SIEMENS

SIMATIC

System Software for S7-300 and S7-400 Program Design

Programming Manual

C79000-G7076-C506-01

Safety Guidelines

This manual contains notices which you should observe to ensure your own personal safety, as well as to protect the product and connected equipment. These notices are highlighted in the manual by a warning triangle and are marked as follows according to the level of danger:

Danger

indicates that death, severe personal injury or substantial property damage will result if proper precautions are not taken.

Warning

indicates that death, severe personal injury or substantial property damage can result if proper precautions are not taken.

Caution

indicates that minor personal injury or property damage can result if proper precautions are not taken.

Note

draws your attention to particularly important information on the product, handling the product, or to a particular part of the documentation.

Qualified Personnel

The device/system may only be set up and operated in conjunction with this manual.

Only **qualified personnel** should be allowed to install and work on this equipment. Qualified persons are defined as persons who are authorized to commission, to ground, and to tag circuits, equipment, and systems in accordance with established safety practices and standards.

Correct Usage

Note the following:

Warning

This device and its components may only be used for the applications described in the catalog or the technical description, and only in connection with devices or components from other manufacturers which have been approved or recommended by Siemens.

This product can only function correctly and safely if it is transported, stored, set up, and installed correctly, and operated and maintained as recommended.

Trademarks

SIMATIC® and SINEC® are registered trademarks of SIEMENS AG.

Third parties using for their own purposes any other names in this document which refer to trademarks might infringe upon the rights of the trademark owners.

Siemens AG
Automation Group
Industrial Automation Systems
Postfach 4848, D-90327 Nürnberg

Technical data subject to change.
© Siemens AG 1996

Siemens Aktiengesellschaft

C79000-G7076-C506

Preface

Purpose

This manual describes the various ways in which you can program your S7-300/S7-400 programmable logic controller (PLC). The manual focuses primarily on the tasks you need to perform when designing a project "on paper".

The manual has the following aims:

- To familiarize you with the operating systems of the S7-300 and S7-400 CPUs
- To support you when designing your user program
- To inform you about the opportunities for communication and diagnostics with the S7-300 and S7-400 CPUs

For information about the different programming languages, refer to the corresponding manuals (refer also to the overview of the STEP 7 documentation).

Audience

This manual is intended for users involved in controlling processes and who are responsible for designing programs for programmable controllers. The manual describes the tasks that can be performed without using the STEP 7 software, such as determining the program sequence for a design project.

Scope of the Manual

This manual applies to the following CPUs of the S7-300 and S7-400:

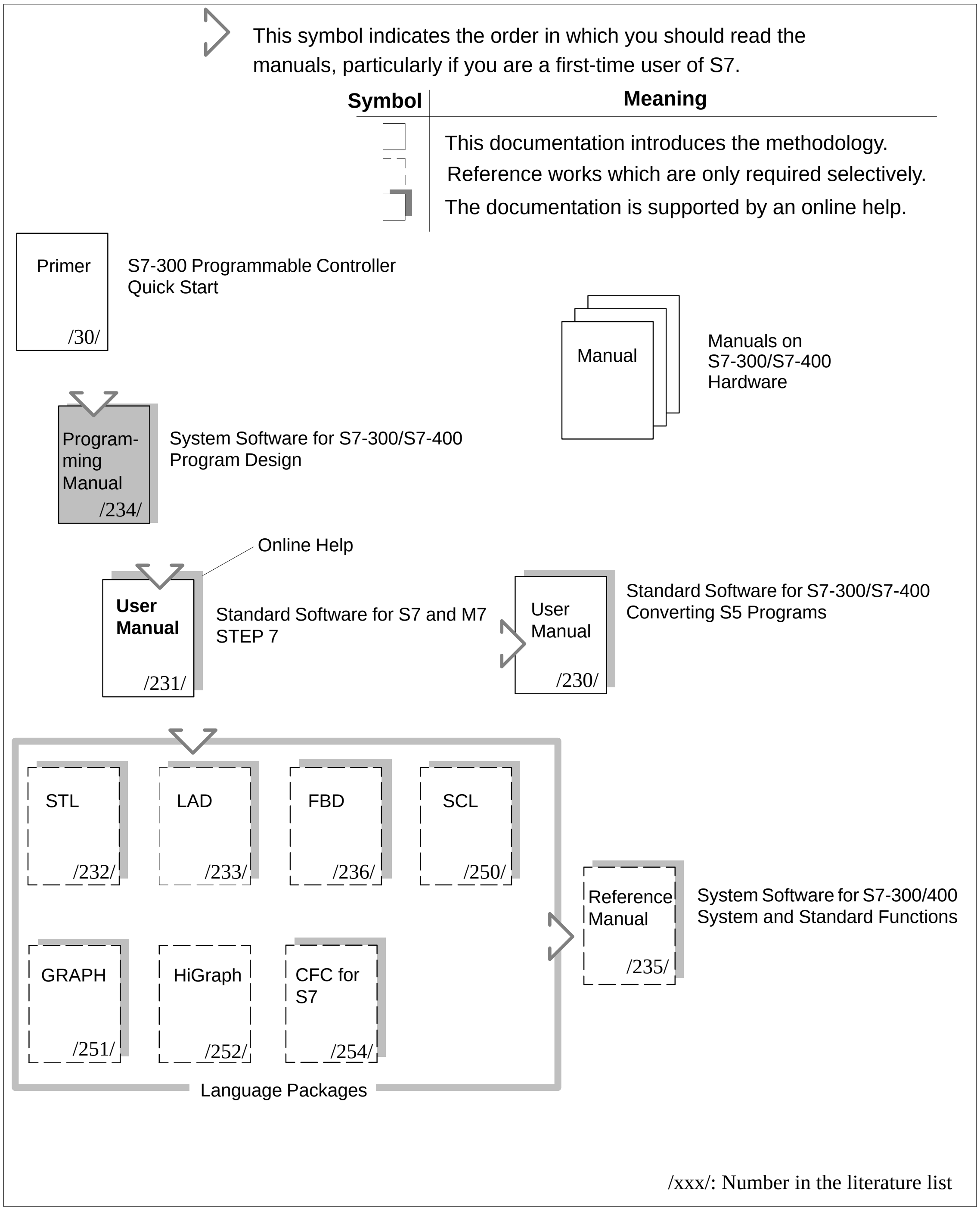
CPU	Order Number	Version (or higher)
CPU 312 IFM	6ES7312-5AC00-0AB0	03
CPU 313	6ES7313-1AD00-0AB0	01
CPU 314	6ES7314-1AE00-0AB0	04
CPU 314 IFM	6ES7312-5AE00-0AB0	01
CPU 315	6ES7314-1AF00-0AB0	03
CPU 315-2 DP	6ES7314-2AF00-0AB0	03
CPU 412-1	6ES7412-1XF00-0AB0	01
CPU 413-1	6ES7413-1XG00-0AB0	01
CPU 413-2	6ES7413-2XG00-0AB0	01
CPU 414-1	6ES7414-1XG00-0AB0	01
CPU 414-2 with 128K	6ES7414-2XG01-0AB0	01

CPU	Order Number	Version (or higher)
CPU 414-2 with 384K	6ES7414-2XJ00-0AB0	01
CPU 416-1	6ES7416-1XJ01-0AB0	01
CPU 416-2 with 0.8M	6ES7416-2XK00-0AB0	01
CPU 416-2 with 1.6M	6ES7416-2XL00-0AB0	01

The CPU functions described in this manual can be used from Version 3.1 or higher of the STEP 7 standard software.

Overview of the STEP 7 Documentation

There is a wide range of general and specific user documentation available to support you when configuring and programming an S7 programmable logic controller. The following tables and the figure below will help you find the user documentation you require.


This symbol indicates the order in which you should read the manuals, particularly if you are a first-time user of S7.
Symbol
Meaning
This documentation introduces the methodology.
Reference works which are only required selectively.
The documentation is supported by an online help.
Primer
/30/
S7-300 Programmable Controller Quick Start
Manual
Manuals on S7-300/S7-400 Hardware
Program-ming Manual
/234/
System Software for S7-300/S7-400 Program Design
Online Help
User Manual
/231/
Standard Software for S7 and M7 STEP 7
User Manual
/230/
Standard Software for S7-300/S7-400 Converting S5 Programs
STL /232/
LAD /233/
FBD /236/
SCL /250/
GRAPH /251/
HiGraph /252/
CFC for S7 /254/
Language Packages
Reference Manual
/235/
System Software for S7-300/400 System and Standard Functions
/xxx/: Number in the literature list

Title	Subject
S7-300 Programmable Controller Quick Start, Primer	The primer provides you with a very simple introduction to the methods of configuring and programming an S7-300/400. It is particularly suitable for first-time users of an S7 programmable controller.
S7-300 and S7-400 Program Design Programming Manual	The *"S7-300/400 Program Design"* programming manual provides you with the basic information you require about the structure of the operating system and a user program for an S7 CPU. First-time users of an S7-300/400 should use this manual to get a basic overview of programming methods on which to base the design of a user program.
S7-300 and S7-400 System and Standard Functions Reference Manual	The S7 CPUs have system functions and organization blocks integrated in the operating system that can be used when programming. The manual provides you with an overview of the system functions, organization blocks and loadable standard functions available with an S7 programmable controller and contains detailed interface descriptions explaining how to use the functions and blocks in your user program.
STEP 7 User Manual	The *"STEP 7" User Manual* explains the basic use and functions of the STEP 7 automation software. Whether you are a first-time user of STEP 7 or an experienced STEP 5 user, the manual will provide you with an overview of the procedures for configuring, programming and getting started with an S7-300/400 programmable controller. When working with the software, you can call up the online help which supports you with information about specific details of the program.
Converting S5 Programs Manual	You require the *"Converting S5 Programs" User Manual* if you want to convert existing S5 programs and to run them on S7 CPUs. The manual explains how to use the converter. The online help system provides more detailed information about using the specific converter functions. The online help system also includes an interface description of the available converted S7 functions.
STL, LAD, FBD, SCL[1] Manuals	The manuals for the language packages STL, LAD, FBD, and SCL contain both instructions for the user and a description of the language. To program an S7-300/400, you only require one of the languages, but you can, if required, mix the languages within a project. When using one of the languages for the first time, it is advisable to familiarize yourself with the methods of creating a program as explained in the manual. When working with the software, you can use the online help system which provides you with detailed information about using the editors and compilers.
GRAPH[1] , HiGraph[1], CFC[1] Manuals	The GRAPH, HiGraph, and CFC languages provide you with optional methods for implementing sequential control systems, status control systems, or graphical interconnection of blocks. The manuals contain both the user instructions and the description of the language. When using one of these languages for the first time, it is advisable to familiarize yourself with the methods of creating a program based on the *"S7-300 and S7-400 Program Design"* manual. When working with the software, you can also use the online help system (with the exception of HiGraph) that provides you with detailed information about using the editors and compilers.

1 Optional package for system software for S7-300/S7-400

Other Manuals

The various S7-300 and S7-400 CPUs, the S7-300 and S7-400 modules, and the instructions of the CPU are described in the following manuals:

- For the S7-300 programmable logic controller, refer to the manuals: Hardware and Installation (CPU Data, Module Data) and the Instruction List.
- For the S7-400 programmable logic controller, refer to the manuals: Hardware and Installation (CPU Data, Module Data) and the Instruction List.

How to Use this Manual

Since this manual provides you with a basic overview of the operating system of the S7-300/400, we recommend that you first have a look at the general contents of the chapters and then select the topics that you will require when designing your program for more intensive reading.

- Chapter 1 describes the basic tasks involved in planning an automation project.
- Chapter 2 shows you how to select the block structure for your S7 program.
- Chapters 3 and 4 describe the role of the organization blocks when the CPU executes your program.
- Chapters 5 and 6 describe the memory areas of the CPU and explain how the I/Os are addressed.
- Chapters 7 and 8 describe how you can exchange data between S7-CPUs and how you can adapt certain properties of a programmable logic controller by setting system parameters.
- Chapter 9 provides an overview of the operating modes and the various types of startup on the S7-CPUs. The chapter also explains how the operating system supports you when debugging your user program.
- Chapter 10 describes the multicomputing mode and the points to note when programming for this mode.
- Chapter 11 describes system diagnostics for S7-CPUs and explains how to eliminate errors and problems.
- Appendix A and Appendix B contain sample programs for an industrial blending process and for the data exchange using communication function blocks.
- Appendix C is a reference section listing data and parameter types.
- Appendix D contains the list of Literature referred to in the manual.
- The Glossary explains important terms used in the manual. The Index helps you to locate sections of text and topics quickly.

Conventions

References to other manuals and documentation are indicated by numbers in slashes /.../. These numbers refer to the titles of manuals listed in Appendix KEIN MERKER.

Additional Assistance

If you have any questions regarding the software described in this manual and cannot find an answer here or in the online help, please contact the Siemens representative in your area. You will find a list of addresses in the Appendix of **/70/** or /**100**/, or in catalogs, and in Compuserve (`go autforum`). You can also speak to our Hotline under the following phone or fax number:

Tel. (+49) (911) 895-7000 (Fax 7001)

If you have any questions or comments on this manual, please fill out the remarks form at the end of the manual and return it to the address shown on the form. We would be grateful if you could also take the time to answer the questions giving your personal opinion of the manual.

Siemens also offers a number of training courses to introduce you to the SIMATIC S7 automation system. Please contact your regional training center or the central training center in Nuremberg, Germany for details:

D-90327 Nuremberg, Tel. (+49) (911) 895-3154.

Contents

How to Design Control Programs 1

What Does This Chapter Describe?

This chapter outlines the basic tasks involved in planning an automation project and designing a user program for a programmable controller (PLC).

Based on an example of automating an industrial blending process, you are guided step by step through the procedure.

Where to Find More Information

The example of a program for an industrial blending process is described in Appendix A.

Chapter Overview

Section	Description	Page
1.1	Planning the Automation Project	1-2
1.2	Dividing the Process into Individual Tasks	1-3
1.3	Describing the Individual Tasks and Areas	1-5
1.4	Establishing the Safety Requirements	1-9
1.5	Describing the Required Operator Displays and Controls	1-10
1.6	Creating a Configuration Diagram	1-11

1.1 Planning the Automation Project

Overview

There are many ways of planning an automation project. This section describes a basic procedure that you can use for any project.

Figure 1-1 outlines the basic steps.

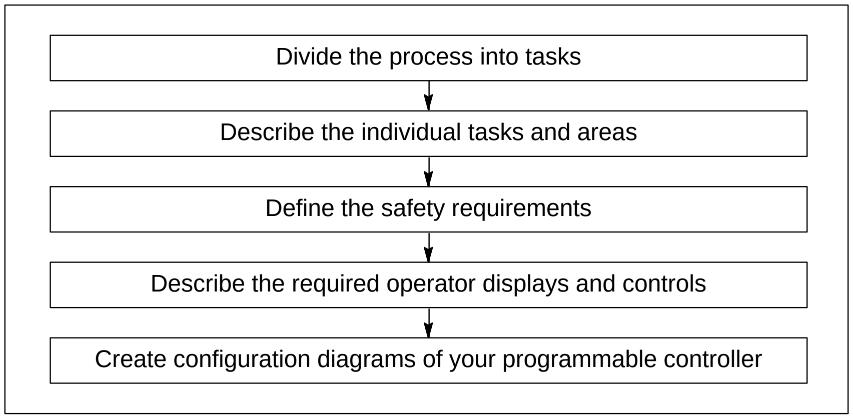

Divide the process into tasks
Describe the individual tasks and areas
Define the safety requirements
Describe the required operator displays and controls
Create configuration diagrams of your programmable controller

Figure 1-1 Basic Steps When Planning an Automation Project

The individual steps are described in detail in Sections 1.2 to 1.6.

1.2 Dividing the Process into Individual Tasks

Overview

A process consists of individual tasks. By identifying groups of related tasks within a process and then breaking these groups down into smaller tasks, even the most complex process can be defined.

The following example of an industrial blending system can be used to illustrate how to organize a process into functional areas and individual tasks. (see Figure 1-2).

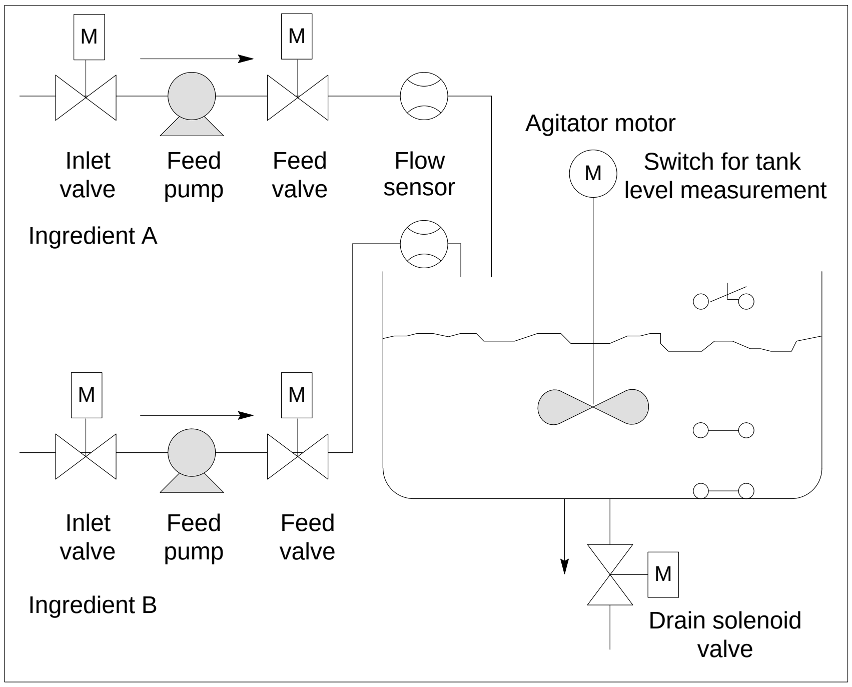


Figure 1-2 Example of an Industrial Blending Process

Identifying Areas and Tasks within the Process

After defining the process to be controlled, divide the project into related groups or areas (see Figure 1-3). As each group is divided into smaller tasks, the tasks required for controlling that part of the process become less complicated.

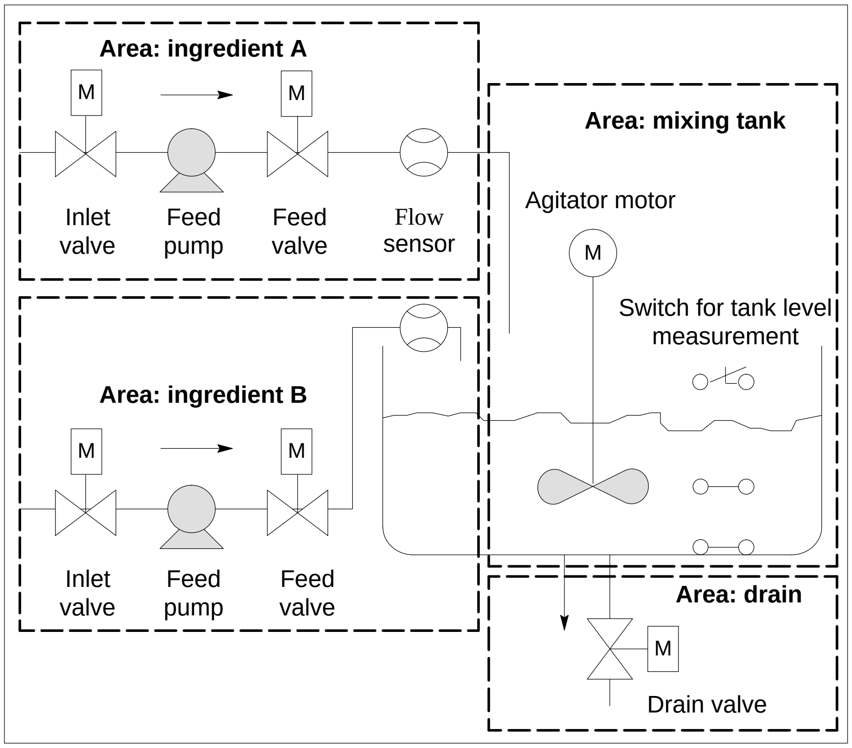


Figure 1-3 Defining Areas Within a Process

In our example of an industrial blending process, you can identify four distinct areas (see Table 1-1). In this example, the area for ingredient A contains the same equipment as the area for ingredient B.

Table 1-1 Functional Areas and Equipment in the Sample Process

Functional Area	Equipment Used
Ingredient A	Feed pump for ingredient A Inlet valve for ingredient A Feed valve for ingredient A Flow sensor for ingredient A
Ingredient B	Feed pump for ingredient B Inlet valve for ingredient B Feed valve for ingredient B Flow sensor for ingredient B
Mixing tank	Agitator motor Tank level switches
Drain	Drain valve

1.3 Describing the Individual Tasks and Areas

Overview

As you describe each area and task within your process, you define not only the operation of each area, but also the various elements that control the area. These include:

- Electrical, mechanical, and logical inputs and outputs for each task
- Interlocks and dependencies between the individual tasks

Describing How the Areas Function

The sample industrial blending process uses pumps, motors and valves. These must be described precisely to identify the operating characteristics and type of interlocks required during operation. Tables 1-2 to 1-6 provide examples of the description of the equipment used in an industrial blending process. When you have completed description, you could also use it to order the required equipment.

Table 1-2 Description of the Feed Pump Motors for Ingredients A and B

Ingredients A/B: Feed Pump Motors
1. The feed pump motors convey ingredients A and B to the mixing tank. – Flow rate: 400 l (100 gallons) per minute – Rating: 100 kW (134 hp) at 1200 rpm
2. The pumps are controlled (start/stop) from an operator station located near the mixing tank. The number of starts is counted for maintenance purposes. Both the counters and the display can be reset with one button.
3. The following conditions must be satisfied for the pumps to operate: – The mixing tank is not full. – The drain valve of the mixing tank is closed. – The emergency stop is not active.
4. The pumps are switched off if the following condition is satisfied: – The flow sensor signals no flow 7 seconds after the pump motor is started. – The flow sensor signals that the flow has ceased.

Table 1-3 Description of the Inlet and Feed Valves

Ingredients A/B: Inlet and Feed Valves
1. The inlet and feed valves for ingredients A and B allow or prevent the flow of the ingredients into the mixing tank. The valves have a solenoid with a spring return. – When the solenoid is activated, the valve is opened. – When the solenoid is deactivated, the valve is closed.
2. The inlet and feed valves are controlled by the user program.

Table 1-3 Description of the Inlet and Feed Valves, continued

Ingredients A/B: Inlet and Feed Valves
3. For the valves to be activated, the following condition must be satisfied: – The feed pump motor has been running for at least 1 second.
4. The valves are deactivated if the following condition is satisfied: – The flow sensor signals no flow.

Table 1-4 Description of the Agitator Motor

Agitator Motor
1. The agitator motor mixes ingredient A with ingredient B in the mixing tank. – Rating: 100 kW (134 hp) at 1200 rpm
2. The agitator motor is controlled (start/stop) from an operator station located near the mixing tank. The number of starts is counted for maintenance purposes. Both the counter and the display can be reset with one button.
3. To operate the agitator motor, the following conditions must be satisfied: – The tank level sensor is not signaling ”Tank Below Minimum”. – The drain valve of the mixing tank is closed. – The emergency stop is not active.
4. The agitator motor is switched off if the following condition is satisfied**:** – The tachometer does not indicate that the rated speed has been reached within 10 seconds of starting the motor.

Table 1-5 Description of the Drain Valve

Drain Valve
1. The drain valve allows the mixture to drain (using gravity feed) to the next stage in the process. The valve has a solenoid with a spring return. – If the solenoid is activated, the outlet valve is opened. – If the solenoid is deactivated, the outlet valve is closed.
2. The outlet valve is controlled (open/close) from an operator station.
3. The drain valve can be opened under the following conditions: – The agitator motor is off. – The tank level sensor is not signaling “Tank_empty”. – The emergency stop is not active.
4. The drain valve is switched off if the following condition is satisfied: – The tank level measurement indicates “Tank empty”.

Table 1-6 Description of the Mixing Tank Level Switches

Mixing Tank Level Switches
1. The switches in the mixing tank indicate the level in the tank and are used to interlock the feed pumps and the agitator motor.

Creating Input/Output Diagrams

After writing a physical description of each device to be controlled, draw diagrams of the inputs and outputs for each device or task area. (see Figure 1-4). These diagrams correspond to the logic blocks to be programmed.

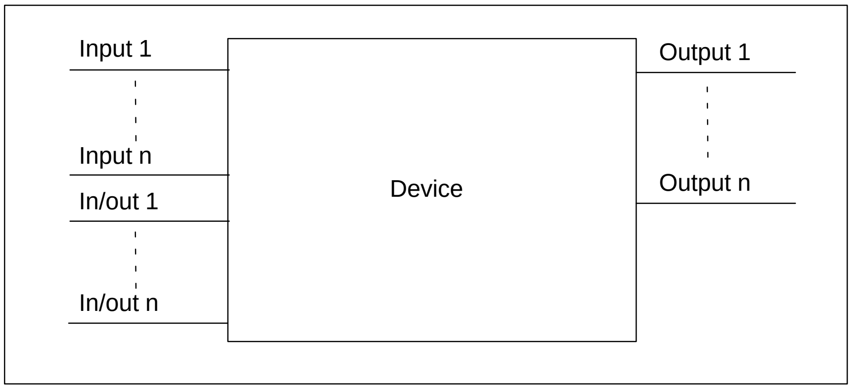


Figure 1-4 Input/Output Diagram

Creating an I/O Diagram for the Motor

In the example of the industrial blending process, two feed pumps and an agitator are used. The required motors are controlled by a “motor block” that is the same for all three devices. This block requires six inputs: two to start or stop the motor, one to reset the maintenance display, one for the motor response signal (motor running/not running), one for the time during which the response signal must be received, and one for the number of the timer used to measure the time.

The logic block also requires four outputs: two to indicate the operating state of the motor, one to indicate faults, and one to indicate that the motor is due for maintenance.

An in/out is also necessary to activate the motor. This is also processed or modified in the “motor block” program.

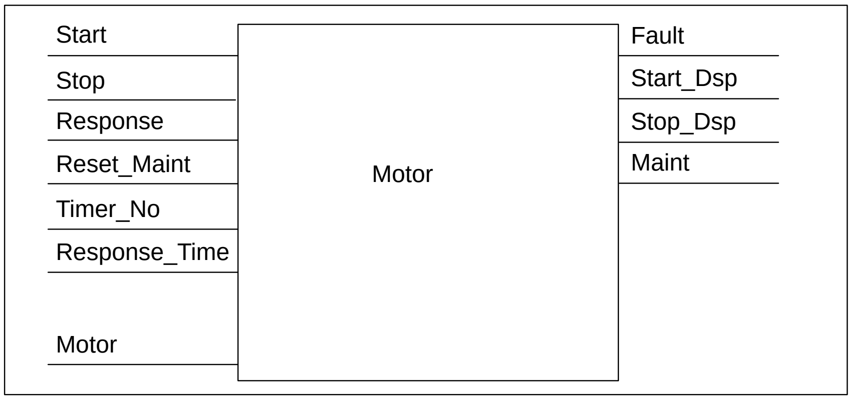


Figure 1-5 I/O Diagram of the Agitator Motor “Motor Block”

Creating an I/O Diagram for the Valves

Each valve is controlled by a “valve block” that is the same for all the valves used. The logic block has two inputs: one to open and one to close the valve. It also has two outputs: one to indicate that the valve is open and the other to indicated that it is closed.
The block has an in/out to activate the valve. This is also processed or modified in the “valve block” program.

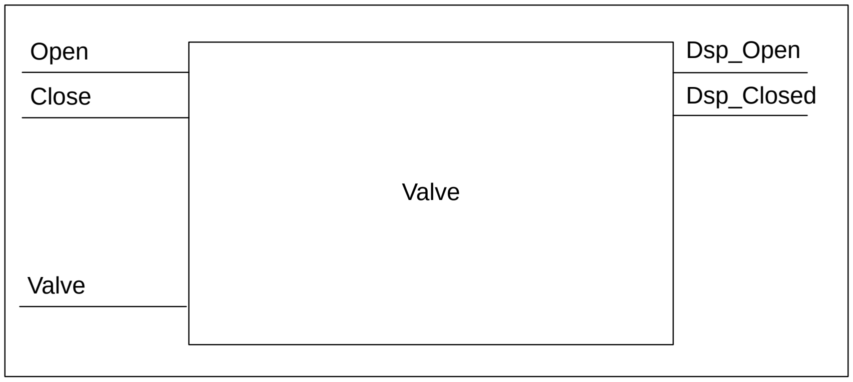


Figure 1-6 I/O Diagram of the Valves

1.4 Establishing the Safety Requirements

Overview

Decide which additional elements are needed to ensure the safety of the process, based on legal requirements and corporate policy. In your description, you should also include any influences that the safety elements have on your process areas.

Defining Safety Requirements

Find out which devices require hardwired circuits to meet safety requirements. By definition, these safety circuits operate independently of the programmable controller (although the safety circuit generally provides an I/O interface to allow coordination with the user program). Normally, you configure a matrix to connect every actuator with its own emergency off range. This matrix is the basis for the circuit diagrams of the safety circuits.

To design safety mechanisms, follow the steps outline below:

- Determine the logical and mechanical/electrical interlocks between the individual automation tasks.
- Design circuits to allow the devices belonging to the process to be operated manually in an emergency.
- Establish any further safety requirements for safe operation of the process.

Creating a Safety Circuit

The sample industrial blending process uses the following logic for its safety circuit:

- One Emergency Stop push button shuts down the following devices independent of the programmable controller (PLC):
  - Ingredient A feed pump
  - Ingredient B feed pump
  - Agitator motor
  - Valves
- The Emergency Stop push button is located on the operator station.
- An input to the controller indicates the state of the Emergency Stop push button.

1.5 Describing the Required Operator Displays and Controls

Overview

Every process needs an operator interface that allows human intervention in the process. Part of the design specification includes the design of the operator station.

Defining an Operator Station

In the industrial blending process described in our example, each device can be started or stopped by a push button located on the operator station. This operator station includes indicators to show the status of the operation (see Figure 1-7). The console also includes display lamps for devices that require maintenance after a certain number of starts and the emergency stop switch with which the process can be stopped immediately. The console also has a reset button for the maintenance display of the three motors. Using this, you can turn off the maintenance display lamps for the motors due for maintenance and reset the corresponding counters to 0.

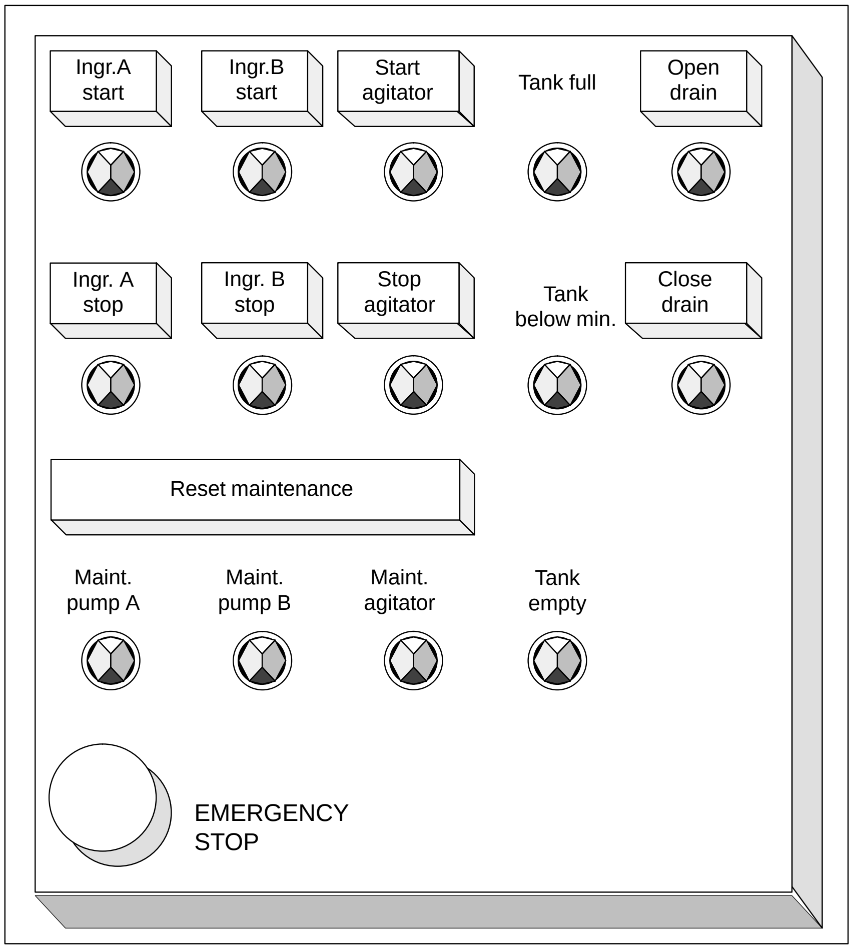


Figure 1-7 Example of an Operator Station Console

1.6 Creating a Configuration Diagram

Overview

After you have documented the design requirements, you must then decide on the type of control equipment required for the project.

Determining the PLC Configuration

By deciding which modules you want to use, you also specify the structure of the programmable controller. Create a configuration diagram specifying the following aspects:

- Type of CPU
- Number and type of I/O modules
- Configuration of the physical inputs and outputs

Figure 1-8 illustrates the configuration for the industrial blending process in our example.

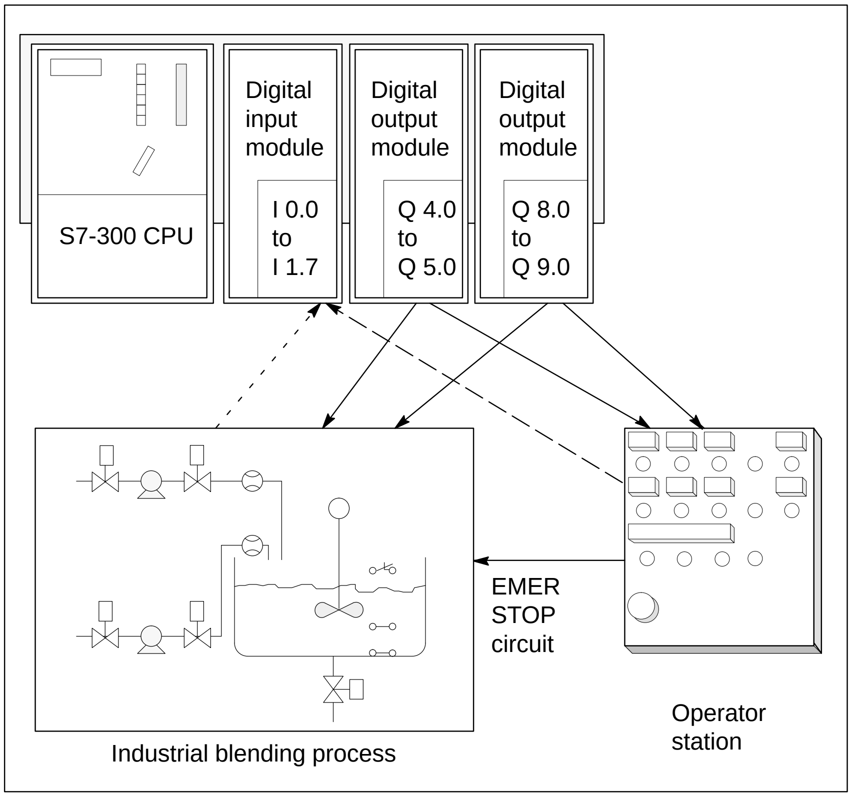


Figure 1-8 Example of an S7 Configuration Diagram

Structuring the User Program 2

What Does This Chapter Describe?

This chapter will help you when you are deciding on the block structure of your S7 program. It describes the following:

- The programs of a CPU: operating system and user program
- The structure of user programs
- The elements of a user program

Where to Find More Information

The reference manual **/235/** contains a detailed description of the individual organization blocks and system functions.

The Instruction Lists **/72/** and **/102/** contain an overview of the range of instructions of the S7-300 and S7-400 CPUs.

Chapter Overview

Section	Description	Page
2.1	The Programs in a CPU	2-2
2.2	Elements of the User Program	2-3
2.3	Call Hierarchy of the Blocks	2-4
2.4	Variables of a Block	2-5
2.5	Range of Instructions of the S7 CPUs	2-7
2.6	Organization Blocks (OB) and Program Structure	2-9
2.7	System Function Blocks (SFB) and System Functions (SFC)	2-10
2.8	Functions (FC)	2-11
2.9	Function Blocks (FB)	2-12
2.10	Instance Data Blocks	2-15
2.11	Shared Data Blocks (DB)	2-17
2.12	Saving the Data of an Interrupted Block	2-18
2.13	Avoiding Errors when Calling Blocks	2-20

2.1 The Programs in a CPU

Introduction

Two different types of program run on a CPU:

- The operating system
- The user program.

Operating System

Every CPU has an operating system that organizes all the functions and sequences of the CPU that are not associated with a specific control task. The tasks of the operating system include the following:

- Handling a complete restart and restart
- Updating the process image table of the inputs and outputting the process image table of the outputs
- Calling the user program
- Detecting interrupts and calling the interrupt OBs
- Detecting and dealing with errors
- Managing the memory areas
- Communicating with programming devices and other communications partners

If you change operating system parameters (the operating system default settings), you can influence the activities of the CPU in certain areas (see Chapter 8).

User Program

You yourself must create the user program and load it on the CPU. This contains all the functions required to process your specific automation task. The tasks of the user program include the following:

- Specifying the conditions for a complete restart and warm restart on the CPU (for example initializing signals with a particular value)
- Processing process data (for example logically combining binary signals, reading in and evaluating analog signals, specifying binary signals for output, outputting analog values)
- Specifying the reaction to interrupts
- Handling disturbances in the normal running of the program

2.2 Elements of the User Program

Overview

The STEP 7 programming software allows you to structure your user program, in other words to break down the program into individual, self-contained program sections. This has the following advantages:

- Extensive programs are easier to understand.
- Individual program sections can be standardized.
- Program organization is simplified.
- It is easier to make modifications to the program.
- Debugging is simplified since you can test separate sections.
- Commissioning your system is made much easier.

The example of an industrial blending process in Chapter 1 illustrated the advantages of breaking down an automation process into individual tasks. The program sections of a structured user program correspond to these individual tasks and are known as the blocks of a program.

An S7 user program consists of blocks, instructions and addresses. Table 2-1 provides you with an overview.

Table 2-1 Elements of a User Program

Element	Function	Refer to
Organization Blocks (OBs)	OBs determine the structure of the user program. • They form the interface between the operating system and the user program. • They control the startup of the programmable logic controller, the cyclic and interrupt-driven program execution and are responsible for handling errors.	Section 2.6, Chapters 3, 4, 11
System function blocks (SFBs) and system functions (SFCs)	These are standard, preprogrammed blocks that you do not need to program yourself. SFBs and SFCs are integrated in the S7 CPU. They can be called by the user program. Since these blocks are part of the operating system they do not need to be loaded as part of the program like other blocks.	Section 2.7, Chapters 7, 8
Functions (FCs) and function blocks (FBs)	These are logic blocks that you yourself must program. FBs are blocks with an associated memory area that is used to supply parameters. FCs are blocks that do not have an associated memory area for supplying parameters.	Sections 2.8, 2.9
Data blocks	These are data areas containing user data. There are two types of data block: • Instance data blocks that are assigned to an FB • Shared data blocks that can be accessed by all logic blocks	Sections 2.10, 2.11
Instructions of the S7 CPU	The CPUs provide you with instructions with which you can create blocks in various programming languages.	Section 2.5
Addresses	Memory and I/O areas of the S7 CPUs	Chapters 5, 6

2.3 Call Hierarchy of the Blocks

Introduction

Before the blocks in a user program can be processed, they must be called. These calls are special STEP 7 instructions known as block calls. You can only program block calls within logic blocks (OBs, FBs, FCs, SFBs and SFCs). They can be compared with jumps to a subroutine. Each jump means that you change to a different block. The return address in the calling block is saved temporarily by the system.

The order and nesting of the block calls is known as the call hierarchy. The number of blocks that can be nested, (the nesting depth) depends on the particular CPU.

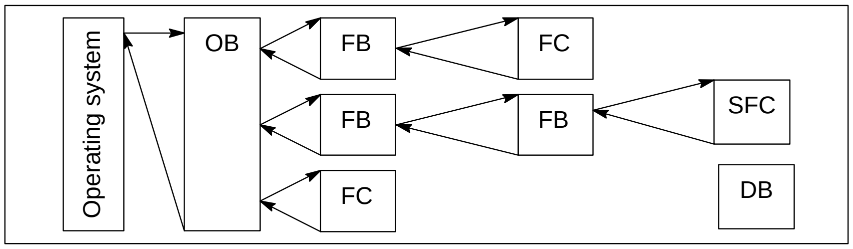


Figure 2-1 Example of the Call Hierarchy of a User Program

Block Calls

Figure 2-2 shows the sequence of a block call within a user program. The program calls the second block whose instructions are then executed completely. Once the second or called block has been executed, execution of the interrupted block that made the call is resumed at the operation following the block call.

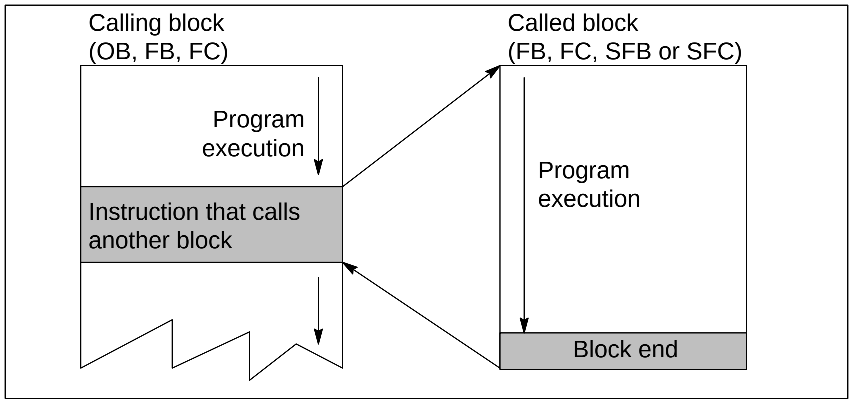


Figure 2-2 Calling a Block

Before you program a block, you must specify which data will be used by your program, in other words, you must declare the variables of the block.

2.4 Variables of a Block

Introduction

Apart from the instructions of the user program, blocks also contain block variables that you declare using STEP 7 when you program your own blocks. In the variable declaration, you can specify variables that the block will use when it is being executed. Variables are as follows:

- Parameters that are transferred between logic blocks.
- Static variables that are saved in an instance data block and are retained after the function block to which they belong has been executed.
- Temporary variables that are only available while the block is being executed and are then free to be overwritten when the block is completed. The operating system assigns a separate memory area for temporary data (see also Section 3.8 Local Data Stack).

