SIEMENS

Preface, Contents

Part 1: Designing Programs

Part 2: Operating and Debugging

Part 3: Language Description

Appendix

Glossary, Index

SIMATIC

Structured Control Language (SCL) for S7-300/S7-400 Programming

Manual

This manual has the order number:

6ES7811-1CA02-8BA0

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.

Trademarks

SIMATIC®, SIMATIC NET® and SIMATIC HMI® 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.

Preface

Purpose

This manual is your guide to creating user programs in the Structured Control Language (SCL) programming language. The manual explains the basic procedures for creating programs using the SCL editor, SCL compiler and SCL debugger.

This manual also includes a reference section that describes the syntax and functions of the language elements of SCL.

Audience

This manual is intended for S7 programmers, commissioning engineers, and maintenance/service personnel. A working knowledge of automation procedures is essential.

Scope of the Manual

This manual is valid for release 3.0 of the STEP 7 standard programming software package.

Compliance with Standards

SCL corresponds to the Structured Control Language defined in the DIN EN-61131-3 (IEC 1131-3) standard, although there are essential differences with regard to the operations. For further details, refer to the table of standards in the STEP 7 file NORM.TAB.

Overview of the STEP 7 Documentation

There is a wide range of both general and task-oriented user documentation available to support you when configuring and programming an S7 programmable controller. The following descriptions and the figure below will help you to find the user documentation you require.

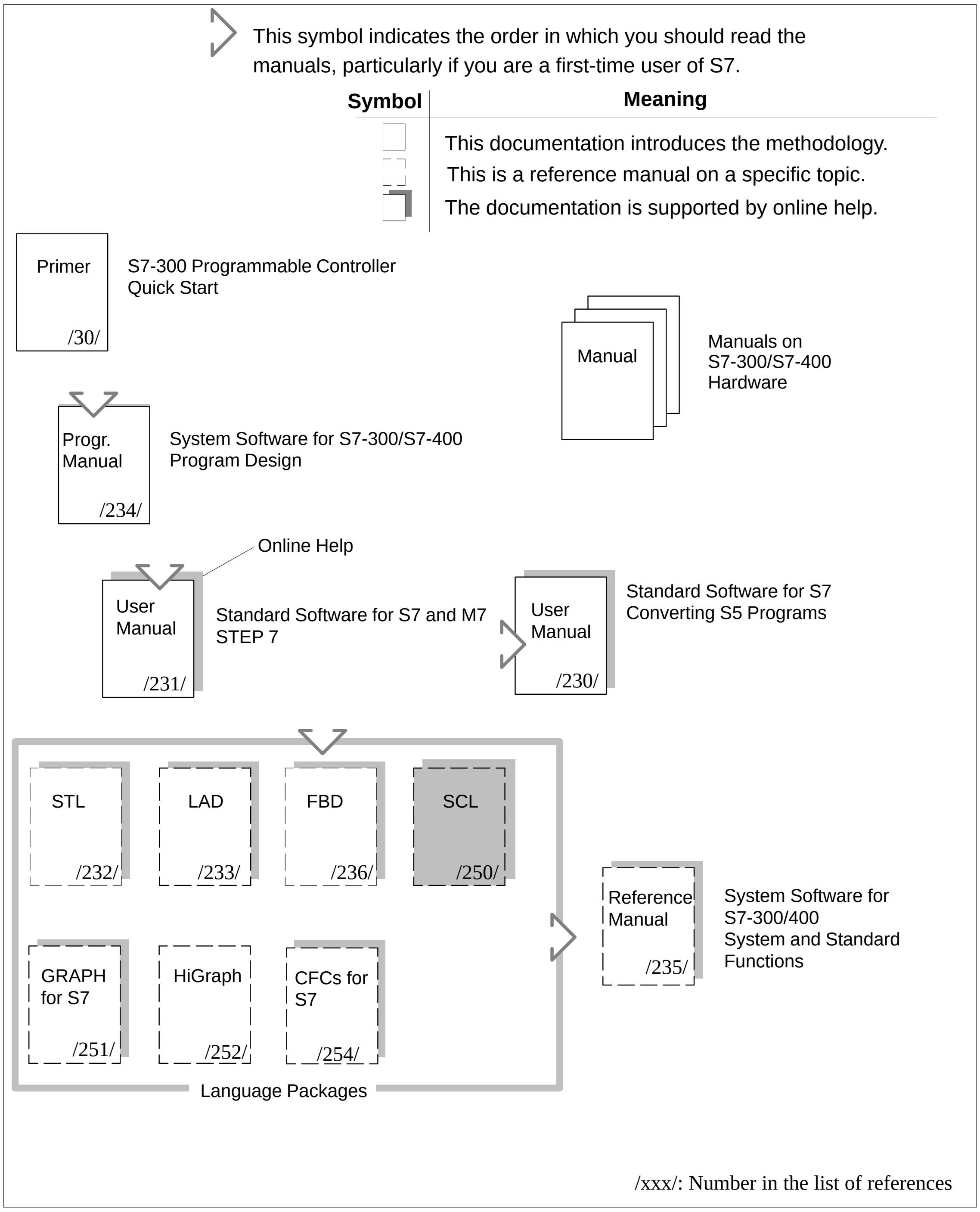

This symbol indicates the order in which you should read the manuals, particularly if you are a first-time user of S7.
Symbol
Meaning
This documentation introduces the methodology.
This is a reference manual on a specific topic.
The documentation is supported by online help.
Primer
/30/
S7-300 Programmable Controller Quick Start
Manual
Manuals on S7-300/S7-400 Hardware
Progr. 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 Converting S5 Programs
STL /232/
LAD /233/
FBD /236/
SCL /250/
GRAPH for S7 /251/
HiGraph /252/
CFCs for S7 /254/
Language Packages
Reference Manual
/235/
System Software for S7-300/400 System and Standard Functions
/xxx/: Number in the list of references

Table 1-1 Summary of the Documentation

Title	Subject
S7-300 Programmable Logic 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.
S7-300/400 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 read this manual to get a basic overview of programming methods on which to base the design of a user program.
S7-300/400 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 User 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) which provides you with detailed information about using the editors and compilers.

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

How to Use This Manual

To use this SCL manual effectively, you should already be familiar with the theory behind S7 programs. This is explained in the *Programming Manual* **/234/.** The language packages also use the standard software for S7, so you you should also be familiar with the standard software as described in the *User Manual* **/231/**.

The manual is divided into the following parts:

- Chapter 1 introduces you to programming with SCL.
- Chapter 2 describes the design process on the basis of an example which you can also run.
- Chapters 3 to 6 demonstrate how to use the SCL development environment. They introduce you to the SCL Editor, Compiler and Debugger.
- Chapters 7 to 19 form the reference section which provides you with detailed information about the functions of the individual SCL instructions.

The Appendix contains the following:

- A complete explanation of the SCL syntax conventions.
- The glossary includes definitions of the basic terms.
- The index will help you to locate a topic 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 D.

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 contact 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.

Notes on Using the Manual

The user’s guide sections in this manual do not describe procedures in step-by-step detail, but simply outline basic procedures. You will find more detailed information on the individual dialogs in the software and how to use them in the online help.

Contents

Part 1: Designing Programs

Part 2: Operating and Debugging

Part 3: Language Description

Appendix

Part 1: Designing Programs

1 Product Overview

SCL Programming Language

Apart from their traditional control tasks, programmable controllers nowadays increasingly have to perform data management tasks and complex mathematical operations. It is for these functions in particular that we offer SCL for S7300/400 (Structured Control Language), the programming language that makes programming easier and conforms to IEC 113-3.

SCL not only assists you with “normal” control tasks but also with extensive applications and is thus superior to the “traditional” programming languages in the following areas of application:

- Data management
- Process optimization
- Recipe management
- Mathematical/statistical operations

Technical Specifications

In order to be able to work with SCL, you need a SIMATIC programming device or a PC (80486 processor or higher, 16 Mbytes of RAM).

Language Capability	

Operators	`Exponential/Mathematical`
	`Comparators`
	`Links`

Functions	`Timers/Counters`
	`Function block calls`

Control structures	`Loops (FOR/WHILE/REPEAT)`
	`Alternatives (IF THEN/CASE/GOTO)`

Data Types	

Elementary	`BOOL/BYTE/WORD/DWORD/`
	`INT/DINT/REAL/TIME/`
	`TIME_OF_DAY`

Complex	`Strings/Arrays/Structures/User-defined`

Chapter Overview

1.1 What is SCL?

High-Level Programming Language

SCL (*S̲tructured C̲ontrol L̲anguage*) is a high-level textual programming language which is based on PASCAL. It is also based on a standard for PLCs (programmable logic controllers).

The standard DIN EN-61131-3 (IEC 1131-3) sets down standardized requirements for programming languages for programmable controllers. The basis for SCL is the section *"structured text"*. For precise details of standards conformity, refer to the "Compliance List" in the NORM.TBL file in STEP 7.

In addition to high-level language elements, SCL also includes language elements typical of PLCs such as inputs, outputs, timers, bit memory, block calls, etc. In other words, SCL complements and extends the STEP 7 programming software and its programming languages Ladder Logic and Statement List.

Development Environment

For optimum use and practical application of SCL, there is a powerful development environment which is matched both to specific characteristics of SCL and STEP 7. This development environment consists of the following components:

- an **Editor** for writing programs consisting of functions (FCs), function blocks (FBs), organization blocks (OBs), data blocks (DBs) and user-defined data types (UDTs); the programmer is supported in his/her tasks by powerful functions;
- a **Batch Compiler** for translating the program written using the Editor into MC7 machine code. The MC7 code generated will run on all S7-300/400 CPUs from CPU 314 upwards;
- a **Debugger** which enables the programmer to check for logical programming errors within an error-free environment; the debugging operation is performed in the source language.

The individual components are simple and convenient to use since they run under Windows 95 and thus benefit from all the advantages of that system.

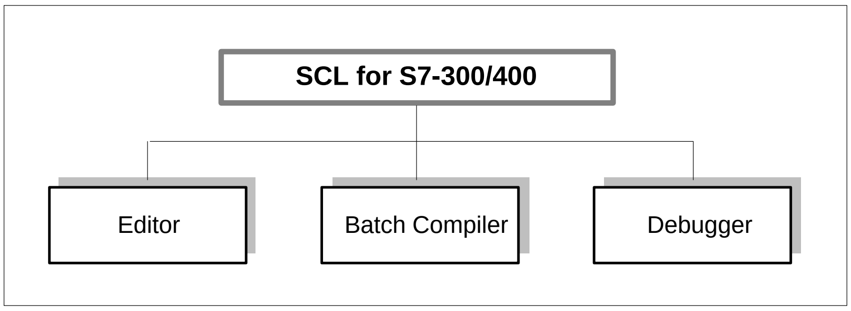


Figure 1-1 SCL development environment

1.2 What Are the Advantages of SCL?

High-Level Programming Language

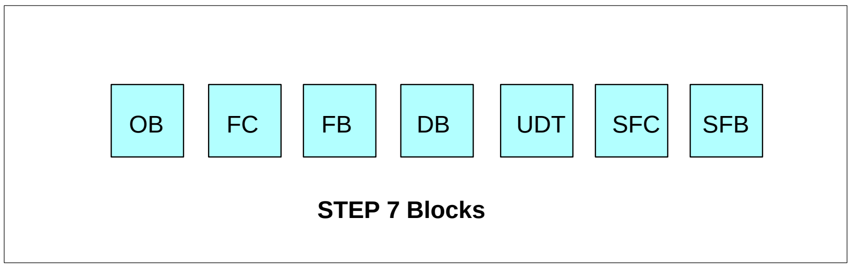
SCL offers you all the advantages of a high-level programming language. In addition, however, it also has a number of characteristics designed to provide assistance with structured programming, such as:

- the block structure of STEP 7
- ready-made blocks
- compatibility with STEP 5

Proven Block Structure of STEP 7

SCL is ideally suited to dealing with all the tasks involved in automation projects, which means that you can combine SCL effectively with STEP 7 at all stages of your project.

In particular, SCL supports the STEP 7 block concept and therefore, alongside Statement List and Ladder Logic, enables standardized block programming.



Types of Block

STEP 7 blocks are subunits of a user program which are delimited on the basis of their structure or purpose. SCL provides the facility for creating the following types of blocks:

Abbrevi-ation	Block Type	Function
OB	Organization block	Interface between operating system and user program
FC	Function	Block with parameter transfer capability but no memory
FB	Function block	Block with parameter transfer capability and memory
DB	Data block	Block for storing user data
UDT	User-defined data type	Block for storing user-defined data types

Ready-Made Blocks

You do not have to program every function yourself. You can also make use of ready-made blocks. These are integrated in the CPU operating system or stored in libraries *(S7lib)* in the STEP 7 Standard package and can be used to program communications functions, for example. The specific block types involved are as follows:

Abbrevi-ation	Block Type	Function
SFC	System function	Characteristics similar to a function (FC)
SFB	System function block	Characteristics similar to a function block (FB)

Mutual Compatibility of Blocks

You can use blocks programmed using SCL in combination with Statement List (STL), Ladder Logic (LAD), and Function Block Diagram (FBD) blocks. This means that a block written in SCL can call a block written in STL, LAD, or FBD. In the same way, SCL blocks can be called by STL, LAD, or FBD programs. The programming languages of STEP 7 and SCL (optional package) thus complement one another perfectly.

Decompilation

SCL blocks can be recompiled into the STEP 7 programming language Statement List. Recompilation from STL to SCL is not possible.

Compatibility with STEP 5

Blocks written in SCL for STEP 5 are, apart from a few exceptions, upwardly compatible; that is, they can also be edited, compiled and tested using SCL for STEP 7.

Programming Methods

Thanks to modern software engineering techniques, SCL supports structured programming.

Ease of Learning

Provided you have some experience of using a high-level programming language, SCL is easy to learn because the repertoire of language constructs in SCL is based on other high-level programming languages.

1.3 Performance Characteristics of the Development Environment

Editor

The SCL Editor is a text editor which can be used for editing any text files. Its central purpose is the creation and editing of source files for STEP 7 programs. In a source file you can write one or more program blocks (see below).

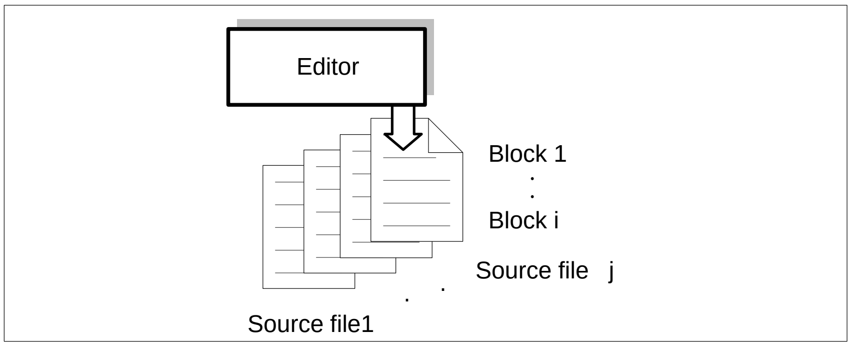


Figure 1-2 SCL Editor

The SCL Editor allows you to:

- Edit a complete source file incorporating one or more blocks
- Edit a compilation control file which with which you can automate the compilation of a series of source files
- Use additional functions which simplify the task of editing the source file, for example, Search and Replace
- Customize the Editor settings to suit your specific requirements

The Editor does not check the syntax of text while it is being entered.

Compiler

Once you have created your source files using the SCL Editor, you must translate them into MC code.

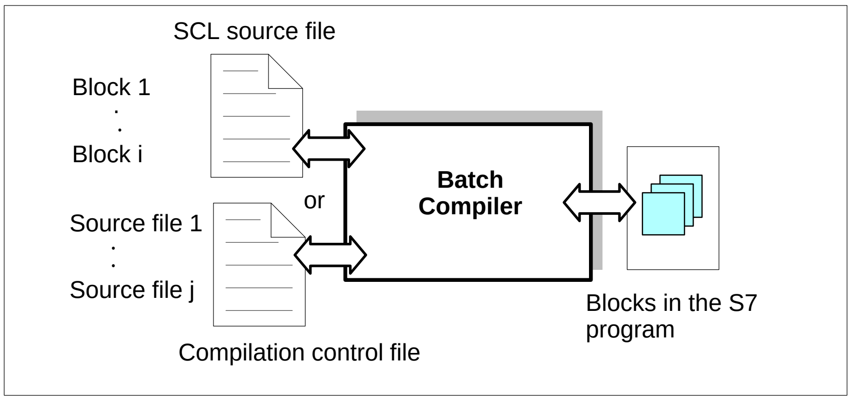


Figure 1-3 SCL Compiler

The SCL Compiler allows you to:

- Compile an SCL source file consisting of a number of blocks in a single compilation run
- Compile a series of SCL source files using a compilation control file which specifies the names of the source files
- Customize the Compiler settings to suit your specific requirements
- view all errors and warning messages which occur during the compilation process
- Easily locate errors in the source file with an additional facility which provides descriptions of the errors and hints on how to rectify them.

Debugger

The SCL Debugger provides a means of checking how a program will run on the PLC and thereby a means of identifying any logical errors.

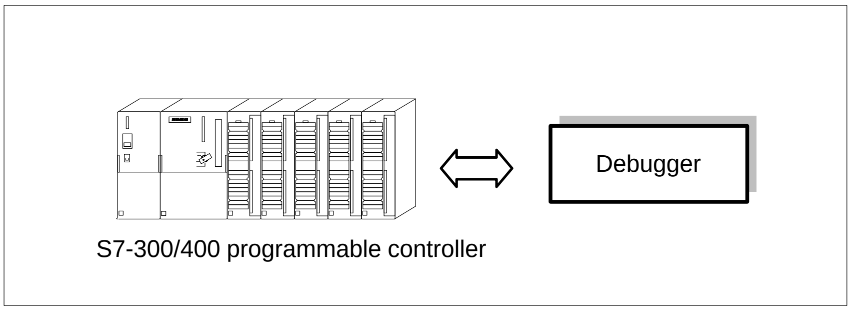


Figure 1-4 SCL Debugger

SCL provides two different debugging modes:

- **single-step** monitoring – this follows the logical processing sequence of the program; you can execute the program algorithm one instruction at a time and observe how the variable values being processed alter in a Result window;
- **continuous** monitoring – in this mode you can test out a group of instructions within a block of the source file; during the test run the values of the variables and parameters are displayed in chronological sequence and – where possible – cyclically updated.

STEP 7 Standard Package

The SCL development environment allows you to perform STEP 7 standard package functions such as displaying and modifying the CPU mode and setting the time directly from within SCL.

Designing SCL Programs 2

Introduction

Experience shows that the easiest and quickest way to program is if you structure your tasks by splitting them up into individual self-contained sections. SCL helps you to do this by enabling you to design individual blocks efficiently.

This chapter describes how to design and implement a user program in SCL. The explanations are illustrated by a sample program which you can run using the debugging data supplied and your own input and output modules.

Chapter Overview

2.1 Overview

Objective

The design section shows you how to use SCL effectively. At first, you will probably have lots of questions, such as:

- How do I go about creating a program with SCL?
- Which SCL language functions are suitable for performing the task?
- What debugging functions are there for me to use?

These and other questions are answered in this section.

SCL Language Functions

The sample program introduces the following SCL language functions, among others:

- Structure and use of the various SCL block types
- Calling blocks with transfer and analysis of parameters
- Different input and output formats
- Programming with elementary data types and arrays
- Initializing variables
- Program structure and the use of branches and loops

Hardware for the Sample Program

You can run the sample program on a SIMATIC S7-300 or SIMATIC S7-400, and you will need the following peripherals:

- One 16-channel input module
- One 16-channel output module

Debugging Functions

The program is constructed in such a way that you can perform a quick test using the switches on the input module and the displays on the output module. To perform a thorough test, use the SCL debugging functions (see Chapter 6).

You also have all other system functions provided by the STEP 7 Standard package.

2.2 Defining the Tasks

Summary

The measured data are to be recorded by an input module, sorted and processed. Assuming a required range for the measured data of 0 to 255, one byte is required for input.

The processing functions to be used are square root and square. The results are to be displayed on an output module which will require one word. Program control is to be performed via an input byte.

Recording Measured Data

A measured value set by means of the eight input switches is to be copied to the measured data array in the memory at precisely the point when a signal pulse is detected at the Enter switch (see Figure 2-1). The measured data array is to be organized as a cyclic buffer with a maximum of eight entries.

Processing Measured Data

When a signal is detected at the Sort switch, the values stored in the measured data array must be arranged in ascending order. After that, the square root and the square of each number must be calculated.

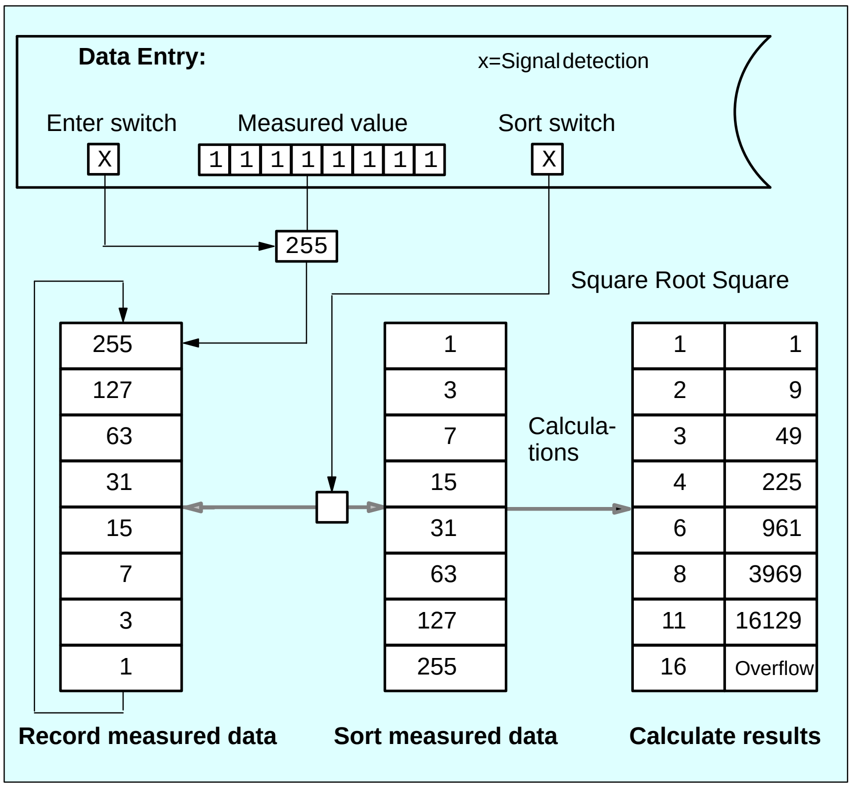


Figure 2-1 Recording and Processing Measured Data

Programmable Output

Since only one value at a time can be displayed, the following options must be available:

- Selection of an item from a list
- Selection of measured value, square root or square

The selection of an item from a list is to be implemented in such a way that a list item is addressed by means of the following switch setting:

- Three switches are used to set a code which is copied if a signal is detected at the fourth switch, the Code switch. From this, an address is calculated which is used to access the output data.
- The same address identifies three possible values; that is, the measured value, its square root and its square. To select one of these three options, two changeover switches are required (see Figure 2-2).

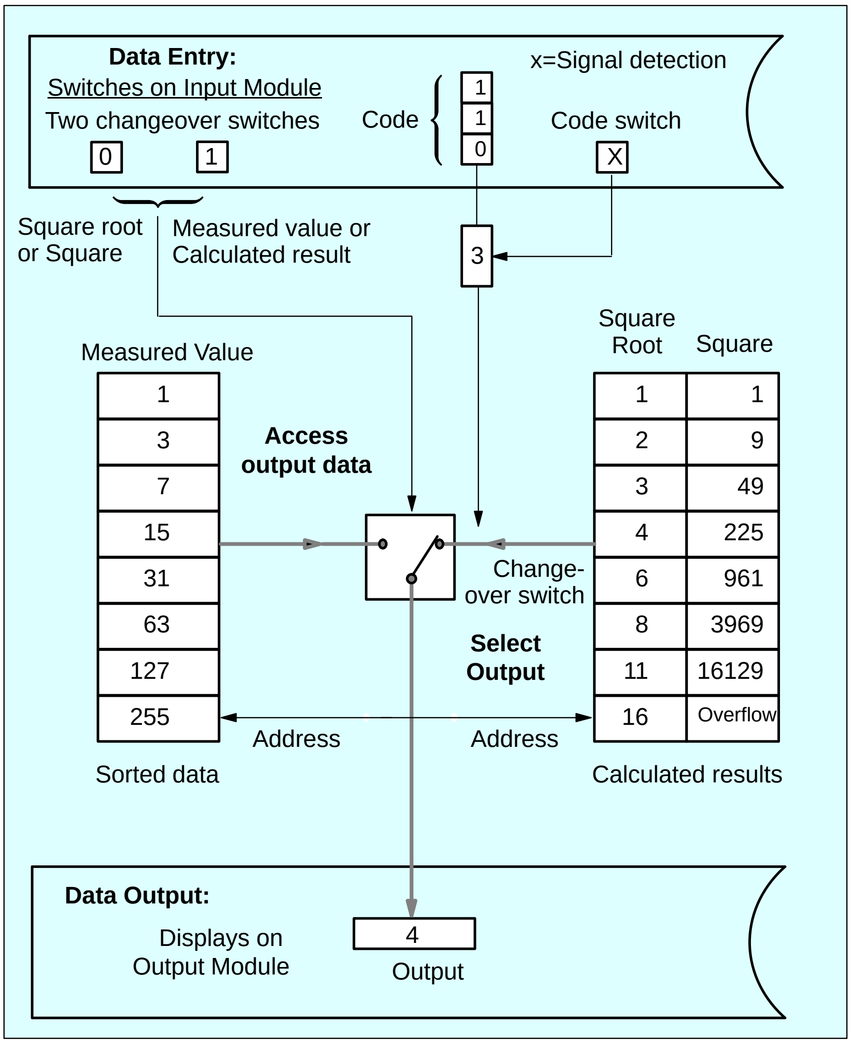


Figure 2-2 Programmable Output

2.3 Using SCL Blocks to Perform the Tasks

Overview

The task defined above is best performed by means of a **structured SCL program**. This involves using a modular design; that is, the program is subdivided into a number of blocks, each of which performs a specific subtask. In SCL, as with the other programming languages in STEP 7, you have a number of block types available. For more information on these types, see Chapters 1, 7 and 8.

