.

If no project has been set up, you establish the connection to the CPU with PLC → DISPLAY ACCESSIBLE NODES. This screens a project window with the structure *"Accessible Nodes"* – "Module (*MPI=n*)" – "Online User Program *(Blocks)*". When you select the *Module* object, you may utilize the online functions, such as changing the operational status and checking the module status. Selecting the *Blocks* object displays the blocks in the CPU's user memory. You can then edit (modify, delete, insert) individual blocks.

You can fetch back the system data from a connected CPU for the purpose of, say, continuing to work on the basis of the existing configuration, without having the relevant project in the programming device data management system. Create a new project in the SIMATIC Manager, select the project and then PLC → UPLOAD STATION TO PG. After specifying the desired CPU in the dialog box that then appears, the online system data are loaded onto the hard disk.

If there is a **CPU-independent program** in the project window, create the associated online project window. If several CPUs are connected to the MPI and accessible, select EDIT → OBJECT PROPERTIES with the online S7 program selected and set the number of the mounting rack and the CPU's slot on the "Addresses Module" tab.

If you select the *S7 Program* in the online window all the online functions to the connected CPU are available to you. *Blocks* shows the blocks located in the CPU's user memory. If the blocks in the offline program agree with the blocks in the online program you can edit the blocks in the user memory with the information from the data management system of the programming device (symbolic address, comments).

When you switch a **CPU-assigned program** into online mode using VIEW → ONLINE, you can carry out program modifications just as you would in a CPU-independent program. In addition, it is now possible for you to configure the SIMATIC station, that is, to set CPU parameters and address and parameterize modules.

2.6.2 Protecting the User Program

With appropriately equipped CPUs, access to the user program can be protected with a password. Everyone in possession of the password has unrestricted access to the user program. For those who do not know the password, you can define 3 protection levels. You set the protection levels with the "Protection" tab of the Hardware Configuration tool when parameterizing the CPU.

The access privilege using the password applies until the SIMATIC Manager has been exited or the password protection canceled again using PLC → ACCESS RIGHTS → CANCEL.

Protection level 1: mode selector switch

This protection level is set as default (without password). With CPUs with a keylock switch, protection level 1 is used to set protection of the user program by the mode selector switch on the front of the CPU. In the RUN-P and STOP positions, you have unrestricted access to the user program; in the RUN position, only read access via the programming device is possible. In this position, you can also remove the keylock switch so that the mode can no longer be changed via the switch.

You can bypass protection via the keylock switch RUN position by selecting the option "Removable with password", e.g. if the CPU, and with it the keylock switch, are not easily accessible or are located at a distance.

If the mode selector switch is designed as a toggle switch, protection level 1 means no limitation in access to the user program.

With the system function SFC 109 PROTECT, the write protection (protection level 2) can be switched on and off via the program in protection level 1 (see Chapter 20.3.8 "Changing program protection").

Protection level 2: write protection

At this protection level, the user program can only be read, regardless of the position of the keylock switch.

Protection level 3: read/write protection

No access to the user program, regardless of the keylock switch position. Exception: reading of diagnostics buffer and monitoring of variables in tables is possible in every protection level.

Password protection

If you select protection level 2 or 3 or protection level 2 with "Removable with password", you will be prompted to define a password. The password can be up to 8 characters long.

If you try to access a user program that is protected with a password, you will be prompted to enter the password. Before accessing a protected CPU, you can also enter the password via PLC → ACCESS RIGHTS → SETUP. First, select the relevant CPU or the S7 program.

In the "Enter Password" dialog box, you can select the option "Use password as default for all protected modules" to get access to all modules protected with the same password.

Password access authorization remains in force until the last S7 application has been terminated.

Everyone in possession of the password has unrestricted access to the user program in the CPU regardless of the protection level set and regardless of the keylock position.

2.6.3 CPU Information

In online mode, the CPU information listed below is available to you. The menu commands are screened when you have selected a module (in online mode and without a project) or S7 program (in the online project window).

▷ PLC → DIAGNOSTIC/SETTING

→ HARDWARE DIAGNOSTICS
(see Chapter 2.7.1 "Diagnosing the Hardware")

→ MODULE INFORMATION
General information (such as version), diagnostics buffer, memory (current map of work memory and load memory, compression), cycle time (length of the last, longest, and shortest program cycle), timing system (properties of the CPU clock, clock synchronization, run-time meter), performance

data (available organization blocks and system blocks, sizes of the address areas), communication (data transfer rate and communication links), stacks in STOP state (B stack, I stack, and L stack)

→ OPERATING MODE
Display of the current operating mode (for instance RUN or STOP), modification of the operating mode

→ CLEAR/RESET
Resetting of the CPU in STOP mode

→ SET TIME OF DAY
Setting of the internal CPU clock and - in the enhanced dialog - the difference from a time zone

▷ PLC → CPU MESSAGES
Reporting of asynchronous system errors and of user-defined messages generated in the program with SFC 52 WR_USMSG, SFC 18 ALARM_S, SFC 17 ALARM_SQ, SFC 108 ALARM_D and SFC 107 ALARM_DQ.

▷ PLC → DISPLAY FORCE VALUES, PLC → MONITOR/MODIFY VARIABLES
(see Chapters 2.7.3 "Monitoring and Modifying Variables" and 2.7.4 "Forcing Variables")

2.6.4 Loading the User Program into the CPU

When you transfer your user program (compiled blocks and configuration data) to the CPU, it is loaded into the CPU's load memory. Physically, load memory can be a memory integrated in the CPU, a memory card, or a micro memory card (see Chapter 1.1.6 "CPU Memory Areas").

With a micro memory card or a flash EPROM memory card you can write die card in the programming device and use it as data medium. You plug the card into the CPU in the off-circuit state; on power up following memory reset, the relevant data of the card are transferred to the work memory of the CPU. With appropriately equipped CPUs, you can also overwrite a flash EPROM memory card or a micro memory card if it is plugged into the CPU, but only with the entire program.

In the case of a RAM load memory (integrated in the CPU, as a memory card or micro memory card), you transfer a complete user program by switching the CPU to the STOP state, performing memory reset and transferring the user program. The configuration data are also transferred.

If you only want to change the configuration data (CPU properties, the configured connections, GD communications, module parameters, and so on), you need only load the *System Data* object into the CPU (select the object and transfer it with menu command PLC → DOWNLOAD). The parameters for the CPU go into effect immediately; the CPU transfers the parameters for the remaining modules to those modules during startup.

Please note that the entire configuration is loaded onto the PLC with the *System data* object. If you use PLC → DOWNLOAD in an application, e.g. in global data communications, only the data edited by the application are transferred.

Note: select PLC → SAVE TO MEMORY CARD to load the compressed archive file (see Chapter 2.2.2 "Managing, Reorganizing and Archiving"). The project in the archive file cannot be edited direct either with the programming device or from the CPU.

2.6.5 Block Handling

Transferring blocks

In the case of a RAM load memory, you can also modify, delete or reload individual blocks in addition to transferring the entire program online.

You transfer individual blocks to the CPU by selecting them in the offline window and selecting PLC → DOWNLOAD. With the offline and online windows opened at the same time, you can also drag the blocks with the mouse from one window and drop them in the other.

Special care is needed when transferring individual blocks during operation. If blocks that are not available in the CPU memory are called within a block, you must first load the "lower-level" blocks. This also applies for data blocks whose addresses are used in the loaded block.

You load the "highest-level" block last. Then, provided it is called, it will be executed immediately in the next program cycle.

Modifying or deleting blocks online

You can edit blocks incrementally with STL in the online user program (on the CPU) in exactly the same way as in the offline user program. With a programming device connected online to the CPU, you can read, modify or delete blocks in the load memory.

If the RAM component of the load memory is large enough to accommodate the complete user program and also the modified blocks, you can edit blocks without limitation.

If the user program is saved on a flash EPROM memory card, you can edit the blocks as long as the RAM component of the load memory is large enough to accommodate the modified blocks. During runtime, the modified blocks in the RAM are valid, those in the FEPROM as invalid. Please note that you must load the original blocks again from the FEPROM into the work memory following an overall reset or non-buffered switching on.

If you use a micro memory card such as e.g. with the compact CPUs, all blocks in the load memory are non-volatile. You can modify individual blocks online, and these blocks retain their changes even following an overall reset or non-buffered switching on. Deleted blocks are then no longer existent.

In incremental programming mode, you can modify blocks independent of one another in the offline data management on the programming device and in the online data management on the CPU. However, if online and offline data management diverge, it may result in the editor being unable to display the additional information of the offline database; these data can then be lost (symbolic identifiers, jump labels, comments, user data types).

Blocks that have been modified online are best stored offline on the hard disk to avoid data inconsistency (e.g. a "time stamp conflict" when the interface of the called block is later than the program in the calling block).

The following still applies even if you work online: with FILE → SAVE you save the current block in the offline user program in the PG data management; with PLC → DOWNLOAD you write the block back into the user memory and the CPU.

Compressing

When you load a new or modified block into the CPU, the CPU places the block in load memory and transfers the relevant data to work memory. If there is already a block with the same number, this "old block" is declared invalid (following a prompt for confirmation) and the new block "added on at the end" in memory. Even a deleted block is "only" declared invalid, not actually removed from memory.

This results in gaps in user memory which increasingly reduce the amount of memory still available. These gaps can be filled only by the *Compress* function. When you compress in RUN mode, the blocks currently being executed are not relocated; only in STOP mode can you truly achieve compression without gaps.

The current memory allocation can be displayed in percent with the menu command PLC → DIAGNOSTIC/SETTING → MODULE INFORMATION, on the "Memory" tab. The dialog box which then appears also has a button for preventive compression.

You can initiate event-driven compressing per program with the call SFC 25 COMPRESS.

Data blocks offline/online

During programming, you assign the data addresses in a data block with a default value and an initial value (see also Chapter 3.4 "Programming Data Blocks"). If a data block is loaded into the CPU, the initial values are transferred to load memory and subsequently to work memory, where they become actual values. Every value change made to a data address per program corresponds to a change to the actual value in the work memory (Figure 2.13).

You can upload the actual values generated in the work memory from a programmed (loaded) data block into the offline data management by opening the data block online and transferring it

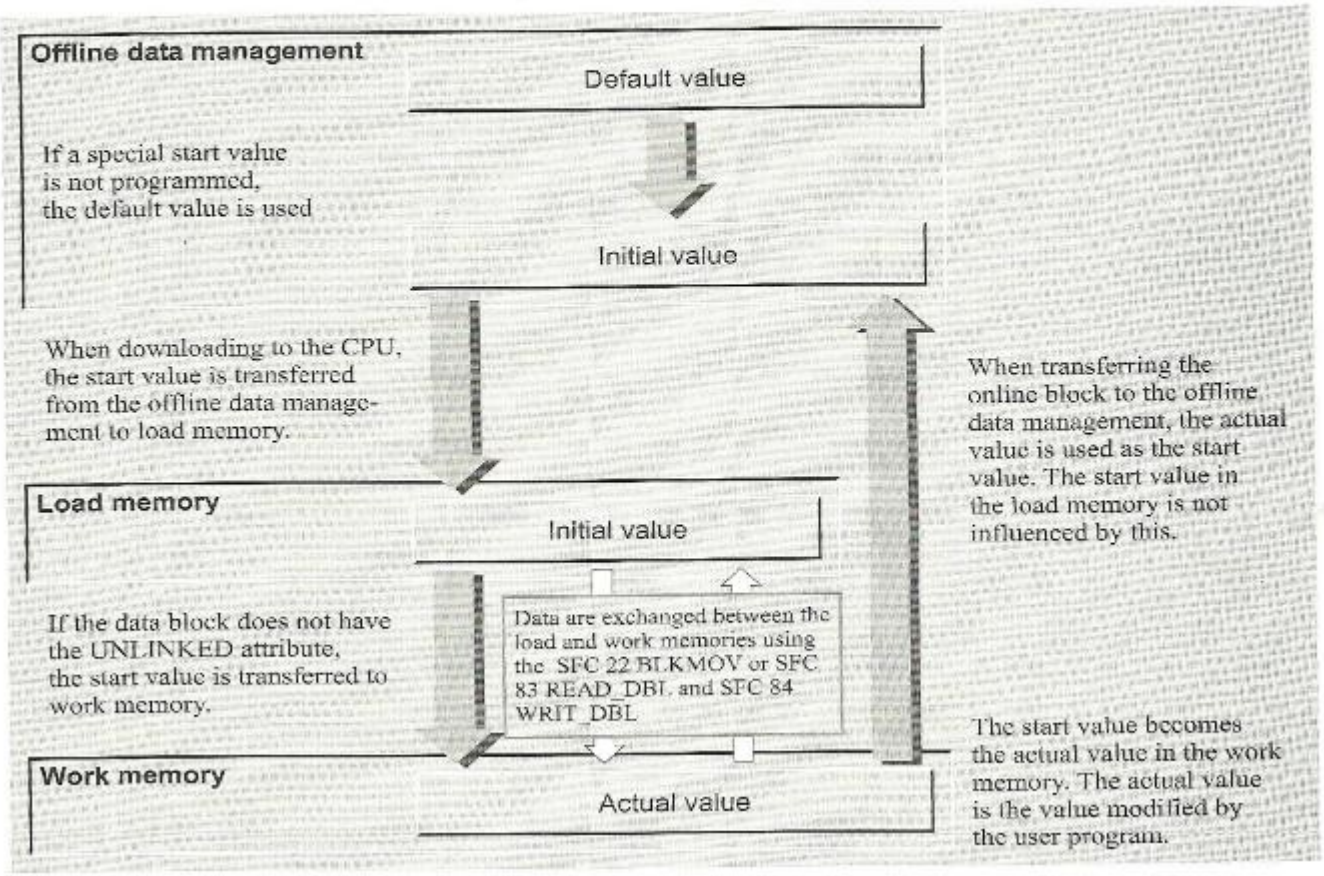


Figure 2.13 Data storage in user memory

in the program editor to the offline data management using FILE → SAVE. The names of the variables and the data types saved in the offline data management are then retained. If you upload the online data block in the SIMATIC Manager using PLC → UPLOAD TO PG or by dragging the data block from the online window to the offline window in the offline data management, the description of the addresses, such as e.g. the name of the variables or the data type, is lost.

If you upload a data block from the CPU back into the offline data management, the actual values present in the work memory are imported into the offline data management as initial values. This does not change the initial values in the load memory. Following an overall reset or non-buffered switching on, and when using a flash EPROM memory card or a micro memory card, the (old) initial values present in the load memory are imported into the work memory as (new) actual values.

If you wish to import the actual values into the load memory when using a RAM load memory or a micro memory card, load the data block from the CPU into the programming device and then back again into the CPU. CPUs with micro memory card provide the system function SFC 84 WRIT_DBL with which you can directly write actual values into the load memory. With correspondingly designed CPUs, you can transfer the complete contents of the work memory into the ROM component of the load memory using PLC → COPY RAM TO ROM.

A data block generated with the property *Unlinked* is not transferred to work memory; it remains in load memory. A data block with this property can only be read with SFC 20 BLKMOV or – with correspondingly designed CPUs – with SFC 83 READ_DBL.

In incremental programming mode, you can create data blocks directly in the work memory of the CPU. It is recommendable to also save

these blocks offline immediately they have been created.

With the system functions SFC 22 CREAT_DB, SFC 85 CREA_DB and SFC 82 CREA_DBL you can generate data blocks during runtime where the description of the addresses, such as the name of the variable and the type of data, are missing. When reading with the programming device, a BYTE field with a name and index assigned by the program editor is therefore displayed. If you transfer such a data block to the offline data management, this declaration is imported. If the data block has the property *Unlinked*, the initial values from the load memory are imported into the offline data management as new initial values, otherwise the actual values from the work memory. When transferring to the offline data management, the checksum of the (offline) program is changed.

2.7 Testing the Program

After establishing a connection to a CPU and loading the user program, you can test (debug) the program as a whole or in part, such as individual blocks. You initialize the variables with signals and values, e.g. with the help of simulator modules and evaluate the information returned by your program. If the CPU goes to the STOP state as a result of an error, you can get support in finding the cause of the error from the CPU information among other things.

Extensive programs are debugged in sections. If, for example, you only want to debug one block, load this block into the CPU and call it in OB 1. If OB 1 is organized is in such a way that the program can be debugged section by section "from beginning to end", you can select the blocks or program sections for debugging by using jump functions to skip those calls or program sections that are not to be debugged.

With the S7 PLCSIM optional software, you can simulate a CPU on the programming device and so debug your program without additional hardware.

2.7.1 Diagnosing the Hardware

In the event of a fault, you can fetch the diagnostics information of the faulty modules with the help of the function "Diagnose Hardware". You connect the programming device to the MPI bus and start the SIMATIC Manager.

If the project associated with the plant configuration is available in the programming device database, open the online project window with VIEW → ONLINE. Otherwise, select PLC → DISPLAY ACCESSIBLE NODES in the SIMATIC Manager, and select the CPU.

Now you can get a quick overview of the faulty modules with PLC → DIAGNOSTIC/SETTING → HARDWARE DIAGNOSTICS (default in the SIMATIC Manager under OPTIONS → CUSTOMIZE in the tab "View"). If the quick overview is deselected, you are provided with the detailed diagnostics information of all modules.

If you are in the Hardware Configuration tool, select the online view using VIEW → ONLINE. You can now display the existing diagnostics information for the selected module using PLC → MODULE INFORMATION.

2.7.2 Determining the Cause of a STOP

If the CPU goes to STOP because of an error, the first measure to take in order to determine the reason for the STOP is to output the diagnostics buffer. The CPU enters all messages in the diagnostic buffer, including the reason for a STOP and the errors which led to it.

To output the diagnostic buffer, switch the PG to online, select an S7 program, and choose the "Diagnostics Buffer" tab with the menu command PLC → DIAGNOSTIC/SETTING → MODULE INFORMATION. The last event (the one with the number 1) is the cause of the STOP, for instance "STOP because programming error OB not loaded". The error which led to the STOP is described in the preceding message, for example "FC not loaded". By clicking on the message number, you can screen an additional comment in the next lower display field. If the message relates to a programming error in a block, you can open and edit that block with the "Open Block" button.

If the cause of the STOP is, for example, a programming error, you can ascertain the surrounding circumstances with the "Stacks" tab. When you open "Stacks", you will see the B stack (block stack), which shows you the call

path of all non-terminated blocks up to the block containing the interrupt point. Use the "I stack" button to screen the interrupt stack, which shows you the contents of the CPU registers (accumulators, address register, data block register, status word) at the interrupt point at the instant the error occurred. The L stack (local data stack) shows the block's temporary local data, which you select in the B stack by clicking with the mouse.

2.7.3 Monitoring and Modifying Variables

One excellent resource for debugging user programs is the monitoring and modifying of variables with VAT variable tables. Signal states or values of variables of elementary data types can be displayed. If you have access to the user program, you can also modify variables, i.e. change the signal state or assign new values.

Please note that you can only modify data addresses if the write protection for the data block is switched off, i.e. the block property *DB is write-protected in the AS* is not activated.

Operands in data blocks with the block property *Unlinked* cannot be monitored since these data blocks are located in the load memory on the micro memory card. A one-off update takes place when the data block is opened online.

Caution: you must ensure that no dangerous states can result from modifying variables!

Creating a variable table

For monitoring and modifying variables, you must create a VAT variable table containing the variables and the associated data formats. You can generate up to 255 variable tables (VAT 1 to VAT 255) and assign them names. The maximum size of a variable table is 1024 lines with up to 255 characters (Figure 2.14).

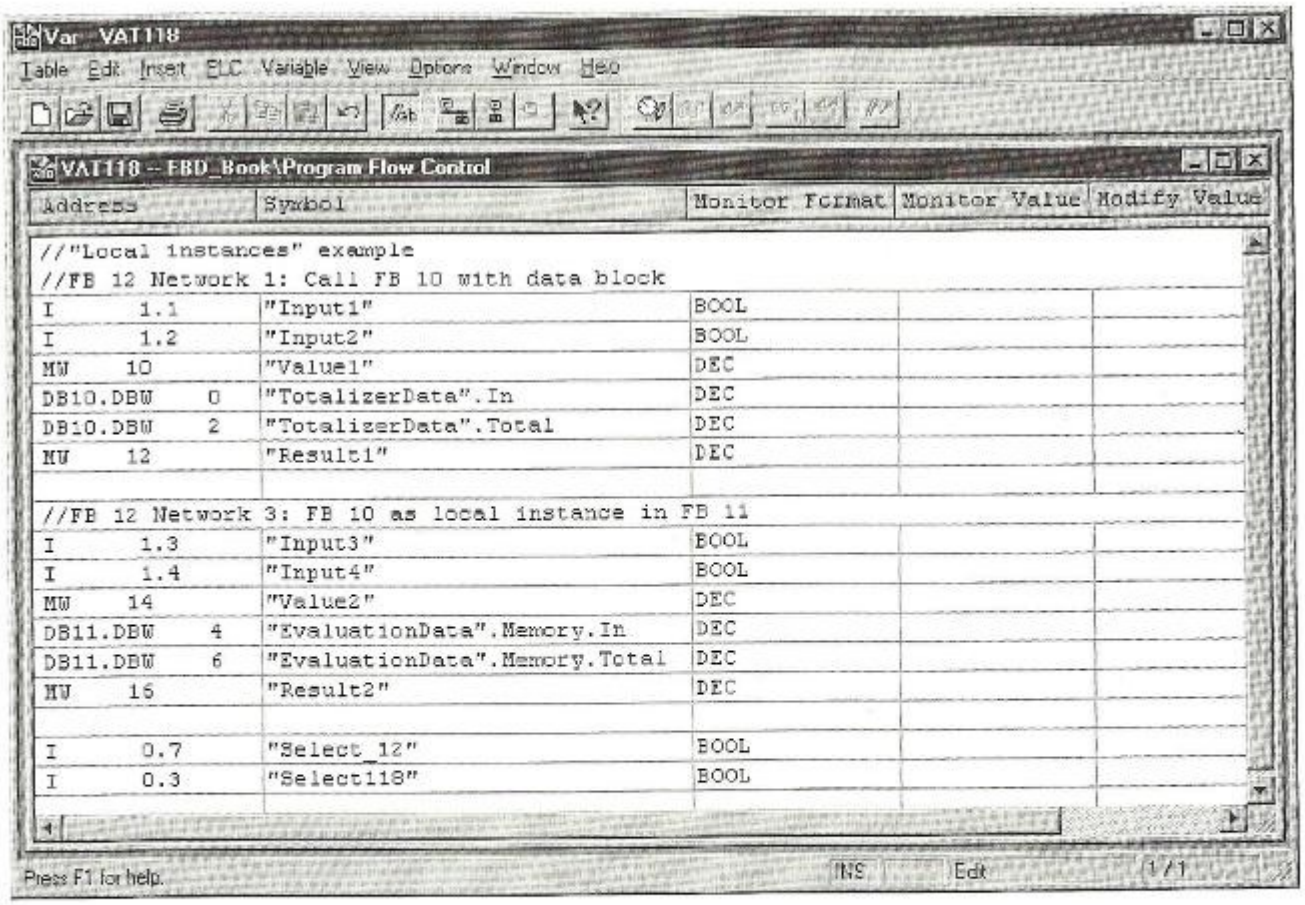


Figure 2.14 Variable Table Example

You can generate a VAT offline by selecting the user program *Blocks* and then INSERT → S7 BLOCK → VARIABLE TABLE, and you can generate an unnamed VAT online by selecting *S7 Program* and selecting PLC → MONITOR/ MODIFY VARIABLES.

You can specify the variables with either absolute or symbolic addresses and choose the data type (display format) with which a variable is to be displayed and modified (with VIEW → SELECT DISPLAY FORMAT or by clicking the right mouse button directly on the "Display format").

Use comment lines to give specific sections of the table a header. You may also stipulate which columns are to be displayed. You can change variable or display format or add or delete lines at any time. You save the variable table in the *Blocks* object container with TABLE → SAVE.

Establishing an online connection

To operate a variable table that has been created offline, switch it online with PLC → CONNECT TO → ... online. You must switch each individual VAT online and you can clear down the connection again with PLC → DISCONNECT.

Trigger conditions

In the variable table, select VARIABLE → TRIGGER to set the trigger point and the trigger conditions separately for monitoring and modifying. The trigger point is the point at which the CPU reads values from the system memory or writes values to the system memory. You specify whether reading and writing is to take place once or periodically.

If monitoring and modifying have the same trigger conditions, monitoring is carried out before modifying. If you select the trigger point "Start of cycle" for modifying, the variables are modified after updating of the process input image and before calling OB 1. If you select the trigger point "End of cycle" for monitoring, the status values are displayed after termination of OB 1 and before output of the process output image.

Monitoring variables

Select the Monitor function with the menu command VARIABLE → MONITOR. The variables in the VAT are updated in accordance with the specified trigger conditions. Permanent monitoring allows you to follow changes in the values on the screen. The values are displayed in the data format which you set in the "Display format" column. The ESC key terminates a permanent monitor function.

VARIABLE → ACTIVATE MODIFY VALUES updates the monitor values once only and immediately without regard to the specified trigger conditions.

Modifying variables

Use VARIABLE → MODIFY to transfer the specified values to the CPU dependent on the trigger conditions. Enter values only in the lines containing the variables you want to modify. You can expand the commentary for a value with "//" or with VARIABLE → MODIFY VALUE AS COMMENT; these values are not taken into account for modification. You must define the values in the data format which you set in the "Display format" column. Only the values visible on starting the modify function are modified. The ESC key terminates a permanent modify function.

VARIABLE → ACTIVATE MODIFY VALUES transfers the modify values only once and immediately, without regard to the specified trigger conditions.

2.7.4 Forcing Variables

With appropriately equipped CPUs, you can specify fixed values for certain variables. The user program can no longer change these values ("forcing"). Forcing is permissible in any CPU operating state and is executed immediately.

Caution: you must ensure that no dangerous states can result from forcing variables!

The starting point for forcing is a variable table (VAT). Create a VAT, enter the addresses to be forced and establish a connection to the CPU. You can open a window containing the force values by selecting VARIABLE → DISPLAY FORCE VALUES.

If there are already force values active in the CPU, these are indicated in the force window in bold type. You can now transfer some or all addresses from the variable table to the force window or enter new addresses. You save the contents of a force window in a VAT with TABLE → SAVE AS.

The following address areas can be provided with a force value:

- Inputs I (process image) [S7-300 and S7-400]
- Outputs Q (process image) [S7-300 and S7-400]
- Peripheral inputs PI [S7-400]
- Peripheral outputs PQ [S7-400]
- Memory bits M [S7-400]

You start the force job with VARIABLE → FORCE. The CPU accepts the force values and permits no more changes to the forced addresses.

While the force function is active, the following applies:

- All read accesses to a forced address via the user program (e.g. load) and via the system program (e.g. updating of the process image) always yield the force value.
- *On the S7-400, all write accesses to a forced* address via the user program (e.g. transfer) and via the system program (e.g. via SFCs) remain without effect. On the S7-300, the user program can overwrite the force values.

Forcing on the S7-300 corresponds to cyclic modifying: after the process input image has been updated, the CPU overwrites the inputs with the force value; before the process output image is output, the CPU overwrites the outputs with the force value.

Note: forcing is not terminated by closing the force window or the variable table, or by breaking the connection to the CPU! You can only delete a force job with VARIABLE → STOP FORCING.

Forcing is also deleted by memory reset or by a power failure if the CPU is not battery-backed. When forcing is terminated, the addresses retain the force values until overwritten by either the user program or the system program.

Forcing is effective only on I/O assigned to a CPU. If, following restart, forced peripheral inputs and outputs are no longer assigned (e.g. as a result of reparameterizing), the relevant peripheral inputs and outputs are no longer forced.

Error handling

If the access width when reading is greater than the force width (e.g. forced byte in a word), the unforced component of the address value is read as usual. If a synchronization error occurs here (access or area length error) the "error substitute value" specified by the user program or by the CPU is read or the CPU goes to STOP.

If, when writing, the access width is greater than the force width (e.g. forced byte in a word), the unforced component of the address value is written to as usual. An errored write access leaves the forced component of the address unchanged, i.e. the write protection is not revoked by the synchronization error.

Loading forced peripheral inputs yields the force value. If the access width agrees with the force width, input modules that have failed or have not (yet) been plugged in can be "replaced" by a force value.

The input I in the process image belonging to a forced peripheral input PI is not forced; it is not preassigned and can still be overwritten. When updating the process image, the input receives the force value of the peripheral input.

When forcing peripheral outputs PQ, the associated output Q in the process image is not updated and not forced (forcing is only effective "externally" to the module outputs). The outputs Q are retained and can be overwritten; reading the outputs yields the written value (not the force value). If an output module is forced and if this module fails or is removed, it will receive the force value again immediately on reconnection.

The output modules output signal state "0" or the substitute value with the OD signal (disable

output modules at STOP, HOLD or RESTART) – even if the peripheral outputs are forced (exception: analog modules without OD evaluation continue to output the force value). If the OD signal is deactivated, the force value becomes effective again.

If, in STOP mode, the function *Enable PQ* is activated, the force values also become effective in STOP mode (due to deactivation of the OD signal). When *Enable PQ* is terminated, the modules are set back to the "safe" state (signal state "0" or substitute value); the force value becomes effective again at the transition to RUN.

2.7.5 Enabling Peripheral Outputs

In STOP mode, the output modules are normally disabled by the OD signal; with the Enable peripheral outputs function, you can deactivate the OD signal so that you can modify the output modules even at CPU STOP. Modifying is carried out via a variable table. Only the peripheral outputs assigned to a CPU are modified. Possible application: wiring test of the output at STOP and without user program.

Caution: you must ensure that no dangerous states can result from enabling the peripheral outputs!

Create a variable table and enter the peripheral outputs (PQ) and the modify values. Switch the variable table online with PLC → CONNECT TO → ... and stop the CPU if necessary, e.g. with PLC → OPERATING MODE and "STOP".

You deactivate the OD signal VARIABLE → ENABLE PERIPHERAL OUTPUTS; the module outputs now have signal state "0" or the substitute value or force value. You modify the peripheral outputs with VARIABLE → ACTIVATE MODIFY VALUES. You can change the modify value and modify again.

You can switch the function off again by selecting VARIABLE → ENABLE PERIPHERAL OUTPUTS again, or by pressing the ESC key. The OD signal is then active again, the module outputs are set to "0" and the substitute value or the force value is reset.

If STOP is exited while "enable peripheral outputs" is still active, all peripheral inputs are deleted, the OD signal is activated at the transition to RESTART and deactivated again at the transition to RUN.

2.7.6 Test and process operation

The recording of the program status information requires additional execution time in the program cycle. For this reason, you can choose two operating modes for debugging purposes: test mode and process mode. In *test mode*, all debugging functions can be used without restriction. You would select this, for example, to debug blocks without connection to the system, because this can significantly increase the cyclic execution time. In *process mode*, care is taken to keep the increase in cycle time to a minimum and this results in debugging restrictions, e.g. the status display with program loops is aborted at the return point. Debugging and step-by-step program execution cannot be performed in process operation.

Test mode is set in the factory on the S7-300 CPUs. You can set test or process mode on these CPUs with the Hardware Configuration on the "Protection" tab. Following this, the configuration must be compiled again and downloaded to the CPU.

The process mode is set in the factory on the S7-400 CPUs. You can change the operating mode online with the Program Editor. DEBUG → OPERATION... displays the set operating mode and offers the facility of changing this online.

2.7.7 LAD/FBD Program Status

With the *Program status* function, the program editor provides an additional test method for the user program. The editor shows you the binary signal flow and digital values within a network.

The block whose program you want to debug is in the CPU's user memory and is called and edited there. Open this block, for example by double-clicking on it in the SIMATIC Manager's online window. The editor is started and shows the program in the block.

Select the network you want to debug. Activate the Program Status function with DEBUG → MONITOR. Now you can see the binary signal

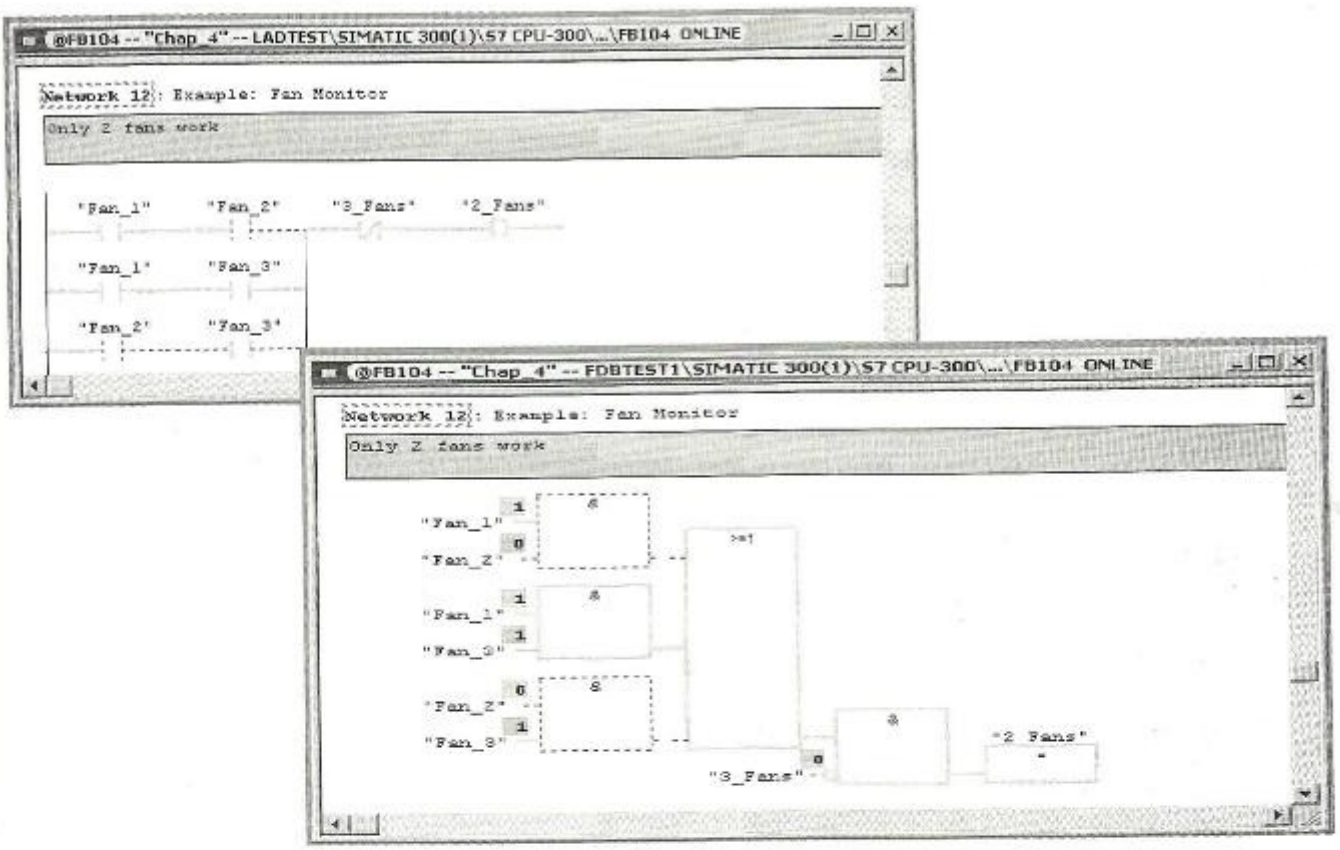


Figure 2.15 LAD/FBD program status

flow in the block window and you can follow the changes in it (Figure 2.15). You define the representation (e.g. color) in the program editor with OPTIONS → CUSTOMIZE on the "LAD/FBD" tab. You can deactivate the Program Status function again by selecting DEBUG → MONITOR again.

You set the trigger conditions with DEBUG → CALL ENVIRONMENT with debug mode switched on (see chapter 2.7.6 "Test and process operation"). You require this setting if the block to be debugged is called more than once in your program. You can initiate status recording either by specifying the call path (determined from the reference data or manually) or by making it dependent on the opened data blocks when calling the block to be debugged. If you do not set the call environment, you monitor the block when it is called for the first time.

Modifying addresses

You can modify addresses in the program status function. If the address is of data type BOOL, mark it and select DEBUG → MODIFY ADDRESS TO 0 or DEBUG → MODIFY ADDRESS TO 1. With a different data type, select DEBUG → MODIFY ADDRESS, and enter the modified value for the marked address in the dialog box displayed.

Operations on the contact

In the Program Status function, you can directly modify binary inputs and bit memories in the user program by means of a button. The following prerequisites exist for this function:

- In the symbol table, you assign the inputs and bit memories with the attribute CC (Control at Contact, see "Special object properties" in Chapter 2.5.2 "Symbol Table").

- ▷ You have enabled operations on the contact in the program editor using OPTIONS → CUSTOMIZE on the "General" tab.
- ▷ You are online in the program status with DEBUG → MONITOR and additionally select DEBUG → CONTROL AT CONTACT.

The symbols and addresses of the binary inputs and bit memories are displayed as buttons which you can access using the mouse. Addresses programmed as NO contacts or addresses with scanning for signal status "1" then deliver the address status "1"; addresses programmed as NC contacts or addresses with scanning for signal status "0" deliver the status "0" when accessed. You can use the Ctrl/Strg key and the mouse to select several addresses and to access them simultaneously when operating on the contact. You deselect the operands in the same manner.

2.7.8 Monitoring and Modifying Data Addresses

If the variables to be debugged are present in data blocks, you can also view and modify them directly. Select the data block in the SIMATIC Manager and select EDIT → OPEN OBJECT. With STEP 7 V5.2 and later, you will be asked in the default setting whether you wish to open the data block using the program editor or using the application "Parameter assignment for data blocks".

Switch the data view on in the program editor using VIEW → DATA VIEW, and select DEBUG → MONITOR. You can now view the actual values in the work memory, and also set (modify) them if required. Using PLC → DOWNLOAD, download the modified actual values into the work memory, or use FILE → SAVE to import the modified values into the offline data management (first switched off DEBUG → MONITOR).

Using "Parameter assignment for data blocks" you can directly view and modify the actual values in the work memory of the CPU. You can also view the actual values here using DEBUG → MONITOR, and you can also adjust them. Using PLC → DOWNLOAD PARAMETER SETTING DATA you have the possibility for only writing the actual values into the work memory, and not the complete data block. Using DATA BLOCK → SAVE you can import the data block into the offline data management.

The advantage of the application "Parameter assignment for data blocks" is to be found in the possibility for displaying and parameterizing data blocks in the parameterization view. Prerequisite: the system attribute *S7-techparam* (technological functions) is set, and a parameterization desktop is available, e.g. from an option package. Figure 2.16 shows a comparison between parameterization view and data view using an example of the instance data block for the controller function block FB 58 TCONT_CP from the standard library *PID Control Blocks*. Its parameterization desktop is supplied together with STEP 7.

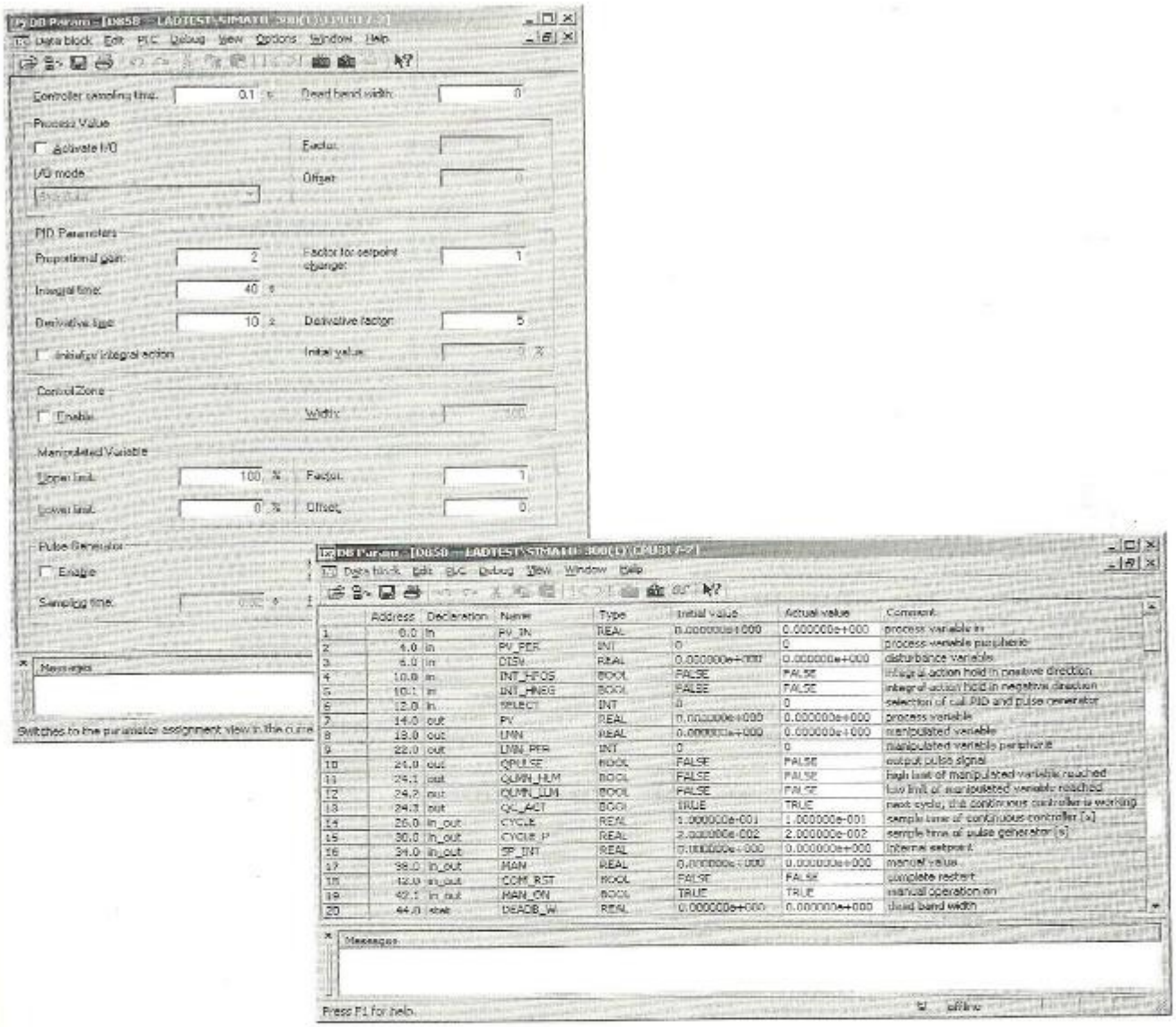


Figure 2.16 Comparison between parameterization view and data view

3 SIMATIC S7 Program

This chapter shows you the structure of the user program for the SIMATIC S7-300/400 CPUs starting from the different priority classes (program execution types) via the component parts of a user program (blocks) right up to the variables and data types. The focus of this chapter is the description of block programming with LAD and FBD. Following this is a description of the data types.

You define the structure of the user program right back at the design phase when you adapt the technological and functional conditions; it is decisive for program creation, program test and startup. To achieve effective programming, it is therefore necessary to devote special attention to the program structure.

3.1 Program Processing

The overall program for a CPU consists of the operating system and the user program.

The operating system is the totality of all instructions and declarations which control the system resources and the processes using these resources, and includes such things as data backup in the event of a power failure, the activation of priority classes, and so on. The operating system is a component of the CPU to which you, as user, have no write access. However, you can reload the operating system from a memory card, for instance in the event of a program update.

The user program is the totality of all instructions and declarations for signal processing, through which a plant (process) is affected in accordance with the defined control task.

3.1.1 Program Processing Methods

The user program may be composed of program sections which the CPU processes in dependence on certain events. Such an event might be the start of the automation system, an interrupt, or detection of a program error (Figure 3.1). The programs allocated to the events are divided into priority classes, which determine the program processing order (mutual interruptibility) when several events occur.

The lowest-priority program is the main program, which is processed cyclically by the CPU. All other events can interrupt the main program at any location, the CPU then executes the associated interrupt service routine or error handling routine and returns to the main program.

A specific organization block (OB) is allocated to each event. The organization blocks represent the priority classes in the user program. When an event occurs, the CPU invokes the assigned organization block. An organization block is a part of a user program which you yourself may write.

Before the CPU begins processing the main program, it executes a startup routine. This routine can be triggered by switching on the mains power, by actuating the mode switch on the CPU's front panel, or via the programming device. Program processing following execution of the startup routine always starts at the beginning of the main program in the case of a cold restart or warm restart; in S7-400 systems, it is also possible to resume the program scan at the point at which it was interrupted (hot restart).

The main program is in organization block OB 1, which the CPU always processes. The start of the user program is identical to the first network in OB 1. After OB 1 has been processed (end of program), the CPU returns to the oper-

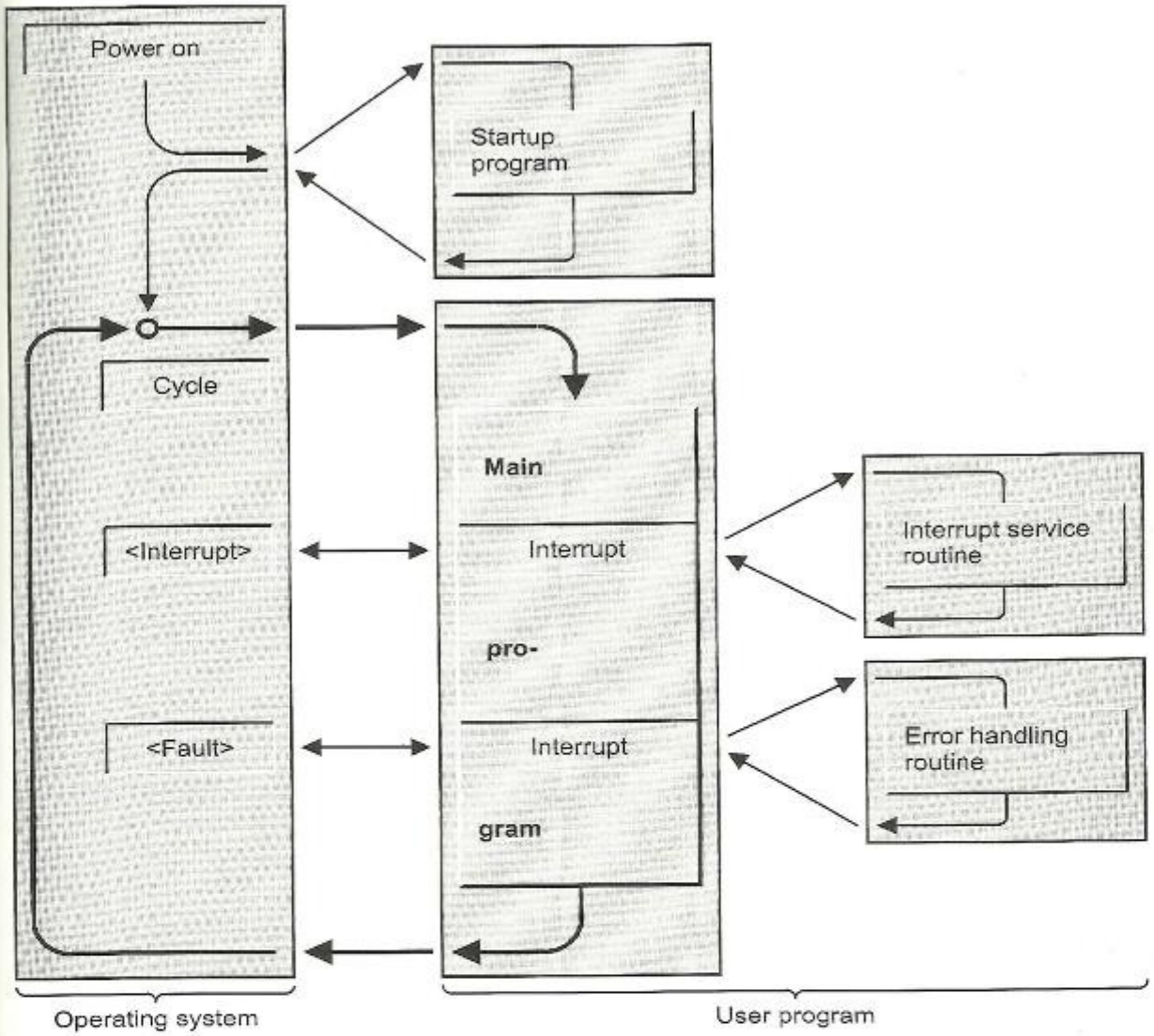


Figure 3.1 Methods of Processing the User Program

ating system and, after calling for the execution of various operating system functions, such as the updating of the process images, it once again calls OB 1.

Events which can intervene in the program are interrupts and errors. Interrupts can come from the process (hardware interrupts) or from the CPU (watchdog interrupts, time-of-day interrupts, etc.). As far as errors are concerned, a distinction is made between synchronous and asynchronous errors.

An asynchronous error is an error which is independent of the program scan, for example failure of the power to an expansion unit or an interrupt that was generated because a module was being replaced.

A synchronous error is an error caused by program processing, such as accessing a non-existent address or a data type conversion error. The type and number of recorded events and the associated organization blocks are CPU-specific; not every CPU can handle all possible STEP 7 events.

3.1.2 Priority Classes

Table 3.1 lists the available SIMATIC S7 organization blocks, each with its priority. In some

priority classes, you can change the assigned priority when you parameterize the CPU. The Table shows the lowest and highest possible priority classes; each CPU has a different low/high range; a specific CPU occupies a section of this overview.

Organization block OB 90 (background processing) executes alternately with organization block OB 1, and can, like OB 1, be interrupted by all other interrupts and errors.

The startup routine may be in organization block OB 100 (warm restart), OB 101 (hot restart) or OB 102 (cold restart), and has priority 27. Asynchronous errors occurring in the startup routine have priority class 28. Diagnostic interrupts are regarded as asynchronous errors.

You determine which of the available priority classes you want to use when you parameterize the CPU. Unused priority classes (organization blocks) must be assigned priority 0.

The relevant organization blocks must be programmed for all priority classes used; otherwise the CPU will invoke OB 85 "Program Processing Error" or go to STOP.

For each priority class selected, temporary local data (L stack) must be available in sufficient volumes (see Chapter 18.1.5 "Temporary Local Data" for more details).

3.1.3 Specifications for Program Processing

The CPU's operating system normally uses default parameters. You can change these defaults when you parameterize the CPU (in the Hardware object) to customize the system to suit your particular requirements. You can change the parameters at any time.

Every CPU has its own specific number of parameter settings. The following list provides an overview of all STEP 7 parameters and their most important settings.

- General
  Name of CPU, plant identifier, location ID, settings for MPI interface (if the interface is not combined with DP), comment
- Startup
  Specifies the type of startup (cold restart, warm restart, hot restart); monitoring of Ready signals or module parameterization; maximum amount of time which may elapse before a warm restart
- Cycle/Clock Memory
  Enable/disable cyclic updating of the process image; specification of the cycle monitoring time and minimum cycle time; amount of cycle time, in percent, for communication; number of the clock memory byte; size of the process images
- Retentive Memory
  Number of retentive memory bytes, timers and counters; specification of retentive areas for data blocks
- Memory
  max. number of temporary local data in the priority classes (organization blocks); max size of the L stack and number of communications jobs
- Interrupts
  Specification of the priority for hardware interrupts, time-delay interrupts, asynchronous errors and DPV1 interrupts; assignment of partial process images with process and time delay interrupts
- Time-of-Day Interrupts
  Specification of the priority and assignments of partial process images; specification of the start time and periodicity
- Cyclic Interrupts
  Specification of the priority, the time cycle and the phase offset; assignment of partial process images
- Synchronous cycle Interrupts
  Specification of the priority; assignment of DP master system and partial process images
- Diagnostics/Clock
  Specification of the system diagnostics; type and interval for clock synchronization, correction factor
- Protection
  Specification of the protection level; defining a password; setting of process or debug mode
- Multicomputing
  Specification of the CPU number

Table 3.1 SIMATIC S7 Organization Blocks

Organization block	Called	Priority	
		Default	Modifiable
Free cycle OB 1	Cyclically via the operating system	1	No
TOD interrupts OB 10 to OB 17	At a specific time of day or at regular intervals (e.g. monthly)	2	0, 2 to 24
Time-delay interrupts OB 20 to OB 23	After a programmable time, controlled by the user program	3 to 6	0, 2 to 24
Watchdog interrupts OB 30 to OB 38	Regularly at programmable intervals (e.g. every 100 ms)	7 to 15	0, 2 to 24
Process interrupts OB 40 to OB 47	On interrupt signals from I/O modules	16 to 23	0, 2 to 24
DPV1 interrupts OB 55 to OB 57	With status, update or vendor alarms from PROFIBUS DPV1 slaves	2	0, 2 to 24
Multiprocessor interrupt OB 60	Event-driven via the user program in multiprocessor mode	25	No
Synchronous cycle interrupts OB 61 to OB 64	With Synchronous cycle interrupt of DP master (synchronous with DP cycle)	25	0, 2 to 26
Technology sync interrupt OB 65	Synchronous following updating of the technology data blocks of a CPU 317T	25	No
Redundancy error interrupts OB 70, OB 72, OB 73	In the case of loss of redundancy resulting from I/O errors, In the case of CPU redundancy error In the case of communications redundancy error	25 28 25	2 to 26 2 to 28 2 to 26
Asynchronous error interrupts OB 80 OB 81 to OB 84, 86, 87 OB 85 OB 88	In the case of errors not involved in program execution (e.g. time error, SE error, diagnostics interrupt, insert/remove module interrupt, rack/station failure, processing abort)	 26 [2] 25 [2] 25 [2] 28	 No 2 to 26 24 to 26 No
Background processing OB 90	Minimum cycle time duration not yet reached	29 [1]	No
Startup routine OB 100, OB 101, OB 102	At programmable controller startup	27	No
Synchronous errors OB 121, OB 122	In the case of errors connected with program execution (e.g. I/O access error)	Priority of the OBs causing the errors	

[1] see text [2] at startup: 28

- ▷ Integrated function
  Activation and parameterization of the integrated functions
- ▷ Communications
  Definition of connection resources
- ▷ Web
  Activation of the Web server, language selection

On startup, the CPU puts the user parameters into effect in place of the defaults, and they remain in force until changed.

Program length, memory requirements

The memory requirements of a compiled block are listed in the block properties. If you select the block in the SIMATIC Manager, and then select the "General - Part 2" tab using EDIT →

OBJECT PROPERTIES, you will be provided with the load and work memory requirements for this block.

The length of the user program is listed in the properties of the offline *Blocks* container (select *Blocks* and EDIT → OBJECT PROPERTIES). On the "Blocks" tab you will find the data "Size in work memory" and "Size in load memory".

Note that the configuration data (system data blocks) are missing in the value for the load memory. The SIMATIC Manager shows you blocks in the detail view with the container open (displayed as table), and the memory requirements in the status line (bottom right in window) with the *System data* object selected.

With the programming device switched online, the current occupation of the CPU memory is shown by the SIMATIC Manager under PLC → DIAGNOSTIC/SETTING → MODULE INFORMATION, "Memory" tab.

Checksum

The program editor generates a checksum for all blocks of the user program, and stores it in the object properties of the *Blocks* container. Identical programs have the same checksum, each change in the program also changes the checksum. A checksum is also generated from the system data. You can view the checksums in the SIMATIC Manager with the *Blocks* container selected and EDIT → OBJECT PROPERTIES.

The checksum of the user program is generated from the program code and the default and initial values of the data blocks. The writing of data addresses in the work memory (actual values) does not change the checksum. The checksum is only adapted when the data blocks are uploaded to the offline data management, when the actual values become the initial values. This also applies to the data blocks generated by a system function.

If a data block generated by system functions is written or deleted, the checksum is not changed. The checksum is adapted if a programmed (loaded) data block is deleted, or if the initial values in the load memory are modified by the system function SFC 84 WRIT_DBL.

Module ID

Innovated S7-CPUs, PROFIBUS DPV1 slaves and PROFINET IO devices can support functions for identification & maintenance (I&M functions). For example, you can provide a station with a higher-level item designation and a location ID and evaluate it later in the program. The higher-level item designation is used to identify parts of the plant uniquely according to their function. The location ID is part of the item designation and describes, for example, the precise location of a SIMATIC device within a process plant.