Block Parameters

Since you can transfer parameters to blocks, you can create general, re-usable blocks whose programs can be used by other blocks in your program. There are two types of parameter as follows:

- Formal parameters that identify the parameters. These are specified in the variable declaration.
- Actual parameters that replace the formal parameters when the block is called.

For every formal parameter, you must specify a declaration type and a data type.

Declaration Types

You specify how a parameter is used by the logic block. You can define a parameter as an input value or output value. You can also use a parameter as an in/out variable that is transferred to the block and then output again by the block. Figure 2-3 shows the relationship of the formal parameters to an FB called “Motor”.

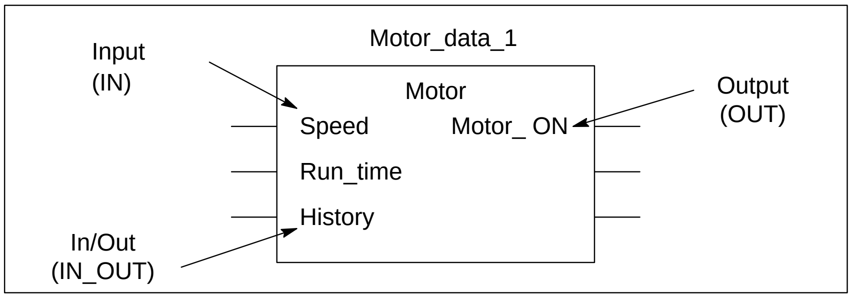


Figure 2-3 Defining the Input, Output and In/Out Parameters of a Logic Block

Table 2-2 describes the declaration types.

Table 2-2 Declaration Types for Parameters and Local Variables

Parameter/ Variable	Description	Permitted in
IN	Input parameter provided by the calling logic block.	FB, FC
OUT	Output parameter provided by the calling block.	FB, FC
IN_OUT	Parameter whose value is supplied by the calling block, modified by the called block and returned to the calling block.	FB, FC
STAT	Static variable that is saved in an instance DB.	FB
TEMP	Temporary variable that is saved temporarily in the local data stack. Once the logic block has been executed completely, the value of the variable is no longer available.	FB, FC, OB

With FBs, the data that was declared as IN, OUT, IN_OUT, and all static variables (STAT) are saved in the instance DB. Temporary variables of the type TEMP are not saved.

FCs cannot have any static variables. The input, output and in/out parameters are saved as pointers to the actual parameters made available by the calling block.

Data Types

All the data used in a user program must be identified by a data type. When you define the data type for parameters and static or temporary variables, you also specify the length and structure of the variables. The actual parameter supplied when the block is called must have the same data type as the formal parameter. Variables can have the following data types:

- Elementary data types that are provided by STEP 7
- Complex data types that you can create by combining elementary data types
- User-defined data types
- Parameter types that define special parameters that are transferred to FBs or FCs

Data types and parameter types are described in detail in Appendix C.

Initial Values

You can specify initial values for all parameters and static data. The value you select must be compatible with the data type. If you do not specify an initial value, a default value will be assigned depending on the data type of the variable.

2.5 Range of Instructions of the S7 CPUs

Overview

The STEP 7 programming software is the link between the user and the S7-300 and S7-400 programmable logic controllers. Using STEP 7, you can program your automation task in various programming languages.

The programming languages use the instructions provided by the S7 CPUs. The range of instructions is described in detail in the instruction lists of the CPUs, **/72/** and **/102/**. The instructions can be divided into the following groups:

- Block instructions
- Logic instructions (bit, word)
- Math instructions (integer, floating point)
- Comparison instructions
- Logic control instructions
- Load and transfer instructions
- Logarithmic and trigonometric instructions
- Shift and rotate instructions
- Conversion instructions
- Timer and counter instructions
- Jump instructions

Programming Languages

Table 2-3 shows the programming languages that are available and their most important characteristics. Which language you choose depends largely on your own experience and which language you personally find easiest to use.

Table 2-3 Programming Languages in STEP 7

Programming Language	User Group	Application	Incremental Input	Source-oriented Input	Block can be "Decompiled" from the CPU
Statement list STL	Users who prefer programming in a language similar to machine code	Programs optimized in terms of run time and memory requirements	yes	yes	yes
Ladder Logic LAD	Users who are accustomed to working with circuit diagrams	Programming logic controls	yes	no	yes
Function Block Diagram FBD	Users who are familiar with the logic boxes of Boolean algebra.	Programming logic controls	yes	no	yes

Table 2-3 Programming Languages in STEP 7, continued

Programming Language	User Group	Application	Incremental Input	Source-oriented Input	Block can be "Decompiled" from the CPU
SCL (Structured Control Language) Optional package	Users who have programmed in high-level languages such as PASCAL or C.	Programming data process tasks	no	yes	no
GRAPH Optional package	Users who want to work oriented on the technological functions without extensive programming or PLC experience.	Convenient description of sequential processes	yes	no	yes
HiGraph Optional package	Users who want to work oriented on the technological functions without extensive programming or PLC experience.	Convenient description of asynchronous, non-sequential processes	no	yes	no
CFC Optional package	Users who want to work oriented on the technological functions without extensive programming or PLC experience.	Description of continuous processes	no	yes 1)	no

1) But with syntax check when editing

For a detailed description of these programming languages, refer to the manuals **/232/**, **/233/**, **/236/**, **/250/**, **/251/**, **/252/** and **/254/**.

2.6 Organization Blocks (OB) and Program Structure

Definition

Organization blocks (OBs) are the interface between the operating system and the user program. They are called by the operating system and control cyclic and interrupt-driven program execution and how the programmable logic controller starts up. They also handle the response to errors. By programming the organization blocks you specify the reaction of the CPU.

Cyclic Program Execution

In most situations, the predominant type of program execution on programmable logic controllers is cyclic execution. This means that the operating system runs in a program loop (the cycle) and calls the organization block OB1 once each time the loop is executed. The user program in OB1 is therefore executed cyclically.

Interrupt-Driven Program Execution

Cyclic program execution can be interrupted by certain events (interrupts). If such an event occurs, the block currently being executed is interrupted at a command boundary and a different organization block that is assigned to the particular event is called. Once the organization block has been executed, the cyclic program is resumed at the point at which it was interrupted.

In SIMATIC S7, the following non-cyclic types of program execution are possible:

- Time-driven program execution
- Process interrupt-driven program execution
- Diagnostic interrupt-driven program execution
- Processing of synchronous and asynchronous errors
- Processing of the different types of startup
- Multicomputing-controlled program execution
- Background program execution

For more detailed information about program execution and the interrupt OBs, refer to Sections 3 and 4.

Linear Versus Structured Programming

You can write your entire user program in OB1 (linear programming). This is only advisable with simple programs written for the S7-300 CPU and requiring little memory.

Complex automation tasks can be controlled more easily by dividing them into smaller tasks reflecting the technological functions of the process (see Section 1.2) or that can be used more than once. These tasks are represented by corresponding program sections, known as the blocks (structured programming).

2.7 System Function Blocks (SFB) and System Functions (SFC)

Preprogrammed Blocks

You do not need to program every function yourself. S7 CPUs provide you with preprogrammed blocks that you can call in your user program.

System Function Blocks

A system function block (SFB) is a function block integrated on the S7 CPU. SFBs are part of the operating system and are not loaded as part of the program. Like FBs, SFBs are blocks “with memory”. You must also create instance data blocks for SFBs and load them on the CPU as part of the program.

S7 CPUs provide the following SFBs

- for communication on configured connections
- for integrated special functions (for example SFB29 “HS_COUNT” on the CPU 312 IFM and the CPU 314 IFM).

System Functions

A system function is a preprogrammed, tested function that is integrated on the S7 CPU. You can call the SFC in your program. SFCs are part of the operating system and are not loaded as part of the program. Like FCs, SFCs are blocks “without memory”.

S7-CPUs provide SFCs for the following functions:

- Copying and block functions
- Checking the program
- Handling the clock and run-time meters
- Transferring data records
- Transferring events from a CPU to all other CPUs in the multicomputing mode
- Handling time-of-day and time-delay interrupts
- Handling synchronous errors, interrupts and asynchronous errors
- System diagnostics
- Process image updating and bit field processing
- Addressing modules
- Distributed peripheral I/Os
- Global data communication
- Communication on non-configured connections
- Generating block-related messages

Additional Information

For more detailed information about SFBs and SFCs, refer to the reference manual **/235/**. The CPU descriptions **/70/** and **/101/** explain which SFBs and SFCs are available.

2.8 Functions (FC)

Definition

Functions (FCs) belong to the blocks that you program yourself. A function is a logic block “without memory”. Temporary variables belonging to the FC are saved in the local data stack. This data is then lost when the FC has been executed. To save data permanently, functions can also use shared data blocks.

Since an FC does not have any memory of its own, you must always specify actual parameters for it. You cannot assign initial values for the local data of an FC.

Application

An FC contains a program section that is always executed when the FC is called by a different logic block. You can use functions for the following purposes:

- To return a function value to the calling block (example: math functions)
- To execute a technological function (example: single control function with a bit logic operation).

Assigning Actual Parameters to Formal Parameters

You must always assign actual parameters to the formal parameters of an FC. The input, output and in/out parameters used by the FC are saved as pointers to the actual parameters of the logic block that called the FC.

2.9 Function Blocks (FB)

Definition

Function blocks FBs) belong to the blocks that you program yourself. A function block is a block “with memory”. It is assigned a data block as its memory (instance data block). The parameters that are transferred to the FB and the static variables are saved in the instance DB. Temporary variables are saved in the local data stack.

Data saved in the instance DB is not lost when execution of the FB is complete. Data saved in the local data stack is, however, lost when execution of the FB is completed.

Note

To avoid errors when working with FBs, read Section 2.13.

Application

An FB contains a program that is always executed when the FB is called by a different logic block. Function blocks make it much easier to program frequently occurring, complex functions.

FBs and Instance DBs

An instance data block is assigned to every function block call that transfers parameters.

By calling more than one instance of an FB, you can control more than one device with one FB. An FB for a motor type, can, for example, control various motors by using a different set of instance data for each different motor. The data for each motor (for example speed, ramping, accumulated operating time etc.) can be saved in one or more instance DBs (see also Section 2.10). Figure 2-4 shows the formal parameters of an FB that uses actual parameters saved in the instance DB.

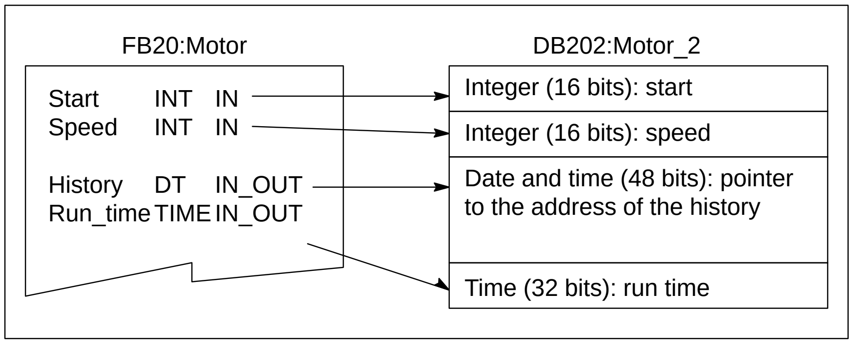


Figure 2-4 Relationship Between the Declarations of the FB and the Data of the Instance DB

Variables of the Data Type FB

If your user program is structured so that an FB contains calls for further already existing function blocks, you can include the FBs to be called as static variables of the data type FB in the variable declaration table of the calling FB. This technique allows you to nest variables and concentrate the instance data in one instance data block (multiple instance) see also Section 2.10.

Assigning Actual Parameters to Formal Parameters

It is not generally necessary in STEP 7 to assign actual parameters to the formal parameters of an FB. There are, however, exceptions to this. Actual parameters must be assigned in the following situations:

- For an in/out parameter of a complex data type (for example STRING, ARRAY or DATE_AND_TIME)
- For all parameter types (for example TIMER, COUNTER or POINTER)

STEP 7 assigns the actual parameters to the formal parameters of an FB as follows:

- *When you specify actual parameters in the call statement:* the instructions of the FB use the actual parameters provided.
- *When you do not specify actual parameters in the call statement:* the instructions of the FB use the value saved in the instance DB.

Table 2-4 shows which variables must be assigned actual parameters.

Table 2-4 Assigning Actual Parameters to the Formal Parameters of an FB

Variable	**Data Type**		
	Elementary Data Type	**Complex Data Type**	**Parameter Type**
Input	No parameter required	No parameter required	Actual parameter required
Output	No parameter required	No parameter required	Actual parameter required
In/out	No parameter required	Actual parameter required	–

Assigning Initial Values to Formal Parameters

You can assign initial values to the formal parameters in the declaration section of the FB. These values are written into the instance DB assigned to the FB.

If you do not assign actual parameters to the formal parameters in the call statement, STEP 7 uses the values saved in the instance DB. These values can also be the initial values that were entered in the variable declaration table of an FB.

Table 2-5 shows which variables can be assigned an initial value. Since the temporary data are lost after the block has been executed, you cannot assign any values to them.

Table 2-5 Assigning Initial Values to the Variables of an FB

Variable	**Data Type**		
	Elementary Data Type	**Complex Data Type**	**Parameter Type**
Input	Initial value permitted	Initial value permitted	–
Output	Initial value permitted	Initial value permitted	–
In/out	Initial value permitted	–	–
Static	Initial value permitted	Initial value permitted	–
Temporary	–	–	–

2.10 Instance Data Blocks

Definition

An instance data block is assigned to every function block call that transfers parameters. The actual parameters and the static data of the FB are saved in the instance DB. The variables declared in the FB determine the structure of the instance data block.

Instance means a function block call. If, for example, a function block is called five times in the S7 user program, there are five instances of this block.

Creating an Instance DB

Before you create an instance data block, the corresponding FB must already exist. You specify the number of the FB when you create the instance data block.

An Instance Data Block for Every Instance

If you assign several instance data blocks to a function block (FB) that controls a motor, you can use this FB to control different motors.

The data for each specific motor (for example speed, run-up time, total operating time) are saved in different data blocks. The DB assigned to the FB when it is called determines which motor is controlled. With this technique, only one function block is necessary for several motors (see Figure 2-5).

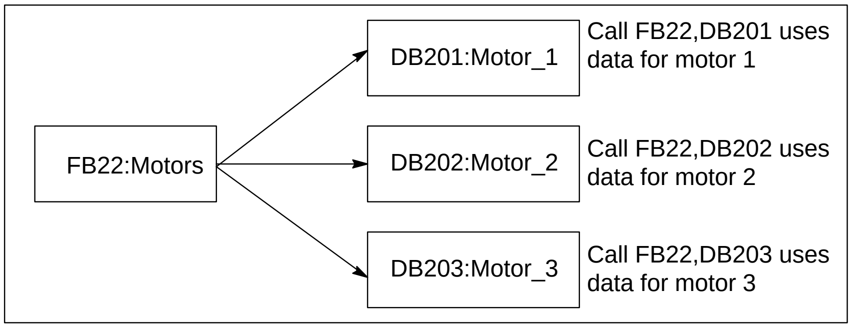

FB22:Motors
DB201:Motor_1
DB202:Motor_2
DB203:Motor_3
Call FB22,DB201 uses data for motor 1
Call FB22,DB202 uses data for motor 2
Call FB22,DB203 uses data for motor 3

Figure 2-5 Using an Instance DB for Each Separate Instance

One Instance DB for Several Instances of an FB

You can also transfer the instance data for several motors at the same time in one instance DB. To do this, you must program the calls for the motor controllers in a further FB and declare static variables with the data type FB for the individual instances (multiple instances) in the declaration section of the calling FB.

If you use one instance DB for several instances of an FB, you save memory and optimize the use of data blocks.

In Figure 2-6, the calling FB is FB21 “Motor processing”, the variables are of data type FB22 and the instances are identified by Motor_1, Motor_2 and Motor_3. In this example, FB22 does not need its own instance data block, since its instance data are saved in the instance data block of the calling FB.

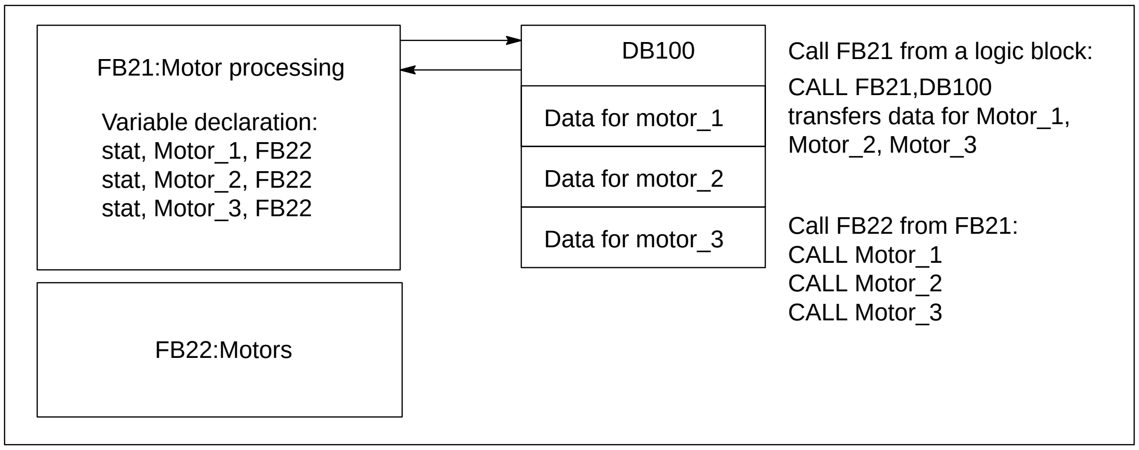


Figure 2-6 Using an Instance DB for Several Instances

One Instance DB for Several Instances of Different FBs

In a function block, you can call the instances of other existing FBs. The example in Figure 2-7 shows the assigned instance data, once again saved in a common instance DB.

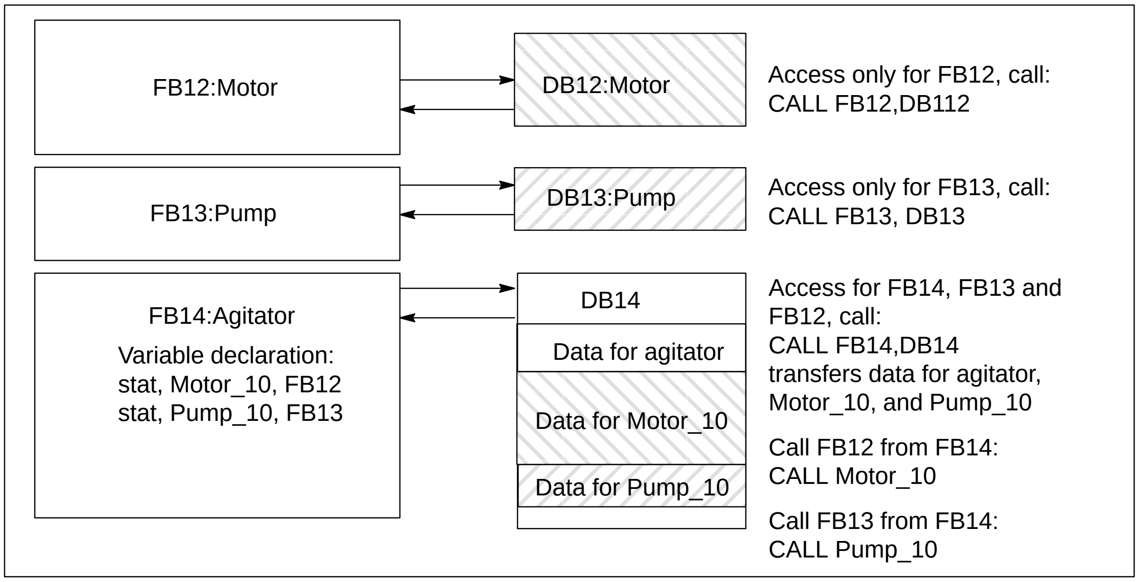


Figure 2-7 Using one Instance DB for Several Instances of Different FBs

2.11 Shared Data Blocks (DB)

Definition

In contrast to logic blocks, data blocks do not contain STEP 7 instructions. They are used to store user data, in other words, data blocks contain variable data with which the user program works. Shared data blocks are used to store user data that can be accessed by all other blocks.

The size of DBs can vary. Refer to the description of your CPU for the maximum possible size (**/70/** and **/101/**).

Structure

You can structure shared data blocks in any way to suit your particular requirements.

Shared Data Blocks in the User Program

If a logic block (FC, FB or OB) is called, it can occupy space in the local data area (L stack) temporarily. In addition to this local data area, a logic block can open a memory area in the form of a DB. In contrast to the data in the local data area, the data in a DB are not deleted when the DB is closed, in other words, after the corresponding logic block has been executed.

Each FB, FC or OB can read the data from a shared DB or write data to a shared DB. This data remains in the DB after the DB is exited.

A shared DB and an instance DB can be opened at the same time. Figure 2-8 shows the different methods of access to data blocks.

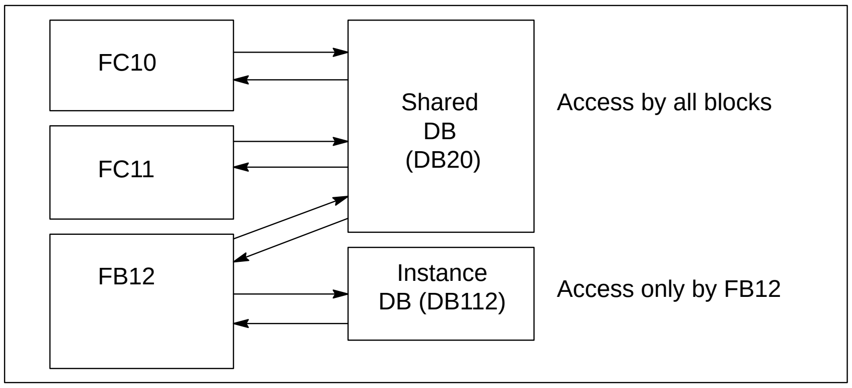


Figure 2-8 Access to Shared DBs and Instance DBs

2.12 Saving the Data of an Interrupted Block

Overview

The CPU has a “block stack” (B stack) for saving information belonging to a logic block that has been interrupted. Using this data, the user program can then be resumed after the interrupt. When one of the following events occurs, block information is saved in the B stack:

- When a different block is called within a CPU program.
- When a block is interrupted by a higher priority class (for more detailed information about priority classes, refer to Chapter 3).

Block Stack

The block stack (B stack) is a memory area in the system memory of the CPU (see also Chapter 5). If the execution of a block is interrupted by a call for a different block, the following data is saved in the B stack:

- Number, type (OB, FB, FC, SFB, SFC) and return address of the block that was interrupted.
- Numbers of the data blocks (from the DB and DI register) that were open when the block was interrupted.

If the CPU is in the STOP mode, you can display the B stack with STEP 7 on a programming device. The B stack lists all the blocks that had not been completely executed when the CPU changed to the STOP mode. The blocks are listed in the order in which they were called in the program (see Figure 2-9).

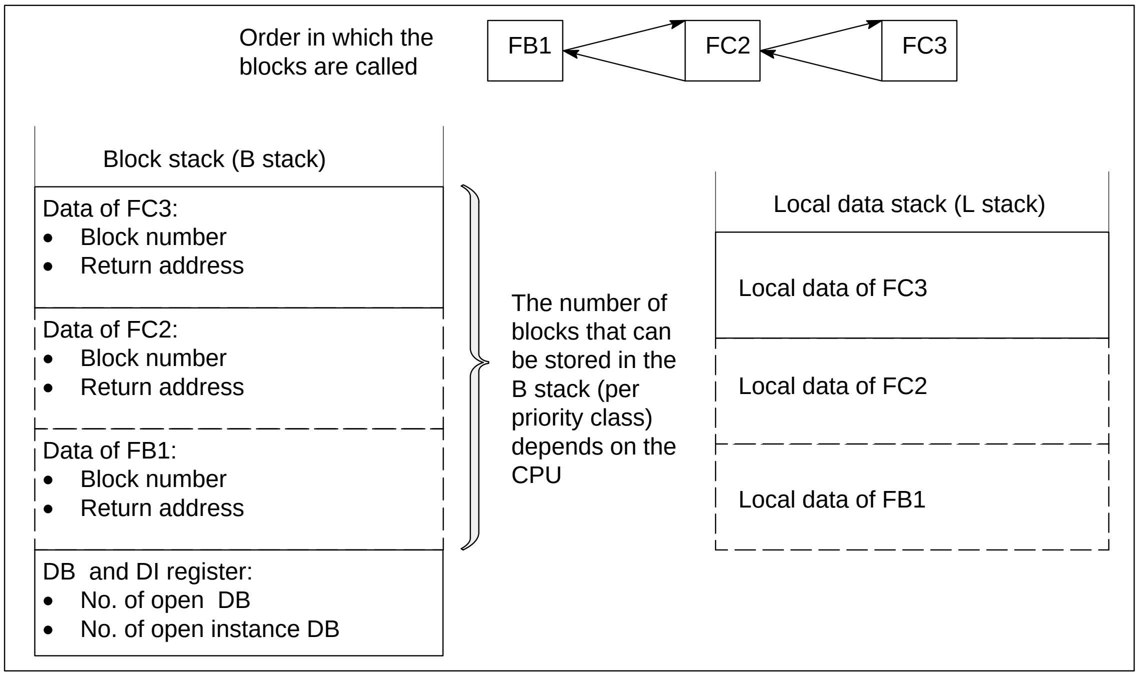


Figure 2-9 Information in the B Stack and L Stack

Local Data Stack

The local data stack (L stack) is a memory area in the system memory of the CPU. The L stack saves the temporary variables (local data) of the block (see also Section 3.8).

Note

The L stack not only saves the temporary data of a block but also provides additional memory space, for example, for transferring parameters.

Data Block Registers

There are two data block registers. These contain the numbers of opened data blocks, as follows:

- The DB register contains the number of the open shared data block
- The DI register contains the number of the open instance data block.

2.13 Avoiding Errors when Calling Blocks

STEP 7 Overwrites Data in the DB Register

STEP 7 modifies the registers of the S7-300/S7-400 CPU when various instructions are executed. The contents of the DB and DI registers are, for example, swapped when you call an FB. This allows the instance DB of the called FB to be opened without losing the address of the previous instance DB.

If you work with absolute addressing, errors can occur accessing data saved in the registers. In some cases, the addresses in the register AR1 (address register 1) and in the DB register are overwritten. This means that you could read or write to the wrong addresses.

Warning

Risk of personal injury or damage to equipment

The following programming techniques can cause the contents of the DB registers (DB and DI), the address register (AR1 and AR2), and the accumulators (ACCU1 and ACCU2) to be modified:

- CALL FC, CALL FB, CALL multiple instance
- Accessing a DB using the complete absolute address (for example DB20.DBW10)
- Accessing variables of a complex data type

In addition, you cannot use the RLO bit of the status word as an additional (implicit) parameter when you call an FB or FC.

When using the programming techniques mentioned above, you must make sure that you save and restore the contents yourself; otherwise errors may occur.

Saving Correct Data

The contents of the DB register can cause critical situations if you access the absolute addresses of data using the abbreviated format. If, for example, you assume that DB20 is open (and that its number is saved in the DB register), you can specify DBX0.2 to access the data in bit 2 of byte 0 of the DB whose address is entered in the DB register (in other words DB20). If, however, the DB register contains a different DB number you access the wrong data.

You can avoid errors when accessing data of the DB register by using the following methods to address data:

- Use the symbolic address
- Use the complete absolute address (for example *DB20.DBX0.2*)

If you use these addressing methods, STEP 7 automatically opens the correct DB. If you use the AR1 register for indirect addressing, you must always load the correct address in AR1.

Situations in which Data is Overwritten

In the following situations, the contents of the address register AR1 and the DB register of the calling block are overwritten:

- When an FB is called, AR1 and the DB register of the calling block are overwritten.
- After a call for an FC that transfers a parameter with a complex data type (for example STRING, DATE_AND_TIME, ARRAY, STRUCT or UDT), the content of AR1 and the DB register of the calling block are overwritten.
- After you have assigned an actual parameter located in a DB to an FC (for example *DB20.DBX0.2*), STEP 7 opens the DB (DB20) by overwriting the content of the DB register.

In the following situations, the contents of the address register AR1 and the DB register of the called block are overwritten:

- After an FB has addressed an in/out parameter with a complex data type (for example STRING, DATE_AND_TIME, ARRAY, STRUCT or UDT), STEP 7 uses the address register AR1 and the DB register to access data. This overwrites the contents of both registers.
- After an FC has addressed a parameter (input, output or in/out) with a complex data type (for example STRING, DATE_AND_TIME, ARRAY, STRUCT or UDT), STEP 7 uses the address register AR1 and the DB register to access data. This overwrites the contents of both registers.

When using function blocks, remember the following points:

- When calling an FB and a multiple instance, the address register AR2 is written.
- If the address register AR2 is overwritten while an FB is being executed, the correct execution of this FB can no longer be guaranteed.

Note

There are also other situations in addition to those listed above in which data are overwritten.

Organization Blocks and Executing the Program 3

What Does This Chapter Describe?

This chapter provides an overview of the following topics:

- Types of organization block
- Cyclic program execution
- Interrupt-driven program execution

Where to Find More Information

You will find more detailed information about interrupt-driven program execution in Chapter 4. Error OBs are described in greater detail in Chapter 11.

For a detailed description of the individual organization blocks, refer to the reference manual **/235/**.

Chapter Overview

3.1 Types of Organization Block

Overview

STEP 7 provides you with various types of organization block (OB) with which you can adapt the program to the requirements of your process, as follows.

- Using the startup OBs, you can decide the conditions under which the programmable logic controller goes through a complete restart or a restart.
- With some of the OBs, you can execute a program at a certain point in time or at certain intervals.
- Other OBs react to interrupts or errors detected by the CPU.

Priority

Organization blocks determine the order in which the individual program sections are executed. The execution of an OB can be interrupted by calling a different OB. Which OB is allowed to interrupt another OB depends on its priority. Higher priority OBs can interrupt lower priority OBs. The lowest priority is 1. The background OB has the lowest priority, namely 0.29.

Types of Interrupt and Organization Blocks

The events that lead to an OB being called are known as interrupts. Table 3-1 shows the types of interrupt in STEP 7 and the priority of the organization blocks assigned to them. Not all S7 CPUs have the complete range of organization blocks and priority classes listed in the table below (see CPU descriptions **/70/** and **/101/**).

Table 3-1 Types of Interrupt and Priority Classes

Type of Interrupt	Organization Blocks	Priority Class
Main program scan	OB1	1
Time-of-day interrupts	OB10 to OB17	2
Time-delay interrupts	OB20 OB21 OB22 OB23	3 4 5 6
Cyclic interrupts	OB30 OB31 OB32 OB33 OB34 OB35 OB36 OB37 OB38	7 8 9 10 11 12 13 14 15

Table 3-1 Types of Interrupt and Priority Classes

Type of Interrupt	Organization Blocks	Priority Class
Hardware interrupts	OB40 OB41 OB42 OB43 OB44 OB45 OB46 OB47	16 17 18 19 20 21 22 23
Multicomputing interrupt	OB60 Multicomputing	25
Asynchronous error interrupts	OB80 Time error OB81 Power supply error OB82 Diagnostic interrupt OB83 Insert/remove module interrupt OB84 CPU hardware error OB85 Priority class error OB86 Rack failure OB87 Communication error	26 (or 28 if the asynchronous error OB exists in the startup program)
Background cycle	OB90	29[1)]
Startup	OB100 Complete restart OB101 Restart	27 27
Synchronous error interrupts	OB121 Programming error OB122 Access error	Priority of the OB that caused the error

1) The priority class 29 corresponds to priority 0.29. The background cycle has a lower priority than the main program cycle.

Changing the Priority

The priority of organization blocks on S7-300 CPUs is fixed. With S7-400 CPUs, you can change the priority of the organization blocks OB10 through OB47 and the priority in the RUN mode of organization blocks OB81 through OB87 with STEP 7. Priority classes 2 through 23 are permitted for OB10 through OB87 and priority classes 24 through 26 for OB81 through OB87. You can assign the same priority to several OBs. OBs with the same priority are processed in the order in which their start events occur.

Start Information of an OB

Every organization block has a start information field of 20 bytes that is transferred by the operating system when the OB is started. The start information specifies the start event of the OB, the date and time of the OB start, errors that have occurred, and diagnostic events.

For example, OB40, a hardware interrupt OB, contains the address of the module that caused the interrupt in its start information.

3.2 Organization Blocks for the Startup Program

Types of Startup

There are two different types of startup: COMPLETE RESTART and RESTART (see also Chapter 9). S7-300 CPUs only have the COMPLETE RESTART type.

During startup, the operating system calls the appropriate startup OB, as follows:

- In a complete restart, the complete restart OB (OB100)
- In a restart, the restart OB (OB101).

Start Events

The startup OBs are started following the following events:

- POWER UP
- after switching from the STOP mode to the RUN mode
- when a complete restart or restart is triggered on the programming device or using communication functions

Whether or not the complete restart or restart OB is called depends on the type of startup specified during parameter assignment for manual and automatic startup, the setting of the startup switch CRST/WRST, and whether the mode selector has been set to POWER OFF (see also Section 8.3).

Startup Program

You can specify the conditions for starting up your CPU (initialization values for RUN, startup values for I/O modules) by writing your program for the startup in the organization blocks OB100 for complete restart or OB101 for a restart.

There are no restrictions to the length of the startup program and no time limit since the cycle monitoring is not active. Time- or interrupt-driven execution is not possible in the startup program. During the start up, all digital outputs have the signal state 0.

3.3 Organization Block for Cyclic Program Execution

Introduction

Cyclic program execution is the “normal” type of program execution on programmable logic controllers. The operating system calls OB1 cyclically and with this call it starts cyclic execution of the user program.

Sequence of Program Execution

Figure 3-1 illustrates the phases of cyclic program execution:

- The operating system starts the cycle monitoring time.
- The CPU reads the state of the inputs of the input modules and updates the process image table of the inputs.
- The CPU processes the user program and executes the instructions contained in the program.
- The CPU writes the values from the process image table of the outputs to the output modules.
- At the end of a cycle, the operating system executes any tasks that are pending, for example loading and deleting blocks, receiving and sending global data (see Chapter 7).

Finally, the CPU returns to the start of the cycle and restarts the cycle monitoring time.

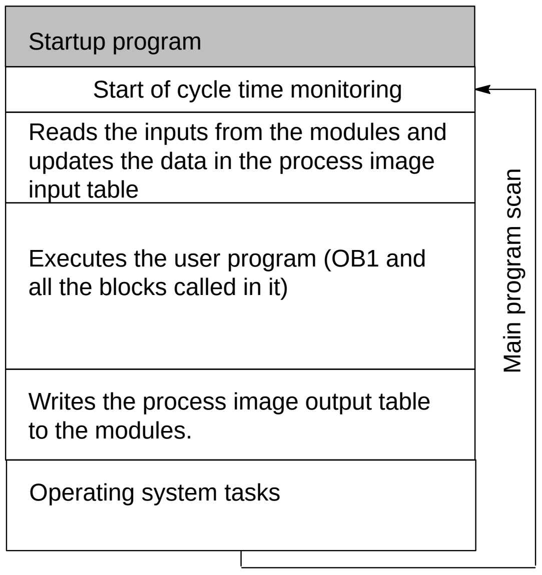

Startup program
Start of cycle time monitoring
Reads the inputs from the modules and updates the data in the process image input table
Executes the user program (OB1 and all the blocks called in it)
Writes the process image output table to the modules.
Operating system tasks
Main program scan

Figure 3-1 Main Program Scan

Process Images

So that the CPU has a consistent image of the process signals during cyclic program execution, the CPU does not address the input (I) and output (Q) address areas directly on the I/O modules but rather accesses an internal memory area of the CPU that contains an image of the inputs and outputs.

Programming Cyclic Program Execution

You program cyclic program execution by writing your user program in OB1 and in the blocks called within OB1 using STEP 7.

Start Event

Cyclic program execution begins as soon as the startup program is completed without errors.

Interrupts

Cyclic program execution can be interrupted by the following:

- An interrupt
- A STOP command (mode selector, menu option on the programming device, SFC 46 STP, SFB 20 STOP)
- A power outage
- The occurrence of a fault or program error

Cycle Time

The cycle time is the time required by the operating system to run the cyclic program and all the program sections that interrupt the cycle (for example, executing other organization blocks) and system activities (for example, updating the process image). The cycle time (T_C) is not the same in every cycle (see also Section 8.4).

Figure 3-2 illustrates different cycle times ($T_{C1} \neq T_{C2}$). In the current cycle, OB1 is interrupted by a time-of-day interrupt.

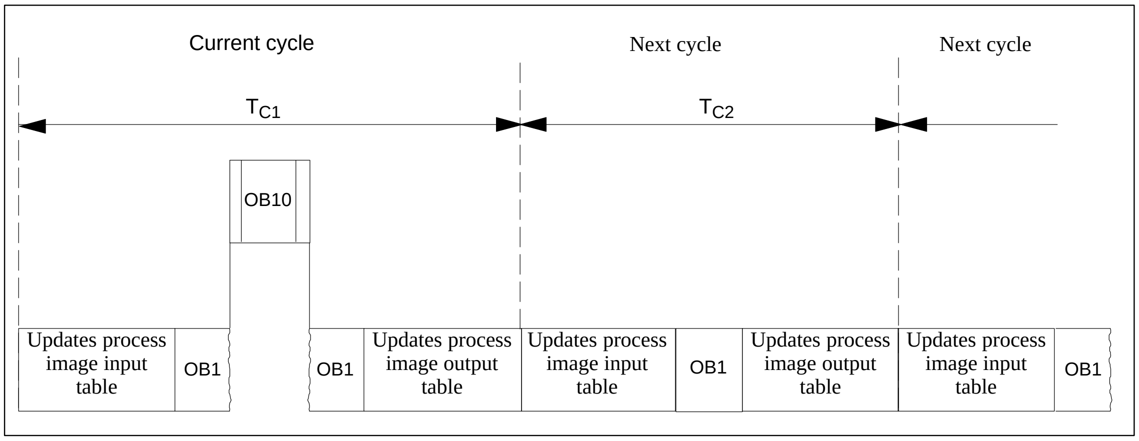

Current cycle
Next cycle
Next cycle
TC1
TC2
OB10
Updates process image input table
OB1
OB1
Updates process image output table
Updates process image input table
OB1
Updates process image output table
Updates process image input table
OB1

Figure 3-2 Cycle Times of Different lengths

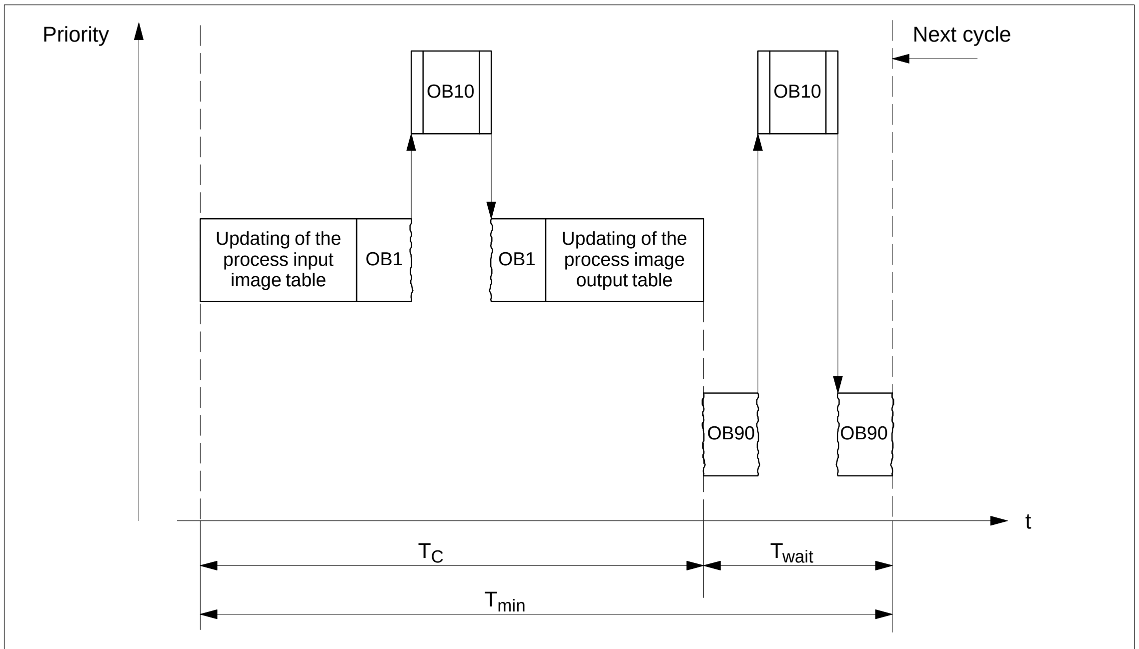
3.4 Organization Block for Background Program Execution

Description

If you have specified a minimum scan cycle time with STEP 7 and this is longer than the actual scan cycle time (see Section 3.3), the CPU still has processing time available at the end of the cyclic program. This time is used to execute the background OB. If OB90 does not exist on your CPU, the CPU waits until the specified minimum scan cycle time has elapsed.

Priority

The background OB has priority class 29, which corresponds to priority 0.29. It is therefore the OB with the lowest priority. Its priority class cannot be changed by reassigning parameters.


Priority
Next cycle
OB10
OB10
Updating of the process input image table
OB1
OB1
Updating of the process image output table
OB90
OB90
t
T_C
T_{wait}
T_{min}

T_C:	the actual cycle time required for a main program cycle
T_{min}:	the minimum cycle time specified with STEP 7
T_{wait}:	the time available before the start of the next cycle

Figure 3-3 Example of the Background Cycle, the Main Program Cycle, and OB10

Programming OB90

The run time of OB90 is not monitored by the CPU operating system so that you can program loops of any length in OB90.
Make sure that the data you use in the background program are consistent by taking the following into account in your program:

- The reset events of OB90 (see Reference Manual **/235/**)
- The asynchronous process image table updating of OB90

3.5 Organization Blocks for Interrupt-Driven Program Execution

Overview

STEP 7 provides different types of OBs that can interrupt OB1 at certain intervals or when certain events occur.

You can configure these OBs either with STEP 7 or by programming a system function (SFC). For more detailed information about configuring OBs, refer to the STEP 7 user manual **/231/**. For more detailed information about SFCs, refer to the reference manual **/235/**.

Non-Cyclic Program Execution

With STEP 7, you can select parts of your user program that do not need to be executed cyclically and only execute these sections when the situation makes it necessary. The user program can be divided up into “subroutines” and distributed in different organization blocks. If you want your user program to react to an important signal that seldom occurs (for example a limit value sensor indicates that a tank is full), the section of program to be executed when this signal is output can be written in an OB that is not executed cyclically.