Steps in the Task

You can adopt the following procedure:

1. Define the subtasks
2. Select and assign the available block types
3. Define the interfaces between the blocks
4. Define the input/output interface
5. Program the blocks

2.3.1 Defining the Subtasks

Overview

The subtasks are shown as boxes in Figure 2-3. The rectangular shaded areas represent the blocks. The order of the code blocks from left to right corresponds to the order in which they are called.

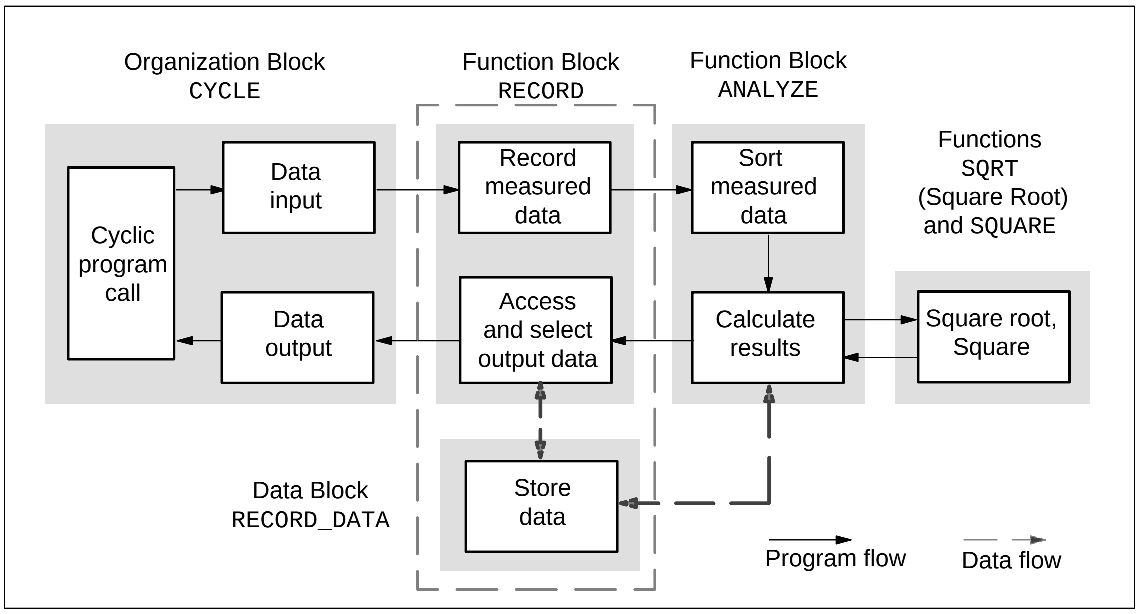


Figure 2-3 Creating Blocks Based on the Subtasks

2.3.2 Selecting and Assigning the Available Block Types

Overview

The individual blocks were selected according to the following criteria:

CYCLE

User programs can only be called by an OB. Since the measured data are to be received cyclically, an OB for a *cyclic operation call* (OB1) is required. Part of the processing – *Data Input* and *Data Output* – is programmed in the OB.

RECORD

The subtask *Record Measured Data* requires a block with a memory; that is, a function block (FB), since certain block-specific data (for example, the cyclic buffer) must be retained from one program cycle to the next. The location for the task *Store Data* (memory) is the instance data block RECORD_DATA.

The same FB can also perform the subtask *Access and Select Output Data,* since this is where the required data is kept.

ANALYZE

When selecting the type of block for performing the subtasks *Sort Measured Data* and *Calculate Results* you must remember that an output buffer has to be set up which contains the calculated results Square Root and Square for each measured value.

For that reason, this block can only be an FB. Since this FB is called by a higher-level FB it does not require its own DB. Its instance data can be stored in the instance data block of the calling FB.

SQRT (Square Root) and SQUARE

The type of block best suited to performing the subtasks *Calculate Square Root and Square* is a function (FC) since the the result can be returned as a function value. In addition, no data which has to be stored for more than one program cycle is required for the calculation.

The standard SCL function SQRT can be used for calculating the square root. A special function SQUARE is to be created for calculating the square and will also check that the value is within the permissible range.

2.3.3 Defining the Interfaces Between the Blocks

Overview

The interface between two blocks is defined by declaring the **formal parameters.** SCL offers the following possibilities:

- Input parameters: declared by means of VAR_INPUT
- Output parameters: declared by means of VAR_OUTPUT
- In/out parameters: declared by means of VAR_IN_OUT

When a block is called, input data is passed to it as **actual parameters**. After the program returns to the calling block, the output data is prepared for copying. An FC can transfer its result as a **function value** (for details, refer to Chapter 16).

`RECORD`

The OB `CYCLE` has no formal parameters itself. It calls the FB `RECORD` and passes to it the measured value and the control data for its formal parameters (Table 2-1):

Table 2-1 Formal Parameters of `RECORD`

Parameter Name	Data Type	Declaration Type	Description
`measval_in`	`INT`	`VAR_INPUT`	Measured value
`newval`	`BOOL`	`VAR_INPUT`	Switch for copying measured value to cyclic buffer
`resort`	`BOOL`	`VAR_INPUT`	Switch for sorting and analyzing measured data
`select function`	`BOOL`	`VAR_INPUT`	Two-way switch for selecting square root or square
`selection`	`WORD`	`VAR_INPUT`	Code for selecting output value
`newselection`	`BOOL`	`VAR_INPUT`	Switch for copying code
`result_out`	`DWORD`	`VAR_OUTPUT`	Output of calculated result
`measval_out`	`DWORD`	`VAR_OUTPUT`	Output of corresponding measured value

ANALYZE

The FB RECORD calls the FB ANALYZE. The information they share is the measured value array to be sorted. For that reason, this array is declared as an in/out parameter. A structured array is set up as an output parameter for the calculated results Square Root and Square. For details of formal parameters, see Table 2-2:

Table 2-2 Formal Parameters of ANALYZE

Parameter Name	Data Type	Declaration Type	Description
sortbuffer	ARRAY[..] OF REAL	VAR_IN_OUT	Measured value array, corresponds to cyclic buffer
calcbuffer	ARRAY[..] OF STRUCT	VAR_OUTPUT	Array for results: Structure having components ”Square Root” and ”Square” of type INT

SQRT and SQUARE

These functions are called by ANALYZE. They require an input value and return their results as a function value, see Table 2-3.

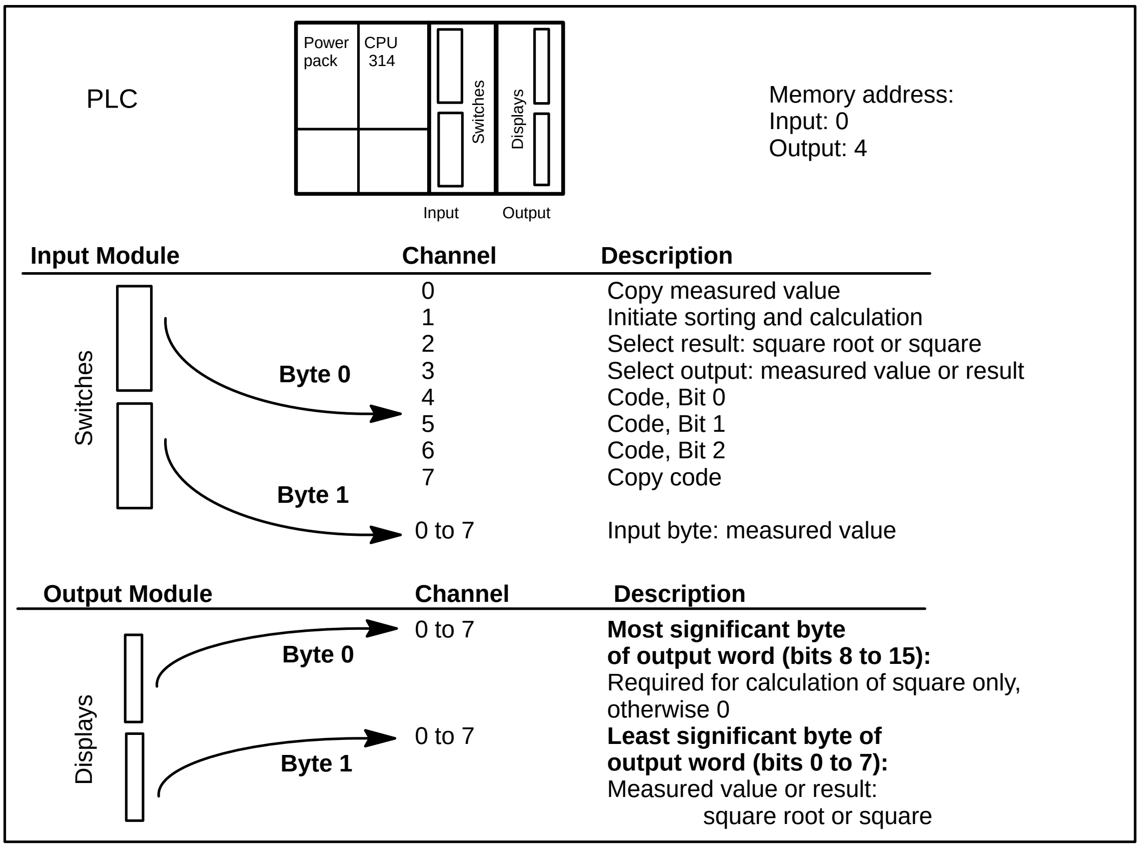
Table 2-3 Formal Parameters and Function Values of SQRT and SQUARE

Name	Data Type	Declaration Type	Description
value	REAL	VAR_INPUT	Input for SQRT
SQRT	REAL	Function value	Square root of input value
value	INT	VAR_INPUT	Input for SQUARE
SQUARE	INT	Function value	Square of input value

2.3.4 Defining the Input/Output Interface

Overview

Figure 2-4 shows the input/output interface. Note that in the case of input/output in bytes, the least significant byte is at the top and the most significant byte is at the bottom. In the case of input/output in words on the other hand, the opposite is true.



Input Module	Channel	Description
Byte 0	0	Copy measured value
	1	Initiate sorting and calculation
	2	Select result: square root or square
	3	Select output: measured value or result
	4	Code, Bit 0
	5	Code, Bit 1
	6	Code, Bit 2
	7	Copy code
Byte 1	0 to 7	Input byte: measured value

Output Module	Channel	Description
Byte 0	0 to 7	**Most significant byte of output word (bits 8 to 15):** Required for calculation of square only, otherwise 0
Byte 1	0 to 7	**Least significant byte of output word (bits 0 to 7):** Measured value or result: square root or square

Figure 2-4 Displays and Controls

2.3.5 Programming the Blocks

Programming Blocks

Once the interfaces have been defined, you can create each of the blocks separately from one another. This is best done from the top down; that is, in the order `CYCLE`, `RECORD`, `ANALYZE` and `SQUARE`. This is the order in which the blocks are described below.

When compiling the blocks, you must remember that a block must exist before you can use it; that is, call it from another block. This dictates that the order of the blocks in the SCL source file must be `SQUARE`, `ANALYZE`, `RECORD`, and `CYCLE` (for details, refer to Chapter 8).

Symbolic Programming

The comprehensibility of the program will be improved if you use **symbolic names** for module addresses and blocks. To do this, you must enter definitions in the symbol table as shown in Figure 2-5 (see Chapter 7). The names must conform to the naming conventions for either IDENTIFIERS or symbols (for example, ”Input 0.0”), see Appendix A.

Introductory Comment and Symbol Table

Figure 2-5 shows the introductory comment of the SCL source file and the symbolic names which are to be declared in the symbol table to permit its error–free compilation.

```
(*##################################################################################

SCL Program for Recording and Processing Measured Data:

–     A measured value whose signal is present on the input module is copied from
      input 0.0 (input switch)
–     Subsequent processing of the measured values can be controlled by various
      switches
–     All values are stored in the working section of the function block RECORD,
      the instance data block RECORD_DATA.

The program is programmed symbolically. In order for it to be compiled, details of
the assignment of the symbolic names to the module addresses and the blocks running
on the CPU must be specified. This requires the following symbol table:

Input             IB1      BYTE     // Measured value
Input 0.0         I0.0     BOOL     // Input switch for copying measured value
Sort switch       I0.1     BOOL     // Initiates sorting and calculation
Function switch   I0.2     BOOL     // Selects result: square root or square
Output switch     I0.3     BOOL     // Selects output: measured value or result
Code              IW0      WORD     // Code, relevant bits 12,13 and 14
Code switch       I0.7     BOOL     // Copies code
Output            QW4      INT      // Measured value or result: square root or square

RECORD            FB10     FB10     // Records measured values,
                                    // accesses and selects output
RECORD_DATA       DB10     FB10     // Instance data block for RECORD
ANALYZE           FB20     FB20     // Analyzes measured values, calculates results
SQUARE            FC41     FC41     // Function for calculating square
CYCLE             OB1      OB1      // Cyclic operation call and input/output

##################################################################################*)
```

Figure 2-5 Introductory Comment and Symbol Table

2.4 Creating the Organization Block CYCLE

Processing Sequence

An OB1 was chosen because it is called **cyclically** by the STEP 7 system. It performs the following tasks for the program:

- Calls and supplies the function block RECORD with input and control data.
- Copies the results data from the function block RECORD
- Outputs the data to the display

At the beginning of the declaration section is the 20-byte temporary data array “system data” (see also Chapter 8).

```
ORGANIZATION_BLOCK CYCLE
(********************************************************************************
      CYCLE corresponds to OB1; that is, it is called cyclically by the S7 system
      Part 1   :  Calls function block and transfers input data
      Part 2   :  Copies output data and outputs data with switch to output
*********************************************************************************)

VAR_TEMP
      system data    :  ARRAY[0..20] OF BYTE;   // Range for OB1
END_VAR
BEGIN
(*    Part 1   :  ***************************************************************)

RECORD.RECORD_DATA(
         measval_in     := WORD_TO_INT(Input),
         newval         := ”Input 0.0”, //Input switch as symbol
         resort         := Sort switch,
         selectfunction := Function switch,
         newselection   := Code switch,
         selection      := Code);
(*    Part 2   :  ***************************************************************)

IF Output switch THEN                            //Switch to output
      Output   := RECORD_DATA.result_out;        //Square root or Square
ELSE
      Output   := CREATE_DATA.measval_out;       //Measured value
END_IF;
END_ORGANIZATION_BLOCK
```

Figure 2-6 Organization Block CYCLE (OB1)

Data Type Conversion

The measured value is present at the input as data type BYTE. It has to be converted to data type INT. To do so, you must convert it from WORD to INT – prior conversion from BYTE to WORD is implicit in the compilation process (see Chapter 18). The output on the other hand requires no conversion, since it has been declared as data type INT in the symbol table, see Figure 2-5.

2.5 Creating the Function Block RECORD

Processing Sequence

The block type FB was chosen because certain data has to be retained from one program cycle to the next. This relates to the static variables which are declared in the declaration block "VAR, END_VAR" (see Table 2-4).

Static variables are local variables whose values are retained throughout the processing of every block. They are used to save values of a function block, and are stored in the instance data block.

```
FUNCTION_BLOCK RECORD

(*****************************************************************************
      Part 1 :    Records measured data
      Part 2 :    Initiates sorting and calculation
      Part 3 :    Analyzes code and prepares data for output
*****************************************************************************)
```

Figure 2-7 Header of Function Block RECORD

Static Variables

Table 2-4 Static Variables for RECORD

Name	Data Type	Declaration Type	Initialization Value	Description
measdata	ARRAY [..] OF INT	VAR	8(0)	Cyclic buffer for measured data
results-buffer	ARRAY [..] OF STRUCT	VAR	–	Array for structures with the components "square root" and "square" of the type INT
index	INT	VAR	0	Index for cyclic buffer identifying location for next measured value
prevval	BOOL	VAR	FALSE	Previous value for copying measured value using "newval"
prevsort	BOOL	VAR	FALSE	Previous value for sorting using "resort"
prev-selection	BOOL	VAR	FALSE	Previous value for copying code using "newselection"
address	INT	VAR	0	Address for output of measured value or result
analyzing_ block	ANALYZE, = FB 20	VAR	–	Local instance for the FB ANALYZE

Please note the initialization values which are assigned to the variables when the block is initialized (after being downloaded to the CPU). The local instance for the FB ANALYZE is also declared in the declaration block "VAR, END_VAR". This name is used subsequently for calling and accessing the output parameters. The global instance RECORD_DATA is used to store the data.

Declaration Section of RECORD

The declaration section in this block consists of the following components:

- Constants: declared between CONST and END_CONST
- Input parameters: declared between VAR_INPUT and END_VAR
- Output parameters: declared between VAR_OUTPUT and END_VAR
- Static variables: declared between VAR and END_VAR (this also includes declaration of the local instance for the block ANALYZE).

```
CONST
      LIMIT          := 7;
      COUNT          := LIMIT + 1;
END_CONST

VAR_INPUT
      measval_in     :  INT;  //    New measured value
      newval         :  BOOL; //    Copies measured value into cyclic buffer
      resort         :  BOOL; //    Sorts measured data
      selectfunction :  BOOL; //    Selects calculation function, Square Root/Square
      newselection   :  BOOL; //    Copies output address
      selection      :  WORD; //    Output address
END_VAR

VAR_OUTPUT
      result_out     :  INT;  //    Calculated value
      measval_out    :  INT;  //    Corresponding measured value
END_VAR

VAR
      measdata       :  ARRAY[0..LIMIT] OF INT := 8(0);
      resultsbuffer  :  ARRAY[0..LIMIT] OF
         STRUCT
          squareroot :  INT;
          square     :  INT;
         END_STRUCT;
      index          :  INT   :=    0;
      prevval        :  BOOL  :=    TRUE;
      prevsort       :  BOOL  :=    TRUE;
      prevselection  :  BOOL  :=    TRUE;
      address        :  INT   :=    0;    //Converted output address
      analyzing_block:  ANALYZE;          //Declaration of local instance
END_VAR
```

Figure 2-8   Declaration Section of the Function Block RECORD

Designing the Code Section

This is split into three sections:

Recording measured data

If the input parameter ”newval” is different from the ”prevval”, a new measured value is copied to the cyclic buffer.

Initiating sorting and calculation

Performed by calling the function block `ANALYZE` if the input parameter ”resort” is different from ”prevsort”.

Analyzing the code and preparing output data

The code is read word by word. According to SIMATIC conventions, this means that the upper group of switches (byte 0) contains the most significant eight bits of the input word and the lower group of switches (byte 1) the least significant. Figure 2-9 shows the location of the switches for setting the code.

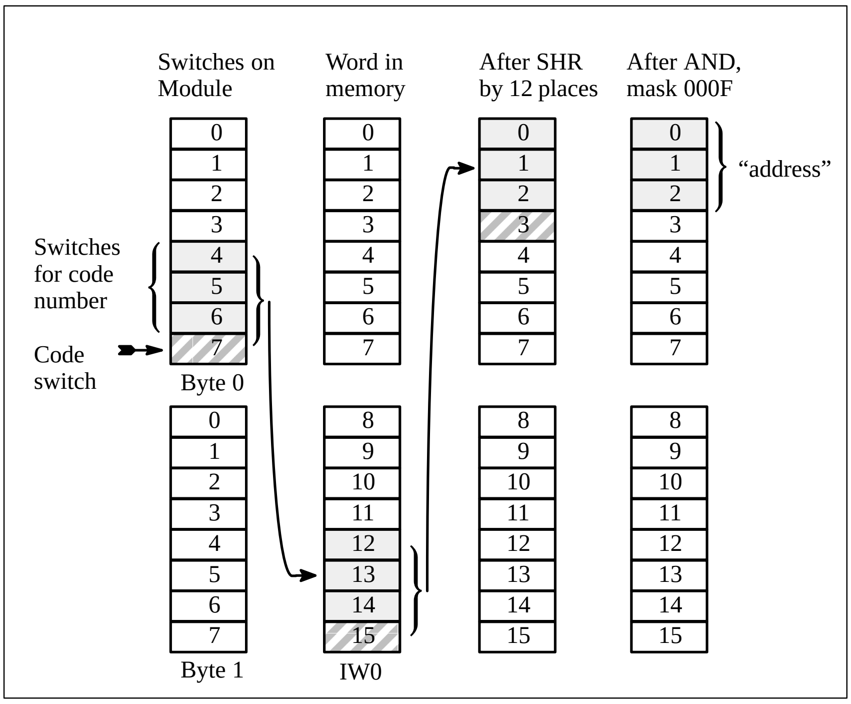


Figure 2-9 Analysis of the Code

Calculating the Address

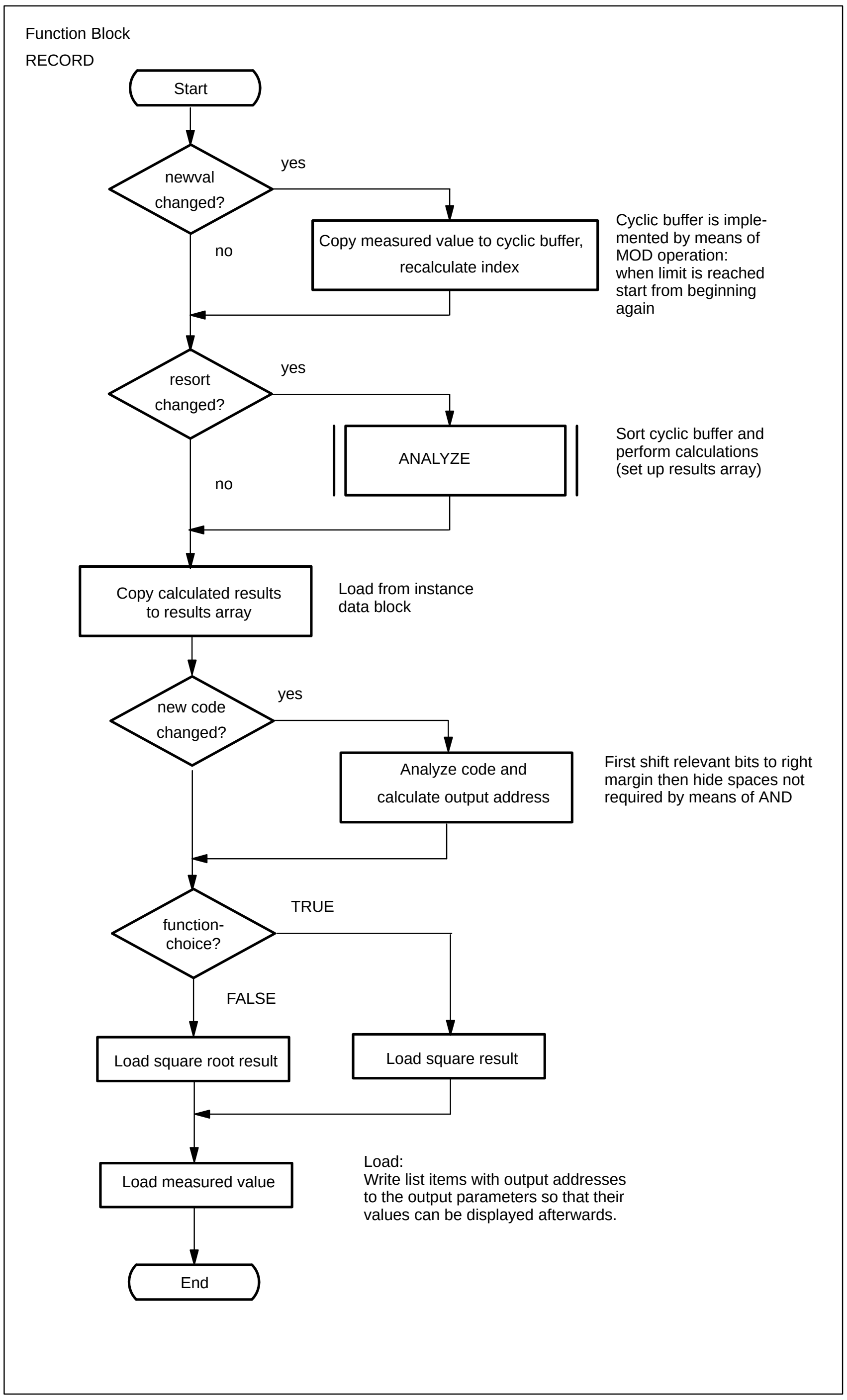
Figure 2-9 also shows how the address is calculated. The input word IW0 contains in bits 12 to 14 the code which is copied when a signal is detected at the code switch (bit 15). By shifting right using the standard function SHR and hiding the relevant bits using an AND mask, the “address” is calculated.