To enter the I&M data, select the module in the Hardware Configuration, then select EDIT → OBJECT PROPERTIES, and then – with an appropriately designed module – you can enter the higher-level item designation and the location ID on the "General" tab or the "Identification" tab. In online mode, you select the module and can then exchange the I&M data between offline data management and the module with PLC→ DOWNLOAD MODULE IDENTIFICATION or PLC → UPLOAD MODULE IDENTIFICATION TO PG .

To analyze the I&M data, use SFC 51 RDSYSST to read the system status list with the system status list ID 16#011C Index 16#0003 for the higher-level item designation and Index 16#000B for the location ID.

3.2 Blocks

You can divide your program into as many sections as you want to in order to make it easier to read and understand. The STEP 7 programming languages support this by providing the necessary functions. Each program section should be self-contained, and should have a technological or functional basis. These program sections are referred to as "Blocks". A block is a section of a user program which is defined by its function, structure or intended purpose.

3.2.1 Block Types

STEP 7 provides different types of blocks for different tasks:

- User blocks
  Blocks containing user program and user data
- System blocks
  Blocks containing system program and system data
- Standard blocks
  Turnkey, off-the-shelf blocks, such as drivers for FMs and CPs

User blocks

In extensive and complex programs, "structuring" (dividing) of the program into blocks is recommended, and in part necessary. You may choose among different types of blocks, depending on your application:

Organization blocks (OBs)

These blocks serve as the interface between operating system and user program. The CPU's operating system calls the organization blocks when specific events occur, for example in the event of a hardware or time-of-day interrupt. The main program is in organization block OB 1. The other organization blocks have permanently assigned numbers based on the events they are called to handle.

Function blocks (FBs)

These blocks are parts of the program whose calls can be programmed via block parameters. They have a variable memory which is located in a data block. This data block is permanently allocated to the function block, or, to be more precise to the function block call. It is even possible to assign a different data block (with the same data structure but containing different values) to each function block call. Such a permanently assigned data block is called an instance data block, and the combination of function block call and instance data block is referred to as a call instance, or "instance" for short. Function blocks can also save their variables in the instance data block of the calling function block; this is referred to as a "local instance".

Functions (FCs)

Functions are used to program frequently recurring or complex automation functions. They can be parameterized, and return a value (called the function value) to the calling block. The function value is optional, in addition to the function value, functions may also have other output parameters. Functions do not store information, and have no assigned data block.

Data blocks (DBs)

These blocks contain your program's data. By programming the data blocks, you determine in which form the data will be saved (in which block, in what order, and in what data type). There are two ways of using data blocks: as global data blocks and as instance data blocks. A global data block is, so to speak, a "free" data block in the user program, and is not allocated to a code block. An instance data block, however, is assigned to a function block, and stores part of that function block's local data.

The number of blocks per block type and the length of the blocks is CPU-dependent. The number of organization blocks, and their block numbers, are fixed; they are assigned by the CPU's operating system. Within the specified range, you can assign the block numbers of the other block types yourself. You also have the option of assigning every block a name (a symbol) via the symbol table, then referencing each block by the name assigned to it.

System blocks

System blocks are components of the operating system. They can contain programs (system functions (SFCs) or system function blocks (SFBs)) or data (system data blocks (SDBs)). System blocks make a number of important system functions accessible to you, such as manipulating the internal CPU clock, or various communications functions.

You can call SFCs and SFBs, but you cannot modify them, nor can you program them yourself. The blocks themselves do not reserve space in user memory; only the block calls and the instance data blocks of the SFBs are in user memory.

SDBs contain information on such things as the configuration of the automation system or the

Table 3.2 Number ranges for system data blocks

SDB No.	Meaning, Contents
0	CPU parameters which control the response of the operating system and the CPU-internal default settings; overwrites the SDB 2 when transmitting to the CPU
1	Module addressing for central I/O (preset configuration), e.g. assignment of logical and geographical addresses, address space of modules, etc.
2	CPU parameters (default settings in CPU's operating system, become effective following overall reset if a configuration has not yet been transmitted)
3 to 7	Miscellaneous CPU and module parameters, e.g. for saving consistency of transmitted configuration data
20 to 89	Module addressing for distributed I/O (preset configuration), e.g. assignment of logical and geographical addresses, address space of modules, etc.
90 to 99	Configuration data for fault-tolerant and failsafe systems
100 to 149	Parameters for central and distributed modules assigned to the CPU
150 to 152	Parameters for interface submodules
153 to 189	Parameters for distributed modules
200 to 998	Parameters for configuration of communications (e.g. global data communications, symbol-based messages, configuration of connections)
999	Configuration data for routing of connections
1000 and greater	Parameters for distributed I/O, parameters for CP and FM modules, parameters for H/F and TD/OP systems

parameterization of the modules. STEP 7 itself generates and manages these blocks. You, however, determine their contents, for instance when you configure the stations. As a rule, SDBs are located in load memory. You cannot open SDBs, and can only read them from your program using special system blocks, e.g. when parameterizing modules.

Double click the *System blocks* object in the *Blocks* container to display a list of current system data blocks generated by the Hardware Configuration tool (in the offline container) or present on the CPU (in the online container). Table 3.2 shows an overview of the numbering system for system data blocks.

Standard blocks

In addition to the functions and function blocks you create yourself, off-the-shelf blocks (called "standard blocks") are also available. They can either be obtained on a storage medium or are contained in libraries delivered as part of the STEP 7 package (for example IEC functions, or functions for the S5/S7 converter).

Chapter 25 "Block Libraries" contains an overview of the standard blocks supplied in the *Standard Library*.

3.2.2 Block Structure

Essentially, code blocks consist of three parts (Figure 3.2):

- The block header,
  which contains the block properties, such as the block name
- The declaration section,
  in which the block-local variables are declared, that is, defined
- The program section,
  which contains the program and program commentary

A data block is similarly structured:

- The block header contains the block properties
- The declaration section contains the definitions of the block-local variables, in this case the data addresses with data type specification

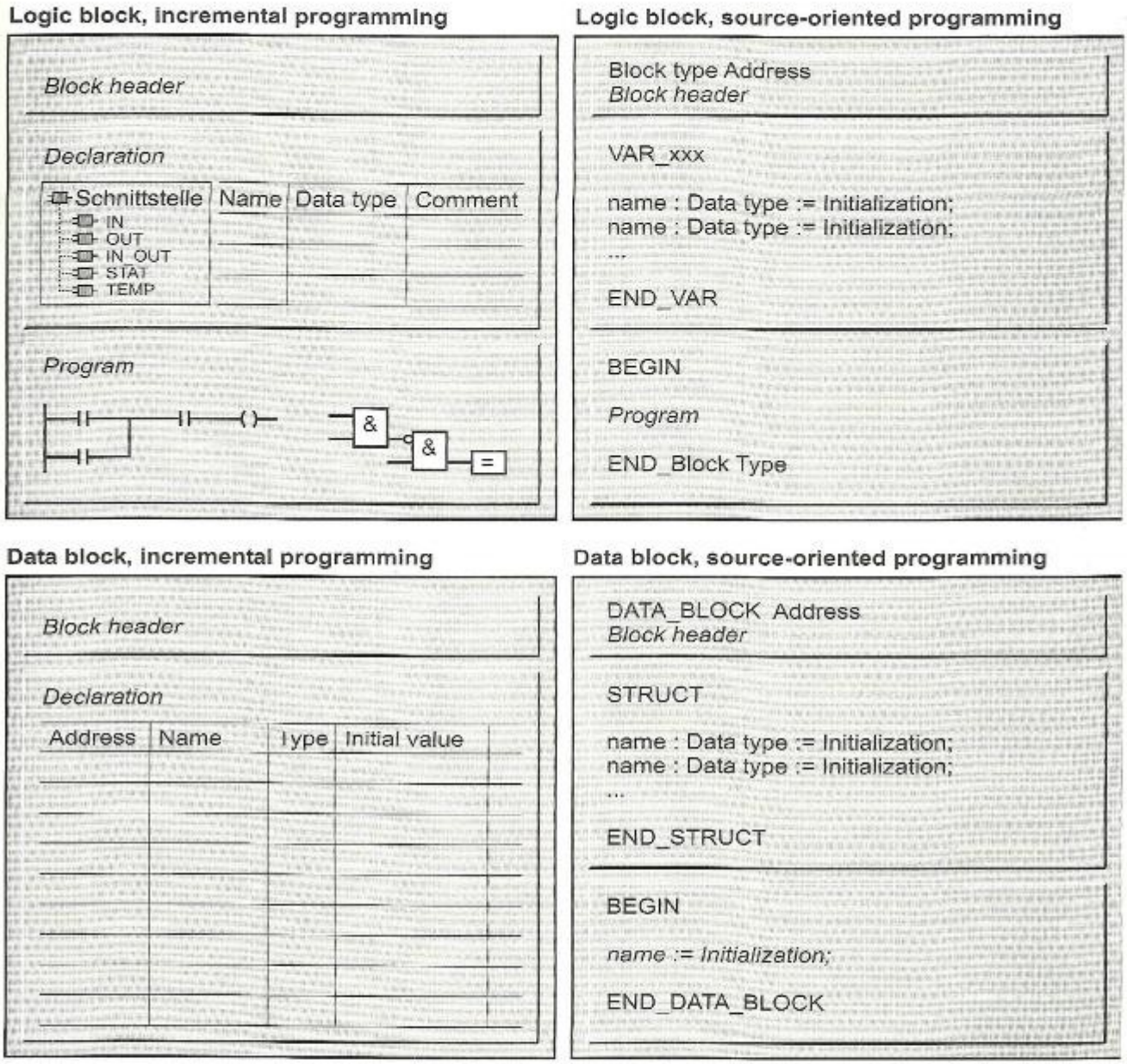


Figure 3.2 Structure of a Block

- The initialization section, in which initial values can be specified for individual data addresses

In incremental programming, the declaration section and the initialization section are combined. You define the data addresses and their data types in the "declaration view", and you can initialize each data address individually in the "data view".

3.2.3 Block Properties

The block properties, or attributes, are contained in the block header. You can view and modify the block properties with the menu command EDIT → OBJECT PROPERTIES in the SIMATIC Manager when the block is selected, or with FILE → PROPERTIES in the Program Editor (Figure 3.3).

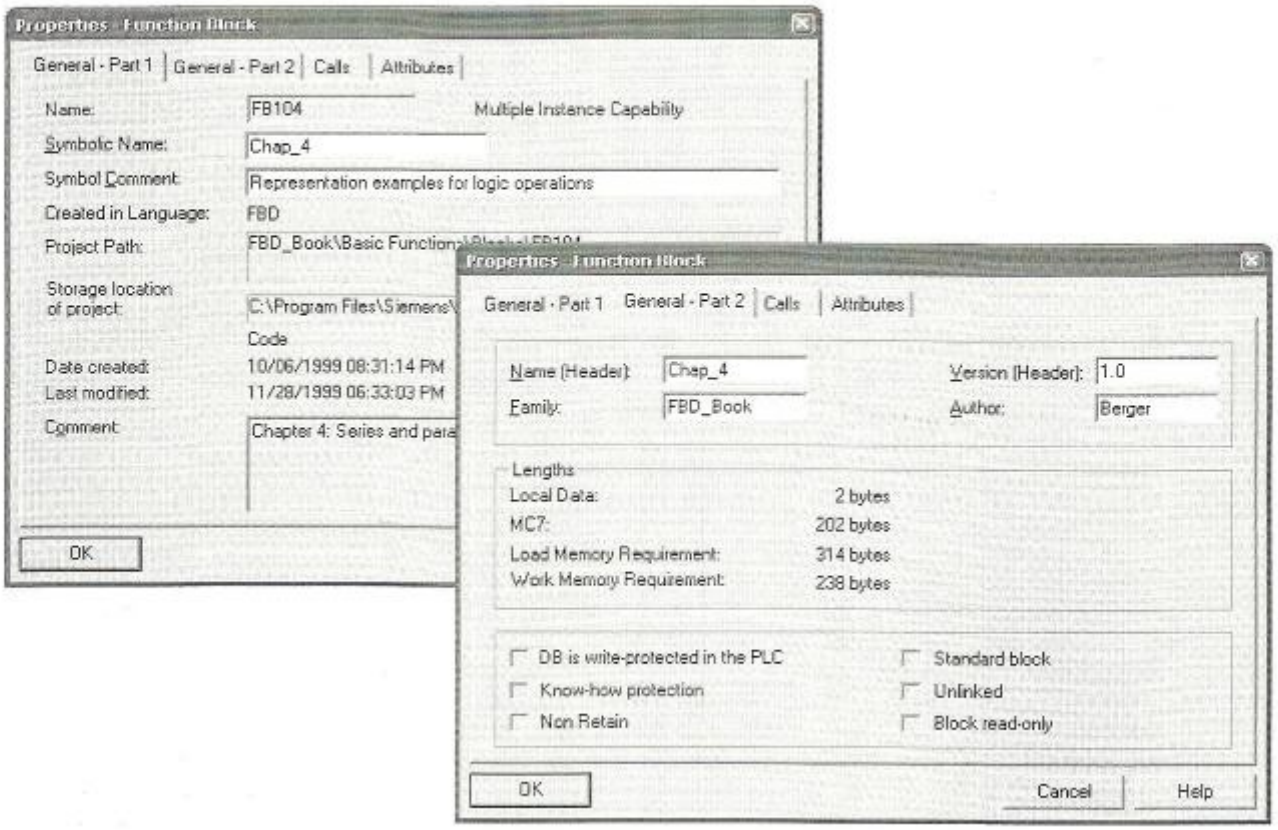


Figure 3.3 Structure of a Block

"General - Part 1" tab

Under *Name*, this tab contains the block's absolute address with block type and number, as well as the symbolic name and symbol comment from the symbol table.

In the case of function blocks, an indication next to the name shows whether the block has a multi-instance capability. If the *multi-instance capability* is switched on, which is normally the case, you can call the block as a local instance, and can also call further function blocks with multi-instance capability within it as local instances. You can deselect the *multi-instance capability* when creating the function block; with a source-oriented program input, the keyword for deselecting is CODE_VERSION1. The advantage of a function block "without multi-instance capability" is the unlimited application of instance data during indirect addressing, which is only of significance with STL programming.

The tab also shows the creation language of the block (which you set when creating the block), and the memory locations of the block and the project.

The program editor saves the creation or modification date of the block in two time stamps: these are the block parameters and the static local data for the program code and the interface. Note that the modification date of the interface must be equal to or smaller (older) than the modification date of the program code in the calling block. If this is not the case, the program editor signals a "time stamp conflict" during output of the calling block.

The comment displayed consists of the block title and the block comment which you entered when programming the block.

"General - Part 2" tab

The *Name (Header)* displayed on this tab is the block name; it is not identical to the symbol address. Different blocks can have the same name. Using *Family* you can assign a common feature to a group of blocks. The block name

and family are displayed when inserting blocks if you select the block in the dialog window of the program element catalog. Use *Author* to enter the name of the block's creator. The name, family and author may have up to eight characters each, commencing with a letter. The letters, digits and the underline character are permissible. The *Version* is entered with two 2-digit numbers from 0 to 15.

The length data show the memory allocation for the block in bytes:

- Local data: allocation in the local data stack (temporary local data)
- MC 7: size of the block (code only)
- Load memory requirement
- Work memory requirement

A block occupies more space in the load memory since the data not relevant to processing are saved here in addition.

The *Know-how protection* attribute is used for block protection. If a block is know-how-protected, the program in that block can not be viewed, printed out or modified. The Editor shows only the block header and the declaration table with the block parameters. You can assign this attribute during source-oriented input of the block with the keyword KNOW_HOW_PROTECT. When you do this to a block, no one can view the compiled version of that block, not even you (make sure you keep the source file in a safe place!).

The attribute *DB is write-protected in the PLC* is an attribute for data blocks only. It means that you can only read that data block in your program. Output of an error message prevents the overwriting of the data in that data block. The write protection applies to the data relevant to processing (actual values) in work memory; the data in load memory (initial values) can be overwritten even if the data block is write-protected. This write protection feature must not be confused with block protection. A data block with block protection can be read out and written to in the user program, but its data can no longer be viewed with a programming or operator monitoring device. The attribute *DB is write-protected in the PLC* is switched off as standard, but can be changed at any time using the program editor. The keyword with source-oriented program input for switching on the write protection is READ_ONLY.

The block header of any standard block which comes from Siemens contains the *Standard block* attribute.

Data blocks can be assigned the *Unlinked* attribute. Such data blocks are only present in load memory, and are not relevant to processing. Since their data are not located in work memory, direct access is no longer possible. Data in load memory can be read using system functions and – if the load memory is a micro memory card – also written. Data blocks with the *Unlinked* attribute are suitable for recording data which are only rarely accessed, e.g. recipes or archives. This attribute is switched off as standard, but can be changed at any time using the program editor. The keyword with source-oriented program input for switching on this attribute is UNLINKED.

The *Non retain* attribute means "non-retentive" and is assigned to CPUs designed for this for data blocks. If *Non retain* is switched on, the data block transfers the initial values from load memory to work memory in the event of a power off/on and with a RUN-STOP transition (response as with a cold restart). If *Non retain* is switched off, the corresponding data block therefore being retentive, it retains its actual values in the event of a power off/on and with a RUN-STOP transition (response as with a warm restart). This attribute is switched off as standard, but can be changed at any time using the program editor. The keyword with source-oriented program input for switching on this attribute is NON_RETAIN.

Blocks saved in the Program Editor with the menu command FILE → STORE READ-ONLY for reference purposes, for example, receive the block property *Block read-only*. These can be all code blocks, data blocks, and user-defined data types. This property can only be set with the Program Editor, and there is no keyword for source-oriented programming for this purpose.

"Calls" tab

This tab shows a list of all blocks called in this block with the time stamps for the code and the interface. With instance data blocks, the basic function block is shown here together with the

local instances (function blocks) called in this instance, in each case with the time stamps for code and interface.

"Attributes" tab

Blocks may have system attributes. System attributes control and coordinate functions between applications, for example in the SIMATIC PCS7 control system.

3.2.4 Block Interface

The declaration table contains the interface of the block to the rest of the program. It consists of the block parameters (input, output and in/out parameters) and also – in the case of function blocks – the static local data. The temporary local data, which do not basically belong to the block interface, are also handled at this point. The block interface is defined in the interface window when programming the block, and is initialized with variables when the block is called (see Chapter 19 "Block Parameters").

The Program Editor checks that the block parameter initialization in the called block agrees with the interface of the called block. The Editor uses the time stamp for this: the interface of the called block must be older than the code in the calling block, that is, the last interface modification must have been made prior to integration of the block. The Program Editor updates the interface time stamp when the number of parameters changes or when a data type or a default value changes.

Time stamp conflict

A time stamp conflict occurs when the interface of the called block has a later time stamp than the code of the calling block. You will notice a time stamp conflict if you open an already compiled block again. The Program Editor then indicates the incorrect block call in red. A time stamp conflict can be caused, for example, if you modify the interfaces of blocks that are already called in other blocks, or if you combine blocks from different programs into a new program, or if you re-compile a section of the overall program with a source file.

However, the interface conflict generally described as a "time stamp conflict" can also have other causes. It also occurs if a called or referenced block is younger than the calling block. Examples of the occurrence of time stamp conflicts include the following:

- The interface of a called block is younger than the code of the calling block.
- The interface initialization does not agree with the block interface.
- A function block is younger than its instance data block (the instance data block is generated from the interface description of the function block and should therefore be younger than or the same age as the function block).
- A local instance is younger than the calling instance (affects function blocks).
- A user data type UDT is younger than the block whose variables are declared with the UDT; this can be any block including a data block or another UDT.

Correcting invalid block calls

The Program Editor supports you in different manners in finding and correcting invalid block calls. See the next section for how to check the block consistency in a complete program ("Checking block consistency").

You can check block calls which have become invalid with the block open (with the cursor at the invalid block call) using the menu command EDIT → BLOCK CALL → UPDATE. Block calls can become invalid following insertion, deletion or shifting of block parameters, or when changing the name and type.

With EDIT → BLOCK CALL → CHANGE TO MULTI-INSTANCE CALL and EDIT → BLOCK CALL → CHANGE TO FB/DB CALL you transfer calls from function blocks into local instance calls or into calls with data block. Following modification of the block calls, you must regenerate the associated instance data blocks.

A further possibility is provided by the menu command FILE → CHECK AND UPDATE ACCESSES. The invalid block calls in an opened block are then updated or presented for modification.

Checking block consistency

The Program Editor only indicates a time stamp conflict when you open a block containing a time stamp conflict. If you want to check an entire program, you can use the function "Check block consistency" in the SIMATIC Manager an. This purges a majority of interface conflicts and directs you to the program locations that require editing.

To carry out a consistency check, select the *Blocks* container in the SIMATIC Manager and then EDIT → CHECK BLOCK CONSISTENCY. If a call tree is not displayed in the window, e.g. because the program has been compiled using an earlier version of STEP 7, select PROGRAM → COMPILE in this window.

Please note that after checking the block consistency, the instance data blocks and the data blocks generated from the UDT are assigned the *initial* values again in the compiled program.

The Program Editor displays the progress and result of the consistency check in the output window (VIEW → ERRORS AND WARNINGS). The consistency check cannot be used on programs in libraries.

The dependencies in the case of called or referenced blocks are displayed in the form of a tree diagram (Figure 3.4). You can choose between the following two representations:

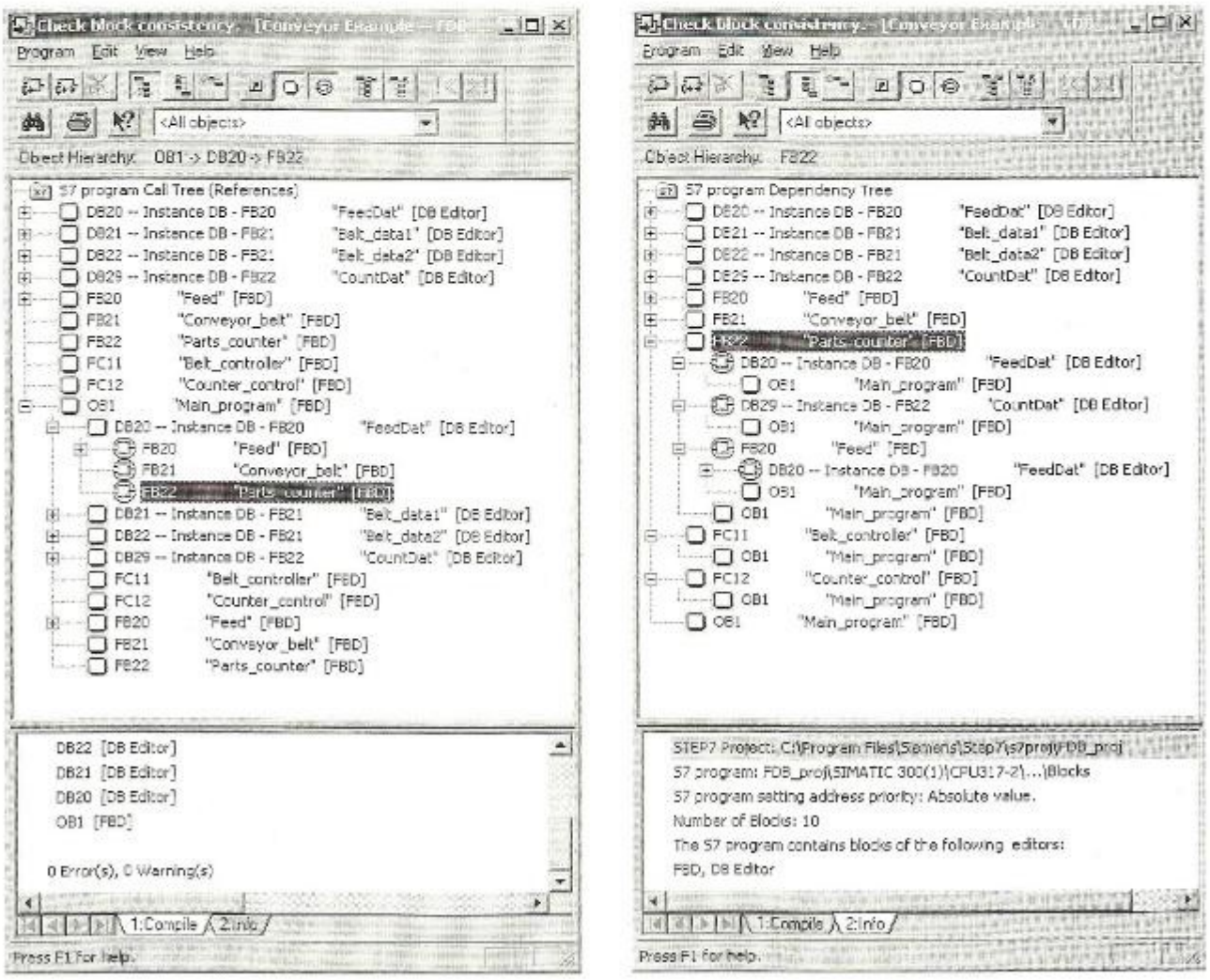


Figure 3.4 Example of the Representation of the "Check Block Consistency" Dependencies

The *reference tree* view displays the dependencies in a similar way to the program structure: on the left are the calling blocks, further to the right are the blocks called in the blocks on the left. Example: instance DB 20/FB 20 is called in OB 1 and local instances FB 21 and FB 22 are called in FB 20.

The *dependency tree* view displays the dependencies starting from all called or referenced blocks. They are located in the left-hand column, and the calling blocks are listed to the right of this. Example: FB 22 stores its data in instance DB 20/FB 20 that is called in OB 1. It also has its own DB 29 and it is called as a local instance in FB 20.

The determined information is displayed in compact form by symbols. An exclamation mark, for example, indicates that the object caused a time stamp conflict. A white cross on a red background indicates that the associated block must be recompiled.

If you select a block in the tree diagram or in the output window, you can edit it with EDIT → OPEN OBJECT, e.g. correct an incorrect call.

3.3 Programming Code Blocks

Chapter 2.5 "Creating the S7 Program" contains an introduction to program creation and to operating the program editor.

3.3.1 Opening Blocks

You begin block programming by opening a block. Open an existing block either by double-clicking on the block in the SIMATIC Manager's project window or by selecting FILE → OPEN in the program editor.

If you open a compiled block in the *Blocks* container, e.g. by double-clicking, it is open for incremental programming. This is the case both with offline and online programming.

If the block does not yet exist, you can generate it in the following ways:

- ▷ In the SIMATIC Manager by selecting the *Blocks* object in the left half of the project window and generating a new block with INSERT → S7 BLOCK → ... . You are provided with the properties window of the block. On the "General – Part 1" tab, select the number of the block under *Name* and the language "LAD" or "FBD". You can enter the remaining attributes later.
- ▷ In the Editor with menu command FILE → NEW, which displays a dialog box in which you can enter the desired block under *object name*. After closing the dialog box you can program the contents of the block. The Program Editor uses the language set on the "Create Block" tab under OPTIONS → CUSTOMIZE.

You can enter the information for the block header when you generate the block or you can enter the block attributes later in the Editor by opening the block and selecting the menu command FILE → PROPERTIES.

3.3.2 Block Window

The program editor shows the variable declaration table (block parameters and local data) and the program window (code and comments) for an opened code block. The program elements cannot be additionally displayed in the overview window (Figure 3.5).

Variable declaration table

The variable declaration table is in the window above the program window. If it is not visible, position the mouse pointer to the upper line of demarcation for the program window, click on the left mouse button when the mouse pointer changes its form, and pull down. You will see the overview of the types of variable on the left, and the variable declaration table on the right, which is where you define the block-local variables (see Table 3.3).

In order to declare a variable, select its type in the left area, and fill in the table on the right. Not every type of variable can be programmed in every kind of code block. If you do not use a variable type, the corresponding line remains empty.

The declaration for a variable consists of the name, the data type, a default value, if any, and a variable comment (optional). Not all variables can be assigned a default value (for instance, it

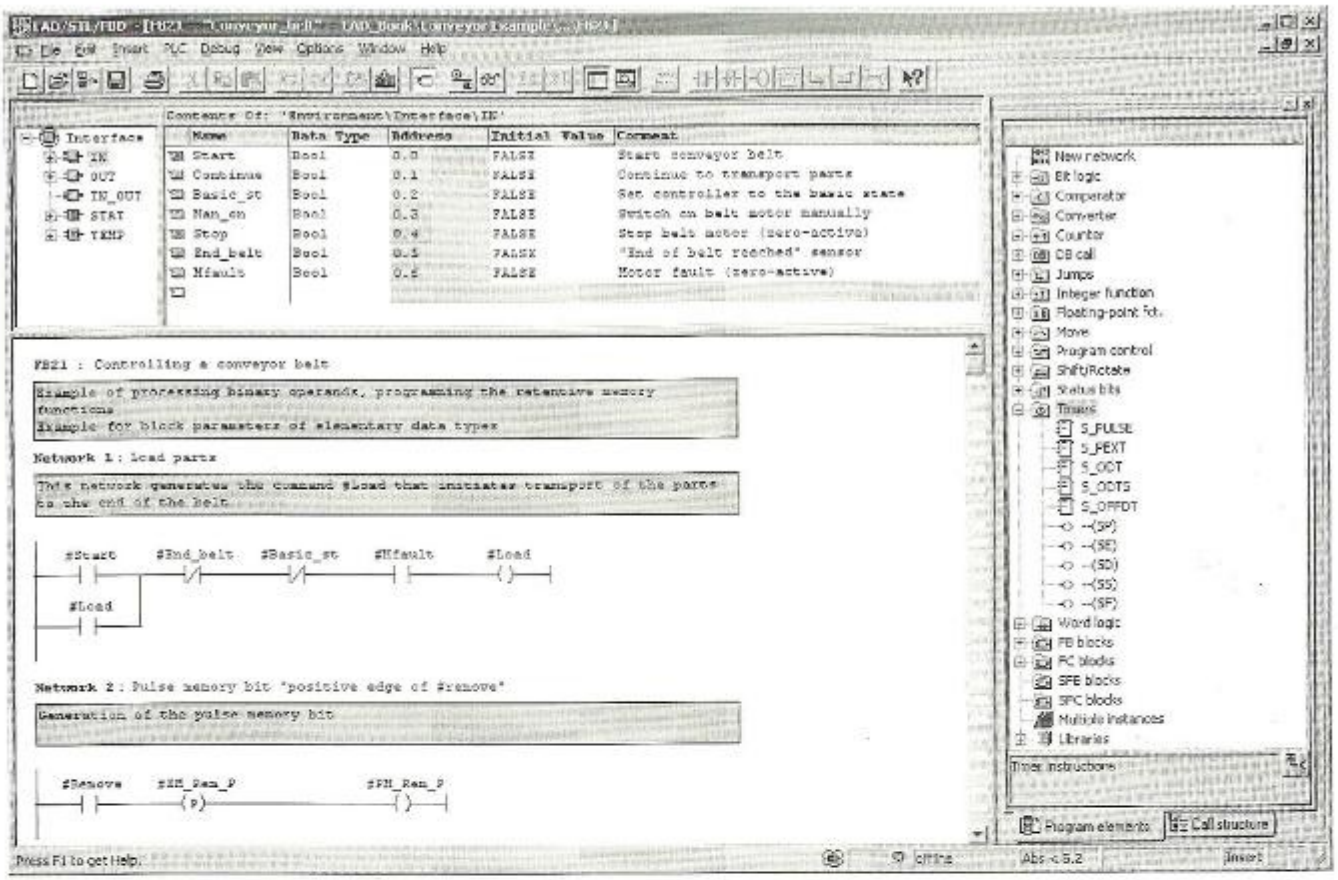


Figure 3.5 Example of an Opened LAD Block

is not possible for temporary local data). The default values for functions and function blocks are described in detail in Chapter 19 "Block Parameters".

The order of the declarations in code blocks is fixed (as shown in the table above), while the order within a variable type is arbitrary. You can save room in memory by bundling binary variables into blocks of 8 or 16 and BYTE variables into pairs. The Editor stores a (new) BOOL or BYTE variable at a byte boundary and a variable of another data type at a word boundary (beginning at a byte with an even address).

Program window

In the program window, you will see – depending on the Editor's default settings – the fields for the block title and the block comment and,

Table 3.3 Variable Types in the Declaration Section

Variable Type	Declaration	Possible in Block Type		
Input parameters	IN	-	FC	FB
Output parameters	OUT	-	FC	FB
In-out parameters	IN_OUT	-	FC	FB
Static local data	STAT	-	-	FB
Temporary local data	TEMP	OB	FC	FB
Function value	RETURN	-	FC	-

if it is the first network, the fields for the network title, the network comment, and the field for the program entry. In the program section of a code block, you control the display of comments and symbols with the menu commands VIEW → DISPLAY WITH. You can change the size of the display with VIEW → ZOOM IN, VIEW → ZOOM OUT and VIEW → ZOOM FACTOR.

3.3.3 Overview Window

The overview window contains the program element catalog and the call structure.

If the overview window is not visible, fetch it onto the screen with VIEW → OVERVIEWS or with INSERT → PROGRAM ELEMENTS.

The overviews are presented in a separate window which you can "dock" at the edge of the editor window and also release again (double-click in each case on the title bar of the overview window).

Program elements catalog

The program elements catalog supports programming in LAD and FBD by providing the available graphic elements (Figure 3.5). You can drag all program elements into the program window using the mouse.

In addition, it lists the blocks already located in the offline *Blocks* container, as well as the already-programmed multi-instances and the available libraries. By clicking with the right mouse button on a block or a block type, you can select whether the blocks are to be sorted according to type and number or according to the block family.

Call structure

The call structure shows the block hierarchy in the current user program. You are shown the call environment of the currently opened block together with the blocks used.

3.3.4 Programming Networks

You can divide a LAD/FBD program into networks which each represent a current path or a logic operation. The Editor numbers the networks automatically, beginning with 1. Each block can accommodate up to 999 networks. You may give each network a network title and a network comment. During editing, you can select each network directly with the menu command EDIT → GO TO → ... .

To enter the program code, click once below the window for the network comment, or, if you have set "Display with Comments", click once below the shaded area for network comments. You will see a framed empty window. You can begin entering your program anywhere within this window. The chapters below show you what a LAD current path or an FBD logic operation looks like.

You program a new network with INSERT → NETWORK. The Editor then inserts an empty network behind the currently selected network.

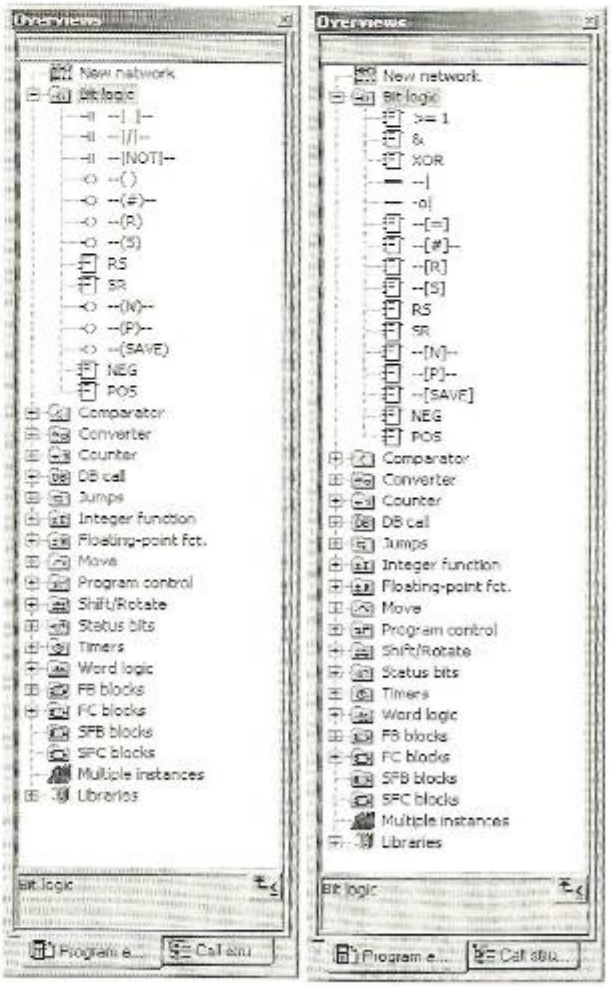


Figure 3.6
Program Elements Catalog for LAD and FBD

You need not terminate a block with a special statement, simply stop making entries. However, you can program a last (empty) network with the title "Block End", providing an easily seen visual end of the block (an advantage, particularly in the case of exceptionally long blocks).

In the Program Editor, you can create new blocks, or open and edit existing blocks without having to change back to the SIMATIC Manager.

Network templates

Just as you store blocks in a library to reuse them in other programs, you also save network templates in order to copy them again and again in, for example, other blocks.

To save network templates, create a library containing at least one S7 program and the *Sources* container.

You program the networks that you want to use as templates quite "normally" in (any) block. Then you replace the addresses that are to change with the dummy characters %00 to %99. You can also vary the network title and the network comment in this way.

The dummy characters replacing the addresses are presented in red because a block cannot be stored in this form. This is not significant because following saving of the network template(s), this block can be rejected (close the block without saving).

After entering the dummy characters, mark the network by clicking on the network number at top left before the network title. You can also combine several networks to form one template; hold down the Ctrl key while you click on further network numbers.

Now select EDIT → CREATE NETWORK TEMPLATE. In the dialog box that then appears, you can assign meaningful comments to the network and all the dummy characters. In the next dialog box, you assign a name for the network template and you define the storage location (*Sources* container in a library).

If you want to use the network templates, open the relevant library in the Program Elements Catalog and then select the desired network template (double-click or drag to the Editor window). A dialog box appears automatically and here you replace the dummy characters with valid entries. The network template is inserted after the selected network.

3.3.5 Addressing

The addresses used in the program, such as inputs and outputs or bit memories, are addressed in absolute or symbolic mode.

Absolute addressing

Absolute addressing references addresses and block parameters with the address ID and the bit/byte address. If there are three red question marks in the network in the place of addresses and parameters, you must replace this character string with valid addresses. If there are three black points, replacement is optional.

The Program Editor checks that the data types of the addresses and parameters are correct. You can deactivate some of these type checks (in the Program Editor under OPTIONS → CUSTOMIZE, "LAD/FBD" tab, "Type check for addresses" option).

Symbolic addressing

If you want to use symbolic names for global operands in incremental programming, these names must already be assigned to absolute addresses in the Symbol Table. While entering the program with the Program Editor, you can call up the Symbol Table for editing with OPTIONS → SYMBOL TABLE and then you can change symbols or enter new symbols.

You activate display of the symbol addresses with VIEW → DISPLAY WITH → SYMBOLIC REPRESENTATION. The menu point VIEW → DISPLAY WITH → SYMBOL INFORMATION provides, for each network, a list of the symbol-to-absolute-address assignments for each symbol used in the network.

While entering the symbols, you can view a list of all the symbols in the symbol table with INSERT → SYMBOL (or right mouse click and INSERT SYMBOL) and you can then transfer one of the symbols with a click of the mouse. The list is displayed automatically if you have set VIEW → DISPLAY WITH → SYMBOL SELECTION.

If a symbol is not yet included in the symbol table, you can select EDIT → SYMBOLS, make the assignment to the absolute address, and possibly also assign a symbol comment. This symbol is then transferred to the symbol table when you click OK.

You can also edit the symbols in Register "4: Address info" in the Details window. If the columns with the symbol and the symbol comment are not displayed, fetch them by clicking with the right mouse button on the address table and DISPLAY COLUMNS (ON/OFF).

If the Program Editor opens a compiled block, it carries out "decompiling" to the LAD or FBD method of representation. In doing so, it uses the non-execution-relevant program sections in the offline data management, in order, for example, to represent symbols, comments and jump labels. If the information from the offline data management system is missing, the Program Editor uses substitute symbols.

3.3.6 Editing LAD Elements

Programming in general

The program consists of individual LAD elements arranged in series or parallel to one another. Programming of a current path, or rung, begins on the left power rail. You select the location in the rung at which you want to insert an element, then you select the program elements you want

- with the corresponding function key (for example F2 for a normally open (NO) contact),
- with the corresponding button on the function bar or
- from the Program Elements Catalog (with INSERT → PROGRAM ELEMENTS or VIEW → OVERVIEWS).

You terminate a rung with a coil or a box.

Most program elements must be assigned memory locations (variables). The easiest way to do this is to first arrange all program elements, and then label them.

Contacts

Binary addresses such as inputs are scanned using contacts. The scanned signal states are combined according to the arrangement of the contacts in a serial or parallel layout.

"Current flows" through a *normally open contact* if the scanned binary address has signal state "1" (the contact is activated) "current

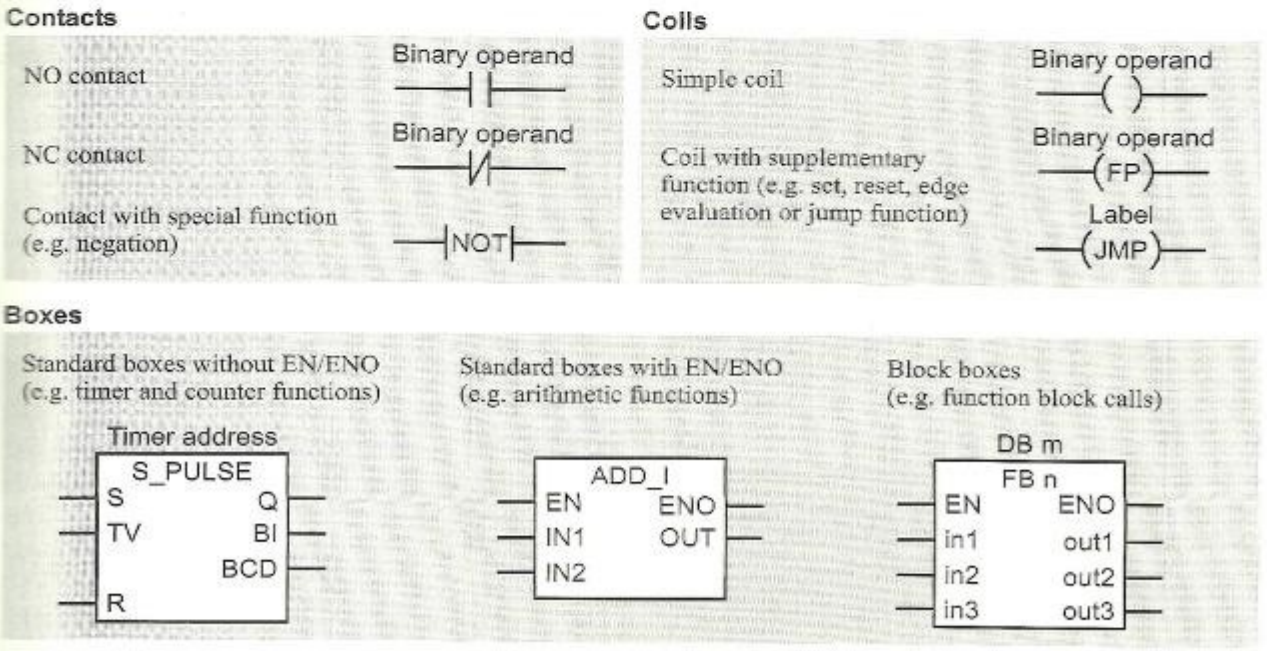


Figure 3.7 Examples of LAD Program Elements

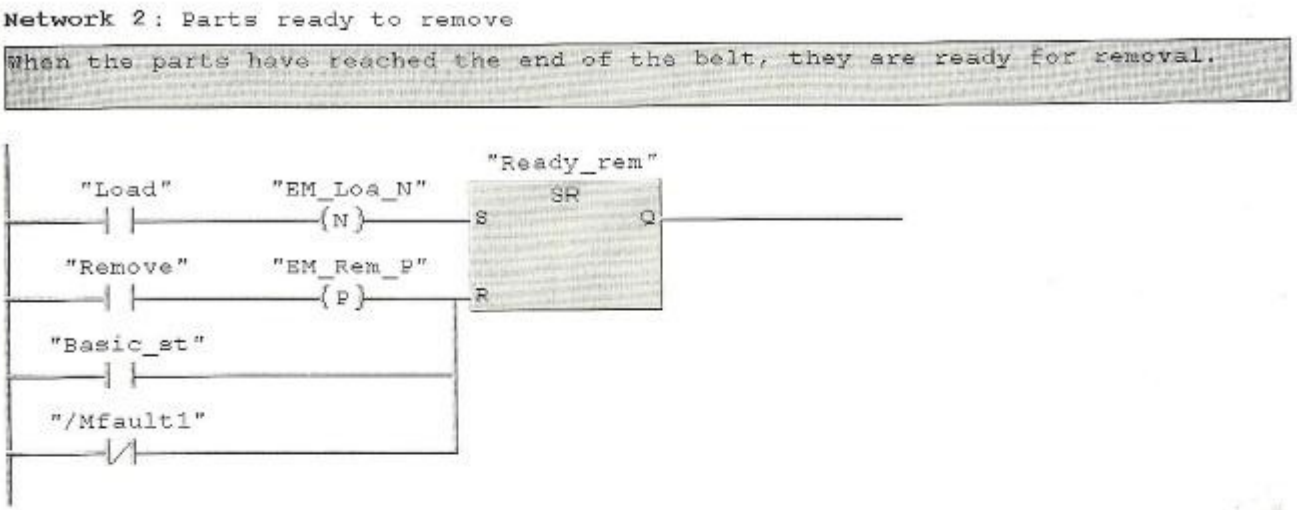


Figure 3.8 Example of a LAD Network in "3-Dimensional" Representation

flows" through a *normally closed contact* if the scanned binary address has signal state "0" (the contact is not activated). You can also scan status bits or negate the result of the logic operation (NOT contact).

Coils

Coils are used to control binary addresses, such as outputs. A simple coil sets the binary addresses when current flows in the coil, and resets it when power no longer flows.

There are coils with additional labels, such as Set and Reset coils, which serve a special function. You can also use coils to control timers and counters, call blocks without parameters, execute jumps in the program, and so on.

Boxes

Boxes represent LAD elements with complex functions. STEP 7 provides "standard boxes" of two different types: without EN/ENO mechanism (such as memory functions, timer and counter functions, comparison boxes), and with EN/ENO (such as MOVE, arithmetic and math functions, data type conversions). When you call code blocks (FCs, FBs, SFCs and SFBs), LAD also represents the calls as boxes with EN/ENO. LAD also provides an "empty box" in which you can enter the desired function when programming.

Layout restrictions

The LAD editor sets up a network according to the "main rung" principle. This is the uppermost branch, which begins directly on the left power rail and must terminate with a coil or a box. All LAD elements can be located in this rung. In parallel branches which do not begin on the left power rail, there are sometimes restrictions in the choice of program elements.

Additional restrictions dictate that no LAD element may be "short-circuited" with an "empty" parallel branch, and that no "power" may flow through an element from right to left (a parallel branch must be closed to the branch in which it was opened). Any further rules applying to the layout of special LAD elements are discussed in the relevant chapters.

When using boxes as program elements, you can

- program a single box per network
- arrange boxes in T branches in branches that start at the left power rail
- arrange boxes in series by switching the ENO output of one box to the EN input of the following box
- switch boxes in parallel in branches on the left power rail via its ENO output

With the arrangement of the boxes, you evaluate the signal states of the ENO outputs: if you

terminate the ENO outputs with a coil, "power" flows into the coil if all the boxes have all been processed without errors in the case of series connection, or if one of the boxes has been processed without errors in the case of parallel connection (see also Chapter 15.4 "Using the Binary Result").

3.3.7 Editing FBD Elements

Programming in general

The program consists of individual program elements that are connected together via the binary signal flow to form logic operations or networks. You begin programming a logic operation by selecting the programming elements on the left of the logic operation

- with the function key, (e.g. F2 for the AND function),
- via the menu (INSERT → FBD LANGUAGE ELEMENTS → AND BOX) or
- from the Program Elements Catalog (with INSERT → PROGRAM ELEMENTS or VIEW → OVERVIEWS).

You terminate a binary logic operation in the simplest case with an assign box.

Most program elements must be assigned memory locations (variables). The easiest way to do this is to first arrange all program elements, then label them.

Binary functions

You scan the binary addresses such as inputs and combine the scanned signal states using the binary functions AND, OR and exclusive OR. Each binary input of a box also scans the binary address at the input.

The scanning of an address can be negated so that scan result "1" can be obtained for status "0" of the address. You can also scan status bits or the result of a logic operation within a logic operation.

Simple boxes

You control binary addresses such as outputs with simple boxes. Simple boxes generally have only one input and may have an additional label.

There are simple boxes for controlling a binary address, evaluating an edge, controlling timer and counter addresses, calling blocks without parameters, executing jumps in the program, and so on.

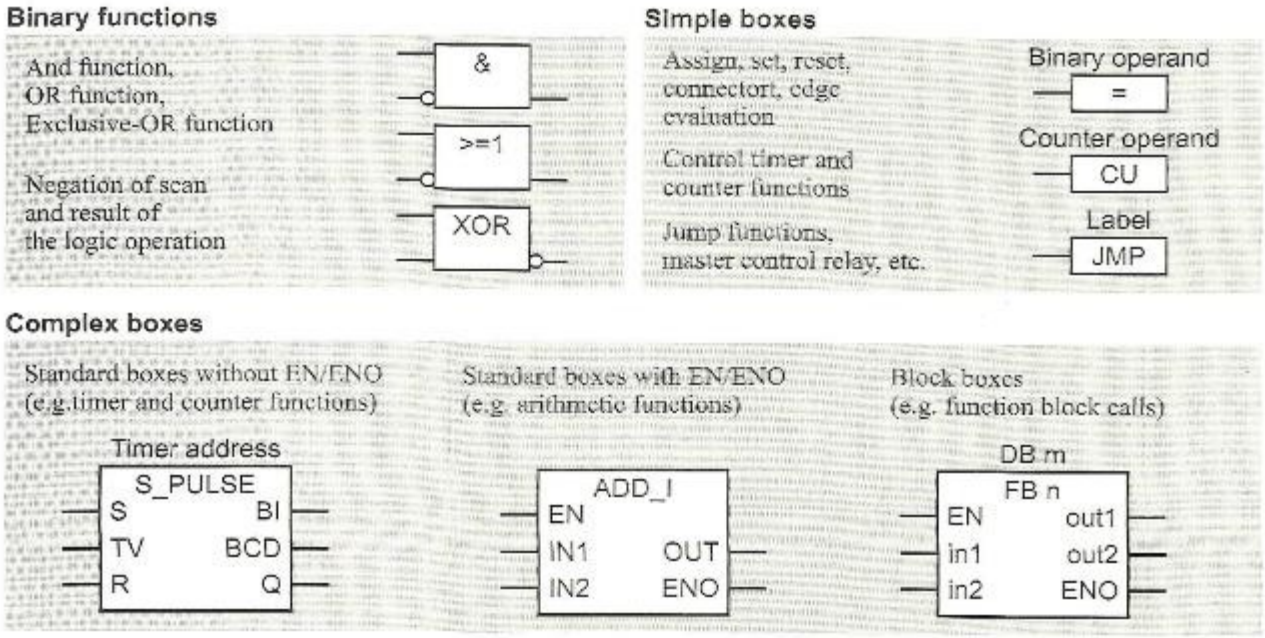


Figure 3.9 Examples of FBD Program Elements

Network 2 : Parts ready to remove

When the parts have reached the end of the belt, they are ready for removal.

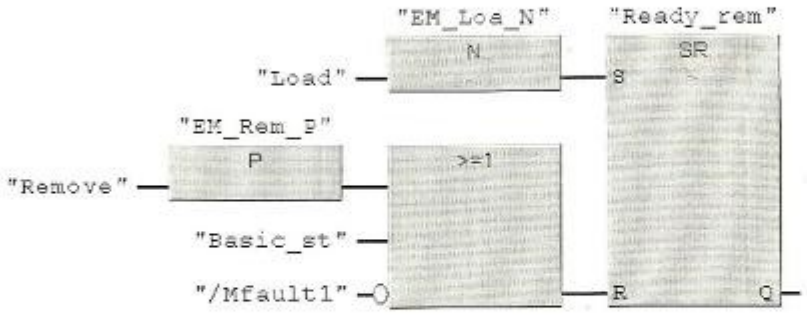


Figure 3.10 Example of an FBD Network in "3-Dimensional" Representation

Complex boxes

Complex boxes represent program elements with complex functions. STEP 7 provides "standard boxes" in two versions:

- without EN/ENO mechanism (such as memory functions, timers and counters, comparison boxes) and
- with EN/ENO (such as MOVE, arithmetic and math functions, data type conversion).

If you call code blocks (FCs, FBs, SFCs and SFBs), FBD also represents the calls as boxes with EN/ENO.

FBD also provides an "empty box" in which you can enter the desired function when programming.

Layout restrictions

The FBD editor sets up a network from left to right and from the top down. From the left, the inputs lead to the functions and the outputs exit to the right.

A logic operation always has a "terminating function". In its simplest form, this is an assignment of the result of the logic operation to a binary address.

With the help of a T branch of a logic operation, you can program further "terminating functions" for a logic operation ("multiple output"). Following a T branch however, the selection of programmable elements is restricted. For example, you cannot arrange edge evaluations and call boxes following a T branch. Any further rules applying to the layout of special FBD elements are discussed in the relevant chapters.

When using boxes as program elements, you can

- program a single box per network
- arrange boxes in T branches in branches that start at the left power rail
- arrange boxes in series by switching the ENO output of one box to the EN input of the following box
- AND or OR boxes via the ENO output.

In the case of boxes switched in series, you can control their processing as a group (see also Chapter 15.4 "Using the Binary Result"). You evaluate the error messages of the boxes by combining the ENO outputs: ANDing of the ENO outputs is fulfilled if all boxes have been processed without error, and ORing of the ENO outputs is fulfilled if one of the boxes has been processed without error.

3.4 Programming Data Blocks

Chapter 2.5 "Creating the S7 Program" gives an introduction to program creation and the use of the program editor. Data blocks are programmed in the same way in LAD and FBD.

3.4.1 Creating Data Blocks

You begin block programming by opening a block, either with a double-click on the block in the project window of the SIMATIC Manager or by selecting FILE → OPEN in the editor. If the block does not yet exist, create it as follows:

- In the SIMATIC Manager: select the object *Blocks* in the left-hand portion of the project window and create a new data block with INSERT → S7 BLOCK → DATA BLOCK. You see the properties window of the block. Specify the number and type of the data block on the "General – Part 1" tab (see Chapter 3.4.2 "Types of Data Blocks"). "Instance DB" and "DB of type" can only be selected if function blocks FB, system function blocks SFB or user data types UDT are present in the block container. You can also enter the remaining block properties later.
- In the program editor: with FILE → NEW, you get a dialog box in which you can enter the desired block under "Object name". In the dialog window "New data block" which is subsequently displayed, you are requested to define the type of data block (see Chapter 3.4.2 "Types of Data Blocks"). After closing the dialog box, you can program the block contents.

You can fill out the header of a block as you create it or you can add the block properties at a later point. You program later additions to the block header in the editor by selecting FILE → PROPERTIES while the block is open.

3.4.2 Types of Data Blocks

When creating a new data block, you are requested to define its type. When creating using the SIMATIC Manager, you set the type in the selection box of the properties window; when creating with the program editor, by clicking one of the options offered in the "New data block" window.

A differentiation is made between three types of data block depending on their creation and application:

- "Data block" or "Shared DB"

  Creation as a global data block; you declare the data addresses when programming the data block in this case
- "Data block referencing a user-defined data type" or "DB of type"

  Creation as a data block of user data type; in this case the data structure is used which you have declared when programming the corresponding user data type UDT.
- "Data block referencing a function block" or "Instance DB"

  Creation as an instance data block; here, the data structure that you have declared when programming the relevant function block is transferred.

When creating a data block on the basis of a user data type, you simultaneously define the UDT on which it is based; i.e. the UDT must already have been present in the block container. The same applies to the creation of a data block with assigned function block.

3.4.3 Block Windows and Views

When opening a data block whose structure is based on a user data type or a (system) function block, you will be asked in the standard setting whether you wish to open the data block using the program editor or the application "Parameterization of data blocks". The parameter view presents the data values grouped technologically, and permits more convenient parameterization (see Chapter 2.7.8 "Monitoring and Modifying Data Addresses"). The data views are described below.

The program editor provides two views for programming (creating) data blocks:

- The declaration view is used to define the data structure for global data blocks, as well as the default values.
- You can handle the online values in the data view.