Apart from cyclic program execution, STEP 7 provides the following types of program execution:

- Time-driven program execution
- Hardware interrupt-driven program execution
- Diagnostic interrupt-driven program execution
- Multicomputing interrupt-driven program execution
- Error handling

Table 3-2 Organization Blocks that Can Interrupt OB1

Types of OB	Start Events
Time-of-day interrupt OBs (OB10 to OB17)	Date, time-of-day
Time-delay OBs (OB20 to OB23)	Delay time after programmed events
Cyclic interrupt OBs (OB30 to OB38)	Intervals (1 ms to 1 minute)
Hardware interrupt OBs (OB40 to OB47)	Process signal from an I/O module to the CPU or interrupt from a function module
Synchronous error OBs (OB121 and OB122)	Errors in the user program (programming errors and access errors)
Asynchronous error OBs (OB80 to OB87)	Priority class errors or faults on the PLC.
Multicomputing interrupt OB(OB60)	SFC35 call

Masking Start Events

With system functions (SFCs), you can mask, delay or disable the start events for several OBs. Refer to Table 3-3.

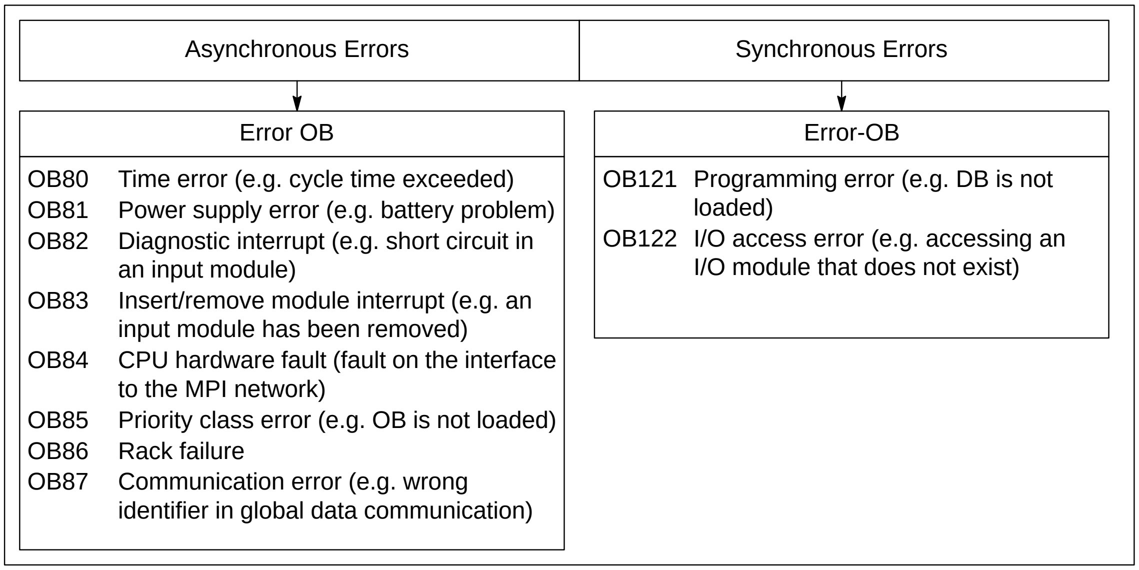
3.6 Organization Blocks for Handling Errors

Types of Errors

The errors that can be detected by the S7 CPUs and to which you can react with the help of organization blocks can be divided into two basic categories:

- Synchronous errors: these errors can be assigned to a specific part of the user program. The error occurs during the execution of a particular instruction. If the corresponding synchronous error OB is not loaded, the CPU changes to the STOP mode when the error occurs.
- Asynchronous errors: these errors cannot be directly assigned to the user program being executed. These are priority class errors or faults on the programmable logic controller (for example a defective module). If the corresponding asynchronous error OB is not loaded, the CPU changes to the STOP mode when the error occurs (exception OB81).

Figure 3-4 shows the two categories of error OBs and describes the types of errors that can occur.



Error OB	
OB80	Time error (e.g. cycle time exceeded)
OB81	Power supply error (e.g. battery problem)
OB82	Diagnostic interrupt (e.g. short circuit in an input module)
OB83	Insert/remove module interrupt (e.g. an input module has been removed)
OB84	CPU hardware fault (fault on the interface to the MPI network)
OB85	Priority class error (e.g. OB is not loaded)
OB86	Rack failure
OB87	Communication error (e.g. wrong identifier in global data communication)

Error-OB	
OB121	Programming error (e.g. DB is not loaded)
OB122	I/O access error (e.g. accessing an I/O module that does not exist)

Figure 3-4 Types of Error

Using DBs for Synchronous Errors

Synchronous errors occur during the execution of a particular instruction. When these errors occur, the operating system makes an entry in the I stack and starts the OB for synchronous errors.

The error OBs called as a result of synchronous errors are executed as part of the program in the same priority class as the block that was being executed when the error was detected. OB121 and OB122 can therefore access the values in the accumulators and other registers as they were at the time when the interrupt occurred. You can use these values to react to an error and then return to your normal program (for example, if an access error occurs on an analog input module, you can specify a substitute value in OB122 using SFC 44 RPL_VAL, see Section 11.7). The local data of the error OBs, do, however, take up additional space in the L stack of this priority class.

With S7-400 CPUs, one synchronous error OB can start a further synchronous error OB. This is not possible with S7-300 CPUs.

Using OBs for Asynchronous Errors

If the operating system of the CPU detects an asynchronous error, it starts the corresponding error OB (OB80 to OB87). The OBs for asynchronous errors have the highest priority and they cannot be interrupted by other OBs. If more than one asynchronous error occurs, the error OBs are executed in the order in which the errors occurred.

Masking Start Events

Using system functions (SFCs), you can mask, delay, or disable the start events for some of the error OBs. For more detailed information about these SFCs and the organization blocks, refer to the reference manual **/235/**.

Table 3-3 System Functions for Masking, Disabling and Delaying Start Events

Type of Error OB	SFC	Function of the SFC
Synchronous error OBs	SFC36 MSK_FLT	Masks individual synchronous errors. Masked errors do not start an error OB and do not trigger programmed reactions.
	SFC37 DMSK_FLT	Unmasks synchronous errors.
Asynchronous error OBs	SFC39 DIS_IRT	Disables all interrupts and asynchronous errors. Disabled errors do not start an error OB in any of the subsequent CPU cycles and do not trigger programmed reactions.
	SFC40 EN_IRT	Enables interrupts and asynchronous errors.
	SFC41 DIS_AIRT	Delays higher priority interrupts and asynchronous errors until the end of the OB.
	SFC42 EN_AIRT	Enables higher priority interrupts and asynchronous errors.

3.7 Interrupting Program Execution

Introduction

The operating system starts program execution by calling OB1. OB1 has the lowest priority. This means that any other OB call is capable of interrupting the cyclic program.

Sequence of Program Execution

When the operating system recognizes a start event for an OB with higher priority, the execution of the program is interrupted after the currently active instruction. The operating system saves the data of the interrupted block that will be required when the operating system resumes execution of the interrupted block.

An OB that interrupts the execution of another block can also call functions (FCs) and function blocks (FBs). The number of nested calls depends on the particular CPU. Refer to the CPU descriptions **/70/** and **/101/** for the maximum nesting depth of your particular CPU.

Saving the Data

If program execution is interrupted by a higher priority OB, the operating system saves the current contents of the accumulators and address registers and the number and size of the open data blocks in the interrupt stack (I stack).

Once the new OB has been executed, the operating system loads the information from the I stack and resumes execution of the interrupted block at the point at which the interrupt occurred.

When the CPU is in the STOP mode, you can display the I stack on a programming device using STEP 7. This allows you to find out why the CPU changed to the STOP mode.

3.8 Managing Local Data (L Stack)

Overview

When you are programming organization blocks, you can declare temporary variables (TEMP), that are only available when the block is executed and are then overwritten again (see also Section 2.4). In addition to this, every organization block also requires 20 bytes of local data for its start information.

Local Data Stack

The CPU has a limited amount of memory for the temporary variables (local data) of blocks currently being executed. The size of this memory area, (the local stack) depends on the particular CPU (refer to the CPU descriptions **/70/** and **/101/**). The local data stack is divided up equally among the priority classes (default). This means that every priority class has its own local data area which ensures that the high priority classes and their OBs have space for local data.

Figure 3-5 shows an example of the L stack in which OB1 is interrupted by OB10 that is, in turn, interrupted by OB81.

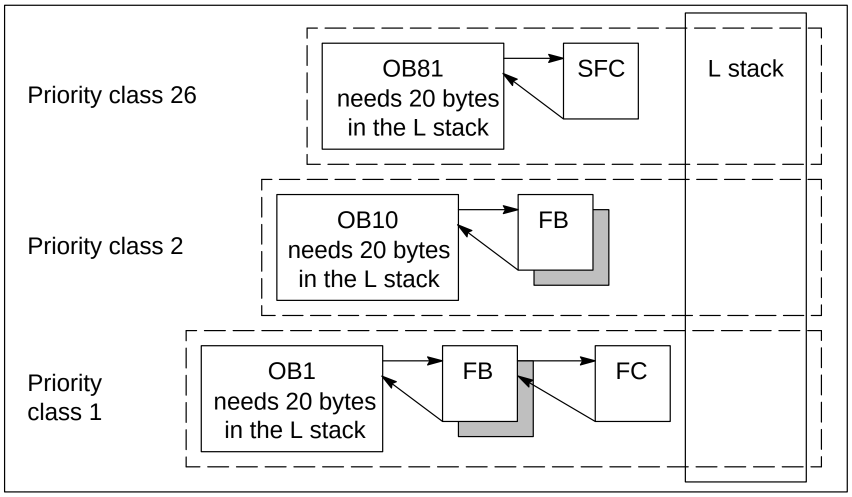

Priority class 26
OB81 needs 20 bytes in the L stack
SFC
L stack
Priority class 2
OB10 needs 20 bytes in the L stack
FB
Priority class 1
OB1 needs 20 bytes in the L stack
FB
FC

Figure 3-5 Assignment of Local Data to the Priority Classes

Caution

S7-CPUs change to the STOP mode if the permitted L stack size for a program is exceeded.

All the temporary variables (TEMP) of an OB and its associated blocks are saved in the L stack. If you use too many nesting levels when executing your blocks, the L stack can overflow.

Test the L stack (the temporary variables) in your program.

Assigning Local Data to Priority Classes

Different priority classes require different amounts of memory in the local data stack. By assigning parameters with STEP 7, you can select the size of the local data area for the individual priority classes on S7-400 CPUs. If you are not using certain priority classes you can use their areas for other priority classes on S7-400 CPUs. Deactivated OBs are ignored during program execution and save cycle time.

On S7-300 CPUs, each priority class is assigned a fixed number of local data (256 bytes) and this setting cannot be changed.

Handling Interrupts 4

What Does This Chapter Describe?

This chapter describes the interrupt OBs for time-of-day interrupts, time-delay interrupts, cyclic interrupts and hardware interrupts.

Where to Find More Information

The use of synchronous and asynchronous error OBs is described in Chapter 11.

For a detailed description of the individual organization blocks, refer to the reference manual **/235/**. For further information about assigning parameters for interrupts, refer to the manuals **/70/** and **/101/**.

Chapter Overview

4.1 Using Interrupt OBs

Interrupt-Driven Program Execution

By providing interrupt OBs, the S7 CPUs allow the following:

- Program sections can be executed at certain times or intervals (time driven)
- Your program can react to external signals from the process.

The cyclic user program does not need to query whether or not interrupt events have occurred. If an interrupt does occur, the operating system makes sure that the user program in the interrupt OB is executed so that there is a programmed reaction to the interrupt by the programmable logic controller,

Types of Interrupt and Applications

Table 4-1 shows how the different types of interrupt can be used.

Table 4-1 Examples of Applications

Type of Interrupt	Interrupt OBs	Examples of Applications
Time-of-day interrupt	OB10 to OB17	Calculation of the total flow into a blending process at the end of a shift
Time-delay interrupt	OB20 to OB23	Controlling a fan that must continue to run for 20 seconds after a motor is switched off
Cyclic interrupt	OB30 to OB38	Scanning a signal level for a closed-loop control system
Hardware interrupt	OB40 to OB47	Signaling that the maximum level of a tank has been reached

Using Interrupt OBs

To allow the operating system to execute an interrupt OB, you must perform the following steps:

- Create the required interrupt OB as an object in your S7 program using STEP 7.
- Write the program to be executed in the interrupt OB in the block you have created.
- Download the interrupt OB to the CPU as part of your user program.

Assigning Parameters for Interrupts

Interrupts can be assigned parameters with STEP 7. By assigning parameters, you can, for example, deselect interrupt OBs or modify priority classes.

4.2 Time-of-Day Interrupts (OB10 to OB17)

Description

The S7 CPUs provide the time-of-day interrupt OBs that can be executed at a specified date or at certain intervals.

Time-of-day interrupts can be triggered as follows:

- Once at a particular time (specified in absolute form with the date)
- Periodically by specifying the start time and the interval at which the interrupt should be repeated (for example every minute, every hour, daily).

Starting

To allow the CPU to start a time-of-day interrupt, you must first set and then activate the time-of-day interrupt. There are three ways of starting the interrupt:

- Automatic start of the time-of-day interrupt by assigning appropriate parameters with STEP 7 (parameter field “time-of-day interrupts”)
- Setting and activating the time-of-day interrupt with SFC28 SET_TINT and SFC30 ACT_TINT from within the user program
- Setting the time-of-day interrupt by assigning parameters with STEP 7 and activating the time-of-day interrupt with SFC30 ACT_TINT in the user program.

Querying

To query which time-of-day interrupts are set and when they are set to occur, you can do one of the following:

- Call SFC31 QRY_TINT
- Request the list “interrupt status of the system status list” (see Chapter 11).

Deactivating

You can deactivate time-of-day interrupts that have not yet been executed with SFC29 CAN_TINT. Deactivated time-of-day interrupts can be set again using SFC28 SET_TINT and activated with SFC30 ACT_TINT.

Priority

All eight time-of-day interrupt OBs have the same priority class (2) as default (see also Section 3.1) and are therefore processed in the order in which their start event occurs. You can, however, change the priority class by selecting suitable parameters.

Changing the Set Time

You can change the time-of-day set for the interrupt as follows:

- A clock master synchronizes the time for masters and slaves.
- SFC0 SET_CLK can be called in the user program to set a new time.

Reaction to Changing the Time

Table 4-2 shows how time-of-day interrupts react after the time has been changed.

Table 4-2 Time-of-Day Interrupts After Changing the Time

If...	Then...
If the time was moved ahead and one or more time-of-day interrupts were skipped,	OB80 is started and the time-of-day interrupts that were skipped are entered in the start information of OB80.
You have deactivated the skipped time-of-day interrupts in OB80,	the skipped time-of-day interrupts are no longer executed.
You have not deactivated the skipped time-of-day interrupts in OB80,	the first skipped time-of-day interrupt is executed, the other skipped time-of-day interrupts are ignored.
By moving the time back, the start events for the time-of-day interrupts occur again,	the execution of the time-of-day interrupt is not repeated.

Note

Time-of-day interrupts can only be executed when the interrupt has been assigned parameters and a corresponding organization block exists in the user program. If this is not the case, an error message is entered in the diagnostic buffer and an asynchronous error routine is executed (OB80, see Chapter 11).

Periodic time-of-day interrupts must correspond to a real date. Repeating an OB10 monthly starting on the 31st January is not possible. In this case, the OB would only be started in the months that have 31 days.

A time-of-day interrupt activated during startup (complete restart or restart) is only executed after the startup is completed.

Time-of-day interrupt OBs that are deselected by the parameter assignment, cannot be started. The CPU recognizes a programming error and changes to the STOP mode.

Following a complete restart, time-of-day interrupts must be set again (for example using SFC30 ACT_TINT in the startup program).

4.3 Time-Delay Interrupts (OB20 to OB23)

Description

The S7-CPUs provide time-delay OBs with which you can program the delayed execution of parts of your user program.

Time-delay interrupts are triggered when the delay time specified in SFC32 SRT_DINT has expired.

Starting

To start a time-delay interrupt, you must specify the delay time in SFC32 after which the corresponding time-delay interrupt OB is called. For the maximum permitted length of the delay time, refer to the individual CPU descriptions **/70/** and **/101/**.

Priority

The default priority for the time-delay interrupt OBs is priority class 3 to 6 (see also Section 3.1). You can assign parameters to change the priority classes.

Note

Time-delay interrupts can only be executed when the corresponding organization block exists in the CPU program. If this is not the case, an error message is entered in the diagnostic buffer and an asynchronous error routine is executed (OB80, see Chapter 11).

Time-delay interrupt OBs that were deselected by the parameter assignment cannot be started. The CPU recognizes a programming error and changes to the STOP mode.

4.4 Cyclic Interrupts (OB30 to OB38)

Description

The S7-CPUs provide cyclic interrupt OBs that interrupt cyclic program execution at certain intervals.

Cyclic interrupts are triggered at intervals. The time at which the interval starts is the mode change from STOP to RUN.

Starting

To start a cyclic interrupt, you must specify the interval in the cyclic interrupts parameter field using STEP 7. The interval is always a whole multiple of the basic clock rate of 1 ms.

Interval = n $\times$ basic clock rate 1 ms

Each of the nine available cyclic interrupt OBs has a default interval (see Table 4-3). The default interval becomes effective when the cyclic interrupt OB assigned to it is loaded. You can, however, assign parameters to change the default values. For the upper limit of the intervals for cyclic interrupts, refer to the CPU descriptions **/70/** and **/101/**.

Phase Offset

To avoid cyclic interrupts of different cyclic interrupt OBs being started at the same point and possibly causing a time error (cycle time exceeded) you can specify a phase offset. The phase offset ensures that the execution of a cyclic interrupt is delayed by a certain time after the interval has expired.

Phase offset = m $\times$ basic clock rate (where $0 \leq m < n$)

Figure 4-1 shows how a cyclic interrupt OB with phase offset is executed in contrast to a cyclic interrupt without phase offset.

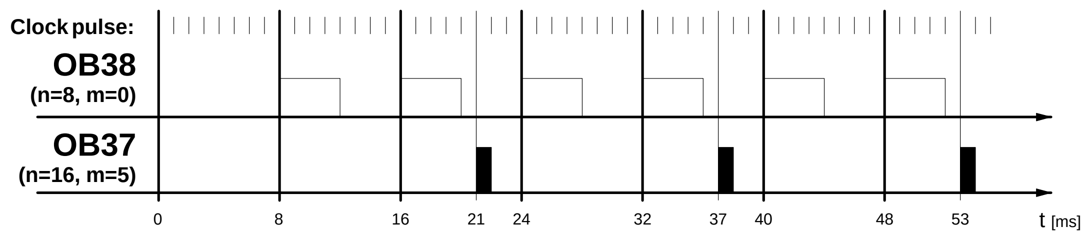


Figure 4-1 Executing Cyclic Interrupts with and without Phase Offset

Priority

Table 4-3 shows the default intervals and priority classes of the cyclic interrupt OBs. You can assign parameters to change the interval and the priority class.

Table 4-3 Intervals and Priority Classes of the Cyclic Interrupt OBs (Defaults)

Cyclic Interrupt OBs	Interval in ms	Priority Class
OB30	5000	7
OB31	2000	8
OB32	1000	9
OB33	500	10
OB34	200	11
OB35	100	12
OB36	50	13
OB37	20	14
OB38	10	15

Note

When you specify the intervals, make sure that there is enough time between the start events of the individual cyclic interrupts for processing the cyclic interrupts themselves.

If you assign parameters to deselect cyclic interrupt OBs, they can no longer be started. The CPU recognizes a programming error and changes to the STOP mode.

4.5 Hardware Interrupts (OB40 to OB47)

Description

The S7 CPUs provide hardware interrupt OBs that react to signals from the modules (for example signal modules SMs, communications processors CPs, function modules FMs). With S7, you can decide which signal from a configurable digital or analog module starts the OB. With CPs and FMs, use the appropriate parameter assignment dialogs.

Hardware interrupts are triggered when a signal module with hardware interrupt capability and with an enabled hardware interrupt passes on a received process signal to the CPU or when a function module of the CPU signals an interrupt.

Assigning Parameters

Each channel of a signal module with hardware interrupt capability can trigger a hardware interrupt. For this reason, you must specify the following in the parameter records of signal modules with hardware interrupt capability using STEP 7:

- What will trigger a hardware interrupt.
- Which hardware interrupt OB will be executed (the default for executing all hardware interrupts is OB40).

Using STEP 7, you activate the generation of hardware interrupts on the function blocks. You assign the remaining parameters in the parameter assignment dialogs of these function modules.

Priority

The default priority classes for the hardware interrupt OBs are 16 to 23 (see also Section 3.1). You can assign parameters to change the priority classes.

Note

Hardware interrupts can only be executed when the corresponding organization block is located in the CPU program. If this is not the case, an error message is entered in the diagnostic buffer and an asynchronous error routine is executed (see Chapter 11).

If you have deselected hardware interrupt OBs in the parameter assignment, these cannot be started. The CPU detects a programming error and changes to the STOP mode.

Memory Areas of S7 CPUs 5

What Does This Chapter Describe?

This chapter describes the memory areas of the S7-300 and S7-400 CPUs.

Chapter Overview

5.1 Memory Areas of the CPU

Distribution of the Memory Areas

The memory of the S7 CPUs has three basic areas:

- The load memory is used for user programs without symbolic address assignments or comments (these remain in the memory of the programming device). The load memory can be either RAM or FEPROM.

  Blocks identified as being not relevant to the running of your program are all located in the load memory.
- The work memory (integrated RAM) contains the parts of the S7 program relevant for running your program. The program is executed only in the work memory and system memory areas.
- The system memory (RAM) contains the memory elements provided by every CPU for the user program, such as the process image input and output tables, bit memory, timers and counters. The system memory also contains the block stack and interrupt stack.

  In addition to the areas above, the system memory of the CPU also provides temporary memory (local data stack) that contains temporary data for a block when it is called. This data only remains valid as long as the block is active.

Figure 5-1 illustrates the memory areas of the CPU.

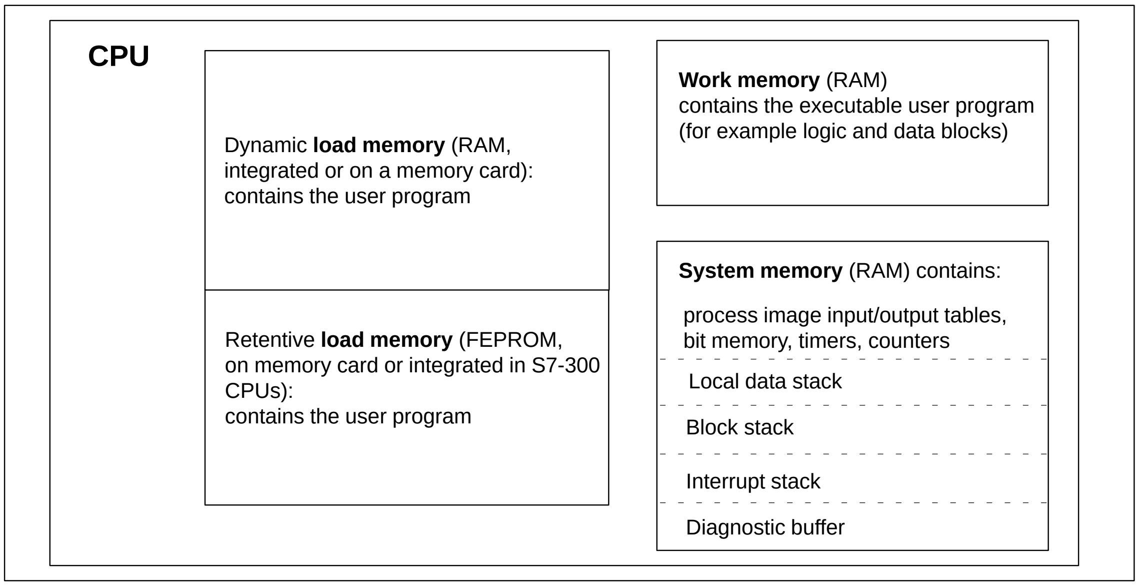

CPU
Dynamic load memory (RAM, integrated or on a memory card): contains the user program
Retentive load memory (FEPROM, on memory card or integrated in S7-300 CPUs): contains the user program
Work memory (RAM) contains the executable user program (for example logic and data blocks)
System memory (RAM) contains:
process image input/output tables, bit memory, timers, counters
Local data stack
Block stack
Interrupt stack
Diagnostic buffer

Figure 5-1 Memory Areas of S7 CPUs

Special Features of the S7-300

With S7-300 CPUs, the load memory can have both an integrated RAM as well as an integrated FEPROM section (refer to the CPU descriptions **/70/** and **/101/**). Areas in data blocks can be declared as being retentive by assigning parameters with STEP 7 (see Section 5.4).

Special Features of the S7-400

Use of a memory card (RAM or FEPROM) is required to expand the load memory on S7-400 CPUs. The integrated load memory is a RAM memory and is used essentially for loading and correcting blocks individually.

Consequences of the Load Memory Structure

The structure of the load memory (RAM and FEPROM area) affects the possibilities for downloading your user program or for downloading individual blocks. Table 5-1 shows how the program and blocks are downloaded:

Table 5-1 Load Memory Structure and What can be Downloaded

Memory Type	Possible Functions	Loading Method
RAM	Downloading and deleting individual blocks	PG-CPU connection
	Downloading and deleting an entire S7 program	PG-CPU connection
	Downloading individual blocks later	PG-CPU connection
FEPROM integrated (only on S7-300) or plug-in	Downloading entire S7 programs	PG-CPU connection
FEPROM plug-in	Downloading entire S7 programs	Uploading the FEPROM to the PG and plugging in the memory card in the CPU Downloading the FEPROM to the CPU

Programs stored in RAM are lost when you reset the CPU memory (MRES) or if you remove the CPU or RAM memory card.

Programs saved on FEPROM memory cards are not erased by a CPU memory reset and are retained even without battery backup (transport, backup copies).

Using the Memory Areas

The memory of the S7 CPUs is divided into address areas (see Table 5-2). Using instructions in your program, you address the data directly in the corresponding address areas. To find out which address areas are available on your CPU, refer to the CPU descriptions **/70/**, **/101/** or the instruction lists **/72/**, **/102/**.

Table 5-2 Address Areas

Address Area	Access in Units of the Following Size:	S7 Notation	Description
Process-image input	Input (bit) Input byte Input word Input double word	I IB IW ID	At the beginning of the scan cycle, the CPU reads the inputs from the input modules and records the values in this area.
Process-image output	Output (bit) Output byte Output word Output double word	Q QB QW QD	During the scan cycle, the program calculates output values and places them in this area. At the end of the scan cycle, the CPU sends the calculated output values to the output modules.
Bit memory	Memory (bit) Memory byte Memory word Memory double word	M MB MW MD	This area provides storage for interim results calculated in the program.
Timer	Timer (T)	T	This area provides storage for timers.
Counter	Counter (C)	C	This area provides storage for counters.
Data block	Data block opened with “OPN DB”: Data bit Data byte Data word Data double word	DB DBX DBB DBW DBD	Data blocks contain information for the program. They can be defined for general use by all logic blocks (shared DBs) or they are assigned to a specific FB or SFB (instance DB).
	Data block opened with “OPN DI”: Data bit Data byte Data word Data double word	DI DIX DIB DIW DID	
Local (temporary) data	Local data bit Local data byte Local data word Local data double word	L LB LW LD	This area contains the temporary data of a block while the block is being executed. The L stack also provides memory for transferring block parameters and for recording interim results from Ladder Logic networks.
I/O: external inputs	Peripheral input byte Peripheral input word Peripheral input double word	PIB PIW PID	The peripheral input and output areas allow direct access to central and distributed input and output modules (DP, see Section 6.3.)
I/O: external outputs	Peripheral output byte Peripheral output word Peripheral output double word	PQB PQW PQD	

5.2 Absolute and Symbolic Addressing

Types of Addressing

You can access addresses in a STEP 7 program using either absolute addressing (for example I 1.7) or using symbolic addressing (for example motor contact 1).

Absolute Addressing

The absolute address of a memory location contains the address identifier (for example “M”) and the type of access to the data area: B (byte), W (word or two bytes) or D (double word or four bytes). If you do not specify B, W or D, it is assumed that bit access is required. The absolute address also contains the number of the first byte and the bit number for bit access.

Table 5-3 Examples of Absolute Addressing

Absolute Address	**Description**
MD 100	This relates to a double word (a double word consists of 4 bytes), beginning in memory byte 100 (in other words bytes 100, 101, 102 and 103)
M 100.1	Relates to bit 1 in memory byte 100

Symbolic Addressing

By assigning a symbol to an address, you can identify the function of the address and make your program easier to understand. The following distinction is made when you assign a symbolic name:

- Global symbols, in other words the symbolic name is valid for all blocks in an S7 program; these are declared in the symbol table of the S7 program.
- Block-local symbols, in other words the symbolic name is valid for only one block; this is declared in the local data (for example parameters) in the declaration table of the block.

5.3 Storing Programs on the CPU

Downloading the User Program

When you download the user program from the programming device to the CPU, only the logic and data blocks are loaded in the load and work memory of the CPU.

The symbolic address assignment (symbol table) and the block comments remain on the programming device.

Distribution of the Program in the Load and Work Memory

To ensure fast execution of the user program and to avoid unnecessary load on the work memory that cannot be expanded, only the parts of the blocks relevant for program execution are loaded in the work memory (see Figure 5-2). Parts of blocks that are not required for executing the program (for example block headers) remain in the load memory.

The load memory can be expanded using memory cards. For the maximum size of your load memory, refer to the CPU descriptions **/70/** and **/101/**.

Depending on whether you select a RAM or an FEPROM memory card to expand the load memory, the load memory may react differently during downloading, reloading or memory reset (see also Section 5.1).

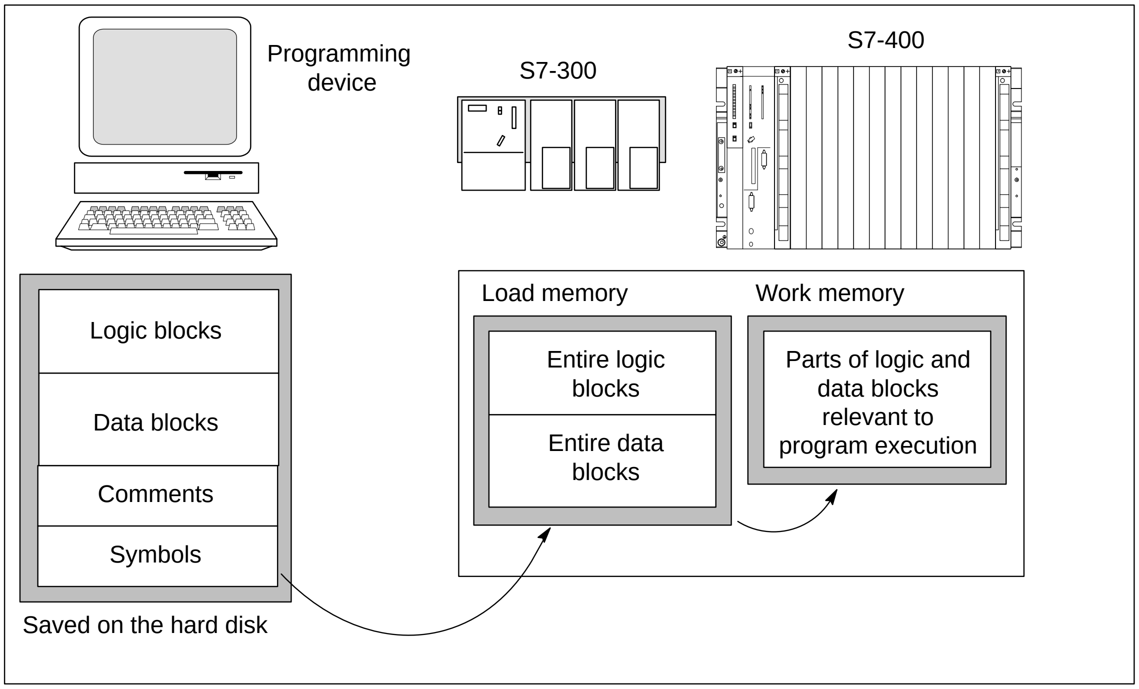


Figure 5-2 Downloading the Program to the CPU Memory

Data Blocks Created with SFCs

The CPU stores data blocks that were created using system functions (for example SFC22 CREAT_DB) in the user program only in the work memory and not in the load memory.

Data Blocks Not Relevant for Program Execution

Data blocks that were programmed in a source file as part of an STL program can be identified as “Not Relevant for Execution” (keyword UNLINKED). This means that when they are downloaded to the CPU, the DBs are stored only in the load memory. The content of such blocks can, if necessary, be copied to the work memory using SFC20 BLKMOV.

This technique saves space in the work memory. The expandable load memory is then used as a buffer (for example for formulas for a mixture; only the formula for the next batch is loaded in the work memory).

5.4 Retentive Memory Areas on S7-300 CPUs

Overview

If a power outage occurs or the CPU memory is reset (MRES), the memory of the S7-300 CPU (dynamic load memory (RAM), work memory and system memory) is reset and all the data previously contained in these areas is lost. With S7-300 CPUs, you can protect your program and its data in the following ways:

- You can protect all the data in the load memory, work memory and in parts of the system memory with battery backup.
- You can store your program in the FEPROM (either memory card or integrated on the CPU, refer to the CPU descriptions **/70/**).
- You can store a certain amount of data depending on the CPU in an area of the non-volatile NVRAM.

Using the NVRAM

Your S7-300 CPU provides an area in the NVRAM (non-volatile RAM) (see Figure 5-3). If you have stored your program in the FEPROM of the load memory, you can save certain data (if there is a power outage or when the CPU changes from STOP to RUN) by configuring your CPU accordingly. To do this set the CPU so that the following data are saved in the non-volatile RAM:

- Data contained in a DB (This is only useful if you have also stored your program in an FEPROM of the load memory.)
- Values of timers and counters
- Data saved in bit memory.

On every CPU, you can save a certain number of timers, counters and memory bits. A specific number of bytes is also available in which the data contained in DBs can be saved. For more detailed information, refer to the CPU descriptions **/70/**.

The MPI address of your CPU is stored in the NVRAM. This makes sure that your CPU is capable of communication following a power outage or memory reset.

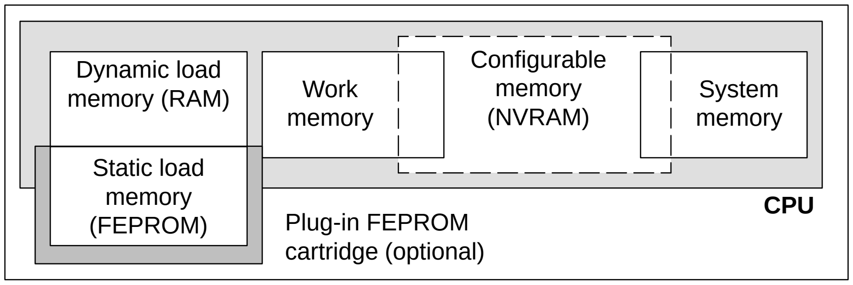

Dynamic load memory (RAM)
Work memory
Configurable memory (NVRAM)
System memory
Static load memory (FEPROM)
Plug-in FEPROM cartridge (optional)
CPU

Figure 5-3 Non-Volatile Memory Area on S7-300 CPUs

Using Battery Backup to Protect Data

By using a backup battery, the data of the load memory and the work memory are protected from loss in case of a power outage. If you configure your CPU so that timers, counters and memory bits are saved in the NVRAM, this information is also retained regardless of whether you use a backup battery or not.

Configuring the Data of the NVRAM

When you configure your CPU with STEP 7, you can decide which memory areas will be retentive.

The amount of memory that can be configured in the NVRAM depends on the CPU you are using. You cannot back up more data than specified for your CPU. For more detailed information about retentive memory, refer to the manual **/70/**.

5.5 Retentive Memory Areas on S7-400 CPUs

Operation Without Battery Backup

If you operate your system without battery backup, when a power outage occurs or when you reset the CPU memory (MRES), the memory of the S7-400 CPU (dynamic load memory (RAM), work memory and system memory) is reset and all the data contained in these areas is lost.

Without battery backup, only a complete restart is possible and there are no retentive memory areas. Following a power outage, only the MPI parameters (for example the MPI address of the CPU) are retained. This means that the CPU remains capable of communication following a power outage or memory reset.

Operation With Battery Backup

If you use a battery to back up your memory:

- The entire content of all RAM areas is retained when the CPU restarts following a power outage.
- During a complete restart, the address areas for memory bits, timers, and counters is cleared. The contents of data blocks are retained.
- The contents of the RAM work memory are also retained apart from memory bits, timers and counters that were designed as non-retentive.

Configuring Retentive Data Areas

You can declare a certain number of memory bits, timers, and counters as retentive (the number depends on your CPU). During a complete restart when you are using a backup battery, this data is also retained during a complete restart.

When you assign parameters with STEP 7, you define which memory bits, timers, and counters should be retained during a complete restart. You can only back up as much data as is permitted by your CPU.

For more detailed information about defining retentive memory areas, refer to the manual **/101/**.

5.6 Process Image Input/Output Tables

Introduction

If the input (I) and output (Q) address areas are accessed in the user program, the program does not scan the signal states on the digital signal modules but accesses a memory area in the system memory of the CPU and distributed I/Os. This memory area is known as the process image.

The process image is divided into two parts: the process image input table and the process image output table.

Prior Requirement

The CPU can only access the process image of the modules that you have configured with STEP 7 or that are obtainable using the default addressing.

Updating the Process Image

The process image is updated cyclically by the operating system. At the beginning of cyclic program execution, the signal states of the input modules are transferred to the process image input table. At the end of each program cycle, the signal states are transferred from the process image output table to the output modules.

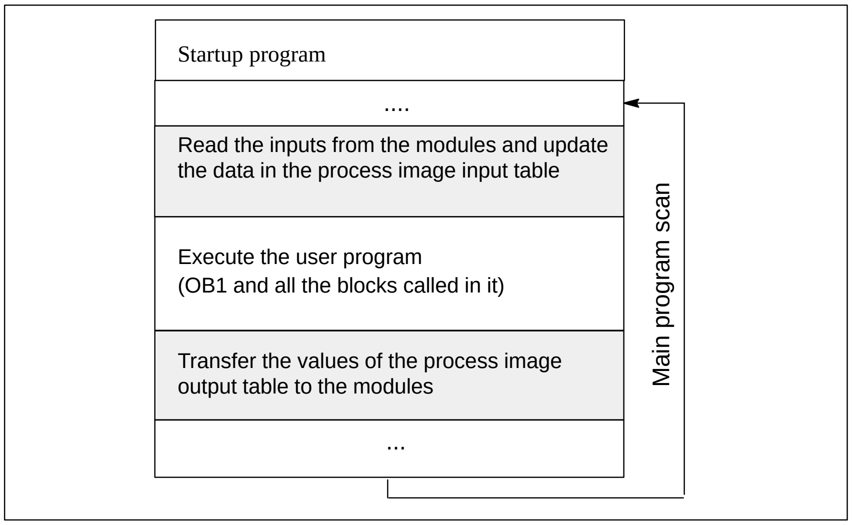

Startup program
....
Read the inputs from the modules and update the data in the process image input table
Execute the user program
(OB1 and all the blocks called in it)
Transfer the values of the process image output table to the modules
...
Main program scan

Figure 5-4 Updating the Process Image

Advantages

Compared with direct access to the input/output modules, the main advantage of accessing the process image is that the CPU has a consistent image of the process signals for the duration of one program cycle. If a signal state on an input module changes while the program is being executed, the signal state in the process image is retained until the process image is updated again at the beginning of the next cycle. Access to the process image also requires far less time than direct access to the signal modules since the process image is located in the internal memory of the CPU.

Updating Sections of the Process Images

With some CPUs, you can create and update up to eight sections of the process image tables (refer to the CPU descriptions **/70/** and **/101/**). This means that the user program can update sections of the process image table, when necessary, independent of the cyclic updating of the process image table.

You define the process image sections with STEP 7. SFCs are used to update a section of the process image.

Using SFCs

By using the following SFCs, the user program can update an entire process image table or sections of a process image table:

- SFC26 UPDAT_PI updates the process image input table.
- SFC27 UPDAT_PO updates the process image output table.

Note

On S7-300 CPUs, inputs and outputs that are not used for the process image tables can be used as additional bit memory areas. Programs that make use of this option cannot run on S7-400 CPUs.

5.7 Local Data Stack

L Stack

The local data stack (L stack) is a memory area in the system memory of the CPU (see also Section 3.8). It contains the following:

- The temporary variables of the local data of blocks
- The start information of the organization blocks
- Information about transferring parameters
- Interim results of the logic in Ladder Logic programs

Size

The size of the local data stack depends on the particular CPU (refer to the CPU descriptions **/70/** and **/101/**). The local data stack is divided up equally among the priority classes (default). This means that each priority class has its own local data area which ensures that higher priority classes and their OBs also have space for their local data (see also Section 3.8).

Addressing Peripheral I/Os 6

What Does This Chapter Describe?

This chapter describes how the peripheral I/O data areas are addressed (user data, diagnostic and parameter data).

Where to Find More Information

For further information about the system functions mentioned in this chapter, refer to the reference manual **/235/**.

Chapter Overview

6.1 Access to Process Data

Overview

The CPU can access inputs and outputs of central and distributed digital input/output modules either indirectly using the process image tables or directly via the backplane/P bus.

Addressing Modules

You assign the addresses used in your program to the modules when you configure the modules with STEP 7, as follows:

- With central I/O modules: arrangement of the rack and assignment of the modules to slots in the configuration table.
- With distributed I/Os (SINEC L2-DP): arrangement of the DP slaves in the configuration table “master system” with the L2 address and assignment of the modules to slots.

By configuring the modules, it is no longer necessary to set addresses on the individual modules using switches. As a result of the configuration, the PG sends data to the CPU that allows the CPU to recognize the modules assigned to it.

Peripheral I/O Addressing

There is a separate address area for inputs and outputs. This means that the address of a peripheral area must not only include the byte or word access type but also the I identifier for inputs and Q identifier for outputs.

To find out which address areas are possible on individual modules, refer to the manuals **/70/**, **/71/** and **/101/**.

Table 6-1 Peripheral I/O Address Areas

Address Area	Access With Units of the Following Size:	S7 Notation
Peripheral area: inputs	Peripheral input byte	PIB
	Peripheral input word	PIW
	Peripheral input double word	PID
Peripheral area: outputs	Peripheral output byte	PQB
	Peripheral output word	PQW
	Peripheral output double word	PQD

Module Start Address

The module start address is the lowest byte address of a module. It represents the start address of the user data area of the module and is used in many cases to represent the entire module.