This address is used to write the array elements (calculated result and corresponding measured value) to the output parameters. Whether square root or square is output depends on “functionchoice”.

A signal at the code switch is detected by virtue of the fact that “newselection” is different from “prevselection”.

Flow Chart for RECORD

Figure 2-10 represents the algorithm in the form of a flow chart:



Function Block
RECORD
Start
newval changed?
yes
no
Copy measured value to cyclic buffer, recalculate index
Cyclic buffer is implemented by means of MOD operation: when limit is reached start from beginning again
resort changed?
yes
no
ANALYZE
Sort cyclic buffer and perform calculations (set up results array)
Copy calculated results to results array
Load from instance data block
new code changed?
yes
Analyze code and calculate output address
First shift relevant bits to right margin then hide spaces not required by means of AND
function-choice?
TRUE
FALSE
Load square root result
Load square result
Load measured value
Load:
Write list items with output addresses to the output parameters so that their values can be displayed afterwards.
End

Figure 2-10 Algorithm for Recording Measured Data

Code Section of RECORD

Figure 2-11 shows the SCL formulation of the flow chart shown in Figure 2-10; that is, the **code section** of the logic block.

```
BEGIN

(* Part 1 :    Records measured data ******************************************
               If ”newval” changes, the measured value is entered.
               A cyclic buffer for the measured data is implemented by means of
               the operation MOD.*)

IF newval <> prevval THEN
      index            :=    index MOD COUNT;
      measdata[index]   :=    measval_in;
      index            :=    index + 1;
END_IF;
prevval  :=    newval;

(* Part 2 :    Initiates sorting and calculation *******************************
               If ”resort” changes, sorting of cyclic buffer and performing of
               calculations on measured data is initiated. Results
               are stored in a new array ”calcbuffer”. *)

IF resort <> prevsort THEN
      index := 0; //Reset cyclic buffer index
      analyzing_block( sortbuffer := measdata); //Call ANALYZE
END_IF;
prevsort    :=    resort;

resultsbuffer  :=    analyzing_block.calcbuffer; //Square and Square Root

(* Part 3 :    Analyzes code and prepares data for output *********************
               If ”newselection” changes, the address code for accessing the
               array element for the output data is recalculated. The
               relevant bits of ”newselection” are hidden and converted into
               integers. Depending on the setting of the switch ”functionchoice”,
               either ”squareroot” or ”square” is prepared for output. *)

IF newselection <> prevselection THEN
      address  := WORD_TO_INT(SHR(IN := selection, N := 12) AND 16#0007);
END_IF;
prevselection  :=    newselection;

IF functionchoice THEN
      result_out  :=    resultsbuffer[address].square;
ELSE
      result_out  :=    resultsbuffer[address].squareroot;
END_IF;


measval_out :=    measdata[address]; //Display measured data

END_FUNCTION_BLOCK
```

Figure 2-11 Code Section of the Function Block RECORD

2.6 Creating the Function Block ANALYZE

Declaration Section of EVALUATE

The declaration section of this block consists of the following components:

- Constants: declared between CONST and END_CONST
- In/out parameters: declared between VAR_IN_OUT and END_VAR
- Output parameters: between VAR_OUTPUT and END_VAR
- Temporary variables: declared between VAR_TEMP and END_VAR

```
FUNCTION_BLOCK ANALYZE

(******************************************************************************
      Part 1 :    Sorts measured data in cyclic buffer
      Part 2 :    Initiates calculation of results
******************************************************************************)
```

Figure 2-12 Header of Function Block ANALYZE

```
CONST
      LIMIT                := 7;
END_CONST

VAR_IN_OUT
      sortbuffer           :  ARRAY[0..LIMIT] OF INT;
END_VAR

VAR_OUTPUT
      calcbuffer           :  ARRAY[0..LIMIT] OF
         STRUCT
            squareroot     :  INT;
            square         :  INT;
         END_STRUCT;
END_VAR

VAR_TEMP
      swap                 :  BOOL;
      index, aux           :  INT;
      valr, resultr        :  REAL;
END_VAR
```

Figure 2-13 Declaration Section of the Function Block ANALYZE

Procedure

The in/out parameter “sortbuffer” is linked to the cyclic buffer “measdata”; that is, the original contents of the buffer are overwritten by the sorted measured data.

The new array “calcbuffer” is created as an output parameter for the calculated results. Its elements are structured in such a way that they contain the square root and the square of each measured value.

Figure 2-14 shows you the relationship between the fields described.

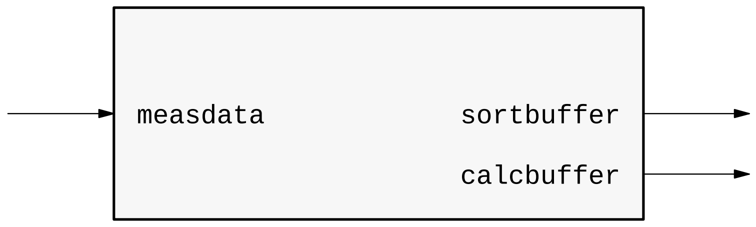


Figure 2-14 Interface of the FB ANALYZE

This interface shows the core element of data exchange for processing the measured data. The data is stored in the instance data block `RECORD_DATA`, since a local instance for the FB `ANALYZE` has been created in the calling FB `RECORD`.

Designing the Code Section

First of all, the measured data in the cyclic buffer is sorted and then the calculations performed.

- Sort algorithm method

The permanent exchange of values method is used for sorting the measured data buffer; that is, adjacent pairs of values are compared and their order reversed until the desired overall order is obtained. The buffer used is the in/out parameter ”sortbuffer”.

- Initiation of calculations

Once the sorting operation is complete, the program runs through a calculation loop in which the functions SQUARE and SQRT are called to obtain the square and square root respectively of the number in question. Their results are stored in the structured array ”calcbuffer”.

Flow Chart for ANALYZE

Figure 2-15 depicts the algorithm in the form of a flow chart:

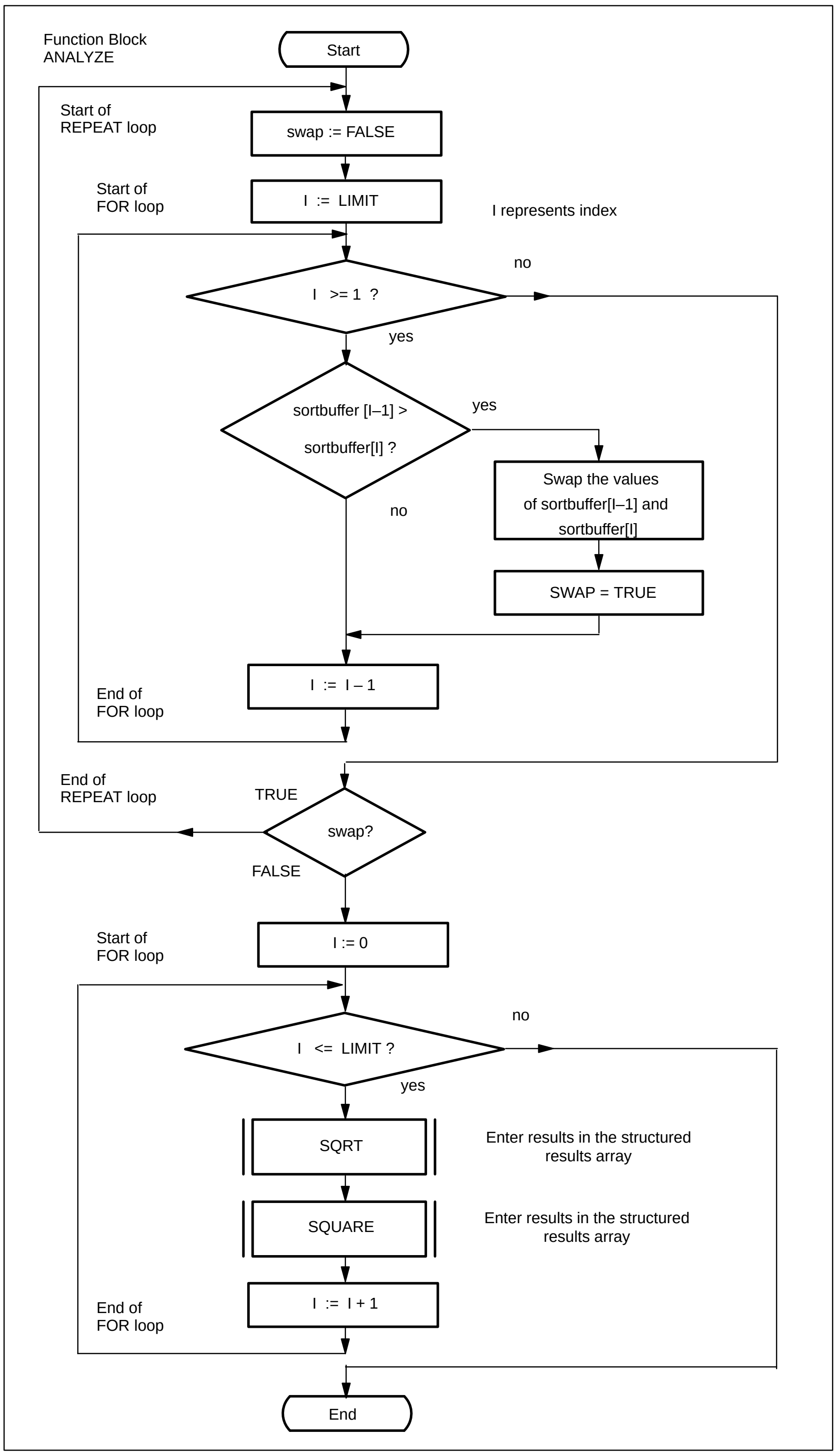


Figure 2-15 Algorithm for Analyzing the Measured Data

Code Section of ANALYZE

Figure 2-16 shows the SCL formulation of the flow chart shown in Figure 2-15; that is, the **code section** of the logic block.

```
BEGIN

(* Part 1   Sorting of data ******************************************************
            Swaps adjacent pairs of values using the ”bubble sort”
            method until the measured data buffer is correctly sorted. *)

REPEAT
      swap       := FALSE;

      FOR index   := LIMIT TO 1 BY –1 DO
         IF sortbuffer[index–1] > sortbuffer[index] THEN
            aux                    :=    sortbuffer[index];
            sortbuffer[index]      :=    sortbuffer[index–1];
            sortbuffer[index–1]    :=    aux;
            swap                   :=    TRUE;
         END_IF;
      END_FOR;

UNTIL NOT swap
END_REPEAT;


(* Part 2   Calculation of results  **********************************************
            Calculates square root using standard function SQRT and
            square using function SQUARE. *)

FOR index   := 0 TO LIMIT BY 1 DO
      valr     := INT_TO_REAL(sortbuffer[index]);
      resultr  :=       SQRT(valr);
      calcbuffer[index].squareroot  :=    REAL_TO_INT(resultr);
      calcbuffer[index].square      :=    SQUARE(sortbuffer[index]);
END_FOR;

END_FUNCTION_BLOCK
```

Figure 2-16 Code Section of the Function Block ANALYZE

2.7 Creating the Function SQUARE

Designing the Code Section

The program first checks whether the input value exceeds the limit at which the result would be outside the integer range. If it does, the maximum value for an integer is inserted. Otherwise, the square calculation is performed. The result is passed over as a function value.

```
FUNCTION SQUARE : INT
(*******************************************************************************
This function returns as its function value the square of the input value or, in
the event of overrun, the maximum value representable by an integer.
********************************************************************************)
VAR_INPUT
      value   :   INT;
END_VAR
BEGIN
IF value <= 181 THEN
      SQUARE   :=    val * val;     // Calculates function value
ELSE
      SQUARE   :=    32_767;        // Set to maximum value in the event of overrun
END_IF;
END_FUNCTION
```

Figure 2-17 The Function SQUARE

2.8 Debugging Data

Basic Requirements

To perform the test, you require an input module at address 0 and an output module at address 4 (see Figure).

Before performing the test, set all eight switches in the upper group to the left (“0”) and all eight switches in the lower group to the right (“1”).

Reload the blocks to the CPU, since the initial values of the variables must also be tested.

Stages of the Test

Now carry out the test as described in Table 2-5.

Table 2-5 Stages of the Test

Test	Action	Result
1	Set the code to ”111” (I0.4, I0.5 and I0.6) and copy that code by means of the code switch (I0.7).	All outputs on the output module (least significant byte) are activated and the displays light up.
2	Display the corresponding square root by setting the output switch (I0.3) to ”1”.	The displays on the output module indicate the binary number ”10000” (=16).
3	Display the corresponding square by setting the function switch (I0.2) to ”1”.	15 displays on the output module light up. This indicates a memory overflow since 255 x 255 gives too large a figure for the integer range.
4a	Reset the output switch (I0.3) to ”0” again.	The measured value is displayed again. All displays on the outputs of the least significant output byte are set.
4b	Set the number 3; that is, the binary number ”11” as the new measured value on the input module.	The output does not change at this stage.
5a	Observe the process of reading the measured value as follows: set the code to ”000” and copy it by means of the code switch (I0.7) so that you can subsequently observe the input of the data.	The output module shows 0; that is, none of the displays lights up.
5b	Switch over the input switch ”Input 0.0” (I0.0). This copies the value set in test stage 4.	The output module displays the measured value 3, binary ”11”.
6	Initiate sorting and calculation by switching over the sort switch (I0.1).	The output module again shows 0 since the sorting process has moved the measured value to a higher position in the array.
7	Display the measured value after sorting as follows: Set the code ”110” (I0.6 = 1, I0.5 = 1, I0.4 = 0 on IB0; corresponds to bit 14, bit 13 and bit 12 on IW0) and copy it by switching over the code switch.	The output module now shows the measured value ”11” again since it is the second highest value in the array.
8a	Display the corresponding results as follows: switching over the output switch (I0.3) displays the square of the measured value from stage 7.	The output value 9 (binary ”1001”) is displayed.
8b	Switch over the function switch (I0.2) to obtain the square root.	The output value 2 (binary ”10”) is displayed.

Supplementary Test

Tables 2-6 and 2-7 describe the switches on the input module and the examples for square and square root. These descriptions will enable you to define your own tests:

- Input is effected by means of switches. The top eight switches perform control functions, the bottom eight are used to set the measured value.
- Output is effected via displays. The top group displays the most significant output byte, the bottom group the least significant byte.

Table 2-6 Control Switches

Control Switches	Name	Description
Channel 0	Input switch	Switch over to copy measured value
Channel 1	Sort switch	Switch over to initiate sorting/calculation
Channel 2	Function switch	Set to left (“0”) for square root Set to right (“1”) for square
Channel 3	Output switch	Set to left (“0”) for measured value Set to right (“1”) for calculated result
Channel 4	Code	Output address Bit 0
Channel 5	Code	Output address Bit 1
Channel 6	Code	Output address Bit 2
Channel 7	Code switch	Switch over to copy code

Table 2-7 contains eight examples of measured values arranged in order.

You can enter the values in any order. Set the bit combination for each value and transfer this value by operating the input switch. Once all values have been entered, initiate sorting and calculation by operating the sort switch. After that, you can view the sorted data or the calculated results (square root or square).

Table 2-7 Sample Data for Square Root and Square

Measured Value	Square Root	Square
0000 0001 = 1	0, 0000 0001 = 1	0000 0000, 0000 0001 = 1
0000 0011 = 3	0, 0000 0010 = 2	0000 0000, 0000 1001 = 9
0000 0111 = 7	0, 0000 0011 = 3	0000 0000, 0011 0001 = 49
0000 1111 = 15	0, 0000 0100 = 4	0000 0000, 1110 0001 = 225
0001 1111 = 31	0, 0000 0110 = 6	0000 0011, 1100 0001 = 961
0011 1111 = 63	0, 0000 1000 = 8	0000 1111, 1000 0001 = 3969
0111 1111 = 127	0, 0000 1011 = 11	0011 1111, 0000 0001 = 16129
1111 1111 = 255	0, 0001 0000 = 16	0111 111, 1111 1111 = Overflow!

Part 2:
Operating and Debugging

3 Installing the SCL Software

Introduction

A menu-driven Setup program guides you through the process of installing the SCL software. The Setup program must be started using the standard procedure for installing software under Windows 95.

Installation Requirements

To install the SCL software, you require the following:

- A programming device or PC on which the STEP 7 Standard package has previously been installed and with
  - a 80486 processor (or higher) and
  - 16 Mbytes of RAM
- A color monitor, keyboard and mouse supported by Microsoft Windows 95
- A hard disk with 78 Mbytes of free storage space (10 Mbytes for user data, 60 Mbytes for swap-out files and 8 Mbytes for the SCL optional package)
- At least 1 Mbyte of free disk space on drive C: for the Setup program (the Setup files are erased once installation is completed)
- The Windows 95 operating system
- An MPI interface between the programming device/PC and the PLC consisting of:
  - Either a PC/MPI cable which is connected to the communications port of your device
  - Or an MPI module installed in your device. Some programming devices already have an MPI interface fitted.

Chapter Overview

Section	Description	Page
3.1	User Authorization	3-2
3.2	Installing / Uninstalling the SCL Software	3-4

3.1 User Authorization

Introduction

Product-specific user authorization is required for using the SCL software package. The software is protected in such a way that it can only be used if it detects the required authorization for the program or software package on the hard disk of the programming device/PC.

Authorization Diskette

In order to obtain user authorization, you require the read-protected authorization diskette. This contains the user authorization and the AUTHORS program, which are necessary for displaying, installing and uninstalling the authorization.

The number of possible user authorizations is specified by an authorization counter on the authorization diskette. For each authorization granted, the counter is reduced by one. Once it reaches zero, the disk in question can not be used for any further authorization.

For more details and rules governing the use of authorization, please refer to the User Manual **/231/**.

Caution

Read the notes in the README.WRI file on the authorization diskette. If you do not adhere to these guidelines, the authorization may be irretrievably lost.

Recording Authorization during First-Time Installation

You should complete the authorization procedure when the program prompts you to do so during first-time installation. Proceed as follows:

1. Insert the authorization diskette when prompted to do so.
2. Acknowledge the prompt.

The authorization details are transferred to a physical drive (in other words, your computer records that you have authorization).

Recording Authorization at a Later Date

If you start the SCL software when there is no record of authorization present, a message to that effect appears. To record authorization at any point after installation, start the program AUTHORS from the authorization diskette. This allows you to view, install or remove user authorization. The program is menu-driven.

Note

Always enter drive C: as the destination drive when installing authorization for SCL.

Removing Authorization

If you need to re-install authorization details, for example after re-formatting the drive on which the authorization details are recorded, you must first save those details elsewhere. To do this, you require the original authorization diskette.

Proceed as follows to retransfer the authorization details to the authorization diskette:

1. Insert the original authorization diskette in drive A: (3.6 inch).
2. Start the program AUTHORS.EXE from the authorization diskette.
3. Choose the menu command **Authorization** ▸ **Remove**.
4. In the dialog box which then appears, enter the drive on which the authorization details are recorded and confirm your entry. A list of the authorizations recorded on the specified drive appears.
5. Select the authorization you wish to remove and confirm your entry. If the operation is completed without errors, you will receive the following message:
   ”**Authorization for** <Name> **successfully removed from drive** <X:>.”
6. Acknowledge the message.

   The dialog box with the list of remaining authorizations on the drive then appears again. Close the dialog box if you do not wish to remove any other authorizations.

You can then use this disk to reinstall the authorization concerned at a later date.

If Your Hard Disk is Defective ...

If a fault occurs on your hard disk before you can save the authorization details, please contact your local SIEMENS representative.

3.2 Installing / Uninstalling the SCL Software

Summary

SCL includes a Setup program that automatically installs the software. Prompts which appear on the screen guide you step by step through the complete installation process.

Preparations

Before you can start installation, Windows 95 must be running and the STEP 7 Standard package must be also be loaded.

Starting the Installation Program

Proceed as follows:

1. In the Windows 95 Control Panel window, double-click on the Add/Remove Programs icon.
2. Select Install...
3. Insert the floppy disk (Disk 1) or the CD-ROM in the drive and then click Next. Windows 95 then automatically searches for the installation program SETUP.EXE.
4. Follow the instructions given by the installation program as and when they appear.

The program guides you through the installation process step by step. After each step you can choose to continue to the next step or go back to the previous one.

If a Version of SCL is Already Installed

If the installation program detects that there is already a version of SCL on the programmable logic controller, a message to that effect appears and you have the following options:

- Cancel installation (to uninstall the existing version of SCL from Windows 95 and then restart installation of the new version) or
- Continue with the installation process and allow the older version to be replaced by the new one.

It is always preferable to remove an earlier version before installing the new version. The disadvantage of simply overwriting the old version is that if you subsequently remove it, the uninstall procedure may not remove files which were components of the earlier version.

During the installation process, dialog boxes appear asking you questions or offering you a choice of options. Please read the notes below to enable you to respond more quickly and easily to the dialog boxes.

Uninstalling

Use the standard Windows 95 procedure for removing programs as follows:

1. In the Windows 95 Control Panel window, double-click on the Add/Remove Programs icon.
2. From the list of installed programs, select STEP 7 and then click the Add/Remove... button.
3. If dialog boxes appear asking you to confirm deletion of ”released files”, click the No button if in doubt.

Scope of Installation

All languages in the user interface and all examples require approximately 8 Mbytes of RAM.

Authorization

During installation, the program checks whether the appropriate authorization exists by looking to see if details are recorded on the hard disk. If no authorization details are found, a message appears indicating that the software can only be used with the appropriate authorization. If you wish you can record the authorization immediately or continue with the installation procedure and record authorization details at a later stage.

In the former case, you must insert the authorization diskette when prompted to do so and confirm the operation. Information about the authorization procedure is given in Section 3.1.

When Installation is Complete

If installation is successfully completed, this is indicated by a message to that effect on the screen.

Errors During Installation

The following errors will cause installation to be aborted:

- If an initialization error occurs immediately after the Setup program is started this most probably means that Windows 95 was not running when the program SETUP.EXE was started.
- Insufficient disk space – you require at least 8 Mbytes of free space on the hard disk.
- Faulty disk – if you discover that your floppy disk is faulty, please contact your Siemens representative.
- Operator errors: restart the installation process and follow the instructions carefully.

Using SCL 4

Introduction

This chapter introduces you to using SCL. It provides information about the SCL Editor user interface.

Chapter Overview

4.1 Starting the SCL Program

Starting from the Windows Interface

Once you have installed the SCL software on your programming device/PC, you can start SCL USING the Start button in the Taskbar of Windows 95 (entry under “SIMATIC / STEP 7”).

Starting from the SIMATIC Manager

The quickest way to start SCL is to position the mouse pointer on an SCL source file in the SIMATIC Manager and double-click on it. For more information, refer to the user manual /**231**/.

Figure 4-1 shows the SCL window after the program has been started.

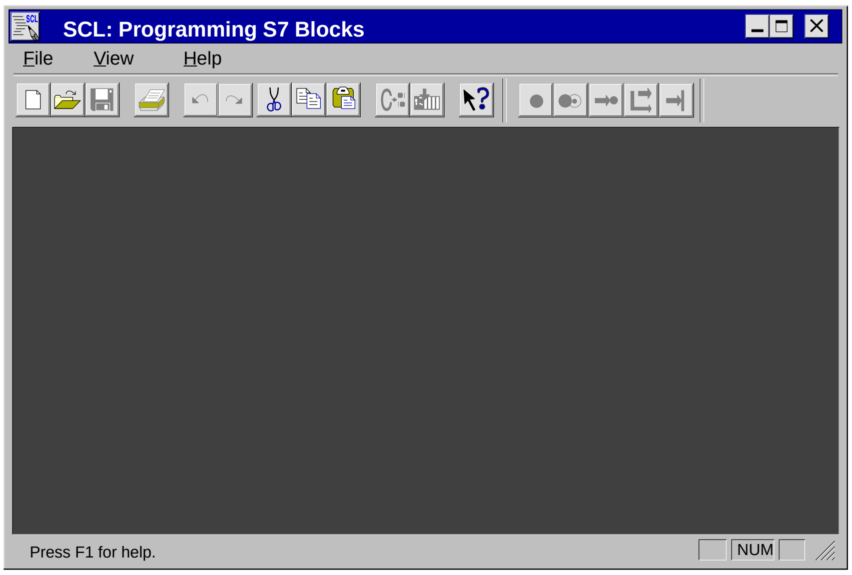


Figure 4-1 *SCL Window*

Note

Precise details of standard operations and options in Windows 95 are given in your Windows 95 documentation or the online Windows 95 Tutorial.