Each view presents a table containing the absolute data addresses in sequence, the names and data types, the initial values and comments (Figure 3.11). The data view contains an additional column with the actual value.

If you open a data block from the offline data management, you are provided with the offline window with which you can edit the data in the programming device. If you open a data block

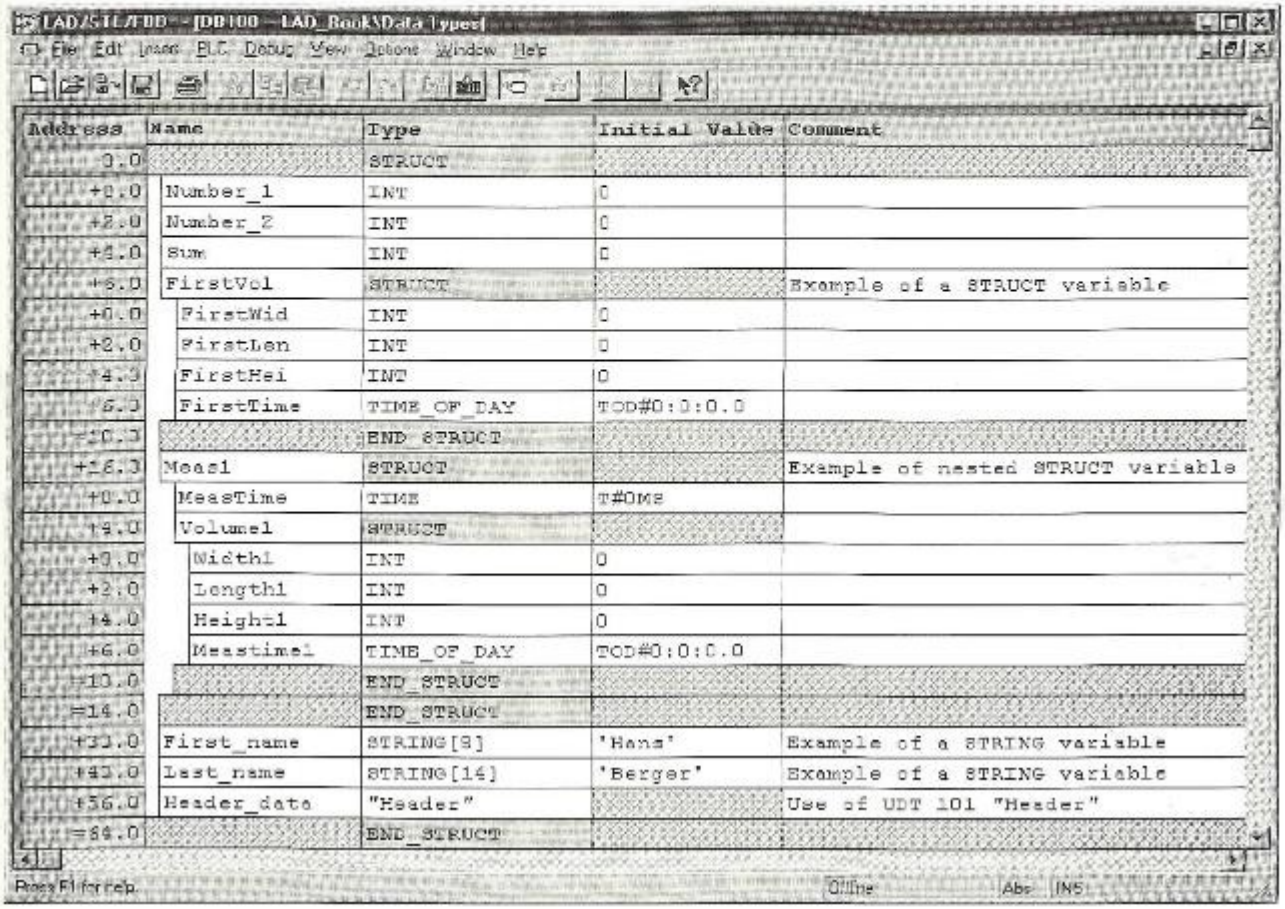


Figure 3.11 Example of an Opened Data Block (Declaration View)

which is present in the CPU's user memory, the editor displays the online window with which you can edit the data values on the CPU.

Offline window

You use the declaration view for inputs with global data blocks. You declare the data addresses in this view: you define the sequence of data addresses, assign a name and data type to each data address, and can additionally enter a comment. Each data address is assigned a default value. This is zero, the smallest value or empty depending on the data type. You can modify the default value in the initial value column.

The data addresses and the default values are already defined for data blocks which are derived from a user data type or from a function block. They are obtained from the declaration of the user data type or from the declaration of the function block.

The data view additionally shows the actual value column. The default values from the initial value column are entered as standard in this column. In the data view, you can enter a different initial value for the load memory and thus an actual value for the work memory (Figure 3.12).

The possibility which exists for assigning individual default values to each data block is particularly important for the data blocks derived from a user data type or from a function block. For example, if you generate several instance data blocks of a function block, all data blocks have the default value set in the function block. In the data view, you can now individually assign other values to various data addresses for each instance.

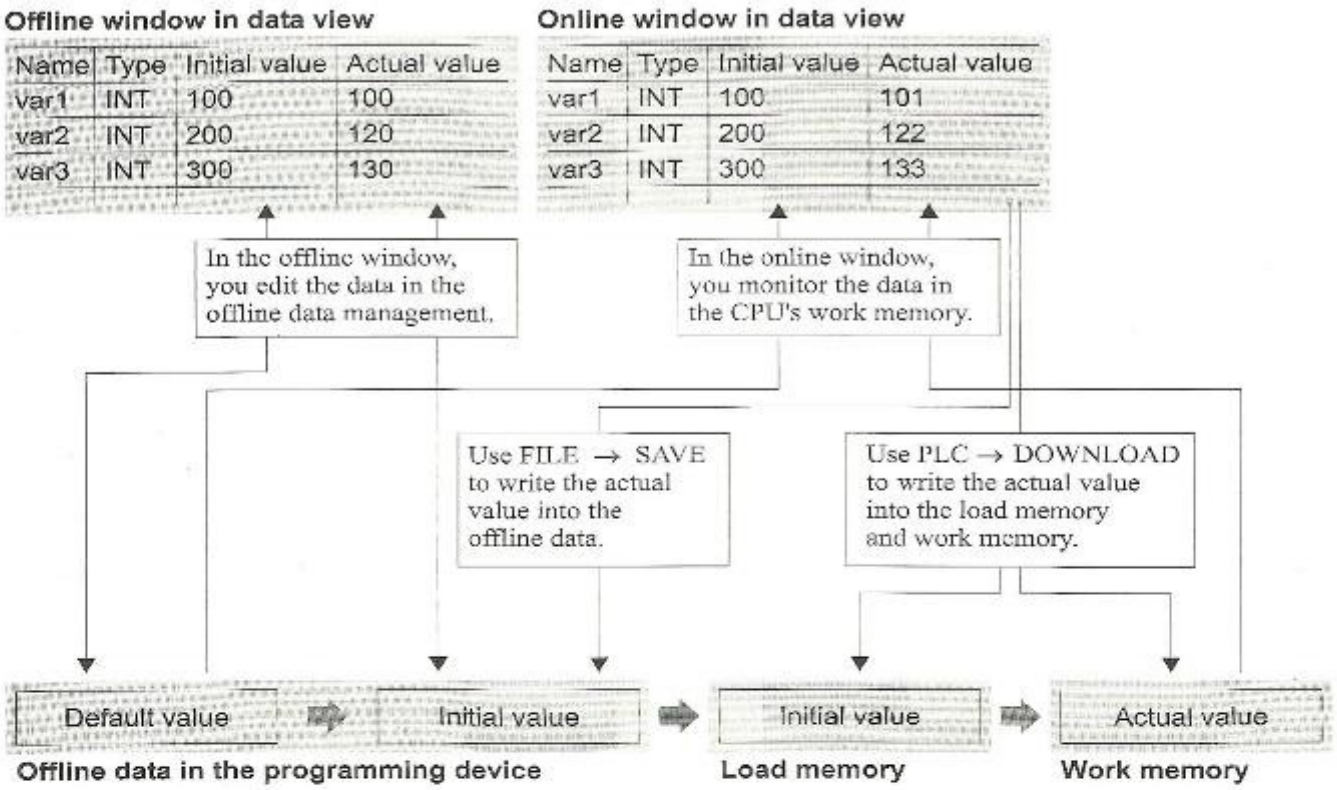


Figure 3.12 Data Storage with Incremental Programming

Online window

You usually use the online window to view the actual data values in the user memory. However, you can also use it to generate data blocks.

The initial value column in the declaration view shows the initial value from the offline data management or the initial value from the load memory if the offline project associated with the CPU program is not available. The actual value column in the data view displays the actual value from the work memory. With EDIT → INITIALIZE DATA BLOCK you can request the editor to replace all actual values by the initial values again.

When writing back with PLC → DOWNLOAD, you write the value in the actual value column into the work memory. You are therefore able to use the programming device to influence the values of data addresses during program execution. The value in the initial value column is rejected.

When writing back with FILE → SAVE, you write the value in the initial value column as the default value, and the value in the actual value column as the initial value into the offline data management.

Note that the complete information concerning data addresses, such as e.g. the name, is only present in the offline data management. It is recommendable to also write the data blocks generated in the CPU's user memory into the offline data management so that data consistency is retained (Chapter 2.6.5 "Block Handling" under "Data blocks offline/online").

3.5 Variables, Constants and Data Types

3.5.1 General Remarks Concerning Variables

A variable is a value with a specific format (Figure 3.13). Simple variables consist of an address (such as input 5.2) and a data type (such as BOOL for a binary value). The address, in

turn, comprises an address identifier (such as I for input) and an absolute storage location (such as 5.2 for byte 5, bit 2). You can also reference an address or a variable symbolically by assigning the address a name (a symbol) in the symbol table.

A bit of data type BOOL is referred to as a *binary address (*or *binary operand)*. Addresses comprising one, two or four bytes or variables with the relevant data types are called *digital operands*.

Variables, which you declare within a block, are referred to as (block-) local variables. These include the block parameters, the static and temporary local data, even the data addresses in global data blocks. When these variables are of an elementary data type, they can also be accessed as operands (for instance static local data as DI operands, temporary local data as L operands, and data in global data blocks as DB operands).

Local variables, however, can also be of complex data type (such as structures or arrays). Variables with these data types require more than 32 bits, so that they can no longer, for example, be loaded into the accumulator. And for the same reason, they cannot be addressed with "normal" STL statements. There are special functions for handling these variables, such as the IEC functions, which are provided as a standard library with STEP 7 (you can generate variables of complex data type in block parameters of the same data type).

If variables of complex data type contain components of elementary data type, these components can be treated as though they were separate variables (for example, you can load a component of an array consisting of 30 INT values into the accumulator and further process it).

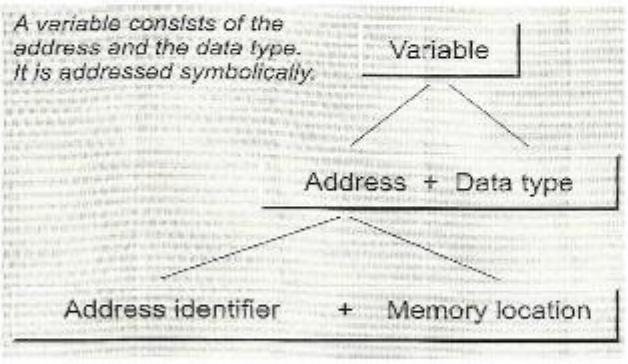


Figure 3.13 Structure of a Variable

Constants are used to preset variables to a fixed value. The constant is given a specific prefix depending on the data type.

3.5.2 Addressing Variables

When addressing variables, you may choose between absolute addressing and symbolic addressing.

- Absolute addressing uses numerical addresses beginning with zero for each address area.
- Symbolic addressing uses alphanumeric names, which you yourself define in the symbol table for global addresses or in the declaration section for block-local addresses.

Absolute addressing of variables

Variables of elementary data type can be referenced by absolute addresses.

The absolute address of an input or output is computed from the module start address, which you set or had set in the configuration table and the type of signal connection on the module. A distinction is made between binary signals and analog signals.

Binary signals

A binary signal contains one bit of information. Examples of binary signals are the input signals from limit switches, momentary-contact switches and the like which lead to digital input modules, and output signals which control lamps, contactors, and the like via digital output modules.

Analog signals

An analog signal contains 16 bits of information. An analog signal corresponds to a "channel", which is mapped in the controller as a word (2 bytes) (see below). Analog input signals (such as voltages from resistance thermometers) are carried to analog input modules, digitized, and made available to the controller as 16 information bits. Conversely, 16 bits of

information can control an indicator via an analog output module, where the information is converted into an analog value (such as a current).

The information width of a signal also corresponds to the information width of the variable in which the signal is stored and processed. The information width and the interpretation of the information (for instance the positional weight), taken together, produce the data type of the variable. Binary signals are stored in variables of data type BOOL, analog signals in variables of data type INT.

The only determining factor for the addressing of variables is the information width. In STEP 7, there are four widths, which can be accessed with absolute addressing:

- 1 bit — Data type BOOL
- 8 bits — Data type BYTE or another data type with 8 bits
- 16 bits — Data type WORD or another data type with 16 bits
- 32 bits — Data type DWORD or another data type with 32 bits

Variables of data type BOOL are referenced via an address identifier, a byte number, and – separated by a decimal point – a bit number. Numbering of the bytes begins at zero for each address area. The upper limit is CPU-specific. The bits are numbered from 0 to 7.
Examples:

I 1.0 Input bit no. 0 in byte no. 1

Q 16.4 Output bit no. 4 in byte no. 16

Variables of data type BYTE have as absolute address the address identifier and the number of the byte containing the variable. The address identifier is supplemented by a B. Examples:

IB 2 Input byte no. 2

QB 18 Output byte no. 18

Variables of data type WORD consist of two bytes (a word). They have as absolute address the address identifier and the number of the low-order byte of the word containing the variable. The address identifier is supplemented by a W. Examples:

IW 4 Input word no. 4;
contains bytes 4 and 5

QW 20 Output word no. 20;
contains bytes 20 and 21

Variables of data type DWORD consist of four bytes (a doubleword). They have as absolute address the address identifier and the number of the low-order byte of the word containing the variable. The address identifier is supplemented by a D. Examples:

ID 8 Input doubleword no. 8;
contains bytes 8, 9, 10 and 11

QD 24 Output doubleword no. 24;
contains bytes no. 24, 25, 26 and 27

Addresses for the data area include the data block. Examples:

DB 10.DBX 2.0
Data bit 2.0 in data block DB 10

DB 11.DBB 14
Data byte 14 in data block DB 11

DB 20.DBW 20
Data word 20 in data block DB 20

DB 22.DBD 10
Data doubleword 10 in data block DB 22

Additional information on addressing the data area can be found in Chapter 18.2.2 "Accessing Data Operands".

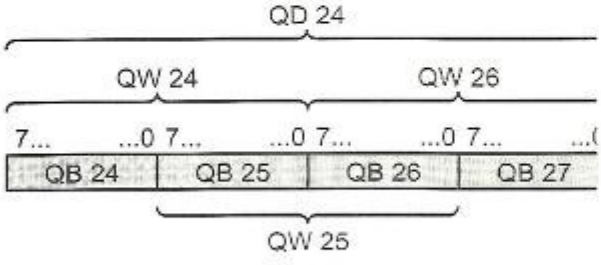


Figure 3.14
Byte Contents in Words and Doublewords

Symbolic addressing of variables

Symbolic addressing uses a name (called a symbol) in place of an absolute address. You yourself choose this name. Such a name must begin with a letter and may comprise up to 24

characters. A keyword is not permissible as a symbol.

There is no difference between upper-case and lower-case letters when entering a symbol. During the output, the editor applies the notation defined during declaration of the symbol.

The name, or symbol, must be allocated to an absolute address. A distinction is made between global symbols and symbols that are local to a block.

Global symbols

You may assign names in the symbol table to the following objects:

- Data blocks and code blocks
- Inputs, outputs, peripheral inputs and peripheral outputs
- Memory bits, timers and counters
- User data types
- Variable tables

A global symbol may also include spaces, special characters and country-specific characters such as the umlaut. Exceptions to this rule are the characters 00_{hex} and FF_{hex} and the quotation mark ("). When using symbols containing special characters, you must put the symbols in quotation marks in the program. In compiled blocks, the STL Editor always shows global symbols in quotation marks.

You can use global symbols throughout the program; each such symbol must be unique within a program.

Editing, importing and exporting of global symbols are described in Chapter 2.5.2 "Symbol Table".

Block-local symbols

The names for the local data are specified in the declaration section of the relevant block. These names may contain only letters, digits and the underline character (no umlauts!).

Local symbols are valid only within a block. The same symbol (the same variable name) may be used in a different context in another block. The Editor shows local symbols with a leading "#". When the Editor cannot distinguish a local symbol from an address, you must precede the symbol with a "#" character during input.

Local symbols are available only in the programming device database (in the offline container *Blocks)*. If this information is missing on decompilation, the Editor inserts a substitute symbol.

Using symbol names

If you use symbolic names while programming with the incremental Editor, they must have already been allocated to absolute addresses. You also have the option of entering new symbolic names in the symbol table during program input. You can subsequently continue program input with the new symbolic name.

If you compile a source text file generated e.g. from LAD/FBD blocks, the complete assignment of symbolic names to absolute addresses is only made available during the compilation.

In the case of arrays, the individual components are accessed via the array name and a subscript, for example MSERIES[1] for the first component. In LAD and FBD, the index is a constant INT value.

In structures, each subidentifier is separated from the preceding subidentifier by a decimal point, for instance FRAME.HEADER.CNUM. Components of user data types are addressed exactly like structures.

Data addresses

Symbolic addressing of data uses complete addressing including the data block. Example: the data block with the symbolic address MVALUES contains the variables MVALUE1, MVALUE2 and MTIME. These variables can be addressed as follows:

```
"MVALUES".MVALUE_1
"MVALUES".MVALUE_2
"MVALUES".MTIME
```

Please refer to Chapter 18.2.2 "Accessing Data Operands" for further information on addressing of data.

Table 3.4 Division of the Data Types

Elementary Data Types	Complex Data Types	User Data Types	Parameter Data Types
BOOL, BYTE, CHAR, WORD, INT, DATE, DWORD, DINT, REAL, S5TIME, TIME, TOD	DT, STRING, ARRAY, STRUCT	UDT, Global data blocks, Instances	TIMER, COUNTER, BLOCK_DB, BLOCK_SDB, BLOCK_FC, BLOCK_FB, POINTER, ANY
Data types comprising no more than one doubleword (32 bits)	Data types that can comprise more than one doubleword (DT, STRING) or which consist of several components	Structures or data areas which can be assigned a name	Block parameters
Can be mapped to operands referenced with absolute and symbolic addressing	Can be mapped only to variables that are addressed symbolically		Can be mapped only to block parameters (symbolic addressing only)
Permitted in all address areas	Permitted in data blocks (as global data and instance data), as temporary local data and as block parameters		Permitted in conjunction with block parameters

3.5.3 Overview of Data Types

Data types stipulate the characteristics of data, essentially the representation of the contents of a variable, and the permissible ranges. STEP 7 provides predefined data types, which you can *combine into user data types.*

The data types are available on a global basis, and can be used in every block. LAD and FBD use the same data types.

Depending on structure and application, the data types with STEP 7 are classified as follows:

- Elementary data types
- Complex data types
- User data types
- Parameter types

Table 3.4 shows the properties of these data type classes.

You can find examples of the declaration and use of variables of all data types in the libraries "LAD_Book" and "FBD_Book" under the program "Data Types" program that you can download from the publisher's Website (see *page 8*).

3.5.4 Elementary Data Types

Elementary data types can reserve a bit, a byte, a word or a doubleword.

Table 3.6 shows the elementary data types. For many data types, there are two constant representations that you can use equally (e.g. TIME# or T#). The table contains the minimum value for a data type in the upper line and the maximum value in the lower line.

Declaration of elementary data types

Table 3.5 shows some examples of the declaration of variables of elementary data types.

Table 3.5 Examples of Declaration and Initial Value for Elementary Data Types

Name	Type:	Initial Value	Comments
Automatic	BOOL	FALSE	Initial value is signal state "0"
Manual_off	BOOL	TRUE	Initial value is signal state "1"
Measured_value	DINT	L#0	Initial value of a DINT variable
Memory	WORD	W#16#FFFF	Initial value of a WORD variable
Waiting_time	S5TIME	S5T#20s	Initial value of an S5 time variable

Table 3.6 Overview of Elementary Data Types

Data Type	(Width)	Description	Example for Constant Notation
BOOL	(1 bit)	Bit	FALSE TRUE
BYTE	(8 bits)	8-bit hexadecimal number	B#16#00, 16#00 B#16#FF, 16#FF
CHAR	(8 bits)	One character (ASCII)	Printable character, e.g. 'A'
WORD	(16 bits)	16-bit hexadecimal number	W#16#0000, 16#0000 W#16#FFFF, 16#FFFF
		16-bit binary number	2#0000_0000_0000_0000 2#1111_1111_1111_1111
		Count value, 3 decades BCD	C#000 C#999
		Two 8-bit unsigned decimal numbers	B#(0,0) B#(255,255)
DWORD	(32 bits)	32-bit hexadecimal number	DW#16#0000_0000, 16#0000_0000 DW#16#FFFF_FFFF, 16#FFFF_FFFF
		32-bit binary number	2#0000_0000_..._0000_0000 2#1111_1111_..._1111_1111
		Four 8-bit unsigned decimal numbers	B#(0,0,0,0) B#(255,255,255,255)
INT	(16 bits)	Fixed-point number	–32 768 +32 767
DINT	(32 bits)	Fixed-point number	L#–2 147 483 648 [1] L#+2 147 483 647 [1]
REAL	(32 bits)	Floating-point number	+1.234567E+02 [2] in exponential representation
			123.4567 [2] as decimal number
S5TIME	(16 bits)	Time value in SIMATIC format	S5T#0ms S5TIME#2h46m30s
TIME	(32 bits)	Time value in IEC format	T#–24d20h31m23s647ms TIME#24d20h31m23s647ms
			T#–24.855134d TIME#24.855134d
DATE	(16 bits)	Date	D#1990-01-01 DATE#2168-12-31
TIME_OF_DAY	(32 bits)	Time of day	TOD#00:00:00 TIME_OF_DAY#23:59:59.999

[1] "L#" may be omitted if the number is outside the INT number range

[2] for value range see text

Name is the identifier for a block-local variable (up to 24 characters, alphanumeric and underscore only). You enter the associated data type in the *Type* column.

With the exception of the temporary local data and block parameters of functions, you can assign an *initial value* to the variables. Use the syntax suitable for the data type for this purpose. *Comments* are optional.

BOOL, BYTE, WORD, DWORD, CHAR

A variable of data type BOOL represents a bit value (for example input I 1.0). Variables with data types BYTE, WORD and DWORD are bit strings comprising 8, 16 and 32 bits, respectively. The individual bits are not evaluated.

Special forms of these data types are the BCD numbers and the count as used in conjunction with counter functions, as well as data type CHAR, which represents an ASCII character.

BCD numbers

BCD numbers have no special identifier. Simply enter a BCD number with the data type 16# (hexadecimal) and use only digits 0 to 9.

BCD numbers occur in coded processing of time values and counts and in conjunction with conversion functions. Data type S5TIME# is used to specify a time value for starting a timer (see below), data type 16# or C# for specifying a count value. A C# count value is a BCD number between 000 and 999, whereby the sign is always 0.

As a rule, BCD numbers have no sign. In conjunction with the conversion functions, the sign of a BCD number is stored in the leftmost (highest) decade, so that there is one less decade for the number.

When a BCD number is in a 16-bit word, the sign is in the uppermost decade, whereby only bit position 15 is relevant. Signal state "0" means that the number is positive. Signal state "1" stands for a negative number. The sign has no affect on the contents of the individual decades. An equivalent assignment applies for a 32-bit word.

The available value range is 0 to ± 999 for a 16-bit BCD number and 0 to ± 9 999 999 for a 32-bit number.

CHAR

A variable with data type CHAR (character) reserves one byte. Data type CHAR represents a single character in ASCII format. Example: "A".

You can use any printable character in apostrophes. Some special characters require use of the notation shown in Table 3.7. Example: '$$' represents a dollar sign in ASCII code.

Table 3.7 Special Characters for CHAR

CHAR	Hex	Description
$$	24_{hex}	Dollar sign
$'	27_{hex}	Apostrophe
$L or $l	$0A_{hex}$	Line feed (LF)
$P or $p	$0C_{hex}$	New page (FF)
$R or $r	$0D_{hex}$	Carriage return (CR)
$T or $t	09_{hex}	Tabulator

The MOVE function allows you to use two or four ASCII characters enclosed in apostrophes as a special form of data type CHAR for writing ASCII characters in a variable.

INT

A variable with data type INT is stored as an integer (16-bit fixed-point number). Data type INT has no special identifier.

A variable with data type INT reserves one word. The signal states of bits 0 to 14 represent the digit positions of the number; the signal state of bit 15 represents the sign (S). Signal state "0" means that the number is positive, signal state "1" that it is negative. A negative number is represented as two's complement. The permissible number range is

from +32 767 ($7FFF_{hex}$)

to –32 768 (8000_{hex}).

DINT

A variable with data type DINT is stored as an integer (32-bit fixed-point number). An integer is stored as a DINT variable when it exceeds 32 767 or falls below –32 768 or when the number is preceded by type identifier L#.

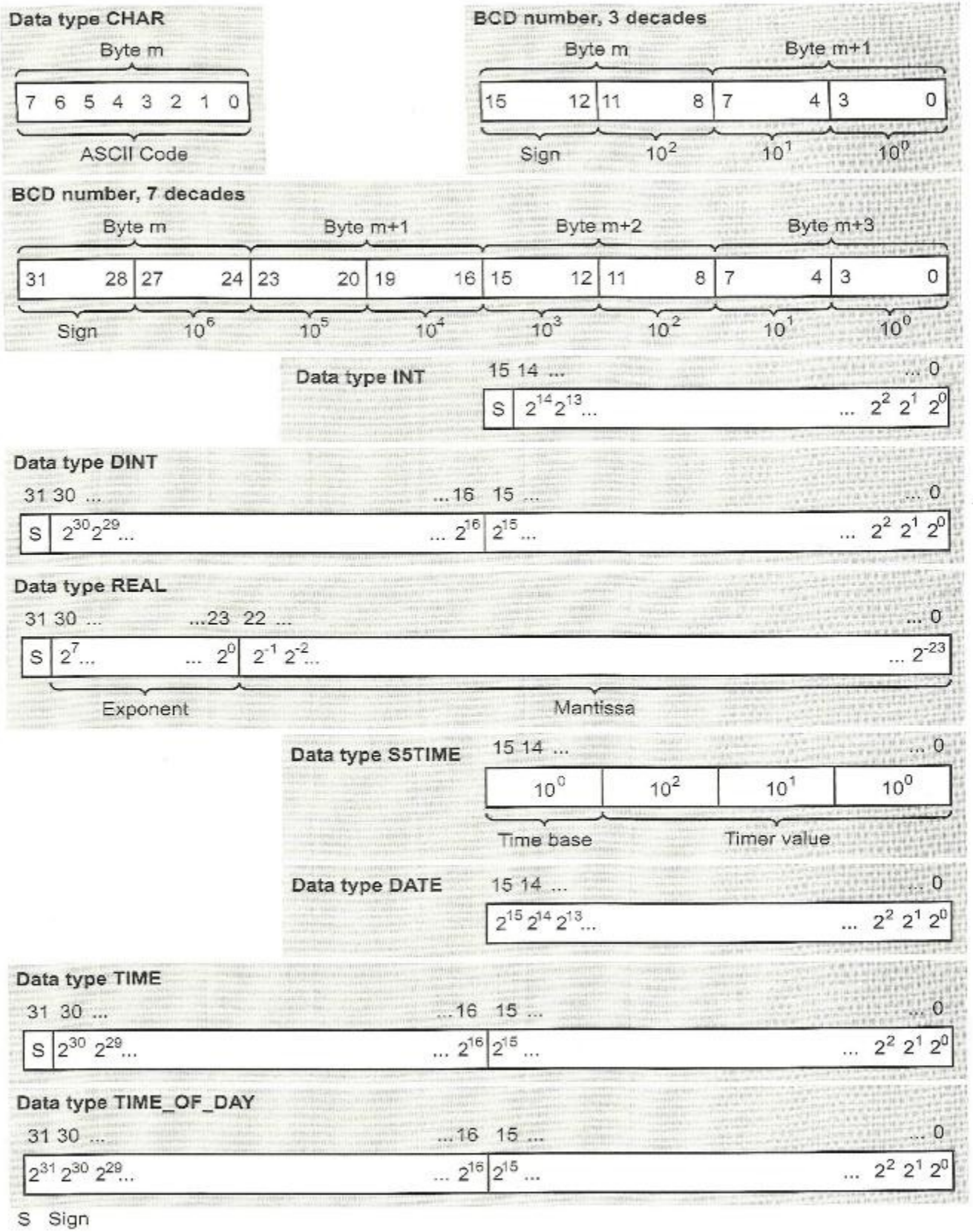


Figure 3.15 Structure of the Variables of Elementary Data Type

A variable with data type DINT reserves one doubleword. The signal states of bits 0 to 30 represent the digit positions of the number; the sign is stored in bit 31. Bit 31 is "0" for a positive and "1" for a negative number. Negative numbers are stored as two's complement. The number range is

from +2 147 483 647 ($7FFF\ FFFF_{hex}$)

to –2 147 483 648 ($8000\ 0000_{hex}$).

REAL

A variable of data type REAL represents a fraction, and is stored as a 32-bit floating-point number. An integer is stored as a REAL variable when you add a decimal point and a zero.

In exponent representation, you can precede the "e" or "E" with an integer number or fraction with seven relevant digits and a sign. The digits that follow the "e" or "E" represent the exponent to base 10. STEP 7 handles the conversion of the REAL variable into the internal representation of a floating-point number.

REAL variables are divided into numbers, which can be represented with complete accuracy ("normalized" floating-point numbers) and those with limited accuracy ("denormalized" floating-point numbers).

The value range of a normalized floating-point number lies between:

$-3.402\ 823 \times 10^{+38}$ 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 may be in the following range:

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

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

A variable of data type REAL consists internally of three components, namely the sign, the 8-bit exponent to base 2 and the 23-bit mantissa. The sign may assume the value "0" (positive) or "1" (negative). Before the exponent is stored, a constant value (bias, +127) is added to it so that it shows a value range of from 0 to 255. The mantissa represents the fractional portion of the number. The integer portion of the mantissa is not saved, as it is either always 1 (in the case of normalized floating-point numbers) or always 0 (in the case of denormalized floating-point numbers). Table 3.8 shows the internal range of a floating-point number.

S5TIME

A variable with data type S5TIME is used in the basic languages STL, LAD and FBD to set the SIMATIC timers. It reserves one 16-bit word with 1 + 3 decades.

The time is specified in hours, minutes, seconds and milliseconds. STEP 7 handles conversion into internal representation. Internal represen-

Table 3.8 Range Limits of a Floating-Point Number

Sign	Exponent	Mantissa	Description
0	255	Not equal to 0	Not a valid floating-point number
0	255	0	+ infinite
0	1 ... 254	Arbitrary	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	Arbitrary	Negative normalized floating-point number
1	255	0	– infinite
1	255	Not equal to 0	Not a valid floating-point number

tation is as BCD number in the range 000 to 999. The time interval can assume the following values: 10 ms (0000), 100 ms (0001), 1 s (0010), and 10 s (0011). The duration is the product of time interval and time value.
Examples:

S5TIME#500ms (= 0050_{hex})
S5T#2h46m30s (= 3999_{hex})

DATE

A variable with data type DATE is stored in a word as an unsigned fixed-point number. The contents of the variable correspond to the number of days since 01.01.1990. Its representation shows the year, month and day, separated from one another by a hyphen. Examples:

DATE#1990-01-01 (= 0000_{hex})
D#2168-12-31 (= $FF62_{hex}$)

TIME

A variable with data type TIME reserves one doubleword. Its representation contains the information for days (d), hours (h), minutes (m), seconds (s) and milliseconds (ms), whereby individual items of this information may be omitted. The contents of the variable are interpreted in milliseconds (ms) and stored as a signed 32-bit fixed-point number. Examples:

TIME#24d20h31m23s647ms
(= $7FFF\_FFFF_{hex}$)
TIME#0ms (= 0000_0000_{hex})
T#–24d20h31m23s648ms
(= 8000_0000_{hex})

A "decimal representation" is also possible for TIME, e.g. TIME#2.25h or T#2.25h. Examples:

TIME#0.0h (= 0000_0000_{hex})
TIME#24.855134d (= $7FFF\_FFFF_{hex}$)

TIME_OF_DAY

A variable of data type TIME_OF_DAY reserves one doubleword. It contains the number of milliseconds since the day began (0:00 o'clock) in the form of an unsigned fixed-point number. Its representation contains the information for hours, minutes and seconds, separated by a colon. The milliseconds, which follow the seconds and are separated from them by a decimal point, may be omitted.
Examples:

TIME_OF_DAY#00:00:00 (= 0000_0000_{hex})
TOD#23:59:59.999 (= 0526_5BFF_{hex})

3.5.5 Complex Data Types

STEP 7 defines the following four complex data types:

- DATE_AND_TIME
  Date and time (BCD-coded)
- STRING
  Character string with up to 254 characters
- ARRAY
  Array variable (combination of variables of the same type)
- STRUCT
  Structure variable (combination of variables of different types)

The data types are pre-defined, with the length of the data type STRING (character string) and the combination and size of the data types ARRAY and STRUCT (structure) being defined by the user.

You can declare variables of complex data types only in global data blocks, in instance data blocks, as temporary local data or as block parameters.

Variables of complex data types can only be applied at block parameters as complete variables.

There are IEC functions for processing variables of data types DT and STRING, e.g. extraction of the date and conversion to the DATE representation or combining two character strings to one variable. These IEC functions are loadable standard FC blocks that you can find in the *Standard Library* under the *IEC Function Blocks* program.

DATE_AND_TIME

The data type DATE_AND_TIME represents a time consisting of the date and the time of day. You can also use the abbreviation DT in place of DATE_AND_TIME.

The individual components of a DT variable are ASCII coded (Figure 3.16).

Table 3.9 Examples of the Declaration of DT Variables and STRING Variables

Name	Type:	Initial Value	Comments
Date1	DT	DT#1990-01-01-00:00:00	DT variable minimum value
Date2	DATE_AND_TIME	DATE_AND_TIME# 2089-12-31-23:59:59.999	DT variable maximum value
First_name	STRING[10]	'Jack'	STRING variable, 4 out of 10 char. specified
Last_name	STRING[14]	'Daniels'	STRING variable, all 7 char. specified
NewLine	STRING[2]	'RL'	STRING variable, special char. specified
BlankString	STRING[16]	''	STRING variable, no specification

STRING

The data type STRING represents a character string consisting of up to 254 characters. You specify the maximum permissible number of characters in square brackets following the keyword STRING.

This specification can also be omitted; the Editor then uses a length of 254 bytes. In the case of functions FCs, the Editor does not permit specification of the length or it demands the standard length of 254. A variable of data type STRING occupies two bytes more of memory than the declared maximum length.

Pre-assignment is carried out with ASCII-coded characters between single inverted commas or with a prefixed dollar sign in the case of certain characters (see data type CHAR).

If the initial or pre-assigned value is shorter than the declared maximum length, the remaining character locations are not reserved. When a variable of data type STRING is post-processed, only the currently reserved character locations are taken into consideration. It is also possible to define an "empty string" as the initial value. Figure 3.16 shows the structure of a STRING variable.

ARRAY

Data type ARRAY represents an array or field comprising a fixed number of elements of the same data type.

You specify the range of field indices in square brackets following the data type ARRAY. The initial value on the left must be less than or

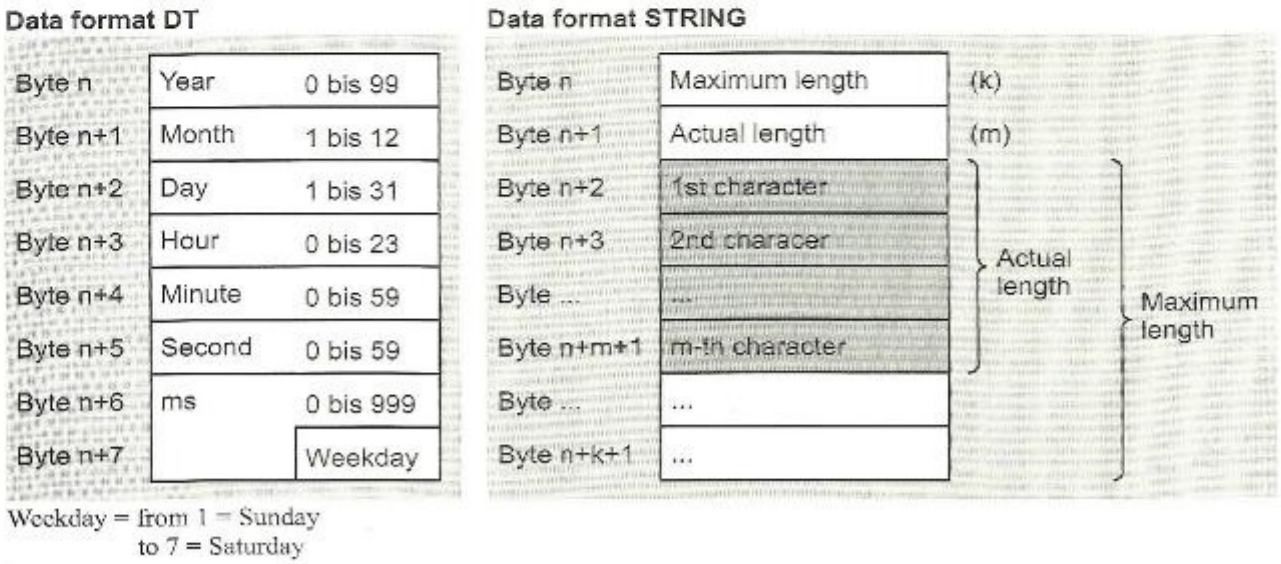


Figure 3.16 Structure of a DT and a STRING Variable

Table 3.10 Examples of an Array Declaration

Name	Type	Initial Value	Comments
Meas. val.	ARRAY[1..24]	0.4, 1.5, 11 (2.6, 3.0)	Array variable with 24 REAL elements
	REAL		
TOD	ARRAY[-10..10]	21 (TOD#08:30:00)	TOD array with 21 elements
	TIME_OF_DAY		
Result	ARRAY[1..24,1..4]	96 (L#0)	Two-dimensional array with 96 elements
	DINT		
Char.	ARRAY[1..2,3..4]	2 ("a"), 2 ("b")	Two-dimensional array with 4 elements
	CHAR		

equal to the final value on the right. Both indices are INT numbers in the range –32,768 to +32,767. A field can have up to 6 dimensions each of whose limits are separated by a comma.

The data type of the individual field components is located in the line under the data type ARRAY. All data types except ARRAY are permissible; it can also be a user data type.

Pre-assignment

At the declaration stage, you can pre-assign values to individual field components (not as a block parameter in a function, as an in/out parameter in a function block or as a temporary variable). The data type of the pre-assignment value must match the data type of the field.

You do not require to pre-assign all field components; if the number of pre-assignment values is less than the number of field components, only the first components are pre-assigned. The number of pre-assignment values must not be greater than the number of field components. The pre-assignment values are each separated by a comma. Multiple pre-assignment with the same values is specified within round brackets with a preceding repetition factor.

Application

You can apply a field as a complete variable at block parameters of data type ARRAY with the same structure or at a block parameter of data type ANY. For example, you can copy the contents of a field variable using the system function SFC 20 BLKMOV. You can also specify individual field components at a block parameter if the block parameter is of the same data type as the components.

If the individual field components are of elementary data types, you can process them with "normal" LAD or FBD functions.

A field component is accessed with the field name and an index in square brackets. The index is a fixed value in LAD and FBD and cannot be modified at runtime (no variable indexing possible).

Multi-dimensional fields or arrays

Fields can have up to 6 dimensions. Multi-dimensional fields are analogous to one-dimensional fields. At the declaration stage, the ranges of the dimensions are written in square brackets, each separated by a comma.

Structure of the variables

An ARRAY variable always begins at a word boundary, that is, at a byte with an even address. ARRAY variables occupy the memory up to the next word boundary.

Components of data type BOOL begin in the least significant bit; components of data type BYTE and CHAR begin in the right-hand byte. The individual components are listed in order.

In multi-dimensional fields, the components are stored line-wise (dimension-wise) starting with the first dimension. With bit and byte components, a new dimension always starts in the next byte, and with components of other data types a new dimension always starts in the next word (in the next even byte).

Table 3.11 Example of Declaring a Structure

Name	Type	Initial Value	Comment
MotCont	STRUCT		Simple structure variable with 4 components
On	BOOL	FALSE	Variable MotCont.On of type BOOL
Off	BOOL	TRUE	Variable MotCont.Off of type BOOL
Delay	S5TIME	S5TIME#5s	Variable MotCont.Delay of type S5TIME
maxSpeed	INT	5000	Variable MotCont.maxSpeed of type INT
	END_STRUCT		

STRUCT

The data type STRUCT represents a data structure consisting of a fixed number of components that can each be of a different data type.

You specify the individual structure components and their data types under the line with the variable name and the keyword STRUCT. All data types can be used including other structures.

Pre-assignment

At the declaration stage, you can pre-assign values to the individual structure components (not as a block parameter in a function, as an in/out parameter in a function block or as a temporary variable). The data types of the pre-assignment values must match the data types of the components.

Application

You can apply a complete variable at block parameters of data type STRUCT with the same structure or at a block parameter of data type ANY. For example, you can copy the contents of a STRUCT variable with the system function SFC 20 BLKMOV. You can also specify an individual structure component at a block parameter if the block parameter is of the same data type as the component.

If the individual structure components are of elementary data types, you can process them with "normal" LAD or FBD functions.

A structure component is accessed with the structure name and the component name separated by a dot.

Structure of the variables

A STRUCT variable always begins at a word boundary, that is, at a byte with an even address; following this, the individual components are located in the memory in the order of their declaration. STRUCT variables occupy the memory up to the next word boundary.

Components of data type BOOL begin in the least significant bit; components of data type BYTE and CHAR begin in the right-hand byte. Components of other data types begin at a word boundary.

A nested structure is a structure as a component of another structure. A nesting depth of up to 6 structures is possible. All components can be accessed individually with "normal" LAD or FBD functions provided they are of elementary data type. The individual names are each separated by a dot.

3.5.6 Parameter Types

Parameter types are data types for block parameters (Table 3.12). The length specifications in the Table refer to the memory requirements for block parameters for function blocks. You can also use TIMER and COUNTER in the symbol table as data types for timers and counters.

3.5.7 User Data Types

A user data type (UDT) corresponds to a structure (combination of components of any data type) with global validity. You can use a user data type if a data structure occurs frequently in your program or you want to assign a name to a data structure.

UDTs have global validity; i.e., once declared, they can be used in all blocks. UDTs can be

Table 3.12 Overview of Parameter Types

Parameter Type	Description		Examples of Actual Addresses
TIMER	Timer	16 bits	T 15 or symbol
COUNTER	Counter	16 bits	Z 16 or symbol
BLOCK_FC	Function	16 bits	FC 17 or symbol
BLOCK_FB	Function block	16 bits	FB 18 or symbol
BLOCK_DB	Data block	16 bits	DB 19 or symbol
BLOCK_SDB	System data block	16 bits	(not used yet)
POINTER	DB pointer	48 bits	As pointer: P#M10.0 or P#DB20.DBX22.2 As address: MW 20 or I 1.0 or symbol
ANY	ANY pointer	80 bits	As range: P#DB10.DBX0.0 WORD 20 or any (complete) variable

addressed symbolically; you assign the absolute address in the symbol table. The data type of a UDT (in the symbol table) is identical with the absolute address.

If you want to give a variable the data structure defined in the UDT, assign the UDT to it at declaration like a "normal" data type. The UDT can be absolutely addressed (UDT 0 to UDT 65,535) or symbolically addressed.

You can also define a UDT for an entire data type. When programming the data block, you assign this UDT to the block as a data structure.

The example "Message Frame Data" in Chapter 24.3 "Brief Description of the "Message Frame Example"" shows you how to work with user data types.

Creating UDTs

You can create a user data type using the SIMATIC Manager or also the program editor:

- In the SIMATIC Manager: select the *Blocks* object in the left part of the project window, and create a new UDT using INSERT → S7 BLOCK → DATA TYPE. You are then provided with the attributes window of the data type. On the "General – Part 1" tab, enter the number (the absolute address UDTn) under "Name". You can also enter the other block attributes later.
- In the program editor: use FILE → NEW to obtain a dialog box in which you can enter the desired data type (the absolute address UDTn) under "Object name".

You can fill in the block header immediately when creating the data type, or enter the attributes later. With the data type open, you can program subsequent extensions in the program editor using FILE → PROPERTIES.

A double-click on data type *UDTn* in the SIMATIC Manager opens a declaration table that looks exactly like the declaration table of a data block. A UDT is programmed in exactly the same way as a data block, with individual lines for Name, Type, Initial value and Comments. The only difference is that switching to the data view is not possible. (With a UDT, you do not create any variables but only a collection of data types; for this reason, there can be no actual values here).

The initial values you program in the UDT are transferred to the variables at declaration.

Basic Functions

This section of the book describes those functions of the LAD and FBD programming languages which represent a certain "basic functionality". These functions allow you to program a PLC on the basis of contactor or relay controls.

In a ladder diagram (LAD), the arrangement of the contacts in **series and parallel circuits** determines the combining of binary signal states. In a function block diagram (FBD), boxes analogous to electronic switching systems represent the **boolean functions** AND and OR.

The **memory functions** hold onto an RLO so that it can, for example, be scanned and processed further in another part of the program.

The **move functions** are used to exchange the values of individual operands and variables or to copy entire data areas.

The timing relays in contactor control systems are **timers** in programmable controllers. The timers integrated in the CPU allow you, for example, to program waiting and monitoring times.

Finally, the **counters** can count up and down in the range 0 to 999.

This section of the book describes the functions for the operand areas for inputs, outputs, and memory bits. Inputs and outputs are the link to the process or plant. The memory bits correspond to auxiliary contactors which store binary states. The subsequent sections of the book describe the remaining operand areas, on which you can also use binary logic. Essentially, these are the data bits in global data blocks as well as the temporary and static local data bits.

In Chapter 5 "Memory Functions", you will find a programming example for the binary logic operations and memory functions, and in Chapter 8 "Counters", an example for timers and counters. In both cases, the example is in an FC function without block parameters. You will find the same examples as function blocks (FBs) with block parameters in Chapter 19 "Block Parameters".

4 **Binary Logic Operations**
Series and parallel circuits (LAD), AND, OR and exclusive OR functions (FBD): negation; taking account of the sensor type

5 **Memory functions**
LAD coils; FBD boxes; midline outputs; edge evaluation; conveyor belt example

6 **Move functions**
MOVE box, system functions for moving data

7 **Timers**
Starting 5 different kinds of timer, resetting and scanning a timer; IEC timers

8 **Counters**
Setting a counter; up and down counting; resetting and scanning a counter; IEC counters; feed example

4 Binary Logic Operations

4.1 Series and Parallel Circuits (LAD)

Binary signal states are combined in LAD through series and parallel connection of contacts. Series connection corresponds to an AND function and parallel connection to an OR function. You use the contacts to check the signal states of the following binary operands:

- Input and output bits, memory bits
- Timers and counters
- Global data bits
- Temporary local data bits
- Static local data bits
- Status bits (evaluation of calculation results)

You can reference an operand via a contact using either an absolute or a symbolic address. LAD uses only NO contacts (scan for signal state "1") and NC contacts (scan for signal state "0").

A rung may consist of a single contact, but it may also consist of a large number of contacts connected together. A rung must always be terminated, for example with a coil. The coil controls a binary operand with the RLO (the "power flow") of the rung.

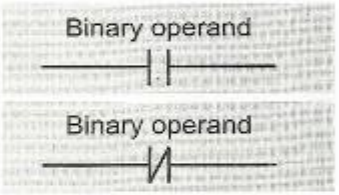
The examples shown in this chapter can be found in function block FB 104 of the "Basic Functions" program in the "LAD_Book" library that you can download from the publisher's Website (see page 8).

For incremental programming, you will find the elements for binary logic operations in the Program Element Catalog (with VIEW → OVERVIEWS [Ctrl -K] or with INSERT → PROGRAM ELEMENTS) under "Bit Logic".

4.1.1 NO Contact and NC Contact

In order to explain the bit logic combinations in a ladder diagram, we will refer below as graphically as possible to "contact closed", "power flowing", and "coil energized". If "power" is flowing at a point in the ladder diagram, this means that the bit logic combination applies up to this point; the result of the logic operation (RLO) is "1". If "power" is flowing in a single coil, the coil is energized; the associated binary operand then carries signal state "1".

LAD has two kinds of contacts for scanning bit operands: the NO contact and the NC contact.



Normally open (NO) contact

A normally open contact corresponds to a scan for signal state "1". If the scanned binary operand has signal state "1", the NO contact is activated, so it closes and "power flows".

The example in Figure 4.1 (left side) shows sensor S1 connected to input I 1.0 and scanned by an NO contact. If sensor S1 is open, input I 1.0 is "0" and no power flows through the NO contact. Contactor K1, controlled by output Q 4.0, does not switch on.

If sensor S1 is now activated, input I 1.0 has signal state "1". Power flows from the left power rail through the NO contact into the coil, and contactor K1, which is connected to output Q 4.0, is activated.

The NO contact scans the input for signal state "1" and then closes, regardless of whether the sensor at the input is an NO or NC contact.

Normally closed (NC) contact

Power flows through an NC contact if the binary operand has the signal state "0". If the

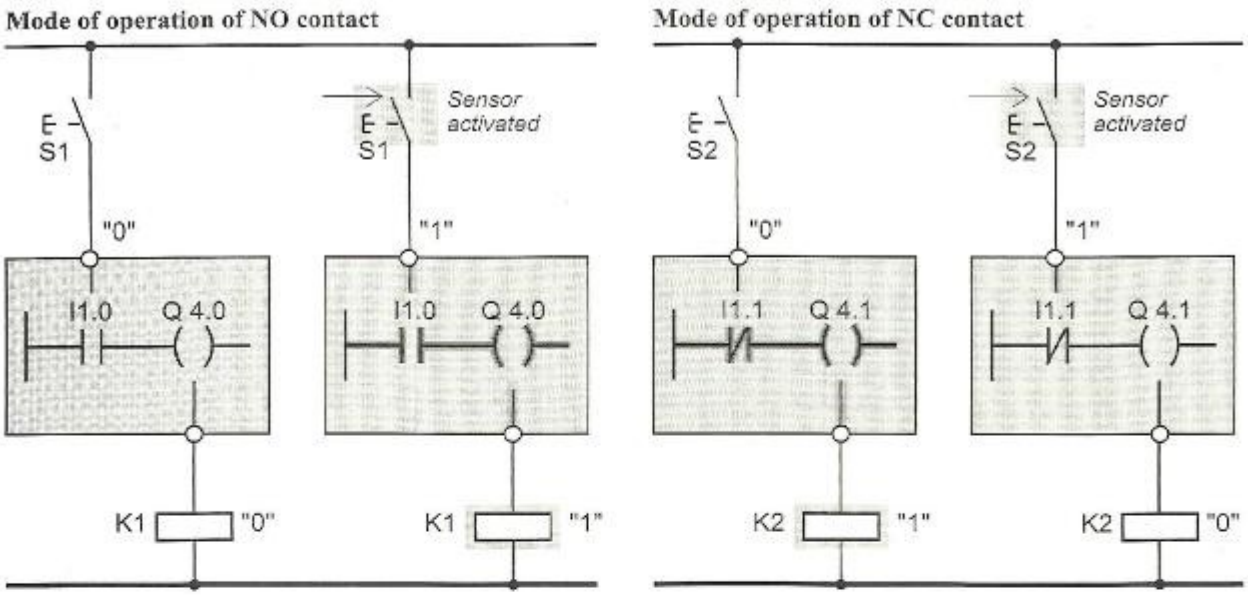


Figure 4.1 NO Contacts and NC Contacts

signal state is "1", an NC contact "opens" and the flow of power is interrupted.

In the example in Figure 4.1 (right side), power flows through the NC contact if sensor S2 is not closed (input I 1.1 has signal state "0"). Power also flows in the coil and energizes contactor K2 at output Q 4.1.

If sensor S2 is now activated, input I 1.1 has signal state "1" and the NC contact opens. The power flow is interrupted and contactor K2 releases.

The NC contact checks the input for signal state "0" and then remains closed, regardless of whether the sensor at the input is an NO or NC contact (also see Chapter 4.3 "Taking Account of the Sensor Type").

4.1.2 Series Circuits

In series circuits, two or more contacts are connected in series. Power flows through a series circuit when all contacts are closed.

Figure 4.2 shows a typical series circuit. In network 1, the series circuit has three contacts; any binary operands can be scanned. All contacts are NO contacts. If the associated operands all have signal state "1" (that is, if the NO contacts are activated), power flows through the rung to the coil. The operand controlled by the coil is set to "1". In all other cases, no power flows and the operand *Coil1* is reset to "0".

Network 2 shows a series circuit with one NC contact. Power flows through an NC contact if the associated operand has signal state "0" (that is, the NC contact is not activated). So power only flows through the series circuit in the example if the operand *Contact4* has signal state "1" and the operand *Contact5* has signal state "0".

4.1.3 Parallel Circuits

When two or more contacts are arranged one under the other, we refer to a parallel circuit. Power flows through a parallel circuit if one of the contacts is closed.

Figure 4.2 shows a typical parallel circuit. In network 3, the parallel circuit consists of three contacts; any binary operands can be scanned. All contacts are NO contacts. If one of the operands has signal state "1", power flows through the rung to the coil. The operand controlled by the coil is set to "1". If all operands scanned have signal state "0", no power flows to the coil and the operand *Coil3* is reset to "0".

Network 4 shows a parallel circuit with one NC contact. Power flows through an NC contact if the associated operand is "0", that is, power flows through the series circuit in the example if the operand *Contact4* has signal state "1" or the operand *Contact5* has signal state "0".