The module start address is, for example, entered in process interrupts, diagnostic interrupts, insert/remove module error interrupts and power supply error interrupts in the start information of the corresponding organization block and is used to identify the module that initiated the interrupt.

6.2 Access to the Peripheral Data Area

Overview

The peripheral data area can be divided into the following:

- User data and
- Diagnostic and parameter data.

Both areas have an input area (can only be read) and an output area (can only be written).

User Data

User data is addressed with the byte address (for digital signal modules) or the word address (for analog signal modules) of the input or output area. User data can be accessed with load and transfer commands, communication functions (operator interface access) or by transferring the process image. User data can be as follows:

- Digital and analog input/output signals from signal modules
- Control and status information from function modules
- Information for point-to-point and bus connections from communication modules (only S7-300)

When transferring user data, a consistency of a maximum of 4 bytes can be achieved (with the exception of DP standard slaves, see Section 6.3). If you use the “transfer double word” statement, four contiguous and unmodified (consistent) bytes are transferred. If you use four separate “transfer input byte” statements, a hardware interrupt OB could be inserted between the statements and transfer data to the same address do that the content of the original 4 bytes is changed before they were all transferred.

Diagnostic and Parameter Data

The diagnostic and parameter data of a module cannot be addressed individually but are always transferred in the form of complete data records. This means that consistent diagnostic and parameter data are always transferred.

The diagnostic and parameter data is accessed using the start address of the module and the data record number. Data records are divided into input and output data records. Input data records can only be read, output data records can only be written. You can access data records using system functions or communication functions (user interface). Table 6-2 shows the relationship between data records and diagnostic and parameter data.

Table 6-2 Assignment of the Data Records

Data	Description
Diagnostic data	If the modules are capable of diagnostics, you obtain the diagnostic data of the module by reading data records 0 and 1.
Parameter data	If the modules are configurable, you transfer the parameters to the module by writing data records 0 and 1.

Accessing Data Records

You can use the information in the data records of a module to reassign parameters to configurable modules and to read diagnostic information from modules with diagnostic capability.

Table 6-3 shows which system functions you can use to access data records.

Table 6-3 System Functions for Accessing Data Records

SFC	Application
Assigning parameters to modules	
SFC55 WR_PARM	Transfers modifiable parameters (data record 1) to the addressed signal module.
SFC56 WR_DPARM	Transfers the parameters (data records 0 **or** 1) from SDBs 100 to 129 to the addressed signal module.
SFC57 PARM_MOD	Transfers all parameters (data records 0 **and** 1) from SDBs 100 to 129 to the addressed signal module.
SFC58 WR_REC	Transfers any data record to the addressed signal module.
Reading out diagnostic information	
SFC59 RD_REC	Reads the diagnostic data

Addressing S5 Modules

You can access STEP 5 modules as follows:

- By connecting an S7-400 to SIMATIC S5 expansion racks using the network adapter IM 463-2
- By plugging in certain S5 modules in an adapter casing in the central rack of the S7-400

How you address S5 modules with SIMATIC S7 is explained in the manual **/100/** or in the description supplied with the adapter casing.

6.3 Special Features of Distributed Peripheral I/Os (DP)

Distributed I/Os

With SIMATIC S7, you can use the distributed peripheral I/Os (DP). Distributed I/Os are analog and digital modules installed close to the process and function modules (FM) installed on the P bus and therefore normally at some distance from the CPU.

Attachment to S7

You can attach distributed peripherals to the S7 programmable logic controller using the SINEC L2-DP bus system and one of the following:

- The integrated DP master interface of a CPU (for example CPU 315-2-DP, CPU 413-2 DP, CPU 414-2 DP)
- An interface module assigned to a CPU/FM (for example IF 964-DP in the CPU 388-5, CPU 488-5).
- An external DP master interface (for example CP 443-5, CP 342-5, IM 467)

Configuration

Distributed modules are configured in the same way as central modules using STEP 7 (refer to the STEP 7 user manual **/231/**).

Addressing the DP Master and DP Slaves

The address area of the distributed I/Os is the same for DP master and DP slave modules and corresponds to the peripheral I/O address shown in Table 6-1.

Access to User Data

The DP master module provides a data area for the user data of the distributed I/Os. The CPU accesses this data area when it addresses the distributed I/Os.

It is also possible to access user data just as with the central I/Os using load and transfer commands, communication functions (operator interface) and process image transfer. The maximum data consistency is 4 bytes.

Access to Diagnostic and Parameter Data

Just as with the central I/Os, diagnostic and parameter data can be accessed using SFCs (see Table 6-3) (exception DP standard slaves).

Addressing DP Standard Slaves

If you want to exchange data longer than 4 bytes with DP standard slaves, you must use special SFCs for this data exchange.

Table 6-4 System Functions for DP Standard Slaves

SFC	Application
Assigning parameters to modules	
SFC15 DPWR_DAT	Transfers any data record to the addressed signal module
Reading diagnostic information	
SFC13 DPNRM_DG	Reads the diagnostic information (asynchronous read access)
SFC14 DPRD_DAT	Reads consistent diagnostic data (length 3 or greater than 4 bytes)

When a DP diagnostic frame arrives, a diagnostic interrupt with 4 bytes of diagnostic data is signaled to the CPU. You can read out these 4 bytes with SFC13 DPNRM_DG. The entire DP diagnostic information can be read with SFC14 DPRD_DAT by specifying the diagnostic address of the DP standard slave.

Data Exchange Between Programmable Modules 7

What Does This Chapter Describe?

This chapter describes communication possible with the S7-300 and S7-400 programmable controllers.

Where to Find More Information

Global data communication
When data is exchanged using global data communication, two or more networked CPUs share common data, the global data.

For further information about the topic of global data communication and configuring connections, refer to the STEP 7 online help and the STEP 7 user manual **/231/**.

The communication SFBs for configured connections are described in the reference manual **/235/**.

Heterogeneous communication using SIMATIC CPs is described in detail in the manuals **/500/** and **/501/**. These manuals also describe the corresponding communication function blocks.

Chapter Overview

Section	Description	Page
7.1	Types of Communication	7-2
7.2	Data Exchange Using Communication SFBs for Configured Connections	7-3
7.3	Configuring a Communication Connection between Communication Partners	7-5
7.4	Working with Communication SFBs for Configured Connections	7-7
7.5	Data Exchange with Communication SFCs for Non-Configured Connections	7-8

7.1 Types of Communication

Overview

The following types of communication are possible with SIMATIC S7:

- Homogeneous communication is communication between S7 components that use the S7 protocol.
- Heterogeneous communication is communication between S7 components and S5 components and between S7 components and devices from other vendors using different protocols (for example TF, FMS).

S7 Communication

- Communication with communication SFBs for configured connections

  The S7-400 CPUs provide communication SFBs for configured connections for data exchange between programmable modules.

  These include system function blocks (SFBs) with which you can exchange data between two communication partners on a subnet controlled by the program (for example CPU, FM, CP).

  SFBs are also available for checking and changing the operating modes of remote devices.

- Communication with communication SFCs for non-configured connections

  The S7-300 and S7-400 CPUs provide communication SFCs for non-configured connections, to allow data to be exchanged between two communication partners.

  The two communication partners must be attached to the same MPI subnet or belong to the same S7 station (modules capable of communication in the central rack, in an expansion rack or in a DP station).

7.2 Data Exchange with SFBs for Configured Connections

Prior Requirements

To transfer data between communication partners using communication SFBs for configured connections, the following conditions must be satisfied:

- The partners must be on one subnet (MPI, PROFIBUS, Industrial Ethernet).
- You have configured a connection between the partners.
- You call the required system function blocks and corresponding instance data blocks in the user program.

Communication SFBs for Configured Connections

S7 CPUs provide communication SFBs for configured connections to exchange data between communication partners (CPU, CP, FM) in a network, to control a remote device and to monitor or query the internal status of a local communication SFB.

Table 7-1 SFBs and SFC for Data Exchange

SFB/SFC		Brief Description	Connection
Send and receive functions			
SFB8 SFB9	USEND URCV	Uncoordinated data exchange using a send and a receive SFB	bilateral
SFB12 SFB13	BSEND BRCV	Exchange of blocks of data of variable length between a send SFB and a receive SFB	bilateral
SFB14	GET	Reads data from a remote device	unilateral
SFB15	PUT	Writes data to a remote device	unilateral
Control functions			
SFB19	START	Initiates a complete restart on a remote device	unilateral
SFB20	STOP	Sets a remote device to the STOP mode	unilateral
SFB21	RESUME	Initiates a restart on a remote device	unilateral
Monitoring functions			
SFB22	STATUS	Specific query of the status of a remote device	unilateral
SFB23	USTATUS	Receives status messages from remote devices	bilateral
Query function			
SFC62	CONTROL	Queries the internal state of a local communication SFB using its instance DB.	–

Types of Communication

The following types of communication are distinguished when exchanging data with communication SFBs:

- Bilateral communication indicated by the local and remote communication partners each having one SFB of a pair.
- Unilateral communication indicated by the fact that only the local communication partner has a communication SFB.

Figures 7-1 and 7-2 illustrate the two types of communication.

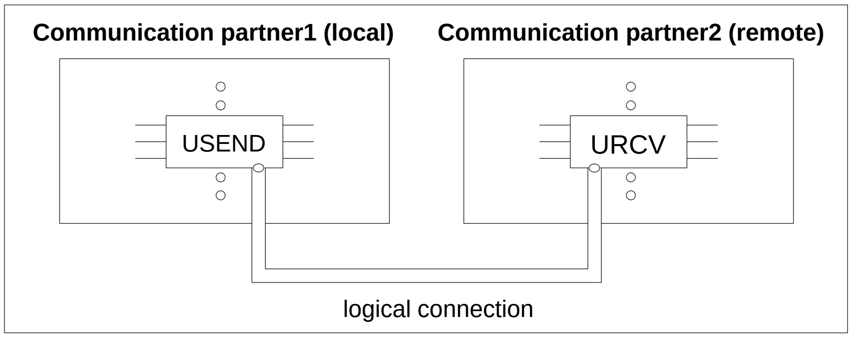

Communication partner1 (local)
Communication partner2 (remote)
USEND
URCV
logical connection

Figure 7-1 Bilateral Communication

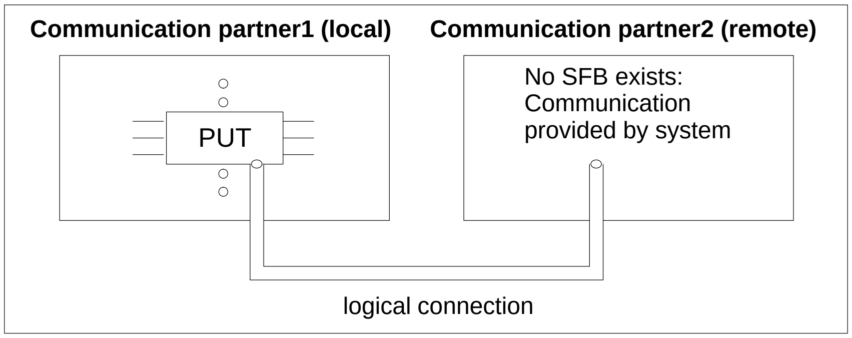

Communication partner1 (local)
Communication partner2 (remote)
PUT
No SFB exists: Communication provided by system
logical connection

Figure 7-2 Unilateral Communication

Pogramming Example

A sample program for transferring data between communication partners using communication SFBs for configured connections is supplied with STEP 7. This sample program is described in Appendix B, the source code is in the directory step7\examples\com.slb.

7.3 Configuring a Communication Connection Between Partners

Communication Partners

Data exchange using communication function blocks is possible between the following partners:

- S7 CPUs
- M7 CPUs
- S7 CPUs and M7 CPUs
- CPUs and FMs
- CPUs and CPs.

Figure 7-3 and Table 7-2 are examples indicating which communication partners can exchange data.

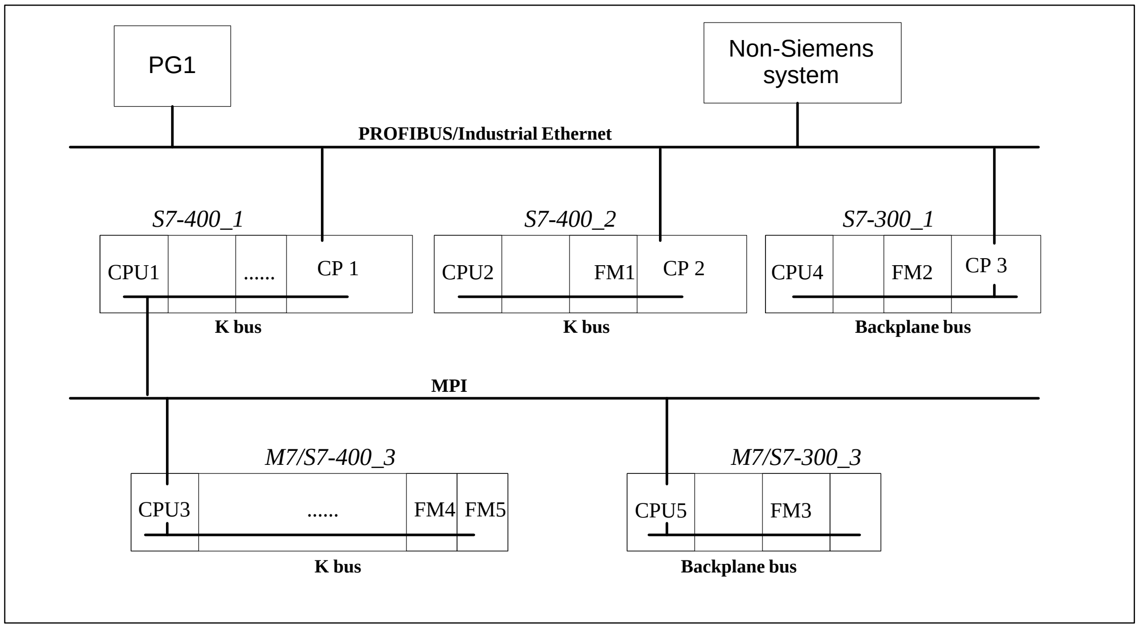


Figure 7-3 Communication Partners that Can Exchange Data

Table 7-2 Examples of Communication Partners that Can Exchange Data

Data Exchange Possible Between	Type of Communication
CPU 1 ↔ CPU 2	homogeneous
CPU 1 ↔ CPU 3	homogeneous
CPU 2 → CPU 4 (only with PUT/GET/START/STOP/STATUS)	homogeneous
CPU 3 → CPU 5 (only with PUT/GET/START/STOP/STATUS)	homogeneous
CPU 1 ↔ system from other vendor	heterogeneous

Note

Distributed FMs (on the P bus) cannot currently take part in data exchange using communication SFBs for configured connections.

Configuring a Connection

To allow data to be exchanged between communication partners, the partners must be networked (MPI, PROFIBUS, Industrial Ethernet) and there must be a connection between the partners. In STEP 7, you configure this connection by creating a connection table and loading it on the corresponding module with the user program. The table contains the following information:

- The connection IDs for both communication partners
- The remote communication partner
- The type of communication

Unilateral/Bilateral Connections

Just as there is unilateral and bilateral communication, there are also unilateral and bilateral connections:

- Unilateral connection: there is only one communication SFB on the local communication partner
- Bilateral connection: there is a pair of blocks on the local and remote communication partners

You must also specify the type of connection when you create the connection table. The number of possible connections per configurable module depends on the CPU.

Connection ID

Each configured connection is identified by a connection ID. This represents the local reference between the block and the connection. The local and the remote communication partners of a configured connection can have different connection IDs. The connection IDs are assigned by STEP 7.

When you call the communication SFBs, you must specify the corresponding connection ID as the input parameter for each block.

7.4 Working with Communication SFBs for Configured Connections

System Function Blocks and System Functions

System function blocks and system functions are part of the operating system of an S7 CPU. They can be called by the user program and are not loaded as part of the user program.

Instance Data Blocks

Just like function blocks, system function blocks require an instance data block that contains the actual parameters and local data areas of the SFB. Instance data blocks must be created with STEP 7 and loaded as part of the user program.

Addressing the Communication Partner

The logical connection between two communication partners is identified by their connection IDs.

It is possible to use the same logical connection for different send/receive jobs. For this reason, you must also specify a job ID R_ID in addition to the connection ID to indicate that the send and receive blocks belong together.

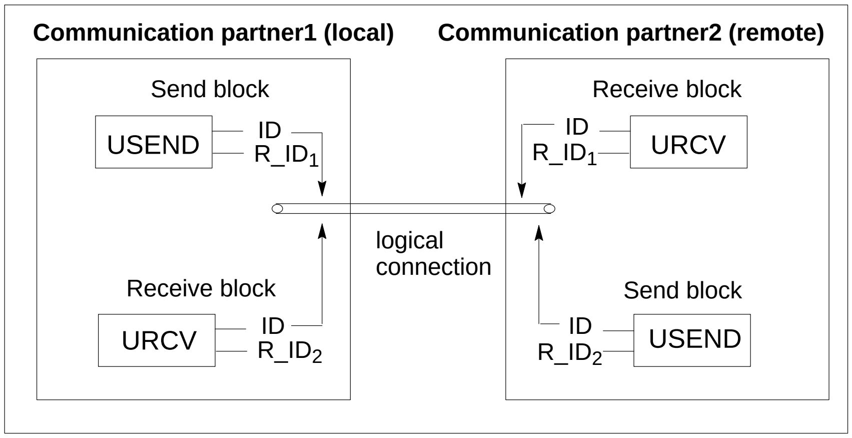


Figure 7-4 Addressing Parameters ID and R_ID

Sample Program

Appendix B contains a sample program illustrating data exchange using communication SFBs for configured connections.

7.5 Data Exchange with Communication SFCs for Non-Configured Connections

Requirements

Using communication SFCs for non-configured connections, you can exchange data between an S7 CPU and another module that is capable of communication. The following conditions must be satisfied:

- The communication partner must be attached to the same MPI subnet or belong to the same S7 station (module capable of communication in the central rack, in an expansion rack or in a DP station).
- The required SFCs must be called in the user program.

Communication SFCs for Non-Configured Connections

The S7-300 and S7-400 CPUs provide communication SFCs for non-configured connections to allow data exchange between two communication partners and to terminate existing connections.

Table 7-3 SFCs for Communication between S7 Stations

Block	Description
SFC65 “X_SEND”/ SFC66 “X_RCV”	Data exchange between communication partners using a send and a receive SFC
SFC67 “X_GET”	Reads a variable from a communication partner
SFC68 “X_PUT”	Writes a variable to a communication partner
SFC69 “X_ABORT”	Aborts an existing connection to a communication partner. This releases connection resources at both ends of the connection.

Table 7-4 SFCs for Communication within an S7 Station

Block	Description
SFC72 “I_GET”	Reads a variable from a communication partner (for example FM)
SFC73 “I_PUT”	Writes a variable to a communication partner (for example FM)
SFC74 “I_ABORT”	Aborts an existing connection to a communication partner. This releases connection resources at both ends of the connection.

Connection to Communication Partner

To allow data to be exchanged between communication partners, the partners must be networked (K bus or PROFIBUS DP for SFCs I_GET, I_PUT and I_ABORT, MPI for SFCs X_SEND, X_RCV, X_GET, X_PUT and X_ABORT). A communication connection is established by the operating system of the CPU while the SFC is being executed.

You decide whether or not the connection is terminated after the data transfer using an input parameter (for more detailed information, refer to the Reference Manual **/235/**).

If a connection cannot be established at the present time, you must call the SFC again later.

A connection is established by the CPU on which a communication SFC is called (exception: the SFC66 “X_RCV” call does not establish a connection). A maximum of one connection in both directions is possible between two communication partners.

Addressing the Communication Partner

- If the communication partner is not in the same S7 station:

  The logical connection is specified by the MPI address of the communication partners (parameter DEST_ID). You configured this with STEP 7.

- If the communication partner is in the same S7 station:

  The logical connection is specified by the address area ID (parameter IOID) and the logical address (parameter LADDR) of the communication partner.

Setting System Parameters 8

What Does This Chapter Describe?

This chapter explains how you can modify certain properties of S7-300 and S7-400 programmable controllers by setting system parameters or using system functions (SFCs).

Where to Find More Information

For detailed information about module parameters, refer to the STEP 7 online help and the manuals **/70/**, **/71/** and **/101/**. The SFCs are described in detail in the reference manual **/235/**.

Chapter Overview

8.1 Changing the Behavior and Properties of Modules

Default Settings

When supplied, all the configurable modules of the S7 programmable controller have default settings suitable for standard applications. With these defaults, you can use the modules immediately without making any settings. The defaults are explained in the module descriptions **/70/**, **/71/** and **/101/**.

Which Modules Can You Assign Parameters to?

You can, however, modify the behavior and the characteristics of the modules to adapt them to your requirements and the situation in your plant. Configurable modules are CPUs, FMs, CPs and some of the analog input/output modules and digital input modules.

There are configurable modules with and without backup batteries.

Modules without backup batteries must be supplied with data again following any power down. The parameters of these modules are stored in the retentive memory area of the CPU (indirect parameter assignment by the CPU).

Setting and Loading Parameters

You set module parameters using STEP 7. When you save the parameters, STEP 7 creates the object “System Data Blocks” that is loaded on the CPU with the user program and transferred to the modules when the CPU starts up.

System Data Blocks

System data blocks (SDBs) can only be evaluated by the operating system and cannot be edited with STEP 7.

Not all the existing system data blocks are available on all CPUs (refer to the CPU descriptions **/70/** and **/101/**). Table 8-1 lists several SDBs that are available on every CPU and shows which parameters they contain.

Table 8-1 Parameters in SDBs

SDB	Parameter Record
0	CPU operating system parameters
1	Peripheral I/O assignment list
2	CPU default parameter record
3	Integrated DP interface
22	Distributed I/O assignment list, internal interface
26	Distributed I/O assignment list, external interface
100 to 103	Parameters for modules in the central configuration of an S7-300
100 to 121	Parameters for modules in the central configuration of an S7-400
122	Parameters for modules in the distributed configuration, internal interface of an S7-300 and S7-400
126 to 129	Parameters for modules in the distributed configuration, external interface of an S7-300 and S7-400
1000 to 32767	Parameters for K bus modules

Which Settings can be Made?

The module parameters are divided into parameter fields. Which parameter fields are available on which CPU is explained in the CPU descriptions **/70/** and **/101/**.

Parameter fields exist for the following topics:

- Startup behavior
- Cycle
- MPI
- Diagnostics
- Retentive data
- Clock memory
- Interrupt handling
- On-board I/Os (only for the S7-300)
- Protection level
- Local data
- Real-time clock
- Asynchronous errors

Parameter Assignment with SFCs

In addition to assigning parameters with STEP 7, you can also include system functions in the S7 program to modify module parameters. Table 8-2 illustrates which SFCs transfer which module parameters (see also Section 6.2).

Table 8-2 System Functions for Accessing Data Records

SFC	Application
SFC55 WR_PARM	Transfers the modifiable parameters (data record 1) to the addressed signal module
SFC56 WR_DPARM	Transfers parameters (data records 0 **or** 1) from SDBs 100 to 129 to the addressed signal module
SFC57 PARM_MOD	Transfers all parameters (data records 0 **and** 1) from SDBs 100 to 129 to the addressed signal module
SFC58 WR_REC	Transfers any data record to the addressed signal module

The system functions are described in detail in the reference manual **/235/**.

Which module parameters can be modified dynamically is explained in the manuals **/70/**, **/71/** or **/101/**.

8.2 Using the Clock Functions

Overview

All S7-300-/S7-400 CPUs are equipped with a clock (real-time clock or software clock). The clock can be used in the programmable controller both as clock master or clock slave with external synchronization. The clock is required for time-of-day interrupts and run-time meters.

Time Format

The clock always indicates the time (minimum resolution 1 s), date and weekday. With some CPUs, it is also possible to indicate milliseconds (refer to the CPU descriptions **/70/** and **/101/**).

Setting and Reading the Time

You set the time and date for the CPU clock by calling SFC 0 SET_CLK in the user program or with a menu option on the programming device to start the clock. Using SFC1 READ_CLK or a menu option on the programming device, you can read the current date and time on the CPU.

Assigning Parameters for the Clock

If more than one module equipped with a clock exists in a network, you must set parameters using STEP 7 to specify which CPU functions as master and which as slave when the time is synchronized. When setting these parameters, you also decide whether the time is synchronized via the K bus or via the MPI interface and the intervals at which the time is automatically synchronized.

Synchronizing the Time

To make sure that the time is the same on all modules in the network, the slave clocks are synchronized by the system program at regular (selectable) intervals. You can transfer the date and time from the master clock to the slave clocks using system function SFC48 SFC_RTCB.

Using a Run-Time Meter

A run-time meter counts the run-time hours of connected equipment or the total run-time hours of the CPU.

In the STOP mode, the run-time meter is stopped. Its count value is retained even after a memory reset. During a complete restart, the run-time meter must be restarted by the user program; following a restart, it continues automatically if it had already been started.

You can set the run-time meter to an initial value using SFC2 SET_RTM. You can start or stop the run-time meter with SFC3 CTRL_RTM. You can read the current total operating hours and the state of the counter (”stopped” or “counting”) with SFC4 READ_RTM.

A CPU can have up to eight run-time meters (refer to the CPU descriptions **/70/** and **/101/**). Numbering starts at 0.

8.3 Specifying the Startup Behavior

Introduction

The startup behavior of S7 CPUs is described in Chapter 9. When you select parameters to determine the startup behavior, remember that only S7-400 CPUs are capable of a restart.

Startup After Manual Start

A manual complete restart is the only option on S7-300 CPUs.

On S7-400 CPUs, you can restart manually using the mode selector and the startup type switch (CRST/WRST) if this is permitted by the parameter assignment you made with STEP 7. A manual complete restart is possible without specifically assigning parameters.

Startup After Automatic Start

On S7-300 CPUs, only a COMPLETE RESTART is possible following power up.

With S7-400 CPUs, you can specify whether an automatic startup following power up leads to a COMPLETE RESTART or a RESTART.

Clearing the Process Image

When an S7-400 CPU is restarted, the remaining cycle is executed, and as default, the process image output table is cleared. You can prevent the process image being cleared if you want the user program to continue with the old values following a restart (see also Figure 9-2).

Self-Test During a Complete Restart

On S7-300 CPUs, you can set parameters to specify whether the CPU tests its internal RAM during a complete restart.

Module Exists/Type Monitoring

In the parameters, you can decide whether the modules in the configuration table are checked to make sure they exist and that the module type matches before the startup.

If the module check is activated, the CPU will not start up if a discrepancy is found between the configuration table and the actual configuration.

Monitoring Times

To make sure that the programmable controller starts up without errors, you can select the following monitoring times:

- The maximum permitted time for transferring parameters to the modules
- The maximum permitted time for the modules to signal that they are ready for operation after power up
- On S7-400 CPUs, the maximum time of an interruption following which a restart is permitted.

Once the monitoring times expire, the CPU either changes to STOP and only a complete restart is possible.

8.4 Settings for the Cycle

Cycle Time

The cycle time is the time required by the CPU to execute the cyclic program and all the program sections resulting from interrupts during the cycle (for example servicing hardware interrupts) and the time required for system activities. This time is monitored.

Maximum Cycle Time

With STEP 7, you can modify the default maximum cycle time. If this time expires, the CPU either changes to the STOP mode or OB80 is called in which you can specify how the CPU should react to this error.

Minimum Cycle Time

On S7-400 CPUs, you can set a minimum cycle time using STEP 7. This is useful in the following situations:

- When the interval at which program execution starts in OB1 (main program scan) should always be the same or
- When the process image tables would be updated unnecessarily often if the cycle time is too short.

Figure 8-1 illustrates the cycle monitoring time during program startup.

T_{max}	is the selectable maximum cycle time
T_{min}	is the selectable minimum cycle time
T_c	is the actual cycle time
T_{wait}	is the difference between T_{min} and the actual cycle time. During this time, interrupt OBs can be serviced.
PC	stands for priority class

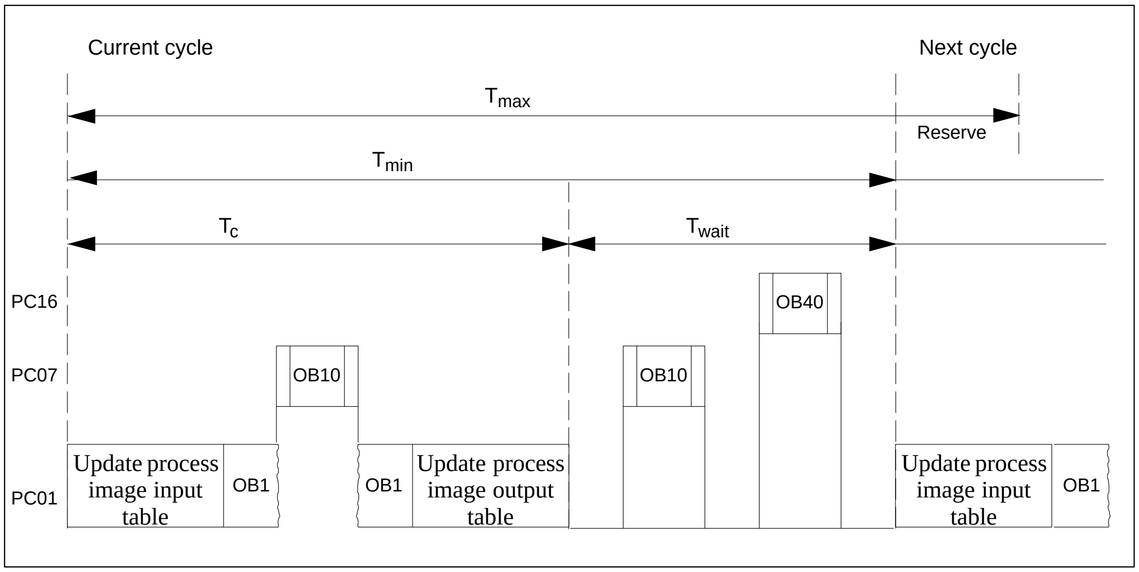

Current cycle
Next cycle
T_{max}
Reserve
T_{min}
T_c
T_{wait}
PC16
PC07
PC01
OB40
OB10
OB10
Update process image input table
OB1
OB1
Update process image output table
Update process image input table
OB1

Figure 8-1 Cycle Monitoring Time

Updating the Process Image

During cyclic program execution by the CPU, the process image is updated automatically. On S7-400 CPUs, you can prevent updating of the process image in one of the two following situations:

- If you want to access the I/Os directly or
- You want to update one or more process image input or output sections at a different point in the program using system functions SFC26 UPDAT_PI and SFC27 UPDAT_PO.

Communication Load

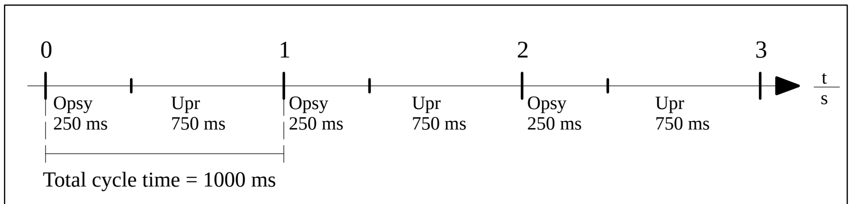
To prevent communication functions extending the time required for program execution too much, you can specify the maximum amount by which the cycle can be extended by communication.

When you decide on the load added to the cycle by communication, remember that the operating system execution time further extends the run time. If you set a communication load of 50%, this does not double the original run time but more than doubles it, the further increase depending on the CPU being used. This is illustrated by an example based on a worst case situation.

Situation:

- The operating system execution time is 250 ms per second cycle time.
- The user program has a run time of 750 ms
- The load on the cycle caused by communication is 0%.

A cycle can be represented in simplified form as follows:

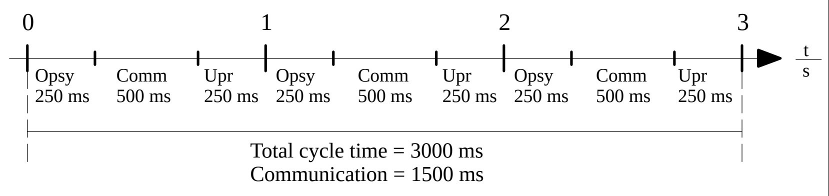


Opsy = run time load caused by operating system
Upr = run time load caused by user program

The cycle load due to communication is now set to 50%:

- The operating system execution time continues to be 250 ms per second cycle time
- The user program continues to run for 750 ms
- The run time load caused by communication is 1500 ms per cycle

The time sequence is then as follows:



Opsy = run time load caused by operating system

Comm = run time load caused by communication

Upr = run time load caused by user program

In this example, setting the communication load to 50% extends the cycle time from 1 second to 3 seconds, in other words, the total cycle time is tripled.

8.5 Specifying the MPI Parameters

Multipoint Interface

Up to 32 devices that communicate with each other can be connected to the multipoint interface (MPI) of a CPU. The following devices can be connected:

- Programmable controllers
- Programming devices
- Operator interface systems.

Values after Memory Reset

To ensure that a CPU can still communicate after a memory reset, the MPI parameters are stored in a retentive memory area of the CPU and are retained after a memory reset, after removing/inserting the module, and if a battery is defective or is not being used.

Setting the Parameters

With STEP 7, you set the following parameters:

- The node address of the CPU
- The extent of the MPI network (highest node address in the MPI network, default 16).

8.6 Specifying Retentive Memory Areas

Uses

To prevent data being lost during a complete restart (and on the S7-300 CPUs following a power outage), you can declare certain data areas as retentive.

For a detailed description of retentive memory areas on the S7-300 and S7-400 CPUs, refer to Chapter 5.

Setting the Parameters

By setting parameters with STEP 7, you select the limits of the retentive memory areas, as follows:

- For the S7-300, retentive areas for memory bits, timers, counters, and areas in data blocks
- For S7-400-CPUs, the retentive areas for memory bits, timers, and counters.

8.7 Using Clock Memory and Timers

Clock Memory

The clock memory is a memory byte that changes its binary state periodically at a pulse-pause ratio of 1:1. You select which memory byte is used on the CPU when you assign parameters for the clock memory using STEP 7.

Uses

You can use clock memory bytes in the user program, for example to activate flashing lights or to trigger periodic activities (for example measuring an actual value).

Possible Frequencies

Each bit of the clock memory byte is assigned a frequency. Table 8-3 illustrates the assignment:

Table 8-3 Possible Frequencies of a Clock Memory Byte

Bit	7	6	5	4	3	2	1	0
Period (s)	2.0	1.6	1.0	0.8	0.5	0.4	0.2	0.1
Frequency (Hz)	0.5	0.625	1	1.25	2	2.5	5	10

Note

Clock memory bytes are not synchronous with the CPU cycle, in other words in long cycles, the state of the clock memory byte may change several times.

Timers

Timers are a memory area of the system memory. You specify the function of a timer in the user program (for example on-delay timer).

The number of timers available depends on the CPU (see **/70/** and **/101/**). If you use less timers in the user program than are actually available, you can set the parameters so that only this number of timers is updated in the STARTUP and RUN modes. This optimizes the operating system run time.

Note

If you use more timers in your user program than the CPU permits, a synchronous error is signaled and OB121 started.

If you use more timers in your user program than you have selected in the parameter settings, no error is signaled, however the timers do not run. You can use the STEP 7 debugging functions to check whether or not a timer is running.

On the S7-300, timers can be started and updated simultaneously only in OB 1 and OB100. Timers can only be started in all the other OBs.

8.8 Changing the Priority Classes and Amount of Local Data

Introduction

On S7-400 CPUs, you can set parameters to change the priority of some of the interrupt OBs. This means that you can decide which interrupt OBs can be interrupted by higher priority interrupt OBs.

Fixed Priority Classes

You cannot change the priority classes of the following OBs:

- Main program scan OB1
- Background OB90
- Startup types OB100 and OB101
- Multicomputing OB60
- Asynchronous errors OB80 to 87
- Error OBs started by synchronous errors. They are executed in the same priority class as the block being executed when the error occurred.

Changing the Priority

You can change the default priority of the interrupt OBs (providing such a change is permitted) by changing the parameters in the parameter fields: time-of-day interrupts, time-delay interrupts, cyclic interrupts, and hardware interrupts (see also Section 3.1).

Local Data

When creating logic blocks (OBs, FCs, FBs), you can declare temporary local data. The local data area on the CPU is divided among the priority classes.

Changing the Amount of Local Data

On S7-400 CPUs, you can change the amount of local data per priority class in the “priority classes” parameter field using STEP 7. Each OB must have at least 20 local data bytes that are required to transfer the OB start information.

Deselected Interrupt OBs

If you assign priority class 0 or assign less than 20 bytes of local data to a priority class, the corresponding interrupt OB is deselected. The handling of deselected interrupt OBs is restricted as follows:

- In the RUN mode, they cannot be copied or linked in your user program.
- In the STOP mode, they can be copied or linked in your user program, but when the CPU goes through a complete restart they stop the startup and an entry is made in the diagnostic buffer.

Uses

By deselecting interrupt OBs that you do not require, you increase the amount of local data area available, and this can be used to save temporary data in other priority classes.

Operating Modes and Mode Changes 9

What Does This Chapter Describe?

This chapter provides you with an overview of the operating modes of the S7 CPUs and describes the different types of startup on S7 CPUs.

This chapter also explains how the operating system supports you when debugging your user program.

Overview of the Chapter

9.1 Operating Modes and Mode Changes

Operating Modes

Operating modes describe the behavior of the CPU at any particular point in time. Knowing the operating modes of CPUs is useful when programming the startup, debugging the control program, and for troubleshooting.
Figure 9-1 shows the operating modes of the S7-300 and S7-400 CPUs: STOP, START UP, RUN and HOLD.

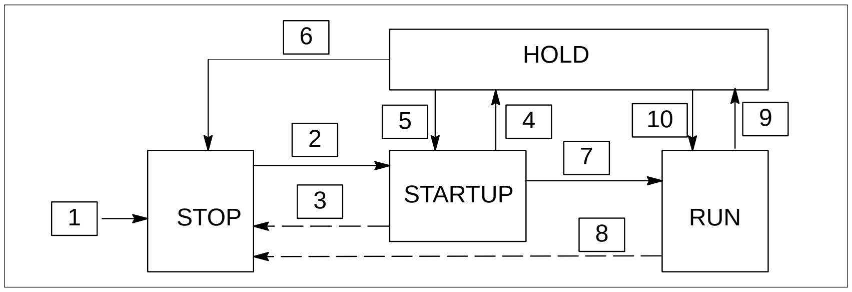


Figure 9-1 How the Operating Modes Change

In the STOP mode, the CPU checks whether all the configured modules or modules set by the default addressing actually exists and sets the I/Os to a predefined initial status. The user program is not executed in the STOP mode.

In the STARTUP mode, a distinction is made between the startup types “Complete Restart” and “Restart”:

- In a complete restart, the program starts at the beginning with initial settings for the system data and user address areas (the non-retentive timers, counters and bit memory are reset).
- In a restart, the program is resumed at the point at which it was interrupted (timers, counters and bit memory are not reset). A restart is only possible on S7-400 CPUs.

In the RUN mode, the CPU executes the user program, updates the inputs and outputs, services interrupts and process error messages.

In the HOLD mode, execution of the user program is halted and you can test the user program step by step. The HOLD mode is only possible when you are testing using the programming device.

In all these modes, the CPU can communicate on the MPI interface.

Other Operating Modes

If the CPU is not ready for operation, it is in one of the following modes:

- Off, in other words, the power supply is turned off.
- Defective, in other words, a fault has occurred.
  To check whether the CPU is really defective, switch the CPU to STOP and turn the power switch off and then on again. If the CPU starts up, read out the diagnostic buffer to analyze the problem. If the CPU does not start up it must be replaced.

Operating Mode Changes

Table 9-1 shows the conditions under which the operating modes can change.

Table 9-1 Changing the Modes of the CPU (Explanation of Figure 9-1)

Point	Description
1.	After you turn on the power supply, the CPU is in the STOP mode.
2.	The CPU changes to the STARTUP mode: • After the CPU is changed to RUN or RUN-P using the keyswitch or by the programming device. • After a startup triggered automatically by turning on the power. In both cases the keyswitch must be set to RUN or RUN-P.
3.	The CPU changes back to the STOP mode when: • An error is detected during the startup. • The CPU is changed to STOP by the keyswitch or on the programming device. • A stop command is executed in the startup OB. • The STOP communication function is executed.
4.	The CPU changes to the HOLD mode when a breakpoint is reached in the startup program.
5.	The CPU changes to the STARTUP mode when the breakpoint in a startup program was set and the “EXIT HOLD” command was executed (test functions).
6.	The CPU returns to the STOP mode when: • The CPU is changed to STOP with the keyswitch or by the programming device. • The STOP communication command is executed.
7.	If the startup is successful, the CPU changes to RUN.
8.	The CPU changes to the STOP mode again when: • An error is detected in the RUN mode and the corresponding OB is not loaded. • The CPU is changed to STOP with the keyswitch or by the programming device. • A stop command is executed in the user program. • The STOP communication function is executed.
9.	The CPU changes to the HOLD mode when a breakpoint is reached in the user program.
10.	The CPU changes to the RUN mode when a breakpoint was set and the “EXIT HOLD” command is executed.

Priority of the Operating Modes

If more than one mode change is requested at the same time, the CPU changes to the mode with the highest priority. If, for example, the mode selector is set to RUN and you attempt to set the CPU to STOP at the programming device, the CPU will change to STOP because this mode has the highest priority.

Priority	Mode
Highest	STOP
	HOLD
	START UP
Lowest	RUN

9.2 STOP Mode

Features

When the CPU is in the STOP mode, the user program is not executed. All the outputs are set to substitute values so that the controlled process is in a safe status. The CPU makes the following checks:

- Are there any hardware problems (for example modules not available)?
- Should the default setting apply to the CPU or are there parameter records?
- Are the conditions for the programmed startup behavior satisfied?
- Are there any system software problems?

In the STOP mode, the CPU can also receive global data and passive unilateral communication is possible using communication SFBs for configured connections and communication SFCs for non-configured connections (see also Table 9-5).

Memory Reset

The CPU memory can be reset in the STOP mode. The memory can be reset manually using the keyswitch (MRES) or from the programming device (for example before downloading a user program.

Resetting the CPU memory returns the CPU to its initial status, as follows:

- The entire user program in the work memory and in the RAM load memory and all address areas are cleared.
- The system parameters and the CPU and module parameters are reset to the default settings. The MPI parameters set prior to the memory reset are retained.
- If a memory card (Flash-EPROM) is plugged in, the CPU copies the user program from the memory card to the work memory (including the CPU and module parameters if the appropriate configuration data are also on the memory card).

The diagnostic buffer, the MPI parameters, the time and the run-time meters are not reset.

9.3 STARTUP Mode

Features

Before the CPU can start executing the user program, a startup program must first be executed. By programming startup OBs in your startup program, you can specify certain settings for your cyclic program.