4.2 Customizing the User Interface

Overview

The SCL windows consist, as do other STEP 7 windows, of the following standard components (see Figure 4-2):

- **Title bar:**
  Contains the window title and window control buttons
- **Menu bar:**
  Shows all menus available in the window concerned
- **Toolbar:**
  Contains a series of buttons which provide shortcuts to frequently used commands
- **Working area:**
  Contains one or more windows in which you can edit program code or read compiler information or debugging data
- **Status bar**
  Displays the status of and other information relating to the active object

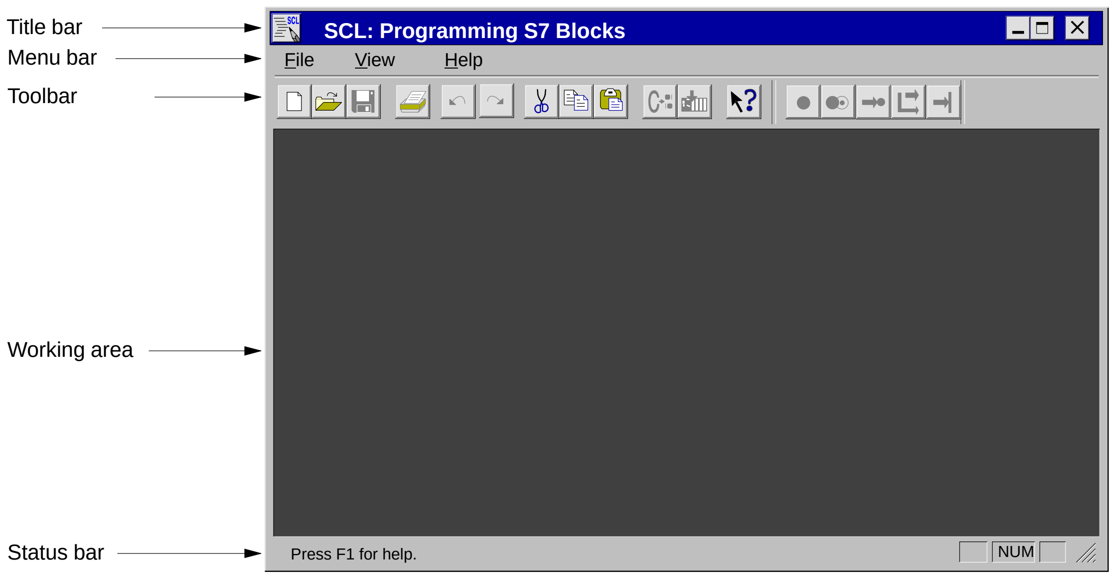


Figure 4-2 Components of the SCL Window

Modifying Components

The following components can be customized to your own personal specifications:

- Display of the toolbar
- Display of the breakpoint bar
- Display of the status bar

Customizing the Tool Bar

You can choose to display or hide the toolbar by selecting or deselecting the menu command **View ▸ Toolbar**. When the function is activated, a check mark appears next to the command.

Customizing the Breakpoint Bar

You can choose to display or hide the breakpoint bar by selecting or deselecting the menu command **View ▸ Breakpoint Bar**. When the function is activated, a check mark appears next to the command.

Customizing the Status Bar

Similarly, you can choose to display or hide the status bar by selecting or deselecting the menu command **View ▸ Status Bar**. When the function is activated, a check mark appears next to the command.

Customizing the Development Environment

The Editor and the Compiler allow you to make certain settings which will make your job easier.

- Settings when creating blocks
- Editor settings
- Compiler settings

Creating Blocks

You can, for example, decide whether existing blocks are overwritten or not when compiling. To do this, select the menu command **Options ▸ Customize** and click the“Create Blocks” tab in the “Customize” dialog box. The options are described in detail in Section 5.5.

Customizing the Compiler

You can also adapt the compilation process to your own requirements. A detailed description of the options is given in Section 5.5.

Select the menu command **Options ▸ Customize** and click the “Compiler” tab in the “Customize” dialog box.

Customizing the Editor

You can specify the tab indent width, save before compiling, and display line numbers settings and other options. To do this, select the menu command **Options ▸ Customize** and click the “Editor” tab in the “Customize” dialog box.

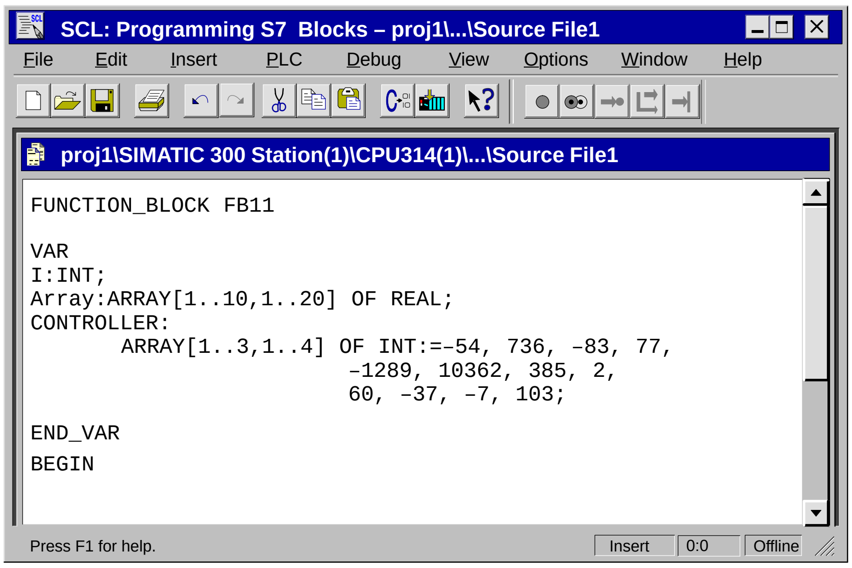
4.3 Working with the SCL Editor

Overview

The SCL source code consists primarily of running text. When entering text, the SCL Editor provides assistance in the form of word processing functions adapted specifically to SCL requirements.

The Editor Window

The source object for your user program is typed in the workspace using the keyboard. You can open more than one window for the same source object or open a number of windows each containing different source objects. The window arrangement can be controlled by means of the commands in the Window menu.


SCL: Programming S7 Blocks – proj1\...\Source File1
File
Edit
Insert
PLC
Debug
View
Options
Window
Help
proj1\SIMATIC 300 Station(1)\CPU314(1)\...\Source File1
Press F1 for help.
Insert
0:0
Offline

```
FUNCTION_BLOCK FB11

VAR
I:INT;
Array:ARRAY[1..10,1..20] OF REAL;
CONTROLLER:
       ARRAY[1..3,1..4] OF INT:=-54, 736, -83, 77,
                         -1289, 10362, 385, 2,
                         60, -37, -7, 103;

END_VAR
BEGIN
```

Figure 4-3 SCL Editor Window

Selecting Text

In SCL you can select text by positioning the cursor at the beginning of the section you wish to select and then pressing and holding the left mouse button while dragging the highlight so that it covers the whole of the desired section of text.

You can also:

- Select the complete source code text by selecting the menu command **Edit ▸ Select All.**
- Select a word by double-clicking on it.
- Select a whole row by clicking on it three times.

Search and Replace

The menu command **Edit ▸ Find/Replace** opens a dialog box in which you can enter a character string you wish to find or replace with different text.

Inserting Templates

Inserting templates enables you to program more efficiently and makes it easier to avoid syntax errors. In SCL you can

- Insert templates for blocks by selecting the menu command **Insert ▸ Block Template**.
- Insert templates for control structures by selecting the menu command **Insert ▸ Control Structure**.

Cut, Copy, Paste and Delete

Text can be cut, copied, pasted and deleted in the normal way. The relevant commands are to be found in the **Edit** menu.

In most cases, you can move and copy objects by “dragging and dropping” with the mouse.

GO TO

With the menu command **Edit ▸ Go To ...**, a dialog box is opened in which you enter the number of the row at which you want to position the insert cursor, and then confirm with “OK”.

Undo, Restore

With the menu command **Edit ▸ Undo**, you can reverse an action, for example, undelete a row. The menu command **Edit ▸ Restore** enables you to restore an action that was undone.

Programming with SCL 5

Introduction

When programming with SCL, you must perform a series of individual tasks which make up the processing sequence, as described in the following.

Chapter Overview

5.1 Creating User Programs Using SCL

Basic Requirements for Writing Programs

Before you start to write a program using SCL, you should first perform the following operations:

1. Set up a project using the SIMATIC Manager.
2. Use the SIMATIC Manager to assign every CPU a communications address in the network.
3. Configure and initialize the CPU module and signal modules.
4. Create a symbol table if you wish to use symbolic addresses for CPU memory areas or block names.

Creating the Symbol Table

If you want to use symbolic addresses for CPU memory areas or block names in your SCL program, you must create a symbol table. SCL will access this table during compilation. Use STEP 7 to create the symbol table and enter values.

You can open the symbol table with the SIMATIC Manager or directly with SCL using the menu command **Options** ▸ **Symbol Table**.

Moreover, you can also import and continue editing other symbol tables which may have been created as text files with any text editor (for more information, consult the manual /**231**/).

Processing Sequence

To create a user program using SCL, you must first create an SCL source file. In this source file you can write one or more program blocks (OBs, FBs, FCs, DBs and UDTs) and then compile them by means of a batch process. The compilation process places the source file blocks into the user program folder (<AP-off>, see Figure 5-1) of the same S7 program in which the source file is stored.

The SCL source file can be created and edited using the integrated Editor or a standard text editor. Source files created using a standard text editor must be imported into the project using the SIMATIC Manager. Once imported, they can be opened, edited and compiled.

5.2 Creating and Opening an SCL Source File

Overview

Source files created in SCL can be integrated in the structure of an S7 program as follows:

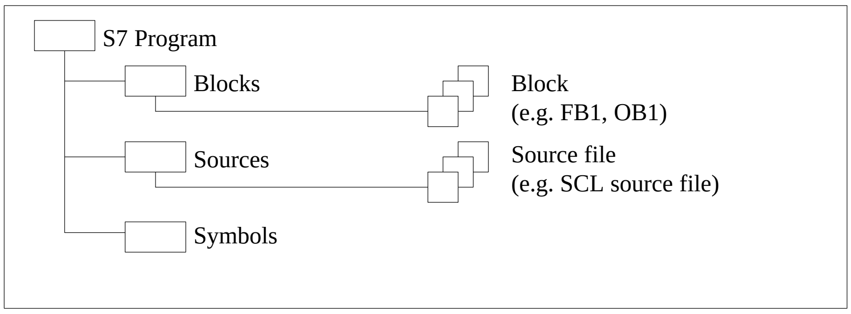


Figure 5-1 Structure of an S7 Program in the SIMATIC Manager

Creating an SCL Source File

To create a new source file for SCL, proceed as follows:

1. Select the menu command **File ▸ New** or click the New File button on the Tool Bar.
2. In the New dialog box, select the desired project and the corresponding S7 program.
3. Open the source folder and select **Insert ▸ S7 Software ▸ Source File** in the menu bar.
4. Mark the source file and select **Edit ▸ Object Properties** in the menu bar. Enter the name of the source object in the “General” tabbed page. The name can be up to 24 characters long. Source file names are case-sensitive.
5. Double-click the source file. A blank window opens in which you can edit the SCL source file.

Opening an SCL Source File

You can open an existing source file previously created and saved in SCL in order to edit or compile it. Proceed as follows:

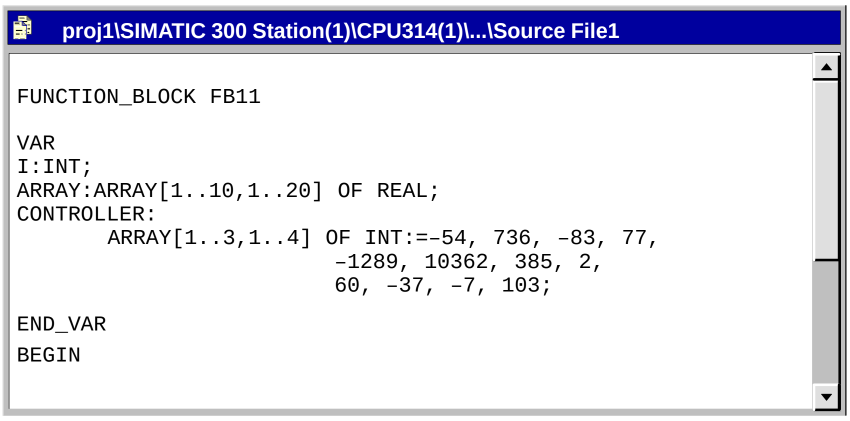
1. Select the menu command **File ▸ Open** or click the Open File button on the Tool Bar.
2. In the Open dialog box, select the desired project and the corresponding S7 program.
3. Make sure that the filter “SCL source file” is activated and select the source file container (SO).
4. The dialog box then displays all SCL source files for the selected S7 program. Select the desired object and confirm your selection by selecting OK or double-clicking the name of the source file.

Source files created using a standard text editor can be opened in the same way once they have been imported into the project by the SIMATIC Manager.

5.3 Entering Declarations, Statements and Comments

Overview

An SCL source file must be written according to strictly defined syntactical rules. Those rules are an integral component of the language definition. For their detailed description, refer to the Appendices.



```
FUNCTION_BLOCK FB11

VAR
I:INT;
ARRAY:ARRAY[1..10,1..20] OF REAL;
CONTROLLER:
       ARRAY[1..3,1..4] OF INT:=–54, 736, –83, 77,
                         –1289, 10362, 385, 2,
                         60, –37, –7, 103;

END_VAR
BEGIN
```

Figure 5-2 SCL Source File

Rules

The following conventions must be observed when writing source files:

- An SCL source file can contain any number of logic blocks (FBs, FCs, OBs), data blocks (DBs) and user-defined data types (UDTs). Each type of block has a standardized structure (see Chapter 8).
- The use of upper or lower case letters is only of significance for symbolic identifiers (for example, variable names and string literals).
- Called blocks must precede blocks in which they are called.
- User-defined data types (UDTs) must precede the blocks in which they are used.
- Global data blocks must precede all blocks which access them.
- Observe the layout and syntax rules described in the *Language Description* section of this manual.

5.4 Saving and Printing an SCL Source File

Saving an SCL Source File

The term ”saving” always refers to saving the source files. Blocks are generated in SCL when the source file is compiled and automatically stored in the appropriate program directory.

There are a number of options available when saving an SCL source file. These are as follows:

- Select the menu command **File ▸ Save** or click the Save button on the tool bar.

  The copy of the SCL source file on disk is updated.

- If you wish to create a copy of the active SCL source file, select the menu command **File ▸ Save As**. The Save As dialog box appears in which you can enter a name and path for the duplicate file.

- If you select the menu command **File ▸ Close** without having saved changes to the SCL source file, you are asked whether you wish to save the changes or not or cancel the **Save** command.

  Instead of using the menu command **File ▸ Close**, you can click the Close button on the title bar.

  Even if you exit SCL by selecting the menu command **File ▸ Exit** when there are open source files in which the current changes have not been saved, the dialog box asking whether or not you wish to save the changes appears for each open file.

Printing a Source Object

You can print out the blocks, declarations and statements in your SCL source file at any time. You must first have installed and set up the printer from the Windows 95 Control Panel. To print a source file, proceed as follows:

- Click the Print button on the tool bar or choose the menu command **File ▸ Print**. A dialog box appears in which you can select various printing options such as sections to be printed and number of copies.

  Choose OK to confirm your selections and print the document.

Page Setup

The menu command **File ▸ Page Setup** allows you to adjust page layout.

Creating Headers and Footers

You can make the settings for headers and footers in your printed documents in the SIMATIC Manager using the **File ▸ Headers and Footers** menu command.

Print Preview

The menu command **File ▸ Print Preview** allows you to obtain a preview of how your page layout settings will look when printed out. You can not alter the settings in this view.

5.5 The Compilation Process

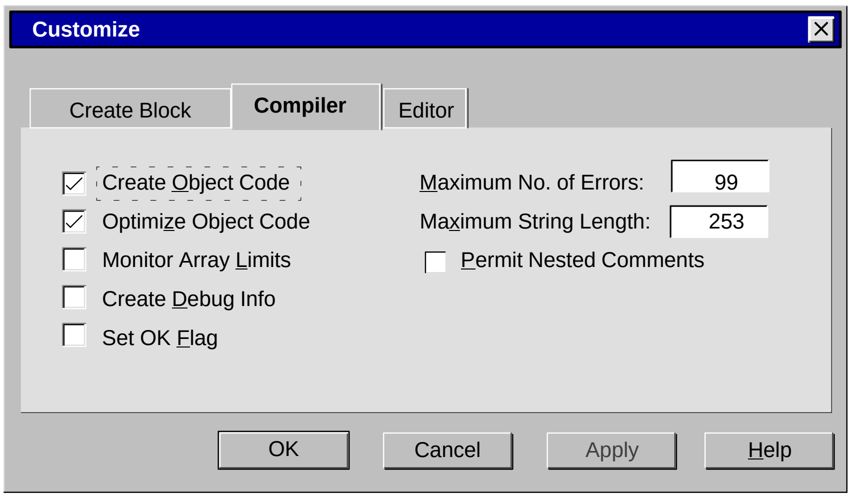
Overview

Before you run can run or test your program, you have to compile it. Initiating the compilation process (see below) activates the Compiler. The Compiler has the following characteristics:

- The Compiler works in batch mode, i.e. it treats an SCL source file as a complete unit. Partial compilation (e.g. line by line) is not possible.
- The Compiler checks the syntax of the SCL source file and subsequently indicates all errors found during the compilation process.
- It generates blocks containing debugging information if the SCL source file is error-free and the appropriate option is set. The Debug Info option has to be set individually for every program that you wish to test with SCL at high language level.
- It generates an instance data block for every function block call if it does not already exist.

Compiler Options

You can adapt the compilation process to suit your specific requirements. To do so, choose the menu command **Options** ▸ **Customize** and click the Compiler tab in the Customize dialog box. The various options can be selected or deselected by clicking on them with the mouse.



[✓] Create Object Code
[✓] Optimize Object Code
[ ] Monitor Array Limits
[ ] Create Debug Info
[ ] Set OK Flag

Maximum No. of Errors: 99
Maximum String Length: 253
[ ] Permit Nested Comments

Figure 5-3 *“Customize” Dialog, “Compiler” Tab*

Options

The various options available are:

- **Maximum No. of Errors**: The compiler will abort compilation of the SCL source file if the number of errors reaches the maximum number specified.
- **Create Object Code**: Create code capable of being run on a PLC? Yes/No
- **Optimize Object Code**: Create shorter code. If the Create Debug Info option is selected, complete optimization is not possible.
- **Monitor Array Limits**: Run-time check on whether array indices are within the permissible range according to the declaration for the array concerned. If an array index is outside the permissible range, the OK flag is set to FALSE (provided the OK flag option is activated).
- **Create Debug Info**: Generate debugging information: Yes/No. Debugging information is required for debugging with the high-level language debugger.
- **Set OK Flag**: Every run-time error sets the OK variable to FALSE.
- **Maximum String Length**: Reduce the standard length of the “STRING” data type. The default standard length is 254 characters. In order to optimize the use of your CPU resources, you can reduce the standard length.
- **Permit Nested Comments**: Several comments can be nested in each other in the SCL source file.

Creating a Block

In the “Create Block” tabbed page, you can make settings that influence the compilation:

- You can decide whether or not existing blocks are overwritten during compilation.
- You can have reference data generated automatically during compilation of a source file. If you activate this option, the compilation takes longer.
- Activate the “Include System Attribute S7_server” if the block is relevant for message or connection configuration. This option also extends the time required for compilation.

Starting Compilation

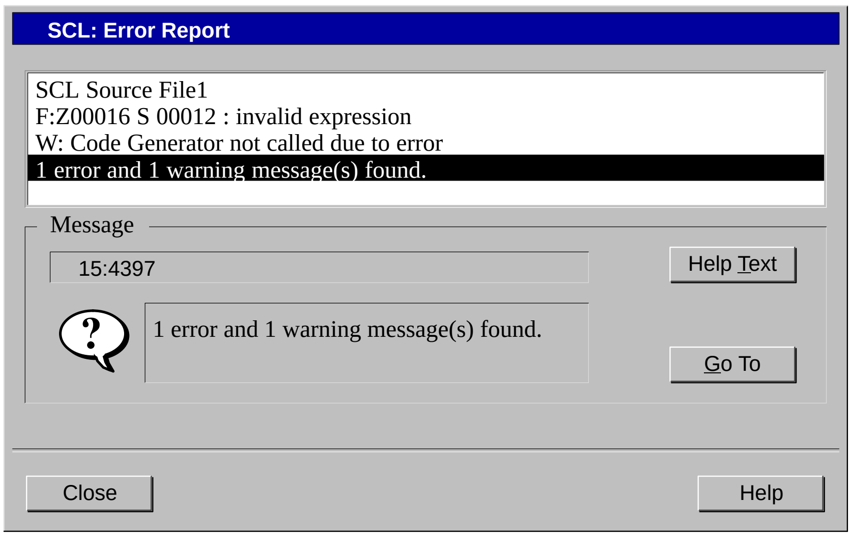
There are two ways in which the compilation can be initiated.

- Select then menu command **File ▸ Compile**, or
- Click the Compile button on the tool bar.

To make sure that you always compile the latest version of your SCL source file, it is advisable to select the menu command **Options ▸ Customize** and to select the option Save Before Compiling in the Editor tabbed page. The menu command **File ▸ Compile** will then automatically save the SCL source file first before compiling it.

After Compiling a Source File

When the compilation process has been completed, you will either receive a message to the effect that compilation has been successfully completed without errors or a window similar to that in Figure 5-4 will appear listing all errors and warning messages encountered.



SCL Source File1
F:Z00016 S 00012 : invalid expression
W: Code Generator not called due to error
1 error and 1 warning message(s) found.

Figure 5-4 Window Listing Errors and Warning Messages

Finding the Causes of Error and Warning Messages

Every message is listed together with the relevant line and column position as well as a brief description. You can obtain a detailed explanation of the error/ warning message by selecting it and then clicking the Help button.

Double-clicking a message will move the cursor to the corresponding point in the SCL source file.

These two functions allow you to locate and correct errors and warning messages quickly and simply.

5.6 Transferring the Compiled User Program to the PLC

Overview

When an SCL source file is compiled, the blocks in the source file are generated and saved in the "Blocks" folder of the S7 program. In SCL you can subsequently download only those blocks from the programming device to the CPU.

Use the SIMATIC Manager if you wish to transfer other blocks of the S7 program to the PLC.

Basic Requirements

In order to be able to load the application program into the PLC, the following requirements must be satisfied:

- There must be a connection between the programming device and the PLC.
- The blocks to be downloaded must have been successfully compiled without errors.

Resetting the CPU Memory

The function Clear/Reset can be used to completely clear an application from a CPU online. Please note that at the same time it resets the CPU, shuts down all existing connections with the CPU and, if a memory card is fitted, copies the contents of the memory card to the internal load memory. To perform the function, proceed as follows:

1. Select the menu command **PLC** ▸ **Operating Mode** and set the CPU to STOP mode.
2. Select the menu command **PLC** ▸ **Clear/Reset**.
3. Confirm the action in the dialog box which then appears.

Downloading to the PLC

It is preferable to transfer the blocks with the CPU in STOP mode since errors can occur if a program is overwritten when the CPU is in RUN mode. To transfer the blocks, proceed as follows:

1. Select the menu command **PLC** ▸ **Download**.
2. If the block is already present in the CPU RAM, confirm when prompted whether the block is to be overwritten.

5.7 Creating a Compilation Control File

Overview

You can automate compilation of a series of SCL source files by creating a compilation control file.

Compilation Control File

You can create a compilation control file for your STEP 7 project. In it, you enter the names of SCL source files in the project which are to be compiled in a batch processing run.

Creating the File

You create the file as follows:

- When you create or open a file with the command New or Open you must activate the Compilation Control File filter.
- The file is then given the special extension ”.inp”.
- When you compile this file, the files specified in it are compiled one after the other.

Compiling

When the files are compiled, the blocks created are stored in the “Blocks” folder of the S7 program.

Debugging Programs 6

Introduction

The SCL debugging functions allow you to check the execution of a program on the CPU and to locate any errors that it might contain.

Syntax errors are indicated by the compiler. Run time errors occurring during the execution of the program are also indicated, in this case, by system alarms. You can locate logical programming errors using the debugging functions.