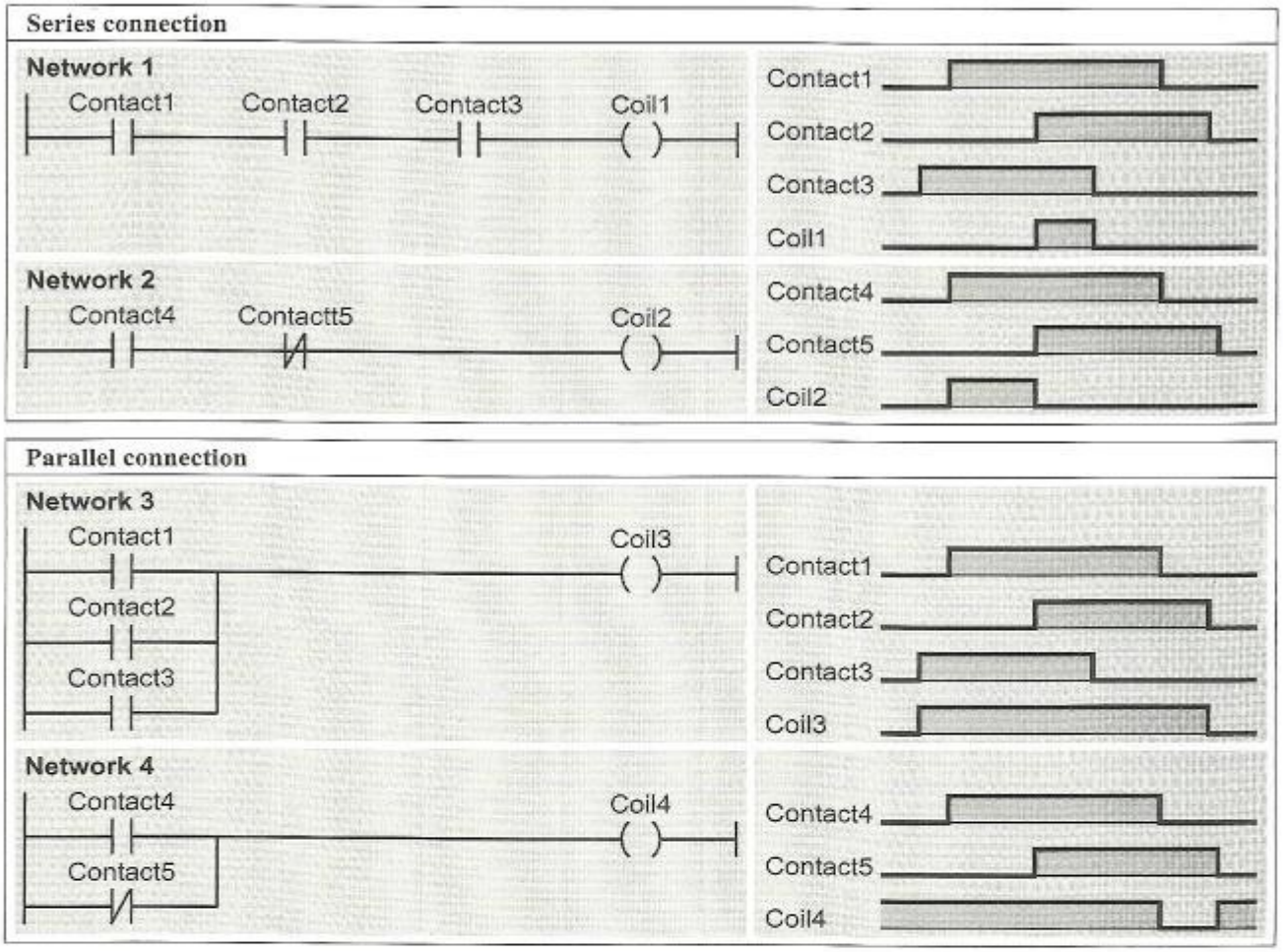


Figure 4.2 Series and Parallel Circuits

LAD, you can also program a branch in the middle of the rung (for an example, see Figure 4.3 network 8). You then get a parallel branch that does not begin at the left power rail. Use of LAD program elements is restricted to this parallel branch; your attention is drawn to this in the relevant chapters.

An "open" parallel circuit is called a "T-branch".

4.1.4 Combinations of Binary Logic Operations

You can combine series and parallel circuits, for example, by arranging several series circuits in parallel or several parallel circuits in series. You can combine series and parallel circuits even when both types are complex in nature (Figure 4.3).

Connecting series circuits in parallel

Instead of contacts, you can also arrange series circuits one under the other. Figure 4.3 shows two examples. In network 5, power flows into the coil if *Contact1* and *Contact2* are closed or if *Contact3* and *Contact4* are closed. In the lower rung (network 6), power flows if *Contact5* or *Contact6* and *Contact7* or *Contact0* are closed.

Connecting parallel circuits in series

Instead of contacts, you can also arrange parallel circuits in series. Figure 4.3 shows two examples. In network 7, power flows into the coil if either *Contact1* or *Contact3* and either *Contact2* or *Contact4* are closed. To allow power to flow in the lower example (network 8), *Contact5*, *Contact0* and either *Contact6* or *Contact7* must be closed.

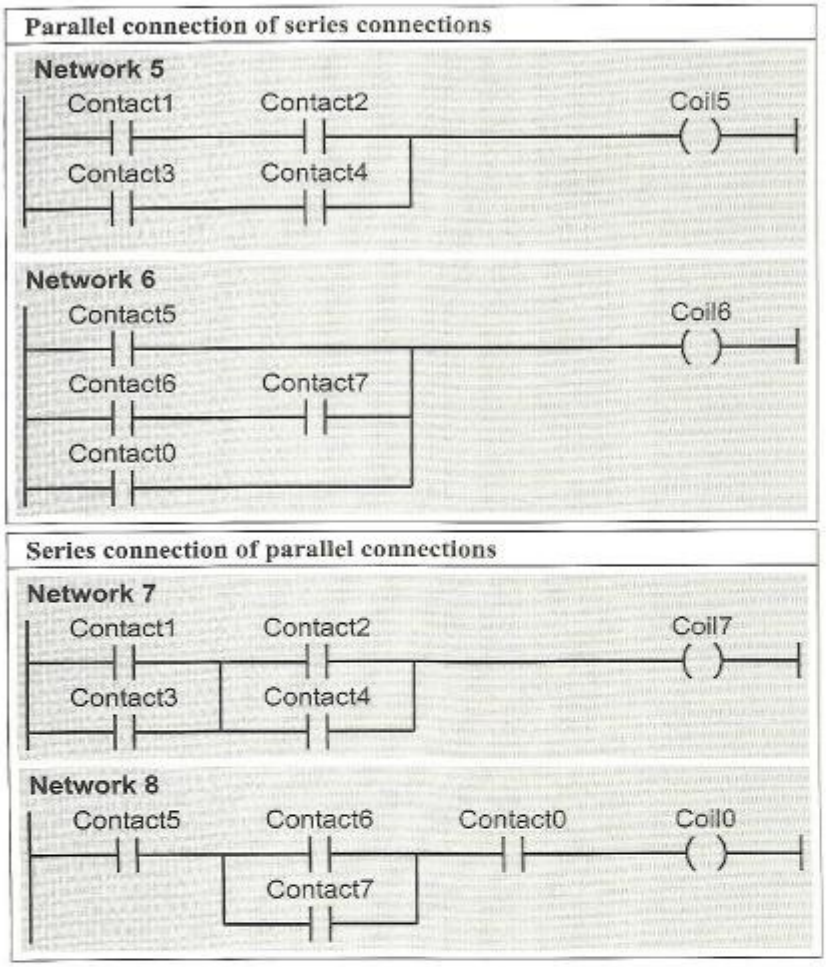


Figure 4.3 Series and Parallel Circuits in Combination

4.1.5 Negating the Result of the Logic Operation

The NOT contact negates the RLO. You can use this contact, for example, to run a series circuit negated to a coil (Figure 4.4 Network 9). Power will then only flow into the coil if there is no power in the NOT contact, that is, if either *Contact1* or *Contact2* is open (see the Figure in the adjacent pulse diagram).

The same applies by analogy for network 10, in which a NOT contact is inserted after a parallel circuit. Here, *Coil10* is set if neither of the contacts is closed.

You can insert NOT instead of another contact into a branch that begins at the left power rail. Inserting a NOT contact in a parallel branch that begins in the middle of a rung is not permissible.

4.2 Binary Logic Operations (FBD)

In FBD, the logic operations performed on binary signal states take the form of AND, OR and Exclusive OR functions. The operands whose signal states you want to scan and combine are written at the inputs of these functions. You can scan the following operands:

- Input and output bits, memory bits (discussed in this section)
- Timers and counters
- Global data bits
- Temporary local data bits
- Static local data bits
- Status bits (evaluation of calculation results)

Every binary operand can be addressed absolutely or symbolically. When scanning a binary

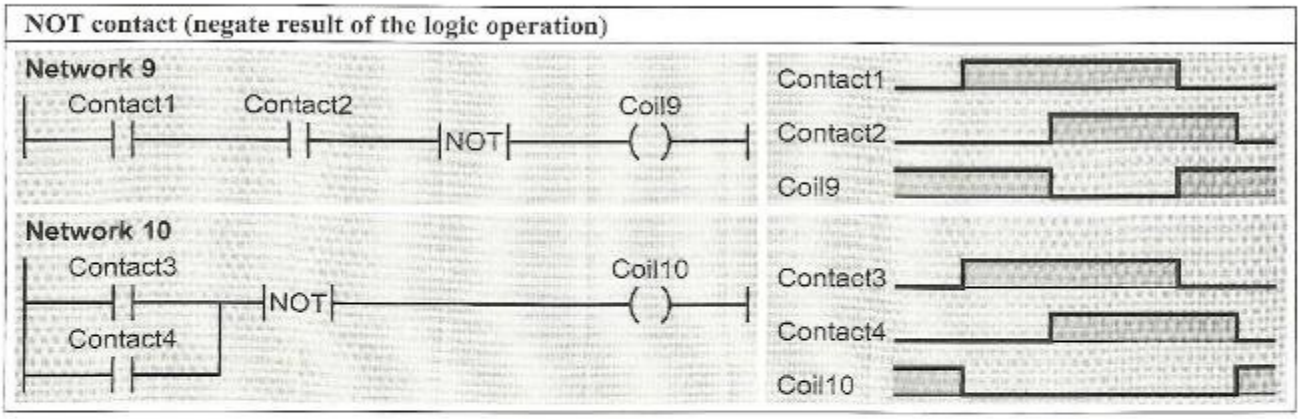


Figure 4.4 Examples of a NOT Contact

operand, or within a binary logic circuit, you can negate the result of the logic operation with the negation symbol (which is a circle).

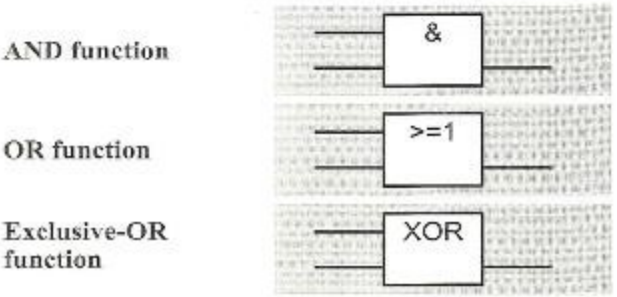
In FBD, you program one binary logic circuit per network. The logic circuit may consist of only one or of a very large number of interconnected functions. A logic circuit, or logic operation, must always be terminated, for example with an assign statement. The assign controls a binary operand with the result of the logic operation.

The examples shown in this chapter can be found in function block FB 104 of the "Basic Functions" program in the "FBD_Book" library that you can download from the publisher's Website (see page 8).

For incremental programming, you will find the elements for binary logic operations in the Program Element Catalog (VIEW → OVERVIEWS [Ctrl - K] or with INSERT → PROGRAM ELEMENTS) under "Bit Logic".

4.2.1 Elementary Binary Logic Operations

FBD uses the binary functions AND, OR, and Exclusive OR. All functions may (theoretically) have any number of function inputs. If an input leads directly to the function element, the signal state of the operand scanned is used directly in the logic operation; if the input has a negation character (a circle), the signal state of the scanned operand is negated prior to execution of the logic operation (see below).



The number of binary functions and the scope of a binary function are theoretically unlimited; in practice, however, limits are set by the length of a block or the size of the CPU's main memory.

Scanning and assigning signal states

Before the binary functions perform logic operations on signal states, they scan the binary operands at the function inputs. An operand can be scanned for "1" or "0". If scanned for "1", the function input leads directly to the box. A scan for "0" is recognizable by the negation character at the function input.

A scan for "1" produces a scan result of "1" when the signal state of the binary operand scanned is "1"; it produces a scan result of "0" when the signal state of the binary operand is

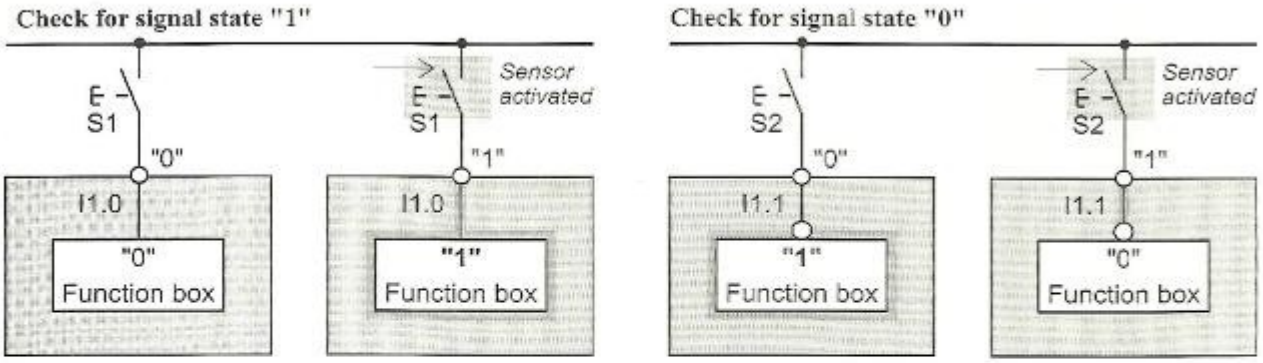


Figure 4.5 Scanning for Signal State "1" and "0"

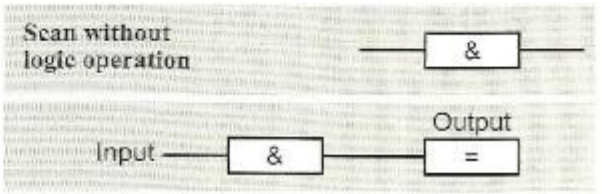
"0". A scan for signal state "0" negates the scan result, that is, the scan result is "1" when the status of the binary operand scanned is "0". The binary functions combine the *scan result*, which is, at it were, the result applied "directly" to the box. As far as functionality is concerned, these two methods of scanning binary operands allow you to treat NO contacts and NC contacts identically.

Here an example: "0" is applied to the input module for a non-activated NO contact (Figure 4.5). A scan for signal state "1" forwards this status to a function box. To effect the same for an NC contact, you have to scan an input with an NC contact for signal state "0" (must include a circle for negation). The signal state "1" applied to the input module for a non-activated NC contact is then converted into signal state "0" at the function box.

If you now activate both the NO and NC contacts, the function box will show signal state "1" in both cases. Additional information can be found in Chapter 4.3 "Taking Account of the Sensor Type").

You must always connect the output of a binary function; in the simplest case, simply connect the output to an Assign box (also see Chapter 5 "Memory Functions"). With this result of the logic operation, you can also start a timer, execute a digital operation, call a block, and so on. The next chapter provides all the information you need.

To assign the signal state of a binary operand directly to another binary operand without performing any additional logic operations, for example to connect an input directly to an output, the AND function is normally used, although it would also be possible to use an OR or Exclusive OR.



Simply select the AND function, connecting only one function input and removing the other.

AND function

The AND function combines two binary states with one another and produces an RLO of "1" when both states (both scan results) are "1". If the AND function has several inputs, the scan results of all inputs must be "1" for the collective RLO to be "1". In all other cases, the AND function produces an RLO of "0" at its function output.

Figure 4.6 shows an example of an AND function. In Network 1, the AND function has three inputs, each of which can be connected to any binary operand. All operands are scanned for signal state "1", so that the signal state of the operands is directly ANDed. If all the operands that were scanned have a signal state of "1", the AND function sets the operand *Output1* to "1" via the Assign box (see next chapter). In all other cases, the AND condition is not fulfilled and operand *Output1* is reset to "0".

Network 2 shows an AND function with a negated input. Negation of the input is indicated by a circle. The scan result for a negated

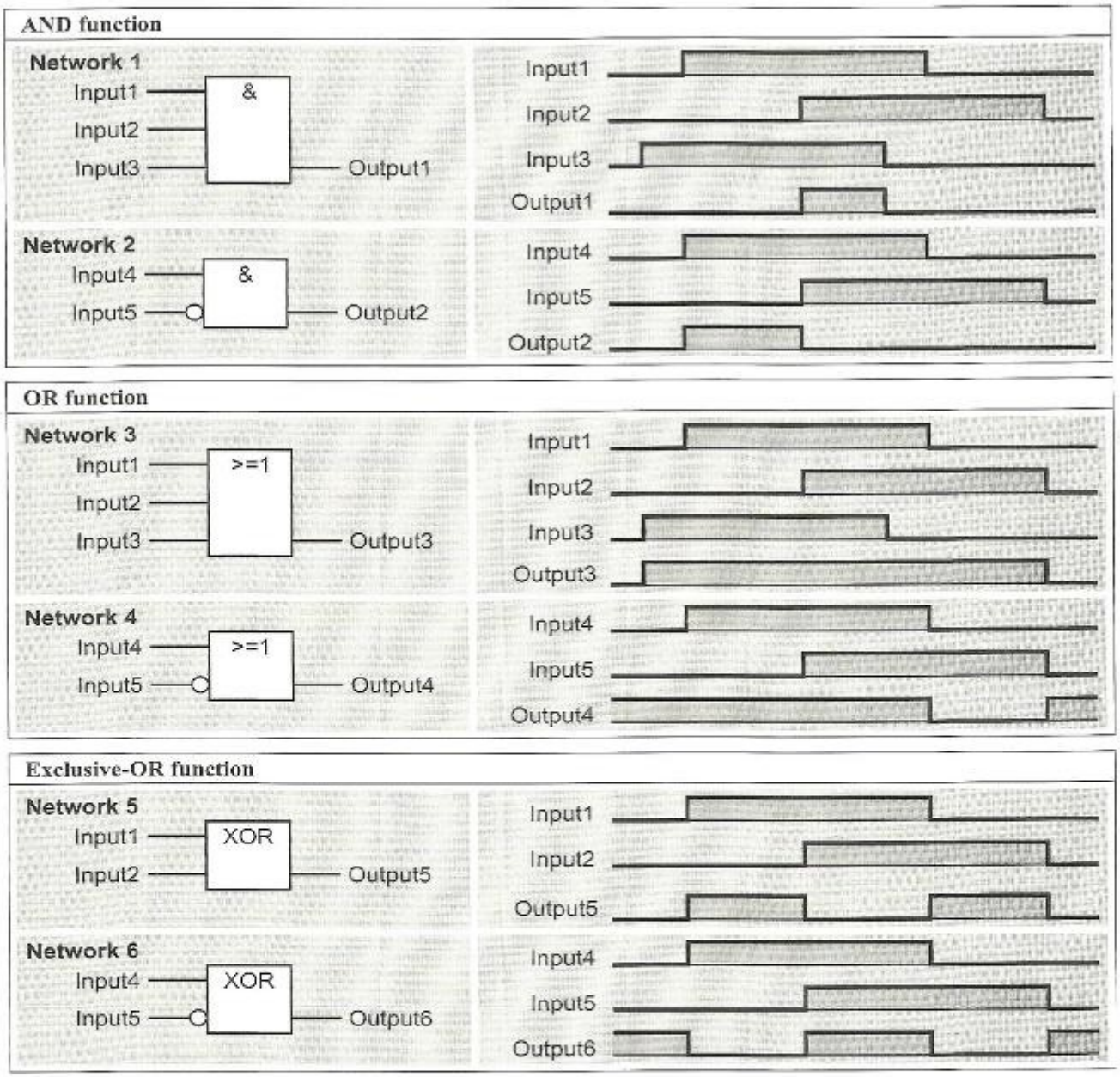


Figure 4.6 Examples of Binary Functions

operand is "1" when this operand is "0", that is, the AND condition in the example is fulfilled when the operand *Input4* is "1" and the operand *Input5* is "0".

OR function

The OR function combines two binary signal states and returns an RLO of "1" when one of these states (one of the scan results) is "1". If the OR function has several inputs, the scan result of only one input need be "1" in order for the result of the logic operation (RLO) to be "1". The OR function returns an RLO of "0" when the scan results of all inputs are "0".

Figure 4.6 shows an example of an OR function. In Network 3, the OR function has three inputs; each of these inputs may be connected to any binary operand. All operands are scanned for signal state "1", so that the signal state of the operands is directly ORed. If one or more of the operands scanned have signal state "1", the next statement sets the operand

Output1 to "1". If all of the operands scanned have signal state "0", the OR condition is not fulfilled and operand *Output1* is reset to "0".

Network 4 shows an OR function with a negated input. Negation is represented by a circle. The scan result of a negated operand is "1" when that operand is "0", that is, the OR condition in the example is fulfilled when the operand *Input4* has a signal state of "1" or the operand *Input5* has a signal state of "0".

Exclusive OR function

The Exclusive OR function combines two binary states with one another and returns an RLO of "1" when the two states (scan results) are not the same, and RLO "0" when the two states (scan results) are identical.

Figure 4.6 shows an example of an Exclusive OR function. In Network 5, two inputs, both of which are scanned for signal state "1", lead to the Exclusive OR function. If only one of the operands scanned is "1", the Exclusive OR condition is fulfilled and the operand *Output1* is set to "1". If both operands are "1" or "0", operand *Output1* is reset to "0".

Network 6 shows an Exclusive OR function with a negated input. Negation is represented by a circle. The scan result of a negated operand is "1" when that operand is "0", that is, the Exclusive OR condition in the example is fulfilled when both input operands have the same signal state.

You can also program an Exclusive OR function with more than two inputs, in which case the Exclusive OR condition is fulfilled (in the case of a direct scan) when an uneven number of the operands scanned have a scan result of "1".

4.2.2 Combinations of Binary Logic Operations

You can easily combine binary functions with one another. For instance, you can combine several AND functions into one OR function or two OR functions into one Exclusive OR function. The number of functions per logic operation (per network) is theoretically unlimited.

The use of a "T-branch" in a logic operation gives you additional options, allowing you to program more than one output per logic operation (see Chapter 5.2 "FBD Boxes").

You can link the output of one binary function with the input of another binary function in order to implement complex binary logic operations. Figure 4.7 provides a number of examples.

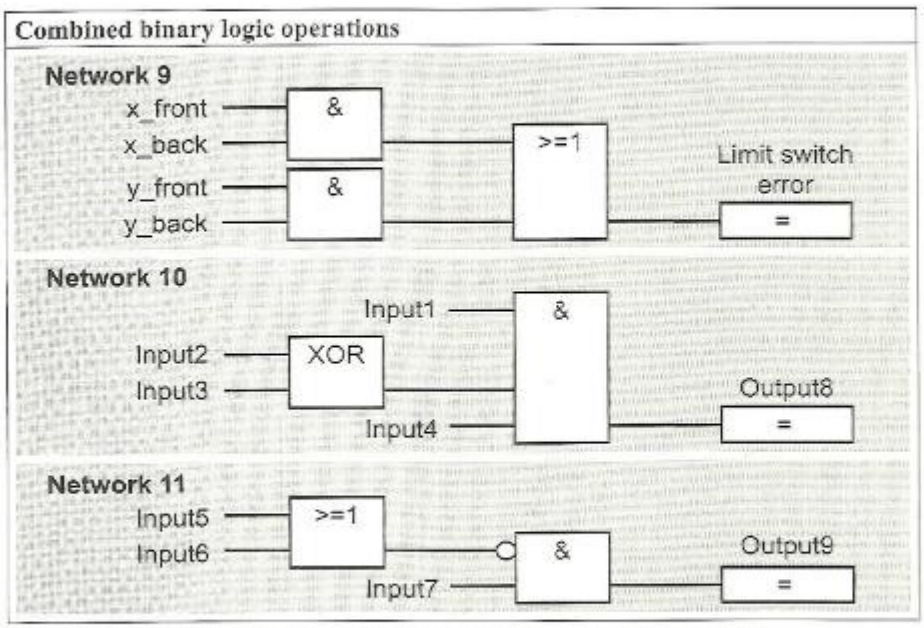


Figure 4.7 Examples of Binary Logic Operations in Combination

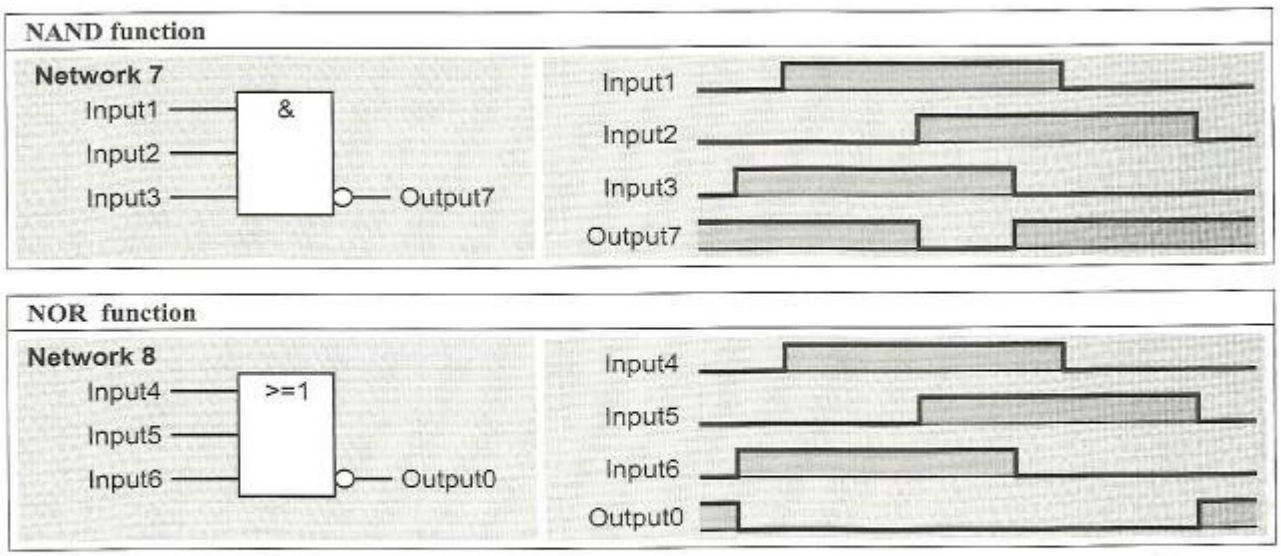


Figure 4.8 NAND and NOR Functions

Network 9: You are monitoring the limit switches at the ends of an X axis and a Y axis. These limit switches may not be actuated in pairs; otherwise, *limit switch error* will be reported.

Network 10: You can link arbitrary function inputs with binary functions, for example you can place an Exclusive OR function in front of the second input of an AND function

Network 11: Using the negation symbol, you negate the RLO, even between binary functions, for instance you can negate the RLO of an OR function and use it as input to an AND function.

4.2.3 Negating the Result of the Logic Operation

The circle at the input or output of a function symbol negates the result of the logic operation. You can use negation

- to scan a binary operand, which is equivalent to scanning for signal state "0" (see above),
- between two binary functions (which is equivalent to negating the result of the logic operation), or
- at the output of a binary function (for example if you want to set or reset a binary operand when the condition is not fulfilled, that is, when RLO = "0").

A negation may not immediately follow a T-branch.

Figure 4.8 shows a NAND function (an AND function with negated output) and a NOR function (an OR function with negated output). The RLO of a NAND function is "0" only when all inputs have a signal state of "1". A NOR function returns an RLO of "1" only when none of the inputs has a signal state of "1".

4.3 Taking Account of the Sensor Type

When scanning a sensor in a user program, you must take account of whether the sensor is an NO contact or an NC contact. Depending on the sensor type, there is a different signal state at the relevant input when the sensor is activated: "1" for an NO contact and "0" for an NC contact. The CPU has no means of determining whether an input is occupied by an NO contact or by an NC contact. It can only detect signal state "1" or signal state "0".

Programming with LAD

If you structure the program in such a way that you want a scan result of "1" when a sensor is activated in order to combine that scan result further, you must scan the input differently for different kinds of sensors. NO contacts and NC

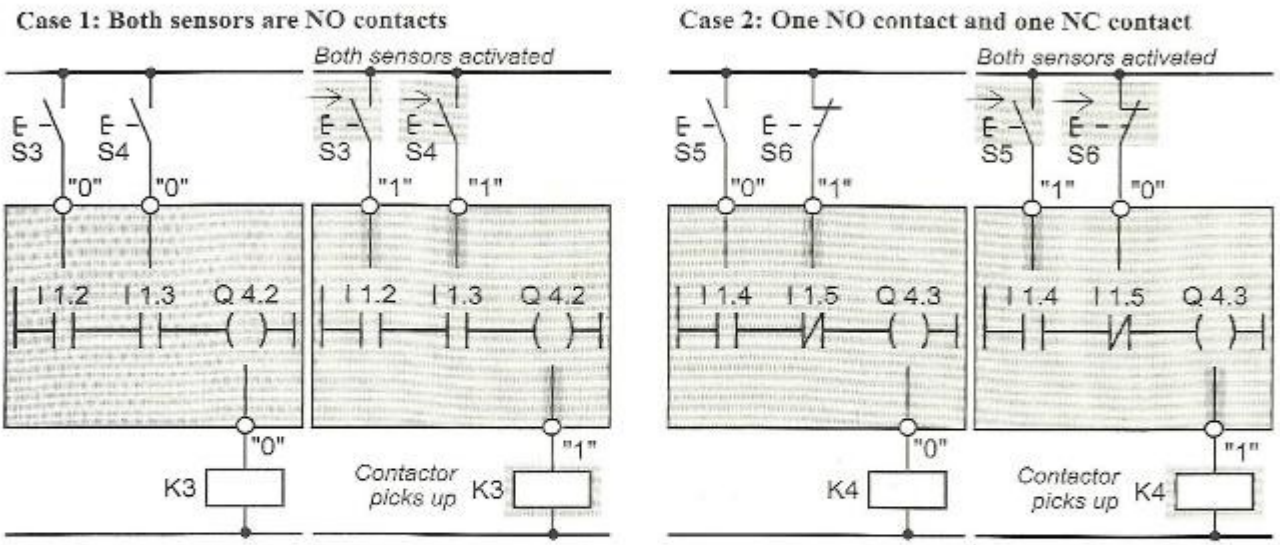


Figure 4.9 Taking Account of the Sensor Type (LAD)

contacts are available to you for this purpose. An NO contact return "1" if the scanned input is also "1". An NC contact returns "1" if the scanned input is "0". In this way, you can also directly scan inputs that are to trigger activities when they are "0" (zero-active inputs) and subsequently re-gate the scan result.

The example in Figure 4.9 shows programming dependent on the sensor type. In the first case, two NO contacts are connected to the programmable controller, and in the second case one NO contact and one NC contact. In both cases, a contactor connected to an output is to pick up if both sensors are activated. If an NO contact is activated, the signal state at the input is "1", and this is scanned with an NO contact so that power can flow when the sensor is activated. If both NO contacts are activated, power flows through the rung to the coil and the contactor picks up.

If an NC contact is activated, the signal state at the input is "0". In order to have power flow in this case when the sensor is activated, the result must be scanned with an NC contact. Therefore, in the second case, an NO contact and an NC contact must be connected in series to make the contactor pick up when both sensors are activated.

Programming with FBD

If you structure the program in such a way that you want a scan result of "1" when a sensor is activated in order to combine that scan result further, you must scan the input differently for different kinds of sensors. An NO contact produces signal state "1" when activated, and is scanned directly when activation of the sensor is to produce a scan result of "1". An NC contact returns signal state "0" when activated; if you want a scan result of "1" when the NC contact is activated, it must be negated, then scanned. In this way, you can also scan inputs that are to trigger activities even when they have a signal state of "0" (zero-active inputs) and further combine the scan result.

The example in Figure 4.10 shows sensor type-dependent programming. In the first case, two NO contacts are connected to the programmable controller, and in the second case one NO contact and one NC contact. In both cases, a contactor connected to an output is to pick up if both sensors are activated. If an NO contact is activated, the signal state at the input is "1", and this is scanned directly so that the scan result is "1" when the sensor is activated. If both NO contacts are activated, and AND condition is fulfilled and the contactor picks up.

If an NC contact is activated, the signal state at the input is "0". In order to obtain a scan result of "1", the input must be negated when scanned. In the second case, you need an AND function with one direct and one negated input in order for the contactor to pick up when both sensors are activated.

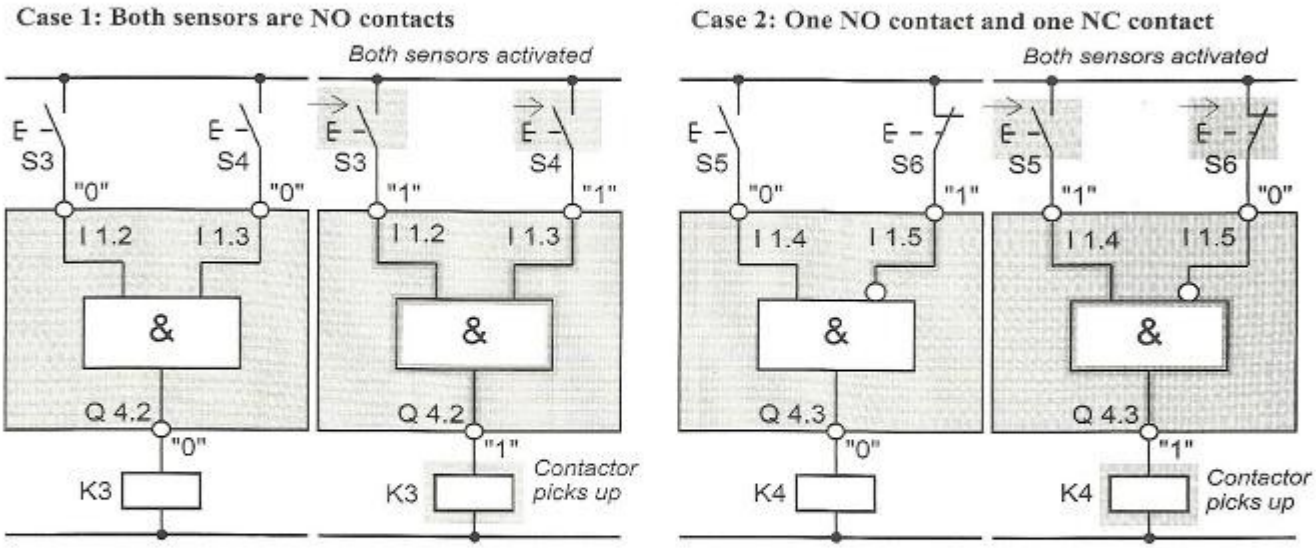


Figure 4.10 Taking Account of the Sensor Type (^FBD)

5 Memory Functions

5.1 LAD Coils

In a ladder diagram (LAD), the memory functions are used in conjunction with series and parallel circuits in order to influence the signal states of the binary operands with the aid of the result of the logic operation (RLO) generated in the CPU.

The following memory functions are available

- The single coil as an assignment of the RLO
- The coils S and R as individually programmed memory functions
- The boxes RS and SR as memory functions
- The midline outputs as intermediate buffers
- The coils P and N as edge evaluations of the power flow
- The boxes POS and NEG as edge evaluations of operands

Midline outputs and edge evaluations are discussed in detail in subsequent chapters.

You can use the memory functions described in this chapter in conjunction with all binary operands. There are restrictions when using temporary local data bits as edge memory bits.

The examples shown in this chapter are found in function block 105 of the "Basic Functions" program in the "LAD_Book" library that you can download from the publisher's Website (see page 8).

For incremental programming, you will find the program elements for the memory functions in the program element catalog (with VIEW → OVERVIEWS [Ctrl - K] or with INSERT → PROGRAM ELEMENTS) under "Bit Logic".

5.1.1 Single Coil

The single coil as terminator of a rung assigns the power flow directly to the operand located at the coil. The function of the single coil depends on the Master Control Relay (MCR): If the MCR is activated, signal state "0" is assigned to the binary operand located over the coil.

If power flows into the coil, the operand is set; if there is no power, the operand is reset (Figure 5.1 Network 1). With a NOT contact before the coil, you reverse the function (Network 2).

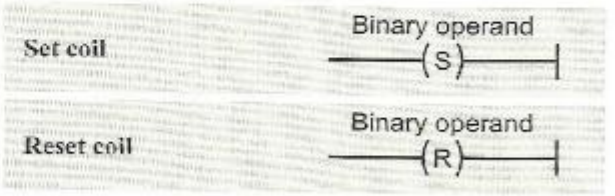
You can also direct the power flow into several coils simultaneously by arranging the coils in parallel with the help of a T-branch (Network 3). All operands specified over the coils respond in the same way. Up to 16 coils can be connected in parallel.

You can arrange further contacts in series and parallel circuits after the T-branch and before the coil (Network 4).

See Chapter 4.1 "Series and Parallel Circuits (LAD)", for further examples of the single coil.

5.1.2 Set and Reset Coil

Set and reset coils also terminate a rung. These coils only become active when power flows through them.



If power flows in the set coil, the operand over the coil is set to signal state "1". If power flows in the reset coil, the operand over the coil is reset to signal state "0". If there is no power in the set or reset coil, the binary operand remains unaffected (Figure 5.1, Networks 5 and 6).

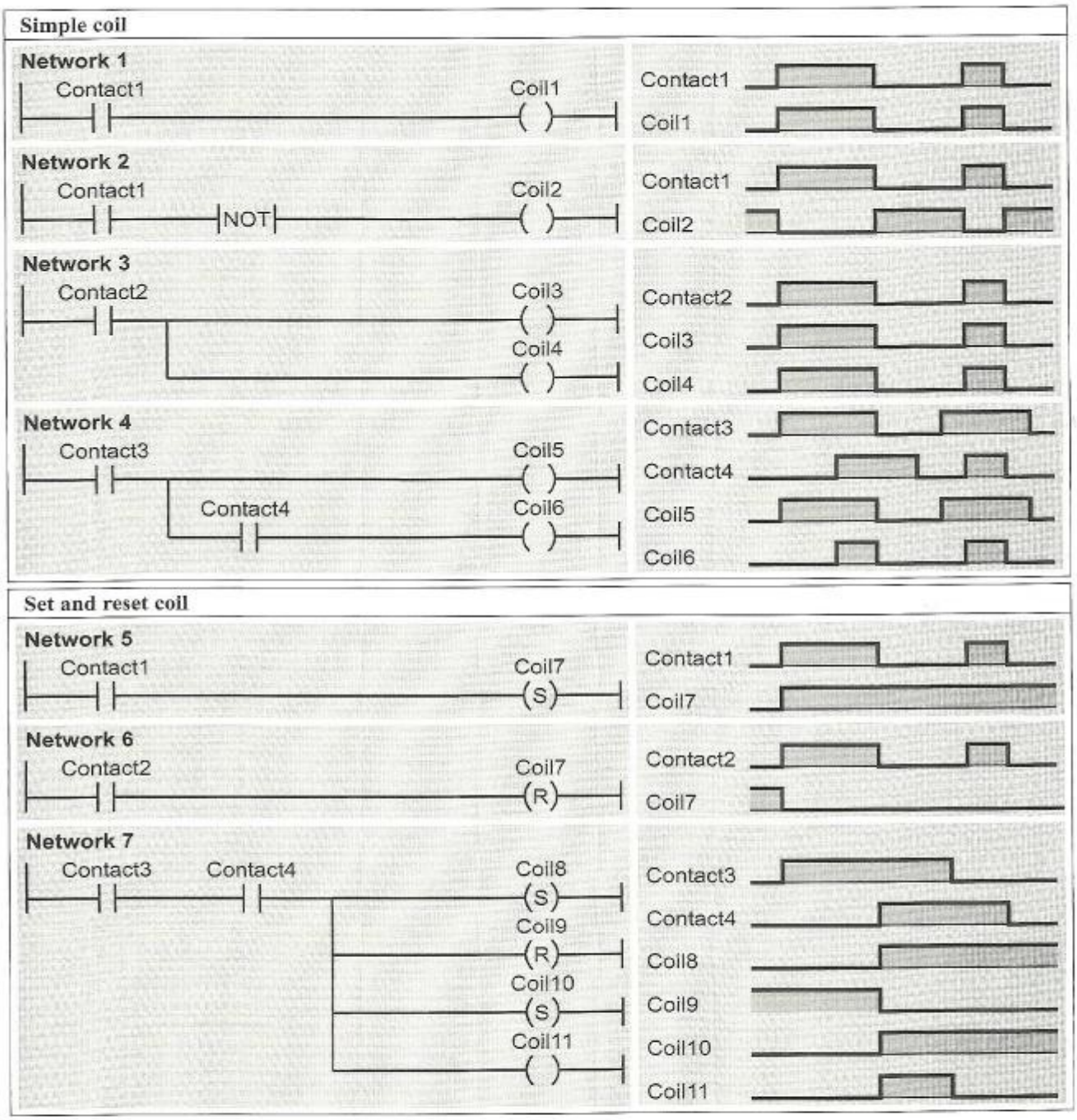


Figure 5.1 Single Coil, Set and Reset Coil

The function of the set and reset coils depends on the Master Control Relay. If the MCR is activated, the binary operand over the coil is not affected.

Please note that the operand used with a set or reset coil is usually reset at startup (complete restart). In special cases, the signal state is retained: This depends on the startup mode (for example warm restart), on the operand used (for example static local data), and on the settings in the CPU (for example retentive characteristics).

You can arrange several set and reset coils in any combination and together with single coils in the same rung (Network 7). To achieve clarity in your programming, it is advisable to

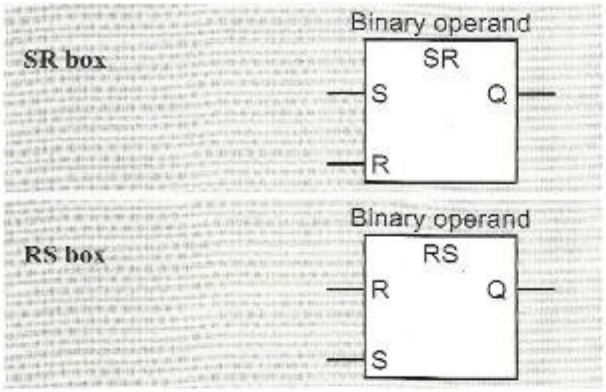
group the set and reset coils affecting an operand together in pairs, and to use them only once in each case. You should also avoid additionally controlling these operands with a single coil.

As with the single coil, you can also arrange contacts after the branch and before a set and reset coil.

5.1.3 Memory Box

The functions of the set and reset coil are summarized in the box of a memory function. The common binary operand is located over the box. Input S of the box corresponds here to the set coil, input R to the reset coil. The signal state of the binary operand assigned to the memory function is at output Q of the memory function.

There are two versions of the memory function: As SR box (reset priority) and RS box (set priority). Apart from the labeling, the boxes also differ from each other in the arrangement of the S and R inputs.



A memory function is set (or, more precisely, the binary operand over the memory box is set) if the set input has signal state "1" and the reset input has signal state "0". A memory function is reset if there is a "1" at the reset input and a "0" at the set input. Signal state "0" at both inputs has no effect on the memory function. If both inputs are "1" at the same time, the two memory functions respond differently: the SR memory function is reset and the RS memory function is set.

The function of the memory box depends on the Master Control Relay. If the MCR is active, the binary operand of a memory box is no longer affected.

Please note that the operand used with a memory function at startup (complete restart) is usually reset. In special cases, the signal state of a memory box is retained. This depends on the startup mode (for example warm restart), on the operand used (for example static local data), and on the settings in the CPU (for instance retentive characteristics).

SR memory function

In the SR memory box, the reset input has priority. Reset priority means that the memory function is or remains reset if power flows "simultaneously" in the set input and the reset input. The reset input has priority over the set input (Figure 5.2, Network 8).

Because the statements are executed in sequence, the CPU initially sets the memory operand because the set input is processed first, but resets it again when it processes the reset input. The memory operand remains reset while the rest of the program is processed.

If the memory operand is an output, this brief setting only takes place in the process-image output table, and the (external) output on the relevant output module remains unaffected. The CPU does not transfer the process-image output table to the output modules until the end of the program cycle.

The memory function with reset priority is the "normal" form of the memory function, since the reset state (signal state "0") is normally the safer or less hazardous state.

RS memory function

In the RS memory box, the set input has priority. Set priority means that the memory function is or remains set if power flows "simultaneously" in the set input and the reset input. The set input then has priority over the reset input (Figure 5.2 Network 9).

In accordance with the sequential execution of the instructions, the CPU resets the memory operand with the reset input first processed, but then sets it again when processing the set input. The memory operand remains set while the rest of the program is processed.

If the memory operand is an output, this brief resetting takes place only in the process-image

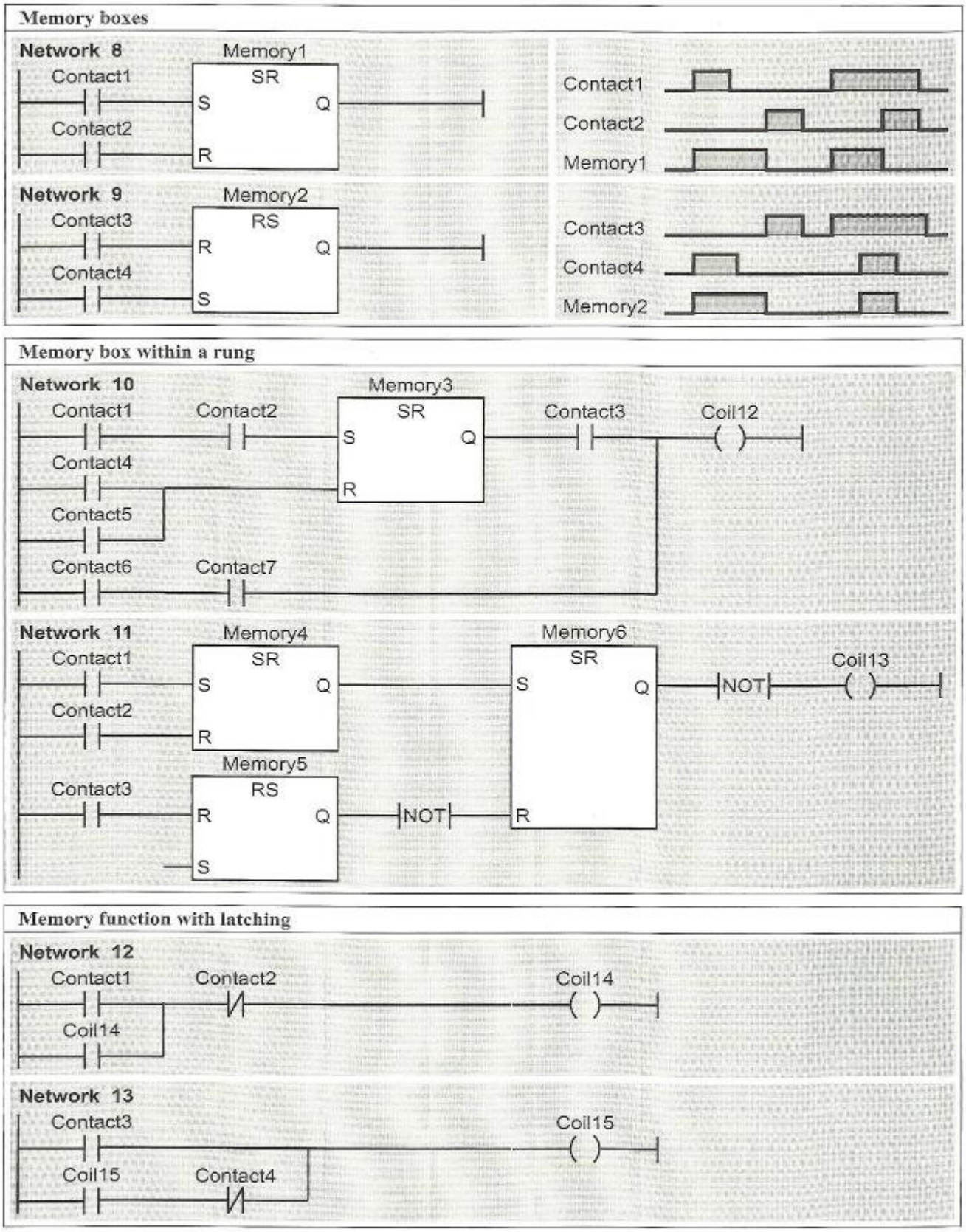


Figure 5.2 Memory Functions (LAD)

output table, and the (external) output on the relevant output module is not affected. The CPU does not transfer the process-image output table to the output modules until the end of the program cycle.

Set priority is the exception when using the memory function. It is used, for example, in the implementation of a fault message buffer if the still current fault message at the set input is to continue to set the memory function despite an acknowledgement at the reset input.

Memory function within a rung

You can also place a memory box within a rung. Contacts can be connected in series and in parallel both at the inputs and at the output (Figure 5.2 Network 10). It is also possible to leave the second input of a memory box unswitched. You can also connect several memory boxes together within one rung. You can arrange the memory boxes in series or in parallel (Network 11).

You can locate a memory function after a T-branch or in a branch that starts at the left power rail

Memory function with latching

In a relay logic diagram, the memory function is usually implemented by latching the output to be controlled. This method can also be used when programming in ladder logic. However, it has the disadvantage, when compared with the memory box, that the memory function is not immediately recognizable.

Networks 12 and 13 in Figure 5.2 show both types of memory function, set priority and reset priority, using latching. The principle of latching is a simple one. The binary operand controlled with the coil is scanned, and this scan (the "contact of the coil") is connected in parallel to the set condition. If *Contact1* closes, *Coil14* energizes and closes the contact parallel to *Contact1*. If *Contact1* now opens again, *Coil14* remains energized. *Coil14* de-energizes if *Contact2* opens. If signal state "1" is present at both *Contact1* and *Contact2*, power does not flow into the coil (reset priority). This situation looks different in the lower network: If signal state "1" is present at both *Contact3* and *Contact4*, power flows into the coil (set priority).

5.2 FBD Boxes

In FBD, the memory boxes are used in conjunction with binary logic operations in order to influence the signal states of binary operands with the aid of the result of the logic operation (RLO) generated in the CPU.

Available memory functions are

- The assign box for dynamic control
- The boxes set and reset as individually programmed memory functions
- The boxes RS and SR as full-fledged memory functions
- The midline output box as intermediate buffer
- The boxes P and N as edge evaluations of the result of the logic operation
- The boxes POS and NEG as edge evaluations of operands

The boxes for midline outputs and edge evaluations are discussed in detail in subsequent chapters.

You can use the memory functions described in this chapter in conjunction with all binary operands. There are restrictions when using temporary local data bits as edge memory bits.

The examples shown in this chapter are found in function block FB 105 of the "Basic Functions" program in the "FBD_Book" library that you can download from the publisher's Website (see page 8).

For incremental programming, you will find the program elements for the memory functions in the program element catalog (with VIEW → OVERVIEWS [Ctrl - K] or with INSERT → PROGRAM ELEMENTS) under "Bit Logic".

5.2.1 Assign

The assign box as terminator of a rung assigns the result of the logic operation directly to the operand adjacent to the box. If the RLO is "1" at the input of the assign box, the binary operand is set; if the RLO is "0", the operand is reset. The function of the assign box depends on the Master Control Relay (MCR): If the

MCR is activated, signal state "0" is assigned to the binary operand over the box.

A number of examples in Figure 5.3 explain how the assign box works.

Network 1: The operand Output1 directly assumes the signal state of the operand Input1.

Network 2: You can use negation to reverse the function of the assign box.

Network 3: You can direct the RLO to several boxes simultaneously by inserting a T-branch and arranging the boxes with the relevant operands one below the other ("multiple output"). All operands over the boxes respond in the same way.

Network 4: You can insert binary functions between the T-branch and the terminating box, thus expanding a logic operation by additional function boxes.

You will find additional examples for the assign box in Chapter 4.2 "Binary Logic Operations (FBD)".

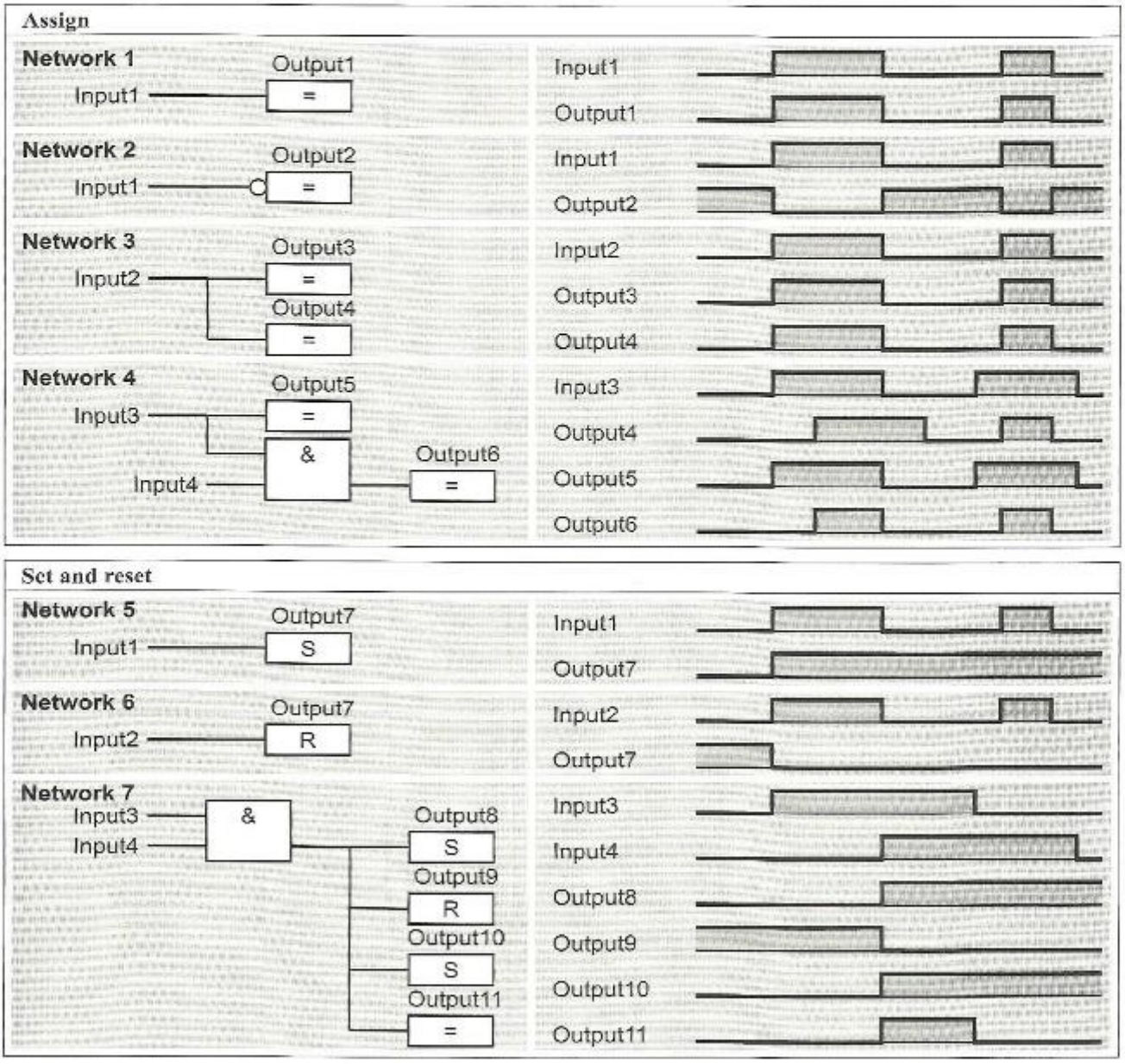
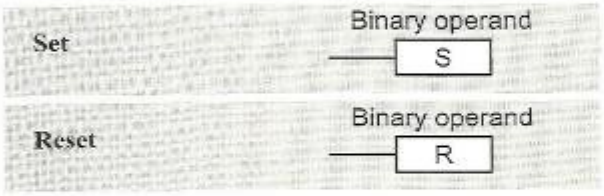


Figure 5.3 Assign, Set and Reset (FBD)

5.2.2 Set and Reset Box

Set and reset boxes also terminate a logic operation. These boxes are activated only when the result of the logic operation going into the box is "1".



If the RLO going into the set box is "1", the operand over the box is set to signal state "1". If the RLO going into the reset box is "1", the operand over the box is set to signal state "0". If the RLO going into the set or reset box is "0", the binary operand remains unaffected. The function of the set and reset boxes depends on the Master Control Relay. If the MCR is activated, the binary operand over the box is not affected.

Figure 5.3 shows several examples of how the set and reset boxes work.

Network 5: The operand *Output7* is set when the operand *Input1* is "1". *Output7* remains set when *Input1* returns to signal state "0".

Network 6: The operand *Output7* is reset when the operand *Input2* is "1". *Output7* remains reset when *Input2* returns to signal state "0".

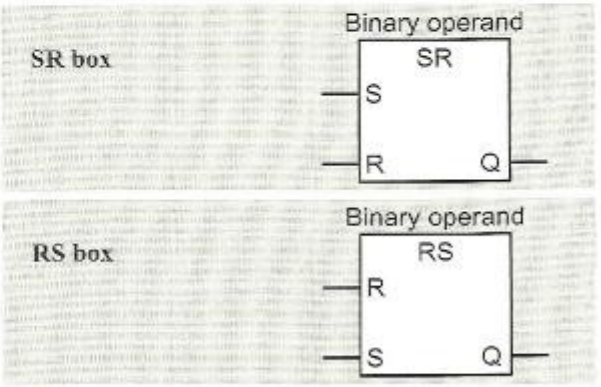
Network 7: You can arrange several set and reset boxes in any combination and together with assign boxes in the same box after a T-branch. As with the assign box, you can also program binary functions after the T-branch and before a set and reset box.