There are two types of startup: complete restart and restart (S7-300 CPUs are only capable of a complete restart). A restart is only possible when this is explicitly specified in the parameter record of the CPU using STEP 7.
The features of the STARTUP mode are as follows:

- The program in the startup OB is processed (OB100 for a complete restart and OB101 for a restart)
- No time or interrupt-driven program execution is possible.
- Timers are updated.
- Run-time meters start running.
- Disabled digital outputs on signal modules (can be set by direct access).

Complete Restart

A complete restart is always permitted unless the system has requested a memory reset. A complete restart is the only possible option after:

- Memory reset
- Downloading the user program with the CPU in the STOP mode
- I stack/B stack overflow
- Complete restart aborted (due to a power outage or changing the mode selector setting)
- When the interruption before a restart exceeds the selected time limit.

Manual Complete Restart

A manual complete restart can be triggered by the following:

- The mode selector

The CRST/WRST switch must be set to CRST.

- The corresponding menu on the programming device or by communication functions (mode selector set to RUN or RUN-P).

Automatic Complete Restart

An automatic complete restart can be triggered following power up in the following situations:

- The CPU was not in the STOP mode when the power outage occurred.
- The mode selector is set to RUN or RUN-P.
- No automatic restart is programmed following power up.
- The CPU was interrupted by a power outage during a complete restart (regardless of the programmed type of restart).

The CRST/WRST switch has no effect on an automatic complete restart.

Automatic Complete Restart Without a Backup battery

If you operate your CPU without a backup battery (if maintenance-free operation is necessary), the CPU memory is automatically reset and a complete restart executed after the power is turned on or when power returns following a power outage. The user program must be located on a flash EPROM (memory card).

Restart

Following a power outage in the RUN mode followed by a return of power, S7-400 CPUs run through an initialization routine and then automatically execute a restart. During a restart, the user program is resumed at the point at which its execution was interrupted. The section of user program that had not been executed before the power outage, is known as the remaining cycle (see also Figure 9-2). The remaining cycle can also contain time and interrupt-driven program sections.

A restart is only permitted when the user program was not modified in the STOP mode (for example by reloading a modified block) and when there are no other reasons for a complete restart (refer to complete restart). Both a manual and automatic restart are possible.

Manual Restart

A manual restart is only possible with the appropriate parameter settings in the parameter record of the CPU and when the STOP resulted from the following causes:

- The mode selector was changed from RUN to STOP.
- The STOP mode was the result of a command from the programming device.

A manual restart can be triggered as follows:

- Using the mode selector.

  The CRST/WRST must be set to WRST.
- Using the menu option on the programming device or by communication functions (when the mode selector is set to RUN or RUN-P).
- When a manual restart following power down is set in the parameter record of the CPU.

Automatic Restart

An automatic restart can be triggered by a power up in the following situations:

- The CPU was not in the STOP mode when the power outage occurred.
- The mode selector is set to RUN or RUN-P.
- Automatic restart following power up is set in the parameter record of the CPU.

The CRST/WRST switch has no effect on an automatic restart.

Retentive Data Areas Following Power Down

S7-300 and S7-400 CPUs react differently to power up following a power outage.

S7-300 CPUs are only capable of a complete restart. With STEP 7, you can, however, specify memory bits, timers, counters and areas in data blocks as retentive to avoid data loss caused by a power outage. When the power returns, an “automatic complete restart with memory” is executed.

S7-400 CPUs react to the return of power by executing either a complete restart or a restart (depending on the parameter settings).

Tables 9-2 and 9-3 show the data that are retained on S7-300 and S7-400 CPUs during a complete restart or a restart.

X	means	data retained
0	means	data reset or cleared (contents of DBs)
I	means	data set to the initialization value taken from the EPROM.

Table 9-2 Data Retention in the EPROM Load Memory

	EPROM (memory card or integrated)								
	CPU with backup battery				**CPU without backup battery**				
Data	Logic blocks	DB	Memory bits, timers, counters		Logic blocks	DB		Memory bits, timers, counters	
			defined as retentive	defined as non-retentive		defined as retentive	defined as non-retentive	defined as retentive	defined as non-retentive
Complete restart on S7-300	X	X	X	0	X	X	I	X	0
Complete restart on S7-400	X	X	X	0	I	I		0	
Restart on S7-400	X	X	X		Only complete restart permitted				

Table 9-3 Data Retention in the RAM Load Memory

	RAM (memory card or integrated)						
	CPU with backup battery				CPU without backup battery		
Data	Logic blocks	DB	Memory bits, timers, counters		Logic blocks	DB	Memory bits, timers, counters
Complete restart on S7-300	X	X	X defined as retentive	0 defined as non-retentive	0	0	0
Complete restart on S7-400	X	X	X	0	0	0	0
Restart on S7-400	X	X	X		Only complete restart permitted		

Startup Activities

The activities performed by the CPU during startup are illustrated by Table 9-4:

X	means	is performed
0	means	is not performed

Table 9-4 Startup Activities

Activities in Order of Execution	In Complete Restart	In Restart
Clear I stack/B stack	X	0
Clear non-retentive memory bits, timers, counters	X	0
Clear process image output table	X	selectable
Clear outputs of digital signal modules	X	selectable
Discard hardware interrupts	X	0
Discard diagnostic interrupts	X	X
Update the system status list (SZL)	X	X
Evaluate module parameters and transfer to modules or transfer default values	X	X
Execution of the relevant startup OB	X	X
Execute remaining cycle (part of the user program not executed due to the power down)	0	X
Update the process image input table	X	X
Enable digital outputs (cancel OD signal)	X	X

Aborting a Startup

If an error occurs during startup, the startup is aborted and the CPU changes to or remains in the STOP mode.

An aborted complete restart must be repeated. After an aborted restart, both a complete restart and a restart are possible.

No startup (complete restart or restart) is executed or it is aborted in the following situations:

- The keyswitch of the CPU is set to STOP.
- A memory reset is requested.
- A memory card with an application identifier that is not permitted for STEP 7 is plugged in (for example STEP 5).
- More than one CPU is plugged in in the single-processor mode.
- If the user program contains an OB that the CPU does not recognize or that has been disabled.
- When, after power up, the CPU recognizes that not all the modules listed in the configuration table created with STEP 7 are actually plugged in.
- If errors occur when evaluating the module parameters.

A restart is not executed or is aborted in the following situations:

- The CPU memory was reset (only a complete restart is possible after memory reset).
- The interruption time limit has been exceeded (this is the time between exiting the RUN mode until the startup OB including the remaining cycle has been executed).
- The module configuration has been changed (for example module replaced).
- The parameter assignment only permits a complete restart.
- When blocks have been loaded, deleted or modified while the CPU was in the STOP mode.

Sequence of Activities

Figure 9-2 shows the activities of the CPU during STARTUP and RUN.

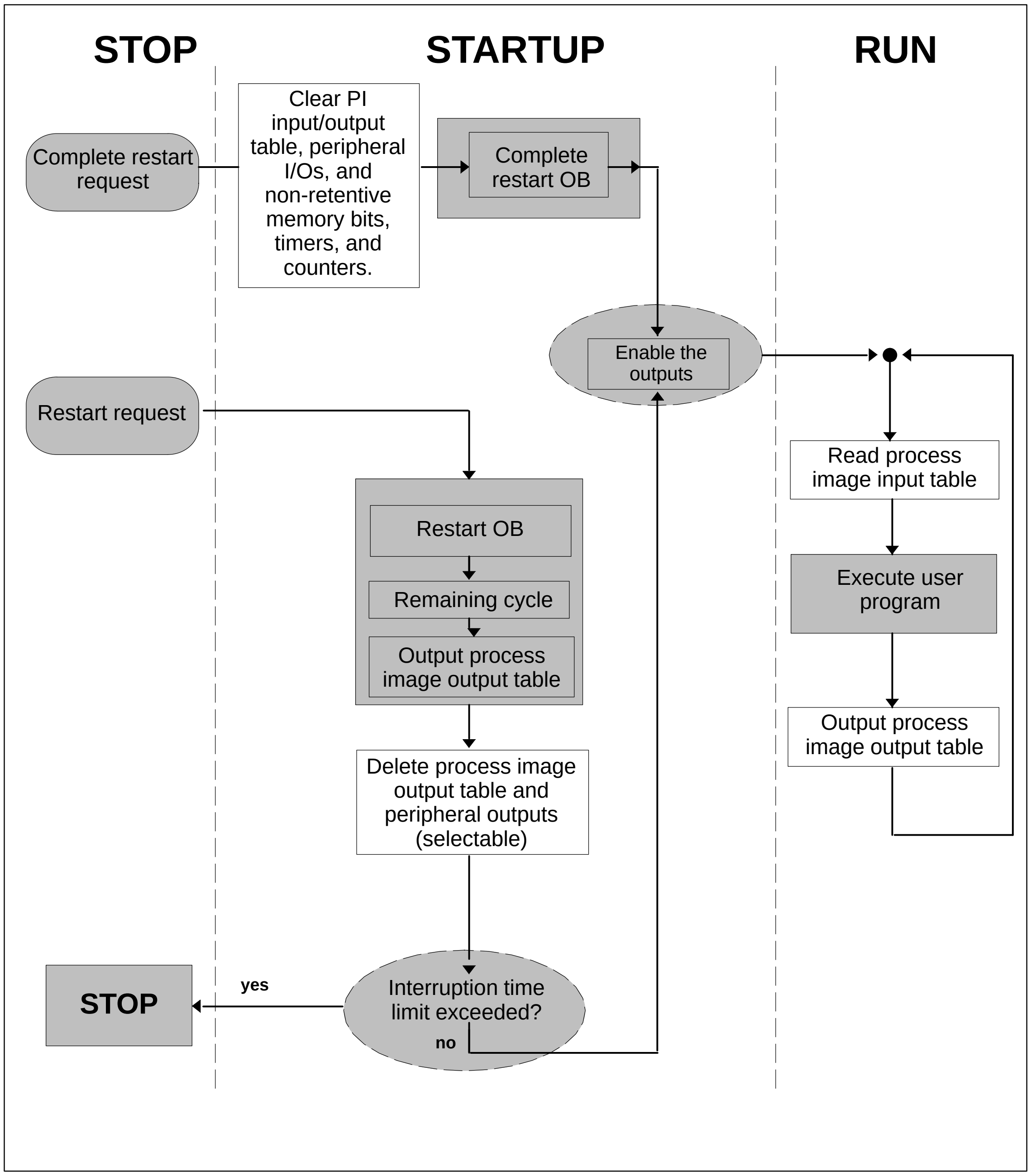

STOP
STARTUP
RUN
Complete restart request
Clear PI input/output table, peripheral I/Os, and non-retentive memory bits, timers, and counters.
Complete restart OB
Enable the outputs
Restart request
Restart OB
Remaining cycle
Output process image output table
Delete process image output table and peripheral outputs (selectable)
Interruption time limit exceeded?
yes
no
STOP
Read process image input table
Execute user program
Output process image output table

Figure 9-2 CPU Activities in START UP and RUN

9.4 RUN Mode

Features

In the RUN mode, the CPU executes the cyclic, time-driven and interrupt-driven program, as follows:

- The process image of the inputs is read in.
- The user program is executed.
- The process image output table is output.

The active exchange of data between CPUs using global data communication (global data table) and using communication SFBs for configured connections and communication SFCs for non-configured connections is only possible in the RUN mode.

Table 9-5 shows examples of when data exchange is possible in different operating modes:

↔	means	data exchange is possible in both directions
→	means	data exchange is possible in only one direction
X	means	data exchange is not possible

Table 9-5 Data Exchange in Different Operating Modes

Type of Communication	Mode of CPU 1	Direction of Data Exchange	Mode of CPU 2
Global data communication	RUN	↔	RUN
	RUN	→	STOP/HOLD
	STOP	←	RUN
	STOP	X	STOP
	HOLD	X	STOP/HOLD
Unilateral with communication SFBs	RUN	→	RUN
	RUN	→	STOP/HOLD
Bilateral with communication SFBs	RUN	↔	RUN
Unilateral with communication SFCs	RUN	→	RUN
	RUN	→	STOP/HOLD
Bilateral with communication SFCs	RUN	↔	RUN

9.5 HOLD Mode

Features

The HOLD mode is a special mode. This is only used for test purposes during startup or in the RUN mode. The HOLD mode means the following:

- All timers are frozen: timers and run-time meters are not processed, monitoring times are stopped, the basic clock pulses of the time-driven levels are stopped.
- The real-time clock runs.
- Outputs are not enabled but can be enabled explicitly for test purposes.
- Inputs and outputs can be set and reset.
- If a power outage occurs on a CPU with a backup battery while in the HOLD mode, the CPU changes to stop when the power returns but does not execute an automatic restart or complete restart. CPUs without battery backup execute an automatic complete restart when power returns.
- Global data can be received and passive unilateral communication using communication SFBs for configured connections and communication SFCs for non-configured connections is possible (see also Table 9-5).

9.6 Testing the User Program

Introduction

The operating system supports you when testing and debugging the user program as follows:

- It provides information about the program.
- It allows you to monitor and modify variables in your user program.

Test Functions

Table 9-6 shows the various options available for testing your program in STEP 7. For more detailed information about testing and debugging user programs, refer to the manuals for the programming languages **/232/**, **/233/** and **/250/** through **/254/** and the STEP 7 user manual **/231/**.

Table 9-6 Testing the User Program

Test Function	Description
Display program status	Displays the program status for each statement (for example result of logic operation RLO, status bit, content of the registers and accumulators).
Set trigger points, monitor and control variables	Allows you to display and modify variables (addresses) at certain points in the program.
Display diagnostic buffer	Allows you to evaluate errors and reasons for changing to the STOP mode.
Display stack contents	Allows you to evaluate the contents of the B stack, I stack and L stack.
Display cycle times	Allows you to check the selected minimum cycle time, the maximum and current cycle time.
Display operating mode	Allows you to display the current operating mode of the CPU

10 Multicomputing

What Does This Chapter Describe?

This chapter describes the following:

- What multicomputing means
- How interrupt servicing functions
- Points to remember when configuring the system
- How to configure modules for the multicomputing mode
- Points to remember when programming
- How the CPUs are synchronized
- What errors can occur in the multicomputing mode

Chapter Overview

Section	Description	Page
10.1	Overview	10-2
10.2	Configuring Modules	10-4
10.3	Programming the CPUs	10-6
10.4	Synchronizing the CPUs	10-8
10.5	Dealing with Errors	10-10

10.1 Overview

Introduction

The multicomputing mode of the S7-400 means simultaneous operation of more than one CPU (up to a maximum of four) in one central rack. This mode allows you to distribute the user program and run it synchronized on several CPUs. In the multicomputing mode:

- The CPUs change their operating modes automatically and mode changes are synchronized with each other.
- The individual CPUs can access the modules assigned to them during configuration with STEP 7.
- All the events occurring on one CPU are passed on to the other CPUs as programmed.

Note

Simultaneous unsynchronized operation of more than one CPU in a segmented rack (physically segmented, cannot be set by user) is also possible. This is, however, not multicomputing. The CPUs in a segmented rack form their own subsystem and behave like single processors. There is no shared address area.

The “multicomputing mode” and “unsynchronized operation in a segmented rack” at the same time is not possible.

When to Use Multicomputing

Multicomputing has advantages in the following situations:

- When your user program is too large for one CPU and memory is used up, distribute your program on more than one CPU.
- If part of your system must be processed quickly, take these program sections out of the main program and run them on a separate fast CPU.
- If your system consists of various parts that can be clearly delineated and can be controlled relatively autonomously, run the program for system section 1 on CPU 1 and system section 2 on CPU 2 etc.

Special Features

The following features characterize multicomputing:

- You can operate up to four CPUs at the same time in the central rack.
- You can plug the CPUs into the rack in any order.
- Each individual CPU has its own interrupt line.
- All CPUs are in the same operating mode.
- When the CPUs exit the STOP mode, the startup types are compared (COMPLETE RESTART / RESTART). This prevents one or more CPUs going through a COMPLETE RESTART while others go through a RESTART.
- The CPUs are interconnected via the K bus (corresponds to a connection via MPI).

Example

Figure 10-1 shows a programmable controller that will operate in the multicomputing mode. Each CPU can access the modules assigned to it (FM, CP, SM).

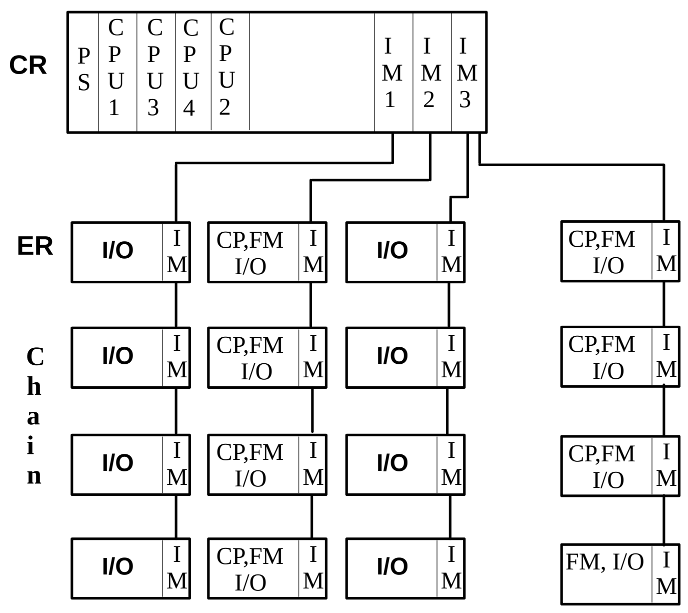


Figure 10-1 Example of Multicomputing

10.2 Configuring Modules

Requirements

Before you can configure modules in your programmable controller for the multicomputing mode, the following condition must be satisfied:

- You have configured your programmable controller as described in the manual **/100/**.

Configuring with STEP 7

To operate CPUs in the multicomputing mode, remember the following points when configuring the programmable controller:

- Plug in the CPUs that you want to operate in the multicomputing mode.
- When you then assign parameters to the modules with STEP 7, you must assign the modules to specific CPUs. This automatically assigns interrupts.
- Parameter assignment for all CPUs must be completed when the configuration is downloaded to the programmable controller.

The configuration procedures are described in the manual **/231/**.

Interrupt Assignment

In the multicomputing mode, each CPU is assigned an interrupt input. Interrupts received at this input cannot be received by the other CPUs. The assignment of a module to a particular CPU must be made in STEP 7. The assignment of the interrupt line is made automatically during parameter assignment.

Figure 10-2 illustrates this assignment.

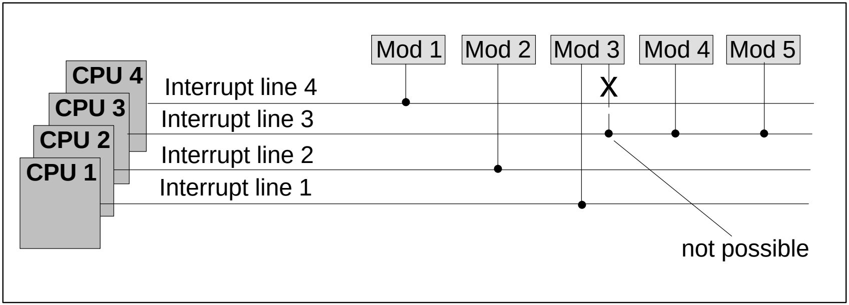


Figure 10-2 Interrupt Assignment in the Multicomputing Mode

Servicing Interrupts

The following rules apply to interrupt servicing:

- Hardware interrupts and diagnostic interrupts are sent to only one CPU.
- If there is a module failure, the interrupt is services by the CPU to which the module was assigned with STEP 7.
- If a rack fails, OB86 is called on every CPU.

Interrupts can be passed on to other CPUs using SFC35 ”MP_ALM” (see Section 10.3).

10.3 Programming the CPUs

Programming

Programming for the multicomputing mode is essentially the same as programming a single CPU.

Extra steps are, however, necessary if you want to synchronize the CPUs so that they react to events together.

Calling SFC35

If you want all the CPUs to react to events (for example interrupts) together in the multicomputing mode, you program an SFC35 “MP_ALM” call. Calling SFC35 triggers a multicomputing interrupt that causes a synchronized request for OB60 on all CPUs. This OB contains local variables that specify the triggering event in greater detail.

When SFC35 is called, the information about the events is transferred to all CPUs in a job identifier. The job identifier allows 16 different events to be distinguished.

When they service the multicomputing interrupt, both the sending user program and the user programs on the other CPUs check whether or not they recognize the job and then react as programmed.

You can call SFC35 at any point in your program. Since the call is only of practical use in the RUN mode, the multicomputing interrupt is suppressed if it is triggered in the STARTUP mode.

The multicomputing interrupt can only be can only be triggered again after the current multicomputing interrupt has been serviced (acknowledged).

The manual **/235/** contains a detailed description of SFC35 and the structure of OB60.

Example

The following example illustrates the use of SFC35.

- You have a programmable controller (S7-400) with four CPUs.
- If there is a hardware interrupt on CPU 1, you want the other three CPUs to react as well.

Step	Explanation	Call/Parameters
1.	On CPU 1, program an SFC35 call in the hardware interrupt OB (OB40).	CALL SFC35
2.	Supply the call with a job identifier that informs the other user programs how they should react to the event.	JOB :=JOB
3.	Check whether an SFC35 is currently active using the RETVAL parameter.	RET_VAL :=RETVAL #RETVAL
4.	SFC35 causes the synchronized start of OB60 on all CPUs. In OB60, the job identifier sent by CPU 1 is evaluated and the CPU programs react accordingly.	The job identifier is in the local data OB60_JOB

Programming OB60

You can program a specific OB60 for each separate CPU and load it on the CPU. This means that the execution times can differ from CPU to CPU. This leads to the following behavior:

- The interrupted priority classes on the CPUs are continued at different times.
- A multicomputing interrupt is not serviced if it occurs during the execution of OB60 on any of the CPUs. A message is, however, generated and you can check this and react accordingly (see example step 3 RETVAL).

If OB60 is not loaded on one of the CPUs, this CPU returns immediately to the last priority class and continues the program there.

10.4 Synchronizing the CPUs

Overview

In the multicomputing mode, the involved CPUs are synchronized automatically, in other words, the individual CPUs are forced to adopt the same operating mode. If, for example, one CPU changes to the STOP mode, all the CPUs are set to STOP. This synchronization in the multicomputing mode makes use of two synchronization points:

- Message points (MP).
- Wait points (WP).

Message Points (MP)

Message points ensure that the CPUs change to STOP as soon as possible. At a message point, one CPU signals a particular event to the other CPUs. The message has high priority and causes all the other CPUs to interrupt the user programs at the next command boundary.

Wait Points (WP)

Wait points ensure that the user programs on all CPUs are started together and that an operating mode change only takes place when all synchronized CPUs are ready to change.

Wait points guarantee simultaneous activities. If, for example, one CPU cannot change mode immediately, the mode change is delayed on the other CPUs until the last CPU is ready.

Synchronization Points of a CPU

There are several synchronization points at the mode transitions. Figure 10-3 shows these synchronization points and their functions are explained in Table 10-1.

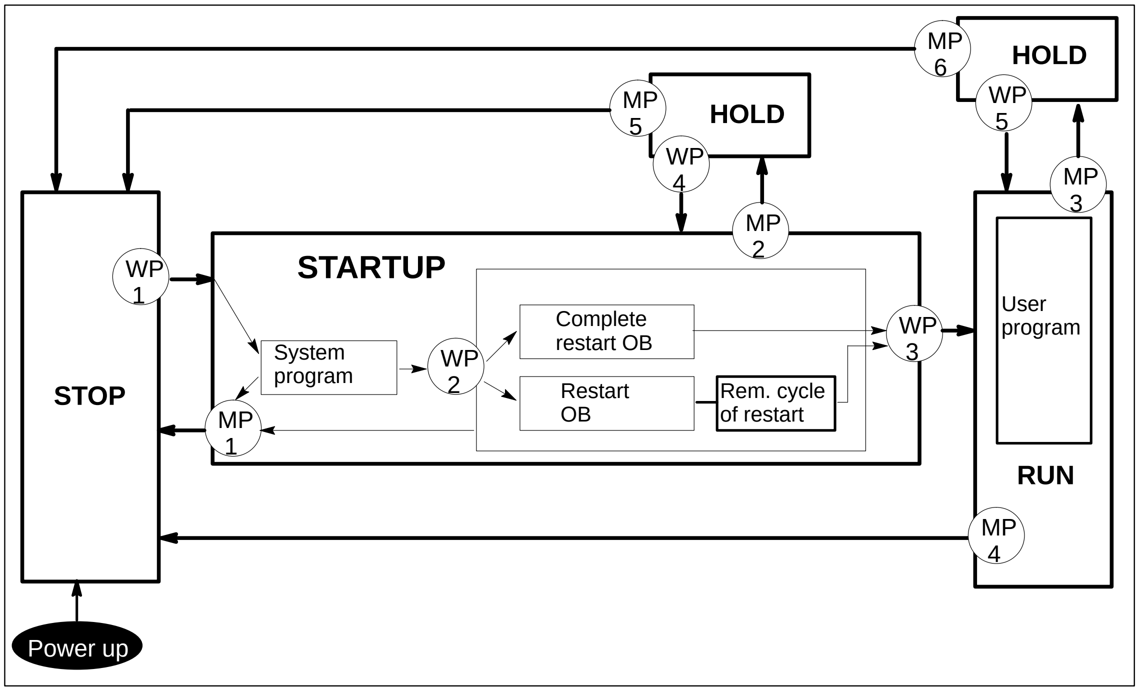


Figure 10-3 Synchronization Points of an S7-400 CPU

Table 10-1 Explanation of the Synchronization Points

Synchronization Point	Explanation
Wait point 1 (WP1)	All CPUs exit the STOP mode at the same time. At this point there is a check to make sure that the same startup type was selected on all CPUs. If different startup types are selected, the startup of the programmable controller is prevented.
Wait point 2 (WP2)	All CPUs start up together. Among other things, make sure that a CPU does not access semaphores in the user program that another CPU deletes during startup due to the different run times of their system programs.
Wait point 3 (WP3)	All CPUs change from STARTUP to RUN at the same time.
Wait points 4 and 5 (WP4/WP5)	All CPUs change from HOLD to STARTUP or RUN at the same time. Execution of the user program is started again at the same time.
Message point 1 (MP1)	The selected startup was stopped due to a system or or user program error. A message is sent to all CPUs.
Message points 2 and 3 (MP2/MP3)	The CPU changes to the HOLD mode. All CPUs are instructed to interrupt their programs at the next command boundary and to change to the HOLD mode.
Message point 4 (MP4)	The CPU changes to the STOP mode. All CPUs are instructed to interrupt their programs at the next command boundary and to change to the STOP mode.
Message points 5 and 6 (MP5/MP6)	The CPU changes to the STOP mode as a result of the mode selector on one or more CPUs being set to the STOP mode. All CPUs are instructed to change to the STOP mode as well.

10.5 Dealing with Errors

Overview

Undesired statuses can occur in the multicomputing mode. Possible causes of errors are described below:

- If a CPU is prevented from starting up, no other CPU starts up since all CPUs change to STOP if one of the CPUs is in the STOP mode.
- If the mode selector on a CPU is set to STOP or if the CPU chages to STOP as a result of a command from the programming device, all other CPUs are also in the STOP mode.
- If the startup type (COMPLETE RESTART / RESTART) is not uniform, no startup takes place.
- If a CPU changes to the STOP mode due to an error or fault, the error/fault must first be eliminated, The other CPUs will only change from STOP to RUN when the CPU that had the problem changes to RUN.

Checking the Consistency of the CPUs

In the multicomputing mode, all CPUs undergo a consistency check. This checks whether the individual CPUs are obtainable and that the time stamps of the CPUs are identical.

If the existing CPUs are not consistent, an event with ID: 0x49A4 is signaled. Depending on the type of entry in the diagnostic buffer, this means:

- A CPU slot is not recorded.
- A CPU is not plugged in or is defective.
- The time stamps of the individual CPUs are not consistent.

The manual **/235/** explains the meaning of the event IDs.

Diagnostics and Troubleshooting 11

What Does This Chapter Describe?

This chapter describes the following:

- System diagnostics on the S7-300 and S7-400 CPUs. The chapter also tells you how to eliminate errors that have been detected and how to deal with various problems.
- Asynchronous and synchronous error OBs.

Where to Find More Information

It is also possible to identify problems based on the display elements on the front panel of the modules. This is, however, beyond the scope of this chapter. For more information refer to the manuals **/70/**, **/71/** or **/101/**.

For a detailed description of the individual organization blocks and system functions, refer to the reference manual **/235/**.

Chapter Overview

11.1 Diagnostic Information

The Aim of Diagnostics

SIMATIC S7 system diagnostics helps you to detect, localize and evaluate errors and problems and to reduce the commissioning time and down times in your system.

Diagnostic Functions

You do not need to program the acquisition of diagnostic data by system diagnostics. This is a standard feature that runs automatically. SIMATIC S7 provides various diagnostic functions. Some of these functions are integrated on the CPU, others are provided by the modules (SMs, CPs and FMs).

Internal and external module faults are displayed on the front panels of the module. The LED displays and how to evaluate them are described in the manuals **/70/**, **/71/** and **/101/**.

The CPU recognizes system errors and errors in the user program and enters diagnostic messages in the system status list and the diagnostic buffer. You can read out these diagnostic messages on the programming device.

Signal and function modules with diagnostic capability detect internal and external module errors and generate a diagnostic interrupt to which you can react using an interrupt OB. Figure 11-1 shows how diagnostic information is provided in SIMATIC S7.

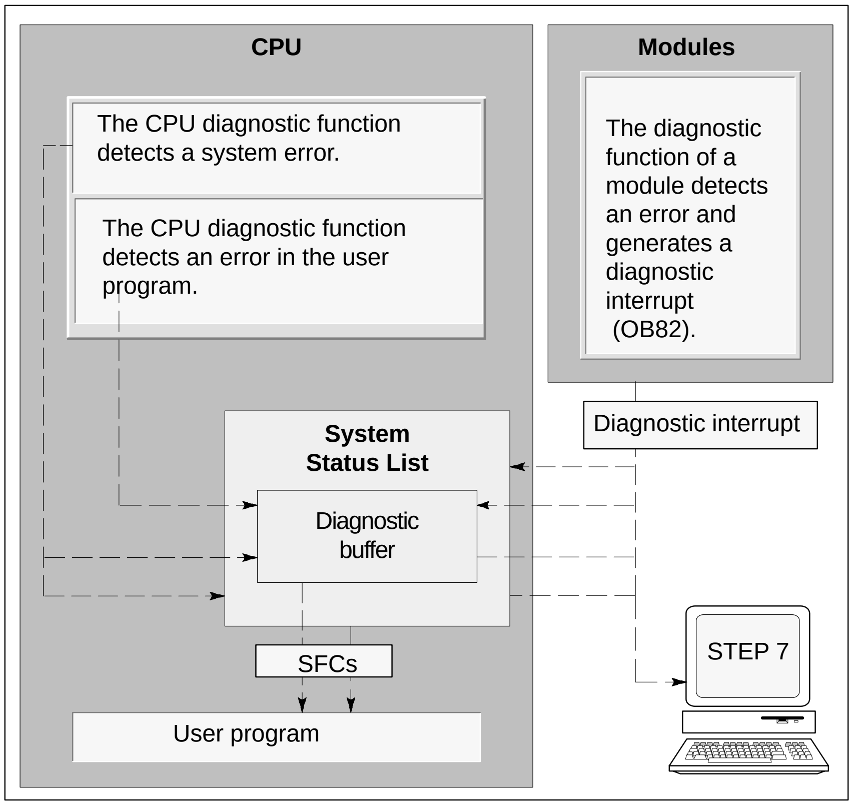

CPU
Modules
The CPU diagnostic function detects a system error.
The CPU diagnostic function detects an error in the user program.
The diagnostic function of a module detects an error and generates a diagnostic interrupt (OB82).
Diagnostic interrupt
System Status List
Diagnostic buffer
SFCs
User program
STEP 7

Figure 11-1 Flow of Diagnostic Information

Diagnostic Event

A diagnostic event causes a diagnostic message from the CPU or a diagnostic interrupt from a signal or function module. Diagnostic events include the following:

- Internal and external faults on a module
- System errors
- Operating mode changes
- Errors in the user program
- Inserting/removing modules

Reading Out the Diagnostic Information

You can read out the diagnostic entries using SFC51 RDSYSST in the user program or display the diagnostic messages in plain language with STEP 7.

They provide information about the following:

- Where and when the error occurred
- The type of diagnostic event to which the entry belongs (user-defined diagnostic event, synchronous/asynchronous error, operating mode change).

Generating Process Control Group Messages

The CPU enters events of the standard diagnostics and extended diagnostics (see Section 11.3) in the diagnostic buffer. It also generates a process control group message for the standard diagnostic events if the following conditions are met:

- You have specified that process control messages will be generated in STEP 7.
- At least one display unit has logged on at the CPU for process control messages.
- A process control group message is only generated when there is not currently a process control group message of the corresponding class (there are seven classes).
- One process control group message can be generated per class.

11.2 System Status List SZL

Definition

The system status list (SZL) describes the current status of the programmable logic controller. It provides an overview of the configuration, the current parameter assignment, the current statuses and sequences on the CPU and the modules belonging to it.

You can only read the data of the system status list but not modify it. It is a virtual list that is only created on request.

Content

The information that you can display using the system status list can be divided into four areas. Figure 11-2 shows the structure of the system status list:

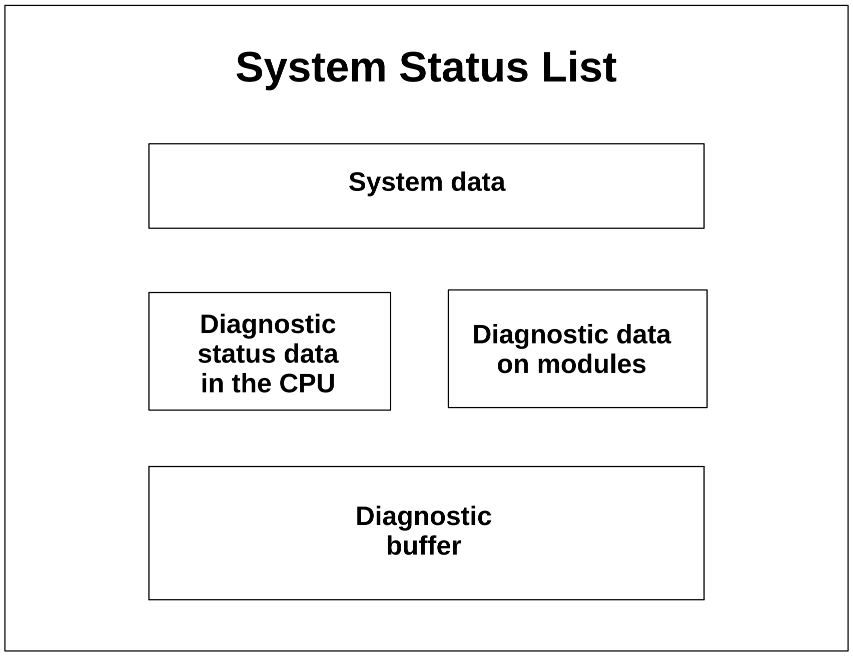


Figure 11-2 System Status List

Reading out the SZL

There are two ways of reading out the information in partial system status lists, as follows:

- Implicitly, using the STEP 7 menu option on the programming device (for example, memory configuration, static CPU data, diagnostic buffer, status bits).
- Explicitly, using the System function SFC51 RDSYSST in the user program by specifying the required partial list number. SFCs are described in detail in the reference manual **/235/**.

System Data

System data are intrinsic or assigned characteristic data of a CPU. Table 11-1 shows the topics about which information can be displayed (partial system status list):

Table 11-1 System Data of the System Status List

Topic	Information
List of all SZL-IDs	Partial lists available for a module
Module identification	Order number, type ID and version of the module
CPU characteristics	Time system, system response, (for example multicomputing) and language description of the CPU
Memory areas	Memory configuration of the module (for example size of the work memory, load memory integrated/plugged in, size of the backup memory)
System areas	System memory of the module (for example number of memory bits, timers, counters, memory type)
Block types	Which blocks (OB, DB, SDB, FC, FB) exist on the module, the maximum number of blocks of one type and the maximum size of a block type
Existing priority classes	Which priority classes exist on the module
List of permitted system data blocks (SDBs)	Which SDBs exist on the module, can be copied/cannot be copied, whether or not generated as default
I/O configuration (only S7-300 CPUs)	Maximum I/O configuration, how many racks, number of slots
Assignment of interrupts and errors	Assignment of interrupts/errors to OBs
Interrupt status	Current status of interrupt processing/interrupts generated
Status of the priority classes	Which OB is being executed, which priority class is disabled due to the parameter setting
Operating mode and mode transition	Which operating modes are possible, the last operating mode change, the current operating mode
Capability parameters for communication	Communication options available (for example operator interface O/I)

Diagnostic Status Data on the CPU

Diagnostic status data describe the current status of the components monitored by the system diagnostics. Table 11-2 shows the topics about which information can be displayed (partial system status lists) :

Table 11-2 Diagnostic Status Data of the System Status List

Topic	Information
Communication status data	All the communication functions currently set in the system
Diagnostic modules	The modules with diagnostic capability logged on at the CPU
Start information list of the OB	Start information about the OBs of the CPU
Start event list	Start events and priority classes of the OBs
Module status information	Status information about all assigned modules that are plugged in, faulty, or generate hardware interrupts

Diagnostic Data on Modules

In addition to the CPU, there are also other modules with diagnostic capabilities (SMs, CPs, FMs) whose data are entered in the system status list. Table 11-3 shows the topics about which information can be displayed (partial system status lists).

Table 11-3 Module Diagnostic Data of the System Status List

Topic	Information
Module diagnostic information	Module start address, internal/external faults, channel faults, parameter errors (4 bytes)
Module diagnostic data	All the diagnostic data of a particular module

Diagnostic Buffer

The diagnostic buffer of the CPU contains diagnostic messages in the order in which they occur. For more information about evaluating the diagnostic buffer, refer to Section 11.3.

Table 11-4 Diagnostic Buffer of the System Status List

Topic	Information
Diagnostic events grouped by topics	For example the most recent events, start messages from standard OBs, operating mode changes, user-defined events

11.3 Diagnostic Buffer

Definition

One part of the system status list is the diagnostic buffer that contains more information about system diagnostic events and user-defined diagnostic events in the order in which they occurred. The information entered in the diagnostic buffer when a system diagnostic event occurs is identical to the start information transferred to the corresponding organization block.

The length of the diagnostic buffer depends on the particular CPU. It is designed as a ring buffer, in other words, if the buffer is full, the next entry overwrites the oldest entry in the buffer.

You cannot clear the entries in the diagnostic buffer and its contents are retained even after a memory reset.

Uses

The diagnostic buffer provides you with the following possibilities:

- If the CPU changes to the STOP mode, you can evaluate the last events leading up to the STOP and locate the cause.
- The causes of errors can be detected far more quickly increasing the availability of the system.
- You can evaluate and optimize the dynamic system response.

Reading Out the Diagnostic Buffer

You can read out the diagnostic buffer with STEP 7 or using SFC51 RDSYSST.

Last Entry Before STOP

You can specify that the last diagnostic buffer entry before the transition from RUN to STOP is signaled automatically to a logged on display device (for example programming device, OP, TD). This helps to locate and remedy the cause of the change to STOP more quickly (see also Section 11.4).

11.4 Sending Your Own Diagnostic Messages

Introduction

You can also extend the standard system diagnostic functions of SIMATIC S7 using the system function SFC52 WR_USMSG as follows:

- To enter your own diagnostic information in the diagnostic buffer (for example information about the execution of the user program).
- To send user-defined diagnostic messages to logged on stations (monitoring devices such as a PG, OP or TD).

For more information about assigning parameters to SFC52, refer to the reference manuals **/235/**.

User-Defined Diagnostic Events

The diagnostic events are divided into event classes 1 to F. The user-defined diagnostic events belong to event classes 8 to B. These can be divided into two groups, as follows:

- Event classes 8 and 9 include messages with a fixed number and predefined text that you can call up based on the number.
- Event classes A and B included messages with freely selectable numbers (A00 to BFF) and freely selectable text.

Sending Diagnostic Messages to Stations

In addition to making a user-defined entry in the diagnostic buffer, you can also send your own user-defined diagnostic messages to logged on display devices using SFC52 WR_USMSG. When SFC52 is called with SEND = 1, the diagnostic message is written to the transmit buffer and automatically sent to the station or stations logged on at the CPU.

If it is not possible to send a diagnostic message (for example because no station is logged on or because the transmit buffer is full), the user-defined entry is nevertheless made in the diagnostic buffer.

Generating a Message with Acknowledgment

If you acknowledge a user-defined diagnostic event and want to record the acknowledgment, follow the steps below:

- When the event enters the event state, write 1 to a variable of the type BOOL, when the event leaves the event state write 0 to the variable.
- You can then monitor this variable using SFB33 ALARM.

For information about using SFB33, refer to the reference manual **/235/**.

11.5 Evaluating the Output Parameter RET_VAL

Introduction

Using the RET_VAL output parameter (return value), a system function indicates whether or not the CPU was able to execute the SFC function correctly.

Error Information in the Return Value

The return value is of the integer data type (INT). The sign of an integer indicates whether it is a positive or negative integer. The relationship of the return value to the value "0" indicates whether or not an error occurred while the function was being executed (see also Table 11-5):

- If an error occurs while the function is being executed, the return value is less than 0. The sign bit of the integer is "1".
- If the function is executed free of errors, the return value is greater than or equal to 0. The sign bit of the integer is "0".

Table 11-5 Error Information in the Return Value

Execution of the SFC by the CPU	Return Value	Sign of the Integer
Error occurred	Less than "0"	Negative (sign bit is "1")
No error	Greater than or equal to "0"	Positive (sign bit is "0")

Reacting to Error Information

If an error occurs while SFC is being executed, the SFC provides an error code in the return value (RET_VAL).

A distinction is made between the following:

- A general error code that all SFCs can output.
- A specific error code that the SFC can output depending on its specific function.

Transferring the Function Value

Some SFCs also use the output parameter RET_VAL to transfer the function value, for example SFC64 TIME_TCK transfers the system time it has read using RET_VAL.

Further Information

For a detailed description of the RET_VAL output parameter and the meaning of the error codes of a return value, refer to the reference manual **/235/**.

11.6 Error OBs as a Reaction to Detected Errors

Detectable Errors

The system program can detect the following errors:

- CPU functioning incorrectly
- Error in the system program execution
- Error in the user program
- Error in the I/Os

Depending on the type of error, the CPU is set to the STOP mode or an error OB is called.

Programming Reactions

You can design programs to react to the various types of errors and to determine the way in which the CPU reacts. The program for a particular error can then be saved in an error OB. If the error OB is called, the program is executed.