Getting Further Information

You can obtain more detailed information on debugging with SCL from the online help. The online help system can provide you with answers to specific problems while you are working with SCL.

Chapter Overview

6.1 Overview

High Language Level

You can use the SCL debugging functions to debug user programs programmed in SCL at high-language level. This debugging method allows you to:

- Identify programming errors
- Observe and monitor the effects of a user program on the processing sequence in the CPU.

Basic Requirements

Before you can debug an SCL program you must first complete the following operations:

1. The program must have been successfully compiled without errors using the compilation options ”Create Object Code” and ”Create Debug Information”. These options are set in the Compiler tabbed page in the **Options ▸ Customize** dialog box.
2. You must have established an online connection between the programming device/PC and the CPU.
3. You must also have loaded the program into the CPU. You can do this by means of the menu command **PLC ▸ Download**.

SCL Debugging Functions

Table 6-1 lists the basic debugging functions available in SCL and gives a brief description of their features.

Table 6-1 Summary of Debugging Functions

Function	Features
Monitor continuously (S7-300/400 CPUs)	Output names and current values of variables of a monitoring range
Breakpoints active (only S7-400 CPUs)	Set, delete and edit breakpoints; single-step debugging
Monitor/modify variables	Monitor/specify current values of shared data
Create reference data	Create an overview of the user data
STEP 7 StandardPackage debugging functions	Check/change the CPU mode

Note

Testing while the system is running could result in serious damage or injury in the event of malfunctions or program errors! Always make sure that no dangerous situations can occur before activating debugging functions.

6.2 “Monitor Continuously” Debugging Function

Summary

Using the “Monitor continuously” function, you can debug a group of statements. This group of statements is also known as the monitoring range.

During the test, the values of the variables and the parameters of this range are displayed in chronological order and updated cyclically. If the monitoring range is in a program section that is executed in every cycle, the values of the variables cannot normally be displayed for consecutive cycles.

Values that have changed in the current run are displayed in black. Values that have not changed are displayed in light gray.

The range of statements that can be tested depends on the performance of the connected CPUs. After compilation, different SCL statements in the source code produce different numbers of statements in machine code, so that the length of the monitoring range is variable and is determined and indicated by the SCL debugger when you select the first statement of the required monitoring range.

Debug Mode

When debugging in the “Monitor Continuously” mode, the current values of the data in the monitoring range are queried and displayed. The values are queried while the debugger is running through the monitoring range. This usually extends the length of the cycle times.

To allow you to influence the extent to which the cycle time is extended, SCL provides two different test environments.

- **“Process” Test Environment**

In the “Process” test environment, the SCL debugger restricts the maximum monitoring range so that the cycle times during testing do not exceed the real run times of the process or only very little.

- **“Laboratory” Test Environment**

In the ”Laboratory” test environment, the monitoring range is only limited by the performance of the connected CPU. The cycle times can however be longer than in the real process so that the maximum monitoring range is greater than in the ”Process” test environment.

Using “Monitor Continuously” Mode

Proceed as follows to execute the “Monitor Continuously” function:

1. Make sure that the basic requirements listed in Section 6.1 are met.
2. Select the window containing the source file of the program to be tested.
3. If you want to change the default test environment (process), select the menu option **Debug ▸ Test Environment ▸ Laboratory.**
4. Position the cursor in the line of the source text containing the first statement of the range to be tested.
5. Select the menu option **Debug ▸ Monitor Continuously.**

   Result: The largest possible monitoring range is determined and indicated by a gray bar at the left edge of the window. The window is split and the names and current values of the variables in the monitoring range are displayed line by line in the right-hand half of the window.
6. Select the menu option **Debug ▸ “Monitor Continuously”** again to interrupt debugging and continue later.
7. Select the menu option **Debug ▸ “Finish Debugging”** to stop debugging.

6.3 “Breakpoints Active” Debugging Function

Overview

When debugging in the ”Breakpoints Active” mode, the program is run through in single steps. You can execute the program statement by statement and see how the values of the variables change.

After setting breakpoints, you can allow the program to be executed as far as a breakpoint and then monitor step-by-step starting at that breakpoint.

Breakpoints

You can define breakpoints at any point in the code section of the source file.

The breakpoints are sent to the programmable controller and activated only after you select the menu command **Debug ▸ Breakpoints Active**. The program is then executed until the first breakpoint is reached.

The maximum possible number of active breakpoints is CPU-dependent.

- CPU 416: maximum of 4 active breakpoints possible
- CPU 414: maximum of 2 active breakpoints possible
- CPU 314: no active breakpoints possible

Single-Step Functions

Once the debugging function **Breakpoints Active** has been activated, the following functions can be performed:

- **Next Statement**

  Continues with next statement – for output of variable values
- **Continue**

  Continues to next activated breakpoint
- **To Cursor**

  Continues to a point currently selected in the source file.

Note

Please make sure that the maximum number of active breakpoints is not exceeded when you use the menu commands **Next Statement** or **To Cursor** since these functions automatically set and activate a breakpoint.

Using “Breakpoints Active”

First make sure that the requirements listed in Section 6.1 are met before you start debugging. You can now test out your program step by step with the “Breakpoints Active” function. The description below and the flow chart in Figure 6-1 explain the procedure.

1. Select and open the window containing the source file for the block you wish to test.
2. Set the breakpoints by positioning the cursor at the desired point in the program source file and selecting the menu command **Debug ▸ Set Breakpoint**. The breakpoints are displayed at the left edge of the window as a red circle.
3. Start single-step execution by selecting the menu command **Debug ▸ Breakpoints Active**.

   Result: The window is split vertically into two halves and the program looks for the next breakpoint. When it finds it, the CPU is switched to the HOLD mode and the point reached is marked with a yellow arrow.
4. You can now select one of the following functions:
   – Select the menu command **Debug ▸ Next Statement** (4a)
   Result: the CPU briefly switches to RUN. When the next statement is reached, it stops again and the values of the variables processed for the last statement are displayed in the left half of the window.

   – Select the menu command **Debug ▸ Continue** (4b)
   Result: the CPU switches to RUN. When the next active breakpoint is reached, it stops again and the breakpoint is displayed at the left edge of the window. To view the values of the variables, select the menu command **Debug ▸ Next Statement** again.

   – Select the menu command **Debug ▸ To Cursor** (4c)
   A breakpoint is automatically set and activated at the currently selected position. The CPU switches to RUN. When it reaches the selected point, it stops again and the breakpoint is displayed. To view the values of the variables, select the menu command **Debug ▸ Next Statement**.
5. Return to step 2 if you wish to continue testing using changed breakpoints. At step 2 you can set new breakpoints or delete existing ones.
6. Select the menu command **Debug ▸Breakpoints Active** again to deactivate the test loop.
7. If you do not want to test any other statements in the source file, quit debugging with the menu command **Debug ▸Finish Debugging**.

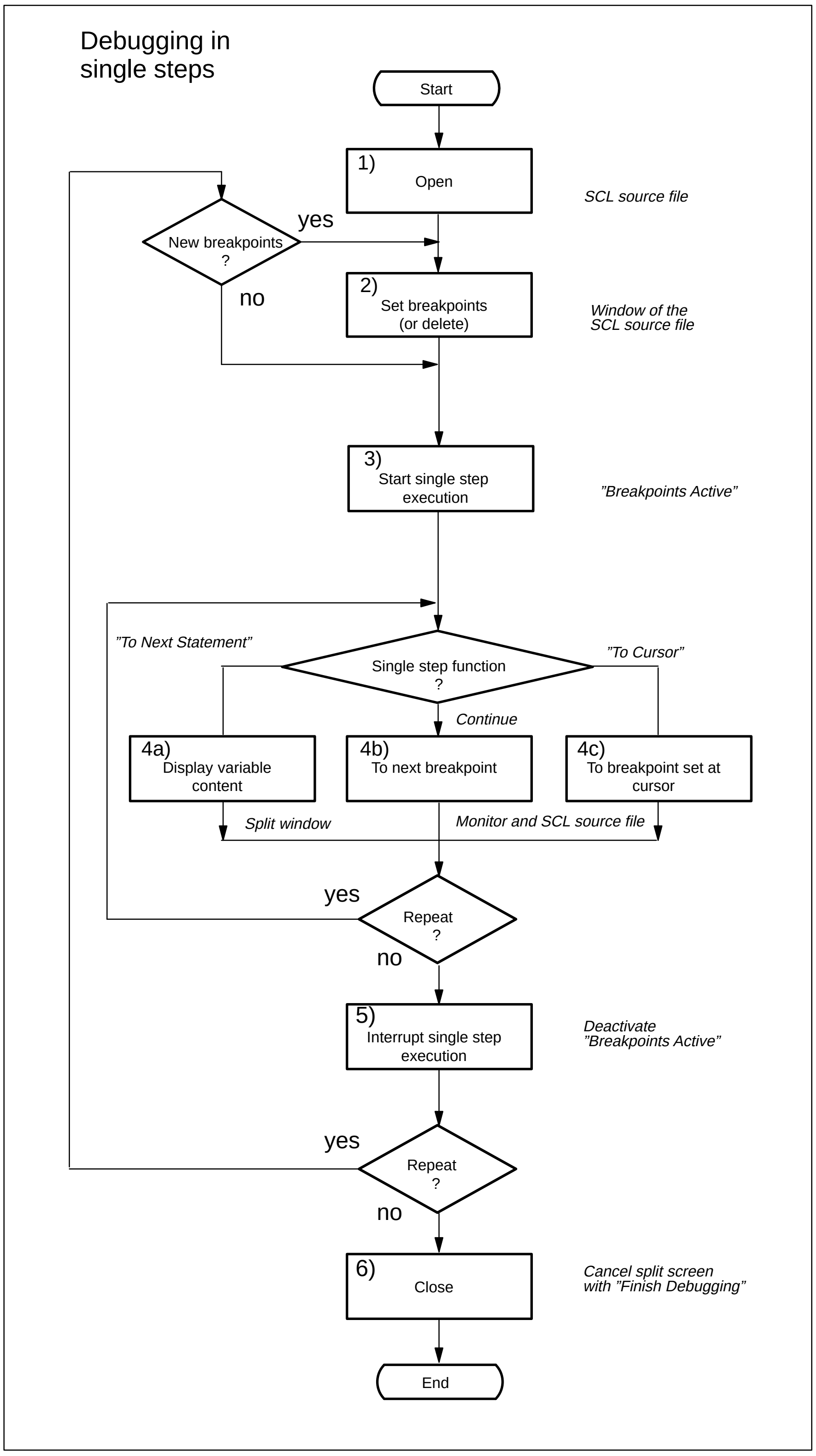

Debugging in single steps
Start
1) Open
SCL source file
New breakpoints ?
yes
no
2) Set breakpoints (or delete)
Window of the SCL source file
3) Start single step execution
"Breakpoints Active"
"To Next Statement"
Single step function ?
"To Cursor"
Continue
4a) Display variable content
4b) To next breakpoint
4c) To breakpoint set at cursor
Split window
Monitor and SCL source file
yes
Repeat ?
no
5) Interrupt single step execution
Deactivate "Breakpoints Active"
yes
Repeat ?
no
6) Close
Cancel split screen with "Finish Debugging"
End

Figure 6-1 *Algorithm for Debugging*

6.4 “Monitoring/Modifying Variables” Debugging Function

Overview

When you use the “Monitor/Modify Variables” function, you can do the following:

- Display (monitor) the current values of shared data from your user program.
- Assign fixed values to the variables of a user program (modify).

Monitoring and Modifying Variables

With the **PLC ▸ Monitor/Modify Variables** menu command, you can do the following:

- Set trigger points and conditions.
- Specify values for the variables of a user program.

In both cases, you must create a variable table, in which you specify the required variables. If you want to modify the variables, you also enter the required values.

The debugging and test functions are described in detail in the STEP 7 User Manual **/231/**.

6.5 “Reference Data” Debugging Function

Overview

You can create and evaluate reference data to help you debug and modify your user program.

Reference data include the following: program structure, cross reference list, assignment list, list of unused addresses, and list of addresses without symbols.

You can use reference data for the following:

- To provide an overview of the entire user program
- As a basis for modifications and tests
- To supplement program documentation

Creating Reference Data

You can create reference data in the folloiwng ways:

- With the **Options** ▸ **Reference Data** menu command, you can create, update and display reference data.
- With the **Options** ▸ **Customize** menu command, you can have the reference data generated automatically when the source file is compiled. If you want the reference data compiled automatically, enter a check mark beside “Create Reference Data” in the “Create Block” tabbed page. Automatic creation of the reference data extends the time required for compilation.

The debugging and test functions are described in detail in the STEP 7 User Manual **/231/**.

6.6 Using the STEP 7 Debugging Functions

STL Editor

Blocks that have been compiled by SCL can be opened in STL and then tested with the STL (Statement List) Editor.

Querying and Changing Operating Mode

Select the menu command **PLC ▸ Operating Mode** to check or change the current operating mode of the CPU.

Viewing CPU Status

The menu command **PLC ▸ Module Information** opens a dialog box in which you can

- determine the reason for the CPU switching to STOP mode by reading the diagnostic buffer
- view the contents of the CPU stacks. The break stack in particular is an important source of information for locating faults
- view the CPU technical data
- view the CPU date and time
- determine the CPU cycle time
- find out which blocks are in the CPU
- view information about CPU communication

For the above functions, the CPU must be online

Part 3: Language Description

General Introduction to Basic SCL Terms 7

Introduction

This chapter explains the language functions provided by SCL and how to use them. Please note that only the basic concepts and essential definitions are dealt with at this point and that more detailed explanations are given in the subsequent chapters.

Chapter Overview

7.1 Language Definition Aids

SCL Language Definition

The language definition is based on syntax diagrams. These provide you with a good overview of the syntactical (in other words grammatical) structure of SCL. Appendix B of this manual contains a collection of all the diagrams with the language elements.

What is a Syntax Diagram?

A syntax diagram is a graphical representation of the structure of the language. The structure is created using a hierarchical sequence of rules. Each rule can be based on preceding rules.

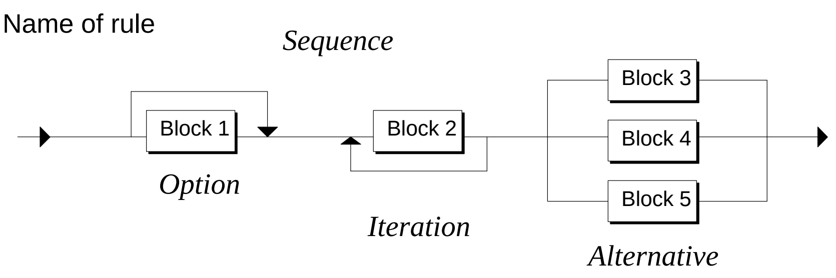


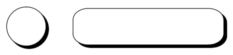
Figure 7-1 Syntax Diagram

The syntax diagram is read from right to left. The following rule structures must be adhered to:

- Sequence: sequence of blocks
- Option: skippable branch
- Iteration: repetition of branches
- Alternative: multiple branch

What Types of Block are there?

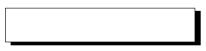
A block is a fundamental element or an element that itself is made up of blocks. The symbols used to represent the various types of block are illustrated below:



Basic element that requires no further explanation.

These are printable characters or special characters, keywords and predefined identifiers.

The details of these blocks are copied unchanged.



Complex element that is described by other syntax diagrams.

What Does Free Format Mean?

When writing source code, the programmer must observe not only the **syntax rules** but also **lexical rules**.

The lexical and syntax rules are described in detail in Appendices B and C. Free format means that you can insert formatting characters such as spaces, tabs and page breaks as well as comments between the rule blocks.

Lexical Rules

In the case of lexical rules such as the example in Figure 7-2, there is **no** freedom of format. When you apply a lexical rule, you must adopt the specifications exactly as set out.

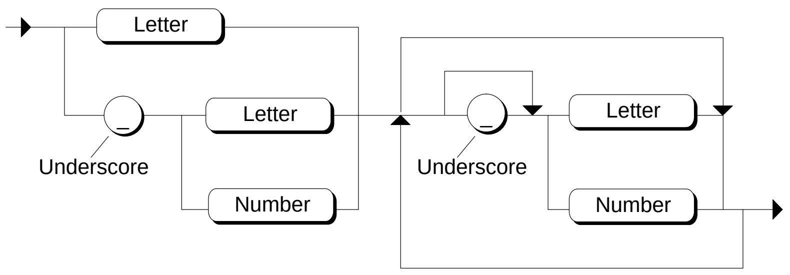


Figure 7-2 Example of a **Lexical** Rule

The following are examples of adherence to the above rule:

```
C_CONTROLLER3
_A_FIELD
_100_3_3_10
```

The following are examples of non-adherence to the above rule:

```
1_1AB
RR__20
*#AB
```

Syntax Rules

The syntax rules (e.g. Figure 7-3) allow free format.

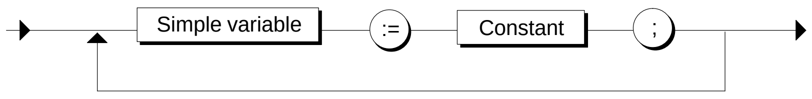


Figure 7-3 Example of a **Syntax** Rule

The following are examples of adherence to the above rule:

```
VARIABLE_1  := 100;     SWITCH:=FALSE;
            VARIABLE_2  := 3.2;
```

7.2 The SCL Character Set

Letters and Numeric Characters

SCL uses the following subset of the ASCII character set:

- The upper and lower case letters A to Z
- The Arabic numbers 0 to 9
- The space character (ASCII value 32) and all control characters (ASCII 0-31) including the end of line character (ASCII 13)

Other Characters

The following characters have a specific meaning in SCL:

+	–	*	/	=	<	>	[	]	(	)
.	,	:	;	$	#	”	’	{	}	

Other Information

Refer to Appendix A of this manual for a detailed list of all permitted characters and how those characters are interpreted in SCL.

7.3 Reserved Words

Explanation

Reserved words are keywords that you can only use for a specific purpose. No distinction is made between upper and lowercase letters.

Keywords

AND	END_STRUCT
ANY	END_VAR
ARRAY	END_WHILE
BEGIN	EXIT
BLOCK_DB	FOR
BLOCK_FB	FUNCTION
BLOCK_FC	FUNCTION_BLOCK
BLOCK_SDB	GOTO
BLOCK_SFB	IF
BLOCK_SFC	INT
BOOL	LABEL
BY	MOD
BYTE	NIL
	NOT
CASE	OF
CHAR	OR
CONST	ORGANIZATION_BLOCK
CONTINUE	POINTER
COUNTER	REAL
DATA_BLOCK	REPEAT
DATE	RETURN
DATE_AND_TIME	S5TIME
DINT	STRING
DIV	STRUCT
DO	THEN
DT	TIME
DWORD	TIMER
ELSE	TIME_OF_DAY
ELSIF	TO
END_CASE	TOD
END_CONST	TYPE
END_DATA_BLOCK	VAR
END_FOR	VAR_TEMP

Keywords, Continuation

END_FUNCTION
END_FUNCTION_BLOCK
END_IF
END_LABEL
END_TYPE
END_ORGANIZATION_BLOCK
END_REPEAT
VOID

UNTIL
VAR_INPUT
VAR_IN_OUT
VAR_OUTPUT
WHILE
WORD
XOR

Other Reserved Words

EN
ENO
OK
TRUE
FALSE
Names of the standard functions

7.4 Identifiers in SCL

Definition

An identifier is a name that you assign to an SCL language object, in other words to a constant, a variable, a function or a block.

Rules

Identifiers can be made up of letters or numbers in any order but the first character must be either a letter or the underscore character. Both upper and lowercase letters are permitted. As with keywords, identifiers are not case-sensitive (Anna and AnNa are, for example, identical).

An identifier can be formally represented by the following syntax diagram:

IDENTIFIER

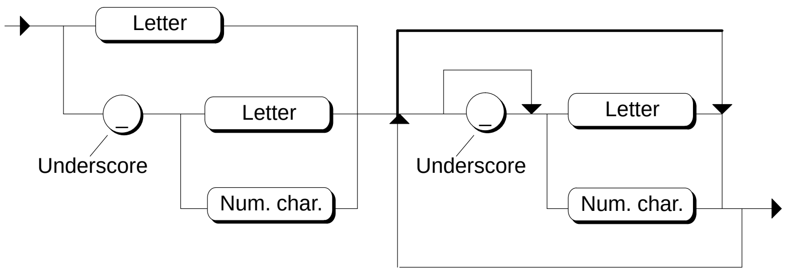


Figure 7-4 Syntax of an Identifier

Please note the following points:

- When choosing names for identifiers, it is advisable to use unambiguous and self-explanatory names which add to the comprehensibility of the program.
- You should check that the name is not already in use by a standard identifier or a keyword (for example, as in Table 7-1).
- The maximum length of an identifier is 24 characters.
- Symbolic names for blocks (that is, other identifiers as in Table 7-1) must be defined in the STEP 7 symbol table (for details refer to **/231/**).

Examples

The following names are examples of valid identifiers:

`x`	`y12`	`Sum`	`Temperature`
`Name`	`Surface`	`Controller`	`Table`

The following names are **not** valid identifiers for the reasons specified:

`4th`	The first character must be a letter or an underscore character
`Array`	ARRAY is a keyword and is not permitted.
`S Value`	Spaces are characters and not allowed.

7.5 Standard Identifiers

Definition

In SCL, a number of identifiers are predefined and are therefore called *standard identifiers*. These standard identifiers are as follows:

- the block keywords and
- the address identifiers for addressing memory areas of the CPU.

Block Keywords

These standard identifiers are used for absolute addressing of blocks.

Table 7-1 is sorted in the order of the SIMATIC mnemonics and the corresponding international IEC mnemonics are also shown.

Table 7-1 Block Keywords

Mnemonic (SIMATIC)	Mnemonic (IEC)	Identifies
DBx	DBx	Data Block
FBx	FBx	Function Block
FCx	FCx	Function
OBx	OBx	Organization Block
SDBx	SDBx	System Data Block
SFCx	SFCx	System Function
SFBx	SFBx	System Function Block
Tx	Tx	Timer
UDTx	UDTx	Global or User-Defined Data Type
Zx	Cx	Counter

x = number between 0 and 65533
DBO = reserved		

STANDARD IDENTIFIER

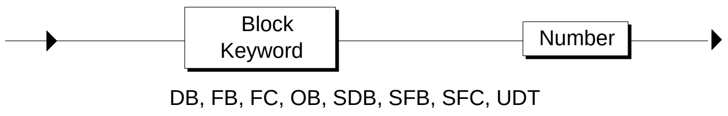


Figure 7-5 Syntax of a Standard Identifier

The following are examples of valid identifiers:

```
FB10
DB100
T141
```

Address Identifiers

You can address memory areas of a CPU at any point in your program using their address identifiers.

The following table is sorted in order of the SIMATIC mnemonics, the corresponding international IEC mnemonic is shown in the second column.

Mnemonic (SIMATIC)	Mnemonic (IEC)	Addresses	Data Type
Ax,y	Qx,y	Output (via process image)	Bit
ABx	QBx	Output (via process image)	Byte
ADx	QDx	Output (via process image)	Double word
AWx	QWx	Output (via process image)	Word
AXx.y	QXx.y	Output (via process image)	Bit
Dx.y [1]	Dx.y [1]	Data block	Bit
DBx [1]	DBx [1]	Data block	Byte
DDx [1]	DDx [1]	Data block	Double word
DWx [1]	DWx [1]	Data block	Word
DXx	DXx	Data block	Bit
Ex.y	Ix.y	Input (via process image)	Bit
EBx	IBx	Input (via process image)	Byte
EDx	IDx	Input (via process image)	Double word
EWx	IWx	Input (via process image)	Word
EXx.y	IXx.y	Input (via process image)	Bit
Mx.y	Mx.y	Bit memory	Bit
MBx	MBx	Bit memory	Byte
MDx	MDx	Bit memory	Double word
MWx	MWx	Bit memory	Word
MXx.y	MXx.y	Bit memory	Bit
PABx	PQBx	Output (I/O direct)	Byte
PADx	PQDx	Output (I/O direct)	Double word
PAWx	PQWx	Output (I/O direct)	Word
PEBx	PIBx	Input (I/O direct)	Byte
PEDx	PIDx	Input (I/O direct)	Double word
PEWx	PIWx	Input (I/O direct)	Word

x = number between 0 and 64535 (absolute address)
y = number between 0 and 7 (bit number)			

The following are examples of valid address identifiers:

```
I1.0          MW10          PQW5          DB20.DW3
```

KEIN MERKER *These address identifiers only apply if the data block is specified*

7.6 Numbers

Summary

There are several ways in which you can write numbers in SCL. A number can have a plus or minus sign, a decimal point, and an exponent. The following rules apply to all numbers:

- A number must not contain commas or spaces.
- To create a visual separation between numbers, the underscore character (_) can be used.
- The number can be preceded if required by a plus (+) or minus (–) sign. If the number is not preceded by a sign, it is assumed to be positive.
- Numbers must not be outside certain maximum and minimum limits.