To achieve clarity in your programming, it is advisable to group the set and reset boxes affecting an operand in pairs, and to use them only once in each case. You should also avoid controlling these operands with an assign box.

Please note that the operand used with a set or reset box is usually reset on startup (cold restart or warm restart). In special cases, the signal state is retained. This depends on the startup mode (for instance hot restart), on the operand used (for example static local data), and on the settings in the CPU (such as retentive characteristics).

5.2.3 Memory Box

The functions of the set and reset boxes are summarized in the box of a memory function. The common binary operand is located over the box. Input S of the box corresponds here to the set box, and input R to the reset box. The signal state of the binary operand assigned to the memory function is at output Q of the memory function.



There are two versions of the memory function: As SR box (reset priority) and as RS box (set priority). Apart from the labeling, the boxes also differ from each other in the arrangement of the S and R inputs.

A memory function is set (or, more precisely, the binary operand over the memory box is set) when the set input is "1" and the reset input is "0". A memory function is reset when the reset input is "1" and the set input is "0". Signal state "0" at both inputs has no effect on the memory function. If both inputs have signal state "1" simultaneously, the two memory functions respond differently: the SR memory function is reset and the RS memory function is set.

The function of the memory box depends on the Master Control Relay. If the MCR is active, the binary operand of a memory box is no longer affected.

Please note that the operand used with a memory function is normally reset on startup (cold restart or warm restart). In special cases, the signal state of a memory box is retained. This depends on the startup mode (for instance hot restart), on the operand used (for example static local data), and on the settings in the CPU (such as retentive characteristics).

SR memory function

In the SR memory box, the reset input has priority. Reset priority means that the memory function is or remains reset if the RLO is "1" at the set and reset inputs "simultaneously". The reset input then has priority over the set input (Figure 5.4, Network 8).

Because the statements are executed in sequence, the CPU first sets the memory operand because the set input is processed first, but resets it again when it processes the reset input. The memory operand remains reset while the rest of the program is processed.

If the memory operand is an output, this brief setting takes place only in the process-image

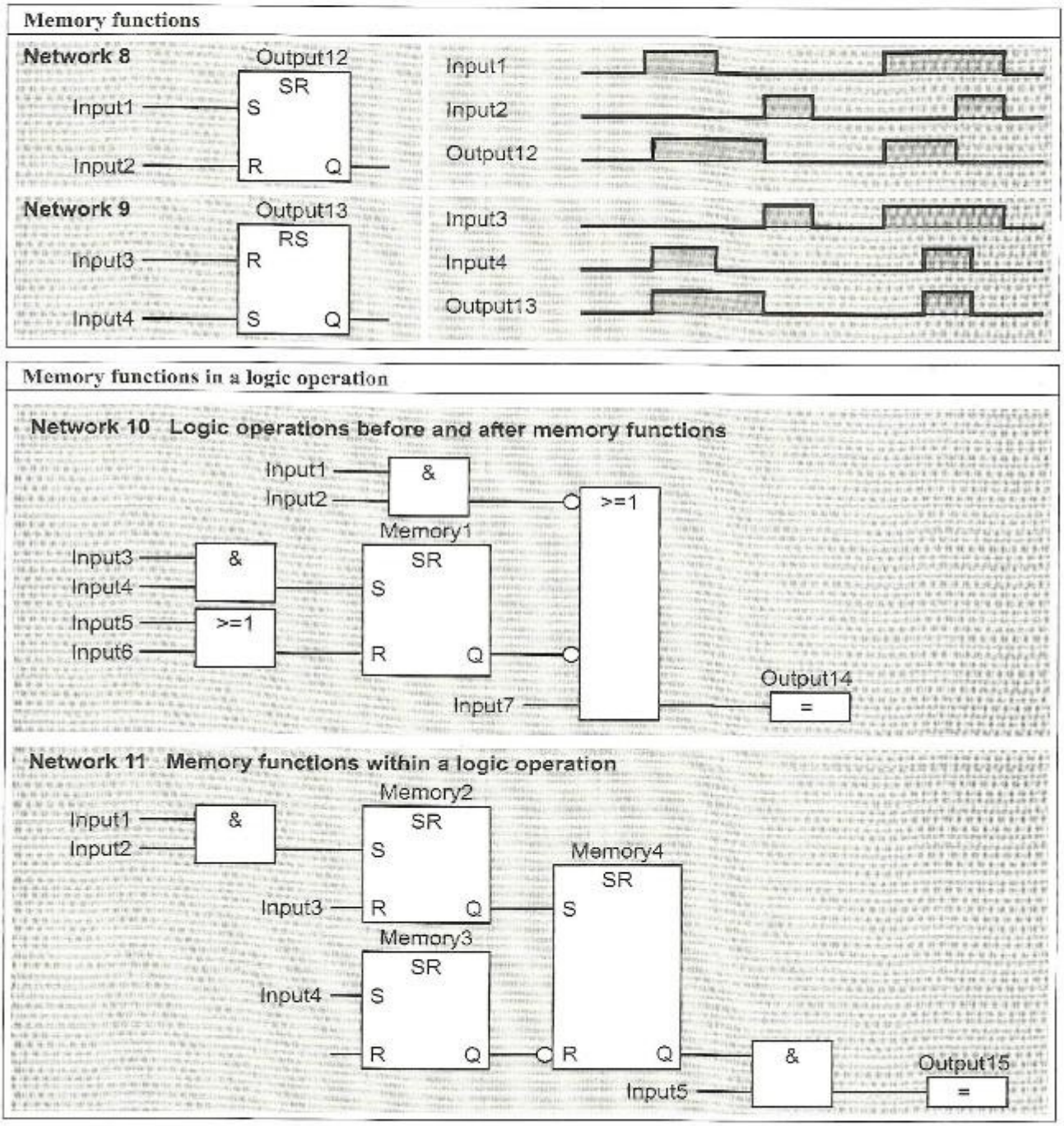


Figure 5.4 Memory Functions (FBD)

output table, and the (external) output on the relevant output module remains unaffected. The CPU does not transfer the process-image output table to the output modules until the end of the program cycle.

The memory function with reset priority is the "normal" form of the memory function, as the reset state (signal state "0") is usually the safer or less hazardous state.

RS memory function

In the RS memory box, the set input has priority. Set priority means that the memory function is or remains set if the RLO is "1" at the set and reset inputs "simultaneously". The set input then has priority over the reset input (Figure 5.4, Network 9).

Because the statements are executed in sequence, the CPU initially resets the memory operand because the reset input is processed first, then sets it again when the set input is processed. The memory operand remains set while the rest of the program is processed.

If the memory operand is an output, this brief resetting takes place only in the process-image output table, and the (external) output on the relevant output module is not affected. The CPU does not transfer the process-image output table to the output modules until the end of the program cycle.

Set priority is the exception rather than the rule. Set priority is used, for example, in the implementation of a fault message buffer if the still current fault message at the set input is to continue to set the memory function despite an acknowledgement at the reset input.

Memory function within a logic operation

You can also place a memory box within a logic operation. Binary functions can be programmed both at the inputs and at the output (Figure 5.4, Networks 10 and 11). It is also possible to leave the second input of a memory box unconnected. Within a logic operation, you can also interconnect several memory boxes. The memory boxes may be placed one behind the other or under one another after a T-branch.

5.3 Midline Outputs

Midline outputs are intermediate binary buffers in a ladder diagram or function block diagram. The RLO valid at the midline output is stored in the operand over the midline output. This operand can be scanned again at another point in the program, allowing you to also post-process the RLO valid at midline output elsewhere in the program.

The following binary operands are suitable for intermediate storage of binary results:

- You can use temporary local data bits if you only require the intermediate result within the block. All code blocks have temporary local data.
- Static local data bits are available only within a function block; they store the signal state until they are reused, even beyond the block boundaries.
- Memory bits are available globally in a fixed CPU-specific quantity; for clarity of programming, try to avoid multiple use of memory bits (the same memory bits for different tasks).
- Data bits in global data blocks are also available throughout the entire program, but before they are used require the relevant data block to be opened (even if implied through mass addressing).

The function of the midline output depends on the Master Control Relay. If the MCR is activated, the binary operand adjacent to the midline output is assigned signal state "0". The RLO is then "0" following the midline output (that is, there is no longer a "flow of power").

Note: You can replace the "scratchpad memory" used with STEP 5 by the temporary local data available in every block.

5.3.1 Midline Outputs in LAD

A midline output is a single coil within a rung. The RLO valid up to this point (the power that flows in the rung at this point) is stored in the binary operand over the midline output. The midline output itself has no effect on the power flow.

You can scan the binary operand over the midline output at another point in the program with NO and NC contacts. Several midline outputs can be programmed in one rung.

You can place a midline output in a branch that starts at the left power rail. However, it must not be located directly at the power rail. A midline output may also follow a T-branch, but may not terminate a rung; the single coil is available for this purpose.

Figure 5.5 shows an example of how an intermediate result is stored in a midline output. The RLO from the circuit formed by *Contact1*, *Contact2*, *Contact4* and *Contact5* is stored in midline output *Midl_out1*. If the condition of the logic operation is fulfilled (power flows in the midline output) and if *Contact3* is closed, *Coil16* is energized. The RLO stored is used in two ways in the next network. On the one hand, a check is made to see if the condition of the logic operation was fulfilled and the bit logic combination made with *Contact6*, and on the other hand a check is made to see if the condition of the logic operation was not fulfilled and a bit combination made with *Contact7*.

5.3.2 Midline Outputs in FBD

A midline output is an assign box within a logic operation. The RLO valid up to this point is stored in the midline output over the midline output box.

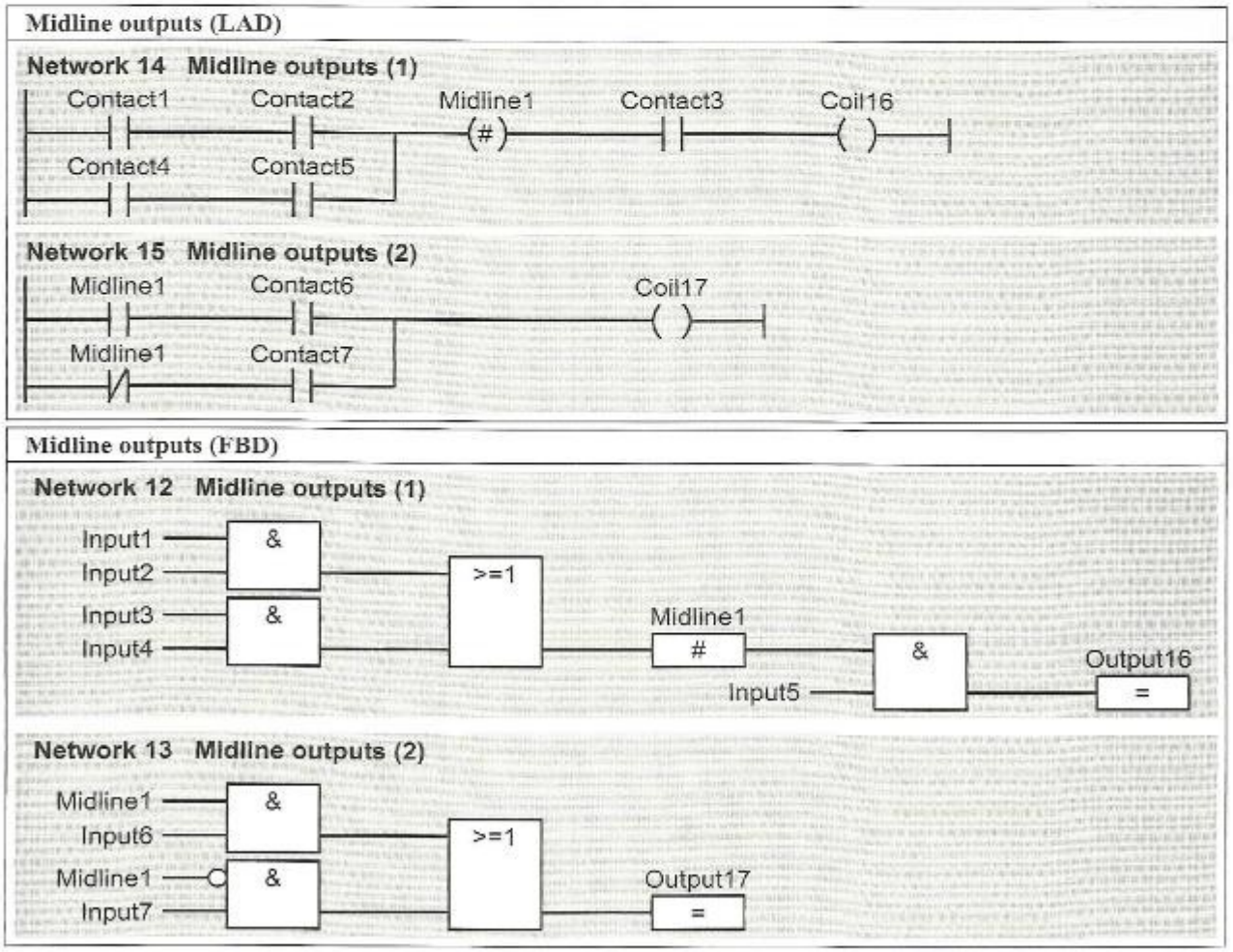


Figure 5.5 Midline Outputs

You can check the binary operand over the midline output at another point in the program. Several midline outputs may be programmed in one logic operation. A midline output box must not terminate a logic operation; the assign box is available for this purpose.

Networks 12 and 13 in Figure 5.5 illustrate how an intermediate result is stored in a midline output. The RLO from the circuit formed by *Input1*, *Input2*, *Input3* and *Input4* is stored in midline output *Midl_out1*. If the condition of the logic operation is fulfilled and if the signal state of *Input5* is "1", *Output16* is activated. The stored RLO is used in two ways in the next network. On the one hand, a check is made to see if the condition of the logic operation was fulfilled and the bit logic combination made with *Input6*, and on the other hand a check is made to see if the condition of the logic operation was not fulfilled and a bit logic combination made with *Input7*.

5.4 Edge Evaluation

5.4.1 How Edge Evaluation Works

With an edge evaluation, you detect the change in a signal state, a signal edge. An edge is positive (rising) when the signal changes from "0" to "1". The opposite is referred to as a negative (falling) edge.

In a circuit diagram, the equivalent of an edge evaluation is the pulse contact element. If this pulse contact element emits a pulse when the relay is switched on, this corresponds to the rising edge. A pulse from the pulse contact element on switching off corresponds to a falling edge.

Detection of a signal edge (change in a signal state) is implemented in the program. The CPU compares the current RLO (the result of an input check, for example) with a stored RLO. If the two signal states are different, a signal edge is present.

The stored RLO is located in an "edge memory bit" (it does not necessarily have to be a memory bit). This must be an operand whose signal state must be available when the edge evaluation is again encountered (in the next program cycle), and which is not used elsewhere in the program. Memory bits, data bits in global data blocks, and static local data bits in function blocks are all suitable as operands.

The edge memory bit stored the "old" RLO with which the CPU last processed the edge evaluation. If a signal edge is now present, that is, if the current RLO differs from the signal state of the edge memory bit, the CPU corrects the signal state of the edge memory bit by assigning it the "value" of the "new" RLO. When the edge evaluation is next processed (usually in the next program cycle), the signal state of the edge memory bit is the same as that of the current RLO (if this has not change in the meantime), and the CPU no longer detects an edge.

A detected edge is indicated by the RLO after edge evaluation. If the CPU detects a signal edge, it sets the RLO to "1" after edge evaluation (power then flows). If there is no signal edge, the RLO is "0".

Signal state "1" after an edge evaluation therefore means "edge detected". Signal state "1" is present only briefly, usually only for the length of one program cycle. Since the CPU does not detect an edge in the next cycle (if the "input RLO" of the edge evaluation does not change), it sets the RLO back to "0" after edge evaluation.

Please note the performance characteristics of the edge evaluation when the CPU is switched on. If no edge is to be detected, the RLO prior to edge evaluation must be identical to the signal state of the edge memory bit when the CPU is switched on. Under certain circumstances, the edge memory bit must be reset in the start-up routine (depending on the required performance and on the operand used).

5.4.2 Edge Evaluation in LAD

The LAD programming language provides four different elements for edge evaluation (see Figure 5.6).

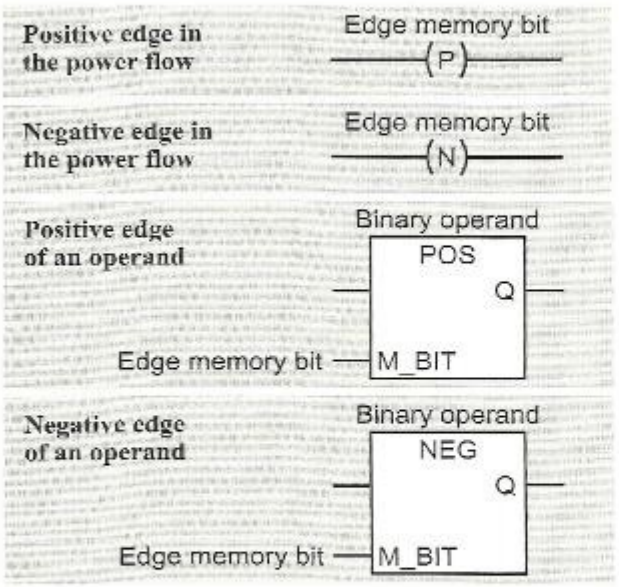


Figure 5.6 LAD elements for edge evaluation

You can process the RLO directly following an edge evaluation, that is to say, store it with a set coil, combine it with downstream contacts, or store it in a binary operand (a so-called "pulse memory bit"). You use a pulse memory bit when the RLO from the edge evaluation is to be used elsewhere in the program; it is, so to speak, the intermediate buffer for a detected edge (the pulse contact element in the circuit diagram). Operands suitable as impulse memory bit are memory bits, data bits in global data blocks, and temporary and static local data bits.

Edge evaluation in the power flow

An edge evaluation in the power flow is indicated by a coil that contains a P (for positive, rising edge) or an N (for negative, falling edge). Above the coil is an edge memory bit, a binary operand, in which the "old" RLO from the preceding edge evaluation is stored. An edge evaluation like this detects a change in the power flow from "power flowing" to "power not flowing" and vice versa.

The example in Figure 5.9 shows a positive and a negative edge evaluation in Network 16. If the parallel circuit consisting of *Contact1* and *Contact3* is fulfilled, the edge evaluation emits a brief pulse with *EmemBit1*. If *Contact2* is closed at this instant, *Memory7* is set. *Memory7* is reset again by a pulse from *EmemBit2* if the series circuit consisting of *Contact4* and *Contact5* interrupts the power flow.

You may program an edge evaluation with coil after a T-branch or in a branch that starts at the left power rail. It must not be placed directly at the left power rail.

Edge evaluation of an operand

LAD represents the edge evaluation of an operand using a box. Above the box is the operand whose signal state change is to be evaluated. The edge memory bit that stores the "old" signal state from the preceding program cycle is located at input M_BIT.

With the unlabeled input and the output Q, the edge evaluation is "inserted" in the rung instead of a contact. If power flows into the unlabeled input, output Q emits a pulse at an edge; if no power flows in this input, output Q is also always reset. You can arrange this edge evaluation in place of any contact, even in a parallel branch that does not begin at the left power rail.

Figure 5.9 shows the use of an edge evaluation of an operand in Network 17. The edge evaluation in the upper branch emits a pulse if the operand *Contact1* changes its signal state from "0" to "1" (positive edge). This pulse sets *Memory0*. The edge evaluation is always enabled by the direct connection of the unlabeled input to the left power rail. The power edge evaluation is enabled by *Contact2*. If it is enabled with "1" at this input, it emits a pulse if the binary operand *Contact3* changes its signal state from "1" to "0" (negative edge).

5.4.3 Edge Evaluation in FBD

The FBD programming language provides four different elements for edge evaluation (see Figure 5.7).

The RLO after an edge evaluation can be directly processed for example stored with a set box, combined with subsequent binary functions, or assigned to a binary operand (a so-called "pulse memory bit"). A pulse memory bit is used when the RLO from the edge evaluation is to be processed elsewhere in the program; it is, as it were, the intermediate buffer

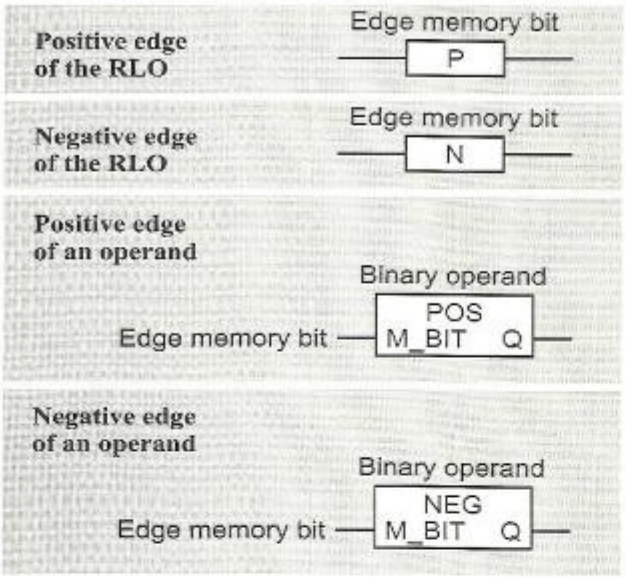


Figure 5.7 FBD elements for edge evaluation

for a detected edge. Operands suitable as pulse memory bits are memory bits, data bits in global data blocks, and temporary and static local data bits.

An edge evaluation is not permitted after a T-branch.

Edge evaluation of the RLO

An edge evaluation of the RLO is indicated by a box that contains a P (for positive, rising edge) or an N (for negative, falling edge). Above the box is the edge memory bit, a binary operand containing the "old" RLO from the previous cycle. An edge evaluation like this detects a change of the RLO within a logic circuit from RLO "1" to RLO "0" and vice versa.

The example in Network 14 of Figure 5.9 shows a positive and a negative edge evaluation. When the OR function consisting of *Input1* and *Input3* is fulfilled, the edge evaluation emits a brief pulse with *EmemBit1*. If *Input2* is "1" at this instant, *Memory1* is set.*Memory1* is reset again by a pulse from *EmemBit2* when the AND function comprising *Input4* and *Input5* is no longer fulfilled.

Edge evaluation of an operand

The edge evaluation of an operand is located at the beginning of a binary logic operation. Over the box is the operand whose signal state change is to be evaluated. The edge memory bit that holds the "old" signal state from the last program cycle is located at input M_BIT. Output Q is "1" when the CPU detects a signal state change in the operand.

Network 15 in Figure 5.9 edge evaluation of an operand. Upper edge evaluation POS emits a pulse when operand *Input1* goes from "0" to "1" (positive edge). This pulse sets *Memory2*. Lower edge evaluation NEG emits a pulse when binary operand *Input3* goes from "1" to "0" (negative edge). This pulse resets *Memory2* when operand *Input2* is "1".

5.5 Binary Scaler

A binary scaler has one input and one output. If the signal at the input of the binary scaler changes its state, for example from "0" to "1", the output also changes its signal state (Figure 5.8). This (new) signal state is retained until the next, in our example positive, signal state change. Only then does the signal state of the output change again. This means that half the input frequency appears at the output of the binary scaler.

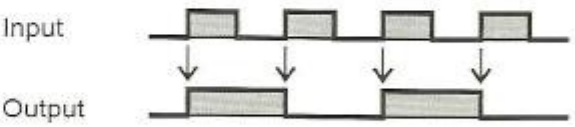


Figure 5.8 Pulse Diagram of a Binary Scaler

5.5.1 Solution in LAD

There are many different ways of solving this task, two of which are presented below.

The first solution uses memory functions (Figure 5.10, Networks 18 and 19). If the signal state of the operand *Input_1* is "1", the operand *Output_1* is set (the operand *Memory_1* is still reset). If the signal state of the operand *Input_1* changes to "0", *Memory_1* is also set (*Output_1* is now "1"). If *Input_1* is "1" the next time around, *Output_1* is reset again (*Memory_1* is

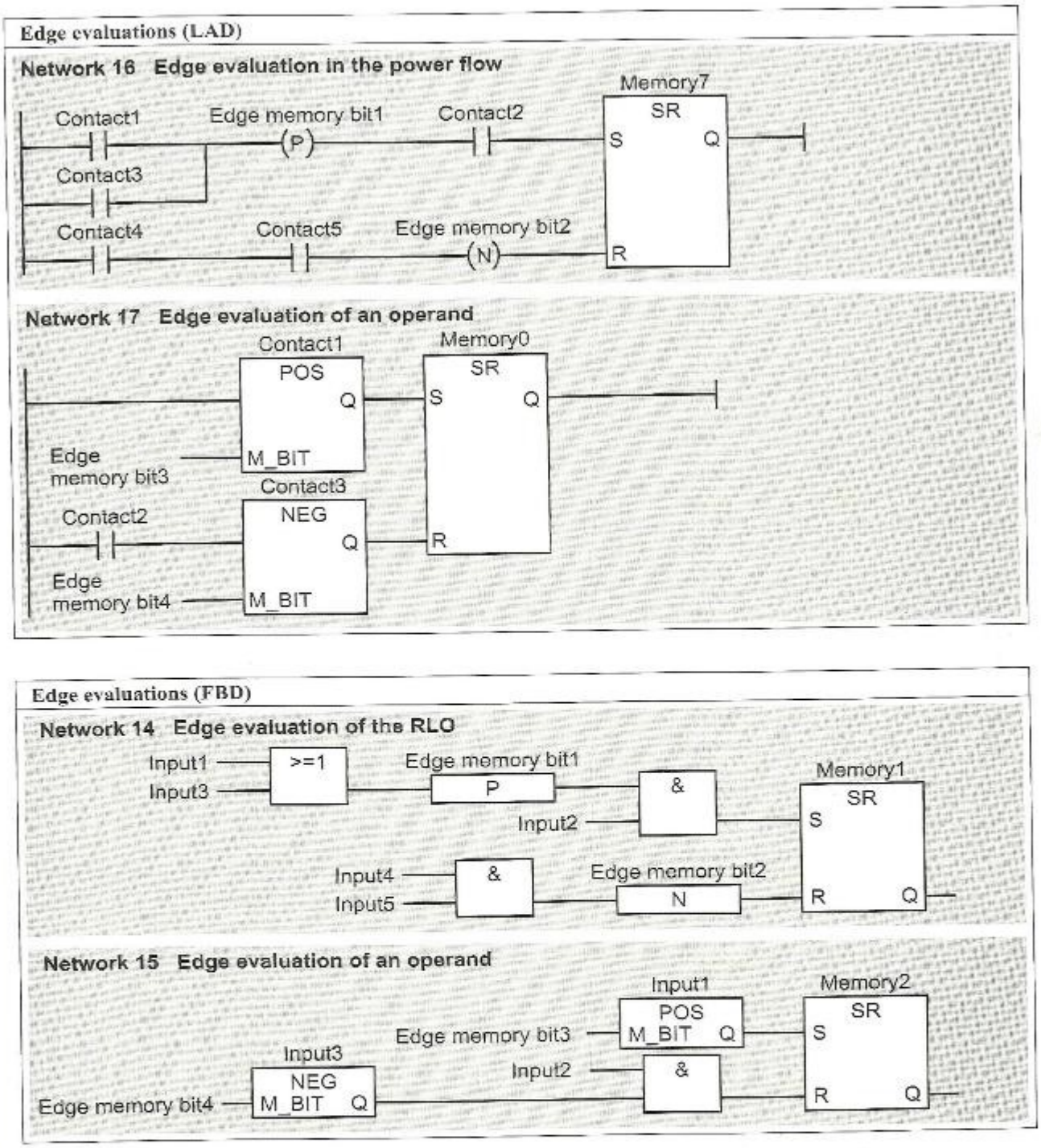


Figure 5.9 Edge Evaluations

now "1"). If *Input_1* is once again "0", *Memory_1* is reset (since *Output_1* is now also reset). Now the basic state has been reached again after two input pulses and one output pulse.

The second solution uses the latching function (Networks 20 and 21) common in circuit diagrams. The principle is the same as in the first solution except that the reset condition – as is usual with latching – is "zero active".

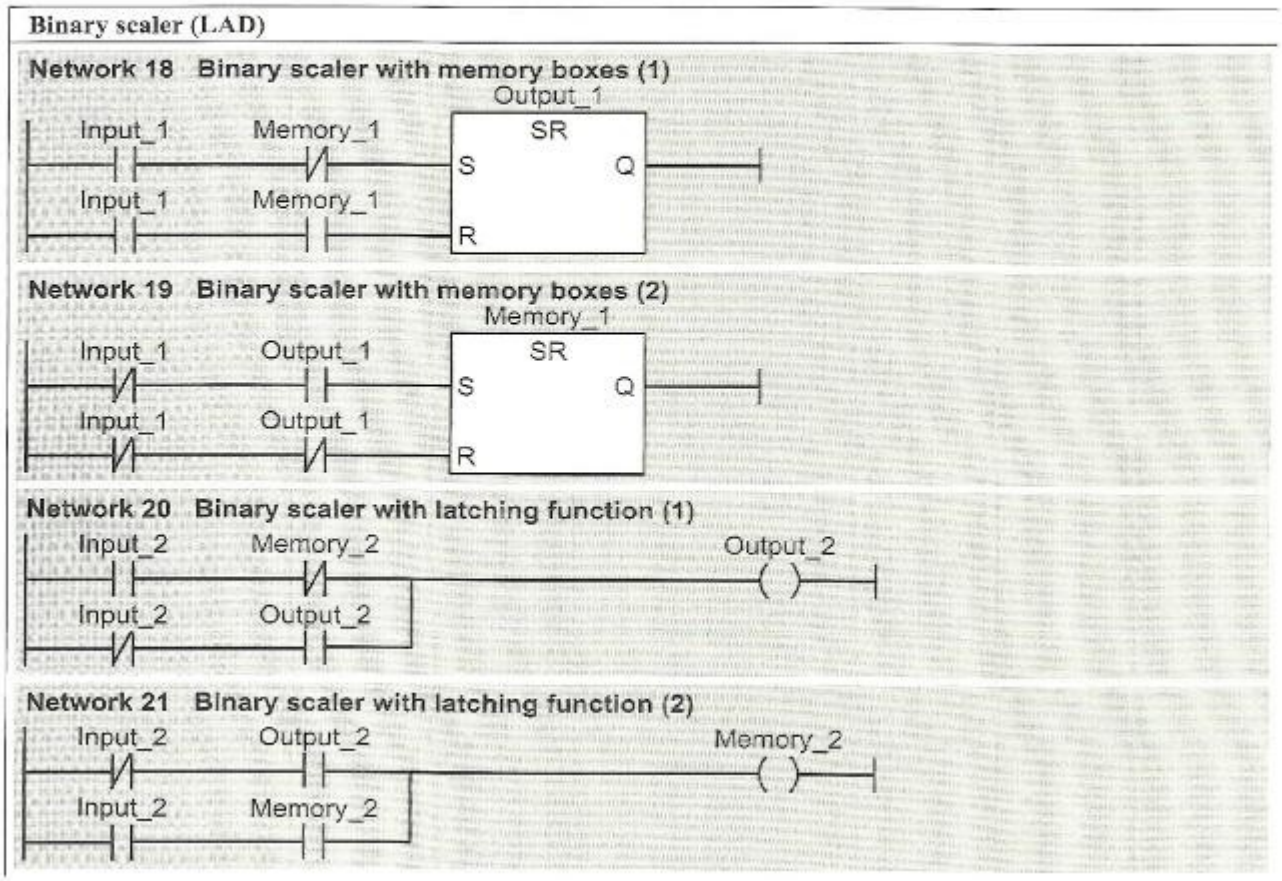


Figure 5.10 Binary Scaler Examples (LAD)

5.5.2 Solution in FBD

There are many different ways of solving this task, two of which are presented below.

The first solution uses memory functions (Figure 5.11, Networks 16 and 17). If the signal state of the operand *Input* is "1", the operand *Output* is set (the operand *Memory* is still reset). If the signal state of the operand *Input* changes to "0", *Memory* is also set (*Output* is now "1"). If *Input* is "1" the next time around, *Output* is reset again (*Memory* is now "1"). If *Input* is once again "0", *Memory* is reset (since *Output* is now also reset). Now the basic state has been reached again after two input pulses and one output pulse.

The second solution uses an edge evaluation of the operand Input (Figure 5.11, Networks 18 to 20). If no edge is detected at *Input0*, the RLO is "0" following edge evaluation and the jump instruction JCN is executed (you will find detailed descriptions of the jump operations in Chapter 16 "Jump Functions").

In our example, the jump label is called "bin" and is in Network 20. It is here that the program scan is resumed if no edge is detected. The actual binary scaler is in Network 19: If *Output0* is "0", it is set; if it is "1", it is reset. This network is processed only when an edge is detected at *Input0*. In essence, every time an edge is detected at *Input0*, *Output0* changes its signal state.

5.6 Example of a Conveyor Control System

The following example of a functionally extremely simple conveyor belt control system illustrates the use of binary logic operations and

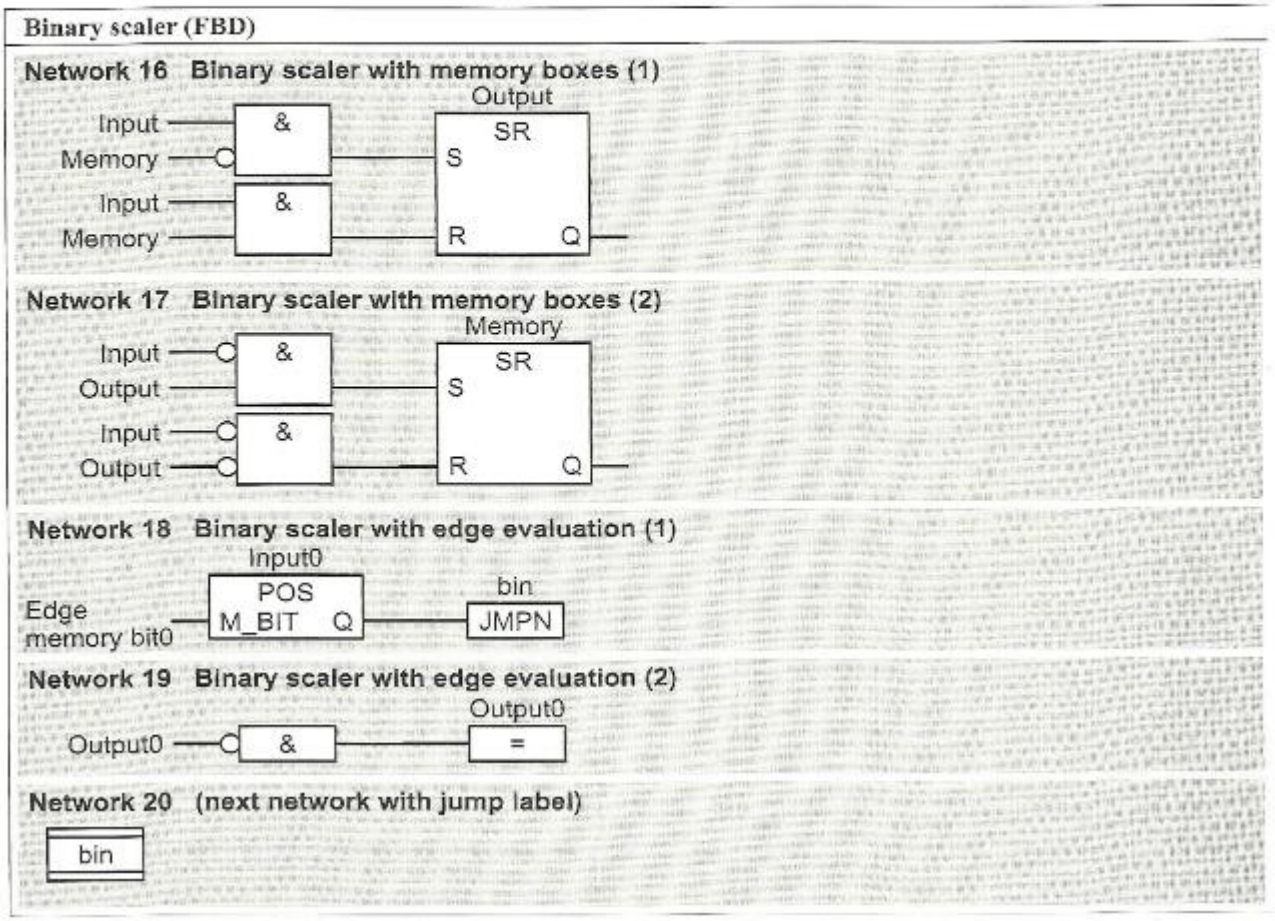


Figure 5.11 Binary Scaler Examples (FBD)

memory functions in conjunction with inputs, outputs and memory bits.

Functional description

Parts are to be transported by conveyor belt, one crate or pallet per belt. The essential functions are as follows:

- When the belt is empty, the controller requests more parts by issuing the "ready-load" signal (ready to load)
- When the "Start" signal is issued, the belt starts up and transports the parts
- At the end of the conveyor belt, an "end-of-belt" sensor (for instance a light barrier) detects the parts, at which point the belt motor switches off and triggers the "ready_rem" signal (ready to remove)
- When the "continue" signal is issued, the parts are transported further until the "end-belt" (end-of-belt) sensors no longer detects them.

The example is programmed with inputs, outputs, and memory bits, and may be programmed in any block at any location. In this case, a function without function value was chosen as block.

Signals, symbols

A few additional signals supplement the functionality of the conveyor belt control system:

- Basic_st
  Sets the controller to the basic state
- Man_on
  Switches the belt on, regardless of conditions
- /Stop
  Stops the conveyor as long as the "0" signal

is present (an NC contact as sensor, "zero active")

- Light_barrier1
  The parts have reached the end of the belt
- /Mfault1
  Fault signal from the belt motor (e.g. motor protection switch); designed as "zero active" signal so that, for example, a wire break also produces a fault signal

We want symbolic addressing, that is, the operands are given names which we then use to write the program. Before entering the program, we create a symbol table (Table 5.1) containing the inputs, outputs, memory bits, and blocks.

Program for LAD

The example is located in a function block that you call in organization block OB 1 (selected from the Program Elements Catalog "FC Blocks") for processing in a CPU.

Here, the example is programmed with memory boxes. In Chapter 19 "Block Parameters", the same example is shown using latches. The program in this Chapter can be found in a function block with block parameters which can also be called as often as needed (for several conveyor belts).

When programming, the global symbols can also be used without quotation marks provided they do not contain any special characters. If a symbol does contain a special character (such as an umlaut or a space), it must be placed in quotation marks. In the compiled block, the editor indicates all global symbols by setting them in quotation marks.

Figure 5.12 shows the program for the conveyor control system (function block FC 11) under "Conveyor Example" in the "LAD_Book" library that you can download from the publisher's Website (see page 8).

Program for FBD

The example is located in a function which you call in organization block OB 1 (from the Program Elements Catalog "FC Blocks") for processing in a CPU.

Table 5.1 Symbol Table for the Example "Conveyor Belt Control System"

Symbol	Address	Data Type	Comment
Belt_control	FC 11	FC 11	Belt control system
Basic_st	I 0.0	BOOL	Set controllers to the basic state
Man_on	I 0.1	BOOL	Switch on conveyor belt motor
/Stop	I 0.2	BOOL	Stop conveyor belt motor (zero-active)
Start	I 0.3	BOOL	Start conveyor belt
Continue	I 0.4	BOOL	Acknowledgment that parts were removed
Light_barrier1	I 1.0	BOOL	(Light barrier) sensor signal "End of belt" for belt 1
/Mfault1	I 2.0	BOOL	Motor protection switch belt 1, zero-active
Readyload	Q 4.0	BOOL	Load new parts onto belt (ready to load)
Ready_rem	Q 4.1	BOOL	Remove parts from belt (ready to remove)
Belt_mot1_on	Q 5.0	BOOL	Switch on belt motor for belt 1
Load	M 2.0	BOOL	Load parts command
Remove	M 2.1	BOOL	Remove parts command
EM_Rem_N	M 2.2	BOOL	Edge memory bit for negative edge of "remove"
EM_Rem_P	M 2.3	BOOL	Edge memory bit for positive edge of "remove"
EM_Loa_N	M 2.4	BOOL	Edge memory bit for negative edge of "load"
EM_Loa_P	M 2.5	BOOL	Edge memory bit for positive edge of "load"

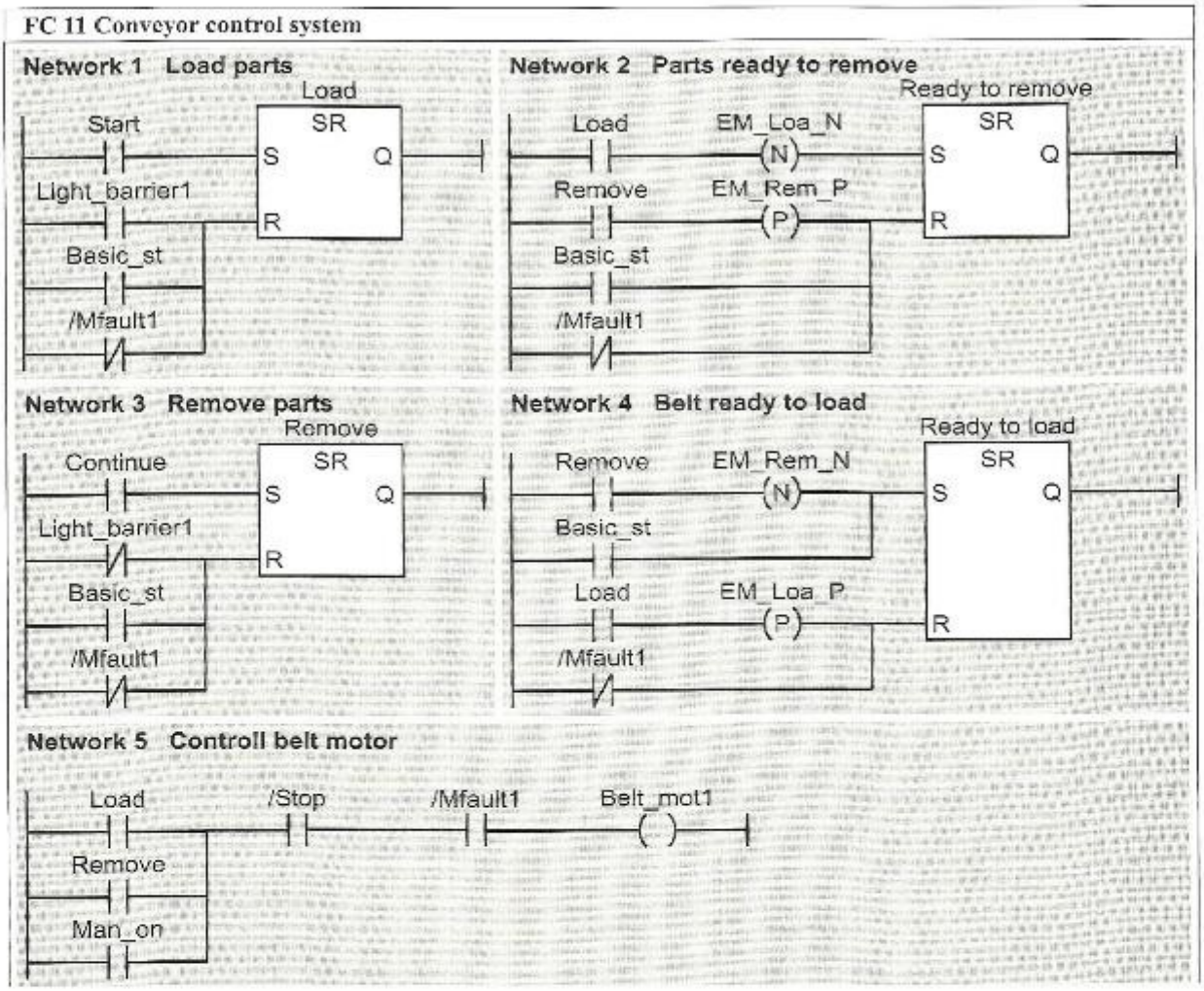


Figure 5.12 Sample Conveyor Control System (LAD)

In Chapter 19 "Block Parameters", the same example is shown using latches. The program in this Chapter can be found in a function block with block parameters which can also be called as often as needed (for several conveyor belts).

When programming, the global symbols can also be used without quotation marks provided they do not contain any special characters. If a symbol does contain a special character (such as an umlaut or a space), it must be placed in quotation marks. In the compiled block, the editor indicates all global symbols by setting them in quotation marks.

Figure 5.13 shows the circuit diagram for the conveyor control system (function block FC 11) of the "Conveyor Example" program in the "FBD_Book" library that you can download from the publisher's Website (see page 8).

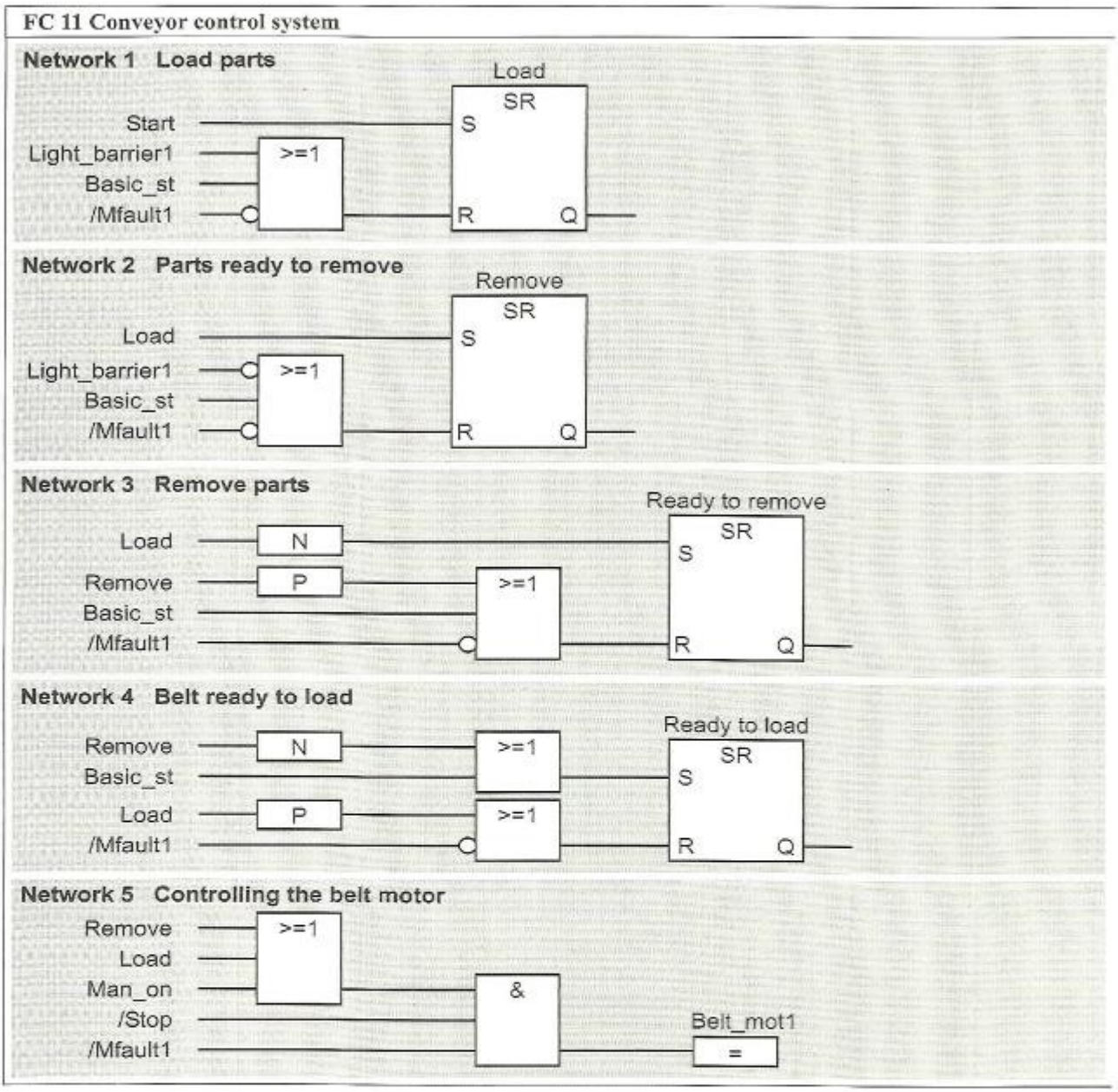


Figure 5.13 Sample Conveyor Control System (FBD)

6 Move Functions

The LAD and FBD programming languages provide the following move functions:

- MOVE box
  Copy operands and variables with elementary data types
- SFC 20 BLKMOV
  Copy data area
- SFC 21 FILL
  Fill data area
- SFC 81 UBLKMOV
  Uninterruptible copying of data area
- SFC 83 READ_DBL
  Read data area from load memory
- SFC 84 WRIT_DBL
  Write data area into load memory

The SFCs are system functions from the standard library *Standard Library* in the *System Function Blocks* program.

6.1 General

You use the move functions to copy information between the system memory, the user memory, and the user data area of the modules (Figure 6.1). Information is transferred via a CPU-internal register that functions as intermediate storage. This register is called accumulator 1. Moving information from memory to accumulator 1 is referred to as "loading" and moving from accumulator 1 to memory is called "transferring". The MOVE box contains

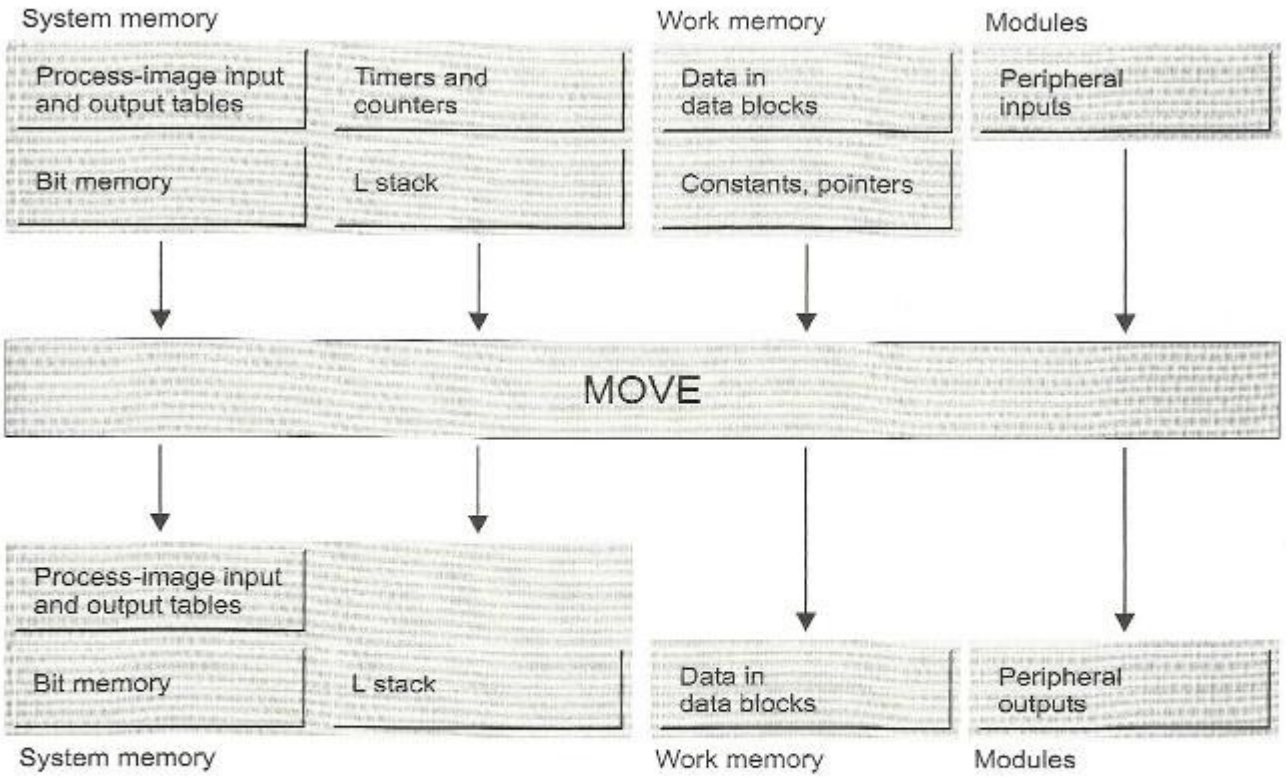


Figure 6.1 Memory Areas for Loading and Transferring

both transfer paths. It moves information at input IN to accumulator 1 (load) and immediately following this from accumulator 1 to the operand at the output (transfer).

6.2 MOVE Box

6.2.1 Processing the MOVE Box

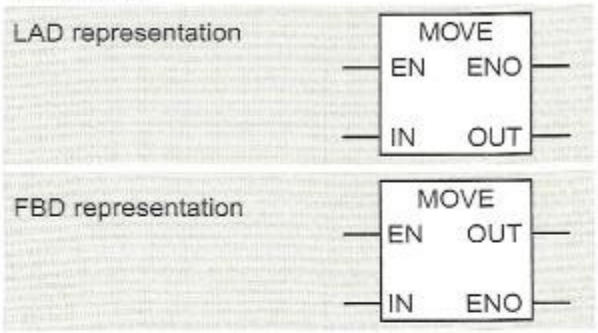
Representation

In addition to the enable input EN and the enable output ENO, the MOVE box has an input IN and an output OUT. At the input IN and the output OUT, you can apply all digital operands and digital variables of elementary data type (except BOOL). The variables at input IN and output OUT can have different data types.

The bits in the data formats are discussed in detail in Chapter 3.5 "Variables, Constants and Data Types".

For incremental programming, you will find the MOVE box in the Program Element Catalog (with VIEW → OVERVIEWS [Ctrl - K] or with INSERT → PROGRAM ELEMENTS) under "Move".

MOVE box



Different operand widths

The operand widths (byte, word, doubleword) at the input and the output of the MOVE box may vary. If the operand at the input is "less" than at the output, it is moved to the output operand right-justified and is padded at the left with zeroes. If the input operand is "greater" than the output operand, only that part of the input operand on the right that fits into the output operand is moved.

Figure 6.2 explains this. A byte or word at the input is loaded right-justified into accumulator

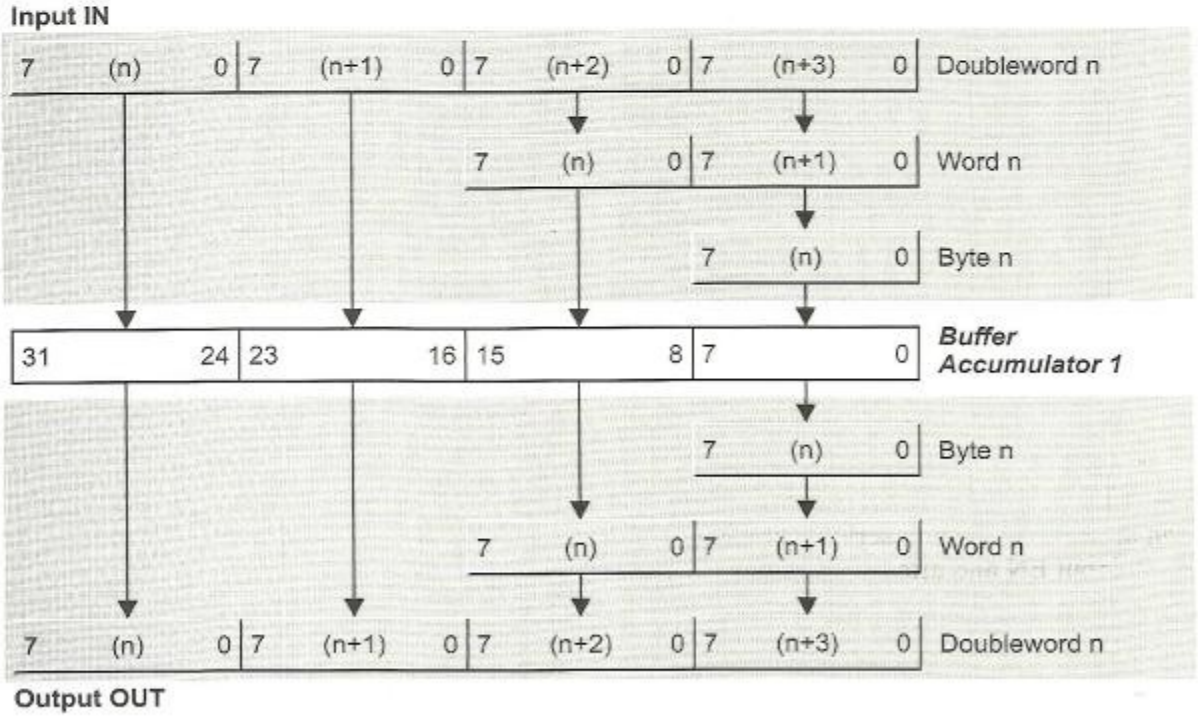


Figure 6.2 Moving Different Operand Widths

1 and the remainder is padded with zeroes. A byte or a word at output OUT is removed right-justified from accumulator 1.