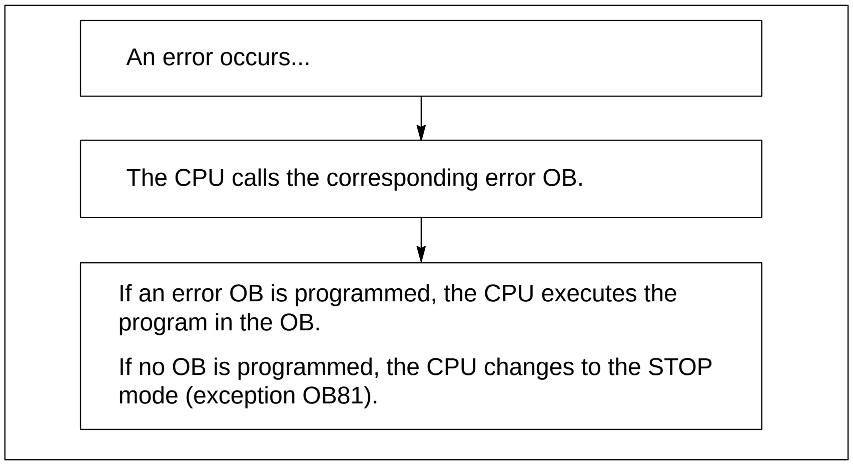

An error occurs...
The CPU calls the corresponding error OB.
If an error OB is programmed, the CPU executes the program in the OB.
If no OB is programmed, the CPU changes to the STOP mode (exception OB81).

Figure 11-3 Error OBs as a Reaction to Detected Errors

Error OBs

A distinction is made between synchronous and asynchronous errors as follows:

- Synchronous errors can be assigned to an MC7 instruction (for example load instruction for a signal module that has been removed).
- Asynchronous errors can be assigned to a priority class or to the entire programmable logic controller (for example cycle time exceeded).

Table 11-6 shows what types of errors can occur. Refer to the CPU descriptions in the manuals **/70/** or **/101/** to find out whether your CPU provides the specified OBs.

Table 11-6 Error OBs

Error Class	Error Type	OB	Priority
Asynchronous	Time error	OB80	26 (or 28 if the error OB is called in the startup program)
	Power supply error	OB81	
	Diagnostic interrupt	OB82	
	Remove/insert module interrupt	OB83	
	CPU hardware fault	OB84	
	Priority class error	OB85	
	Rack failure	OB86	
	Communication error	OB87	
Synchronous	Programming error	OB121	Priority of the OB that caused the error
	Access error	OB122	

Example of Using Error OB81

Using the local data (start information) of the error OB, you can evaluate the type of error that has occurred.

If, for example, the CPU detects a battery fault, the operating system calls OB81 (see Figure 11-4). You can write a program that evaluates the event code triggered by the OB81 call. You can also write a program that brings about a reaction, such as activating an output connected to a lamp on the operator station.

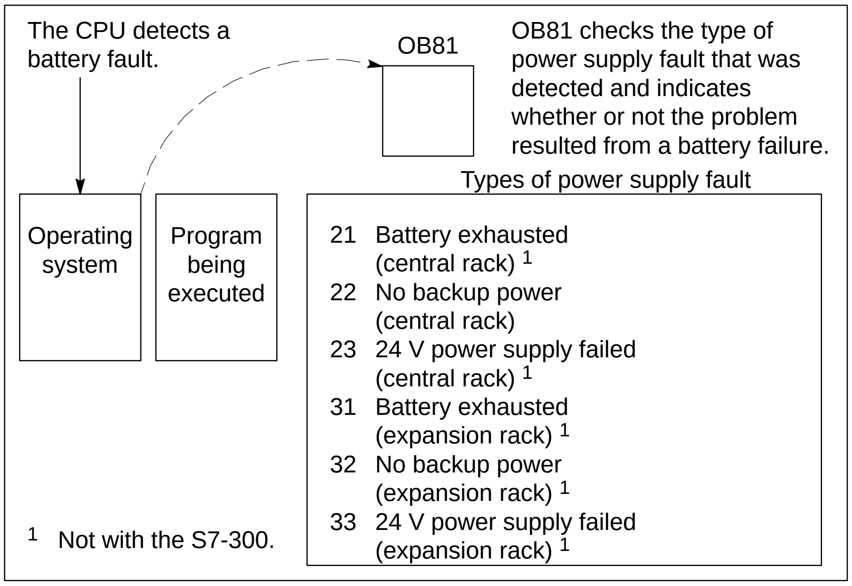

The CPU detects a battery fault.
OB81
OB81 checks the type of power supply fault that was detected and indicates whether or not the problem resulted from a battery failure.
Types of power supply fault
Operating system
Program being executed
21 Battery exhausted (central rack) 1
22 No backup power (central rack)
23 24 V power supply failed (central rack) 1
31 Battery exhausted (expansion rack) 1
32 No backup power (expansion rack) 1
33 24 V power supply failed (expansion rack) 1
1 Not with the S7-300.

Figure 11-4 Using the Local Data of an Error OB

Local Data of the Error OB81

Table 11-7 describes the temporary (TEMP) variables declared in the variable declaration table of OB81.

The symbol *Battery_error* (BOOL) must also be identified as an output in the symbol table (for example Q 4.0) so that other parts of the program can access this data.

Table 11-7 Variable Declaration Table of OB81

Decl.	Name	Type	Description
TEMP	OB81_EV_CLASS	BYTE	Error class/error identifier 39xx
TEMP	OB81_FLT_ID	BYTE	Error code: b#16#21 = At least one backup battery of the CPU is exhausted [1] b#16#22 = No backup voltage in the CPU b#16#23 = Failure of the 24 V power supply in the CPU [1] b#16#31 = At least one backup battery of an expansion rack is exhausted [1] b#16#32 = Backup voltage not present in an expansion rack [1] b#16#33 = Failure of the 24 V power supply of an expansion rack [1]
TEMP	OB81_PRIORITY	BYTE	Priority class = 26/28
TEMP	OB81_OB_NUMBR	BYTE	81 = OB81
TEMP	OB81_RESERVED_1	BYTE	Reserved
TEMP	OB81_RESERVED_2	BYTE	Reserved
TEMP	OB81_MDL_ADDR	INT	Reserved
TEMP	OB81_RESERVED_3	BYTE	Only relevant for error codes B#16#31, B#16#32, B#16#33
TEMP	OB81_RESERVED_4	BYTE	
TEMP	OB81_RESERVED_5	BYTE	
TEMP	OB81_RESERVED_6	BYTE	
TEMP	OB81_DATE_TIME	DATE_AND_ TIME	Date and time at which the OB was started

[1] Not with the S7-300.

Sample Program for the Error OB81

The sample STL program shows how you can read the error code in OB81.

The program is designed as follows:

- The error code in OB81 (OB81_FLT_ID) is read and compared with the value of the event "battery exhausted" (B#16#3921).
- If the error code corresponds to the code for "battery exhausted", the program jumps to the label Berr and activates the output *Battery_error.*
- If the error code does not correspond to the code for "battery exhausted", the program compares the code with the code for "Battery failure".
- If the error code corresponds to the code for "Battery failure" the program jumps to the label Berr and activates the output *Battery_error.* Otherwise the block is terminated.

STL			Description
	L	B#16#3921	Compare event code "battery exhausted"
	L	#OB81_FLT_ID	(B#16#3921) with the error code for OB81.
	== I		If the same (battery is exhausted), then
	JC	Berr	jump to Berr.
	L	b#16#3922	Compare event code "battery failure"
	<> I		(b#16#3922) with the error code for OB81.
	BEC		If not the same (no battery failure in the CPU), then terminate the block.
Berr:	S	#Battery_error	Berr sets the output "Battery_error" if a battery failure or an exhausted battery is detected.

Note

The error codes of all organization blocks are described in the STEP 7 online help and in the reference manual **/235/**.

11.7 Using “Replacement Values” When an Error is Detected

Overview

With certain types of error (for example a wire break affecting an input signal), you can supply replacement values for values that are not available due to the error. There are two ways of supplying replacement values:

- You can assign replacement values for configurable output modules using STEP 7. Output modules that cannot be configured have the default replacement value 0.
- Using SFC 44 RPL_VAL, you can program replacement values in error OBs (only for input modules).

For all load instructions that lead to synchronous errors, you can specify a replacement value for the accumulator content in the error OB.

Sample Program for Replacing a Value

In the following sample program, a replacement value is made available in SFC 44 RPL_VAL. Figure 11-5 shows how OB122 is called when the CPU recognizes that an input module is not reacting. In this example, the replacement value in Figure 11-6 is entered in the program so that the program can continue to operate with feasible values.

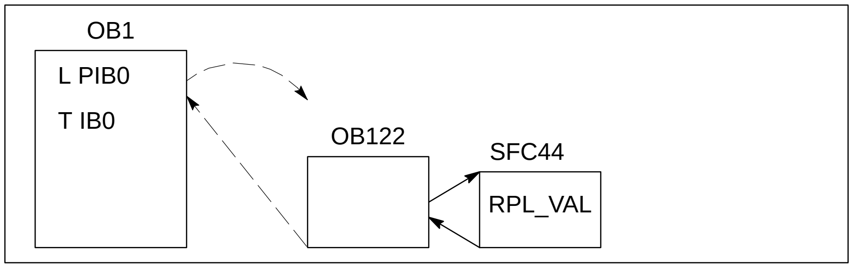


Figure 11-5 Using a Replacement Value

If an input module fails, the L_PIB0 statement generates a synchronous error and starts OB122. As standard, the load instruction reads in the value 0. With SFC44, however, you can define any replacement value suitable for the process. The SFC replaces the accumulator content with the specified replacement value.

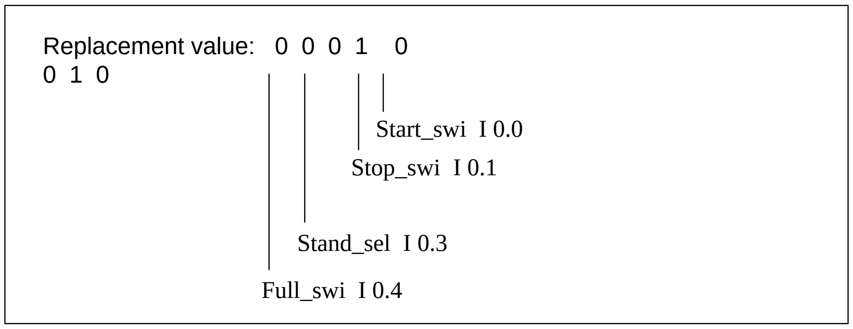


Figure 11-6 Examples of Replacement Values in the Program

The following sample program could be written in OB122. Table 11-8 shows the temporary variables that must be declared, in this case, in the variable declaration table of OB122.

Table 11-8 Local Variables (TEMP) of OB122

Decl.	Name	Type	Description
TEMP	OB122_EV_CLASS	BYTE	Error class/error ID 29xx
TEMP	OB122_SW_FLT	BYTE	Error code: 16#42, 16#43, 16#44 [1], 16#45 [1]
TEMP	OB122_PRIORITY	BYTE	Priority class = priority of the OB in which the error occurred
TEMP	OB122_OB_NUMBR	BYTE	122 = OB122
TEMP	OB122_BLK_TYPE	BYTE	Block type in which the error occurred
TEMP	OB122_MEM_AREA	BYTE	Memory area and type of access
TEMP	OB122_MEM_ADDR	WORD	Address in the memory at which the error occurred
TEMP	OB122_BLK_NUM	WORD	Number of the block in which the error occurred
TEMP	OB122_PRG_ADDR	WORD	Relative address of the instruction that caused the error
TEMP	OB122_DATE_TIME	DATE_AND_ TIME	Date and time at which the OB was started
TEMP	Fehler	INT	Saves the error code of SFC44

[1] Not for the S7-300.

Note

The error codes of all organization blocks are described in the STEP 7 online help and in the reference manual **/235/**.

STL	Description

`L B#16#2942`
`L #OB122_SW_FLT`
`==I`
`JC Aerr`
	Compare the event code of OB122 with the event code (B#16#2942) for the acknowledgment of a time error when reading the I/Os. If the same, jump to “Aerr”.

`L B#16#2943`
`<> I`
`JC Stop`
	Compare the event code of OB122 with the event code (B#16#2943) for an addressing error (writing to a module that does not exist). If not the same, jump to “Stop”.

`Aerr: CALL “REPL_VAL”`
`VAL : = DW#16#2912`
`RET_VAL : = #Error`
`L #Error`
`L 0`
`==I`
`BEC`
	Label “Aerr”: transfers DW#16#2912 (binary 10010) to SFC44 (REPL_VAL). SFC44 loads this value in accumulator 1 (and replaces the value triggered by the OB122 call). The SFC error code is saved in #Error.

	Compare #Error with 0 (if the same, no error occurred when executing OB122). Terminate the block if no error occurred.
`Stop: CALL “STP”`	”Stop” label: calls SFC46 “STP” and changes the CPU to the STOP mode.

11.8 Time Error OB (OB80)

Description

The operating system of the CPU calls OB80 when a time error occurs. Time errors include the following:

- Maximum cycle time exceeded (see also Section 8.4)
- Time-of-day interrupts skipped by moving the time forward
- Delay too great when processing a priority class

Programming OB80

The time error OB (OB80) must be generated as an object in your S7 program using STEP 7. Write the program to be executed in OB80 in the generated block and download it to the CPU as part of your user program.

You can use OB80, for example, for the following purposes:

- To evaluate the start information of OB80 and to determine which time-of-day interrupts were skipped.
- By including SFC29 CAN_TINT, you can deactivate the skipped time-of-day interrupt so that it is not executed and only time-of-day interrupts relative to the new time will be executed.

If you do not deactivate skipped time-of-day interrupts in OB80, the first skipped time-of-day interrupt is executed, all others are ignored (see also Section 4.2).

If you do not program OB80, the CPU changes to the STOP mode when a time error is detected.

11.9 Power Supply Error OB (OB81)

Description

The operating system of the CPU calls OB81 when one of the following fails on the CPU or in an expansion rack:

- The 24 V power supply
- A battery
- The complete backup

This OB is also called when the problem has been eliminated (the OB is called when an event comes and goes).

Programming OB81

You must generate the power supply error OB (OB81) as an object in your S7 program using STEP 7. Write the program to be executed in OB81 in the generated block and download it to the CPU as part of your user program.

You can, for example, use OB81 for the following purposes:

- To evaluate the start information of OB81 and determine which power supply error has occurred.
- To find out the number of the rack with the defective power supply.
- To activate a lamp on an operator station to indicate that maintenance personnel should replace a battery.

If you do not program OB81, the CPU does not change to the STOP mode if a power supply error is detected, in contrast to all other asynchronous error OBs. The error is, however, entered in the diagnostic buffer and the corresponding LED on the front panel indicates the error.

11.10 Diagnostic Interrupt OB (OB82)

Description

The operating system of the CPU calls OB82 when a module with diagnostic capability on which you have enabled the diagnostic interrupt detects an error and when the error is eliminated (OB called when the event comes and goes).

Programming OB82

You must generate the diagnostic interrupt OB (OB82) as an object in your S7 program using STEP 7. Write the program to be executed in OB82 in the generated block and download it to the CPU as part of your user program.

You can, for example, use OB82 for the following purposes:

- To evaluate the start information of OB82.
- To obtain exact diagnostic information about the error that has occurred.

When a diagnostic interrupt is triggered, the module on which the problem has occurred automatically enters 4 bytes of diagnostic data and their start address in the start information of the diagnostic interrupt OB and in the diagnostic buffer. This provides you with information about when an error occurred and on which module.

With a suitable program in OB82, you can evaluate further diagnostic data for the module (which channel the error occurred on, which error has occurred). Using SFC51 RDSYSST, you can read out the module diagnostic data and can enter this information in the diagnostic buffer with SFC52 WR_USRMSG. You can also send a user-defined diagnostic message to a monitoring device (see also Section 11.4).

If you do not program OB82, the CPU changes to the STOP mode when a diagnostic interrupt is triggered.

11.11 Insert/Remove Module Interrupt OB (OB83)

Description

S7-400-CPUs monitor the presence of modules in the central rack and expansion racks at intervals of approximately 1 second.

After the power supply is turned on, the CPU checks whether all the modules listed in the configuration table created with STEP 7 are actually plugged in. If all the modules are present, the actual configuration is saved and is used as a reference value for cyclic monitoring of the modules. In each scan cycle, the newly detected actual configuration is compared with the previous actual configuration. If there are discrepancies between the configurations, an insert/remove module interrupt is signaled and an entry is made in the diagnostic buffer and the system status list (see also module monitoring in Section 8.3). In the RUN mode, the insert/remove module interrupt OB is started.

Note

Power supply modules, CPUs and IMs must not be removed in the RUN mode.

Between removing and inserting a module, at least two seconds must be allowed to pass so that the CPU can detect that a module has been removed or inserted.

Assigning Parameters to a Newly Plugged In Module

If a module is inserted in in the RUN mode, the CPU checks whether the module type of the new module matches the original module. If they match, the module is assigned parameters. Either the default parameters or the parameters you assigned with STEP 7 are transferred to the module.

Programming OB83

The insert/remove module interrupt OB (OB83) must be created as an object in your S7 program using STEP 7. Write the program to be executed in OB83 in the generated block and download it to the CPU as part of your user program.

You can use OB83, for example, for the following purposes:

- To evaluate the start information of OB83.
- By including system functions SFC55 to 59, to assign parameters to a newly plugged in module (see also Section 6.2).

If you do not program OB83, the CPU changes from RUN to STOP when an insert/remove module interrupt occurs.

11.12 CPU Hardware Fault OB (OB84)

Description

The operating system of the CPU calls OB84 when an error is detected on the interface to the MPI network, to the K bus, or to the network card for the distributed I/Os,

- For example an incorrect signal level on the line.

This OB is also called when the problem is eliminated (OB called when an event comes and goes).

Programming OB84

You must generate the CPU hardware fault OB (OB84) as an object in your S7 program using STEP 7. Write the program to be executed in OB84 in the generated block and download it to the CPU as part of your user program.

You can use OB84, for example, for the following purposes:

- To evaluate the start information of OB84.
- By including system function SFC52 WR_USMSG to send a message to the diagnostic buffer.

If you do not program OB84, the CPU changes to the STOP mode when a CPU hardware fault is detected.

11.13 Priority Class Error OB (OB85)

Description

The operating system of the CPU calls OB85 in the following situations:

- When a start event for an interrupt OB exists but the OB cannot be executed because it has not been loaded on the CPU.
- When an error occurs accessing the instance data block of a system function block.
- When an error occurs updating the process image table (module does not exist or defective).

Programming OB85

You must generate the priority class error OB (OB85) as an object in your S7 program using STEP 7. Write the program to be executed in OB85 in the generated block and download it to the CPU as part of your user program.

You can use OB85, for example, for the following purposes:

- To evaluate the start information of OB85 and determine which module is defective or not plugged in (the module start address is specified).
- By including SFC49 LGC_GADR to find out the slot of the module involved.

If you do not program OB85, the CPU changes to the STOP mode when a priority class error is detected.

11.14 Rack Failure OB (OB86)

Description

The operating system of the CPU calls OB86 when a rack failure is detected, for example in the following situations:

- Rack failure (missing or defective IM or break on the connecting cable)
- Distributed power failure on a rack
- Failure of a DP slave in a master system of the SINEC L2-DP bus system

The OB is also called when the error is eliminated (OB call when the event comes and goes).

Programming OB86

You must generate the rack failure OB (OB86) as an object in your S7 program using STEP 7. Write the program to be executed in OB86 in the generated block and download it to the CPU as part of your user program.

You can use OB86, for example, for the following purposes:

- To evaluate the start information of OB86 and determine which rack is defective or missing.
- By including system function SFC52 WR_USMSG, to send a message to the diagnostic buffer and to a monitoring device.

If you do not program OB86, the CPU changes to the STOP mode when a rack failure is detected.

11.15 Communication Error OB (OB87)

Description

The operating system of the CPU calls OB87 when a communication error occurs in data exchange using communication function blocks or in global data communication, for example:

- When receiving global data, an incorrect frame ID was detected
- The data block for the status information of the global data does not exist or is too short.

Programming OB87

You must generate the communication error OB (OB87) as an object in your S7 program using STEP 7. Write the program to be executed in OB87 in the generated block and download it to the CPU as part of your user program.

You can use OB87, for example, for the following purposes:

- To evaluate the start information of OB87.
- To create a data block if the data block for the status information of global data communication is missing.

If you do not program OB87, the CPU changes to the STOP mode when a communication error is detected.

11.16 Programming Error OB (OB121)

Description

The operating system of the CPU calls OB121 when a programming error occurs, for example:

- Addressed timers do not exist.
- A called block is not loaded.

Programming OB121

You must generate the programming error OB (OB121) as an object in your S7 program using STEP 7. Write the program to be executed in OB121 in the generated block and download it to the CPU as part of your user program.

You can use OB121, for example, for the following purposes:

- To evaluate the start information of OB121.
- To enter the cause of an error in a message data block.

If you do not program OB121, the CPU changes to the STOP mode when a programming error is detected.

11.17 I/O Access Error OB (OB122)

Description

The operating system of the CPU calls OB122 when a STEP 7 instruction accesses an input or output of a signal module to which no module was assigned at the last complete restart, for example:

- Errors with direct I/O access (module defective or does not exist)
- Access to an I/O address that is not known to the CPU.

Programming OB122

You must generate the I/O access error OB (OB122) as an object in your S7 program using STEP 7. Write the program to be executed in OB122 in the generated block and download it to the CPU as part of your user program.

You can use OB122, for example, for the following purposes:

- To evaluate the start information of OB122
- To call system function SFC44 and supply a replacement value for an output module so that the program has a feasible process-dependent value for further processing.

If you do not program OB122, the CPU changes to the STOP mode when an I/O access error is detected.

Sample Program for an Industrial Blending Process A

What Does This Chapter Describe?

Based on an example, this chapter explains how you could design a program for an industrial blending process. The emphasis is not to “teach programming style” or to provide the technical knowledge required to control a particular process. The example is simply intended to illustrate the steps that must be followed to design a program.

Chapter Overview

Note

The sample program is supplied with a variable table with which you can modify and monitor the individual variables.

A.1 Example of an Industrial Blending Process

Introduction

The sample program is based on the configuration introduced in earlier chapters as an example of an industrial blending process.

Task

Two ingredients (ingredient A and ingredient B) are mixed together in a mixing tank by an agitator. The finished product is drained from the tank through a drain valve. Figure A-1 is a diagram of the sample process.

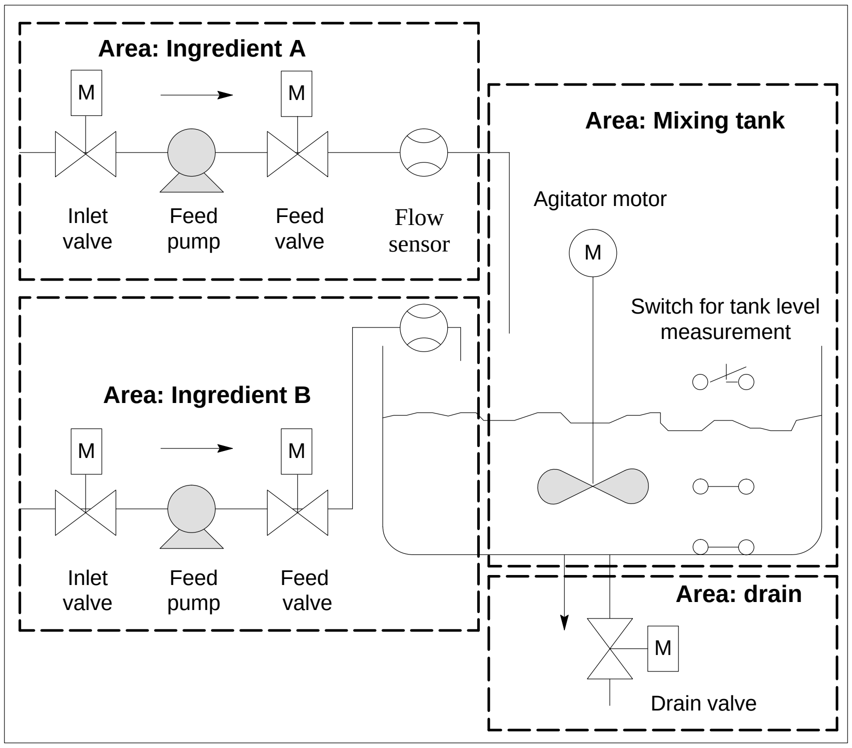


Figure A-1 Defining Areas Within a Process

Describing the Parts of a Process

Section 1.2 described how to divide the sample process into functional areas and individual tasks. The individual areas are described below.

The area for ingredients A and B

- The pipes for each of the ingredients are equipped with an inlet and a feed valve and feed pump.
- The inlet pipes also have flow sensors.
- Turning on the feed pumps must be interlocked when the tank level sensor indicates that the tank is full.

- The activation of the feed pumps must be interlocked when the drain valve is open.
- The inlet and feed valves must be opened at the earliest 1 second after starting the feed pump.
- The valves must be closed immediately after the feed pumps stop (signal from the flow sensor) to prevent ingredients leaking from the pump.
- The activation of the feed pumps is combined with a time monitoring function, in other words, within 7 seconds after the pumps start, the flow sensor must report a flow.
- The feed pumps must be turned off as quickly as possible if the flow sensor no longer signals a flow while the feed pumps are running.
- The number of times that the feed pumps are started must be counted (maintenance interval).

Mixing tank area

- The activation of the agitator motor must be interlocked when the tank level sensor indicates “level below minimum” or the drain valve is open.
- The agitator motor sends a response signal after reaching the rated speed. If this signal is not received within 10 seconds after the motor is activated, the motor must be turned off.
- The number of times that the agitator motor starts must be counted (maintenance interval).
- Three sensors must be installed in the mixing tank:
  - Tank full: a normally closed contact. When the maximum tank level is reached, the contact is opened.
  - Level in tank above minimum: a normally open contact. If the minimum level is reached, the contact is closed.
  - Tank not empty: a normally open contact. If the tank is not empty, the contact is closed.

Drain area

- Drainage of the tank is controlled by a solenoid valve.
- The solenoid valve is controlled by the operator, but must be closed again at the latest when the “tank empty” signal is generated.
- Opening the drain valve is interlocked when
  - the agitator motor is running
  - the tank is empty

Operator Station

To allow an operator to start, stop and monitor the process, an operator station is also required (see also Section 1.5). The operator station is equipped with the following:

- Switches for controlling the most important stages of the process.
- Display lamps to indicate the status of the process.
- The emergency stop switch.

A.2 Defining Logic Blocks

Overview

You structure the program by distributing the user program in various blocks and by establishing a hierarchy for block calls.

Hierarchy of the Block Calls

Figure A-2 shows the hierarchy of the blocks to be called in the structured program.

- The feed pump for ingredient A, the feed pump for ingredient B and the agitator motor can be controlled by a single logic block (FB1).
- The actual parameters and the static data of FB1 are entered in three separate instance DBs, for ingredient A, ingredient B, and for the agitator motor respectively.
- The inlet and feed valves for ingredients A and B and the drain valve also use a common logic block (FC1).

The function block and the function are called in OB1 and the specific parameters required for controlling the process are then transferred.

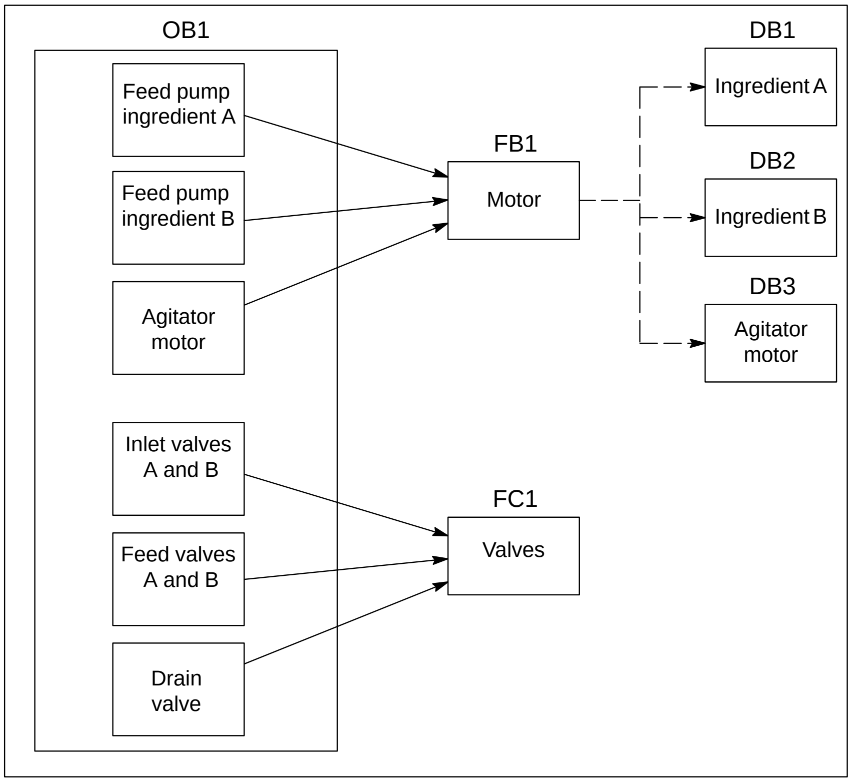


Figure A-2 Specifying the Program Structure

A.3 Assigning Symbolic Names

Defining Symbolic Names

Symbols are used in the example program and they must be defined in the symbol table using STEP 7. Table A-1 shows the symbolic names and the absolute addresses used to control the feed pumps, the agitator motor, and the inlet valves.

Table A-1 Symbolic Addresses of the Feed Pumps, the Agitator Motor and the Inlet Valves

Symbolic Name	Address	Data Type	Description
Feed_pump_A_start	I0.0	BOOL	Start button of the feed pump for ingredient A
Feed_pump_A_stop	I0.1	BOOL	Stop button of the feed pump for ingredient A
Flow_A	I0.2	BOOL	Ingredient A flowing
Inlet_valve_A	Q4.0	BOOL	Activates the inlet valve for ingredient A
Feed_valve_A	Q4.1	BOOL	Activates the feed valve for ingredient A
Feed_pump_A_on	Q4.2	BOOL	Lamp for “feed pump ingredient A running”
Feed_pump_A_off	Q4.3	BOOL	Lamp for “feed pump ingredient A not running”
Feed_pump_A	Q4.4	BOOL	Activates the feed pump for ingredient A
Feed_pump_A_fault	Q4.5	BOOL	Lamp for “feed pump A fault”
Feed_pump_A_maint	Q4.6	BOOL	Lamp for “feed pump A maintenance”
Feed_pump_B_start	I0.3	BOOL	Start button of the feed pump for ingredient B
Feed_pump_B_stop	I0.4	BOOL	Stop button of the feed pump for ingredient B
Flow_B	I0.5	BOOL	Ingredient B flowing
Inlet_valve_B	Q5.0	BOOL	Activates the inlet valve for ingredient B
Feed_valve_B	Q5.1	BOOL	Activates the feed valve for ingredient B
Feed_pump_B_on	Q5.2	BOOL	Lamp for “feed pump ingredient B running”
Feed_pump_B_off	Q5.3	BOOL	Lamp for “feed pump ingredient B not running”
Feed_pump_B	Q5.4	BOOL	Activates the feed pump for ingredient B
Feed_pump_B_fault	Q5.5	BOOL	Lamp for “feed pump B fault”
Feed_pump_B_maint	Q5.6	BOOL	Lamp for ”feed pump B maintenance”
Agitator_running	I1.0	BOOL	Response signal of the agitator motor
Agitator_start	I1.1	BOOL	Agitator start button
Agitator_stop	I1.2	BOOL	Agitator stop button
Agitator	Q8.0	BOOL	Activates the agitator
Agitator_on	Q8.1	BOOL	Lamp for “agitator running”
Agitator_off	Q8.2	BOOL	Lamp for “agitator not running”
Agitator_fault	Q8.3	BOOL	Lamp for “agitator motor fault”
Agitator_maint	Q8.4	BOOL	Lamp for “agitator motor maintenance”

Table A-2 shows the symbolic names and the absolute addresses used to control the sensors and display the level in the tank.

Table A-2 Symbolic Addresses of the Sensors and for Displaying the Level of the Tank

Symbolic Name	Address	Data Type	Description
Tank_below_max	I1.3	BOOL	Sensor “mixing tank not full”
Tank_above_min	I1.4	BOOL	Sensor “mixing tank above minimum level”
Tank_not_empty	I1.5	BOOL	Sensor “mixing tank not empty”
Tank_max_disp	Q9.0	BOOL	Lamp for “mixing tank full”
Tank_min_disp	Q9.1	BOOL	Lamp “mixing tank below minimum level”
Tank_empty_disp	Q9.2	BOOL	Lamp for “mixing tank empty”

Table A-3 shows the symbolic names and the absolute addresses used to control the drain valve.

Table A-3 Symbolic Addresses for the Drain

Symbolic Name	Address	Data Type	Description
Drain_open	I0.6	BOOL	Button for opening the drain valve
Drain_closed	I0.7	BOOL	Button for closing the drain valve
Drain	Q9.5	BOOL	Activates the drain valve
Drain_open_disp	Q9.6	BOOL	Lamp for “drain valve open”
Drain_closed_disp	Q9.7	BOOL	Lamp for “drain valve closed”

Table A-4 shows the symbolic names and the absolute addresses used to control the other elements of the program.

Table A-4 Symbolic Addresses of the Other Program Elements

Symbolic Name	Address	Data Type	Description
EMER_STOP_off	I1.6	BOOL	EMERGENCY STOP switch
Reset_maint	I1.7	BOOL	Reset button for the maintenance display lamps of the three motors
Motor_block	FB1	FB1	FB for controlling pumps and motor
Valve_block	FC1	FC1	FC for controlling the valves
DB_feed_pump_A	DB1	FB1	Instance DB for controlling feed pump A
DB_feed_pump_B	DB2	FB1	Instance DB for controlling feed pump B
DB_agitator	DB3	FB1	Instance DB for controlling the agitator motor

A.4 Creating the FB for the Motor

What is Required of the FB

The FB for the motor contains the following logical functions:

- There is a start and a stop input.
- Interlocks allow the operation of the devices (pumps and agitator motor). The status of the interlocks is saved in the temporary local data (L stack) of OB1 (“Motor_enable”, “Valve_enable”) and is logically combined with the start and stop inputs when the FB for the motor is executed.
- Responses from the devices must appear within a certain time. Otherwise, it is assumed that an error or fault has occurred. The function then stops the motor.
- The point in time and the duration of the response or error/fault cycle must be specified.
- If the start button is pressed and the motor enabled, the device switches itself on and runs until the stop button is pressed.
- If the device is switched on, a timer starts to run. If the response signal from the device is not received before the timer has expired, the device stops.

Specifying the Inputs and Outputs

Figure A-3 illustrates the inputs and outputs of the general FB for the motor.

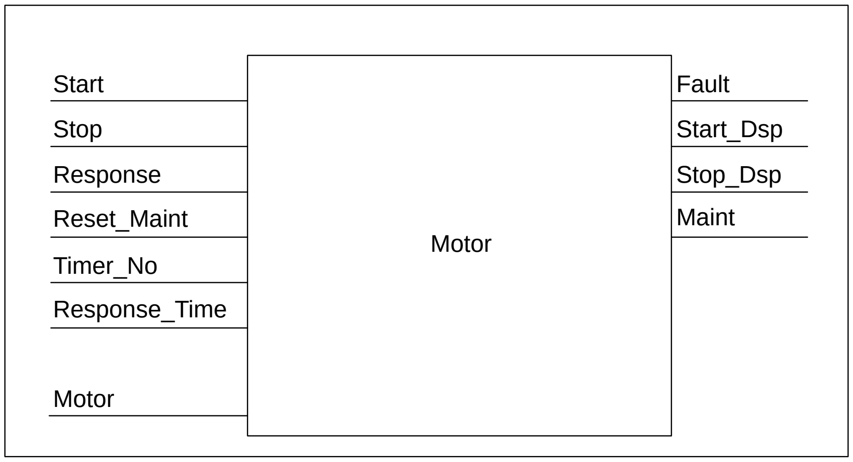


Figure A-3 Inputs and Outputs of the FB for the Motor

Defining the Parameters for the FB

If you use a multiple instance FB for the motor (for controlling both pumps and the motor) you must define general parameter names for the inputs and outputs.

The FB for the motor in the sample process requires the following:

- It must have signals from the operator station to stop and start the motor and pumps.
- It requires a response signal from the motor and pumps to indicate that the motor is running.
- It must calculate the time between sending the signal to activate the motor and receiving the response signal. If no response signal is received in this time, the motor must be switched off.
- It must turn the lamps on the operator station on and off.
- It supplies a signal to activate the motor.

These requirements can be specified as inputs and outputs to the FB. Table A-5 shows the parameters of the FB for the motor in out sample process.

Table A-5 Input and Output Parameters

Parameters for Pumps and Motor	Input	Output	In/Out
Start	✓		
Stop	✓		
Response	✓		
Reset_maint	✓		
Timer_no	✓		
Response_time	✓		
Fault		✓	
Start_dsp		✓	
Stop_dsp		✓	
Maint		✓	
Motor			✓

Declaring the Variables of the FB for the Motor

You must declare the input, in/out and output parameters of the FB for the motor.

With FBs, the input, output, in/out, and static variables are saved in the instance DB specified in the call statement. The temporary variables are stored in the L stack.

Table A-6 Variable Declaration Table of the FB for the Motor

Address	Declaration	Name	Type	Initial value
0.0	IN	Start	BOOL	FALSE
0.1	IN	Stop	BOOL	FALSE
0.2	IN	Response	BOOL	FALSE
0.3	IN	Reset_maint	BOOL	FALSE
2	IN	Timer_no	TIMER	
4	IN	Response_time	S5TIME	S5T#0MS
6.0	OUT	Fault	BOOL	FALSE
6.1	OUT	Start_dsp	BOOL	FALSE
6.2	OUT	Stop_dsp	BOOL	FALSE
6.3	OUT	Maint	BOOL	FALSE
8.0	IN_OUT	Motor	BOOL	FALSE
10.0	STAT	Time_bin	WORD	W#16#0
12.0	STAT	Time_BCD	WORD	W#16#0
14.0	STAT	Starts	INT	0
16.0	STAT	Start_edge	BOOL	FALSE

Programming the FB for the Motor

In STEP 7, every block that is called by a different block must be created before the block that contains its call. In the sample program, you must therefore create the FB for the motor before OB1.

The statement section of FB1 appears as follows in the STL programming language:

Network 1 Start/Stop and latching

```
A(
O     #Start
O     #Motor
)
AN    #Stop
=     #Motor
```

Network 2 Startup monitoring

```
      A     #Motor
      L     #Response_time
      SD    #Timer_no
      AN    #Motor
      R     #Timer_no
      L     #Timer_no
      T     #Time_bin
      LC    #Timer_no
      T     #Time_BCD
      A     #Timer_no
      AN    #Response
      S     #Fault
      R     #Motor
```

Network 3 Start lamp / Fault response

```
      A     #Response
      =     #Start_dsp
      R     #Fault
```

Network 4 Stop lamp

```
      AN    #Response
      =     #Stop_dsp
```

Network 5 Counting the starts

```
      A     #Motor
      FP    #Start_edge
      JCN   lab1
      L     #Starts
      +     1
      T     #Starts
lab1: NOP
```

Network 6 Maintenance

```
      L     #Starts
      L     50
      >=I
      =     #Maint
```

Network 7 Maintenance reset

```
      AN    #Reset_maint
      A     #Maint
      JCN   END
      L     0
      T     #Starts
END:  NOP 0
```

A.5 Creating the FC for the Valves

What is Required of the FC?

The function for the inlet and feed valve and for the drain valve contains the following logical functions:

- There is an input for opening and an input for closing the valves.
- Interlocks allow the valves to be opened. The state of the interlocks is saved in the temporary local data (L stack) of FB1 (“Valve_enable”) and is logically combined with the inputs for opening and closing when the FC for the valves is executed.

Table A-7 shows the parameters that must be transferred to the FC.

Table A-7 Input, In/Out, and Output Parameters

Parameters for the Valves	Input	Output	In/Out
Open	✓		
Close	✓		
Dsp_open		✓	
Dsp_closed		✓	
Valve			✓

Specifying the Inputs and Outputs

Figure A-4 shows the inputs and outputs of the general FC for the valves. The devices that call the FB for the motor transfer input parameters. The FC for the valves returns output parameters.

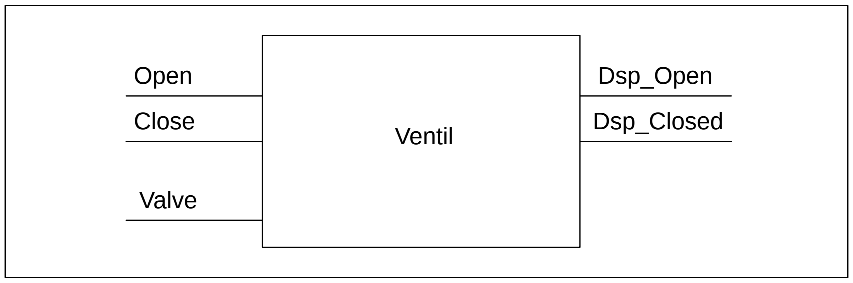


Figure A-4 Inputs and Outputs of the FC for the Valves

Declaring the Variables of the FC for the Valves

Just as with the FB for the motor, you must also declare the input, in/out and output parameters for the FC for the valves.

With FCs, the temporary variables are saved in the L stack. The input, output and in/out variables are saved as pointers to the logic block that called the FC. Additional memory space in the L stack (after the temporary variables) is used for these variables.

Table A-8 Variable Declaration Table of the FC for the Valves

Address	Declaration	Name	Type	Default
0.0	IN	Open	BOOL	FALSE
0.1	IN	Close	BOOL	FALSE
2.0	OUT	Dsp_open	BOOL	FALSE
2.1	OUT	Dsp_closed	BOOL	FALSE
4.0	IN_OUT	Valve	BOOL	FALSE

Programming the FC for the Valves

The FC1 function for the valves must be created before OB1 since the called blocks must be created before the calling blocks.

The statement section of FC1 appears as shown below in the STL programming language:

Network 1 Open/close and latching

```
A(
O     #Open
O     #Valve
)
AN    #Close
=     #Valve
```

Network 2 Display “Valve open”

```
A     #Valve
=     #Dsp_open
```

Network 3 Display “Valve closed”

```
AN    #Valve
=     #Dsp_closed
```

A.6 Creating OB1

Overview

OB1 decides the structure of the sample program. OB1 also contains the parameters that are transferred to the various functions, for example:

- The STL networks for the feed pumps and the motor supply the FB for the motor with the input parameters for starting (“Start”), stopping (“Stop”), the response (“Response”), and for resetting the maintenance display (”Reset_maint”). The FB for the motor is executed in every cycle of the PLC.
- If the FB for the motor is executed, the inputs Timer_no and Response_time inform the function of the timer being used and which time must be measured.
- The outputs of the FB for the motor are saved at the addresses Error and Motor in the network that called the FB.
- The FC for the valves is executed automatically in every cycle of the PLC.