Integers

An integer contains neither a decimal point nor an exponent. This means that an integer is simply a sequence of digits that can be preceded by a plus or minus sign. Two integer types are implemented in SCL, `INT` und `DINT`, each of which has a different permissible range of values (see Chapter 9).

Examples of valid integers:

```
0            1            +1           –1
743          –5280        600_00       –32_211
```

The following integers are **illegal** for the reasons stated in each case:

123,456	Integers must not contain commas.
36.	Integers must not contain a decimal point.
10 20 30	Integers must not contain spaces.

Integers as Binary, Octal or Hexadecimal Numbers

In SCL, you can also represent integers in different numerical systems. To do this, the number is preceded by a keyword for the numerical system. The keyword 2# stands for the binary system, 8# for the octal system and 16# for the hexadecimal system.

Valid integers for decimal 15:

```
             2#1111       8#17         16#F
```

Real Numbers

A real number must contain either a decimal point or an exponent (or both). A decimal point must be between two numeric characters. This means that a real number cannot start or end with a decimal point.

Examples of valid real numbers:

```
0.0          1.0          –0.2         827.602
50000.0      –0.000743    12.3         –315.0066
```

The following real numbers are **illegal**:

`1.`	There must be a number on both sides of the decimal point.
`1,000.0`	Real numbers must not contain commas.
`.3333`	There must be a number on both sides of the decimal point.

A real number can include an exponent in order to specify the position of the decimal point. If the number contains no decimal point, it is assumed that it is to the right of the number. The exponent itself must be either a positive or a negative integer. Base 10 is represented by the letter E.

The value 3 x 10 ^{10} can be correctly represented in SCL by the following real numbers:

```
3.0E+10   3.0E10         3e+10       3E10
0.3E+11   0.3e11         30.0E+9     30e9
```

The following real numbers are **illegal**:

`3.E+10`	There must be a number on both sides of the decimal point.
`8e2.3`	The exponent must be an integer.
`.333e–3`	There must be a number on both sides of the decimal point.
`30 E10`	Spaces are not allowed.

Character Strings

A character string is a sequence of characters (in other words letters, numbers, or special characters) set in quotation marks. Both upper and lowercase letters can be used.

Examples of permissible character strings:

```
'RED'     '7500 Karlsruhe'          '270–32–3456'
'DM19.95' 'The correct answer is:'
```

You can enter special formatting characters, the single quotation mark (’) or a $ character by using the alignment symbol $.

```
Source Code              After Compilation

'SIGNAL$'RED'            SIGNAL'RED'
'50.0$$'                 50.0$
'VALUE$P'                VALUE Page break
'REG-$L'                 REG Line feed
'CONTROLLER$R            CONTROLLER Carriage return
'STEP$T'                 STEP        Tab
```

To enter non-printing characters, type in the substitute representation in hexadecimal code in the form $*hh*, where *hh* stands for the hexadecimal value of the ASCII character.

To enter comments in a character string that are not intended to be printed out or displayed, you use the characters $> and $< to enclose the comments.

7.7 Data Types

Summary

A declaration of a variable must always specify what data type that variable is. The data type determines the permissible range of values for the variable and the operations that it can be used to perform.

The data type determines

- the type and interpretation of a data element,
- the permissible range of values for a data element,
- the permissible number of operations that can be performed by an address of a variable, and
- the format of the data of that data type.

Types of Data Type

The following types of data type are distinguished:

Table 7-2 Elementary Data Types

Data Type	Explanation
Elementary	Standard type provided by SCL
Complex	Can be created by combining elementary data types
User-defined	Defined by the user for specific applications and assigned a user-defined name
Parameter types	Can only be used for declaring parameters

Elementary Data Types

Elementary data types define the structure of data elements which can not be subdivided into smaller units. They conform to the definition given in the standard DIN EN 1131-3.

SCL has twelve predefined elementary data types as follows:

BOOL	CHAR	INT	TIME
BYTE		DINT	DATE
WORD		REAL	TIME_OF_DAY
DWORD			S5TIME

Complex Data Types

Complex data types define the structure of data elements which are made up of a combination of other data elements. SCL allows the following complex data types:

```
DATE_AND_TIME
STRING
ARRAY
STRUCT
```

User-Defined Data Types

These are global data types (UDTs) which can be created in SCL for user-specific applications. This data type can be used with its UDT identifier UDTx (x represents a number) or an assigned symbolic name in the declaration section of a block or data block.

Parameter Types

In addition to elementary, complex and user-defined data types, you can also use parameter types for defining parameters. SCL provides the following parameter types for that purpose:

TIMER	BLOCK_FB	POINTER	ANY
COUNTER	BLOCK_FC		
	BLOCK_DB		
	BLOCK_SDB		

7.8 Variables

Declaration of Variables

An identifier whose assigned value can change during the process of execution of a program is called a *variable*. Each variable must be individually declared (that is, defined) before it can be used in a logic block or data block. The declaration of a variable specifies that an identifier is a variable (rather than a constant, etc.) and defines the variable type by assigning it to a data type.

The following types of variable are distinguished on the basis of their applicability:

- Local data
- Global user data
- Permissible predefined variables (CPU memory areas)

Local Data

Local data are declared in a logic block (FC, FB, OB) and apply only within that logic block. Specifically these are the following:

Table 7-3 Local Data of a Block

Variable Type	Explanation
Static Variables	A static variable is a local variable whose value is retained throughout all block cycles (block memory). It is used for storing values for a **function block**.
Temporary Variables	Temporary variables belong to a local logic block and do **not** occupy any static memory. Their values are retained for a single block cycle only. Temporary variables can **not** be accessed from outside the block in which they are declared.
Block Parameters	Block parameters are formal parameters of a function block. or a function. They are local variables that are used to pass over the current parameters specified when a block is called.

Global User-Defined Data

These are data or data areas that can be accessed from any point in a program. To use global user-defined variables, you must create data blocks (DBs).

When you create a DB, you define its structure in a structure declaration. Instead of a structure declaration, you can use a user-defined data type (UDT). The order in which you specify the structural components determines the sequence of the data in the DB.

CPU Memory Areas

You can access the memory areas of a CPU directly from any point in the program via the address identifiers (see Section 7.5) without having to declare those variables first.

Apart from that, you can always address those memory areas symbolically. Assignment of symbols is performed globally in this case by means of the symbol table in STEP 7. For more details, refer to **/231/**.

7.9 Expressions

Summary

An expression stands for a value that is calculated either when the program is compiled or when it is running. It consists of one or more addresses linked by operators. The order in which the operators are applied is determined by their priority and can also be controlled by bracketing.

- Mathematical expressions
- Logical expressions
- Comparative expressions

Mathematical Expressions

A typical example of a mathematical expression is

```
(b*b–4*a*c)/(2*a)
```

The identifiers a and b and the numbers 4 and 2 are the addresses, the symbols `*`, – and / are the corresponding operators (multiply, subtract and divide). The complete expression represents a numerical value.

Comparative Expressions

A comparative expression is a logical expression that can be either true or false. The following is an example of a comparative expression:

```
Setpoint < 100.0
```

In this expression, `SETPOINT` is a real variable, `100.0` a real number and the symbol < a comparator. The expression has the value True if the value of Setpoint is less than 100.0. If it is not, the value of the expression is **False**.

Logical Expression

The following is a typical example of a logical expression:

```
a AND NOT b
```

The identifiers a and b are the addresses, the keywords AND and NOT are logical operators. The complete expression represents a bit pattern.

7.10 Statements

Summary

An SCL statement is an executable action in the code section of a logic block. There are three basic types of statements in SCL:

- Value assignments (assignment of an expression to a variable)
- Control statements (repetition or branching statements)
- Subroutine calls (statements calling or branching to other logic blocks)

Value Assignments

The following is an example of a typical value assignment:

```
SETPOINT := 0.99*PREV_SETPOINT
```

This example assumes that `SETPOINT` and `PREV_SETPOINT` are real variables. The assignment instruction multiplies the value of `PREV_SETPOINT` by 0.99 and assigns the product to the variable `SETPOINT`. Note that the symbol for assignment is := .

Control Statements

The following is an example of a typical control statement:

```
FOR Count :=1 TO 20 DO
           LIST[Counter]    := VALUE+Counter;
END_FOR;
```

In the above example, the statement is performed 20 times over. Each time, the recalculated value in the array `LIST` is entered in the next highest position on the list.

Subroutine Call

By specifying a block identifier for a function (FC) or a function block (FB) you can call the block declared for that identifier. [1] If the declaration of the logic block includes formal parameters, then current addresses can be assigned to the formal parameters when the formal parameters are called.

All parameters listed in the declaration sections

```
VAR_INPUT, VAR_OUTPUT and VAR_IN_OUT
```

of a logic block are referred to as formal parameters - in contrast, the corresponding parameters included in the subroutine calls within the code section are termed actual parameters.

Assignment of the actual parameters to the formal parameters is part of the subroutine call.

The following is a typical example of a subroutine call:

```
FC31(X:=5, S1:=Sumdigits);
```

KEIN MERKER *If you have declared formal parameters in a function, the assignment of current parameters is mandatory, with function blocks it is optional.*

7.11 SCL Blocks

Overview

An SCL source file can contain any number of blocks as source code.

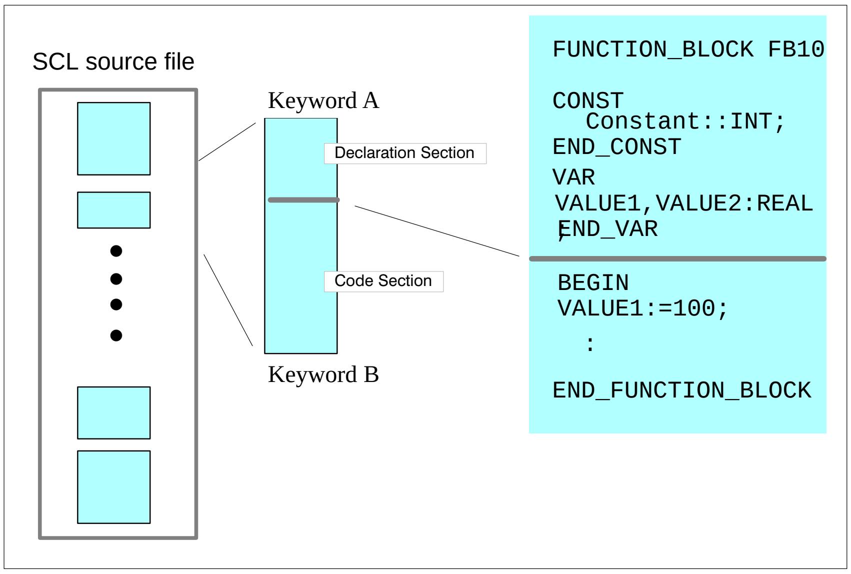
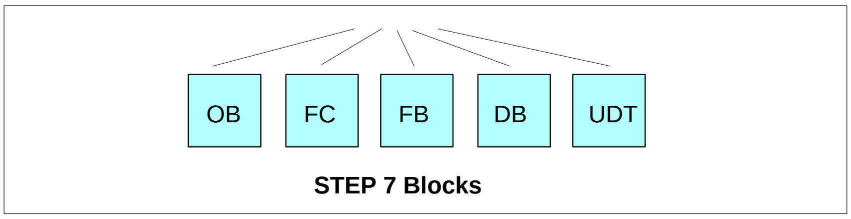


Figure 7-6 Structure of an SCL Source File

Types of Block

STEP 7 blocks are subunits of a user program delimited according to their function, their structure or their intended use. SCL allows you to program the following types of block:



Ready-Made Blocks

You do not have to program every function yourself. You can also make use of various ready-made blocks. They are to be found in the CPU operating system or libraries *(S7lib)* in the STEP7 Standard Package and can be used for programming communication functions, for example.

Structure of an SCL Block

All blocks consist of the following components:

- Start/end of block header (keyword corresponding to block type)
- Declaration section
- Code section (assignment section in the case of data blocks)

Declaration Section

The declaration section must contain all specifications required to create the basis for the code section, for example, definition of constants and declaration of variables and parameters.

Code Section

The code section is introduced by the keyword `BEGIN` and terminated with a standard identifier for the end of block; that is, `END_xxx` (see Section 8.2).

Every statement is concluded with a semicolon (“ ; ”). Each statement can also be preceded by a jump label. The syntax rules for the code section and the individual statements themselves are explained in Chapter 13.

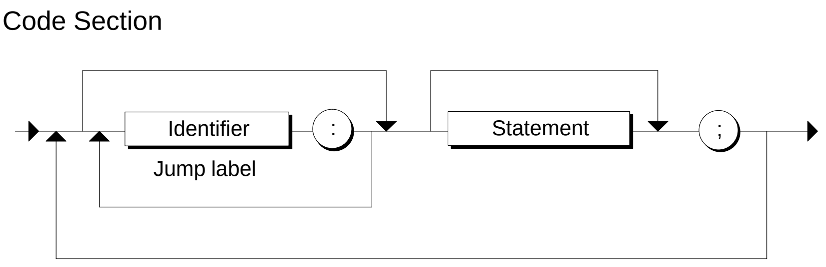


Figure 7-7 Syntax of a Statement

Below is an example of the code section of an FB:

```
:                    //End of declaration section
:
BEGIN                //START of code section
             X := X+1;
LABEL1       Y := Y+10;
             Z := X*Y;
             :
             GOTO LABEL1
LABELn;      FC10 := Z;//End of code section
END_FUNCTION_BLOCK
```

In the code section of a data block, you can assign initialization values to your DB data. For that reason, the code section of a DB is referred to from now on as the **assignment section**.

S7 Program

Following compilation, the blocks generated are stored in the “Blocks” folder of the S7 program. From here, they must be downloaded to the CPU. For details of how this is done, refer to **/231/**.

7.12 Comments

Summary

Comments are used for documentation and to provide an explanation of an SCL block. After compilation, comments have no effect whatsoever on the running of the program. There are the following two types of comments:

- Line comments
- Block comments

Line Comments

These are comments introduced by a double slash // and extending no further than the end of the line. The length of such comments is limited to a maximum of 253 characters including the identifying characters //. Line comments can be represented by the following syntax diagram:

Line Comment

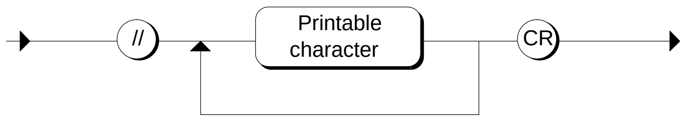


Figure 7-8 Syntax of a Line Comment

For details of the printing characters, please refer to Table A-2 in the Appendix. The character pairings using ‘(*’ and ‘*)’ have no significance inside line comments.

Block Comments

These are comments which can extend over a number of lines and are introduced as a block by ‘(*’ and terminated by ‘*)’. The nesting of block comments is permitted as standard. You can, however, change this setting and make the nesting of block comments impossible.

Block Comment

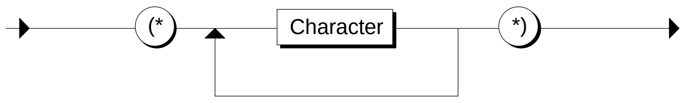


Figure 7-9 Syntax of a Block Comment

For details of the permissible characters, please refer to Table A-2 in the Appendix.

Points to Note

Observe the notation for comments:

- With block comments in **data blocks**, you must use the notation for block comments that is, these comments are also introduced with ‘`//`’.
- Nesting of comments is permitted in the default setting. This compiler setting can, however, be modified with the “Permit Nested Comments” option. To change the setting, select the menu command **Options ▸ Customize** and deselect the option in the “Compiler” tab page.
- Comments must not be placed in the middle of a symbolic name or a constant. They may, however, be placed in the middle of a string.

The following comment is **illegal**:

```
(*// FUNCTION_BLOCK // Adaptation *)
```

Example of the Use of Comments

The example shows two block comments and one line comment.

```
FUNCTION_BLOCK FB15
(* At this point there is a remarks block
which can extend over a number of lines *)
VAR
   SWITCH: INT;        Line comments
END_VAR;
BEGIN
  (* Assign a value to the variable SWITCH *)
  SWITCH:= 3;
END_FUNCTION_BLOCK
```

Figure 7-10 Example for Comments

Note

Line comments which come directly after the variable declaration of a block are copied to an STL program on decompilation.

You can find these comments in STL in the interface area; that is, in the upper part of the window (see also /**231**/).

In the example in Figure 7-10, therefore, the first line comment would be copied.

Structure of an SCL Source File 8

Introduction

An SCL source file basically consists of running text. A source file of this type can contain a number of blocks. These may be OBs, FBs, FCs, DBs, or UDTs.

This chapter explains the external structure of the blocks. The succeeding chapters then deal with the internal structures.

Chapter Overview

8.1 Structure

Introduction

An SCL source file consists of the source code made up of between 1 and n blocks (that is, FBs, FCs, OBs, DBs and UDTs).

In order that the individual blocks can be compiled from your SCL source file, they must must conform to specific structures and syntax rules.

SCL Program Subunit

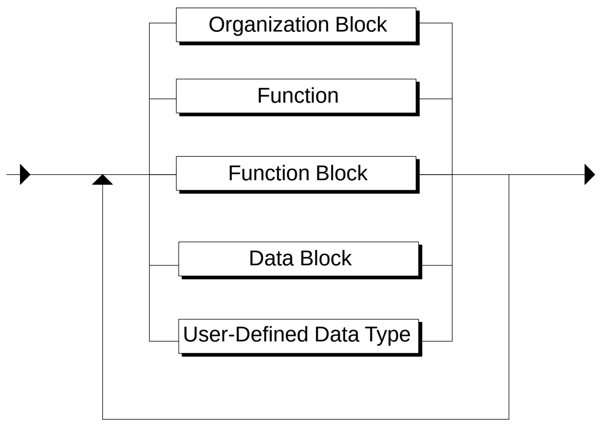


Figure 8-1 SCL Program Subunit

Order of Blocks

With regard to the order of the blocks, the following rules must be observed when creating the source file:

Called blocks must precede the calling blocks. This means:

- User-defined data types (UDTs) must precede the blocks in which they are used.
- Data blocks with an assigned user-defined data type (UDT) must follow the UDT.
- Data blocks that can be accessed by all logic blocks must precede all blocks which access them.
- Data blocks with an assigned function block come after the function block.
- The organization block OB1, which calls other blocks, comes at the very end. Blocks which are called by blocks called by OB1 must precede those blocks.

Blocks that you call in a source file, but that you do not program in the same source file must exist already when the file is compiled into the user program.

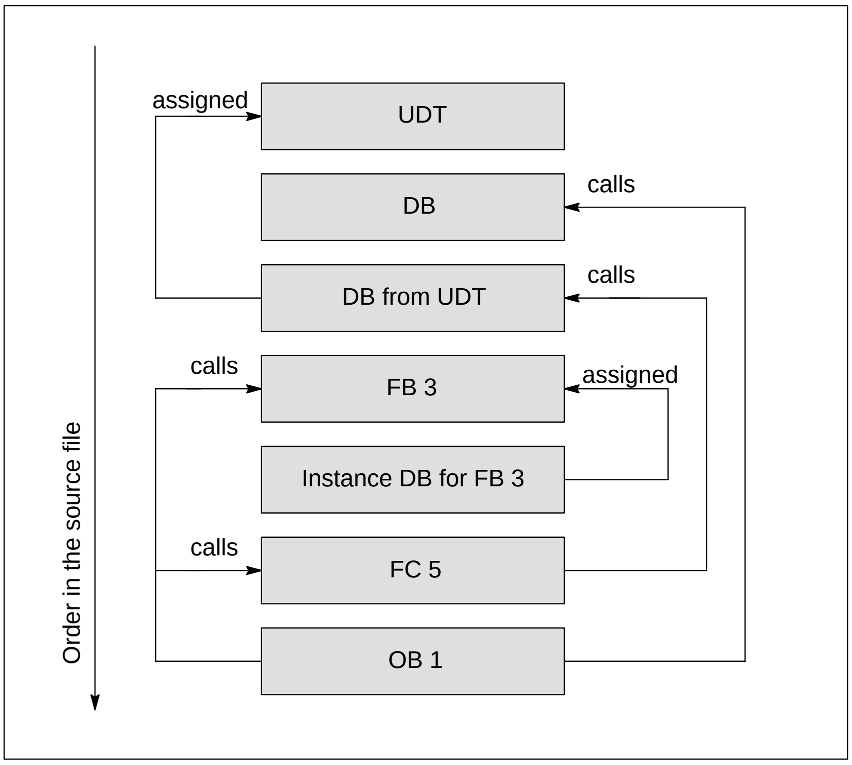


Figure 8-2 Block Structure of a Source File (Example)

General Block Structure

The source code for a block consists of the following sections:

- Block start with specification of the block (absolute or symbolic)
- Block attributes (optional)
- Declaration section (differs from block type to block type)
- Code section in logic blocks or assignment of current values in data blocks (optional)
- Block end

8.2 Beginning and End of a Block

Introduction

Depending on the type of block, the source text for a single block is introduced by a standard identifier for the start of the block and the block name. It is closed with a standard identifier for the end of the block (see Table 8-1).

Table 8-1 Standard Identifiers for Beginning and End of Blocks

Syntax

Syntax	Block Type	Identifier
`ORGANIZATION_BLOCK` *ob_name* : `END_ORGANIZATION_BLOCK`	**OB**	Organization block
`FUNCTION` *fc_name:functiontype* : `END_FUNCTION`	**FC**	Function
`FUNCTION_BLOCK` *fb_name* : `END_FUNCTION_BLOCK`	**FB**	Function block
`DATA_BLOCK` *db_name* : `END_DATA_BLOCK`	**DB**	Data block
`TYPE` *name udt_name* : `END_TYPE`	**UDT**	User-defined data type

Block Name

In Table 8-1 above, *xx_name* stands for the block name according to the following syntax:

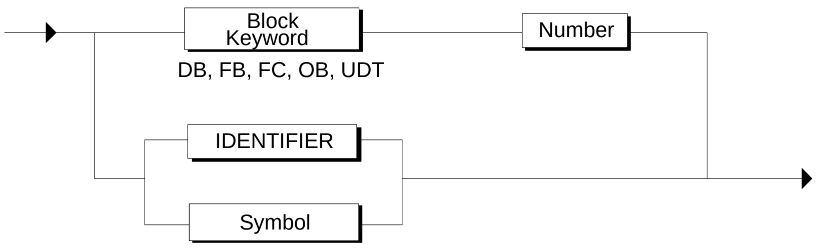


Figure 8-3 Syntax of the Block Name

More detailed information is given in Section 7.5. Please note also that you must define an identifier of a symbol in the STEP 7 symbol table (see **/231/**.).

Example

```
FUNCTION_BLOCK FB10
FUNCTION_BLOCK ControllerBlock
FUNCTION_BLOCK ”Controller.B1&U2”
```

8.3 Block Attributes

Definition

Attributes for blocks can be as follows:

- Block attributes
- System attributes for blocks

Block Attributes

The title, version, block protection, author, name and family of a block can be specified using keywords.

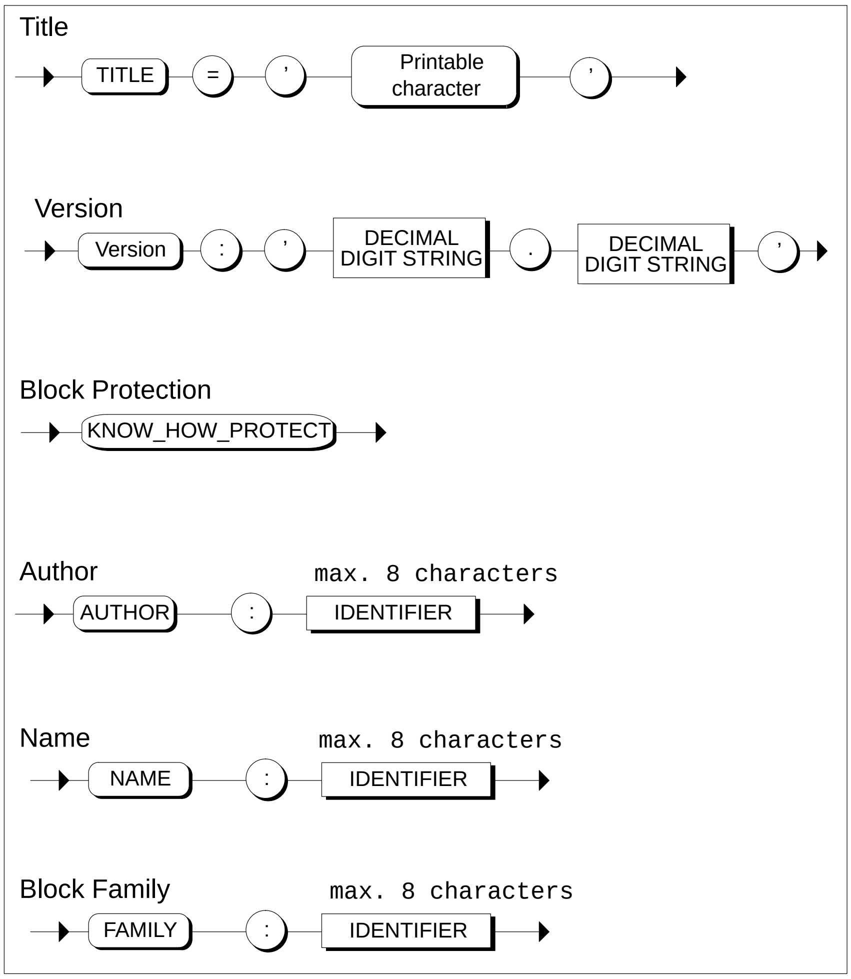


Figure 8-4 Syntax: Block Attributes

System Attributes for Blocks

You can also assign system attributes to blocks, for example for process control configuration.