Function

The MOVE box moves the information of the operand at input IN to the operand at output OUT. The MOVE box only moves information when the enable input is "1" or is unused, and when the master control relay is deenergized.

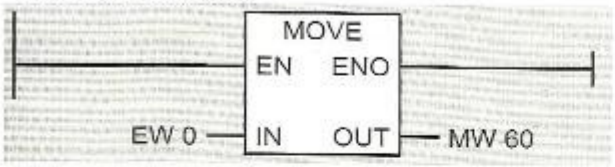
If EN = "1" and the MCR is energized, zero is written to output OUT. With "0" at the enable input, the operand at output OUT is unaffected. MOVE does not report errors.

IF EN == "1" or not wired		
THEN		ELSE
ENO := "1"		
IF MCR enabled		
THEN	ELSE	
OUT := 0	**OUT := IN**	ENO := "0"

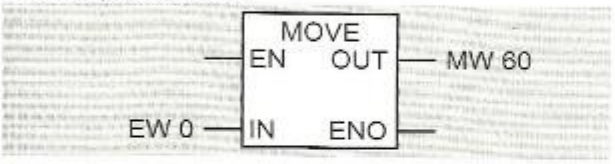
Example

The contents of input word IW 0 are moved to memory word MW 60.

LAD representation:



FBD representation:



MOVE box in a rung (LAD)

You can arrange contacts in series and in parallel before input EN and after output ENO.

The MOVE box must only be placed in a branch that leads directly to the left power rail. This branch can also have contacts before the input EN and it need not be the top branch. With the direct connection to the left power rail you can connect MOVE boxes in parallel. When connecting boxes in parallel, you require a coil to terminate the rung. If you have not provided for error evaluation, assign a "dummy" operand to the coil, for example a temporary local data bit.

You can connect MOVE boxes in series. In doing so, the ENO output of the preceding box leads to the EN input of the following box.

If you arrange several MOVE boxes in one rung (parallel at the left power rail and then continuing in series), the boxes in the uppermost branch are processed first, from left to right, and then the boxes in the parallel branch from left to right, etc.

You can find examples of the move functions in the "Basic Functions" program (FB 106) of the "LAD_Book" library that you can download from the publisher's Website (see page 8).

MOVE box in a logic circuit (FBD)

If you want to process the MOVE box in dependence on specific conditions, you can program binary logic circuits before the EN input. You can connect the ENO output with binary inputs of other functions; for instance, you can arrange MOVE boxes in series, whereby the ENO output of the preceding box leads to the EN input of the following box.

EN and ENO need not be wired.

You can find examples of the move functions in the "Basic Functions" program (FB 106) of the "LAD_Book" library that you can download from the publisher's Website (see page 8).

6.2.2 Moving Operands

In addition to the operands mentioned in this chapter, you can also move timer and counter values (see Chapter 7 "Timers" and Chapter 8 "Counters"). Chapter 18.2 "Block Functions for Data Blocks", deals with using data operands.

Moving inputs

IB n	Moving an input byte
IW n	Moving an input word
ID n	Moving an input doubleword

Loading from the process image or transferring to the process image of the inputs is also permissible for all CPUs with the S7-300 and for the newer CPUs with the S7-400 in the case of the input bytes which are not present as an input module.

Moving outputs

QB n	Moving an output byte
QW n	Moving an output word
QD n	Moving an output doubleword

Loading from the process image or transferring to the process image of the outputs is also permissible for all CPUs with the S7-300 and for the newer CPUs with the S7-400 in the case of the output bytes which are not present as an output module.

Moving from the I/O

PIB n	Loading a peripheral input byte
PIW n	Loading a peripheral input word
PID n	Loading a peripheral input doubleword
PQB n	Transferring to a peripheral input byte
PQW n	Transferring to a peripheral output word
PQD n	Transferring to a peripheral output doubleword

When moving in the I/O area, you can access different operands depending on the direction of the move. You specify I/O inputs (PIs) at the IN input of the MOVE box, and I/O outputs (PQs) at the OUT output.

When moving from the I/O to memory (loading), the input modules are accessed as peripheral inputs (PIs). Only the available modules may be addressed. Please note that direct loading from the I/O modules can move a different value than loading from the inputs of the module with the same address. While the signal states of the inputs corresponds to the values at the start of the program cycle (when the CPU updated the process image), the values loaded directly from the I/O modules are the current values.

The peripheral outputs (PQs) are used for transfers to the I/O. Only those addresses can be accessed that are also occupied by I/O modules. Transferring to I/O modules that have a process-image output table simultaneously updates that process-image output table, so there is no difference between identically addressed outputs and peripheral outputs.

Moving bit memory

MB n	Moving a memory byte
MW n	Moving a memory word
MD n	Moving a memory doubleword

Moving from and to the bit memory address area is always permissible, since the whole bit memory is in the CPU. Please note here the difference in bit memory area size on the various CPUs.

Moving temporary local data

LB n	Moving a local data byte
LW n	Moving a local data word
LD n	Moving a local data doubleword

Moving from and to the L stack is always allowed. Please note the information in Chapter 18.1.5 "Temporary Local Data".

6.2.3 Moving Constants

You may specify constant values only at the IN input of the MOVE box.

Moving constants of elementary data type

A fixed value, or constant, can be transferred to an operand. To enhance clarity, this constant can be transferred in one of several different formats. In Chapter 3.5.4 "Elementary Data Types", you will find an overview of all the different formats. All constants that can be moved using the MOVE box belong to the elementary data types. Examples:

B#16#F1	Moving a 2-digit hexadecimal number
–1000	Moving an INT number
5.0	Moving a REAL number
S5T#2s	Moving an S5 timer
TOD#8:30	Moving a time of day

Moving pointers

Pointers are a special form of constant used for calculating addresses in standard blocks. You can use the MOVE box to store these pointers in operands.

P#1.0 Moving an area-internal pointer

P#M2.1 Moving an area-crossing pointer

6.3 System Functions for Data Transfer

The following system functions are available for data transfer

- SFC 20 BLKMOV
  Copy data area
- SFC 21 FILL
  Fill data area
- SFC 81 UBLKMOV
  Uninterruptible copying of a data area
- SFC 83 READ_DBL
  Read from load memory
- SFC 84 WRIT_DBL
  Write into load memory

ANY parameter for the SFCs 20, 21 and 81

These system functions each possess two parameters of data type ANY (Table 6.1). You can connect (in principle) any operand, any variable or any absolute addressed area.

If you use a variable with combined data type, it must only be a "complete" variable; components of a variable (e.g. individual field or structure components) are not permissible. You can use the ANY pointer to define an area with absolute address (see Chapter 6.3.1 "ANY Pointer").

ANY parameter with the SFC 83 and 84

The system functions SFC 83 READ_DBL and SFC 84 WRIT_DBL transmit data between data blocks present in the load and work memories. Complete data blocks or parts of data blocks are permissible as actual block parameters SRCBLK and DSTBLK. With symbolic addressing, only "complete" variables are accepted which are present in one data block; individual field or structure components are not permissible. Use the ANY pointer to specify an absolute addressed area.

6.3.1 ANY Pointer

You require the ANY pointer when you want to specify an absolute-addressed operand area as block parameter of type ANY. The general format of the ANY pointer is as follows:

P#[DataBlock]Operand Type Quantity

Examples:

P#M16.0 BYTE 8
Area of 8 bytes beginning with MB 16

P#DB11.DBX30.0 INT 12
Area of 12 words in DB 11 beginning with DBB 30

Table 6.1 Parameters for SFC 20, 21 and 81

SFC	Parameter	Declaration	Data Type	Contents, Description
20	SRCBLK	INPUT	ANY	Source area from which data are to be copied
	RET_VAL	RETURN	INT	Error information
	DSTBLK	OUTPUT	ANY	Destination to which data are to be copied
21	BVAL	INPUT	ANY	Source area to be copied
	RET_VAL	RETURN	INT	Error information
	BLK	OUTPUT	ANY	Destination to which the source area is to be copied (including multiple copies)
81	SRCBLK	INPUT	ANY	Source area from which data are to be copied
	RET_VAL	RETURN	INT	Error information
	DSTBLK	OUTPUT	ANY	Destination to which data are to be copied

P#I18.0 WORD 1
Input word IW 18

P#I1.0 BOOL 1
Input I 1.0

Please note that the operand address in the ANY pointer must always be a bit address.

It makes sense to specify a constant ANY pointer when you want to access a data area for which you have not declared variables. In principle, you can assign variables or operands to an ANY parameter. For example, 'P#I1.0 BOOL 1' is identical to 'I 1.0' or the relevant symbolic address.

6.3.2 Copy Data Area

The system function SFC 20 BLKMOV copies the contents of a source area (parameter SLCBLK) to a destination area (parameter DSTBLK) in the direction of ascending addresses (incremental).

The following actual parameters may be assigned:

- Any variables from the operand areas for inputs (I), outputs (Q), bit memory (M), and data blocks (variables from global data blocks and from instance data blocks)
- Variables from the temporary local data (special circumstances govern the use of data type ANY)
- Absolute-addressed data areas, which require specification of an ANY pointer

You cannot use SFC 20 to copy timers or counters, to copy information from or to the modules (operand area P), or to copy system data blocks (SDBs).

In the case of inputs and outputs, the specified area is copied regardless of whether or not the addresses actually reference input or output modules. If the CPU does not possess SFC 83 READ_DBL, you can also specify a variable or an area from a data block in the load memory as the source area.

Source area and destination area may not overlap. If the source area and the destination area are of different lengths, the transfer is completed only up to the length of the shorter of the two areas.

Example (Figure 6.3, Network 4): Starting with memory byte MB 64, 16 bytes are to be copied to data block DB 124 starting from DBB 0.

6.3.3 Uninterruptible Copying of a Data Area

System function SFC 81 UBLKMOV copies the contents of a source area (parameter SRCBLK) to a destination area (parameter DSTBLK) in the direction of ascending addresses (incrementally). The copy function is uninterruptible, creating the possibility of increased response times to interrupts. A maximum of 512 bytes can be copied.

The following actual parameters may be assigned:

- Any variables from the operand areas for inputs (I), outputs (Q), bit memory (M), and data blocks (variables from global data blocks or from instance data blocks)
- Variables from the temporary local data (special circumstances govern the use of data type ANY)
- Absolute-addressed data areas, which require specification of an ANY pointer

SFC 81 cannot be used to copy timers or counters, to copy information from or to the modules (operand area P), or to copy system data blocks (SDBs) or data blocks in load memory (data blocks programmed with the keyword *Unlinked*).

In the case of inputs and outputs, the specified area is copied regardless of whether their addresses reference input or output modules.

Source area and destination area may not overlap. If the source area and the destination area are of different lengths, the transfer is completed only up to the length of the shorter of the two areas.

6.3.4 Fill Data Area

System function SFC 21 FILL copies a specified value (source area) to a memory area (destination area) as often as required to fully overwrite the destination area. The transfer is made in the direction of ascending addresses (incre-

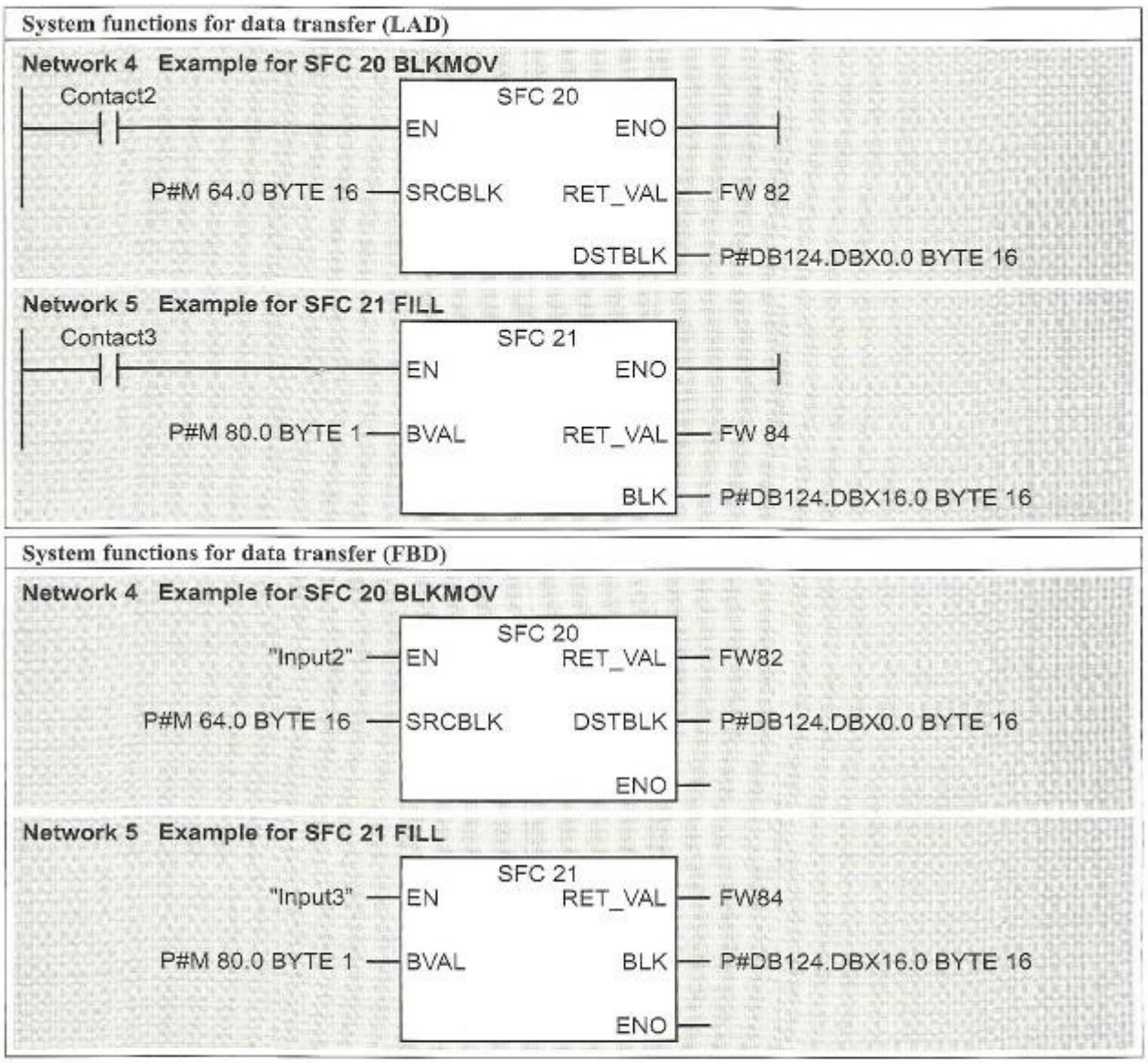


Figure 6.3 Examples for SFC 20 BLKMOV and SFC 21 FILL

mentally). The following actual parameters may be assigned:

- Any variables from the operand areas for inputs (I), outputs (Q), bit memory (M), and data blocks (variables from global data blocks and from instance data blocks)
- Absolute-addressed data areas, requiring specification of an ANY pointer
- Variables from the temporary local data of data type ANY (special circumstances apply)

SFC 21 cannot be used to copy timers or counters, to copy information from or to the modules (operand area P), or system data blocks (SDBs).

In the case of inputs and outputs, the specified area is copied regardless of whether the addresses actually reference input or output modules.

Source area and destination area may not overlap. The destination area is always fully overwritten, even when the source area is longer than the destination area or when the length of

the destination area is not an integer multiple of the length of the source area.

Example (Figure 6.3, Network 5): The contents of memory byte MB 80 is to be copied 16 times to data block DB 124, beginning DBB 16.

6.3.5 Reading from Load Memory

The system function SFC 83 READ_DBL reads data from a data block present in load memory, and writes them into a data block present in work memory. The contents of the read data block are not changed. The block parameters are described in Table 6.2.

The system function SFC 83 READ_DBL operates in asynchronous mode: you trigger the read process with signal state "1" on parameter REQ. You may only access the read and written data areas again when the BUSY parameter has returned to the signal state "0".

A data block is usually present twice in the user memory of a CPU: once in load memory and – the part relevant to processing – in work memory. If a data block has the attribute *Unlinked*, it is only present in load memory (Figure 6.4). The SFC 83 READ_DBL only reads values from load memory. The initial values of the data operands – which may differ from the actual values in work memory – are present here (see also Chapter 2.6.5 "Block Handling" under "Data blocks offline/online").

Complete data blocks, e.g. DB 100 or "Recipe 1", variables from data blocks, or an absolute addressed data area can be specified as ANY pointers, e.g. P#DB100.DBX16.0 BYTE 64, in the parameters SRCBLK and DSTBLK.

If the source area is smaller than the target area, the source area is written completely into the target area. The remaining bytes of the target area are not changed. If the source area is larger than the target area, the target area is written completely; the remaining bytes of the source area are ignored.

6.3.6 Writing into Load Memory

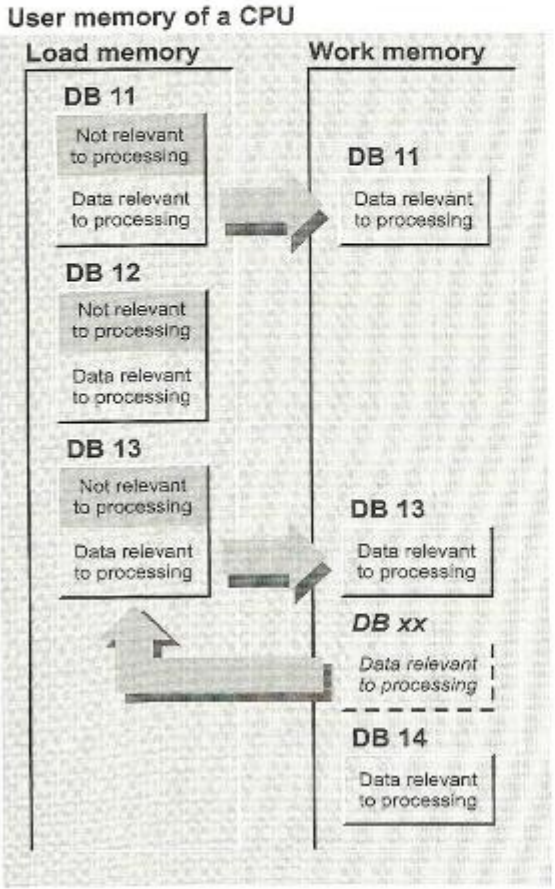
The system function SFC 84 WRIT_DBL reads data from a data block present in work memory, and writes them into a data block present in load memory. The contents of the read data block are not changed. The block parameters are described in Table 6.2.

The system function SFC 84 WRIT_DBL operates in asynchronous mode: you trigger the read process with signal state "1" on parameter REQ. You may only access the read and written data areas again when the BUSY parameter has returned to the signal state "0".

A data block is usually present twice in the user memory of a CPU: once in load memory and – the part relevant to processing – in work memory. If a data block has the attribute *Unlinked*, it is only present in load memory (Figure 6.4). The SFC 84 WRIT_DBL only reads values from work memory. The initial values of the data operands – which may differ from the actual values in load memory – are present here

Table 6.2 Parameters of the SFCs 83 and 84

SFC	Parameter	Declaration	Data Type	Contents, Description
83	REQ	INPUT	BOOL	Trigger reading with signal state "1"
	SRCBLK	INPUT	ANY	Data area in load memory to be read
	RET_VAL	RETURN	INT	Error information
	BUSY	OUTPUT	BOOL	With signal state "1": reading not yet finished
	DSTBLK	OUTPUT	ANY	Data area in work memory to be written
84	REQ	INPUT	BOOL	Trigger writing with signal state "1"
	SRCBLK	INPUT	ANY	Data area in work memory which is read
	RET_VAL	RETURN	INT	Error information
	BUSY	OUTPUT	BOOL	With signal state "1": writing not yet finished
	DSTBLK	OUTPUT	ANY	Data area in load memory to be written



A "normal" data block created using the programming device is present twice in the CPU's user memory: it is completely present in load memory, and the data "relative to processing" are present in the work memory, i.e. the data with which the program works.

A data block created using the programming device and with the attribute *Unlinked* is only present in the CPU's load memory. This data block does not occupy any space in work memory.

A data block created using the SFC 82 CREA_DBL is present in load memory and if *Unlinked* is not activated also in work memory. A data block present in work memory is the template according to which the new data block is created.

A data block created using the SFC 22 CREAT_DB or SFC 85 CREA_DB is only present in the CPU's work memory.

Figure 6.4 Data Blocks in User Memory

(see also Chapter 2.6.5 "Block Handling" under "Data blocks offline/online").

Complete data blocks, e.g. DB 200 or "Archive 1", variables from data blocks, or an absolute addressed data area can be specified as ANY pointers, e.g. P#DB200.DBX0.0 WORD 4, in the parameters SRCBLK and DSTBLK.

If the source area is smaller than the target area, the source area is written completely into the target area. The remaining bytes of the target area are not changed. If the source area is larger than the target area, the target area is written completely; the remaining bytes of the source area are ignored.

Please note: if you write into a data block in load memory (if the initial values are changed), you change the checksum of the user program.

Please also note that the load memory usually only permits a limited number of write operations for physical reasons. Too frequent writing, e.g. cyclic, limits the service life of the load memory.

7 Timers

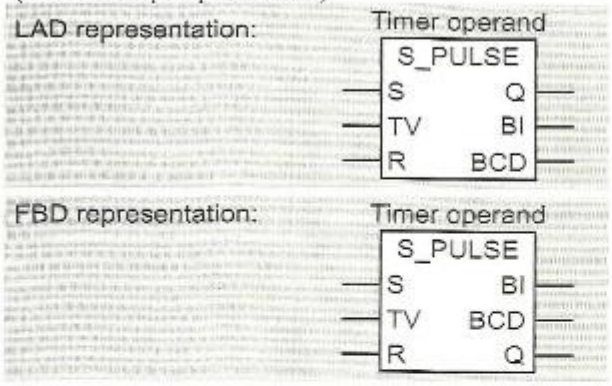
The timers allow software implementation of timing sequences such as waiting and monitoring times, the measuring of intervals, or the generating of pulses.

The following timer types are available:

- ▷ Pulse timers
- ▷ Extended pulse timers
- ▷ On-delay timers
- ▷ Retentive on-delay timers
- ▷ Off-delay timers

You can program a timer complete as box or using individual program elements. When you start a timer, you specify the type of timer you want it to be and how long it should run; you can also reset a timer. A timer is checked by querying its status ("Timer running") or the current time value, which you can fetch from the timer in either binary or BCD code.

Timer box
(in the example: pulse timer)



Name	Data Type	Description
S	BOOL	Start input
TV	S5TIME	Duration of time specification
R	BOOL	Reset input
BI	WORD	Current time value in binary
BCD	WORD	Current time value in BCD
RP	BOOL	Timer status

Figure 7.1 Timer in Box Representation

7.1 Programming a Timer

7.1.1 General Representation of a Timer

You can perform the following operations on a timer:

- ▷ Start a timer with specification of the time
- ▷ Reset a timer
- ▷ Check (binary) timer status
- ▷ Check (digital) timer value in binary
- ▷ Check (digital) time value in BCD

The box for a timer contains the coherent representation of all timer operations in the form of function inputs and function outputs (Figure 7.1). Above the box is the absolute or symbolic address of the timer. In the box, as a header, is the timer mode (S_PULSE means "Start pulse timer"). Assignments for the S and TW inputs are mandatory, while assignments for the other inputs and outputs are optional.

Individual program elements in LAD

You can also program a timer using individual program elements (Figure 7.2). The timer is then started via a coil. The timer mode is in the coil (SP = start pulse timer), and below the coil is the value, in S5TIME format, defining the duration. To reset a timer, use the reset coil, and use an NO or NC contact to check the status of the timer. Finally, you can store the current time value, in binary, in a word operand using the MOVE box.

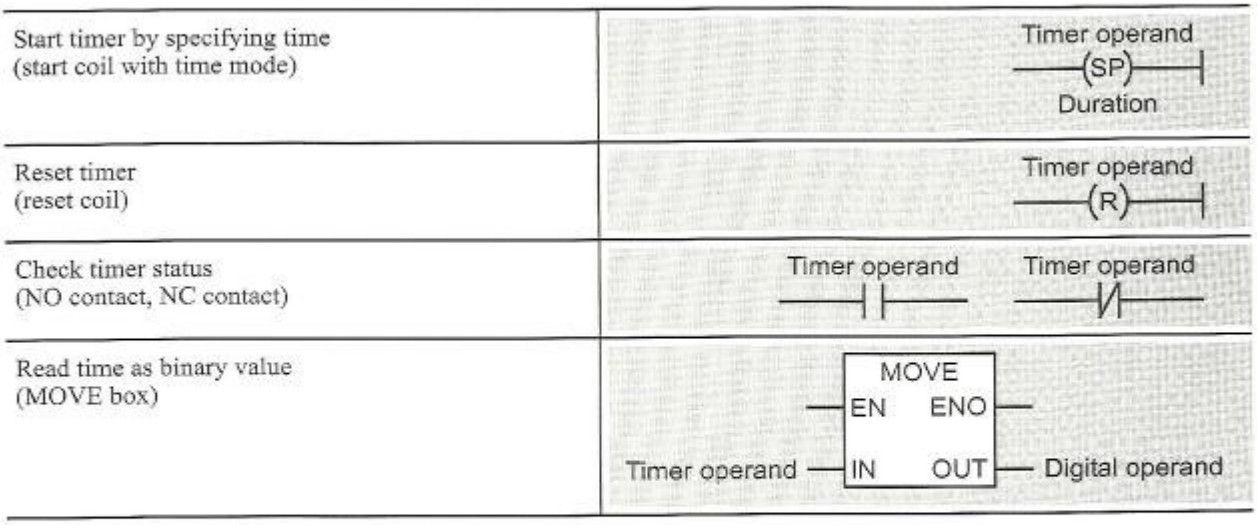


Figure 7.2 Individual Elements of a Timer (LAD)

Individual program elements in FBD

You can also program a timer using individual program elements (Figure 7.3). The timer is then started via a simple box containing the timer mode (SP = start pulse timer). Below the box is the value, in S5TIME format, defining the duration. To reset a timer, use the reset box. You can scan the status of a timer directly or in negated form with any binary input. Finally, you can store the current time value, in binary, in a word operand using the MOVE box.

For incremental programming, you will find the timers in the Program Element Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under "Timers".

7.1.2 Starting a Timer

A timer starts when the result of the logic operation (RLO) changes before the start input or before the start coil/box. Such a signal change is always required to start a timer. In the case of an off-delay timer, the RLO must change from

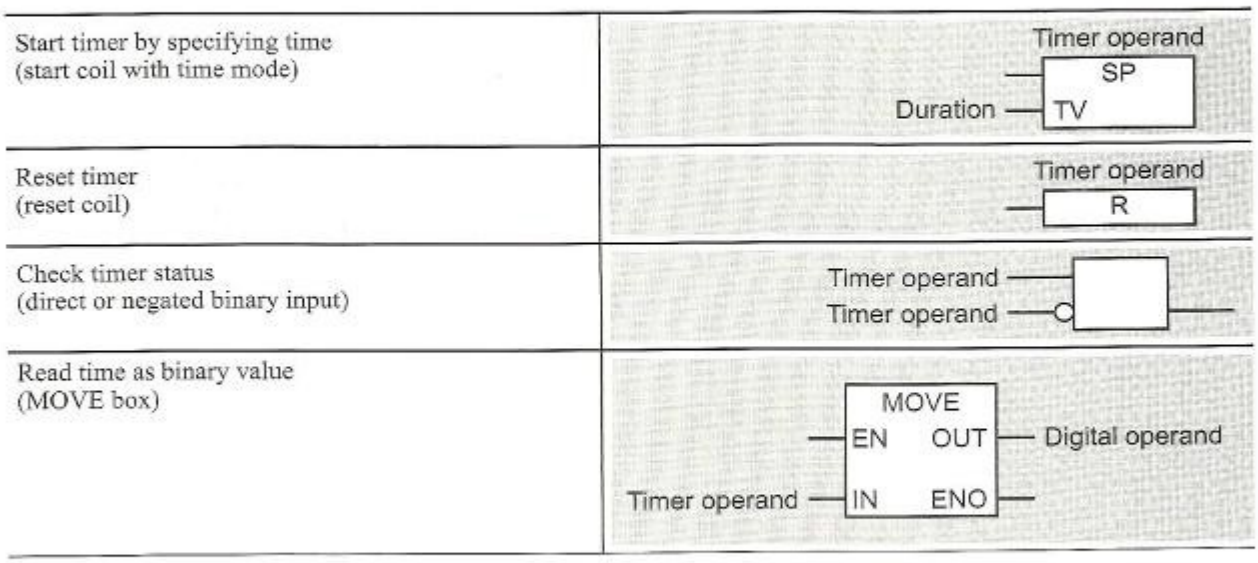


Figure 7.3 Individual Elements of a Timer (FBD)

"1" to "0"; all other timers start when the RLO goes from "0" to "1".

You can start a timer in one of five different modes (Figure 7.4). There is, however, no point in using any given timer in more than one mode.

7.1.3 Specifying the Duration of Time

The timer adopts the value below the coil/box or the value at input TV as the duration. You can specify the duration as constant, as word operand, or as variable of type S5TIME.

Specifying the duration as constant

S5TIME#10s	Duration of 10 s
S5T#1m10ms	Duration of 1 min + 10 ms

The duration is specified in hours, minutes, seconds and milliseconds. The range extends from S5TIME#10ms to S5TIME#2h46min30s (which corresponds to 9990 s). Intermediate values are rounded off to 10 ms. You can use S5TIME# or S5T# to identify a constant.

Specifying the duration as operand or variable

MW 20	Word operand containing the duration
"Time1"	Variable of data type S5TIME

The value in the word operand must correspond to data type S5TIME (see "Structure of the duration of time value", below).

Structure of the duration of time value

Internally, the duration is composed of the time value and the time base: duration = time value × time base. The duration is the time during which a timer is active ("timer running"). The time value represents the number of cycles for which the timer is to run. The time base defines the interval at which the CPU is to change the time value (Figure 7.5).

You can also build up a duration of time right in a word operand. The smaller the time base, the more accurate the actual duration. For example, if you want to implement a duration of one second, you can make one of three specifications:

Duration = 2001_{hex}	Time base 1 s
Duration = 1010_{hex}	Time base 100 ms
Duration = 0100_{hex}	Time base 10 ms

The last of these is the preferred one in this case.

When starting a timer, the CPU adopts the programmed time value. The operating system updates the timers at fixed intervals and independently of the user program, that is, it decrements the time value of all active timers as per the time base. When a timer reaches zero, it has run down. The CPU then sets the timer status (signal state "0" or "1", depending on the mode,

Use	to start the timer as	Start signal
SI S_PULSE	Puls timer	t
SV S_PEXT	Extended pulse timer	t
SE S_ODT	On-delay timer	t
SS S_ODTS	Retentive on-delay timer	t
SA S_OFFDT	Off-delay timer	t

Figure 7.4 Behavioral Characteristics of a Timer

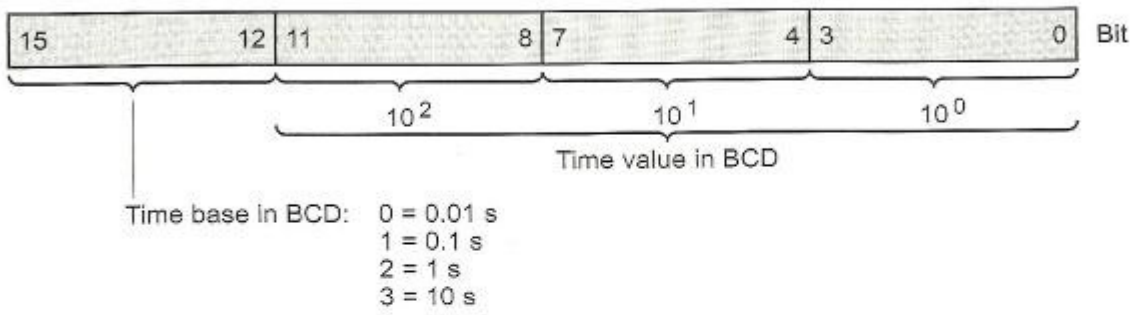


Figure 7.5 Description of the Bits in the Duration

or "type", of timer) and drops all further activities until the timer is restarted. If you specify a duration of zero (0 ms or W#16#0000) when starting a timer, the timer remains active until the CPU has processed the timer and discovered that the time has elapsed.

Timers are updated asynchronously to the program scan. As a result, it is possible that the time status at the beginning of a cycle is different than at the end of the cycle. If you use the timers at only one point in the program and in the suggested order (see below), the asynchronous updating will prevent the occurrence of malfunctions.

7.1.4 Resetting A Timer

LAD: A timer is reset when power flows in the reset input or in the reset coil (when the RLO is "1"). As long as the timer remains reset, a scan with an NO contact will return "0" and a scan with an NC contact will return "1".

FBD: A timer is reset when a "1" is present at the reset input. As long as the timer remains reset, a direct scan of the timer status will return "0" and a negated scan will return "1".

Resetting of a timer sets that timer and the time base to zero. The R input at the timer box need not be wired.

7.1.5 Checking a Timer

Checking the timer status (LAD)

The timer status is found at output Q of the timer box. You can also check the timer status with an NO contact (corresponds to output Q) or with an NC contact. The results of a check with an NO contact or with output Q differ according to the type of timer (see the description of the timer types, below). As is the case with inputs, for example, a check with an NC contact produces exactly the opposite check result as the one produced by a check with an NO contact. Output Q need not be used at the timer box.

Checking the timer status (FBD)

The timer status is available at output Q of the timer box. You can also check the timer status with a binary function input (corresponding to output Q). The results of a timer check depend on the type of timer involved (see the description of the timer types, below). Output Q need not be used at the timer box.

Checking the time value

Outputs BI and BCD provide the timer's time value in binary (BI) or binary-coded decimal (BCD). It is the value current at the time of the check (if the timer is active, the time value is counted from the set value down towards zero). The value is stored in the specified operand (transfer as with a MOVE box). You do not need to use these outputs at the timer box.

Direct checking of a time value

The time value is available in binary-coded decimal, and can be retrieved in this form from the timer. In so doing, the time base is lost and is replaced with "0". The value corresponds to a positive number in INT format. Please note: it is the time value that is checked, not the dura-

tion! You can also program direct checking of a time value with the MOVE box.

Coded checking of a time value

You can also retrieve the binary time value in "coded" form from the timer. In this case, both the time value and the time base are available in binary-coded decimal. The BCD value is structured in the same way as for the specification of a time value (see above).

7.1.6 Sequence of Timer Operations

When you program a timer, you do not need to use all the operations available for it. You need use only the operations required to execute a particular function. Normally, these are the operations for starting a timer and for checking the timer status.

In order for a timer to behave as described in this chapter, it is advisable to observe the following order when programming with individual program elements:

- ▷ Start the timer
- ▷ Reset the timer
- ▷ Check the time value or the duration
- ▷ Check the timer status

Omit unnecessary elements when programming. If you observe the order shown above and the timer is started and reset "simultaneously", the timer will start but will be immediately reset. The subsequent timer check will fail to detect the fact that the timer was started.

7.1.7 Timer Box in a Rung (LAD)

You can connect contacts in series and in parallel before the start input and the reset input as well as after output Q.

The timer box itself may be located after a T-branch and in a branch that is directly connected to the left power rail. This branch can also have contacts before the start input and it need not be the uppermost branch.

You can find further examples of the representation and arrangement of timers in the "Basic Functions" program (FB 107) of the "LAD_Book" library that you can download from the publisher's Website (see page 8).

7.1.8 Timer Box in a Logic Circuit (FBD)

You can program binary functions and memory functions before the start input and the reset input as well as after output Q.

The timer box and the individual elements for starting and resetting the timer may also be programmed after a T-branch.

You can find further examples for the representation and arrangement of timers in the "Basic Functions" program (FB 107) of the "FBD_Book" library that you can download from the publisher's Website (see page 8).

7.2 Pulse Timer

Starting a pulse timer

The diagram in Figure 7.6 describes the characteristics of a timer when it is started as pulse timer and when it is reset. The description applies if you observe the order shown in Chapter 7.1.6 "Sequence of Timer Operations" when programming with individual elements (starting before resetting before checking).

① When the signal state at the timer's start input changes from "0" to "1" (positive edge), the timer is started. It runs for the programmed duration as long as the signal state at the start input is "1". Output Q supplies signal state "1" as long as the timer runs.

With the start value as the starting point, the time value is counted down toward zero as per the time base.

② If the signal state at the timer's start input changes to "0" before the time has elapsed, the timer stops. Output Q then goes to "0". The time value shows how much longer the timer would have run had it not been prematurely interrupted.

Resetting a pulse timer

The resetting of a pulse time has a static effect, and takes priority over the starting of a timer (Figure 7.6).

③ Signal state "1" at the reset input of an active timer resets that timer. Output Q is then "0". The time value and the time base are also set to zero. If the signal state at the reset input goes from "1" to "0" while the signal state at the set input is still "1", the timer remains unaffected.

④ Signal state "1" at the reset input of an inactive timer has no effect.

⑤ If the signal state at the start input goes from "0" to "1" (positive edge) while the reset signal is still present, the timer starts but is immediately reset (shown by a line in the diagram). If the timer status check was programmed after the reset, the brief starting of the timer does not affect the check.

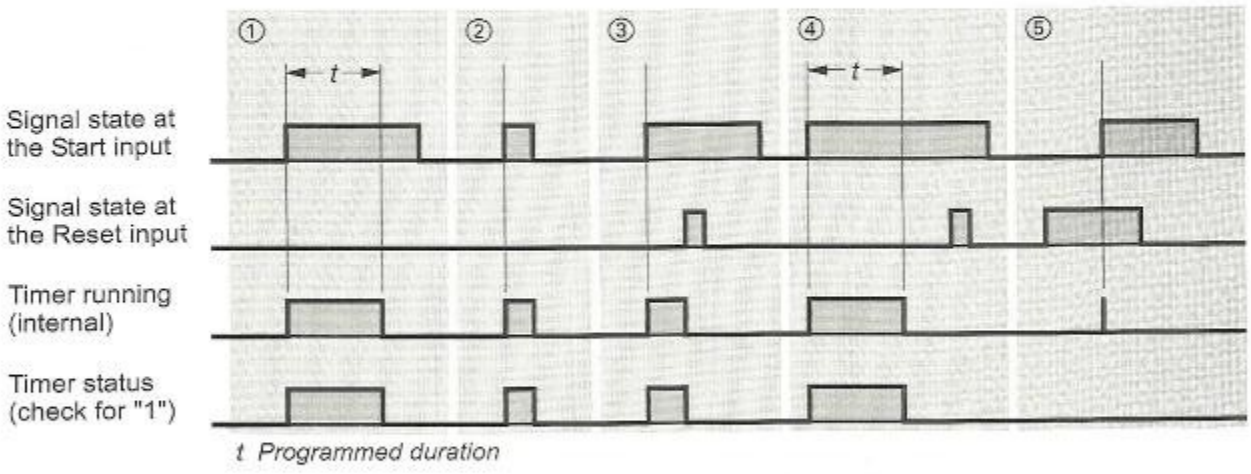


Figure 7.6 Behavioral Characteristics when Starting and Resetting a Pulse Timer

7.3 Extended Pulse Timer

Starting an extended pulse timer

The diagram in Figure 7.7 describes the behavioral characteristics of the timer after it is started and when it is reset. The description applies if you observe the order shown in Chapter 7.1.6 "Sequence of Timer Operations" when programming with individual elements (starting before resetting before checking).

❶❷ When the signal state at the timer's start input goes from "0" to "1" (positive edge), the timer is started. It runs for the programmed duration, even when the signal state at the start input changes back to "0". A check for signal state "1" (timer status) returns a check result of "1" as long as the timer is running.

With the start value as starting point, the time value is counted down towards zero as per the time base.

❸ If the signal state at the start input goes from "0" to "1" (positive edge) while the timer is running, the timer is restarted with the programmed time value (the timer is "retriggered"). It can be restarted any number of times without first elapsing.

Resetting an extended pulse timer

The resetting of an extended pulse timer has a static effect, and takes priority over the starting of a timer (Figure 7.7).

❹❺ Signal state "1" at the timer's reset input while the timer is running resets the timer. A check for signal state "1" (timer status) returns a check result of "0" for a reset timer. The time value and the time base are also reset to zero.

❻ A "1" at the reset input of an inactive timer has no effect.

❼ If the signal state at the start input goes from "0" to "1" (positive edge) while the reset signal is present, the timer is started but is immediately reset (indicated by a line in the diagram). If the timer status check is programmed after the reset, the brief starting of the timer does not affect the timer check.

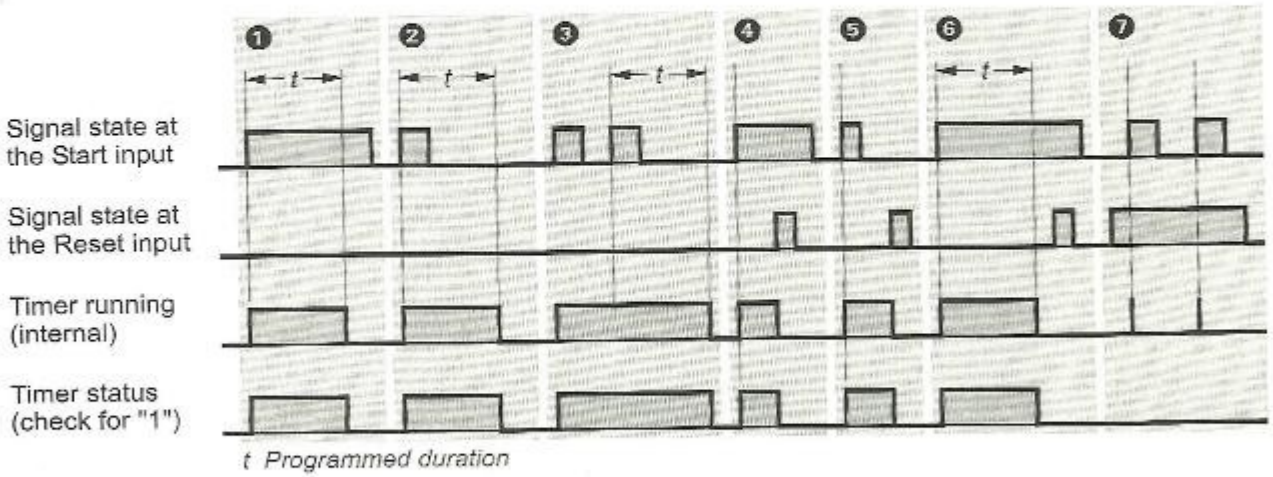


Figure 7.7 Behavioral Characteristics when Starting and Resetting an Extended Pulse Timer

7.4 On-Delay Timer

Starting an on-delay timer

The diagram in Figure 7.8 describes the behavioral characteristics of the timer after it is started and when it is reset. The description applies if you observe the order shown in Chapter 7.1.6 "Sequence of Timer Operations" when programming with individual elements (starting before resetting before checking).

① When the signal state at the timer's start input changes from "0" to "1" (positive edge), the timer is started. It runs for the programmed duration. Checks for signal state "1" return a check result of "1" when the time has duly elapsed and signal state "1" is still present at the start input (on-delay).

With the start value as starting point, the time value is counted down towards zero as per the time base.

② If the signal state at the start input changes from "1" to "0" while the timer is running, the timer stops. A check for signal state "1" (timer status) always returns a check result of "1" in such cases. The time value shows the amount of time still remaining.

Resetting an on-delay timer

The resetting of an on-delay timer has a static effect, and takes priority over the starting of the timer (Figure 7.8).

③④ Signal state "1" at the reset input resets the timer whether it is running or not. A check for signal state "1" (timer status) then returns a check result of "0", even when the timer is not running and signal state "1" is still present at the start input. Time value and time base are also set to zero.

A change in the signal state at the reset input from "1" to "0" while signal state "1" is still present at the start input has no effect on the timer.

⑤ If the signal state at the start input goes from "0" to "1" (positive edge) while the reset signal is present, the timer starts, but is immediately reset (indicated by a line in the diagram). If the timer status check is programmed after the reset, the brief starting of the timer does not affect the check.

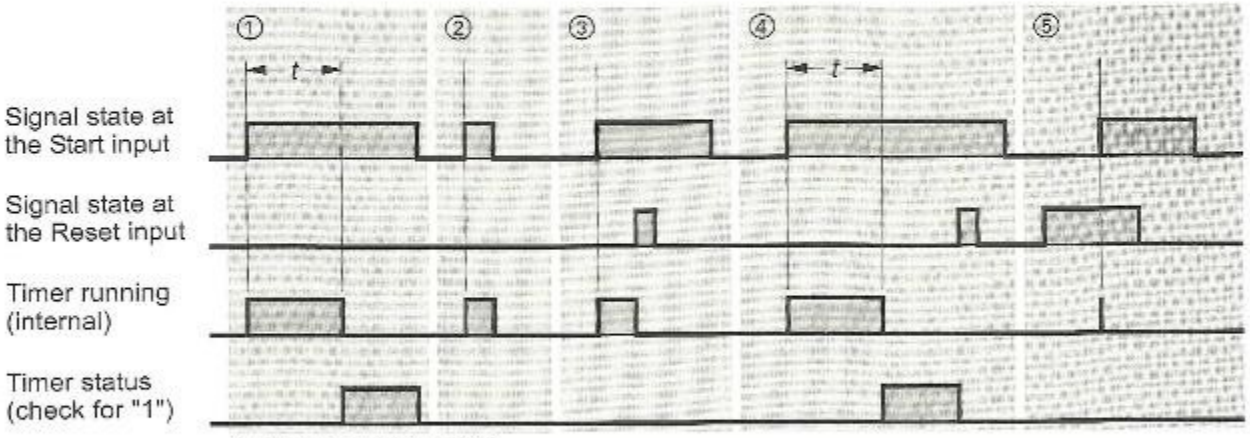


Figure 7.8 Behavioral Characteristics when Starting and Resetting an On-Delay Timer

7.5 Retentive On-Delay Timer

Starting a retentive on-delay timer

The diagram in Figure 7.9 describes the behavioral characteristics of the timer after it is started and when it is reset. The description applies if you observe the order shown in Chapter 7.1.6 "Sequence of Timer Operations" when programming with individual elements (starting before resetting before checking).

❶❷ When the signal state at the timer's start input goes from "0" to "1" (positive edge), the timer is started. It runs for the programmed duration, even when the signal state at the start input changes back to "0". When the time has elapsed, a check for signal state "1" (timer status) returns a check result of "1" regardless of the signal state at the start input. A check result of "0" is not returned until the timer has been reset, regardless of the signal state at the start input. With the start value as starting point, the time value is counted down towards zero as per the time base.

❸ If the signal state at the start input changes from "0" to "1" (positive edge) while the timer is running, the timer restarts with the programmed time value (the timer is "retriggered"). It can be restarted any number of times without first having to run down.

Resetting a retentive on-delay timer

The resetting of a retentive on-delay timer has a static effect, and takes priority over the starting of the timer (Figure 7.9).

❹❺ Signal state "1" at the reset input resets the timer, regardless of the signal state at the start input. A check for signal state "1" (timer status) then returns a check result of "0". The time value and the time base are also set to zero.

❻ If the signal state at the start input goes from "0" to "1" (positive edge) while the reset signal is present, the timer starts, but is immediately reset (indicated by a line in the diagram). If the timer status check is programmed after the reset, the brief starting of the timer has no effect on the check.

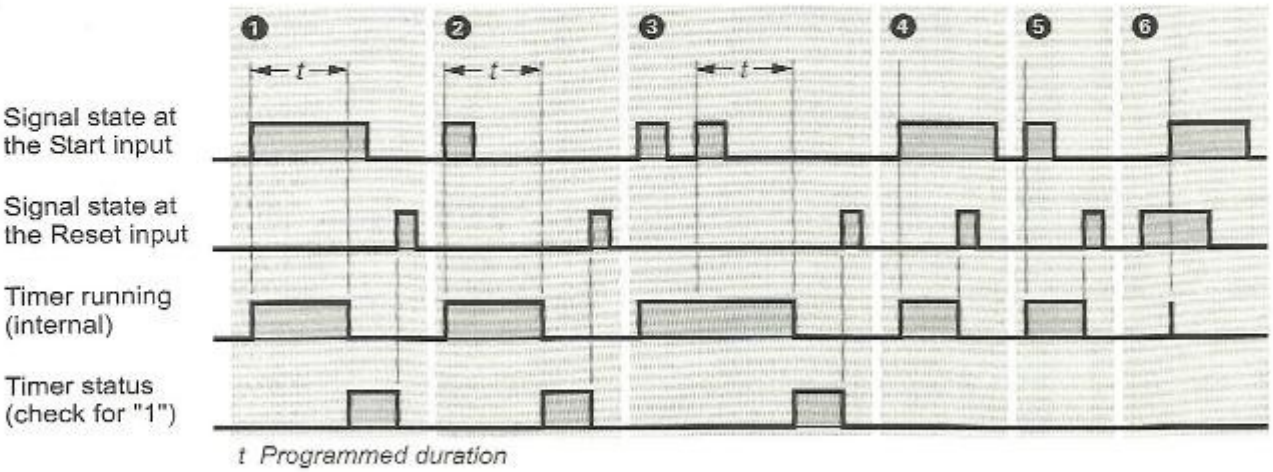


Figure 7.9 Behavioral Characteristics when Starting and Resetting an Retentive On-Delay Timer

7.6 Off-Delay Timer

Starting an off-delay timer

The diagram in Figure 7.10 describes the behavioral characteristics of the timer after it is started and when it is reset. The description applies if you observe the order shown in Chapter 7.1.6 "Sequence of Timer Operations" when programming with individual elements (starting before resetting before checking).

①③ The timer starts when the signal state at the timer's start input changes from "1" to "0" (negative edge). It runs for the programmed duration. Checks for signal state "1" (timer status) return a check result of "1" when the signal state at the start input is "1" or when the timer is running (off-delay).

With the start value as starting point, the time value is counted down towards zero as per the time base.

② If the signal state at the start input changes from "0" to "1" (positive edge) while the timer is running, the timer is reset. It is restarted only when there is a negative edge at the start input.

Resetting an off-delay timer

The resetting of an off-delay timer has a static effect, and takes priority over the starting of the timer (Figure 7.10).

④ Signal state "1" at the timer's reset input while the timer is running resets the timer. The check result of a check for signal state "1" (timer status) is then "0". Time value and time base are also set to zero.

⑤⑥ Signal state "1" at the start input and at the reset input resets the timer's binary output (a check for signal state "1" (timer status) then returns a check result of "0"). If the signal state at the reset input now changes back to "0", the timer's output once again goes to "1.

⑦ If the signal state at the start input goes from "1" to "0" (negative edge) while the reset signal is present, the timer starts, but is immediately reset (indicated by a line in the diagram). The check for signal state "1" (timer status) then returns a check result of "0".

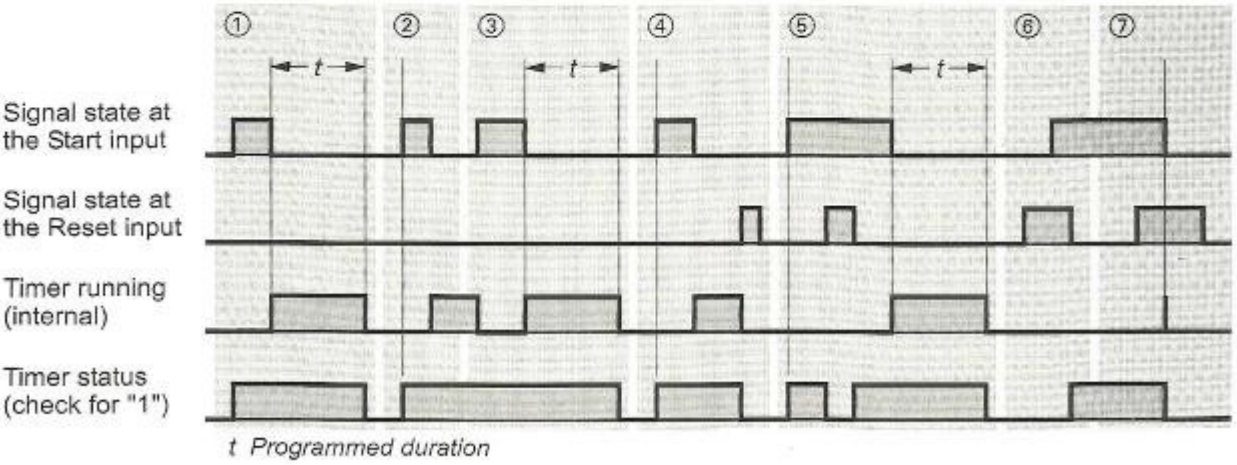


Figure 7.10 Behavioral Characteristics when Starting and Resetting an Off-Delay Timer

7.7 IEC Timers

The IEC timers are integrated in the CPU's operating system as system function blocks (SFBs).

The following timers are available on some CPUs:

- SFB 3 TP
  Pulse timer
- SFB 4 TON
  On-delay timer
- SFB 5 TOF
  Off-delay timer

Figure 7.11 shows the behavioral characteristics of these timers.

These SFBs are called with an instance data block or used as local instances in a function block. You will find the interface description for offline programming in the standard library with the name *Standard Library* under the program *System Function Blocks*.

You will find examples for the call in function block FB 107 of the "Basic Functions" program in the "LAD_Book" and "FBD_Book" libraries that you can download from the publisher's Website (see page 8).

7.7.1 Pulse Timer SFB 3 TP

IEC timer SFB 3 TP has the parameters listed in Table 7.1.

When the RLO at the timer's start input goes from "0" to "1", the timer is started. It runs for the programmed duration, regardless of any subsequent changes in the RLO at the start input. The signal state at output Q is "1" as long as the timer is running.

Table 7.1 Parameters for the IEC Timers

Name	Declaration	Data Type	Description
IN	INPUT	BOOL	Start input
PT	INPUT	TIME	Pulse length or delay duration
RP	OUTPUT	BOOL	Timer status
ET	OUTPUT	TIME	Elapsed time

Output ET supplies the duration of time for output Q. This duration begins at T#0s and ends at the set duration PT. When PT has elapsed, ET remains set to the elapsed time until input IN goes back to "0". If input IN goes to "0" before PT elapses, output ET goes to T#0s the instant PT elapses.

To reinitialize the timer, simply start it with PT = T#0s.

Timer SFB 3 TP is active in START and RUN mode. It is reset (initialized) when a cold start is executed.

7.7.2 On-Delay Timer SFB 4 TON

The IEC timer SFB 4 TON has the parameters listed in Table 7.1.

The timer starts when the RLO at its start input changes from "0" to "1". It runs for the programmed duration. Output Q shows signal state "1" when the time has elapsed and as long as the signal state at the start input remains at "1". If the RLO at the start input changes from "1" to "0" before the time has run out, the timer is reset. The next positive edge restarts the timer.

Output ET supplies the duration of time for the timer. This duration begins at T#0s and ends at set duration PT. When PT has elapsed, ET remains set to the elapsed time until input IN changes back to "0". If input IN goes to "0" before PT elapses, output ET immediately goes to T#0s.