The program uses the FB for the motor with different instance DBs to handle the tasks for controlling the feed pumps and the agitator motor.

Declaring Variables for OB1

Table A-9 contains the variable declaration table for OB1. The first 20 bytes contain the start information of OB1 and must not be modified.

Table A-9 Variable Declaration Table for OB1

Address	Declaration	Name	Type
0.0	TEMP	OB1_EV_CLASS	BYTE
1.0	TEMP	OB1_SCAN1	BYTE
2.0	TEMP	OB1_PRIORITY	BYTE
3.0	TEMP	OB1_OB_NUMBR	BYTE
4.0	TEMP	OB1_RESERVED_1	BYTE
5.0	TEMP	OB1_RESERVED_2	BYTE
6.0	TEMP	OB1_PREV_CYCLE	INT
8.0	TEMP	OB1_MIN_CYCLE	INT
10.0	TEMP	OB1_MAX_CYCLE	INT
12.0	TEMP	OB1_DATE_TIME	DATE_AND_TIME
20.0	TEMP	Enable_motor	BOOL
20.1	TEMP	Enable_valve	BOOL
20.2	TEMP	Start_fulfilled	BOOL
20.3	TEMP	Stop_fulfilled	BOOL
20.4	TEMP	Inlet_valve_A_open	BOOL
20.5	TEMP	Inlet_valve_A_closed	BOOL
20.6	TEMP	Feed_valve_A_open	BOOL
20.7	TEMP	Feed_valve_A_closed	BOOL
21.0	TEMP	Inlet_valve_B_open	BOOL
21.1	TEMP	Inlet_valve_B_closed	BOOL
21.2	TEMP	Feed_valve_B_open	BOOL
21.3	TEMP	Feed_valve_B_closed	BOOL
22.4	TEMP	Open_drain	BOOL
22.5	TEMP	Close_drain	BOOL
22.6	TEMP	Valve_closed_fulfilled	BOOL

Creating the Program for OB1

In STEP 7, every block that is called by a different block must be created before the block containing its call. In the sample program, you must therefore create both the FB for the motor and the FC for the valves before the program in OB1.

The statement section of OB1 appears as shown below in the STL programming language:

Network 1 Interlocks for feed pump A

```
A     “EMER_STOP_off”
A     “Tank_below_max”
AN    “Drain”
=     #Enable_motor
```

Network 2 Calling FB motor for ingredient A

```
A     “Feed_pump_A_start”
A     #Enable_motor
=     #Start_fulfilled
A(
O     “Feed_pump_A_stop”
ON    #Enable_motor
)
=     #Stop_fulfilled
CALL “Motor_block”, “DB_feed_pump_A”
   Start         :=#Start_fulfilled
   Stop          :=#Stop_fulfilled
   Response      :=“Flow_A”
   Reset_maint   :=“Reset_maint”
   Timer_no      :=T12
   Response_time
:=S5T#7S
   Fault         :=“Feed_pump_A_fault”
   Start_dsp     :=“Feed_pump_A_on”
   Stop_dsp      :=“Feed_pump_A_off”
   Maint         :=“Feed_pump_A_maint”
   Motor         :=“Feed_pump_A”
```

Network 3 Delaying the valve enable ingredient A

```
A     “Feed_pump_A”
L     S5T#1S
SD    T   13
AN    “Feed_pump_A”
R     T   13
A     T   13
=     #Enable_valve
```

Network 4 Inlet valve control for ingredient A

```
AN     "Flow_A"
AN     "Feed_pump_A"
=      #Close_valve_fulfilled
CALL "Valve_block"
     Open         :=#Enable_valve
     Close        :=#Close_valve_fulfilled
     Dsp_open     :=#Inlet_valve_A_open
     Dsp_closed   :=#Inlet_valve_A_closed
     Valve        :="Inlet_valve_A"
```

Network 5 Feed valve control for ingredient A

```
AN     "Flow_A"
AN     "Feed_pump_A"
=      #Close_valve_fulfilled
CALL "Valve_block"
     Open         :=#Enable_valve
     Close        :=#Close_valve_fulfilled
     Dsp_open     :=#Feed_valve_A_open
     Dsp_closed   :=#Feed_valve_A_closed
     Valve        :="Feed_valve_A"
```

Network 6 Interlocks for for feed pump B

```
A      "EMER_STOP_off"
A      "Tank_below_max"
AN     "Drain"
=      #Enable_motor
```

Network 7 Calling FB Motor for ingredient B

```
A      "Feed_pump_B_start"
A      #Enable_motor
=      #Start_fulfilled
A(
O      "Feed_pump_B_stop"
ON     #Enable_motor
)
=      #Stop_fulfilled
CALL "Motor_block", "DB_feed_pump_B"
     Start          :=#Start_fulfilled
     Stop           :=#Stop_fulfilled
     Response       :="Flow_B"
     Reset_maint    :="Reset_maint"
     Timer_no       :=T14
     Response_time  :=S5T#7S
     Fault          :="Feed_pump_B_fault"
     Start_dsp      :="Feed_pump_B_on"
     Stop_dsp       :="Feed_pump_B_off"
     Maint          :="Feed_pump_B_maint"
     Motor          :="Feed_pump_B"
```

Network 8 Delaying valve enable for ingredient B

```
A      “Feed_pump_B”
L      S5T#1S
SD     T    15
AN     “Feed_pump_B”
R      T    15
A      T    15
=      #Enable_valve
```

Network 9 Inlet valve control for ingredient B

```
AN     “Flow_B”
AN     “Feed_pump_B”
=      #Close_valve_fulfilled
CALL “Valve_block”
   Open        :=#Enable_valve
   Close       :=#Close_valve_fulfilled
   Dsp_open    :=#Inlet_valve_B_open
   Dsp_closed  :=#Inlet_valve_B_closed
   Valve       :=“Inlet_valve_B”
```

Network 10 Feed valve control for ingredient B

```
AN     “Flow_B”
AN     “Feed_pump_B“
=      #Close_valve_fulfilled
CALL “Valve_block”
   Open        :=#Enable_valve
   Close       :=#Close_valve_fulfilled
   Dsp_open    :=#Feed_valve_B_open
   Dsp_closed  :=#Feed_valve_B_closed
   Valve       :=“Feed_valve_B”
```

Network 11 Interlocks for agitator

```
A      “EMER_STOP_off”
A      “Tank_above_min”
AN     “Agitator_fault”
AN     ”Drain”
=      #Enable_motor
```

Network 12 Calling FB motor for agitator

```
A      "Agitator_start"
A      #Enable_motor
=      #Start_fulfilled
A(
O      "Agitator_stop"
ON     #Enable_motor
)
=      #Stop_fulfilled
CALL "Motor_block", "DB_agitator"
   Start          :=#Start_fulfilled
   Stop           :=#Stop_fulfilled
   Response       :="Agitator_running"
   Reset_maint    :="Reset_maint"
   Timer_no       :=T16
   Response_time  :=S5T#10S
   Fault          :="Agitator_fault"
   Start_dsp      :="Agitator_on"
   Stop_dsp       :="Agitator_off"
   Maint          :="Agitator_maint"
   Motor          :="Agitator_B"
```

Network 13 Interlocks for drain valve

```
A      "EMER_STOP_off"
A      "Tank_not_empty"
AN     "Agitator"
=      #Enable_valve
```

Network 14 Drain valve control

```
A      "Drain_open"
A      #Enable_valve
AN     "Agitator"
=      #Open_drain
A(
O      "Drain_closed"
ON     #Enable_valve
)
=      #Close drain
CALL "Valve_block"
   Open        :=#Open_drain
   Close       :=#Close_drain
   Dsp_open    :="Drain_open_disp"
   Dsp_closed  :="Drain_closed_disp"
   Valve       :="Drain"
```

Network 15 Tank level display

```
AN     "Tank_below_max"
=      "Tank_max_disp"
AN     "Tank_above_min"
=      "Tank_min_disp"
AN     "Tank_not_empty"
=      "Tank_empty_disp"
```

Sample Program for Communication SFBs for Configured Connections B

What Does This Chapter Describe?

This chapter explains the data exchange between two S7-400-CPUs and the use of communication SFBs for configured connections in the user program based on a simple example.

Chapter Overview

B.1 Overview

Introduction

The sample program shows how data are exchanged between two S7 400-CPUs using communication SFBs for configured connections.

Communication SFBs Used

The following communication SFBs are used in the sample program.

Table B-1 Communication SFBs in the Sample Program

SFB		Function
SFB 8/ SFB 9	USEND/ URCV	Uncoordinated data exchange using a send and a receive SFB (bilateral communication)
SFB12/ SFB13	BSEND/ BRCV	Field-oriented data exchange using a send and a receive SFB (bilateral communication)
SFB14	GET	Reads data from the remote device (unilateral communication)
SFB15	PUT	Writes data to the remote device (unilateral communication)
SFB19	START	Triggers a complete restart on the remote device
SFB20	STOP	Sets the remote device to STOP
SFB21	RESUME	Triggers a restart on the remote device
SFB22	STATUS	Queries the status of the remote device
SFB23	USTATUS	Receives the status of the remote device sent unsolicited by the remote device

Connection Type

This example uses a bilateral configured homogeneous S7 connection. Both the single and paired blocks are used on this connection.

The connection ID on both CPUs is W#16#0001.

Hardware Requirements

The description of this example is based on the following hardware configuration.

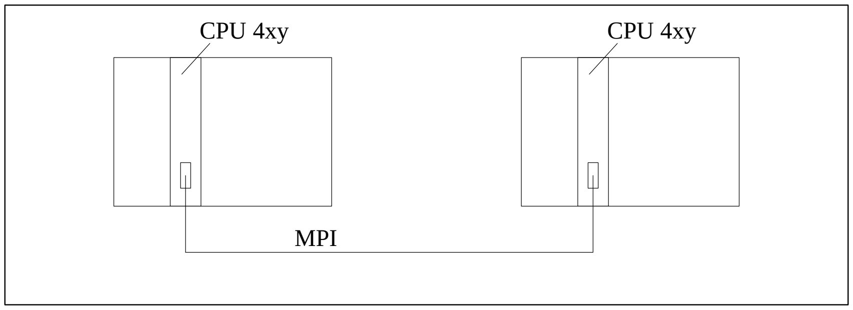


Figure B-1 Hardware Configuration for the Sample Program

B.2 Sample Program on the Sending CPU

Introduction

In the sample program on the sending CPU the data transfer is triggered by memory bits. You can change the memory bits used in the variable table “VAT 1”. A rising edge at a memory bit starts the corresponding communication SFB.

Memory Bits Used

The following table indicates the assignment of the memory area.

Table B-2 Assignment of the Memory Area

Memory Bit	Assignment
M20.0	triggers USEND
M20.2	triggers BSEND
M20.4	triggers GET
M20.5	triggers PUT
M20.6	triggers START
M20.7	triggers STOP
M21.0	triggers RESUME
M21.1	triggers STATUS
M21.2	triggers USTATUS

Blocks on the Sending CPU

Table B-3 User-Defined Blocks on the Sending CPU

Block	Content	Function
OB100	Call for FC EXAMPLE_PRESET_SFBs_1	Startup OB: When the SFBs are called later in the user program, only control and diagnostic parameters need to be specified.
FC EXAMPLE_ PRESET_SFBs_1	Initialization call for SFBs USEND, BSEND, GET, PUT, START, STOP, RESUME, STATUS, USTATUS	
OB35	FC calls to control the SFBs	Cyclic interrupt OB: for cyclic FC calls
FC CHECK	Evaluation of DONE, NDR, ERROR, STATUS	Checks the status of the SFB
FC EXAMPLE_USEND	Call for SFB USEND and FC CHECK	Controlling SFB calls with FCs prevents the SFBs being called again before they are completed.
FC EXAMPLE_BSEND	Call for SFB BSEND and FC CHECK	
FC EXAMPLE_GET	Call for SFB GET and FC CHECK	
FC EXAMPLE_PUT	Call for SFB PUT and FC CHECK	
FC EXAMPLE_START	Call for SFB START and FC CHECK	
FC EXAMPLE_STOP	Call for SFB STOP and FC CHECK	
FC EXAMPLE_RESUME	Call for SFB RESUME and FC CHECK	
FC EXAMPLE_STATUS	Call for SFB STATUS and FC CHECK	
FC EXAMPLE_USTATUS	Call for SFB USTATUS and FC CHECK	
DB IDB_USEND, DB IDB_BSEND, DB IDB_GET, DB IDB_PUT, DB IDB_START, DB IDB_STOP, DB IDB_RESUME DB IDB_STATUS, DB IDB_USTATUS	Actual parameters and static data of the SFBs used	Instance DBs of the SFBs used
DB data_usend	Send, control and check data for FC EXAMPLE_USEND	Shared DBs
DB data_bsend	Send, control and check data for FC EXAMPLE_BSEND	
DB data_get	Send, control and check data for FC EXAMPLE_GET	
DB data_put	Source area, control and check data for FC EXAMPLE_PUT	
DB data_program_cntr	Control and check data for SFBs START, STOP, RESUME, STATUS and USTATUS	
DB data_get_source	DB on the remote CPU,from which the data are read by SFB GET	
DB data_put_destination	DB on the remote CPU in which the data are written by SFB PUT	

Defining Symbolic Names

In the sample program on the sending CPU, symbols are used that were defined with STEP 7 in the symbols table. Table B-4 shows the symbolic names and the absolute addresses of the sample program.

Table B-4 Symbolic Names and the Corresponding Addresses of the Sample Program on the Sending CPU

Symbolic Name	Address	Data Type	Comment
IDB_USEND	DB8	SFB8	Instance DBs
IDB_BSEND	DB12	SFB12	
IDB_GET	DB14	SFB14	
IDB_PUT	DB15	SFB15	
IDB_START	DB19	SFB19	
IDB_STOP	DB20	SFB20	
IDB_RESUME	DB21	SFB21	
IDB_STATUS	DB22	SFB22	
IDB_USTAUTS	DB23	SFB23	
data_usend	DB100	DB100	Shared DBs
data_bsend	DB102	DB102	
data_get	DB104	DB104	
data_put	DB105	DB105	
data_program_cntr	DB106	DB106	
data_get_source	DB107	DB107	
data_put_destination	DB108	DB108	
CHECK	FC99	FC99	FCs
EXAMPLE_USEND	FC100	FC100	
EXAMPLE_PUT	FC101	FC101	
EXAMPLE_STOP	FC102	FC102	
EXAMPLE_STATUS	FC103	FC103	
EXAMPLE_BSEND	FC105	FC105	
EXAMPLE_GET	FC107	FC107	
EXAMPLE_START	FC108	FC108	
EXAMPLE_RESUME	FC109	FC109	
EXAMPLE_USTATUS	FC110	FC110	
EXAMPLE_ PRESET_SFBs 1	FC111	FC111	
CYCL_EXC	OB1	OB1	OBs
CYC_INT5	OB35	OB35	
COMPLETE RESTART	OB100	OB100	
BSEND	SFB12	SFB12	SFBs
GET	SFB14	SFB14	
PUT	SFB15	SFB15	
START	SFB19	SFB19	
STOP	SFB20	SFB20	
RESUME	SFB21	SFB21	
STATUS	SFB22	SFB22	
USTATUS	SFB23	SFB23	

B.3 Sample Program on the Receiving CPU

Introduction

In the sample program on the receiving CPU, the data transfer is triggered by memory bits. You can change the memory bits used in the variable table “VAT 2”. Signal state 1 in a memory bit enables the corresponding receive SFB.

Memory Bits Used

The following table indicates the assignment of the memory area.

Table B-5 Assignment of the Memory Area

Memory Bit	Function
M20.1	Enables URCV
M20.3	Enables BRCV

Blocks on the Receiving CPU

Table B-6 User-Defined Blocks on the Receiving CPU

Block	Content	Function
OB100	Call for the FC EXAMPLE_PRESET_SFBs 2	Startup OB: When the SFB is called later in the user program, only control and diagnostic parameters need to be specified.
FC EXAMPLE_PRESET_SFBs 2	Initialization calls for the SFBs URCV, BRCV	
OB35	Call for the FCs to control the SFBs	Cyclic interrupt OB: cyclic FC calls
FC CHECK	Evaluation of DONE, NDR, ERROR, STATUS	Checks the status of the SFB
FC EXAMPLE_URCV	Call for SFB URCV and FC CHECK	Controlling SFB calls with FCs prevents the SFBs being called again before they are completed.
FC EXAMPLE_BRCV	Call for SFB BRCV and FC CHECK	
DB IDB_URCV, DB IDB_BRCV	Actual parameters and static data of the SFBs used	Instance DBs of the SFBs used
DB data_urcv	Receive, control and check data for FC EXAMPLE_URCV	Shared DBs
DB data_brcv	Receive, control and check data for FC EXAMPLE_BRCV	
DB data_get_source	DB with the data for the SFB GET of the communication partner	
DB data_put_destination	DB in which the data will be written by SFB PUT	

Defining Symbolic Names

In the sample program on the receiving CPU, symbols are used that were defined with STEP 7 in the symbols table. Table B-7 shows the symbolic names and the absolute addresses of the sample program.

Table B-7 Symbolic Names and Corresponding Addresses of the Sample Program on the Receiving CPU

Symbolic Name	Address	Data Type	Comment
IDB_URCV	DB9	SFB9	Instance DBs
IDB_BRCV	DB13	SFB13	
data_urcv	DB101	DB101	Shared DBs
data_brcv	DB103	DB103	
data_get_source	DB107	DB107	
data_put_destination	DB108	DB108	
CHECK	FC99	FC99	FCs
EXAMPLE_URCV	FC104	FC104	
EXAMPLE_BRCV	FC106	FC106	
EXAMPLE_ PRESET_SFBs 2	FC112	FC112	
CYC_INT5	OB35	OB35	OBs
COMPLETE RESTART	OB100	OB100	
URCV	SFB9	SFB9	SFBs
BRCV	SFB13	SFB13	

B.4 Using the Sample Program

Procedure

To use the sample program, follow the steps below:

1. Reset the memory on both CPUs and then download the programs to the appropriate CPUs.
2. Run a complete restart on both CPUs.
   This sets the connection references and the send and receive areas. The send areas have the number of the corresponding SFB entered, the receive areas have the value 0 entered.
3. Call up the variable tables “VAT 1” (on the sending CPU) and “VAT 2” (on the receiving CPU).
4. On the receiving CPU, enable the receive SFBs by setting memory bits M20.1 and M20.3 to 1 in variable table “VAT 2”.
5. Start the data transfer on the sending CPU by setting the corresponding memory bit to 1 in variable table “VAT 1” (see Table B-2).
6. If required, change the content of the send areas.
7. If an error occurs in the data transfer, evaluate the output parameters ERROR and STATUS of the relevant communication SFB.

B.5 Call Hierarchy of the Blocks in the Sample Program

Call Hierarchy on the Sending CPU

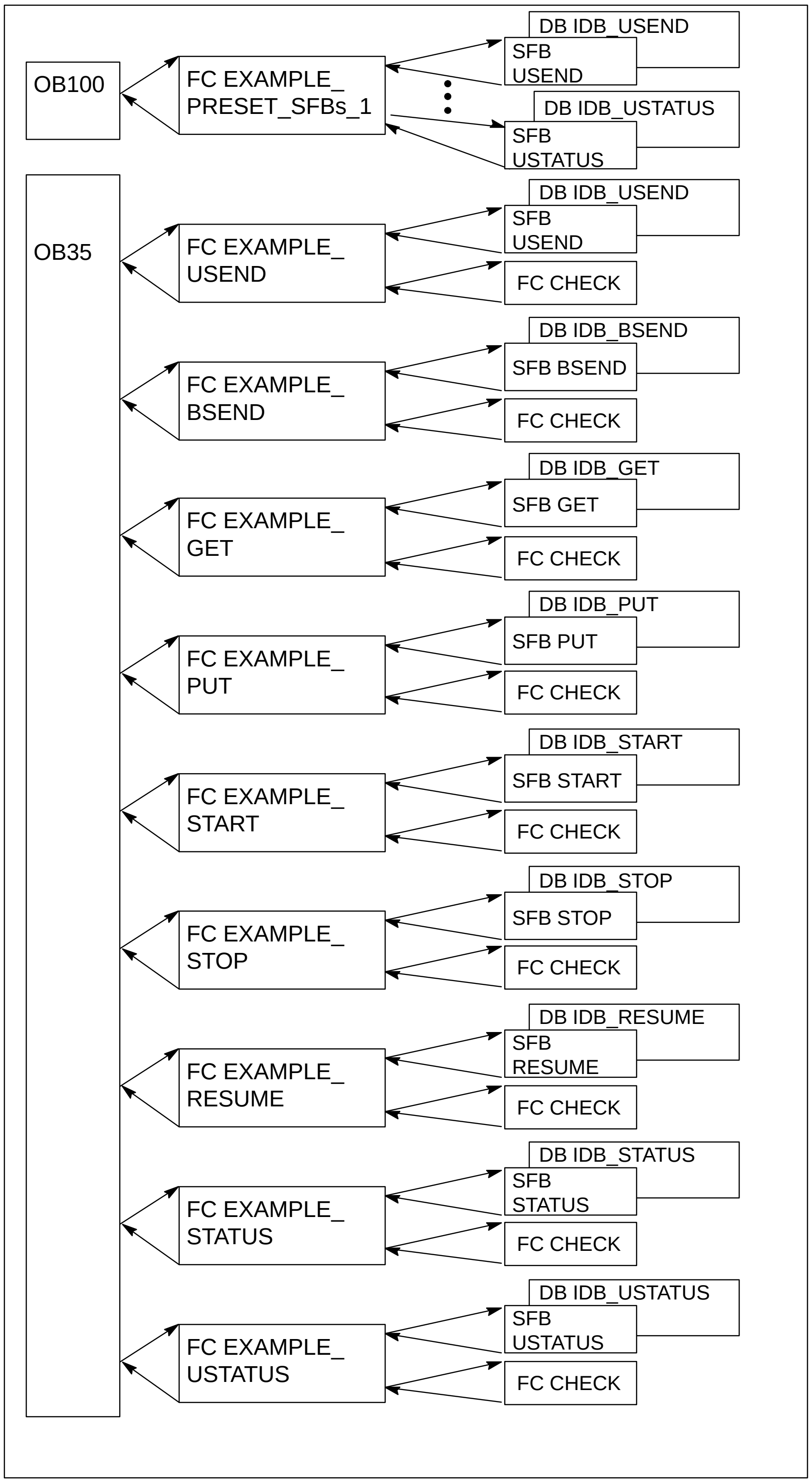


Figure B-2 Call Hierarchy on the Sending CPU

Call Hierarchy on the Receiving CPU

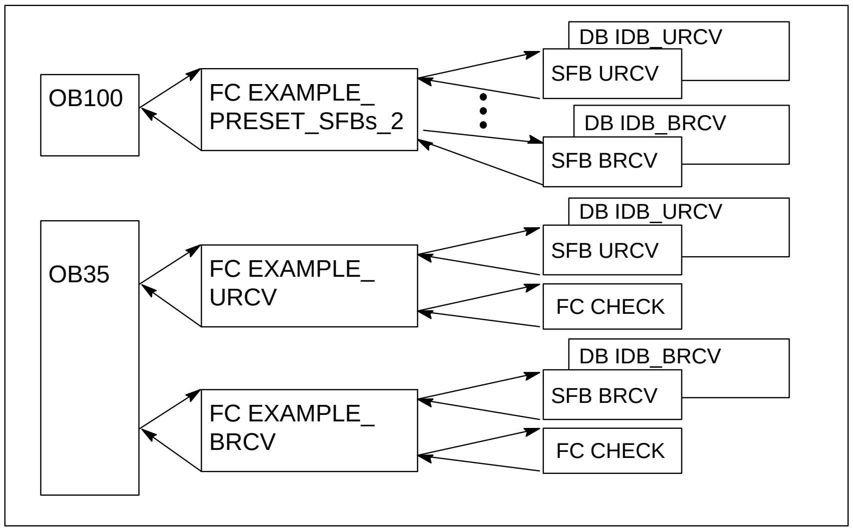


Figure B-3   Call Hierarchy on the Receiving CPU

STL Program

The code for the sample programs is in the directory step 7\examples\com_sfb.

Data and Parameter Types

C

What Does This Chapter Describe?

This chapter describes the following:

- Which data types are available for static or temporary variables and parameters.
- Which data types you can assign to the local data of the individual block types.
- Restrictions you should note when transferring parameters.

Chapter Overview

C.1 Data Types

Introduction

All the data in a user program must be identified by a data type. The following data types are available:

- Elementary data types provided by STEP 7
- Complex data types that you yourself can create by combining elementary data types
- User-defined data types
- Parameter types with which you define parameters to be transferred to FBs or FCs

Elementary Data Types

Each elementary data type has a defined length. The data type BOOL has, for example, only one bit, a byte (BYTE) consists of 8 bits, a word (WORD) consists of 2 bytes (16 bits), a double word (DWORD) has 4 bytes (32 bits). Table C-1 lists the elementary data types.

Table C-1 Description of the Elementary Data Types

Type and Description	Size in Bits	Format Options	Range and Numeric Representation (lowest to highest value)	Example
BOOL (Bit)	1	Boolean text	TRUE/FALSE	TRUE
BYTE (Byte)	8	Hexadecimal number	B16#0 to B16#FF	L B#16#10 L byte#16#10
WORD	16	Binary number Hexadecimal number BCD Decimal number unsigned	2#0 to 2#1111_1111_1111_1111 W#16#0 to W#16#FFFF C#0 to C#999 B#(0,0) to B#(255,255)	L 2#0001_0000_0000_0000 L W#16#1000 L word16#1000 L C#998 L B#(10,20) L byte#(10,20)
DWORD (Double word)	32	Binary number Hexadecimal number Decimal number unsigned	2#0 to 2#1111_1111_1111_1111_ 1111_1111_1111_1111 DW#16#0000_0000 to DW#16#FFFF_FFFF B#(0,0,0,0) to B#(255,255,255,255)	2#1000_0001_0001_1000_ 1011_1011_0111_1111 L DW#16#00A2_1234 L dword#16#00A2_1234 L B#(1, 14, 100, 120) L byte#(1,14,100,120)
INT (Integer)	16	Decimal number signed	-32768 to 32767	L 1
DINT (Integer, 32 bits)	32	Decimal number signed	L#–2147483648 to L#2147483647	L L#1

Table C-1 Description of the Elementary Data Types, continued

Type and Description	**Size in Bits**	**Format Options**	**Range and Numeric Representation (lowest to highest value)**	**Example**
REAL (floating point)	32	IEEE floating point number	Upper limit: ±3.402823e+38 Lower limit: ±1.175 495e-38	L 1.234567e+13
S5TIME (SIMATIC time)	16	S7 time in steps of 10 ms (default)	S5T#0H_0M_0S_10MS to S5T#2H_46M_30S_0MS and S5T#0H_0M_0S_0MS	L S5T#0H_1M_0S_0MS L S5TIME#0H_1H_1M_0S_0MS
TIME (IEC time)	32	IEC time in steps of 1 ms, integer signed	-T#24D_20H_31M_23S_648MS to T#24D_20H_31M_23S_647MS	L T#0D_1H_1M_0S_0MS L TIME#0D_1H_1M_0S_0MS
DATE (IEC date)	16	IEC date in steps of 1 day	D#1990-1-1 to D#2168-12-31	L D#1994-3-15 L DATE#1994-3-15
TIME_OF_DAY	32	Time in steps of 1 ms	TOD#0:0:0.0 to TOD #23:59:59.999	L TOD#1:10:3.3 L TIME_OF_DAY#1:10:3.3
CHAR (Character)	8	ASCII characters	’A’,’B’ etc.	L ’E’

Complex Data Types

Complex data types define data groups that are larger than 32 bits or data groups consisting of other data types. STEP 7 permits the following complex data types:

- DATE_AND_TIME
- STRING
- ARRAY
- STRUCT
- FBs and SFBs

Table C-2 describes the complex data types. They define structures and arrays either in the variable declaration of the logic block or in a data block.

Table C-2 Description of the Complex Data Types

Data Type	Description
DATE_AND_TIME DT	Defines an area with 64 bits (8 bytes). This data type saves the following information (in binary coded decimal format): year in byte 0, month in byte 1, day in byte 2, hours in byte 3, minutes in byte 4, seconds in byte 5, milliseconds in byte 6 and half of byte 7, weekday in the other half of byte 7.
STRING	Defines a group with a maximum of 254 characters (data type CHAR). The standard area reserved for a character string is 256 bytes long. This is the space required to save 254 characters and a header of 2 bytes. You can reduce the memory required for a string by defining the number of characters that will be stored in the character string (for example: string[9] ‘Siemens’).
ARRAY	Defines a multi-dimensional grouping of one data type (either elementary or complex). For example: “ARRAY [1..2,1..3] OF INT” defines and array in the format 2 x 3 consisting of integers. You access the data stored in an array using the Index (“[2,2]”). You can define up to a maximum of 6 dimensions in one array. The index can be any integer (-32768 to 32767).
STRUCT	Defines a grouping of any combination of data types. You can, for example, define an array of structures or a structure of structures and arrays.
FB, SFB	You determine the structure of the assigned instance data block and allow the transfer of instance data for several FB calls in one instance DB (multiple instances, see Section 2.10).

User-Defined Data Types

In STEP 7, you can combine complex and elementary data types to create your own “user-defined” data type (UDT). UDTs have their own name and can therefore be used more than once. In a UDT, you can structure large amounts of data and simplify the input of data types when you want to create data blocks or declare variables in the variable declaration.

Parameter Types

In addition to elementary, complex, and user-defined data types, you can also define parameter types for formal parameters that are transferred between blocks (see Table C-3). STEP 7 recognizes the following parameter types:

- TIMER or COUNTER: this specifies a particular timer or particular counter that will be used when the block is executed. If you supply a value to a formal parameter of the TIMER or COUNTER parameter type, the corresponding actual parameter must be a timer or a counter, in other words, you enter “T” or “C” followed by a positive integer.
- BLOCK: specifies a particular block to be used as an input or output. The declaration of the parameter determines the block type to be used (FB, FC, DB etc.). If you supply values to a formal parameter of the BLOCK parameter type, specify a block address as the actual parameter. Example: “FC101” (when using absolute addressing) or “Valve” (with symbolic addressing).
- POINTER: references the address of a variable. A pointer contains an address instead of a value. When you supply a value to a formal parameter of the parameter type POINTER, you specify an address as the actual parameter. In STEP 7, you can specify a pointer in the pointer format or simply as an address (for example M 50.0). Example of a pointer format for addressing the data beginning at M 50.0: `P#M50.0`
- ANY: this is used when the data type of the actual parameter is unknown or when any data type can be used. For more information about the ANY parameter type refer to Section C.6.

A parameter type can also be used in a user-defined data type (UDT). For more information about UDTs, refer to Section C.5.

Table C-3 Parameter Types

Parameter	Size	Description
TIMER	2 Bytes	Indicates a timer to be used by the program in the called logic block. Format: `T1`
COUNTER	2 Bytes	Indicates a particular counter to be used by the program in the called logic block. Format: `C10`
BLOCK_FB BLOCK_FC BLOCK_DB BLOCK_SDB	2 Bytes	Indicates a particular block to be used by the program in the called logic block. Format: `FC101` `DB42`
POINTER	6 Bytes	Identifies the address. Format: `P#M50.0`
ANY	10 Bytes	Is used when the data type of the current parameter is unknown (see Section C.6). Format: `P#M50.0 BYTE 10` `P#M100.0 WORD 5`

C.2 Using Complex Data Types

Overview

You can create new data types by combining the elementary and complex data types to create the following complex data types:

- Array (data type ARRAY): an array combines a group of one data type to form a single unit.
- Structure (data type STRUCT): a structure combines different data types to form a single unit.
- Character string (data type STRING): a character string defines a one-dimensional array with a maximum of 254 characters (data type CHAR). A character string can only be transferred as a unit. The length of the character string must match the formal and actual parameter of the block.
- Date and time (data type DATE_AND_TIME): the date and time data type stores the year, month, day, hours, minutes, seconds, milliseconds and weekday.

Figure C-1 shows how arrays and structures can structure data types in one area and save information. You define an array or a structure either in a DB or in the variable declaration of an FB, OB or FC.

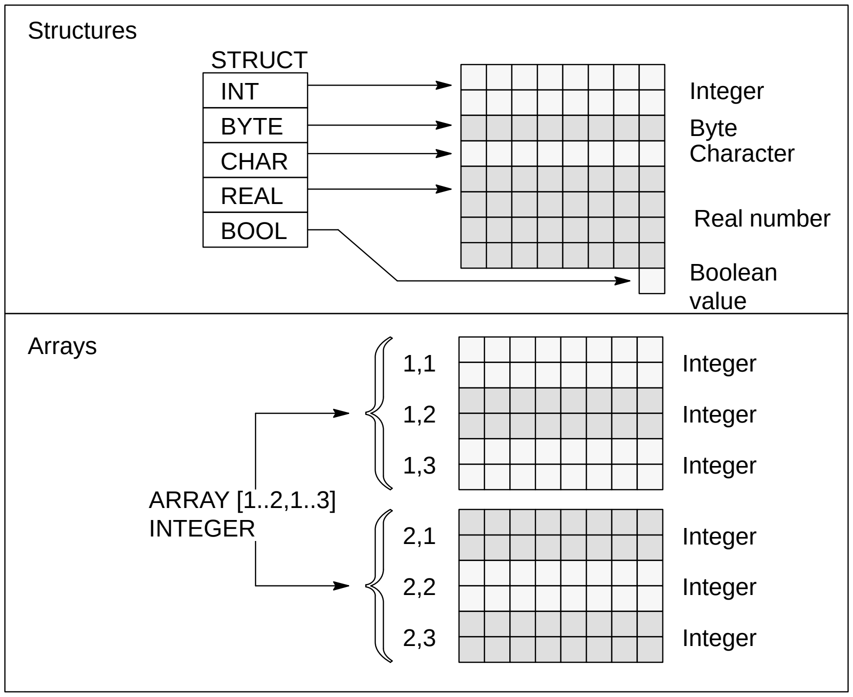


Figure C-1 Structure of Arrays and Structures

C.3 Using Arrays to Access Data

Arrays

An array combines a group of one data type (elementary or complex) to form a unit. You can create an array consisting of arrays. When you define an array, you must do the following:

- Assign a name to the array.
- Declare an array with the keyword ARRAY.
- Specify the size of the array using an index. You specify the first and last number of the individual dimensions (maximum 6) in the array. You enter the index in square braces with each dimension separated by a comma and the first and last number of the dimension by two periods. The following index defines, for example, a three-dimensional field:

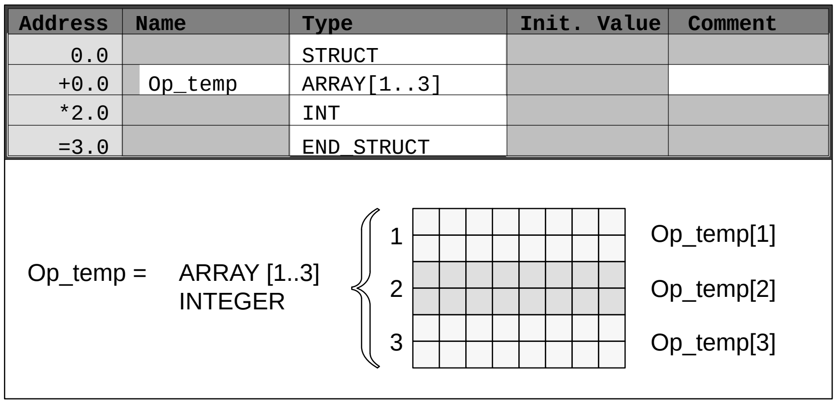
  [1..5,-2..3,30..32]

- You specify the data type of the data to be contained in the array.

Example

Figure C-2 shows an array with three integers. You access the data stored in an array using the index. The index is the number in square braces. The index of the second integer, for example is Op_temp[2].

An index can be any integer (-32768 to 32767) including negative values. The array in Figure C-2 could also be defined as ARRAY [-1..1]. The index of the first integer would then be Op_temp[-1], the second would be Op_temp[0] and the third integer would then be Op_temp[1].



Address	Name	Type	Init. Value	Comment
0.0		STRUCT		
+0.0	Op_temp	ARRAY[1..3]		
*2.0		INT		
=3.0		END_STRUCT		

Figure C-2 Array

An array can also describe a multi-dimensional group of data types. Figure C-3 shows a two-dimensional array consisting of integers. You access the data in a multi-dimensional array using the index. In this example in Figure C-3, the first integer is Op_temp[1,1], the third is Op_temp[1,3], the fourth is Op_temp[2,1], and the sixth is Op_temp[2,3].

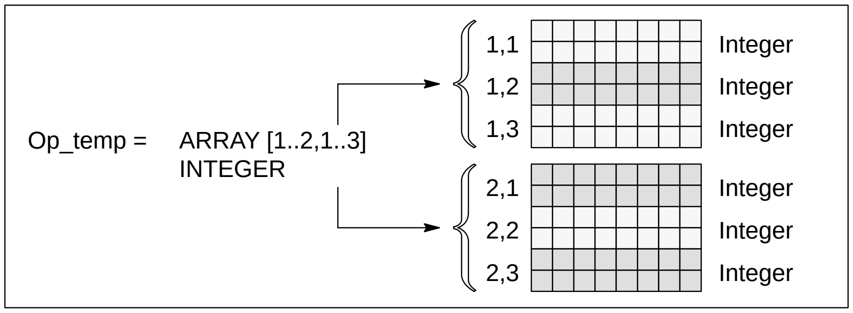


Figure C-3 Multi-dimensional Array

You can define up to a maximum of 6 dimensions (6 indexes) for a field. You could, for example, define the variable Op_temp as follows as a six-dimensional field:

ARRAY [1..3,1..2,1..3,1..4,1..3,1..4]

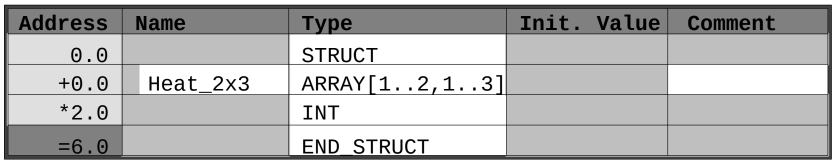
The index of the first element in this array is Op_temp[1,1,1,1,1,1]. The index of the last element Op_temp[3,2,3,4,3,4].

Creating Arrays

You define arrays when you declare the data in a DB or in the variable declaration. When you declare the array, you specify the keyword (ARRAY) followed by the size in square brackets, as follows:

[lower limit value..upper limit value]

In a multi-dimensional array, you also specify the upper and lower limit values and separate the individual dimensions by a comma. Figure C-4 illustrates the declaration for creating a field in the format 2 x 3 (like the array illustrated in Figure C-3).



Address	Name	Type	Init. Value	Comment
0.0		STRUCT		
+0.0	Heat_2x3	ARRAY[1..2,1..3]		
*2.0		INT		
=6.0		END_STRUCT		

Figure C-4 Creating an Array

Specifying Initial Values for an Array

When you are creating the arrays, you can assign an initial value to each element of the array. STEP 7 provides two methods for entering initial values:

- Entry of individual values: for each element of the array, you specify a value that is valid for the data type of the array. You specify the values in the order of the elements: [1,1]. Remember that the individual elements must be separated from each other by a comma.
- Specifying a repetition factor: with sequential elements that have the same initial value, you can specify the number of elements (the repetition factor) and the initial value for these elements. The format for entering the repetition factor is *x(y)*, where *x* is the repetition factor and *y* is the value to be repeated.

If you use the array declared in Figure C-4, you can specify the initial value for all six elements as follows: 17, 23, -45, 556, 3342, 0. You can set the initial value of all six elements to 10 by specifying 6(10). You could specify certain values for the first two elements and then set the remaining four elements to 0 by specifying the following: 17, 23, 4(0).

Accessing Data in an Array

You access the data in an array using the index of the element of the array. The index is used with the symbolic name.

Example: If the array declared in Figure C-4 begins at the first byte of DB20 (motor), you access the second element in the array with the following address:

Motor.Heat_2x3[1,2].

Using Arrays as Parameters

You can transfer arrays as parameters. If a parameter is declared in the variable declaration as ARRAY, you must transfer the entire array (and not individual elements). An element of an array can, however be assigned to a parameter when you call a block, providing the element of the array corresponds to the data type of the parameter.

If you use arrays as parameters, the arrays do not need to have the same name (they do not even need a name). Both arrays (the formal parameter and the actual parameter) must however have the same structure. An array in the format 2 x 3 consisting of integers, for example, can only be transferred as a parameter when the formal parameter of the block is defined as an array in the format 2 x 3 consisting of integers and the actual parameter that is provided by the call operation is also a field in the format 2 x 3 consisting of integers.

C.4 Using Structures to Access Data

Structures

A structure combines various data types (elementary and complex data types, including fields and structures) to form one unit. You can group the data to suit your process control. You can therefore also transfer parameters as a data unit and not as single elements. Figure C-5 illustrates a structure consisting of an integer, a byte, a character, a floating point number and a Boolean value.

A structure can be nested to a maximum of 8 levels (for example a structure consisting of structures containing arrays).

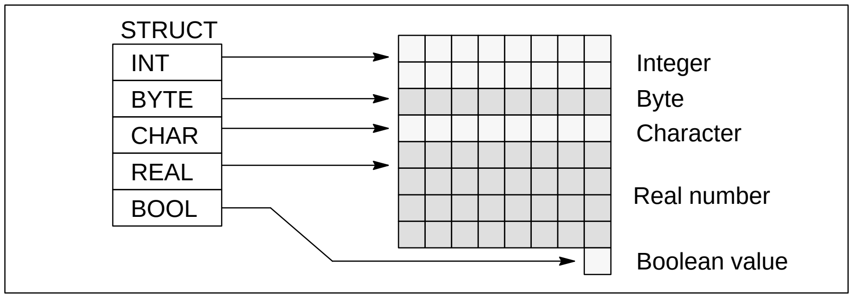
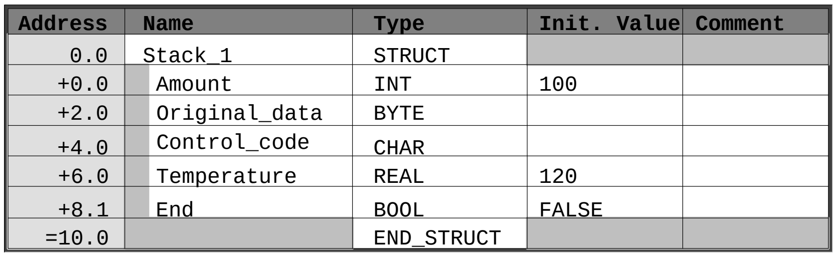


Figure C-5 Structure

Creating a Structure

You define structures when you declare data within a DB or in the variable declaration of a logic block.