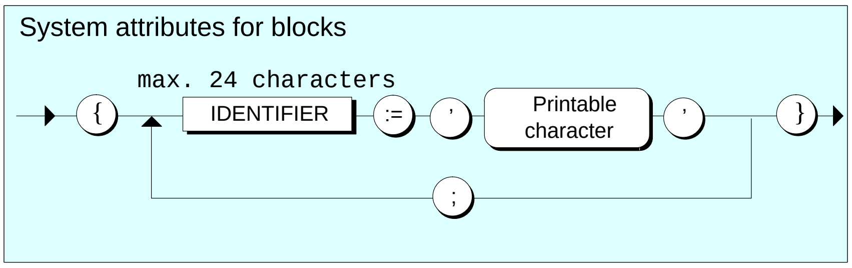


Figure 8-5 *Syntax: System Attributes for Blocks*

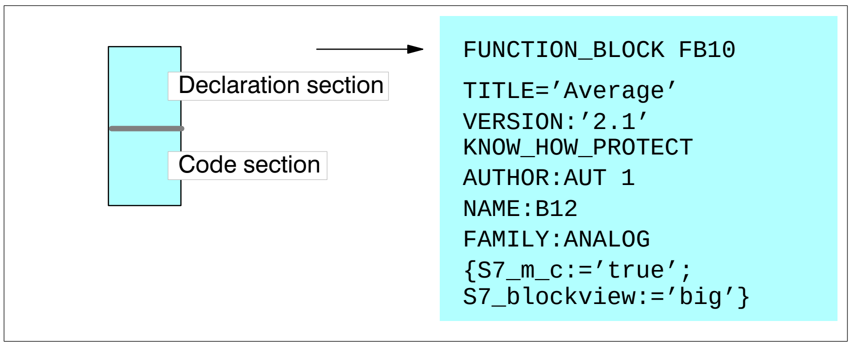
Table 8-2 shows which system attributes you can assign for blocks in SCL.

Table 8-2 System Attributes for Blocks

Attribute	Value	When to Assign the Attribute	Permitted Block Type
S7_m_c	true, false	When the block will be manipulated or monitored from an operator console.	FB
S7_tasklist	taskname1, taskname2, etc.	When the block will be called not only in the cyclic organization blocks but also in other OBs (for example error or startup OBs).	FB, FC
S7_block-view	big, small	When the block will be displayed on an operator console in big or small format.	FB, FC

Assigning Attributes

You assign block attributes **after** the block identifier and **before** the declaration section.



```
FUNCTION_BLOCK FB10
TITLE='Average'
VERSION:'2.1'
KNOW_HOW_PROTECT
AUTHOR:AUT 1
NAME:B12
FAMILY:ANALOG
{S7_m_c:='true';
S7_blockview:='big'}
```

Figure 8-6 Assigning Attributes

8.4 Declaration Section

Overview

The declaration section is used for defining local and global variables, parameters, constants, and jump labels.

- The local variables, parameters, constants, and jump labels that are to apply within a particular block only are defined in the declaration section of the code block.
- The global data that can be addressed by any code block are defined in the DB declaration section.
- In the declaration section of a UDT, you specify the user-defined data type.

Structure

A declaration section is divided into a number of declaration subsections, each delimited by its own pair of keywords. Each subsection contains a declaration list for data of a particular type, such as constants, labels, static data and temporary data. Each subsection type may only occur once and not every subsection type is allowed in every type of block, as shown in the table. There is no fixed order in which the subsections have to be arranged.

Declaration Subsections

Data	Syntax	FB	FC	OB	DB	UDT

Constants	`CONST`
	Declaration list
	`END_CONST`
		X	X	X		

Jump labels	`LABEL`
	Declaration list
	`END_LABEL`
		X	X	X		

Temporary variables	`VAR_TEMP`
	Declaration list
	`END_VAR`
		X	X	X		

Static variables	`VAR`
	Declaration list
	`END_VAR`
		X	X[2]		X[1]	X[1]

Input parameters	`VAR_INPUT`
	Declaration list
	`END_VAR`
		X	X			

Output parameters	`VAR_OUTPUT`
	Declaration list
	`END_VAR`
		X	X			

In/out parameters	`VAR_IN_OUT`
	Declaration liste
	`END_VAR`
		X	X			

Declaration list: the list of identifiers for the data type being declared						

1 *In DBs and UDTs, the keywords VAR and END_VAR are replaced by STRUCT and END_STRUCT respectively.*

2 *Although the declaration of variables within the keyword pair VAR and END_VAR is permitted in functions, the declarations are shifted to the temporary area during compilation.*

System Attributes for Parameters

You can also asssign system attributes to input, output, and in/out parameters, for example for message or connection configuration.

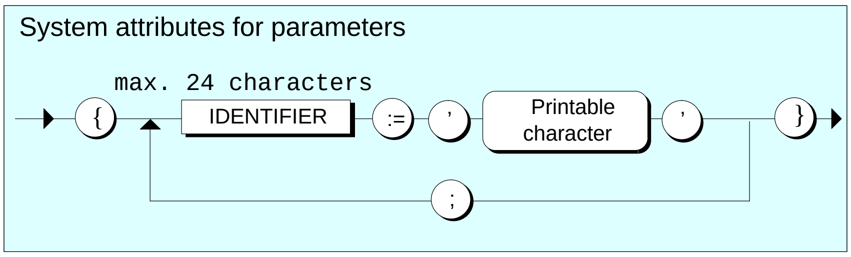


Figure 8-7 *Syntax: System Attributes for Parameters*

Table 8-3 shows which system attributes you can assign to the parameters:

Table 8-3 System Attributes for Parameters

Attribute	Value	When to Assign the Attribute	Permitted Declaration type
S7_server	connection, alarm_archiv	When the parameter is relevant to connection or message configuration. This parameter contains the connection or message number.	IN
S7_a_type	alarm, alarm_8, alarm_8p, alarm_s, notify, ar_send	When the parameter will define the message block type in a message block called in the code section (only possible when the S7_server attribute is set to alarm_archiv).	IN, only with blocks of the type FB
S7_co	pbkl, pbk, ptpl, obkl, fdl, iso, pbks, obkv	When the parameter will specify the connection type in the connection configuration (only possible when the S7_server attribute is set to connection).	IN
S7_m_c	true, false	When the parameter will be modified or monitored from an operator panel.	IN/OUT / IN_OUT, only with blocks of the type FB
S7_shortcut	Any 2 characters, for example, W, Y	When the parameter is assigned a shortcut to evaluate analog values.	IN/OUT / IN_OUT, only with blocks of the type FB
S7_unit	Unit, for example, liters	When the parameter is assigned a unit for evaluating analog values.	IN/OUT / IN_OUT, only with blocks of the type FB
S7_string_0	Any 16 characters, for example OPEN	When the parameter is assigned text for evaluating binary values.	IN/OUT/ IN_OUT, only with blocks of the type FB and FC

Table 8-3 System Attributes for Parameters, continued

Attribute	Value	When to Assign the Attribute	Permitted Declaration type
S7_string_1	Any 16 characters, for example, CLOSE	When the parameter is assigned text for evaluating binary values	IN/OUT / IN_OUT, only with blocks of the type FB and FC
S7_visible	true, false	When you do not want the parameter to be displayed in CFC.	IN/OUT / IN_OUT, only with blocks of the type FB and FC
S7_link	true, false	When you do not want the parameter to be linked in CFC.	IN/OUT / IN_OUT, only with blocks of the type FB and, FC
S7_dynamic	true, false	When you want the parameter to have dynamic capability when testing in CFC.	IN/OUT / IN_OUT, only with blocks of the type FB and FC
S7_param	true, false	When you want the parameter to be protected from incorrect value assignment in CFC.	IN/OUT / IN_OUT, only with blocks of the type FB and FC

Assigning Attributes

You assign system attributes for parameters in the declaration fields for input parameters, output parameters or in/out parameters.

Example:

```
VAR_INPUT
   in1  {S7_server:=’alarm_archiv’;
         S7_a_type:=’ar_send’}:DWORD;
END_VAR
```

8.5 Code Section

Summary

The code section contains statements[1]

- that are executed when a code block is called. These statements are used for processing data or addresses.
- for setting individual initialization values in data blocks.

Syntax

Figure 8-8 shows the syntax of the code section. It consists of a series of individual statements, each of which can be preceded by a jump label (see Section 11.6) which represents the destination for a jump statement.

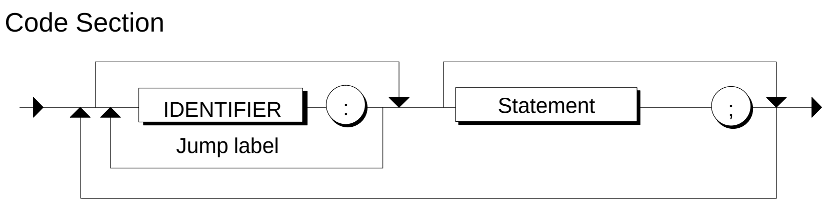


Figure 8-8 Code Section Syntax

Below are some examples of valid statements.

```
BEGIN
            STARTVALUE  :=0;
            ENDVALUE    :=200;
:
SAVE:       RESULT      :=SETPOINT;
:
```

Rules to Observe

The important points to observe when writing the code section are that:

- The code section starts as an option with the keyword `BEGIN`
- The code section is completed with the keyword for the end of the block.
- Every statement must be terminated with a semicolon.
- All identifiers used in the code section must have been declared.

1 *In this manual, the term "statement" is used for all constructs that declare an executable function.*

8.6 Statements

Summary

Each individual statement is one of the following types:

- **Value assignments** used to assign the result of an expression or the value of another variable to a variable.
- **Control statements** used to repeat statements or groups of statements or to branch within a program.
- **Subroutine calls** used to call functions or function blocks.

Statement

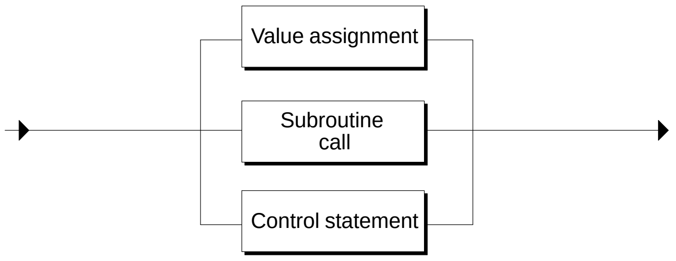


Figure 8-9 Syntax of a Statement

The elements required to formulate these statements are expressions, operators and addresses. These items are treated in more detail in subsequent chapters.

Examples

The following examples illustrate the various types of statement:

```
// Example of a value assignment
       MEASVAL:= 0 ;

// Example of a subroutine call
       FB1.DB11(TRANSFER:= 10) ;

// Example of a control statement
       WHILE COUNT < 10 DO..
       :
       END_WHILE;
```

Example 8-1 Statements

8.7 Structure of a Function Block (FB)

Overview

A function block (FB) is a logic block constituting part of a program and having a memory area assigned to it. Whenever an FB is called, an instance DB (see Chapter 10) must be assigned to it. You specify the structure of this instance DB when you define the FB declaration section.

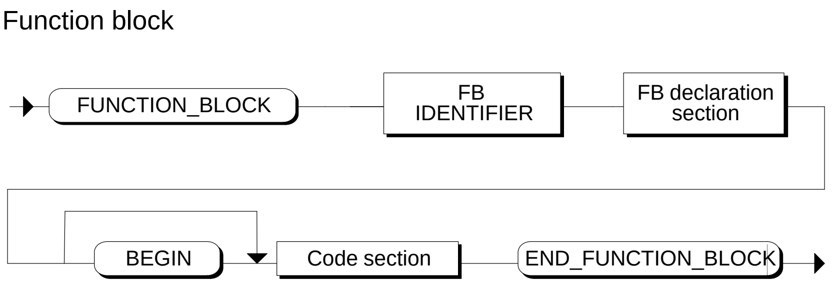


Figure 8-10 Structure of a Function Block (FB)

FB Identifier

After the keyword

```
FUNCTION_BLOCK
```

specify the keyword FB followed by the block number or the symbolic name of the FB as the FB identifier.

Examples:

```
FUNCTION_BLOCK FB10
```

```
FUNCTION_BLOCK MOTOR_1
```

FB Declaration Section

The FB declaration section is used to establish the block-specific data. For details of the permissible declaration subsections, refer to Section 8.4. Note that the declaration section also determines the structure of the assigned instance DB.

Examples:

```
CONST
  CONSTANT:=5;
END_CONST


VAR
  VALUE1,VALUE2,VALUE3:INT;
END_VAR
```

Example

Example 8-2 shows the source code for a function block. The input and output parameters (in this case, V1 and V2) are assigned initial values in this example.

```
FUNCTION_BLOCK FB11
      VAR_INPUT
            V1: INT:= 7;
      END_VAR
      VAR_OUTPUT
            V2: REAL;
      END_VAR
      VAR
            PASS_1:INT;
      END_VAR
      BEGIN
      IF V1 = 7 THEN
      PASS_1:= V1;
      V2:= FC2 (TESTVAL:= PASS_1);
      //Call function FC2 and
      //supply parameters by means of static
      //variable PASS_1
      END_IF;
END_FUNCTION_BLOCK
```

Example 8-2 Example of a Function Block

8.8 Structure of a Function (FC)

Overview

A function (FC) is a logic block that is not assigned its own memory area. For this reason, it does not require an instance DB. In contrast to an FB, a function can return a function result (**return value**) to the point from which it was called. A function can therefore be used like a variable in an expression. Functions of the type VOID do not have a return value.

Function

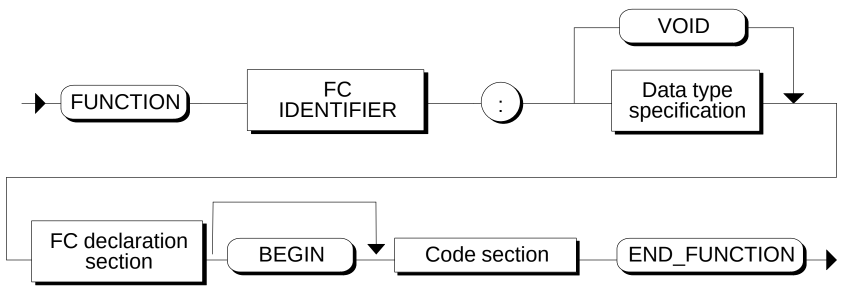


Figure 8-11 Syntax of a Function (FC)

FC Names

After the keyword

```
FUNCTION
```

specify the keyword FC followed by the block number or the symbolic name of the FC as the FC identifier.

Examples:

```
FUNCTION FC100
FUNCTION SPEED
```

Date Type Specification

Here you specify the data type of the return value. The permissible data types are all those described in Chapter 9, with the exception of data types STRUCT and ARRAY. A data type does not need to be specified if a return value is dispensed with by the use of **VOID**.

FC Declaration Section

The permissible declaration sections are described in detail in Section 8.4.

Code Section

Within the code section, the function name must be assigned the **function result**. The following is an example of a valid statement within a function with the name FC31:

```
FC31:= VALUE;
```

Example

The example below shows a function with the formal input parameters x1, x2, y1 and y2, a formal output parameter Q2 and a return value FC11.

For an explanation of formal parameters, refer to Chapter 10.

```
FUNCTION FC11: REAL
       VAR_INPUT
              x1: REAL;
              x2: REAL;
              y1: REAL;
              y2: REAL;
       END_VAR
       VAR_OUTPUT
              Q2: REAL;
       END_VAR
       BEGIN        // Code section
       FC11:= SQRT  // Return of function value

       ( (x2 - x1)**2 + (y2 - y1) **2 );
       Q2:= x1;
END_FUNCTION
```

Example 8-3 Example of a Function

8.9 Structure of an Organization Block (OB)

Overview

An organization block (OB), like an FB or FC, is part of a user program and is called by the operating system cyclically or when certain events occur. It provides the interface between the user program and the operating system.

Organization Block

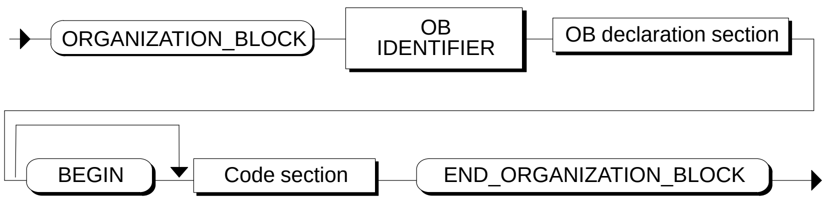


Figure 8-12 Syntax of an Organization Block

OB Name

After the keyword

```
ORANIZATION_BLOCK
```

specify the keyword OB followed by the block number or the symbolic name of the OB as the OB identifier.

Examples:

```
ORGANIZATION_BLOCK OB14
```

```
ORGANIZATION_BLOCK TIMER_ALARM
```

OB Declaration Section

In order to run, each OB has a basic requirement of **20 bytes of local data** for the start information. Depending on the requirements of the program, you can also declare additional temporary variables in the OB. For a description of the 20 bytes of local data, please refer to **/235/** .

Example:

```
ORGANIZATION_BLOCK OB14
//TIMER_ALARM

VAR_TEMP
HEADER:ARRAY [1..20] OF BYTE;// 20 bytes for
startinformation
:
:
END_VAR
```

For details of the remaining permissible declaration subsections for OBs, please refer to Section 8.4.

8.10 Structure of a Data Block (DB)

Overview

A data block (DB) contains global user-specific data which is to be accessible to **all** blocks in the program. Each FB, FC or OB can read or write data from/to global DBs. The structure of data blocks which are assigned to specific FBs only (instance DBs) is described in Chapter 12.

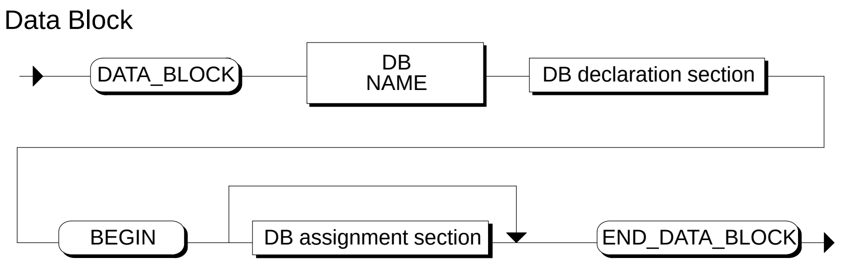


Figure 8-13 Syntax of a Data Block (DB)

DB Name

After the keyword

```
DATA_BLOCK
```

specify the keyword DB followed by the block number or the symbolic name of the DB as the DB identifier.

Examples:
```
DATA_BLOCK DB20
DATA_BLOCK MEASRANGE
```

DB Declaration Section

In the DB declaration section, you define the data structure of the DB. A DB variable can be assigned either a structured data type (STRUCT) or a user-defined data type (UDT).

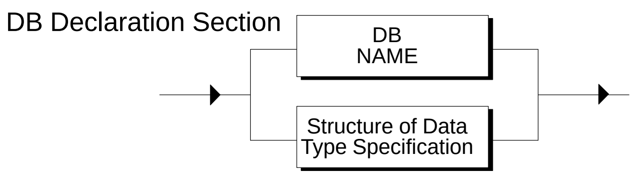


Figure 8-14 Syntax of the DB Declaration Section

Example:
```
DATA_BLOCK DB 20
       STRUCT     // Declaration section
       VALUE:ARRAY [1..100] OF INT;
       END_STRUCT
BEGIN             // Start of assignment section
:
END_DATA_BLOCK    // End of data block
```

DB Assignment Section

In the assignment section, you can adapt the data you have declared in the declaration section so that it has DB-specific values for your particular application. The assignment section begins with the keyword

`BEGIN`

and then consists of a sequence of value assignments with the following syntax:

DB Assignment Section

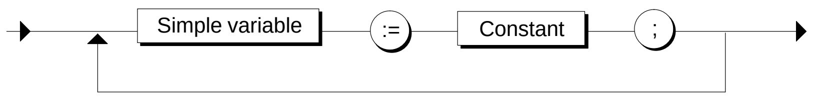


Figure 8-15 Syntax of the DB Assignment Section

Note

When assigning initial values (initialization), STL syntax applies to entering attributes and comments within a DB. For information on how to write constants, attributes and comments, consult the user manual /**231**/ or the manual /**232**/.

Example

The example below illustrates how the assignment section can be formulated if the array values [1] and [5] are to have the integer values 5 and –1 respectively instead of the initialization value 1.

```
DATA_BLOCK    DB20
STRUCT        //Data declaration with
              //initialization values
       VALUE  : ARRAY [ 1..100] OF INT := 100 (1);
       MARKER: BOOL := TRUE;
       S_WORD: WORD := W#16#FFAA;
       S_BYTE: BYTE := Bq16qFF;
       S_TIME: S5TIME := S5T#1h30m30s;
END_STRUCT

BEGIN                //Assignment section
   //Value assignments for specific array elements
   VALUE [1]  := 5;
   VALUE [5]  :=–1;
END_DATA_BLOCK
```

Example 8-4 Assignment Section of a DB

8.11 Structure of a User-Defined Data Type (UDT)

Overview

User-defined data types (UDTs) are special data structures created by the user. Since user-defined data types are assigned names they can be used many times over. By virtue of their definition, they can be used at any point in the CPU program and are thus global data types. As such, they can therefore

- be used in blocks in the same way as elementary or complex data types, or
- be used as templates for creating data blocks with the same data structure.

User-Defined Data Type

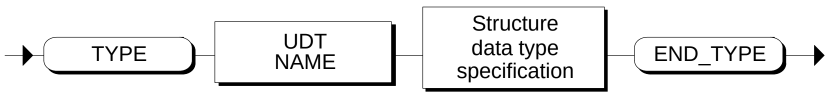


Figure 8-16 Syntax of a User-Defined Data Type (UDT)

Naming UDTs

After the keyword

```
TYPE
```

specify the keyword UDT followed by a number or simply the symbolic name of the UDT.

Examples:

```
TYPE UDT 10
TYPE SUPPLY_BLOCK
```

Specifying Data Types

The data type is always specified with a **STRUCT data type specification**. The data type UDT can be used in the declaration subsections of logic blocks or in data blocks or assigned to DBs. For details of the permissible declaration subsections and other information, please refer to Chapter 9.

Data Types 9

Introduction

A data type is the combination of value ranges and operations into a single unit. SCL, like most other programming languages, has a number of predefined data types (that is, integrated in the language). In addition, the programmer can create complex and user-defined data types.

Chapter Overview

9.1 Overview

Overview

Table 9-1 shows the various data types in SCL:

Table 9-1 Data Types in SCL

Elementary Data Types			
BOOL BYTE WORD DWORD	CHAR	INT DINT REAL	TIME DATE TIME_OF_DAY S5TIME
Complex Data Types			
DATE_AND_TIME	STRING	ARRAY	STRUCT
User-Defined Data Types			
UDT			
Parameter Types			
TIMER COUNTER	BLOCK_FB BLOCK_FC BLOCK_DB BLOCK_SDB	POINTER	ANY

The above data types determine:

- the nature and interpretation of the data elements,
- the permissible value ranges for the data elements,
- the permissible number of operations that can be performed by an operand of a data type, and
- the format of the data of a data type.

9.2 Elementary Data Types

Overview

Elementary data types define the structure of data elements that cannot be subdivided into smaller units. They correspond to the definition given in the standard DIN EN 1131-3. An elementary data type defines a memory area of a fixed size and represents bit, integer, real, period, time and character values. These data types are all predefined in SCL.