To reinitialize the timer, simply start it with PT = T#0s.

SFB 4 TON is active in START and RUN mode. It is reset on a cold start.

7.7.3 Off-Delay Timer SFB 5 TOF

The IEC timer SFB 5 TOF has the parameters listed in Table 7.1.

The signal state at output Q is "1" when the RLO at the timer's start input changes from "0" to "1". The timer is started when the RLO at the start input changes back to "0". Output Q retains signal state "1" as long as the timer runs. Output Q is reset when the time has elapsed. If the RLO at the start input goes back to "1"

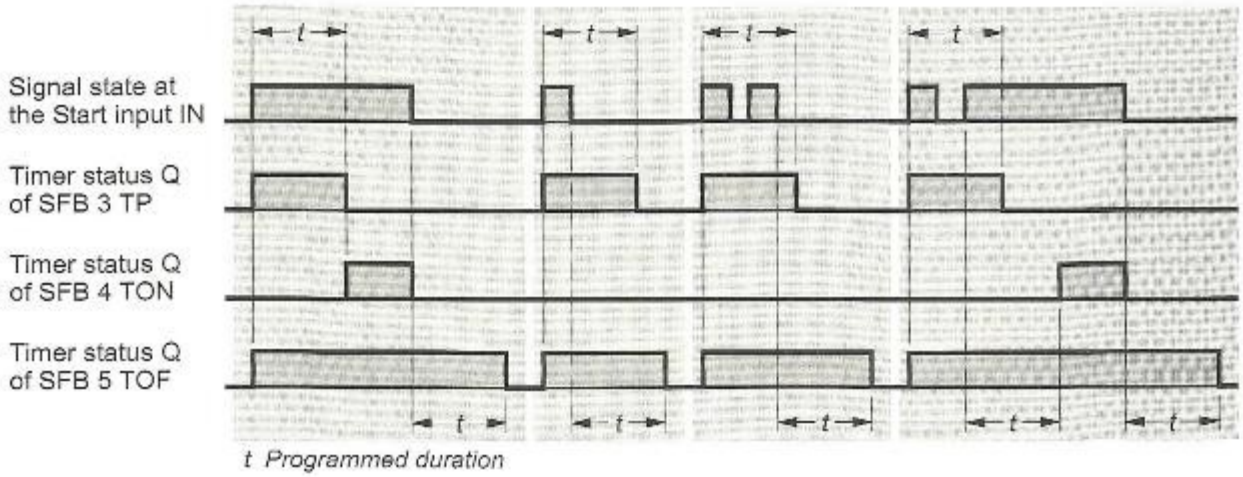


Figure 7.11 Behavioral Characteristics of the IEC Timers

before the time has elapsed, the timer is reset and output Q remains at "1".

Output ET supplies the duration of time for the timer. This duration begins at T#0s and ends at set duration PT. When PT has elapsed, ET remains set to the elapsed time until input IN changes back to "1". If input IN goes to "1" before PT has elapsed, output ET immediately goes to T#0s.

To reinitialize the timer, simply start the timer with PT = T#0s.

SFB 5 TOF is active in START and RUN mode. It is reset on a cold start.

8 Counters

Counters allow you to use the CPU to perform counting tasks. The counters can count both up and down. The counting range extends over three decades (000 to 999). The counters are located in the CPU's system memory; the number of counters is CPU-specific.

You can program a counter complete as box or using individual program elements. You can set the count to a specific initial value or reset it, and you can count up and down. The counter is scanned by checking the counter status (zero or non-zero count value) or the current count, which you can retrieve in either binary or binary-coded decimal.

8.1 Programming a Counter

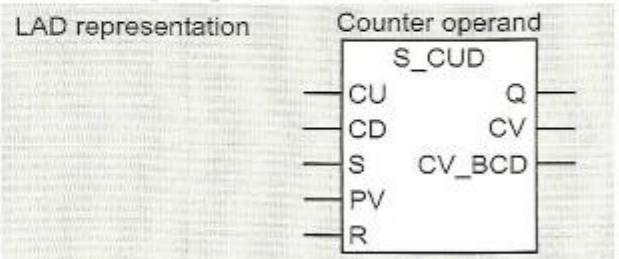
You can perform the following operations on a counter:

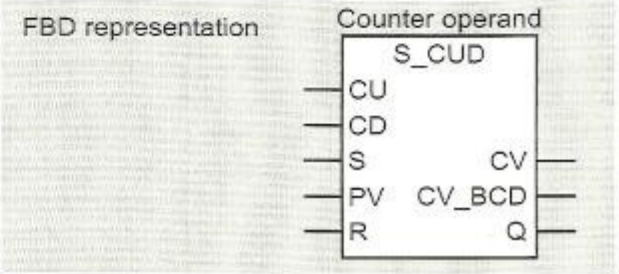
- Set counter, specifying the count value
- Count up
- Count down
- Reset counter
- Check (binary) counter status
- Check (digital) count in binary
- Check (digital) count in binary-coded decimal

Representation of a counter as box

A counter box contains the coherent representation of all counting operations in the form of function inputs and function outputs (Figure 8.1). Over the box is the absolute or symbolic address of the counter. In the box, as header, is the counter type (S_CUD stands for "up-down counter"). An assignment is mandatory for the first input (CU in the example) is mandatory; assignments for all other inputs and outputs are optional.

Counter box
(in the example: up-down counter)





Name	Data Type	Description
CU	BOOL	Up Count input
CD	BOOL	Down Count input
S	BOOL	Set input
PV	WORD	Preset value
R	BOOL	Reset input
CV	WORD	Current value in binary
CV_BCD	WORD	Current value in BCD
RP	BOOL	Counter status

Figure 8.1 Counter in Box Representation

Counter boxes are available in three versions: up-down counter (S_CUD), up counter only (S_CU), and down counter only (S_CD). The differences in functionality are explained below.

For incremental programming, you can find the counters in the Program Element Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under "Counters".

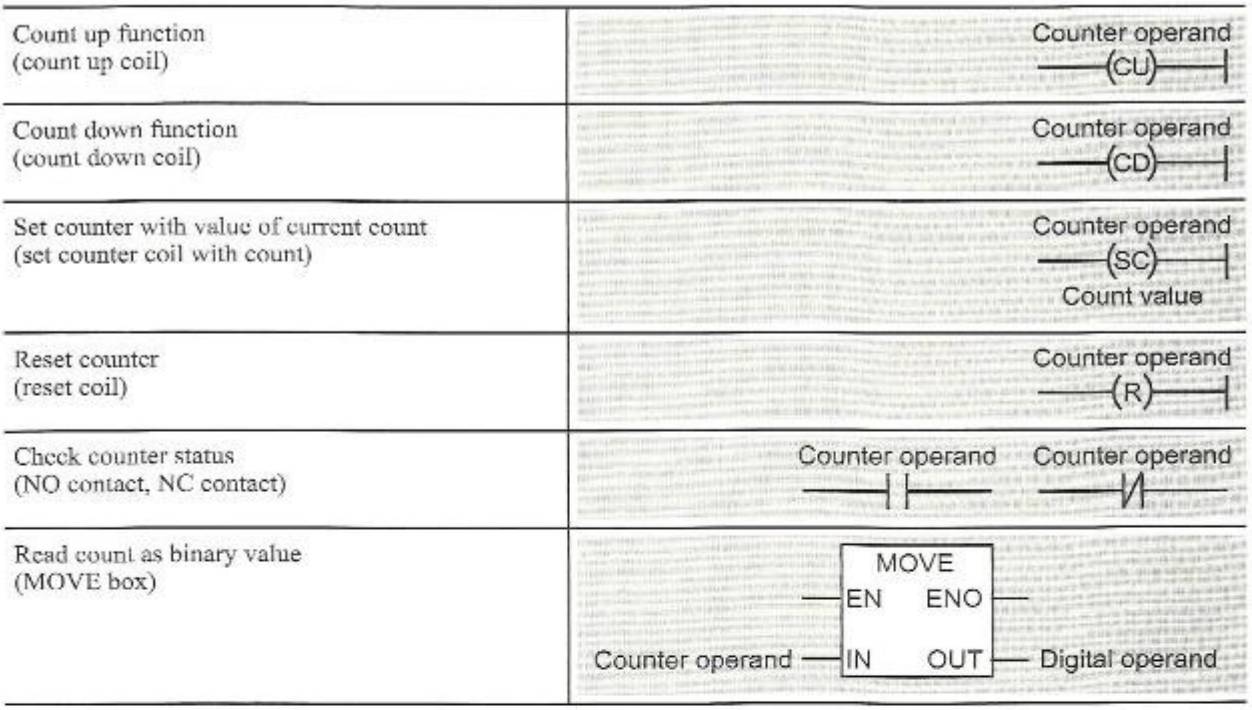


Figure 8.2 Individual Elements of a Counter (LAD)

Representation of a counter using individual elements (LAD)

You can also program a counter using individual elements (Figure 8.2).

Setting and counting are then done via coils. The set counter coil contains the counting operation (SC = Set Counter); below the coil, in WORD format, is the count value to be used to set the counter.

In the coils for counting, CU stands for count up and CD stands for count down. Use the reset coil to reset a counter and an NO or NC contact to check the status of a counter.

Finally, you can transfer the current count, in binary, with the MOVE box.

Counter box in a rung (LAD)

You can arrange contacts in series and in parallel before the counter inputs, the start input and the reset input as well as after output Q.

The counter box may be placed after a T-branch or in a branch that is directly connected to the left power rail. This branch may also have contacts before the inputs and need not be the uppermost branch.

You can find further examples of the representation and arrangement of counters in the "Basic Functions" program (FB 108) of the "LAD_Book" library that you can download from the publisher's Website (see page 8).

Representation of a counter using individual elements (FBD)

You can also program a counter using individual elements (Figure 8.3).

Setting and counting are then done via simple boxes. The set counter box contains the counting operation (SC = Set Counter); at input PV is the count value, in WORD format, is the count value to be used to set the counter.

In the boxes for counting, CU stands for count up and CD stands for count down. Use the reset box to reset a counter and a direct or negated binary function input to check the status of a counter.

Finally, you can transfer the current count, in binary, with the MOVE box.

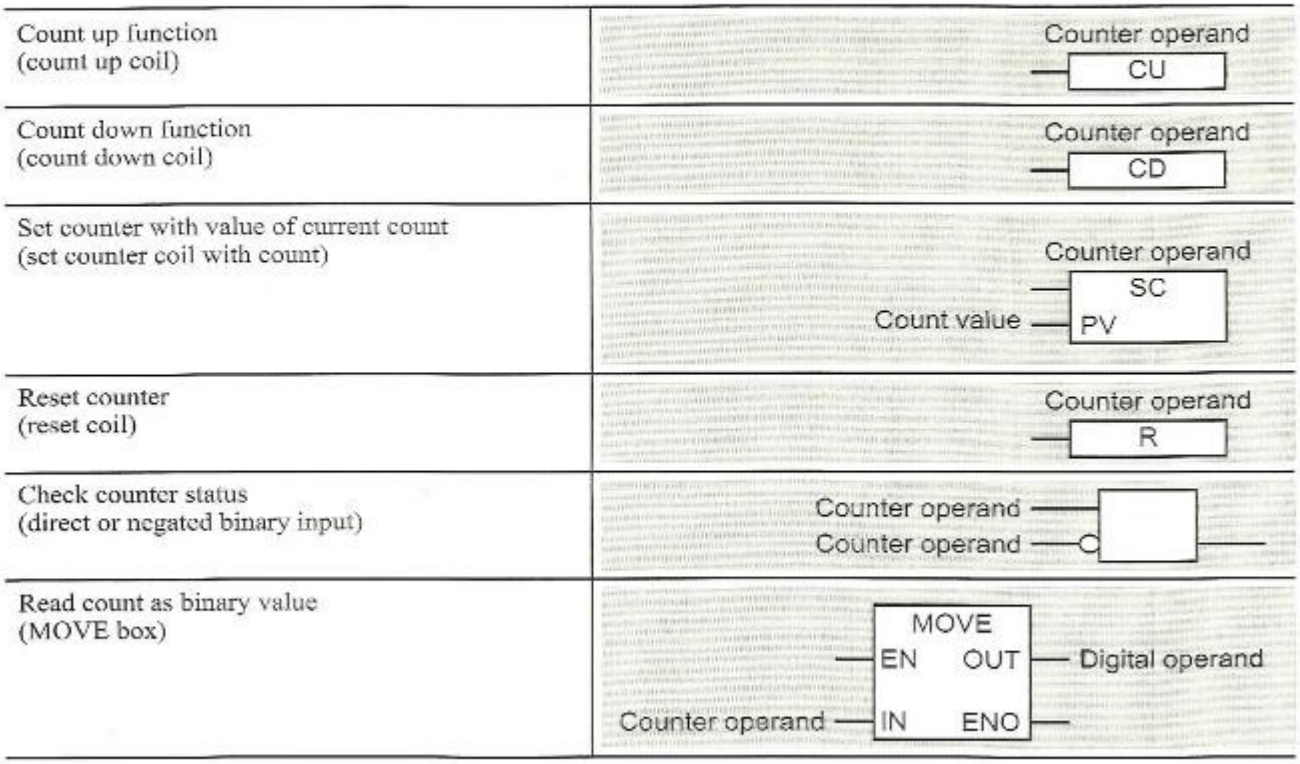


Figure 8.3 Individual Elements of a Counter (FBD)

Counter box in a logic circuit (FBD)

You can arrange binary functions and memory functions before the counter inputs, the start input and the reset input as well as after output Q.

The counter box and the individual elements for counting, setting a counter and resetting a counter may also be placed after a T-branch.

You can find further examples of the representation and arrangement of counters in the "Basic Functions" program (FB 108) of the "FBD_Book" library that you can download form the publisher's Website (see page 8).

Sequence of counting operations

When programming a counter, you do not need to use all the operations available for that counter. The operations required to carry out the desired functions are enough.

For example, to program a down counter, you need only programs the operations for setting the counter to its initial count, down counting, and checking the counter status.

In order that a counter's behavioral characteristics be as described in this chapter, it is advisable to observe the following order when programming with individual program elements:

- ▷ Count (up or down in any order)
- ▷ Set counter
- ▷ Reset counter
- ▷ Check count
- ▷ Check counter status

Omit any individual elements that are not required. If counting, setting, and resetting of the counter take place "simultaneously" when the operations are programmed in the order shown, the count will first be changed accordingly before being reset by the reset operation which follows. The subsequent check will therefore show a count of zero and counter status "0".

If counting and setting take place "simultaneously" when the operations are programmed

in the order shown, the count will first be changed accordingly before being set to the programmed count value, which it will retain for the remainder of the cycle.

The order of the operations for up and down counting is not significant.

8.2 Setting and Resetting Counters

Setting counters

A counter is set when the RLO changes from "0" to "1" before set input S or before the set coil or set box. A positive edge is always required to set a counter.

"Set counter" means that the counter is set to a starting value. The value may have a range of from 0 to 999.

Specifying the count value

When a counter is set, it assumes the value at input PV or the value below the set coil or set box as count value. You may specify the count value as constant, word operand, or variable of type WORD.

Specifying the count value as constant

C#100	Count value 100
W#16#0100	Count value 100

The count value comprises three decades in the range 000 to 999. Only positive BCD values are permissible; the counter cannot process negative values. You can use C# or W#16# to identify a constant (in conjunction with decimal digits only).

Specifying the count value as operand or variable

MW 56	Word operand containing the count value
"Count value1"	Variable of type WORD

Resetting counters (LAD)

A counter is reset when power flows in the reset input or in the reset coil (when RLO "1" is present). When this is the case, checking the counter with an NO contact will result in a check result of "0", and checking with an NC contact will return a check result of "1". Resetting a counter sets its count to "zero". The counter box's R input need not be connected.

Resetting counters (FBD)

A counter is reset when a "1" is present at the reset input. Then a direct scan of the counter status will return "0" and a negated scan will return "1". Resetting a counter sets its count to "zero". The counter box's R input need not be connected.

8.3 Counting

The counting frequency of a counter is determined by the execution time of your program! To be able to count, the CPU must detect a change in the state of the input pulse, that is, an input pulse (or a space) must be present for at least one program cycle. Thus, the longer the program execution time, the lower the counting frequency.

Up counting

A counter is counted up when the RLO changes from "0" to "1" before the up count input CU or at the up count coil or box. A positive edge is always required for up counting.

In up counting, each positive edge increments the count by one unit until it reaches the upper range limit of 999. Each additional positive edge for up counting then has no further effect. There is no carry.

Down counting

A counter is counted down when the RLO changes from "0" to "1" before down count input CD or at the down count coil or box. A positive edge is always required for down counting.

In down counting, each positive edge decrements the count by one unit until it reaches the lower range limit of 0. Each subsequent positive edge for down counting then has no further effect. There are no negative counts.

Different counter boxes

The Editor provides three different counter boxes:

S_CUD	Up/down counter
S_CU	Up counter
S_CD	Down counter

These counter boxes differ only in the type and number of counter inputs. Whereas S_CUD has inputs for both counting directions, S_CU has only the up count input and S_CD only the down count input.

You must always connect the first input of a counter box. If you do not connect the second input (S_CD) on S_CUD, this box will take on the same characteristics as S_CU.

8.4 Checking a Counter

Checking the counter status (LAD)

The counter status is at output Q of the counter box. You can also check the counter status with an NO contact (corresponding to output Q) or an NC contact.

Output Q is "1" (power flows from the output) when the current count is greater than zero. Output Q is "0" if the current count is equal to zero. Output Q does not need to be connected at the counter box.

Checking the counter status (FBD)

The counter status is at output Q of the counter box. You can also check the counter status directly (corresponds to output Q) with a binary function input, or in negated form.

Output Q is "1" when the current count is greater than zero. Output Q is "0" when the current count is equal to zero. Output Q need not be connected at the counter box.

Checking the count value (LAD and FBD)

Outputs CV and CV_BCD make the counter's current count available in binary (CV) or in binary-coded decimal (BCD). The count value made available by this operation is the one which is current at the time the check is made.

The value is stored in the specified operand (transfer as with a MOVE box). You need not switch these outputs at the counter box.

Direct checking of the count

The count is available in binary, and can be fetched from the counter in this form. The value corresponds to a positive number in INT format. Direct checking of a count can also be programmed with a MOVE box.

Coded checking of the count

You can also fetch the binary count from the counter in "coded" form. The binary-coded decimal (BCD) value is structured in the same way as for specifying the count (see above).

8.5 IEC Counters

The IEC counters are integrated in the CPU operating system as system function blocks (SFBs). The following counters are available in the appropriate CPUs:

- SFB 0 CTU
  Up counter
- SFB 1 CTD
  Down counter
- SFB 2 CTUD
  Up/down counter

You can call these SFBs with an instance data block or use them as local instances in a function block.

You will find the interface description for offline programming on standard library *Standard Library* under the *System Function Blocks* program.

You will find sample calls in function block FB 108 of the "Basic Functions" program in the "LAD_Book" and "FBD_Book" libraries that you can download from the publisher's Website (see page 8).

8.5.1 Up Counter SFB 0 CTU

IEC counter SFB 0 CTU has the parameters listed in Table 8.1. When the signal state at up count input CU changes from "0" to "1" (positive edge), the current count is incremented by 1 and shown at output CV. On the first call (with signal state "0" at reset input R), the count corresponds to the default value at input PV. When the count reaches the upper limit of 32767, it is no longer incremented, and CU has no effect.

The count value is reset to zero when the signal state at reset input R is "1". As long as input R is "1", a positive edge at CU has no effect. Output Q is "1" when the value at CV is greater than or equal to the value at PV.

SFB 0 CTU executes in START and RUN mode. It is reset on a cold start.

8.5.2 Down Counter SFB 1 CTD

IEC counter SFB 1 CTD has the parameters listed in Table 8.1. When the signal state at down count input CD goes from "0" to "1" (positive edge), the current count is decremented by 1 and shown at output CV. On the first call (with signal state "0" at load input LOAD), the count corresponds to the default value at input PV. When the current count reaches the lower limit of –32768, it is no longer decremented and CD has no effect.

The count is set to default value PV when load input LOAD is "1". As long as input LOAD is "1", a positive edge at input CD has no effect.

Output Q is "1" when the value at CV is less than or equal to zero.

SFB 1 CTD executes in START and RUN mode. It is reset on a cold start.

8.5.3 Up/down Counter SFB 2 CTUD

IEC counter SFB 2 CTUD has the parameters listed in Table 8.1.

When the signal state at up count input CU changes from "0" to "1" (positive edge), the count is incremented by 1 and shown at output CV. If the signal state at down count input CD changes from "0" to "1" (positive edge), the count is decremented by 1 and shown at output CV. If both inputs show a positive edge, the current count remains unchanged.

If the current count reaches the upper limit of 32767, it is no longer incremented when there is a positive edge at count up input CU. CU then has no further effect. If the current count reaches the lower limit of –32768, it is no longer decremented when there is a positive edge at down count input CD. CD then has no effect.

The count is set to default value PV when load input LOAD is "1". As long as load input LOAD is "1", positive signal edges at the two count inputs have no effect.

Table 8.1 Parameters for the IEC Counters

Name	Present in SFB			Declaration	Data Type	Description
CU	0	-	2	INPUT	BOOL	Up count input
CD	-	1	2	INPUT	BOOL	Down Count input
R	0	-	2	INPUT	BOOL	Reset input
LOAD	-	1	2	INPUT	BOOL	Load input
PV	0	1	2	INPUT	INT	Preset value
Q	0	1	-	OUTPUT	BOOL	Counter status
QU	-	-	2	OUTPUT	BOOL	Up counter status
QD	-	-	2	OUTPUT	BOOL	Down counter status
CV	0	1	2	OUTPUT	INT	Current count value

The count is reset to zero when reset input R is "1". As long as input R is "1", positive signal edges at the two count inputs and signal state "1" at load input LOAD have no effect.

Output QU is "1" when the value at CV is greater than or equal to the value at PV.

Output QD is "1" when the value at CV is less than or equal to zero.

SFB 2 CTUD executes in START and RUN mode. It is reset on a cold start.

8.6 Parts Counter Example

The examples illustrates the use of timers and counters. It is programmed with inputs, outputs and memory bits so that it can be programmed at any point in any block. At this point, a function without block parameters is used; the timers and counters are represented by complete boxes. You will find the same example programmed as a function block with block parameters and with individual elements in Chapter 19 "Block Parameters".

Function description

Parts are transported on a conveyor belt. A light barrier detects and counts the parts. After a set number, the counter sends the *Finished* signal. The counter is equipped with a monitoring circuit. If the signal state of the light barrier does not change within a specified amount of time, the monitor sends a signal.

The *Set* input passes the starting value (the number to be counted) to the counter. A positive edge at the light barrier decrements the counter by one unit. When a value of zero is reached, the counters sends the *Finished* signal. Prerequisite is that the parts be arranged singly (at intervals) on the belt.

The *Set* input also sets the *Active* signal. The controller monitors a signal state change at the light barrier in the active state only. When counting is finished and the last counted item has exited the light barrier, *Active* is reset.

In the active state, a positive edge at the light barrier starts the timer with the time value *Duration1* ("Dura1") as retentive pulse timer. If the timer's start input is processed with "0" in the next cycle, it still continues to run. A new positive edge "retriggers" the timer, that is,

Table 8.2 Symbol Table for the Parts Counter Example

Symbol	Address	Data Type	Comment
Counter_control	FC 12	FC 12	Counter and monitor control for parts
Acknowl	I 0.6	BOOL	Acknowledge fault
Set	I 0.7	BOOL	Set counter, activate monitor
Lbarr1	I 1.0	BOOL	"End_of_belt" sensor signal conveyor belt 1
Finished	Q 4.2	BOOL	Number of parts reached
Fault	Q 4.3	BOOL	Monitor responded
Active	M 3.0	BOOL	Counter and monitor active
EM_LB_P	M 3.1	BOOL	Edge memory bit for positive edge of light barrier
EM_LB_N	M 3.2	BOOL	Edge memory bit for negative edge of light barrier
EM_Ac_P	M 3.3	BOOL	Edge memory bit for positive edge of "Monitor active"
EM_ST_P	M 3.4	BOOL	Edge memory bit for positive edge of "Set"
Quantity	MW 4	WORD	Number of parts
Dura1	MW 6	S5TIME	Monitoring time for light barrier covered
Dura2	MW 8	S5TIME	Monitoring time for light barrier not covered
Count	C 1	COUNTER	Counter function for parts
Monitor	T 1	TIMER	Timer function for monitor

restarts it. The next positive edge to restart the timer is generated when the light barrier signals a negative edge. The timer is then started with the time value *Duration2* ("Dura2"). If the light barrier is now covered for a period of time exceeding *Dura1* or free for a period of time exceeding *Dura2*, the timer runs down and signals *Fault*. The first time it is activated, the timer is started with the time value *Dura2*.

Signals, symbols

The *Set* signal activates the counter and the monitor. The light barrier controls the counter, the *active* state, selection of the time value, and the starting (retriggering) of the monitoring time via positive and negative edges.

Evaluation of the positive and negative edge of the light barrier is required often, and temporary local data are suitable as "scratchpad" memory. Temporary local data are block-local variables; they are declared in the blocks (not in the symbol table). In the example, the pulse memory bits used for edge evaluation are stored in temporary local data. (The edge memory bits require their signal states in the next cycle as well, and must therefore not be temporary local data.)

We want symbolic addressing, that is, the operands are assigned names which are then used for programming. Before entering the program, we create a symbol table (Table 8.2) containing the inputs, outputs, memory bits, timers, counters, and blocks.

Program

The program is located in a function that you call in the CPU in organization block OB 1 (selected from the Program Elements Catalog under "FC Blocks").

During programming, the global symbols can also be used without quotation marks provided they do not contain any special characters. If a symbol contains a special character (an Umlaut or a space, for instance), it must be placed in quotation marks. In the compiled block, the Editor shows all global symbols with quotation marks

Figure 8.4 and Figure 8.5 shows the program for the parts counter (Function FC 12). You can find this program in the "Conveyor Example" program of the "LAD_Book" and "FBD_Book" libraries that you can download from the publisher's Website (see page 8).

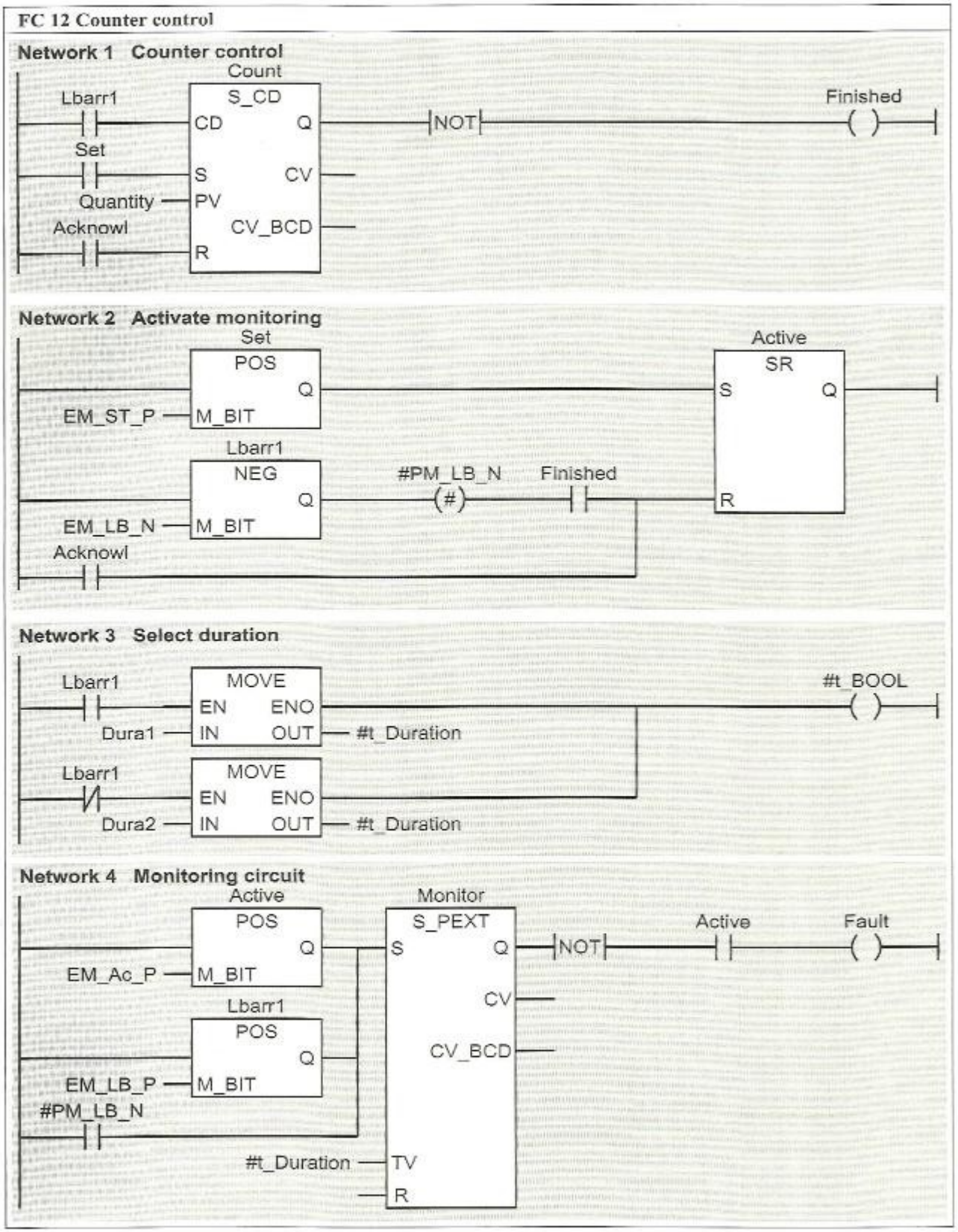


Figure 8.4 Programming Example for a Parts Counter (LAD)

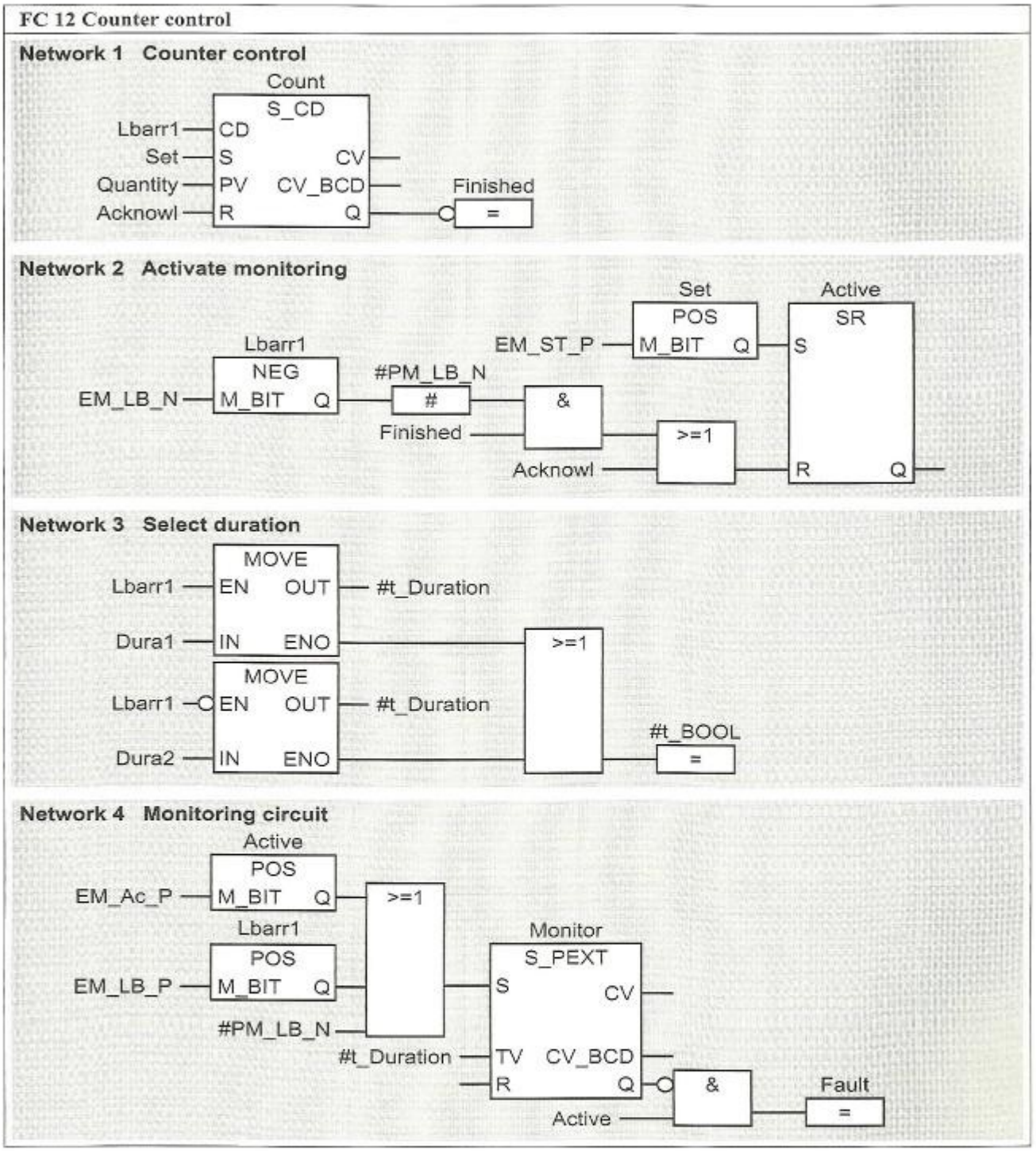


Figure 8.5 Programming Example for a Parts Counter (FBD)

Digital Functions

The digital functions process digital values predominantly with the data types INT, DINT and REAL, and thus extend the functionality of the PLC.

The **comparison functions** form a binary result from the comparison of two values. They take account of the data types INT, DINT and REAL.

You use the **arithmetic functions** to make calculations in your program. All the basic arithmetic functions in data types INT, DINT and REAL are available.

The **mathematical functions** extend the calculation possibilities beyond the basic arithmetic functions to include, for example, trigonometric functions.

Before and after performing calculations, you adapt the digital values to the desired data type using the **conversion functions**.

The **shift functions** allow justification of the contents of a variable by shifting to the right or left.

With **word logic**, you mask digital values by targeting individual bits and setting them to "1" or "0".

The digital logic operations work mainly with values stored in data blocks. These can be global data blocks or instance data blocks if static local data are used. Chapter 18.2, "Block Functions for Data Blocks", shows how to handle data blocks and gives the methods of addressing data operands.

09 Comparison Functions
Comparison for equal to, not equal to, greater than, greater than or equal to, less than, and less than or equal to

10 Arithmetic Functions
Basic arithmetic functions with data types INT, DINT and REAL

11 Mathematical Functions
Trigonometric functions; inverse trigonometric functions; squaring, square-root extraction, exponentiation, and logarithms

12 Conversion Functions
Conversion from INT/DINT to BCD and vice versa; conversion from DINT to REAL and vice versa with different forms of rounding; one's complement, negation, and absolute-value generation

13 Shift Functions
Shifting to left and right, by word and doubleword, shifting with correct sign; rotating to left and right

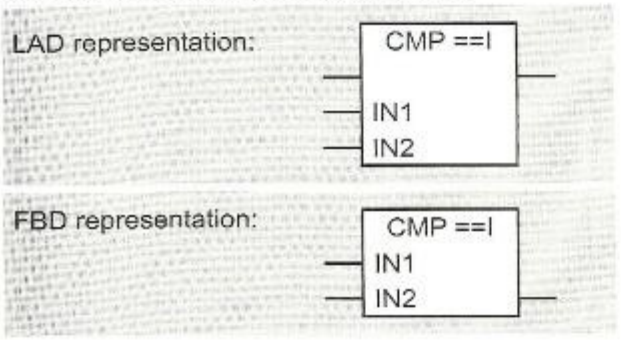
14 Word Logic
AND, OR, exclusive OR; word and doubleword combinations

9 Comparison Functions

The comparison functions compare two digital variables of data type INT, DINT or REAL for equal to, not equal to, greater than greater than or equal to, less than, or less than or equal to. The comparison result is then available as binary value (Table 9.1).

9.1 Processing a Comparison Function

Comparison box
(in example: comparison for equal to INT)



Representation LAD

In addition to the (unlabeled) binary input, the box for a comparison function has two inputs, IN1 and IN2, and an (unlabeled) binary output. The "header" in the box identifies a comparison operation (CMP for compare) and the type of comparison performed (CMP ==I, for instance, stands for the comparison of two INT numbers for equal to).

You can arrange a comparator in a rung in place of a contact. The unlabeled input and the unlabeled output establish the connection to the other (binary) program elements.

The values to be compared are at inputs IN1 and IN2 and the comparison result is the output. A successful comparison is equivalent to a closed contact ("power" flows through the comparator). If the comparison is not successful, the contact is open. The comparator's output must always be interconnected.

Representation FBD

The box for a comparison has two inputs, IN1 and IN2, and an unlabeled binary output. The "header" in the box identifies the comparison performed (CMP ==I, for example, stands for the comparison of two INT numbers for equal to).

The values to be compared are at inputs IN1 and IN2 and the result of the comparison is at the output. If the comparison is successful, the comparator output shows signal state "1"; otherwise, it is "0". It must always be interconnected.

Table 9.1 Overview of the Comparison Functions

Comparison Function	Comparison According to Data Type		
	INT	DINT	REAL
Comparison for equal to	CMP ==I	CMP ==D	CMP ==R
Comparison for not equal to	CMP <>I	CMP <>D	CMP <>R
Comparison for greater than	CMP >I	CMP >D	CMP >R
Comparison for greater than or equal to	CMP >=I	CMP >=D	CMP >=R
Comparison for less than	CMP <I	CMP <D	CMP <R
Comparison for less than or equal to	CMP <=I	CMP <=D	CMP <=R

Data types

The data type of the inputs in a comparison function depends on that function. For example, the inputs are of type REAL in the comparison function CMP >R (compare REAL numbers for greater than). Variables must be of the same data type as the inputs. When using operands with absolute addresses, the operand widths must accord with the data types. For example, you can use a word operand for data type INT.

You can find the bit assignments for the data formats in Chapter 3.5.4, "Elementary Data Types".

A comparison between REAL numbers is not true if one or both REAL numbers are invalid. In addition, status bits OS and OV are set. You can find out how the comparison functions set the remaining status bits in Chapter 15, "Status Bits".

Examples

Figure 9.1 provides an example for each of the data types. A comparison function carries out a comparison according to the characteristics specified even when no data types are declared when using operands with absolute addresses.

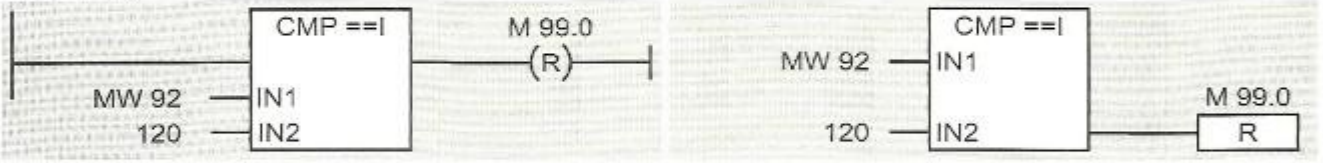
In the case of incremental programming, you will find the comparison functions in the Program Elements Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under "Comparator".

Comparison function in a rung (LAD)

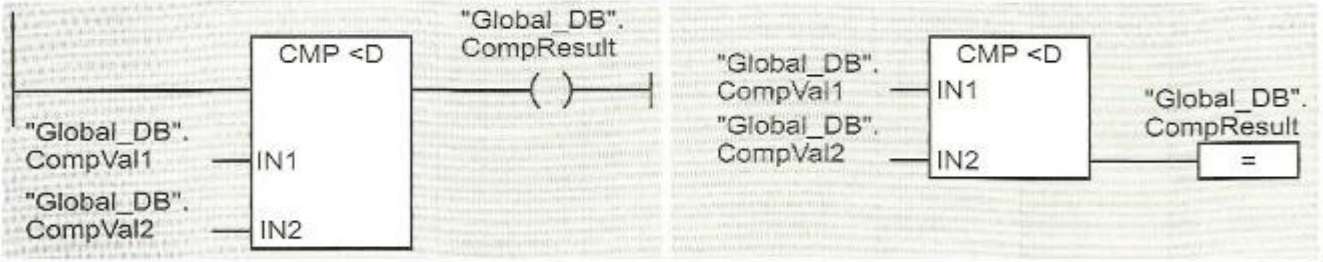
You can use the comparison function in a rung in place of a contact.

You can connect contacts before and after the comparison function in series and in parallel. The comparison boxes themselves can also be

Comparison according to INT Memory bit M 99.0 is reset if the value in memory word MW 92 is equals to 120; otherwise it is not.



Comparison according to DINT The variable "CompResult" in data block "Global_DB" is set if variable "CompVal1" is less than "CompVal2", otherwise it is reset.



Comparison according to REAL If the variable #ActVal is greater than or equal to the variable #Calibra, #Recali is set; otherwise it is not.

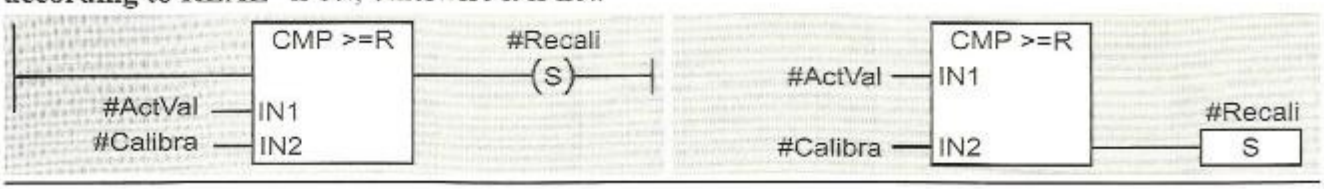


Figure 9.1 Comparison Function Examples

connected in series or in parallel. In the case of comparison functions connected in series, both comparisons must be successful for power to flow in the rung. In the case of comparators connected in parallel, only one compare condition need be fulfilled for power to flow in the parallel circuit.

You can find further examples of the representation and arrangement of comparison functions in the program "Digital Functions" (function block FB 109) in the library "LAD_Book" that you can download from the publisher's Website (see page 8).

Comparison function in a logic circuit (FBD)

You can position the comparison function at any binary input of a program element. The result of the comparison can be subsequently combined with binary functions.

You can find further examples of the representation and arrangement of comparison functions in the program "Digital Functions" (function block FB 109) in the library "FBD_Book" that you can download from the publisher's Website (see page 8).

9.2 Description of the Comparison Functions

Comparison for equal to

The "comparison for equal to" interprets the contents of the input variables in accordance with the data type specified in the comparison function and checks to see if the two values are equal. The compare condition is fulfilled ("power" flows through the comparator output or the RLO is "1") when the two variables have the same value.

If, in the case of a REAL comparison, one or both input variables are invalid, the comparison is not successful. Status bits OV and OS are also set.

Comparison for not equal to

The "comparison for not equal to" interprets the contents of the input variables in accordance with the data type specified in the comparison function and checks to see if the two values differ. The comparison is successful ("power" flows through the comparator output or the RLO is "1") when the two variables have different values.

If, in the case of a REAL comparison, one or both input variables are invalid, the comparison is not successful. In addition, status bits OV and OS are set.

Comparison for greater than

The "comparison for greater than" interprets the contents of the input variables in accordance with the data type specified in the comparison function and checks to see if the value at input IN1 is greater than the value at IN2. If this is the case, the comparison is successful ("power" flows through the comparator output or the RLO is "1").

If, in the case of a REAL comparison, one or both input variables are invalid, the comparison is not successful. In addition, status bits OV and OS are set.

Comparison for greater than or equal to

The "comparison for greater than or equal to" interprets the contents of the input variables in accordance with the data type specified in the comparison function and checks to see if the value at input IN1 is greater than or equal to the value at input IN2. If this is the case, the comparison is successful ("power" flows at the comparator output or the RLO is "1").

If, in the case of a REAL comparison, one or both input variables are invalid, the comparison is not successful. In addition, status bits OV and OS are set.

Comparison for less than

The "comparison for less than" interprets the contents of the input variables in accordance with the data type specified in the comparison function and checks to see if the value at input IN1 is less than the value at input IN2. If this is the case, the comparison is successful ("power" flows at the comparator output or the RLO is "1").

If, in the case of a REAL comparison, one or both input variables are invalid, the comparison is not successful. In addition, status bits OV and OS are set.

Comparison for less than or equal to

The "comparison for less than or equal to" interprets the contents of the input variables in accordance with the data type specified in the comparison function and checks to see if the value at input IN1 is less than or equal to the value at input IN2. If this is the case, the comparison is successful ("power" flows at the comparator output or the RLO is "1").

If, in the case of a REAL comparison, one or both input variables are invalid, the comparison is not successful. In addition, status bits OV and OS are set.

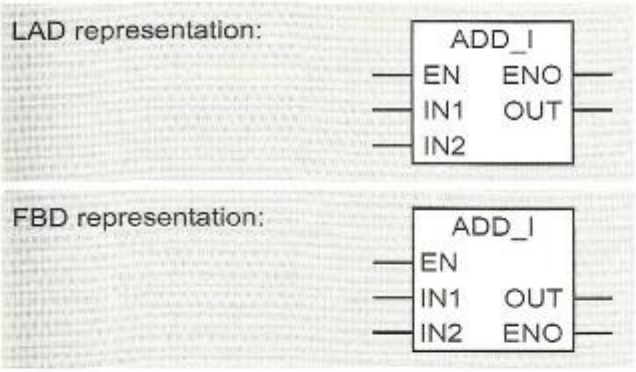
10 Arithmetic Functions

The arithmetic functions combine two values in accordance with the basic arithmetical operations of addition, subtraction, multiplication, and division. You can use the arithmetic functions on variables of type INT, DINT, and REAL (Table 10.1).

10.1 Processing an Arithmetic Function

Representation

Arithmetic box
(in example: addition with INT)



In addition to enable the input EN and the enable output ENO, a box for an arithmetic function has two inputs, IN1 and IN2, and an output, OUT. The "header" in the box identifies the arithmetic function executed (ADD_1, for instance, stands for the addition of INT numbers).

The values to be combined are at inputs IN1 and IN2, and the result of the calculation is at output OUT. The inputs and the output have different data types, depending on the arithmetic function. For example, in the case of the arithmetic function ADD_R (addition of REAL numbers), the inputs and the output are of data type REAL. The variables applied must be of the same data type as the inputs or the output. If you use absolute addresses for the operands, the operand widths must be matched to the data types. For example, you can use a word operand for data type INT.

You can find a description of the individual bits in each data format in Chapter 3.5.4, "Elementary Data Types".

Function

The arithmetic function is executed if "1" is present at the enable input ("power" flows in input EN). If an error occurs during the calculation, the enable output is set to "0"; otherwise, it is set to "1". If execution of the function is not enabled (EN = "0"), the calculation does not take place, and ENO is also "0".

Table 10.1 Overview of Arithmetic Functions

Arithmetic function	With data type		
	INT	DINT	REAL
Addition	ADD_I	ADD_DI	ADD_R
Subtraction	SUB_I	SUB_DI	SUB_R
Multiplication	MUL_I	MUL_DI	MUL_R
Division with quotient as result	DIV_I	DIV_DI	DIV_R
Division with remainder as result	-	MOD_DI	-

IF EN == "1" or not wired		
THEN		ELSE
OUT := IN1 Cfct IN2		
IF error occurred		
THEN	ELSE	
ENO := "0"	ENO := "1"	ENO := "0"

with Cfct as calculation function

If the Master Control Relay (MCR) is activated, output OUT is set to zero when the arithmetic function is processed (EN = "1"). The MCR does not affect the ENO output.

The following errors can occur during execution of an arithmetic function:

- Range violation (overflow) in INT and DINT calculations
- Underflow and overflow in a REAL calculation
- Invalid REAL number in a REAL calculation

See Chapter 15, "Status Bits", to find out how the arithmetic functions set the various status bits.

Examples

Figure 10.1 shows an example for each data type. An arithmetic function executes a calculation in accordance with the characteristic specified, even if no data types have been declared when using operands with absolute addresses.

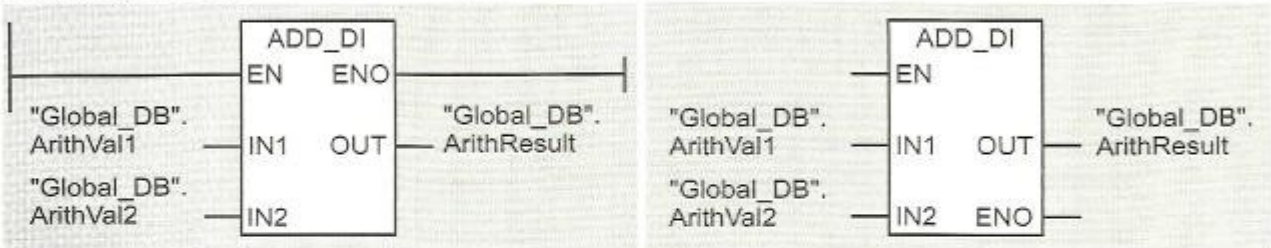
In the case of incremental programming, you will find the arithmetic functions in the Program Element Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under "Integer function" (INT and DINT calculations) and under "Floating-Point fct." (REAL calculations).

Addition according to INT

The value in memory word MW 100 is divided by 250; the integer result is stored in memory word MW 102.

Addition according to DINT

The values in variables "ArithVal1" and "ArithVal2" are added and the result stored in variable "ArithResult". All variables are stored in the data block.



Addition according to REAL

The variable #ActVal and \Factor are multiplied; the product is transferred to variable #Display.

Figure 10.1 Examples of Arithmetic Functions

Arithmetic function in a rung (LAD)

You can connect contacts in series and in parallel before the EN input and after the ENO output.

The arithmetic box itself may be placed after a T-branch and in a branch that leads directly to the left power rail. This branch can also have contacts before the EN input and it need not be the uppermost branch.

Direct connection to the left power rail means that you can connect arithmetic boxes in parallel. When you connect boxes in parallel, you need a coil to terminate the rung. If you have not provided any error evaluation, assign a "dummy" operand to the coil, for example a temporary local data bit.

You can connect arithmetic boxes in series. If the ENO output of the preceding box leads to the EN input of the subsequent box, the subsequent box is processed only if the preceding box has been completed without errors. If you want to use the result from the preceding box as input value for the next box, variables from the temporary local data area make convenient intermediate buffers.

If you arrange several arithmetic boxes in one rung (parallel to the left power rail, then further in series), the boxes in the uppermost branch are processed from left to right, followed by the boxes in the second branch from left to right, and so on.

You can find further examples of the representation and arrangement of arithmetic functions in the program "Digital Functions" (FB 110) in the library "LAD_Book" that you can download from the publisher's Website (see page 8).

Arithmetic function in a logic circuit (FBD)

If you want to process the arithmetic box in dependence on specific conditions, you can arrange binary logic operations before the EN input. You can interconnect the ENO output with binary inputs of other functions; for example, you can arrange arithmetic boxes in series, whereby the ENO output of the preceding box leads to the EN input of the following box. If you want to use the calculation result from the preceding box as input value for a subsequent box, variables from the temporary local data area make convenient intermediate buffers.

EN and ENO need not be wired.

You can find further examples of the representation and arrangement of arithmetic functions in the program "Digital Functions" (FB 110) in the library "FBD_Book" that you can download from the publisher's Website (see page 8).

10.2 Calculating with Data Type INT

INT addition

The function ADD_I interprets the values at inputs IN1 and IN2 as numbers of data type INT. It adds the two numbers and stores the sum in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the sum is negative, zero, or positive. Status bits OV and OS indicate any range violations.

INT subtraction

The function SUB_I interprets the values at inputs IN1 and IN2 as numbers of data type INT. It subtracts the value at IN2 from the value at IN1 and stores the difference in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the difference is negative, zero, or positive. Status bits OV and OS indicate any range violations.

INT multiplication

The function MUL_I interprets the values at inputs IN1 and IN2 as numbers of data type INT. It multiplies the two numbers and stores the product in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the product is negative, zero, or positive. Status bits OV and OS indicate any INT range violations.

INT division

The function DIV_I interprets the values at inputs IN1 and IN2 as numbers of data type INT. It divides the values at input IN1 (dividend) by the value at input IN2 (divisor) and supplies the quotient at output OUT. It is the

integer result of the division. The quotient is zero if the dividend is equal to zero and the divisor is not equal to zero or if the absolute value of the dividend is less than the absolute value of the divisor. The quotient is negative if the divisor is negative.

After execution of the calculation, status bits CC0 and CC1 indicate whether the quotient is negative, zero, or positive. Status bits OV and OS indicate any range violations. Division by zero produces zero as quotient and sets status bits CC0, CC1, OV and OS to "1".

10.3 Calculating with Data Type DINT

DINT addition

The function ADD_DI interprets the values at inputs IN1 and IN2 as numbers of data type DINT. It adds the two numbers and stores the sum in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the sum is negative, zero, or positive. Status bits OV and OS indicate any range violations.

DINT subtraction

The function SUB_DI interprets the values at inputs IN1 and IN2 as numbers of data type DINT. It subtracts the value at input IN2 from the value at input IN1 and stores the difference in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the difference is negative, zero, or positive. Status bits OV and OS indicate any range violations.

DINT multiplication

The function MUL_DI interprets the values at inputs IN1 and IN2 as numbers of data type DINT. It multiplies the two numbers and stores the product in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the product is negative, zero, or positive. Status bits OV and OS indicate any range violations.

DINT division with quotient as result

The function DIV_DI interprets the values at inputs IN1 and IN2 as numbers of data type DINT. It divides the value at input IN1 (dividend) by the value at input IN2 (divisor) and stores the quotient in output OUT. It is the integer result of the division. The quotient is zero if the dividend is equal to zero and the divisor is not equal to zero or if the absolute value of the dividend is less than the absolute value of the divisor. The quotient is negative if the divisor is negative.

After execution of the calculation, status bits CC0 and CC1 indicate whether the quotient is negative, zero, or positive. Status bits OV and OS indicate any range violations. Division by zero produces zero as quotient and sets status bits CC0, CC1, OV and OS to "1".

DINT division with remainder as result

The function MOD_DI interprets the values at inputs IN1 and IN2 as numbers of data type DINT. It divides the value at input IN1 (dividend) by the value at input IN2 (divisor) and stores the remainder of the division in output OUT. The remainder is what is left over from the division; it does not correspond to the decimal places. If the dividend is negative, the remainder is also negative.

After execution of the calculation, status bits CC0 and CC1 indicate whether the remainder is negative, zero, or positive. Status bits OV and OS indicate any range violations. Division by zero produces zero as remainder and sets status bits CC0, CC1, OV and OS to "1".

10.4 Calculating with Data Type REAL

REAL numbers are represented internally as floating-point numbers with two number ranges: One range with full accuracy ("normalized" floating-point numbers) and one range with limited accuracy ("denormalized" floating-point numbers; also see Chapter 3.5.4, "Elementary Data Types"). S7-400 CPUs calculate in both ranges, S7-300 CPUs only in the full-accuracy range. If an S7-300 CPU carries out a calculation whose result is in the limited-accu-

racy range, zero is returned as result and a range violation reported.

REAL addition

The function ADD_R interprets the values at inputs IN1 and IN2 as numbers of data type REAL. It adds the two numbers and stores the sum in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the sum is negative, zero, or positive. Status bits OV and OS indicate any range violations.

In the case of an illegal calculation (one of the input values is an invalid REAL number or you attempt to add $+\infty$ and $-\infty$), ADD_R returns an invalid value at output OUT and sets status bits CC0, CC1, OV and OS to "1".

REAL subtraction

The function SUB_R interprets the values at inputs IN1 and IN2 as numbers of data type REAL. It subtracts the number at input IN2 from the number at input IN1 and stores the difference in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the difference is negative, zero, or positive. Status bits OV and OS indicate any range violations.

In the case of an illegal calculation (one of the input values is an invalid REAL number or you attempt to subtract $+\infty$ from $+\infty$), SUB_R returns an invalid value at output OUT and sets status bits CC0, CC1, OV and OS to "1".