Figure C-6 illustrates the declaration of a structure (*Stack_1*) that consists of the following elements: an integer (for saving the amount), a byte (for saving the original data), a character (for saving the control code), a floating point number (for saving the temperature) and a Boolean memory bit (for terminating the signal).



Address	Name	Type	Init. Value	Comment
0.0	Stack_1	STRUCT		
+0.0	Amount	INT	100	
+2.0	Original_data	BYTE		
+4.0	Control_code	CHAR		
+6.0	Temperature	REAL	120	
+8.1	End	BOOL	FALSE	
=10.0		END_STRUCT		

Figure C-6 Creating a Structure

Assigning Initial Values for a Structure

If you want to assign an initial value to every element of a structure, you specify a value that is valid for the data type and the name of the element. You can for example assign the following initial values (to the structure declared in Figure C-6):

Amount	=	100
Original_data	=	B#(0)
Control_code	=	‘C’
Temperature	=	120
End	=	False

Saving and Accessing Data in Structures

You access the individual elements of a structure. You can use symbolic addresses (for example *Stack_1.Temperature*). You can, however specify the absolute address at which the element is located (example: if *Stack_1* is located in DB20 starting at byte 0, the absolute address for amount is *DB20.DBW0* and the address for temperature is *DB20.DBD6*).

Using Structures as Parameters

You can transfer structures as parameters. If a parameter is declared as STRUCT in the variable declaration, you must transfer a structure with the same components. An element of a structure can, however, also be assigned to a parameter when you call a block providing the element of the structure corresponds to the data type of the parameter.

If you use structures as parameters, both structures (for the formal parameters and the actual parameters) must have the same components, in other words the same data types must be arranged in the same order.

C.5 Using User-Defined Data Types to Access Data

User-Defined Data Types

User-defined data types (UDTs) can combine elementary and complex data types. You can assign a name to UDTs and use them more than once. Figure C-7 illustrates the structure of the UDT consisting of an integer, a byte, a character, a floating point number, and a Boolean value.

Instead of entering all the data types singly or as a structure, you only need to specify “UDT20” as the data type and STEP 7 automatically assigns the corresponding memory space.

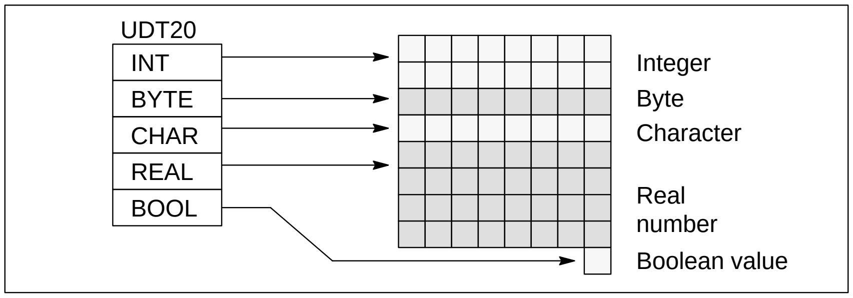
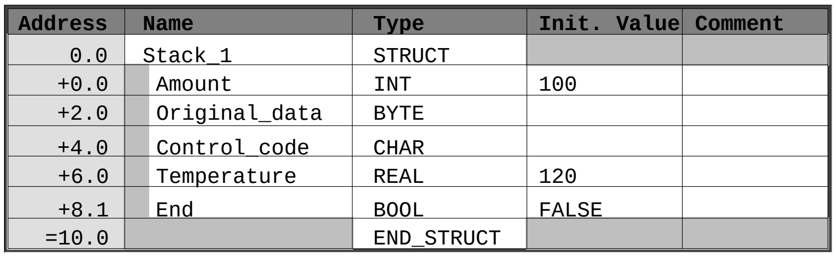


Figure C-7 User-Defined Data Type

Creating a User-Defined Data Type

You define UDTs with STEP 7. Figure C-8 illustrates a UDT consisting of the following elements: an integer (for saving the amount), a byte (for saving the original data), a character (for saving the control code), a floating point number (for saving the temperature) and a Boolean memory bit (for terminating the signal). You can assign a symbolic name to the UDT in the symbol table (for example *process data*).



Address	Name	Type	Init. Value	Comment
0.0	Stack_1	STRUCT		
+0.0	Amount	INT	100	
+2.0	Original_data	BYTE		
+4.0	Control_code	CHAR		
+6.0	Temperature	REAL	120	
+8.1	End	BOOL	FALSE	
=10.0		END_STRUCT		

Figure C-8 Creating a User-Defined Data Type

Once you have created a UDT, you can use the UDT like a data type if for example, you declare the data type *UDT200* for a variable in a DB (or in the variable declaration of an FB). Figure C-9 shows a DB with the variables *process_data_1* with the data type *UDT200*. You only specify *UDT200* and *process_data_1*. The arrays shown in italics are created when you compile the DB.

Address	Name	Type	Init. Value	Comment
0.0		STRUCT		
+6.0	Process_data_1	UDT200		
=6.0		END_STRUCT		

Figure C-9 Using a User-Defined Data Type

Assigning Initial Values for a User-Defined Data Type

To assign initial values to each element of a UDT, specify a value that is valid for the data type and the name of each element. You could, for example, assign the following initial values (for the UDT declared in Figure C-9):

Amount	=	100
Original_data	=	B#(0)
Control_code	=	‘C’
Temperature	=	120
End	=	False

If you declare a variable as a UDT, the initial values of the variables are the values you specified when you created the UDT.

Saving and Accessing Data in a User-Defined Data Type

You access the individual elements of a UDT. You can use symbolic addresses (for example *Stack_1.Temperature*). You can, however specify the absolute address at which the element is located (example: if *Stack_1* is located in DB20 starting at byte 0, the absolute address for amount is *DB20.DBW0* and the address for temperature is *DB20.DBD6*).

Using User-Defined Data Types as Parameters

You can transfer variables of the UDT type as parameters. If a parameter is declared as UDT in the variable declaration, you must transfer a UDT with the same structure. An element of a UDT can, however, also be assigned to a parameter when you call a block providing the element of the UDT corresponds to the data type of the parameter.

Advantages of DBs with an Assigned UDT

By using UDTs you have created once, you can generate a large number of data blocks with the same data structure. You can then use these data blocks to enter different actual values for specific tasks.

If, for example, you structure a UDT for a formula (for example for blending colors), you can assign this UDT to several DBs each containing different amounts.

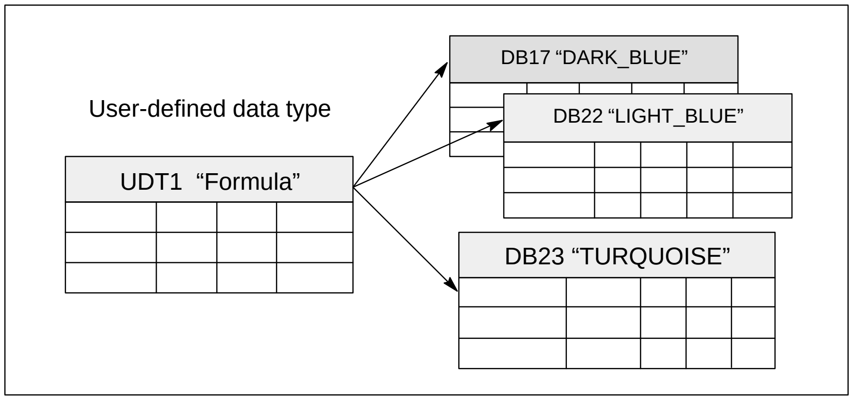


Figure C-10 Example of Assigning Several DBs to One UDT

The structure of the data block is determined by the UDT assigned to it.

C.6 Using the ANY Parameter Type

Overview

You can define formal parameters for a block that are suitable for actual parameters of any data type. This is particularly useful when the data type of the actual parameter that is provided when the block is called is unknown or can vary (and when any data type is permitted). In the variable declaration of the block, you declare the parameter as data type ANY. You can then assign an actual parameter of any data type in STEP 7.

STEP 7 assigns 80 bits of memory for a variable of the ANY data type. If you assign an actual parameter to this formal parameter, STEP 7 codes the start address, the data type and the length of the actual parameter in the 80 bits. (For more detailed information about the structure of the data saved in these 80 bits, refer to Section B.11.) The called block analyzes the 80 bits of data saved for the ANY parameter and obtains the information required for further processing.

Assigning an Actual Parameter to an ANY Parameter

If you declare the data type ANY for a parameter, you can assign an actual parameter of any data type to the formal parameter. In STEP 7, you can assign the following data types as actual parameters:

- Elementary data types: you specify the absolute address or the symbolic name of the actual parameter.
- Complex data types: you specify the symbolic name of the data with a complex data type (for example arrays and structures).
- Timers, counters and blocks: you specify the number (for example T1, C20 or FB6).

Figure C-11 shows how data are transferred to an FC with parameters of the ANY data type. In this example, FC100 has three parameters (*in_par1*, *in_par2* and *in_par3*) declared as the ANY data type.

- When FB10 calls FC100, FB10 transfers an integer (the static variable speed), a word (MW100) and a double word in DB10 (DB10.DBD40).
- When FB11 calls FC10, FB11 transfers a field of real numbers (the temporary variable thermo) a Boolean value (M 1.3) and a timer (T2).

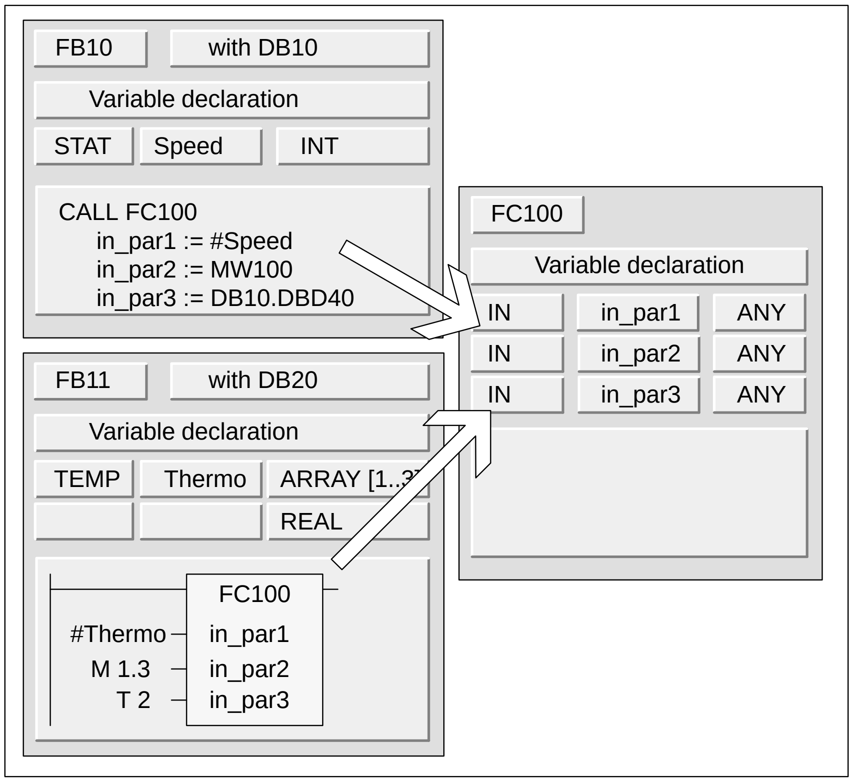


Figure C-11 Assigning Actual Parameters to an ANY Parameter

Specifying a Data Area for an ANY Parameter

It is, however, possible to assign not only individual addresses (for example MW100) to an ANY parameter but you can also specify a data area. If you want to assign a data area as the actual parameter, use the following format of a constant to specify the amount of data to be transferred:

p# Area ID *Byte.Bit* *Data Type* *Repetition Factor*

For the *data type* element, you can specify all elementary data types and the data type DATE_AND_TIME in the format for constants. If the data type is not BOOL, the bit address of 0 (x.0) must be specified. Table C-4 illustrates examples of the format for specifying memory areas to be transferred to an ANY parameter.

Table C-4 Using the Format for Constants for an ANY Parameter

Actual Parameter	Description
p# M 50.0 BYTE 10	Specifies 10 bytes in the byte memory area: MB50 to MB59.
p# DB10.DBX5.0 S5TIME 3	Specifies 3 units of data of the data type S5TIME, that are located in DB10: DB byte 5 to DB byte 10.
p# Q 10.0 BOOL 4	Specifies 4 bits in the outputs area: Q 10.0 to Q 10.3.

C.7 Assigning Data Types to Local Data of Logic Blocks

Valid Data Types

With STEP 7, the data types (elementary and complex data types and parameter types) that can be assigned to the local data of a block in the variable declaration are restricted.

Table C-5 illustrates the restriction for declaring local data for an OB. Since you cannot call an OB, an OB cannot have parameters (input, output or in/out). Since an OB does not have an instance DB, you cannot declare any static variables for an OB. The data types of the temporary variables of an OB can be elementary or complex data types and the data type ANY.

Table C-6 illustrates the restrictions when declaring local data for an FB. Due to the instance DB, there are less restrictions when declaring local data for an FB. When declaring input parameters there are no restrictions whatsoever; for an output parameter you cannot declare any parameter types, and for in/out parameters only the parameter types POINTER and ANY are permitted. You can declare temporary variables as the ANY data type. All other parameter types are illegal.

Table C-7 shows the restrictions when declaring local data for an FC. Since an FC does not have an instance DB, it also has no static variables. For input, output and in/out parameters of an FC, only the parameter types POINTER and ANY are permitted. You can also declare temporary variables of the ANY parameter type.

Table C-5 Valid Data Types for the Local Data of an OB

Declaration Type	Elementary Data Types	Complex Data Types	Parameter Types				
			TIMER	COUNTER	BLOCK	POINTER	ANY
Input	No	No	No	No	No	No	No
Output	No	No	No	No	No	No	No
In/out	No	No	No	No	No	No	No
Static	No	No	No	No	No	No	No
Temporary	Yes[1]	Yes[1]	No	No	No	No	Yes[1]

1 Located in the L stack of the OB

Table C-6 Valid Data Types for the Local Data of an FB

Declaration Type	Elementary Data Types	Complex Data Types	Parameter Types				
			TIMER	**COUNTER**	**BLOCK**	**POINTER**	**Any**
Input	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Output	Yes	Yes	No	No	No	No	No
In/out	Yes	Yes[1]	No	No	No	Yes	Yes
Static	Yes	Yes	No	No	No	No	No
Temporary	Yes[2]	Yes[2]	No	No	No	No	Yes[2]

1 Located as 48 bit pointer in the instance DB
2 Located in the L stack of the FB

Table C-7 Valid Data Types for the Local Data of an FC

Declaration Type	Elementary Data Types	Complex Data Types	Parameter Types				
			TIMER	**COUNTER**	**BLOCK**	**POINTER**	**Any**
Input	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Output	Yes	Yes	No	No	No	Yes	Yes
In/out	Yes	Yes	No	No	No	Yes	Yes
Static	No	No	No	No	No	No	No
Temporary	Yes[1]	Yes[1]	No	No	No	No	Yes[1]

1 Located in the L stack of the FC

C.8 Restrictions When Transferring Parameters

Restrictions When Transferring Parameters Between Blocks

When you assign actual parameters to formal parameters, you can either specify an absolute address, a symbolic address or a constant. STEP 7 restricts the valid assignments for the various parameters. Output and in/out parameters, for example, cannot be assigned a constant value (since the purpose of an output or an in/out parameter is to change its value). These restrictions apply particularly to parameters with complex data types to which neither an absolute address nor a constant can be assigned. Table C-8 illustrates the restrictions involving the data types of actual parameters that are assigned to formal parameters.

Table C-8 Restriction When Transferring Parameters Between Blocks

Elementary Data Types				
Declara-tion Type	**Absolute Address**	**Symbolic Name (in the Symbol Table)**	**Local Block Symbol**	**Constant**
Input	Yes	Yes	Yes	Yes
Output	Yes	Yes	Yes	No
In/out	Yes	Yes	Yes	No
Complex Data Type				
Declara-tion Type	**Absolute Address**	**Symbolic Name of the Element of the DB (in the Symbol Table)**	**Local Block Symbol**	**Constant**
Input	No	Yes	Yes	No
Output	No	Yes	Yes	No
In/out	No	Yes	Yes	No

Restrictions When an FC Calls Another FC

You can assign the formal parameters of a calling FC to the formal parameters of a called FC. Figure C-12 illustrates the formal parameters of FC10 that are assigned as actual parameters to the formal parameters of FC12.

STEP 7 restricts the assignment of formal parameters of an FC as actual parameters for the formal parameters of a different FC. You cannot, for example, assign parameters with complex data types or a parameter type as the actual parameter. Table C-9 shows the restrictions when assigning parameters when one FC calls another.

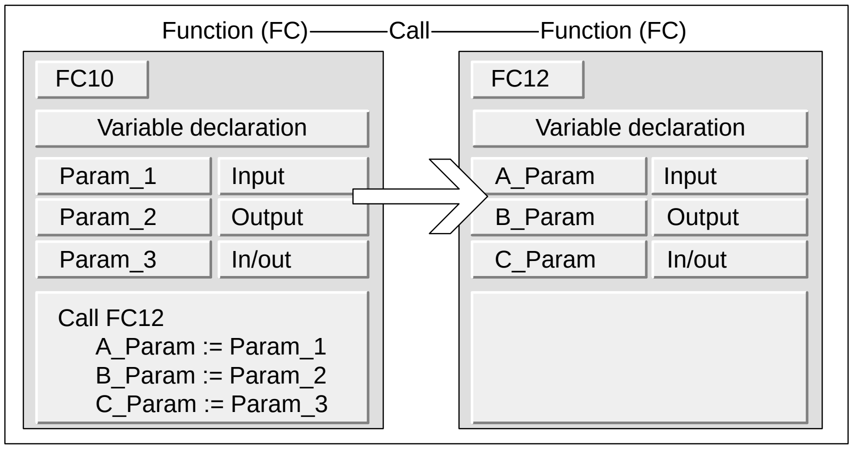


Figure C-12 Transferring Parameters from One FC to Another FC

Table C-9 Restrictions When One FC Calls Another

Declaration Type	Elementary Data Types	Complex Data Types	Parameter Types				
			TIMER	COUNTER	BLOCK	POINTER	ANY
Input → Input	Yes	No	No	No	No	No	No
Input → Output	No	No	No	No	No	No	No
Input → In/out	No	No	No	No	No	No	No
Output → Input	No	No	No	No	No	No	No
Output → Output	Yes	No	No	No	No	No	No
Output → In/out	No	No	No	No	No	No	No
In/out → Input	Yes	No	No	No	No	No	No
In/out → Output	Yes	No	No	No	No	No	No
In/out → In/out	Yes	No	No	No	No	No	No

Restrictions When an FC is Called by an FB

You can assign the formal parameters of a calling FB to the formal parameters of a called FC. Figure C-13 shows the formal parameters of FB10 that are assigned as the actual parameters for the formal parameters of FC12.

STEP 7 restricts the assignment of the formal parameters of an FB to the formal parameters of an FC. You cannot, for example, assign parameters of the parameter type as actual parameters. Table C-10 shows the restrictions for assigning parameters when an FB calls an FC.

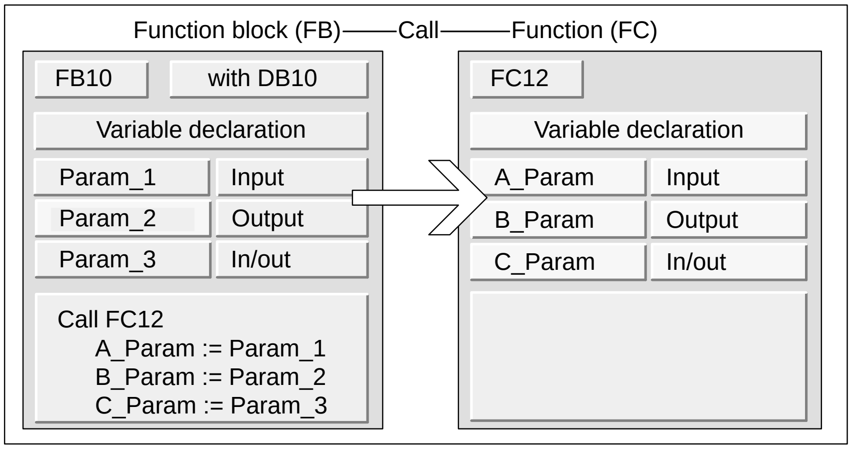


Figure C-13 Transferring Parameters from and FB to an FC

Table C-10 Restrictions When an FC is Called by a FB

Declaration Type	Elementary Data Types	Complex Data Types	Parameter Types				
			TIMER	COUNTER	BLOCK	POINTER	ANY
Input → Input	Yes	Yes	No	No	No	No	No
Input → Output	No	No	No	No	No	No	No
Input → In/out	No	No	No	No	No	No	No
Output → Input	No	No	No	No	No	No	No
Output → Output	Yes	Yes	No	No	No	No	No
Output → In/out	No	No	No	No	No	No	No
In/out → Input	Yes	No	No	No	No	No	No
In/out → Output	Yes	No	No	No	No	No	No
In/out → In/out	Yes	No	No	No	No	No	No

Restrictions when an FC Calls an FB

You can assign the formal parameters of a calling FC to the formal parameters of a called FB. Figure C-14 shows the formal parameters of FC10, that are assigned as actual parameters to the formal parameters of FB12.

STEP 7 restricts the assignment of formal parameters of an FC to the formal parameters an FB. You cannot, for example, assign parameters with a complex data type as actual parameters. You can, however, assign input parameters of the types TIMER, COUNTER, or BLOCK to the input parameters of the called FB. Table C-11 shows the restrictions for assigning parameters when an FC calls an FB.

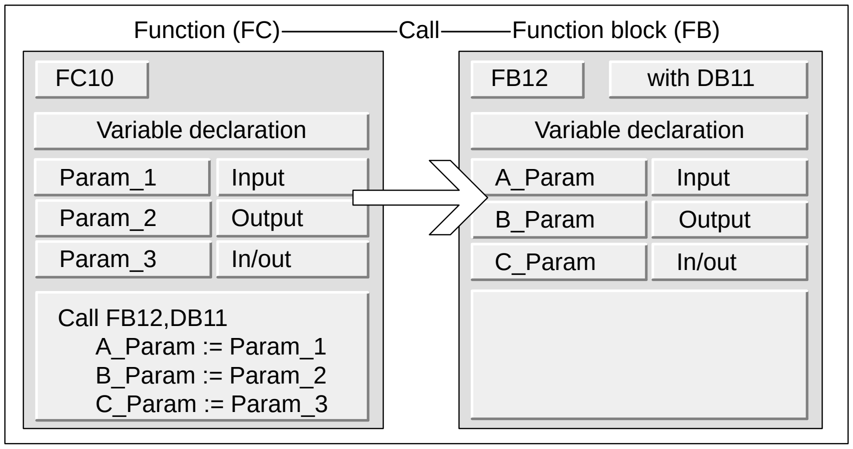


Figure C-14 Transferring Parameters from an FC to an FB

Table C-11 Restrictions when an FC Calls an FB

Declaration Type	Elementary Data Types	Complex Data Types	Parameter Types				
			TIMER	COUNTER	BLOCK	POINTER	ANY
Input → Input	Yes	No	Yes	Yes	Yes	No	No
Input → Output	No	No	No	No	No	No	No
Input → In/out	No	No	No	No	No	No	No
Output → Input	No	No	No	No	No	No	No
Output → Output	Yes	No	No	No	No	No	No
Output → In/out	No	No	No	No	No	No	No
In/out → Input	Yes	No	No	No	No	No	No
In/out → Output	Yes	No	No	No	No	No	No
In/out → In/out	Yes	No	No	No	No	No	No

Restrictions when an FB Calls Another FB

You can assign the formal parameters of a calling FB to the formal parameters of the called FB. Figure C-15 shows the formal parameters of FB10 that are assigned as actual parameters to the formal parameters of FB12.

STEP 7 restricts the assignment of the formal parameters of an FB to the formal parameters of another FB. You cannot, for example, assign input and output parameters with complex data types as the actual parameters for the input and output parameters of a called FB. You can, however, assign input parameters of the parameter types TIMER, COUNTER, or BLOCK to the input parameters of the called FB. Table C-12 shows the restrictions for assigning parameters when an FB calls another FB.

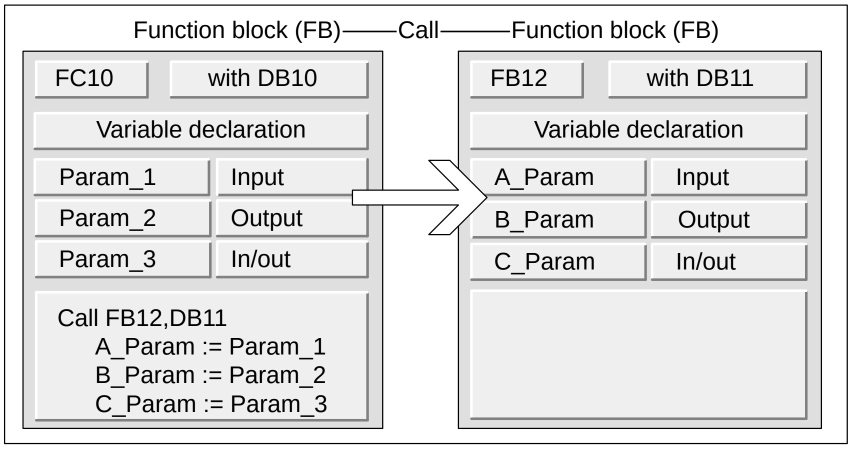


Figure C-15 Transferring Parameters from one FB to Another FB

Table C-12 Restrictions when one FB Calls Another FB

Declaration Type	Elementary Data Types	Complex Data Types	Parameter Types				
			TIMER	COUNTER	BLOCK	POINTER	ANY
Input → Input	Yes	Yes	Yes	Yes	Yes	No	No
Input → Output	No	No	No	No	No	No	No
Input → In/out	No	No	No	No	No	No	No
Output → Input	No	No	No	No	No	No	No
Output → Output	Yes	Yes	No	No	No	No	No
Output → In/out	No	No	No	No	No	No	No
In/out → Input	Yes	No	No	No	No	No	No
In/out → Output	Yes	No	No	No	No	No	No
In/out → In/out	Yes	No	No	No	No	No	No

References D

/30/ Primer: *S7-300 Programmable Controller,*
Quick Start

/70/ Manual: *S7-300 Programmable Controller,*
Hardware and Installation

/71/ Reference Manual: *S7-300, M7-300 Programmable Controllers*
Module Specifications

/72/ Instruction List: *S7-300 Programmable Controller*

/100/ Manual: *S7-400/M7-400 Programmable Controllers,*
Hardware and Installation

/101/ Reference Manual: *S7-400/M7-400 Programmable Controllers*
Module Specifications

/102/ Instruction List: *S7-400 Programmable Controller*

/230/ Manual: *Standard Software for S7,*
Converting S 5 Programs

/231/ User Manual: *Standard Software for S7 and M7,*
STEP 7

/232/ Manual: *Statement List (STL) for S7-300 and S7-400,*
Programming

/233/ Manual: *Ladder Logic (LAD) for S7-300 and S7-400,*
Programming

/235/ Reference Manual: *System Software for S7-300 and S7-400*
System and Standard Functions

/236/ Manual: *FBD for S7-300 and S7-400,*
Programming

/250/ Manual: *Structured Control Language (SCL) for S7-300 and S7-400,*
Programming

/251/ Manual: *GRAPH for S7-300 and S7-400,*
Programming Sequential Control Systems

/252/ Manual: *HiGraph for S7-300 and S7-400,*
Programming State Graphs

/254/ Manual: *Continuous Function Charts for S7-300, S7-400, M7,*
Continuous Function Charts

/270/ Manual: *S7-PDIAG for S7-300 and S7-400*
Configuring Process Diagnostics for LAD, FBD and STL

/500/ Manual: *SIMATIC NET,*
NCM S7 for Industrial Ethernet

/501/ Manual: *SIMATIC NET,*
NCM S7 for PROFIBUS

Glossary

A

Actual Parameters

Actual parameters replace the formal parameters when a function block (FB) or function (FC) is called. Example: the formal parameter “Start” is replaced by the actual parameter “I 3.6”.

Address

An address is part of a STEP 7 statement and specifies what the processor should execute the instruction on. Addresses can be absolute or symbolic.

B

Backplane Bus

The backplane bus of a SIMATIC S7 programmable logic controller supplies the modules in the rack with the internal operating voltage and allows data exchange between the modules. On the S7-400, the backplane bus is divided into the peripheral bus (P bus) and communication bus (C bus). On the S7-300, the backplane bus has a modular design in the form of U-shaped profiles that connect two modules together.

Backup

In SIMATIC S7, information stored in the RAM areas (in the work memory) can be:

- Saved by means of a backup battery; in this case the contents of the work memory and the read/write memory area of the load memory are retained, as are counters, timers, and the bit memory (the area can have parameters assigned)
- Saved without a backup battery (less maintenance); in this case a maximum (CPU-specific) number of data from the work memory, the read/write memory area of the load memory, and a maximum number of counters, timers, and memory bits can be saved permanently in the backup buffer of the CPU.

Backup Memory	The backup memory allows memory areas to be retained during power down without a backup battery. A selectable number of timers, counters, memory bits and bytes of a data block can be declared as backed up or retentive.
Block	Blocks are part of the user program and can be distinguished by their function, their structure, or their purpose. STEP 7 provides the following types of blocks: • Logic blocks (FB, FC, OB, SFB, SFC) • Data blocks (DB, SDB) • User-defined data types (UDT)
Block Stack	The block stack (B stack) in the system memory of the CPU contains the return addresses and the data block register when blocks are called.

C

Central Processing Unit (CPU)	The CPU is the central module in a programmable controller in which the user program is stored and processed. It consists of an operating system, processing unit, and communication interfaces.
Communication Bus (K Bus)	The communication bus (K bus) is part of the backplane bus of the SIMATIC S7-300, S7-400 programmable logic controllers. It allows fast communication between programmable modules, the CPU and the programming device. This means that, for example, all the programmable modules in a programmable controller can be programmed using one programming device connected to the CPU.
Communication SFBs for Configured Connections	The communication SFBs are system function blocks for exchanging data and for program management. Examples for data exchange: USEND, ERCV, GET. Examples of program management: setting the CPU of the remote communication partner to the STOP mode, querying the status of the remote communication partner
Communication SFCs for Non-Configured Connections	The communication SFCs are system functions for exchanging data and for aborting existing connections established by communication SFCs.

Complete Restart	When a CPU starts up (for example, when the mode selector is moved from STOP to RUN or when power is turned on), before cyclic program processing starts (OB1), either the organization block OB101 (restart; only in the S7-400) or OB100 (complete restart) is processed first. In a complete restart the process-image input table is read in and the STEP 7 user program processed starting with the first statement in OB1.
Connection	A connection is established between stations that exchange data with each other. A connection is only possible when the stations are attached to a common physical medium (for example a bus system). A logical connection (software) is then established between the stations.
Counter (C)	Counters are an area in the system memory of the CPU. The contents of these counters can be changed using STEP 7 instructions (for example, up counter, down counter).

D

Data, Static	Static data are the local data of a function block that are saved in the instance data block and are therefore retained until the function block is executed again.
Data, Temporary	Temporary data are local data of a block that are located in the L stack while the block is being executed. When execution of the block is completed, the data are no longer available.
Data Block (DB)	Data blocks are areas in the user program which contain user data. There are shared data blocks which can be accessed by all logic blocks, and there are instance data blocks which are associated with a particular function block (FB) call. In contrast to all other block types, data blocks do not contain any instructions.
Data Type	Using a data type, you specify how the value of a variable or constant is used in the user program. In SIMATIC S7, there are two types of data type available complying with IEC 1131-3: elementary data types and complex data types.
Data Type, Complex	Complex data types are created by the user with the data type declaration. They do not have a name of their own and cannot be used more than once. A distinction is made between arrays and structures. The data types STRING and DATE AND TIME belong to this category.

Data Type, Elementary	Elementary data types are predefined data types complying with IEC 1131-3. Examples: data type ”BOOL” defines a binary variable (bit); data type ”INT” defines a 16-bit fixed-point variable.
Data Type, User-Defined (UDT)	User-defined data types are created by the user with the data type declaration. They have their own names and can therefore be used more than once. A user-defined data type can, for example, be used to create several data blocks with the same structure (for example a controller).
Diagnostic Buffer	The diagnostic buffer is a retentive area of memory within the CPU which stores the diagnostic events in the order they occurred.
Diagnostic Event	A diagnostic event causes an entry in the diagnostic buffer of the CPU. The diagnostic events are divided into the following groups: faults on a module, faults in the system wiring, system errors on the CPU itself, mode change errors on the CPU, errors in the user program and user-defined diagnostic events.
Distributed Peripheral I/Os (DP)	Distributed I/Os are modules located at some distance from the central rack (for example analog and digital modules). The distributed I/Os are characterized by the techniques used to install them. The aim is to reduce wiring (and costs) by installing the modules close to the process.
DP Standard	DP standard indicates data exchange complying with EN 50170, previously DIN E 19245, Part 3.

F

Formal Parameter	A formal parameter is a placeholder for the actual parameter in logic blocks that can be assigned parameters. In FBs and FCs, the formal parameters are declared by the user; in SFBs and SFCs, they already exist. When a block is called, an actual parameter is assigned to the formal parameter so that the called block works with the latest value. The formal parameters belong to the local data of the block and are declared as input, output, and I/O parameters.
Function (FC)	According to the International Electrotechnical Commission’s IEC 1131–3 standard, functions are logic blocks that do not have a ’memory’. A function allows you to transfer parameters in the user program, which means they are suitable for programming complex functions that are required frequently, for example, calculations. Important: Since there is no ’memory’, the calculated values must be processed immediately following the FC call.

Function Block (FB)	According to the International Electrotechnical Commission’s IEC 1131-3 standard, function blocks are logic blocks that have a ’memory’. A function block allows you to pass parameters in the user program, which means they are suitable for programming complex functions that are required frequently, for example, control systems, operating mode selection. Important: Since a function block has a ’memory’ (instance data block), it is possible to access their parameters at any point in the user program.

G

Global Data Communication	Global data communication is a procedure with which global data are transferred between CPUs.

I

Instance	Instance means the call for a function block. If, for example, a function block is called five times in an S7 user program, five instances exist. An instance data block is assigned to every call.
Instance Data Block (DB)	An instance data block is used to save the formal parameters and the static local data of function blocks. An instance DB can be assigned to a function block call or to a call hierarchy of function blocks.
Instruction	An instruction is part of a STEP 7 statement and specifies what the processor should do.
Interrupt Stack	If an interrupt or error occurs, the CPU enters the address of the point at which the interrupt occurred and the current status bits and contents of the accumulators in the interrupt stack (I stack) in the system memory. If more than one interrupt has occurred, a multi-level I stack results. The I stack can be read out with a programming device.

L

Local Data	Local data are assigned to a logic block and that were declared in its declaration table or variable declaration. Depending on the block, they include: formal parameters, static data, temporary data.

Local Data Stack

The local data stack (L stack) in the system memory of the CPU contains part of the local data, known as the temporary data.

Logic Block

In SIMATIC S7, a logic block is a block that contains part of the STEP 7 user program. The other type of block is a data block which contains only data. The following list shows the types of logic blocks:

- Organization block (OB)
- Function block (FB)
- Function (FC)
- System function block (SFB)
- System function (SFC)

M

Memory Reset (MRES)

The memory reset function deletes the following memories in the CPU:

- Work memory
- Read/write area of the load memory
- System memory with the exception of the MPI parameters and the diagnostic buffer

Multicomputing Interrupt

The multicomputing interrupt belongs the priority classes of the operating system of an S7 CPU. On the S7-400, it is generated by a CPU after the CPU has received an interrupt. The appropriate organization block is then called.

Multicomputing Mode

The multicomputing mode on the S7-400 is the simultaneous operation of more than one (maximum four) CPUs in one rack.

Multipoint Interface

The multipoint interface is the programming device interface in SIMATIC S7. It allows simultaneous use of more than one programming device, text display, or operator panel with one or more CPUs. The stations on the MPI are interconnected by a bus system.

N

Network

A network connects network nodes via a cable and allows communication between the nodes.

Node Address	A node address is used to access a device (for example a programming device) or a programmable module (for example a CPU) in a network (for example MPI, PROFIBUS).

O

Organization Block (OB)	Organization blocks form the interface between the CPU operating system and the user program. The sequence in which the user program should be processed is laid down in the organization blocks.

P

Parameter	A parameter is a variable of an S7 logic block (actual parameter, formal parameter) or a variable for setting the behavior of a module. Every configurable module has a basic parameter setting when it is supplied from the factory, but this can be changed using STEP 7.
Peripheral Bus (P Bus)	The peripheral bus (P bus) is part of the backplane bus in the programmable logic controller. It is optimized for the fast exchange of signals between the CPU(s) and the signal modules. User data (for example digital input signals of a signal module) and system data (for example default parameter records of a signal module) are transferred on this bus.
Priority Class	The operating system of an S7 CPU provides a maximum of 28 priority classes (or program execution levels), to which various organization blocks (OBs) are assigned. The priority classes determine which OBs can interrupt other OBs. If a priority class includes more than one OB, these do not interrupt each other but are executed in the order in which they are called.
Process Image	The signal states of the digital input and output modules are stored in the CPU in a process image. There is a process-image input table (PII) and a process-image output table (PIQ).
PROFIBUS	PROFIBUS stands for ”Process Field Bus” and is an open communications standard for networking field devices (for example programmable controllers, drives, actuators, sensors).
Programmable Controller (PLC)	A programmable (logic) controller consists of a central rack, a CPU and various input and output modules.

Programming Device (PG)	A personal computer with a special compact design, suitable for industrial conditions. A programming device is completely equipped for programming the SIMATIC programmable logic controllers.
Project	A project is a container for all objects in an automation task, independent of the number of stations, modules, and how they are connected in a network.

R

Restart	When a CPU starts up (for example, when the mode selector is moved from STOP to RUN or when the power is turned on), before cyclic program processing starts (OB1), either the organization block OB100 (complete restart) or the organization block OB101 (restart; only in the S7-400) is processed first. In a restart the process-image input table is read in and the STEP 7 user program processing is restarted at the point where it was interrupted by the last stop (STOP, power off).

S

S7 Program	An S7 program is a container for blocks, source files, and charts for S7 programmable modules.
Shared Data	Shared data are data which can be accessed from any logic block (function (FC), function block (FB), organization block (OB)). These are bit memory (M), inputs (I), outputs (Q), timers (T), counters (C), and elements of data blocks (DB). You can access shared data either absolutely or symbolically.
SINEC L2-DP	SINEC L2-DP is the Siemens product name for the PROFIBUS DP.
Start Event	Start events are defined events such as errors or interrupts and cause the operating system to start the corresponding organization block.
STARTUP Mode	The CPU goes through the STARTUP mode during the transition from the STOP mode to the RUN mode. It can be set using the mode selector, or following power-on, or by an operation on the programming device. A distinction is made between a complete restart and a restart. In the S7-300, a complete restart is executed. In the S7-400, either a complete restart or a restart is executed, depending on the position of the mode selector.

Symbol

A symbol is a name defined by the user, taking syntax rules into consideration. This name can be used in programming and in operating and monitoring once you have defined it (for example, as a variable, a data type, a jump label, or a block).
Example: Address: I 5.0, Data Type: BOOL, Symbol: Emer_Off_Switch

Symbol Table

A table used to assign symbols (or symbolic names) to addresses for shared data and blocks.
Examples: Emer_Off (Symbol), I 1.7 (Address)
Controller (Symbol), SFB24 (Block)

System Error

System errors are errors which can occur within a programmable logic controller (and are not related to the process). Some examples of system errors are program errors in the CPU and defects on modules.

System Function (SFC)

A system function (SFC) is a function integrated in the CPU operating system which can be called in the user program when required. Its associated instance data block is found in the work memory.

System Function Block (SFB)

A system function block (SFB) is a function block integrated in the CPU operating system which can be called in the STEP 7 user program when required.

System Memory

The system memory is integrated in the CPU and executed in the form of RAM. The address areas (timers, counters, bit memory, etc.) and data areas required internally by the operating system (for example, backup for communication) are stored in the system memory.

System Status List (SZL)

The system status list SZL describes the current status of a programmable logic controller: it provides information about the configuration, the current parameter assignment, the current statuses and sequences on the CPU, and about the modules assigned to the CPU. The data of the system status list can only be read and cannot be modified. It is a virtual list that is only created when requested.

T

Timer (T)

Timers are an area in the system memory of the CPU. The contents of these timers is updated by the operating system asynchronously to the user program. You can use STEP 7 instructions to define the exact function of the timer (for example, on-delay timer) and start processing it (Start).

U

User Program	The user program contains all the statements and declarations and the data required for signal processing to control a plant or a process. The program is linked to a programmable module (for example, CPU, FM) and can be structured in the form of smaller units (blocks).

V

Variable	A variable defines an item of data with variable content which can be used in the STEP 7 user program. A variable consists of an address (for example, M 3.1) and a data type (for example, BOOL), and can be identified by means of a symbolic name (for example, BELT_ON).
Variable Declaration	The variable declaration includes the specification of a symbolic name, a data type (and possibly an initialization value), an address, and comment.
Variable Declaration Table	The local data of a logic block are declared in the variable declaration table when the program is created using incremental input.
Variable Table (VAT)	The variable table is used to collect together the variables that you want to monitor and modify and set their relevant formats.

Index

Siemens AG
AUT E 146

Östliche Rheinbrückenstr. 50
D-76181 Karlsruhe
Federal Republic of Germany

From:
Your Name: _
Your Title: _
Company Name: _
Street: _
City, Zip Code_ _
Country: _
Phone: _

Please check any industry that applies to you:

❐ Automotive	❐ Pharmaceutical
❐ Chemical	❐ Plastic
❐ Electrical Machinery	❐ Pulp and Paper
❐ Food	❐ Textiles
❐ Instrument and Control	❐ Transportation
❐ Nonelectrical Machinery	❐ Other _ _ _ _ _ _ _ _ _ _ _
❐ Petrochemical	

Remarks Form

Your comments and recommendations will help us to improve the quality and usefulness of our publications. Please take the first available opportunity to fill out this questionnaire and return it to Siemens.

Please give each of the following questions your own personal mark within the range from 1 (very good) to 5 (poor).

1.	Do the contents meet your requirements?	☐
2.	Is the information you need easy to find?	☐
3.	Is the text easy to understand?	☐
4.	Does the level of technical detail meet your requirements?	☐
5.	Please rate the quality of the graphics/tables:	☐

Additional comments:

_ _

_ _

_ _

_ _

_ _

_ _

_ _

_ _

_ _

_ _

_ _

_ _