REAL multiplication

The function MUL_R interprets the values at inputs IN1 and IN2 as numbers of data type REAL. It multiplies the two numbers and stores the product in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the product is negative, zero, or positive. Status bits OV and OS indicate any range violations.

In the case of an illegal calculation (one of the input values is an invalid REAL number or you attempt to multiply ∞ and 0), MUL_R returns an invalid value at output OUT and sets status bits CC0, CC1, OV and OS to "1".

REAL division

The function DIV_R interprets the values at inputs IN1 and IN2 as numbers of data type REAL. It divides the number at input IN1 (dividend) by the number at input IN2 (divisor) and stores the quotient in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the quotient is negative, zero, or positive. Status bits CC0 and CC1 indicate any range violations.

In the case of an illegal calculation (one of the input values is an invalid REAL number or you attempt to divide ∞ by ∞ or 0 by 0), DIV_R returns an invalid value at output OUT and sets status bits CC0, CC1, OV and OS to "1".

11 Mathematical Functions

The following mathematical functions are available in LAD and FBD:

- Sine, cosine, tangent
- Arc sine, arc cosine, arc tangent
- Squaring, square-root extraction
- Exponential function to base e, natural logarithm

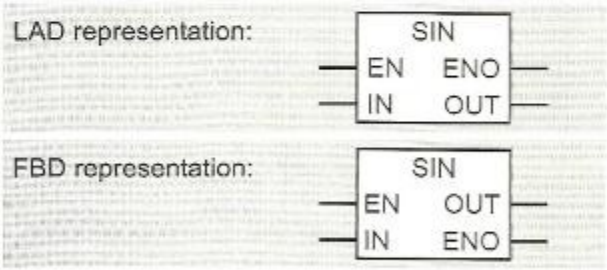
All mathematical functions process REAL numbers.

11.1 Processing a Mathematical Function

Representation

The box for a mathematical function has an input IN and an output OUT in addition to the enable input EN and the enable output ENO. The "header" in the box identifies the mathematical function executed (for example, SIN stands for sine).

Math box
(in example: sine)



The input value is at input IN and the result of the mathematical function is at output OUT. Input and output are of data type REAL. Operands referenced with absolute addresses must be doubleword operands.

See Chapter 3.5.4, "Elementary Data Types", for a description of the bits in REAL format.

Function

The mathematical function is executed if "1" is present at the enable input or if "power" flows in input EN. If an error occurs in the calculation, the enable input is set to "0"; otherwise it is set to "1". If execution of the function is not enabled (EN = "0"), the calculation does not take place and ENO is also "0".

IF EN == "1" or not wired		
THEN		ELSE
OUT := Mfct (IN)		
IF error occurred		
THEN	ELSE	
ENO := "0"	ENO := "1"	ENO := "0"

with Mfct as mathematical function

If the Master Control Relay (MCR) is active, output OUT is set to zero when the mathematical function is processed (EN = "1"). The MCR does not affect the ENO.

The following errors can occur in a mathematical function:

- Range violation (underflow and overflow)
- No valid REAL number as input value

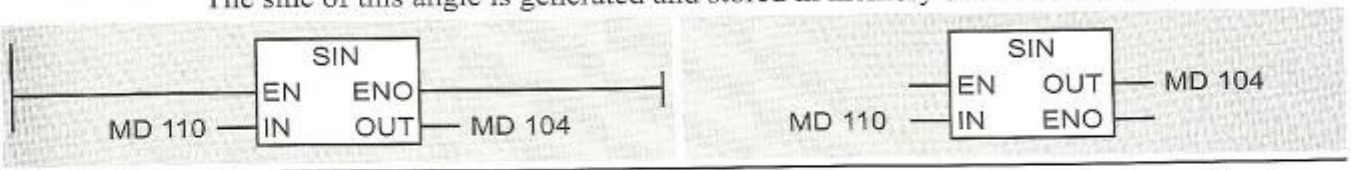
Chapter 15, "Status Bits", explains how the mathematical functions set the status bits.

Examples

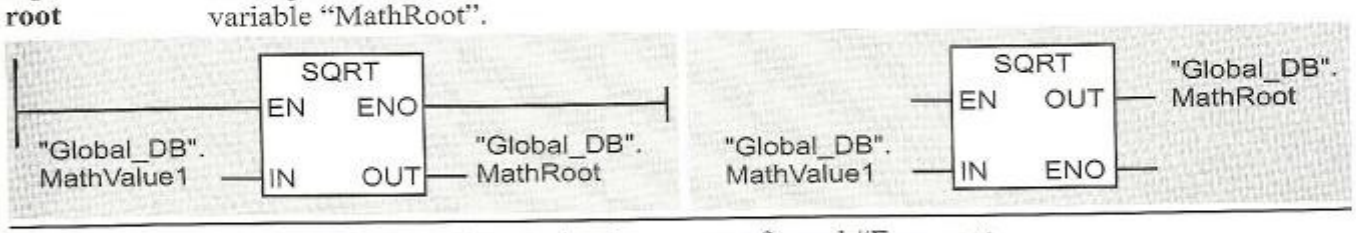
Figure 11.1 shows three examples of mathematical functions. A mathematical function performs the calculation in accordance with REAL even if no data types have been declared when using operands with absolute addresses.

In the case of incremental programming, you will find the mathematical functions in the Program Elements Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under "Floating-Point fct.".

Sine The value in memory doubleword MD 110 contains an angle in radian measure. The sine of this angle is generated and stored in memory doubleword MD 104.



Square root The square root of the value in variable "MathValue1" is generated and stored in the variable "MathRoot".



Exponent The variable #Result contains the power of e and #Exponent.

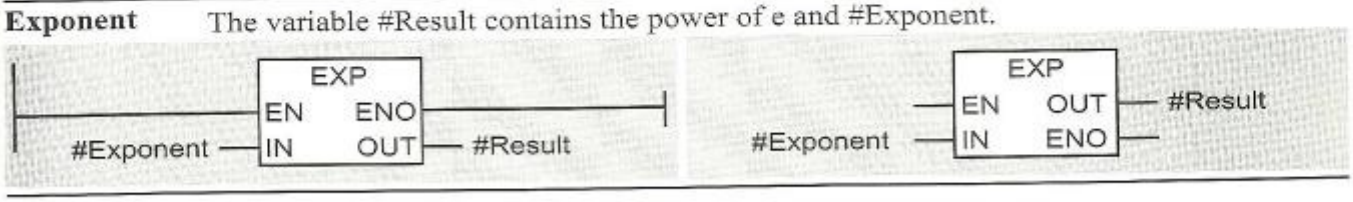


Figure 11.1 Examples of Mathematical Functions

Mathematical function in a rung (LAD)

You can connect contacts in series and in parallel before input EN and after output ENO.

The mathematics box itself may be placed after a T-branch or in a branch that leads directly to the left power rail. This branch can also have contacts before input EN and need not be the uppermost branch.

The direct connection to the left power rail allows you to connect mathematics boxes in parallel. When connecting boxes in parallel, you require a coil to terminate the rung. If you have not provided error evaluation, assign a "dummy" operand to the coil, for example a temporary local data bit.

You can connect mathematics boxes in series. If the ENO output of the preceding box leads to the EN input of the subsequent box, the subsequent box is processed only if the preceding box has been completed without errors. If you want to use the result from the preceding box as the input value for a subsequent box, variables from the temporary local data area make convenient intermediate buffers.

If you arrange several mathematics boxes in one rung (parallel to the left power rail and then further in series), the boxes in the uppermost branch are the first to be processed from left to right, followed by the boxes in the second branch from left to right, and so on.

You can find further examples of the representation and arrangement of mathematical functions in the program "Digital Functions" (FB 111) in the library "LAD_Book" that you can download from the publisher's Website (see page 8).

Mathematical function in a logic circuit (FBD)

If you want to have a mathematics box processed in dependence on specific conditions, you can program binary logic operations before the EN input. You can connect the ENO output with binary inputs from other functions. For example, you can arrange the mathematics boxes in series, whereby the ENO output of the preceding box leads to the EN input of the following box. If you want to use the calculation result of the preceding box as input value to a subsequent box, variables from the temporary local data area make good intermediate buffers.

EN and ENO need not be wired.

You can find further examples of the representation and arrangement of mathematical functions in the program "Digital Functions" (FB 111) in the library "FBD_Book" that you can download from the publisher's Website (see page 8).

11.2 Trigonometric Functions

The trigonometric functions

SIN	Sine,
COS	Cosine and
TAN	Tangent

assume an angle in radian measure as a REAL number at the input.

Two units are conventionally used for giving the size of an angle: degrees from 0° to 360° and radian measure from 0 to 2π (where π = +3.141593e+00). Both can be converted proportionally. For example, the radian measure for a 90° angle is π/2, or +1.570796e+00. With values greater than 2π (+6.283185e+00), 2π or a multiple of 2π is subtracted until the input value for the trigonometric function is less than 2π.

Example (Figure 11.2 or Figure 11.3, Network 4): Calculating the idle power $Ps = U \cdot I \cdot \sin(\varphi)$

Table 11.1 Range of arc functions

Function	Permissible Range	Value Returned
ASIN	–1 to +1	–π/2 to +π/2
ACOS	–1 to +1	0 to π
ATAN	Full range	–π/2 to +π/2

11.3 Arc Functions

The arc functions (inverse trigonometric functions)

ASIN	Arc sine,
ACOS	Arc cosine and
ATAN	Arc tangent

are the inverse functions of the corresponding trigonometric functions. They assume a REAL number in a specific range at input IN and return an angle in the radian measure (Table 11.1).

If the permissible value range is exceeded at input IN, the arc function returns an invalid REAL number and ENO = "0" and sets status bits CC0, CC1, OV and OS to "1".

11.4 Miscellaneous Mathematical Functions

The following mathematical functions are also available

SQR	Compute the square of a number,
SQRT	Compute the square root of a number,
EXP	Compute the exponent to base e and
LN	Find the natural logarithm (logarithm to base e).

Computing the square

The SQR function squares the value at input IN and stores the result in output OUT.

Example: See "Computing the square root".

Computing the square root

The SQRT function extracts the square root of the value at input IN and stores the result in output OUT. If the value at input IN is less than zero, SQRT sets status bits CC0, CC1, OV and OS to "1" and returns an invalid REAL number. If the value at input IN is –0 (minus zero), –0 is returned.

Example: $c = \sqrt{a^2 + b^2}$

Figure 11.2 or Figure 11.3, Network 5: First, the squares of variables a and b are found and then added. Finally, the square root is extracted from the sum. Temporary local data are used as intermediate memory.

(If you have declared a or b as a local variable, you must precede it with # so that the Editor recognizes it as a local variable; if a or b is a global variable, it must be placed in quotation marks.)

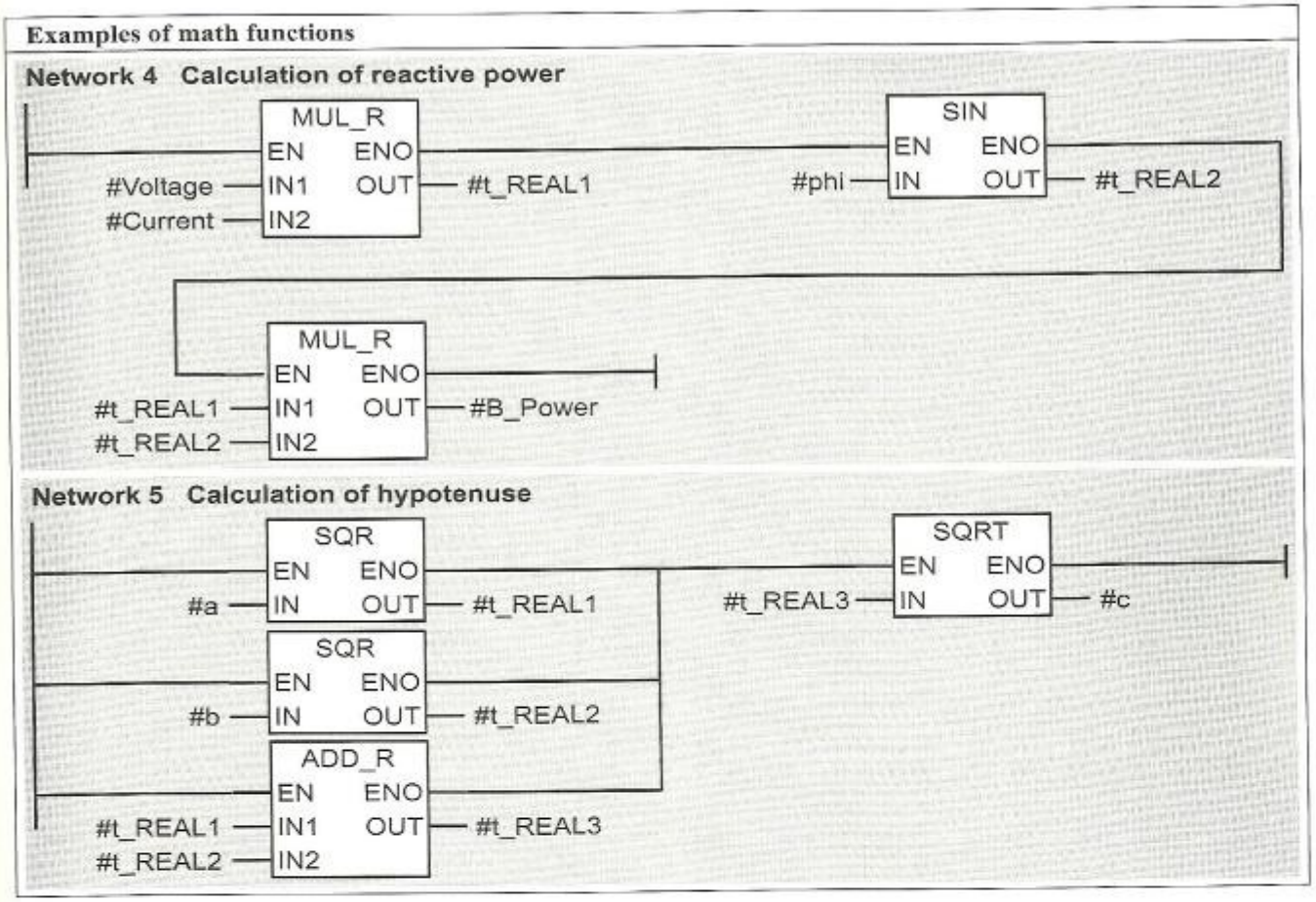


Figure 11.2 Examples of Mathematical Functions (LAD)

Computing the exponent to base e

The EXP function computes the exponential value to base e (= 2.718282*e*+00) and the value at input IN (e^{IN}) and stores the result in output OUT.

You can calculate any exponential value using the formula

$$a^b = e^{b \ln a}$$

Finding the natural logarithm

The LN function finds the natural logarithm to base e (= 2.718282e+00) from the number at input N and stores it in output OUT. If the value at input IN is less than or equal to zero, LN sets status bits CC0, CC1, OV and OS to "1" and returns an invalid REAL number.

The natural logarithm is the inverse function of the exponential function: If $y = e^x$ then $x = \ln y$.

To find any logarithm, use the formula

$$\log_b a = \frac{\log_n a}{\log_n b}$$

where b or n is any base. If you make n = e, you can find a logarithm to any base using the natural logarithm:

$$\log_b a = \frac{\ln a}{\ln b}$$

In the special case for base 10, the formula is as follows:

$$\lg a = \frac{\ln a}{\ln 10} = 0.4342945 \cdot \ln a$$

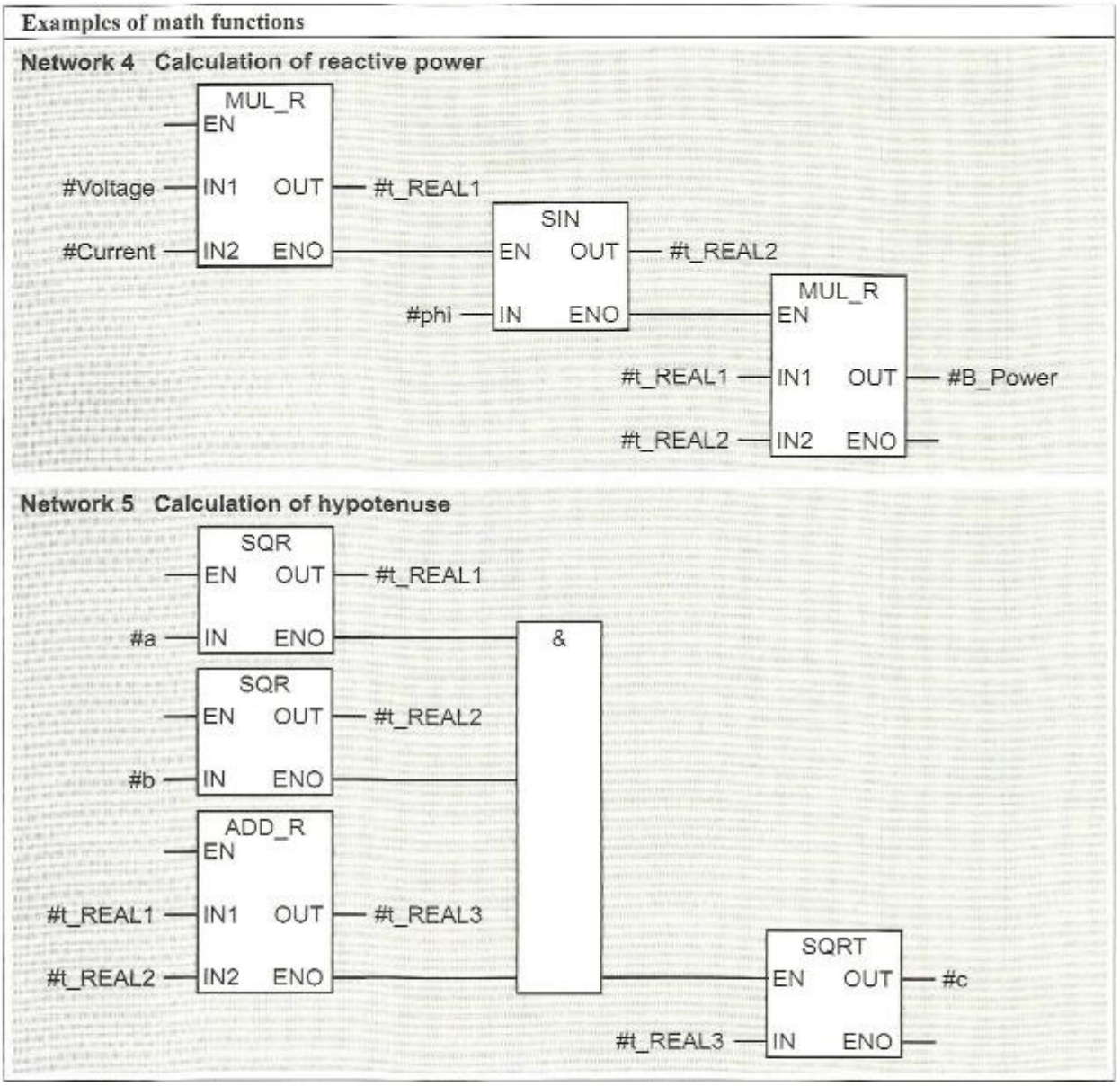


Figure 11.3 Examples of Mathematical Functions (FBD)

12 Conversion Functions

The conversion functions convert the data type of a variable. Figure 12.1 provides an overview of the data type conversions described in this chapter.

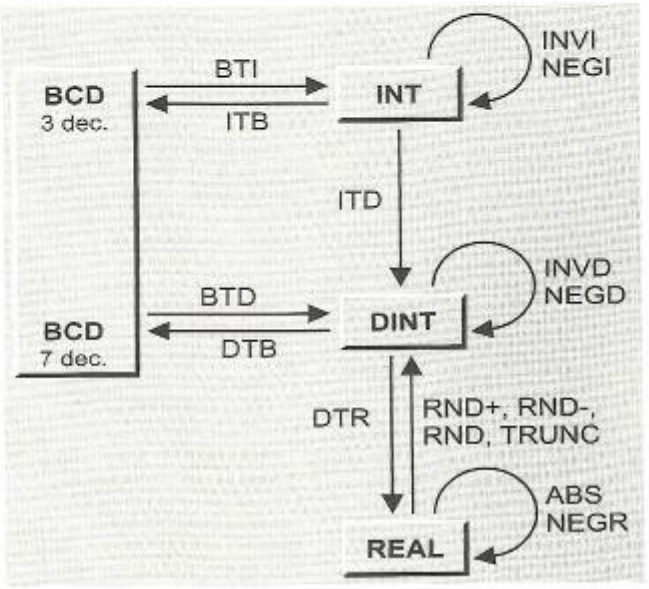


Figure 12.1 Overview of the Conversion Functions

12.1 Processing a Conversion Function

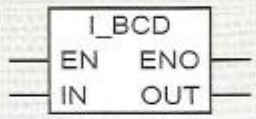
Representation

In addition to the enable input EN and the enable output ENO, the box for a conversion function has an input IN and an output OUT. The "header" in the box identifies the conversion function executed (for example, I_BCD stands for the conversion of INT to BCD).

Conversion box
(in example: INT to BCD)

LAD representation:

FBD representation:



The value to be converted is at input IN and the result of the conversion is at output OUT. The data type of the input and that of the output depend on the conversion function. In the conversion function DI_R (DINT to REAL), for instance, the input is of type DINT and the output of type REAL. The variables applied must be of the same data type as the input or the output. If you use operands with absolute addresses, the sizes of the operands must be matched to the data types; for example, you can use a word operand for data type INT.

You can find the bit descriptions for the data formats in Chapter 3.5.4, "Elementary Data Types".

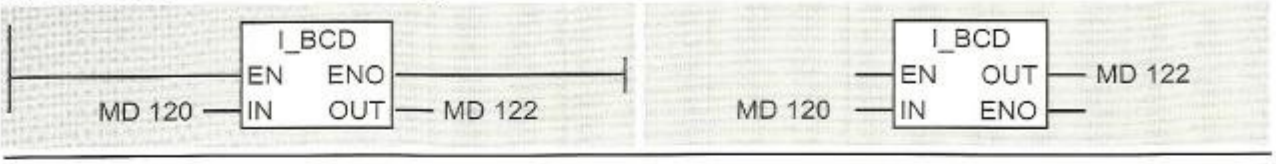
Function

The conversion function is executed if "1" is present at the enable input (if current flows in input EN). If an error occurs during conversion, the enable output ENO is set to "0"; otherwise, it is set to "1". If execution of the function is not enabled (EN = "0"), the conversion does not take place and ENO is also "0".

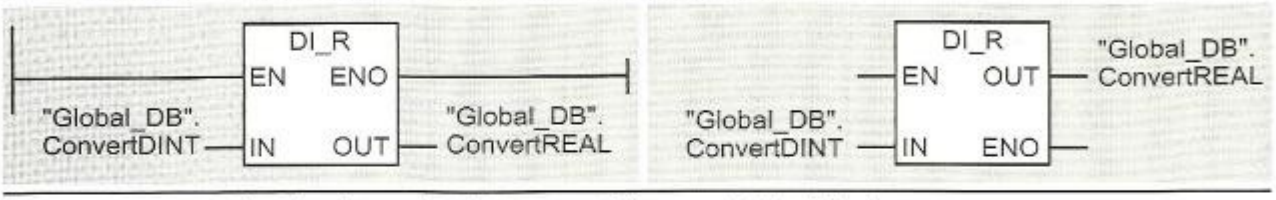
<table>
<tr><td colspan="2">IF EN == "1" or not wired
THEN</td><td>ELSE</td></tr>
<tr><td colspan="2">OUT := Confct (IN)</td><td rowspan="3"></td></tr>
<tr><td colspan="2">IF error occurred
THEN</td></tr>
<tr><td>ENO := "0"</td><td>ELSE
ENO := "1"</td><td>ENO := "0"</td></tr>
</table>

with Confct as conversion function

Converting INT numbers The value in memory doubleword MD 120 is interpreted as an INT number and stored in memory doubleword MD 122 as a BCD number.



Converting DINT numbers The value in variable "ConvertDINT" is interpreted as a DINT number and stored as a real number in the variable "ConvertREAL".



Converting REAL numbers The absolute value is generated from variable #Display.

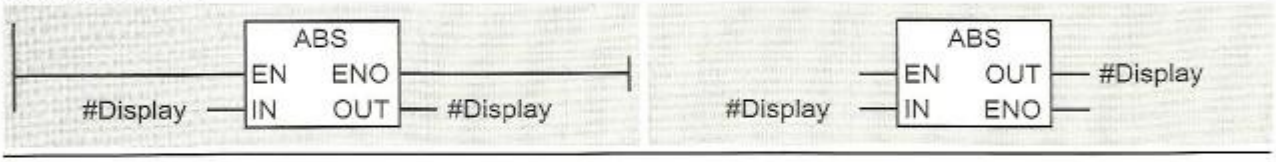


Figure 12.2 Examples of Conversion Functions

If the Master Control Relay (MCR) is active, output OUT is set to zero when the conversion function is processed (EN = "1"). The MCR has no effect on output ENO.

Not all conversion functions report errors. An error occurs only if the permissible number range is exceeded (I_BCD, DI_BCD) or an invalid REAL number if specified (FLOOR, CEIL, ROUND, TRUNC).

If the input value for a BCD_I or BCD_DI conversion contains a pseudo tetrad, program execution is interrupted and error organization block OB 121 (programming errors) called.

Chapter 15, "Status Bits", explains how the conversion functions set the status bits.

Figure 12.2 shows one example for each data type. A conversion function converts according to the specific characteristic even if no data types have been declared when using operands with absolute addresses.

In the case of incremental programming, you will find the conversion functions in the Program Elements Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under "Converter".

Conversion function in a rung (LAD)

You can connect contacts in series and in parallel before the EN input and after the ENO output.

The conversion box itself may be placed after a T-branch or in a branch that leads directly to the left power rail. This branch can also have contacts before input EN and it need not be the uppermost branch.

With the direct connection to the left power rail you can thus connect conversion boxes in parallel. When connecting boxes in parallel, you require a coil to terminate the rung. If you have not provided error evaluation, assign a "dummy" operand to the coil, for instance a temporary local data bit.

You can connect conversion boxes in series. If the ENO output of the preceding box leads to the EN output of the subsequent box, the subsequent box is processed only if the preceding box has been completed without errors. If you

want to use the result from the preceding box as the input value for a subsequent box, variables from the temporary local data area make convenient intermediate buffers.

If you arrange several conversion boxes in one rung (parallel to the left power rail and then further in series), the boxes in the uppermost branch are processed first from left to right, followed by the boxes in the second branch from left to right, and so on.

You can find further examples of the representation and arrangement of conversion functions in the program "Digital Functions" (FB 111) in the library "LAD_Book" that you can download from the publisher's Website (see page 8).

Conversion function in a logic circuit (FBD)

If you want to process the conversion box in dependence on specific conditions, you can arrange binary logic operations before the EN input. You can connect the ENO output with binary inputs of other functions; for example, you can arrange conversion boxes in series, whereby the ENO output of the preceding box leads to the EN input of the subsequent box. If you want to use the result from the preceding box as input value for a subsequent box, variables from the temporary local data area make good temporary buffers.

EN and ENO need not be wired.

You can find further examples of the representation and arrangement of conversion functions in the program "Digital Functions" (FB 111) in the library "FBD_Book" that you can download from the publisher's Website (see page 8).

12.2 Conversion of INT and DINT Numbers

Table 12.1 shows the conversion functions for INT and DINT numbers. Variables of the specified data type or absolute-addressed operands of the relevant size must be applied to the inputs and outputs of the boxes (for example a word operand for data type INT).

Conversion from INT to DINT

The function I_DI interprets the value at input IN as a number of data type INT and transfers it to the low-order word of output OUT. The signal state of bit 15 (the sign) of the input is transferred to bits 16 to 31 of the high-order word of output OUT.

The conversion from INT to DINT reports no errors.

Conversion from INT to BCD

The function I_BCD interprets the value at input IN as a number of data type INT and converts it to a 3-decade BCD number at output OUT. The three right-justified decades represent the absolute value of the decimal number. The sign is in bits 12 to 15. If all bits are "0", the sign is positive; if all bits are "1", the sign is negative.

If the INT number is too large to be converted into BCD (> 999), the I_BCD function sets status bits OV and OS. The conversion does not take place.

Conversion from DINT to BCD

The function DI_BCD function interprets the value at input IN as a number of data type DINT and converts it to a 7-decade BCD number at output OUT. The seven right-justified decades represent the absolute value of the decimal number. The sign is in bits 28 to 31. If all bits are "0", the sign is positive; if all bits are "1", the sign is negative.

If the DINT number is too large to be converted to a BCD number (> 9 999 999), status bits OV and OS are set. The conversion does not take place.

Table 12.1 Conversion of INT and DINT Numbers

Data Type Conversion	Box	Data Type for Parameter	
		IN	OUT
INT to DINT	I_DI	INT	DINT
INT to BCD	I_BCD	INT	WORD
DINT to BCD	DI_BCD	DINT	DWORD
DINT to REAL	DI_R	DINT	REAL

Conversion from DINT to REAL

The function DI_R interprets the value at input IN as a number of data type DINT and converts it to a REAL number at output OUT.

Since a number in DINT format has a higher accuracy than a number in REAL format, rounding make take place during the conversion. The REAL number is then rounded to the next integer (in accordance with the ROUND function).

The DI_R function does not report errors.

12.3 Conversion of BCD Numbers

Table 12.2 shows the functions for converting BCD numbers. Variables of the specified type of absolute-addressed operands of the relevant size must be applied to the input and outputs of the boxes (for example a word operand for data type INT).

Table 12.2 Conversion of BCD Numbers

Data Type Conversion	Box	Data Type for Parameter	
		IN	OUT
BCD to INT	BCD_I	WORD	INT
BCD to DINT	BCD_DI	DWORD	DINT

Conversion from BCD to INT

The function BCD_I interprets the value at input IN as a 3-decade BCD number and converts it to an INT number at output OUT. The three right-justified decades represent the absolute value of the decimal number. The sign is in bits 12 to 15. If these bits are "0", the sign is positive; it they are "1", the sign is negative. Only bit 15 is taken into account in the conversion.

If the BCD number contains a pseudo tetrad (numerical value 10 to 15 or A to F in hexadecimal), the CPU signals a programming error and calls organization block OB 121 (synchronization error). If this block is not available, the CPU goes to STOP.

The function BCD_I sets no status bits.

Conversion from BCD to DINT

The function BCD_DI interprets the value at input IN as a 7-decade BCD number and converts it to an INT number at output OUT. The seven right-justified decades represent the absolute value of the decimal number. The sign is in bits 28 to 31. If these bits are "0", the sign is positive; if they are "1", the sign is negative. Only bit 31 is taken into account in the conversion.

If the BCD number contains a pseudo tetrad (numerical value 10 to 15 or A to F in hexadecimal), the CPU signals a programming error and calls organization block OB 121 (synchronization error). If this block is not available, the CPU goes to STOP.

The function BCD_I sets no status bits.

12.4 Conversion of REAL Numbers

There are several functions for converting a number in REAL format to DINT format (conversion of a fractional value to an integer) (Table 12.3). They differ as regards rounding. Variables of the specified data type or absolute-addressed doubleword operands must be applied to the inputs and outputs of the boxes.

Table 12.3
Conversion of REAL Numbers to DINT Numbers

Data Type Conversion with Rounding	Box	Data Type for Parameter	
		IN	OUT
To next higher integer	CEIL	REAL	DINT
To next lower integer	FLOOR	REAL	DINT
To next integer	ROUND	REAL	DINT
Without rounding	TRUNC	REAL	DINT

Rounding to the next higher integer

The function CEIL interprets the value at input IN as a number in REAL format and converts it to a number in DINT format at output OUT. CEIL returns an integer that is greater than or equal to the number to be converted.

Table 12.4 Rounding Modes for the Conversion of REAL Numbers

Input value		Result			
REAL	DW#16#	ROUND	CEIL	FLOOR	TRUNC
1.0000001	3F80 0001	1	2	1	1
1.00000000	3F80 0000	1	1	1	1
0.99999995	3F7F FFFF	1	1	0	0
0.50000005	3F00 0001	1	1	0	0
0.50000000	3F00 0000	0	1	0	0
0.49999996	3EFF FFFF	0	1	0	0
5.877476E–39	0080 0000	0	1	0	0
0.0	0000 0000	0	0	0	0
–5.877476E–39	8080 0000	0	0	–1	0
–0.49999996	BEFF FFFF	0	0	–1	0
–0.50000000	BF00 0000	0	0	–1	0
–0.50000005	BF00 0001	–1	0	–1	0
–0.99999995	BF7F FFFF	–1	0	–1	0
–1.00000000	BF80 0000	–1	–1	–1	–1
–1.0000001	BF80 0001	–1	–1	–2	–1

If the value at input IN exceeds or falls short of the range permissible for a number in DINT format or if it does not correspond to a number in REAL format, CEIL sets status bits OV and OS. Conversion does not take place.

Rounding to the next lower integer

The function FLOOR interprets the value at input IN as a number in REAL format and converts it to a number in DINT format at output OUT. FLOOR returns an integer that is less than or equal to the number to be converted.

If the value at input IN exceeds or falls short of the range permissible for a number in DINT format or if it does not correspond to a number in REAL format, FLOOR sets status bits OV and OS. Conversion does not take place.

Rounding to the next integer

The function ROUND interprets the value at input IN as a number in REAL format and converts it to a number in DINT format at output OUT. ROUND returns the next integer. If the result lies exactly between an odd and an even number, the even number is given preference.

If the value at input IN exceeds or falls short of the range permissible for a number in DINT format or if it does not correspond to a number in REAL format, ROUND sets status bits OV and OS. Conversion does not take place.

No rounding

The function TRUNC interprets the value at input IN as a number in REAL format and converts it to a number in DINT format at output OUT. TRUNC returns the integer component of the number to be converted; the fractional component is "truncated".

If the value at input IN exceeds or falls short of the range permissible for a number in DINT format or if it does not correspond to a number in REAL format, TRUNC sets status bits OV and OS. Conversion does not take place.

Summary of conversions from REAL to DINT

Table 12.4 shows the different effects of the functions for converting from REAL to DINT. The range –1 to +1 has been chosen as example.

12.5 Miscellaneous Conversion Functions

Other available conversion functions are one's complement generation, negation, and absolute-value generation of a REAL number (Table 12.5). Variables of the specified data type or absolute-addressed operands of the relevant size must be applied to the inputs and outputs of the boxes (for example a doubleword operand for data type DINT).

Table 12.5 Miscellaneous Conversion Functions

Conversion	Box	Data Type for Parameter	
		IN	OUT
One's complement INT	INV_I	INT	INT
One's complement DINT	INV_DI	DINT	DINT
Negation of an INT number	NEG_I	INT	INT
Negation of a DINT number	NEG_DI	DINT	DINT
Negation of a REAL number	NEG_R	REAL	REAL
Absolute value of a REAL number	ABS	REAL	REAL

One's complement INT

The function INV_I negates the value at input IN bit for bit and writes it to output OUT. INV_I replaces the zeroes with ones and vice versa. The function INV_I does not signal errors.

One's complement DINT

The function INV_DI negates the value at input IN bit for bit and writes it to output OUT. INV_DI replaces the zeroes with ones and vice versa. The function INV_DI does not signal errors.

Negation INT

The function NEG_I interprets the value at input IN as an INT number, changes the sign through two's complement generation, and writes the converted value to output OUT. NEG_I is identical to multiplication with –1. The function NEG_I sets status bits CC0, CC1, OV and OS.

Negation DINT

The function NEG_DI interprets the value at input IN as a DINT number, changes the sign through two's complement generation, and writes the converted value to output OUT. NEG_DI is identical to multiplication by –1. The function NEG_DI sets status bits CC0, CC1, OV and OS.

Negation REAL

The function NEG_R interprets the value at input IN as a REAL number, multiplies this number by –1, and writes it to output OUT. NEG_R changes the sign of the mantissa even on an invalid REAL number. NEG_R does not signal errors.

Absolute-value generation REAL

The ABS function interprets the value at input IN as a REAL number, generates the absolute value from this number, and writes it to output OUT. ABS sets the sign of the mantissa to "0" even on an invalid REAL number. ABS does not signal errors.

13 Shift Functions

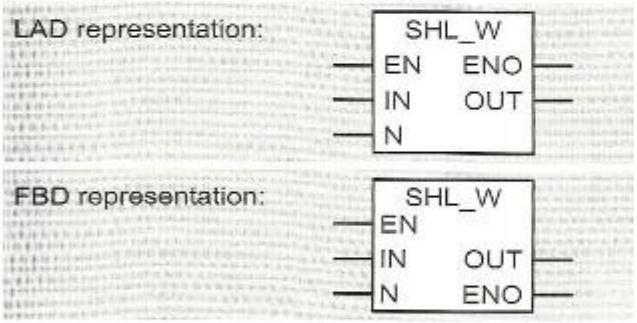
The shift functions shift the contents of a variable bit by bit to the left or right. The bits shifted out are either lost or are used to pad the other side of the variable. Table 13.1 provides an overview of the shift functions.

13.1 Processing a Shift Function

Representation

In addition to an enable input EN and an enable output ENO, the box for a shift function has an input IN, an input N, and an output OUT. The "header" in the box identifies the shift function executed (for example, SHL_W stands for shifting a word variable to the left).

Shift box
(in example: shift to left word by word)



The value to be shifted is at input IN, the number of places to be shifted is at input N, and the result is at output OUT. The input and the output have different data types depending on the shift function. For example, input and output are of type DWORD for the shift function SHR_DW (shift a doubleword variable to the right). The variables applied must be of the same data type as the input or output. If you use operands with absolute addresses, the operand sizes must accord with the data types; for instance, you can use a word operand for data type INT. Input N has data type WORD for every shift function.

See Chapter 3.5.4, "Elementary Data Types", for a description of the bits in each data format.

Table 13.1 Overview of Shift Functions

Shift Functions	Word Variable	Doubleword Variable
Shift left	SHL_W	SHL_DW
Shift right	SHR_W	SHR_DW
Shift with sign	SHR_I	SHR_DI
Rotate left	-	ROL_DW
Rotate right	-	ROR_DW

Function

The shift function is executed if "1" is present at the enable input or when "power" flows through input EN. ENO is then "1". If execution of the function is not enabled (EN = "0"), the shift does not take place and ENO is also "0".

IF EN == "1" or not wired	
THEN	ELSE
OUT := Sfct (IN, N)	
ENO := "1"	ENO := "0"

with Sfct as shift function

When the Master Control Relay (MCR) is activated, output OUT is set to zero when the shift function is processed (EN = "1"). The MCR does not affect output ENO.

Chapter 15, "Status Bits", explains how the shift functions set the status bits.

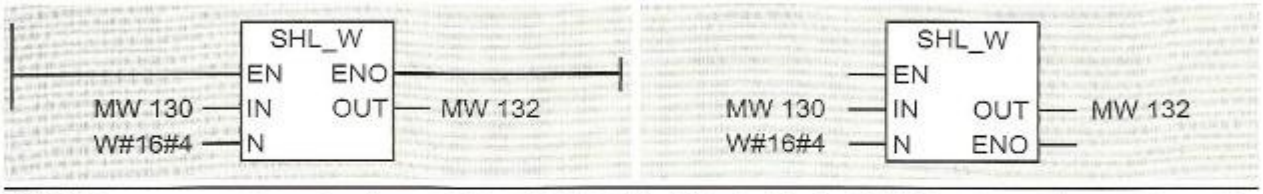
Examples

Figure 13.1 gives one example each for various shift functions.

In the case of incremental programming, you will find the shift functions in the Program Elements Catalog (with VIEW → OVERVIEWS [Ctrl

Shifting word variables

The value in memory word MW 130 is shifted 4 positions to the left and stored in memory word MW 132.



Shifting doubleword variables

The value in variable "ShiftOn" is shifted right by "ShiftPos" positions and stored in "ShiftOff".

Shifting with sign

The variable #ActVal is shifted, with sign, 2 positions to the right and transferred to the variable #Display.

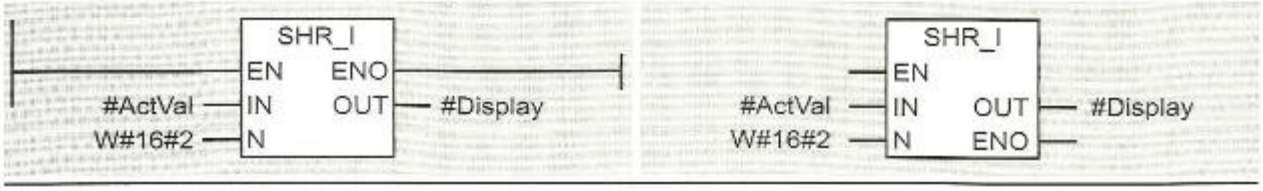


Figure 13.1 Examples of Shift Functions

- K] or INSERT → PROGRAM ELEMENTS) under "Shift/Rotate".

Shift function in a rung (LAD)

You can connect contacts in series and in parallel before input EN and after output ENO.

The shift box itself may be placed after a T-branch or in a branch that leads directly to the left power rail. This branch can also have contacts before input EN and it need not be the uppermost branch.

The direct connection to the left power rail allows you to connect shift boxes in parallel. When connecting boxes in parallel, you require a coil to terminate the rung. If you have not provided error evaluation, assign a "dummy" operand to the coil, for example a temporary local data bit.

You can connect shift boxes in series. If the ENO output from the preceding box leads to the EN input of the subsequent box, the subsequent is always processed. If you want to use the result from the preceding box as input value for a subsequent box, variables from the temporary local data area make a convenient intermediate buffer.

If you arrange several boxes in one rung (parallel to the left power rail and then further in series), the boxes in the uppermost branch are processed first from left to right, followed by the boxes in the second branch from left to right, and so on.

You can find further examples of the representation and arrangement of shift functions in the library "Digital Functions" program (FB 111) in the library "LAD_Book" that you can download from the publisher's Website (see page 8).

Shift function in a rung (FBD)

If you want the shift box processed in dependence on specific conditions, you can arrange binary logic operations before the EN input. You can connect the ENO output with binary inputs of other functions; for instance, you can arrange shift boxes in series, whereby the EBO output of the preceding box leads to the EN output of the subsequent box. If you want to use the result from the preceding box as input value to a subsequent box, variables from the temporary local data area make convenient intermediate buffers.

EN and ENO need not be wired.

You can find further examples of the representation and arrangement of shift functions in the program "Digital Functions" (FB 111) in the library "FBD_Book" that you can download from the publisher's Website (see page 8).

13.2 Shift

Shift word variable to the left

The shift function SHL_W shifts the contents of the WORD variable at input IN bit by bit to the left the number of positions specified by the shift number at input N. The bit positions freed up by the shift are padded with zeroes. The WORD variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is greater than 15, the output variable contains zero following execution of the SHL_W function.

Shift doubleword variable to the left

The shift function SHL_DW shifts the contents of the DWORD variable at input IN bit by bit to the left the number of positions specified by the shift number at input N. The bit positions freed up by the shift are padded with zeroes. The DWORD variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is greater than 31, the output variable contains zero following execution of the SHL_DW function.

Shift word variable to the right

The shift function SHR_W shifts the contents of the WORD variable at input IN bit by bit to the right the number of positions specified by the shift number at input N. The bit positions freed up by the shift are padded with zeroes. The WORD variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is greater than 15, the output variable contains zero following execution of the SHR_W function.

Shift doubleword variable to the right

The shift function SHR_DW shifts the contents of the DWORD variable at input IN bit by bit to the right the number of positions specified by the shift number at input N. The bit positions freed up by the shift are filled with zeroes. The DWORD variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is greater than 31, the output variable contains zero following execution of the SHR_DW function.

Shift word variable with sign

The shift function SHR_I shifts the contents of the INT variable at input IN bit by bit to the right the number of positions specified by the shift number at input N. The bit positions freed up by the shift are filled with the signal state of bit 31 (which is the sign of the INT number), that is, with "0" if the number is positive and

with "1" if the number is negative. The INT variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is greater than 15, all bit positions in the output variable contain the sign following execution of the SHR_I function.

For a number in data format INT, shifting to the right is equivalent to division with an exponential number to base 2. The exponent is the shift number. The result of such a division corresponds to the integer rounded down.

Shift doubleword variable with sign

The shift function SHR_DI shifts the contents of the DINT variable at input IN bit by bit to the right the number of positions specified by the shift number at input N. The bit positions freed up by the shift are padded with the signal state of bit 15 (which is the sign of the DINT number), that is, with "0" if the number is positive and with "1" if the number is negative. The DINT variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is greater than 31, all bit positions in the output variable contain the sign following execution of the SHR_DI function.

For a number in data format DINT, shifting to the right is equivalent to division with an exponential number to base 2. The exponent is the shift number. The result of such a division corresponds to the integer rounded down.

13.3 Rotate

Rotate doubleword variable to the left

The shift function ROL_DW shifts the contents of the DWORD variable at input IN bit by bit to the left the number of positions specified in the shift number at input N. The bit positions freed up by the shift are padded with the bit positions that were shifted out. The DWORD variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is 32, the contents of the input variable are retained and the status bits are set. If the shift number is 33, a shift of one position is executed; if it is 34, a shift of two positions is executed, and so on (shifting is carried out modulo 32).

Rotate doubleword variable to the right

The shift function ROR_DW shifts the contents of the DWORD variable at input IN bit by bit to the right the number of positions specified by the shift number at input N. The bit positions freed up by the shift are padded with the bit positions that were shifted out. The DWORD variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is 32, the contents of the input variable are retained and the status bits are set. If the shift number is 33, a shift of one position is executed; if the shift number is 34, a shift of two positions is executed, and so on (shifting is executed modulo 32).

14 Word Logic

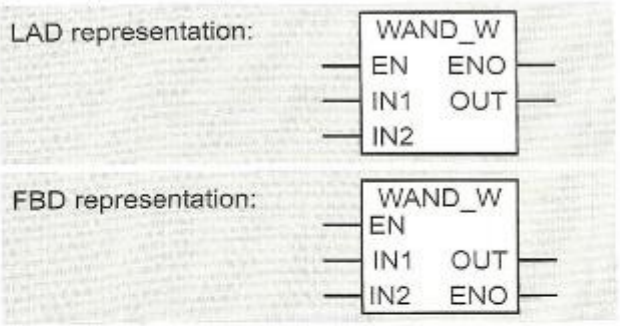
Word logic combines the values of two variables bit by bit according to AND, OR, or Exclusive OR. The logic operation may be applied to words or doublewords. The available word logic operations are listed in Table 14.1.

14.1 Processing a Word Logic Operation

Representation

In addition to the enable input EN and the enable output ENO, the box for a word logic operation has two inputs, IN1 and IN2, and one output, OUT. The "header" in the box identifies the word logic operation executed (for example, WAND_W stands for word ANDing).

Word logic box
(in example: AND operation word by word)



The values to be combined are at inputs IN1 and IN2, and the result of the operation is at output OUT. The inputs and the output have different data types, depending on the operation: WORD for word (16-bit) operations, and DWORD for doubleword (32-bit) operations. The applied variables must be of the same data type as the inputs or the output.

See Chapter 3.5.4, "Elementary Data Types", for a description of the bits in the various data formats.

Table 14.1 Overview of Word Logic Operations

Word logic operation	With a word variable	doubleword variable
AND	WAND_W	WAND_DW
OR	WOR_W	WOR_DW
Exclusive OR	WXOR_W	WXOR_DW

Function

The word logic operation is executed when "1" is present at the enable input (when power flows through the input EN). If execution of the operation is not enabled (EN = "0"), the operation does not take place and ENO is also "0".

IF EN == "1" or not wired	
THEN	ELSE
OUT := IN1 Wlog IN2	
ENO := "1"	ENO := "0"

with Wlog as word logic

If the Master Control Relay (MCR) is active, output OUT is set to zero when the word logic operation is executed (EN = "1"). The MCR does not affect the ENO output.

Word logic operations generate a result bit by bit. Bit 0 of input IN1 is combined with bit 0 of input IN2, and the result is stored in bit 0 of output OUT. The same is done with bit 1, bit 2, and so on, up to bits 15 and 31. Table 14.2 shows the result formation for a single bit.

Chapter 15, "Status Bits", explains how the word logic operations set the status bits.

Examples

Figure 14.1 shows one example for each word logic operation.

In the case of incremental programming, you will find the word logic operations in the Pro-

Table 14.2
Result Formation in Word Logic Operations

Contents of input IN1	0	0	1	1
Contents of input IN2	0	1	0	1
Result with AND	0	0	0	1
Result with OR	0	1	1	1
Result with Exclusive OR	0	1	1	0

gram Elements Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under "Word Logic".

Word logic in a rung (LAD)

You can arrange contacts in series and in parallel before input EN and after output ENO.

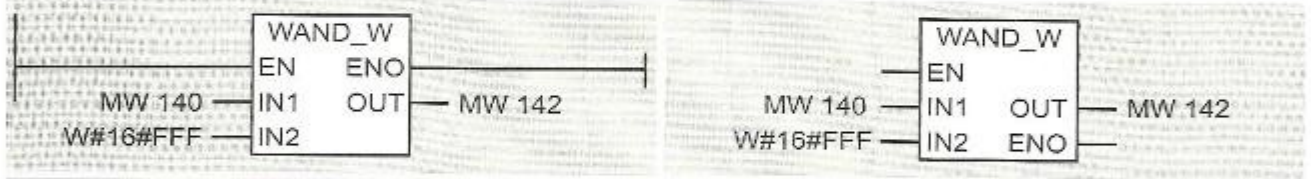
The word logic box itself may be placed after a T-branch or in a branch that leads directly to the left power rail. This branch can also have contacts before input EN and it need not be the uppermost branch.

The direct connection to the left power rail allows you to connect word logic boxes in parallel. When connecting boxes in parallel, you require a coil to terminate the rung. If you have not provided error evaluation, assign a "dummy" operand to the coil, for example a temporary local data bit.

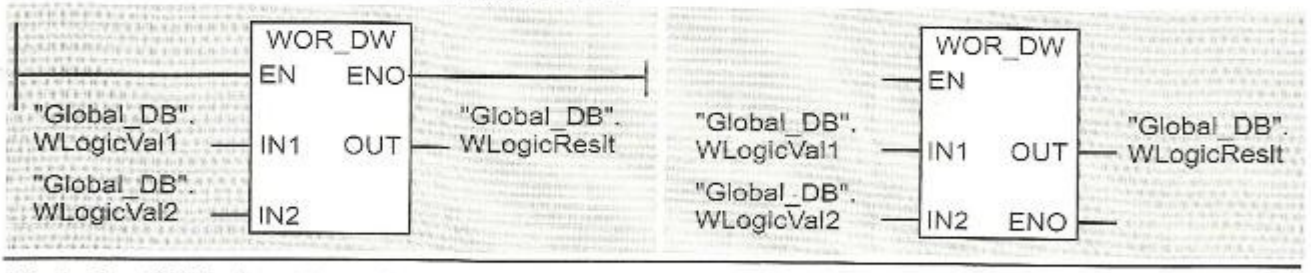
You can connect word logic boxes in series. If the ENO output from the preceding box leads to the EN input of the subsequent box, the subsequent box is always processed. If you want to use the result of the preceding box as input value for a subsequent box, variables from the temporary local data area make convenient intermediate buffers.

If you arrange several word logic boxes in one rung (parallel to the left power rail and then further in series), the boxes in the uppermost branch are processed first from left to right, fol-

AND logic The four high-order bits of memory word MW 140 are set to "0"; the result is stored in memory word MW 142.



OR logic Variables "WLogicVal1" and "WLogicVal2" are ORed bit for bit and the result stored in "WLogicReslt".



Exclusive-OR logic The value generated by combining variables #Input and #Mask with Exclusive-OR is in variable #Buffer.

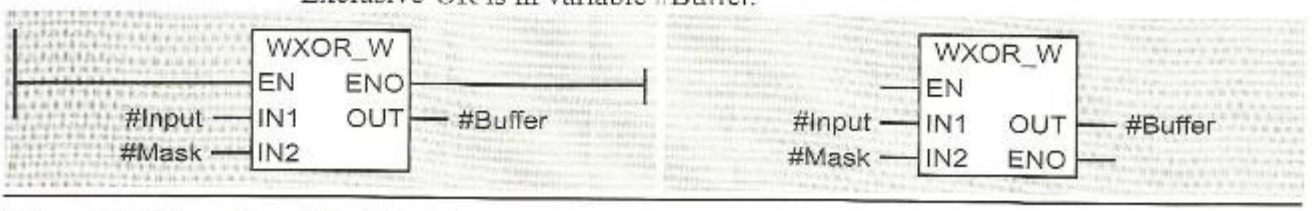


Figure 14.1 Examples of Word Logic Operations

lowed by the boxes in the second branch from left to right, and so on.

You can find further examples of the representation and arrangement of word logic operations in the program "Digital Functions" (FB 111) in the library "LAD_Book" that you can download from the publisher's Website (see page 8).

Word logic in a logic circuit (FBD)

If you want to have the word logic box processed in dependence on specific conditions, you can arrange binary logic operations before the EN input. You can connect the ENO output with binary inputs from other functions; for example, you can arrange word logic boxes in series, whereby the ENO output of the preceding box leads to the EN input of the subsequent box. If you want to use the result from the preceding box as input value for a subsequent box, variables in the temporary local data area make convenient intermediate buffers.

EN and ENO need not be wired.

You can find further examples of the representation and arrangement of logic operations in the program "Digital Functions" (FB 111) in the library "FBD_Book" that you can download from the publisher's Website (see page 8).

14.2 Description of the Word Logic Operations

AND operation

AND combines the individual bits of the value at input IN1 with the corresponding bits of the value at input IN2 according to AND. A bit in result word OUT will be "1" only if the corresponding bit in both values to be combined is "1".

Since the bits that are "0" at input IN2 also set the corresponding result bits to "0" regardless of what value these bits have at input IN1, we also refer to these bits as being "masked". Masking is the primary use for the (digital) AND operation.

OR operation

OR combines the individual bits of the value at input IN1 with the corresponding bits of the value at input IN2 according to OR. A bit result word OUT will be "0" only if the corresponding bit in both values to be combined is "0".

Since the bits that are "1" at input IN2 also set the corresponding result bits to "1" regardless of what value these bits have at input IN1, we also refer to these bits as being "masked". Masking is the primary use for the (digital) OR operation.

Exclusive OR operation

Exclusive OR combines the individual bits of the value at input IN1 with the corresponding bits of the value at input IN2 according to Exclusive OR. A bit in result word OUT will be "1" only if the corresponding bit in only one of the two values to be combined is "1". If a bit at input IN2 is "1", the corresponding bit in the result is the reverse of the bit at the same position in IN1.

In the result, only those bits with opposing signal states in IN1 and IN2 prior to execution of the digital Exclusive OR operation will be "1". Locating bits with opposing signal states or "negating" the signal states of individual bits is the primary use for the (digital) Exclusive OR operation.

Program Flow Control

LAD and FBD provide you with a variety of options for controlling the flow of the program. You can exit linear program execution within a block or you can structure the program with programmable block calls. You can affect program execution in dependence on values calculated at runtime, or in dependence on process parameters, or in accordance with your plant status.

The **status bits** provide information on the result of an arithmetic or mathematical function and on errors (such as a range violation during a calculation). You can incorporate the signal states of the status bits directly in your program using contacts.

You can use **jump functions** to branch unconditionally or in dependence on the RLO.

A further method of affecting program execution is provided by the **Master Control Relay** (MCR). Originally developed for relay contactor controls, LAD and FBD offer a software version of this program control method.

You can use **block functions** to structure your program. You can use functions and function blocks again and again by defining **block parameters**.

Chapter 19, "Block Parameters", contains the examples shown in Chapter 5, "Memory Functions", and Chapter 8, "Counters", this time programmed as function blocks with block parameters. These function blocks are then also called in the "Feed" example as local instances.

15 Status Bits
Status bits RLO, BR, CC0, CC1 and overflow; setting and evaluating the status bits; using the binary result; EN/ENO

16 Jump Functions
Jump unconditionally; jump in dependence on the RLO

17 Master Control Relay
MCR-dependence; MCR range; MCR zone

18 Block Functions
Block types, block call, block end; status local data; data block register, using data operands; handling data blocks

19 Block Parameters
Parameter declaration; formal parameters, actual parameters; passing parameters to called blocks; Examples: Conveyor belt, parts counter and supply