Table 9-2 Bit Widths and Value Ranges of Elementary Data Types

Type	Keyword	Bit Width	Value Range
Bit Data Type	Data elements of this type are either 1Bit (BOOL data type), 8 Bits, 16 Bits or 32 Bits in length.		
Bit	BOOL	1	`0,1 or FALSE, TRUE`
Byte	BYTE	8	A numerical value range can not be specified. These are bit combinations which can not be expressed in numerical terms.
Word	WORD	16	
Double word	DWORD	32	
Character Type	Data elements of this type occupy exactly 1 character in the ASCII character set		
Individual Characters	CHAR	8	Extended ASCII character set
Numeric Types	These are used for processing numerical values		
Integer (whole number)	INT	16	`-32_768` to `32_767`
Double integer	DINT	32	`-2_147_483_648` to `2_147_483_647`
Floating point number (IEE floating point number)	REAL	32	-3.402822E+38 to -1.175495E-38, 0.0, +1.175495E-38 to 3.402822E+38
Time Types	Data elements of this type represent different date values in STEP 7.		
S5 time	S5TIME	16	T#0H_0M_0S_10MS to T#2H_46M_30S
Time IEC time in increments of 1 ms	TIME (=DURATION)	32	`-T#24D_20H_31M_23S_647MS` to `T#24D_20H_31M_23S_647MS`
Date IEC date in increments of 1 day	DATE	16	`D#1990-01-01` to `D#2168-12-31`
Time of day Time of day in increments of 1 ms	TIME_OF_DAY (=TOD)	32	TOD#0:0:0 to TOD#23:59:59.999

Note on S5 time: Depending whether the time base is 0.01S, 0.1S, 1S or 10S, the time resolution is limited accordingly. The compiler rounds the values accordingly.

9.3 Complex Data Types

Overview

SCL supports the following complex data types:

Table 9-3 Complex Data Types

Data Type	Description
DATE_AND_TIME DT	Defines an area of 64 bits (8 bytes). This data type stores date and time (as a binary coded decimal) and is a predefined data type in SCL.
STRING	Defines an area for a character string of up to 254 characters (DATA TYPE CHAR).
ARRAY	Defines an array consisting of elements of one data type (either elementary or complex).
STRUCT	Defines a group of data types in any combination of types. It can be an array of structures or a structure of structures and arrays.

9.3.1 DATE_AND_TIME Data Type

Overview

The data type DATE_AND_TIME is made up of the data types DATE and TIME. It defines an area of 64 bits (8 bytes) for specifying the date and time. The data area stores the following information (in binary coded decimal format): year–month–day–hours: minutes: seconds.milliseconds.

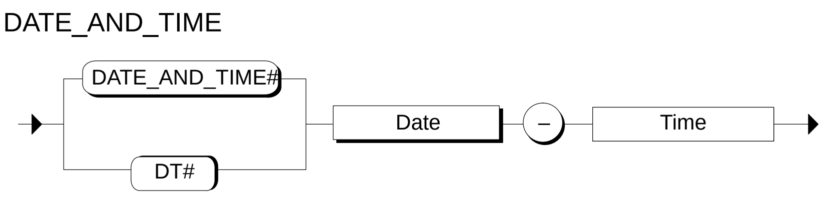


Figure 9-1 Syntax of DATE_AND_TIME

Value Range

Table 9-4 Bit widths and value ranges

Type	Keyword	Bits	Range of Values
Date and time	DATE_AND_TIME (=DT)	64	`DT#1990-01-01:0:0:0.0` to `DT#2089-12-31:23:59:59.999`

The precise syntax for the date and time is described in Chapter 11 of this manual. Below is a valid definition for the date and time 20/10/1995 12:20:30 and 10 milliseconds.

```
DATE_AND_TIME#1995-10–20–12:20:30.10
DT#1995–10–20–12:20:30.10
```

Note

There are standard FCs available for accessing the specific components DATE or TIME.

9.3.2 STRING Data Type

Overview

A STRING data type defines a character string with a maximum of 254 characters.

The standard area reserved for a character string consists of 256 bytes. This is the memory area required to store 254 characters and a header consisting of two bytes.

You can reduce the memory required by a character string by defining a maximum number of characters to be saved in the string. A *null string*, in other words a string containing no data, is the smallest possible value.

STRING Data Type Specification

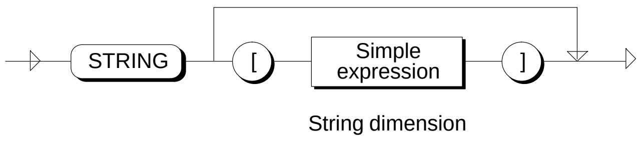


Figure 9-2 Syntax of the STRING Data Type Specification

The simple expression (string dimension) represents the maximum number of characters in the string.

The following are some examples of valid string types:

```
STRING[10]
STRING[3+4]
STRING[3+4*5]
STRING      max. value range (default ≙ 254 characters)
```

Value Range

Any characters in the ASCII character set are permitted in a character string. Chapter 11 describes how control characters and non-printing characters are treated.

Note

In the case of return values, input and in/out parameters, the standard length of the data type STRING can be reduced from 254 characters to a number of your choice, in order to make better use of the resources on your CPU. Select the **Customize** menu command in the **Options** menu and then the “Compiler” tab. Here, you can enter the required number of characters in the “Maximum String Length” option box.

9.3.3 ARRAY Data Type

Overview

The array data type has a specified number of components of particular data type. In the syntax diagram for arrays shown in Figure. 9-3, the data type is precisely specified by means of the reserved word OF. SCL distinguishes between the following types of array:

- The one-dimensional ARRAY type.
  (This is a list of data elements arranged in ascending order).
- The two-dimensional ARRAY type.
  (This is a table of data consisting of rows and columns. The first dimension refers to the row number and the second to the column number).
- The multidimensional ARRAY type.
  (This is an extension of the two-dimensional ARRAY type adding further dimensions. The maximum number of dimensions permitted is six).

ARRAY Data Type Specification

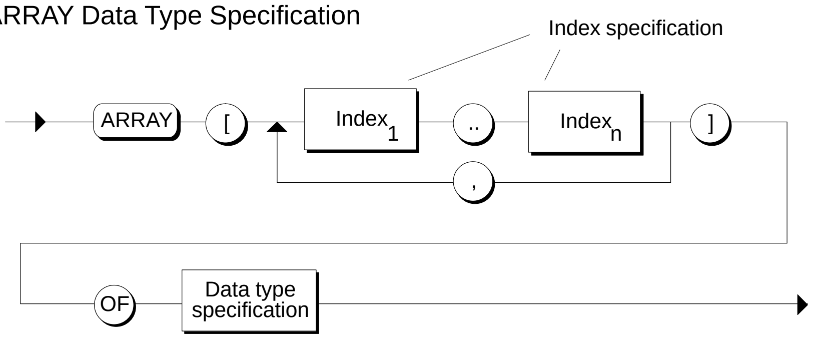


Figure 9-3 Syntax of ARRAY Data Type Specification

Index Specification

This describes the dimensions of the ARRAY data type as follows:

- The smallest and highest possible index (index range) for each dimension. The index can have any integer value (–32768 to 32767).
- The limits must be separated by two full stops.
- The individual index ranges must be separated by commas. The entire index specification is enclosed in square brackets.

Data Type Specification

The data type specification is used to declare the data type of the array components. The permissible data types are all those detailed in this section. The data type of an ARRAY can also be a structure.

The following specifications are examples of possible array types:

```
ARRAY[1..10] OF REAL
ARRAY[1..10] OF STRUCT..END_STRUCT
ARRAY[1..100, 1..10] OF REAL
```

9.3.4 STRUCT Data Type

Overview

A STRUCT data type describes an area consisting of a fixed number of components that can be of different data types. These data elements are specified in Figure 9-4 immediately following the STRUCT keyword in the component declaration. The main feature of the STRUCT data type is that a data element within it can also be structured. This means that nesting of STRUCT data types is permitted. Chapter 10 explains how to access the data of a structure.

STRUCT

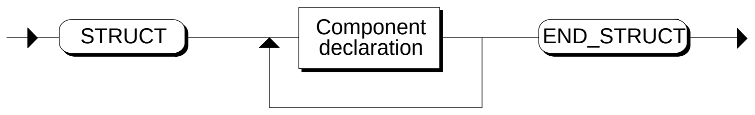


Figure 9-4 Syntax of STRUCT Data Type Specification

Component Declaration

This is a list of the various components in a structure. As shown in the syntax diagram in Figure 9-5, this list consists of:

- 1 to n identifiers
- the assigned data type and
- optional specification of an initial value

Component Declaration

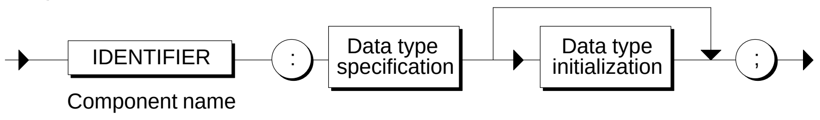


Figure 9-5 Syntax of a Component Declaration

Identifier

This is the name of a structure element to which the subsequent data type specification is to apply.

Data Type Initialization

You have the option of specifying an initial value for a specific structure element after the data type specification. Assignment is made by means of a value assignment as described in Chapter 10.

Example

The example below illustrates a definition of a `STRUCT` data type.

```
STRUCT
 //START of component declaration
  A1       :INT;
  A2       :STRING[254];
  A3       :ARRAY [1..12] OF REAL;
   |              |
   Component names  Data type specifications
//END of component declaration
END_STRUCT
```

Example 9-1 Definition of a `STRUCT` Data Type

9.4 User-Defined Data Type (UDT)

Overview

As explained in Chapter 8, a UDT data type is defined as a block. By virtue of its definition, such a data type can be used at any point of the CPU program and is thus a global data type. You can use these data types with their UDT name, UDTx (x represents a number), or with an assigned symbolic name defined in the declaration section of a block or data block.

User-Defined Data Type

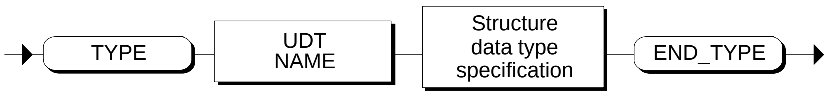


Figure 9-6 Syntax of a User-Defined Data Type (UDT)

UDT Name

A declaration for a UDT is introduced by the keyword TYPE followed by the name of the UDT (UDT identifier). The name of the UDT can either be specified in absolute form, that is, by a standard name in the form UDTx (x stands for a number), or else a symbolic name can be used (see also Chapter 8).

Data Type Specification

The UDT name is followed by the data type specification. The only data type specification permissible in this case is STRUCT (see Section 9.3.4).

```
STRUCT
:
END_STRUCT
```

Subsequently, the complete declaration for the UDT is concluded with the keyword

```
END_TYPE
```

Using UDTs

The data type thus defined can be used for variables or parameters or declaring DBs. Components of structures or arrays, including those inside other UDTs, can also be declared by means of UDTs.

Note

When assigning initial values (initialization) within a UDT, STL syntax applies. For information on how to write constants, consult the user manual /**231**/ or manual /**232**/.

Example

The example below illustrates the definition of a UDT and the use of this data type within a variable declaration. It is assumed that the name ”MEASDATA” has been declared for UDT50 in the symbol table.

```
TYPE MEASDATA // UDT Definition
STRUCT
BIPOL_1 : INT;
BIPOL_2 : WORD := W#16#AFAL;
BIPOL_3 : BYTE := B#16#FF;
BIPOL_4 : WORD := B#(25,25);
BIPOL_5 : INT  := 25;
S_TIME : S5TIME:= S5T#1h20m10s;
READING:      STRUCT
              BIPOLAR_10V: REAL;
              UNIPOLAR_4_20MA: REAL;
              END_STRUCT;
END_STRUCT
END_TYPE
```

```
FUNCTION_BLOCK
VAR
      MEAS_RANGE: MEASDATA;
END_VAR
BEGIN
...
      MESS_RANGE.BIPOL:= -4;
      MESS_RANGE.READING.UNIPOLAR_4_20MA:= 2.7;
...
END_FUNCTION_BLOCK
```

Example 9-2 Declaration of User-Defined Data Types

9.5 Parameter Types

Overview

In addition to elementary, complex and user-defined data types, you can also use so-called **parameter types** for specifying the formal block parameters for FBs and FCs. These data types are used for the following:

- declaring timer/counter functions as parameters (`TIMER/COUNTER`),
- declaring FCs, FBs, DBs and SDBs as parameters (BLOCK_xx)
- allowing an address of any data type as a parameter (`ANY`)
- allowing a memory area as a parameter (POINTER)

Table 9-5 Parameter Types

Parameter	Size	Description
TIMER	2 bytes	Identifies a specific timer to be used by the program in the logic block called. Actual parameter: `e.g. T1`
COUNTER	2 bytes	Identifies a specific counter to be used by the program in the logic block called. Actual parameter: `e.g. C10`
BLOCK_FB BLOCK_FC BLOCK_DB BLOCK_SDB	2 bytes	Identifies a specific block to be used by the program in the block called. Actual parameter: `e.g. FC101` `DB42`
ANY	10 bytes	Used if any data type with the exception of ANY is to be allowed for the data type of the actual parameter.
POINTER	6 bytes	Identifies a particular memory area to be used by the program. Actual parameter: `e.g. M50.0`

TIMER and COUNTER

You specify a particular timer or a counter to be used when processing a block. The TIMER and COUNTER data types are only permitted for input parameters (VAR_INPUT).

BLOCK Types

You specify a certain block to be used as an input parameter. The declaration of the input parameter determines the type of block (FB, FC or DB). When supplying parameters, you specify the absolute block identifier either in absolute form (for example, FB20) or by a symbolic name.

SCL does not provide any operations which manipulate these data types. Parameters of this type can only be supplied with data in the course of subroutine calls. In the case of FCs, input parameters cannot be passed on.

In SCL, you can assign addresses to the following data types as actual parameters:

- Function blocks without formal parameters
- Function blocks without formal parameters and return value (VOID)
- Data blocks and system data blocks.

ANY

In SCL it is possible to declare block parameters of the data type ANY. When such a block is called, these parameters can be supplied with addresses of any data type. SCL, however, provides only one method of processing the ANY data type, namely passing on to underlying blocks.

You can assign addresses of the following data types as the actual parameter:

- 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 and the complex data type.
- `ANY` data type
  This is only possible when the address is a parameter type that does not clash with the formal parameter.
- `NIL` data type
  You specify a zero pointer.
- Timers, counters, and blocks
  You specify the identifier (for example, T1, C20 or FB6).

The data type ANY is permitted for formal input parameters, in/out parameters of FBs and FCs, and for output parameters of FCs.

Note

If you supply a temporary variable to a formal parameter of the ANY type when an FB or FC is called, you must not pass on this parameter to a further block in the block that was called. The addresses of temporary variables lose their validity when they are passed on.

POINTER

In SCL, you can declare block parameters of the POINTER data type and can supply these parameters with addresses of any data type when such a block is called. SCL, however, provides only one method of processing the ANY data type, namely passing on to underlying blocks.

You can assign addresses of the following data types as the actual parameter in SCL:

- 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 and the complex data type (for example arrays and structures).
- `POINTER` data type
  This is only possible when the address is a parameter type that does not clash with the formal parameter.
- `NIL` data type
  You specify a zero pointer.

The `POINTER` data type is permitted for formal input parameters, in/out parameters of FBs and FCs and for output parameters of FCs.

Note

If you supply a temporary variable to a formal parameter of the POINTER type when an FB or FC is called, you must not pass on this parameter to a further block in the block that was called. The addresses of temporary variables lose their validity when they are passed on.

Examples

```
FUNCTION GAP: REAL
      VAR_INPUT
            MyDB:BLOCK_DB;
            TIME   : TIMER;
      END_VAR
      VAR
            INDEX: INTEGER;
      END_VAR
      BEGIN
      MyDB.DB5:=5;
      GAP:=....          // RETURNVALUE

END_FUNCTION
```

Example 9-3 BLOCK_DB and TIMER Data Types

```
FUNCTION FC100: VOID
      VAR_IN_OUT
            in, out:ANY;
      END_VAR
      VAR_TEMP
            ret: INT;
      END_VAR
      BEGIN
      //...
      ret:=SFC20(DSTBLK:=out,SCRBLK:=in);
      //...
END_FUNCTION

FUNCTION_BLOCK FB100
      VAR
            ii:INT;
            aa, bb:ARRAY[1..1000] OF REAL;
      END_VAR
      BEGIN
      //...
      FC100(in:=aa, out:=bb);
      //...
END_FUNCTION_BLOCK
```

Example 9-4 ANY Data Type

Declaring Local Variables and Block Parameters

10

Introduction

Local variables and block parameters are data that are declared within a code block (FC, FB or OB) and are valid only within that logic block. This chapter explains how such data are declared and initialized.

Chapter Overview

10.1 Overview

Categorization of Variables

Local variables can be subdivided into the categories shown in Table 10-1:

Table 10-1 Local Variables

Variable	Explanation
Static Variables	Static variables are local variables whose value is retained throughout all block cycles (block memory). They are used to store values for a function block and are stored in the instance data block.
Temporary Variables	Temporary variables belong to a logic block at local level and do **not** occupy a static memory area, since they are stored in the CPU stack. Their values are retained only for the duration of a single block cycle. Temporary variables can **not** be accessed from outside the block in which they are declared.

Categorization of Block Parameters

Block parameters are placeholders that are definitely specified only when the block is actually used (called). The placeholders in the block are termed formal parameters and the values assigned to them when the block is called are referred to as the actual parameters. The formal parameters of a block can be viewed as local variables.

Block parameters can be subdivided into the categories shown in Table 10-2:

Table 10-2 Block Parameters

Block Parameter Type	Explanation
Input Parameters	Input parameters accept the current input values when the block is called. They are read-only.
Output parameters	Output parameters transfer the current output values to the calling block. Data can be written to and read from them.
In/out parameters	In/out parameters copy the current value of a variable when the block is called, process it and write the result back to the original variable.

Flags (OK Flag)

The SCL compiler provides a flag which can be used for detecting errors when running programs on the CPU. It is a local variable of the type BOOL with the predefined name “OK”.

Declaring Variables and Parameters

As shown in Table 10-3, each category of local variables or parameters is assigned as well a separate declaration subsection as its own pair of keywords.

Each subsection contains the declarations that are permitted for that particular declaration subsection. Each subsection may only appear once in the declaration section of the block. There is no specified order in which the subsections have to be placed.

The declaration subsections permitted within a particular block are marked with an “x” in Table 10-3.

Table 10-3 Declaration Subsections for Local Variables and Parameters

Data	Syntax	FB	FC	OB

Static Variables	`VAR`
	`:`
	`END_VAR`
		X	X1)	

Temporary Variables	`VAR_TEMP`
	`:`
	`END_VAR`
		X	X	X

Block Parameters:
Input parameters
	`VAR_INPUT`
	`:`
	`END_VAR`
		X	X	

Output parameters	`VAR_OUTPUT`
	`:`
	`END_VAR`
		X	X	

In/out parameters	`VAR_IN_OUT`
	`:`
	`END_VAR`
		X	X	

1) *Although the declaration of variables within the keyword pair VAR and END_VAR is permitted in functions, the declarations are shifted to the temporary area during compilation.*

Initialization

When they are declared, the variables and parameters must be assigned a data type which determines the structure and thereby the memory requirements. In addition, you can also assign static variables and function block parameters initial values. Table 10-4 summarizes the situations in which initialization is possible.

Table 10-4 Initialization of Local Data

Data Category	Initialization
Static Variables	Possible
Temporary Variables	Not possible
Block Parameters	Only possible in the case of input or output parameters of a function block

10.2 Declaring Variables and Parameters

Summary

A variable or parameter declaration consists of a user-definable identifier for the variable name and details of the data type. The basic format is shown in the syntax diagram below. Assigning system attributes for parameters is described in Section 8.4.

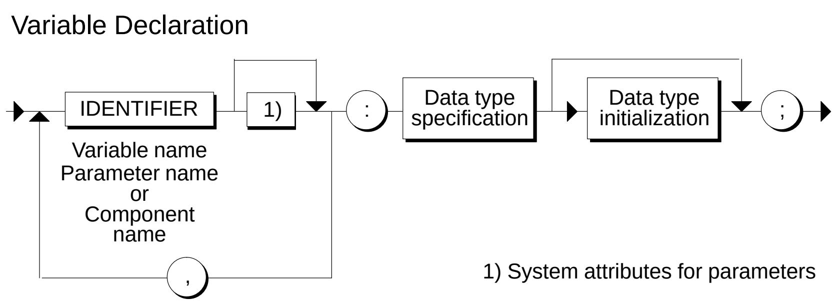


Figure 10-1Syntax of a Variable Declaration

The following are examples of valid declarations:

```
VALUE1 : REAL;
```

Or, if there are several variables of the same type:

```
VALUE2, VALUE2,VALUE4,....: INT;

ARRAY  : ARRAY[1..100, 1..10] OF REAL;

SET    : STRUCT
           MEASBAND:ARRAY[1..20] OF REAL;
           SWITCH:BOOL;
         END_STRUCT
```

Data Type Specification

All data types dealt with in Chapter 9 are permissible.

Note

Reserved words, which are only valid in SCL, can be declared as identifiers by putting the character “#” in front (for example, #FOR). This can be useful if you want to transfer the actual parameters to blocks which were created in a different language (for example, STL).

10.3 Initialization

Principle

Static variables, input parameters and output parameters of an FB can be assigned an initial value when they are declared. The initialization is performed by means of a value assignment (:=) which follows the data type specification. As shown in the syntax diagram in Figure 10-2, you can either:

- assign a simple variable a constant or
- assign an array an initialization list

Initialization

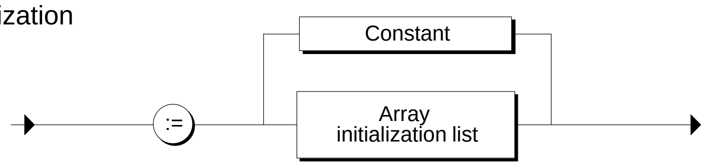


Figure 10-2 Syntax of Data Type Initialization

Example:

VALUE :REAL := 20.25;

Note that initialization of a list of variables (A1, A2, A3,...: INT:=...) is not possible. In such cases, the variables have to be initialized individually. Arrays are initialized as shown in Figure 10-3.

Array Initialization List

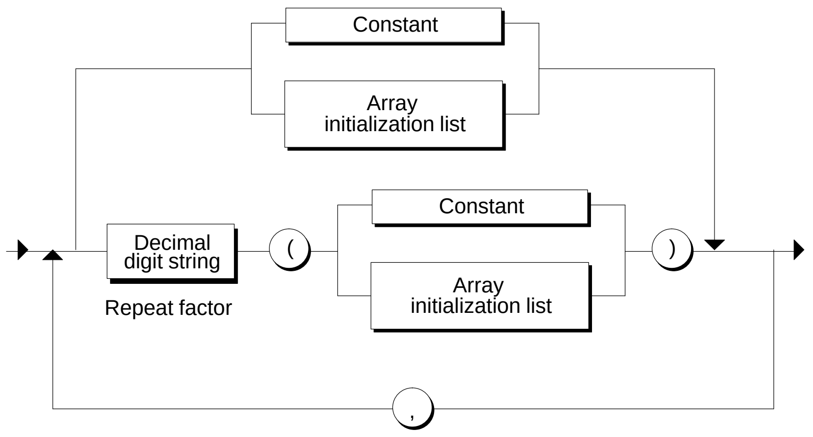
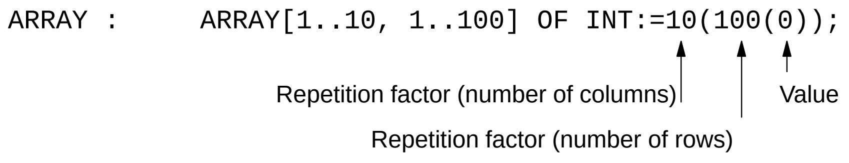


Figure 10-3 Syntax of an Array Initialization List



```
ARRAY :     ARRAY[1..10, 1..100] OF INT:=10(100(0));
```

Repetition factor (number of columns)
Value
Repetition factor (number of rows)

Examples:

Example 10-1 below illustrates the initialization of a static variable.

```
VAR
      INDEX1: INT:= 3;
END_VAR
```

Example 10-1 Initialization of Static Variables

Initialization of a two-dimensional array is shown in Example 10-2. If you wish to declare the following data structure in SCL and assign it the name `CONTROLLER`, you do so as follows:

-54	736	-83	77
-1289	10362	385	2
60	-37	-7	103
60	60	60	60

```
VAR
CONTROLLER:
ARRAY [1..4, 1..4] OF INT:=  -54,   736, -83, 77,
                           -1289, 10362,   385,  2,
                              60,   -37,    -7, 103,
                           4(60);

END_VAR
```

Example 10-2 Array initialization

An example of initialization of a structure is shown in Example 10-3:

```
VAR
GENERATOR:STRUCT
   DATA:   REAL      := 100.5;
   A1:     INT       := 10;
   A2:     STRING[6]:= ’FACTOR’;
   A3:     ARRAY[1..12] OF REAL:= 12(100.0);
END_STRUCT
END_VAR
```

Example 10-3 Structure Intialization

10.4 Instance Declaration

Summary

Apart from the elementary, complex and user-defined variables already mentioned, you can also declare variables of the type FB or SFB in the declaration section of function blocks. Such variables are called **local instances** of the FB or SFB.

The local instance data is stored in the instance data block of the calling function block.

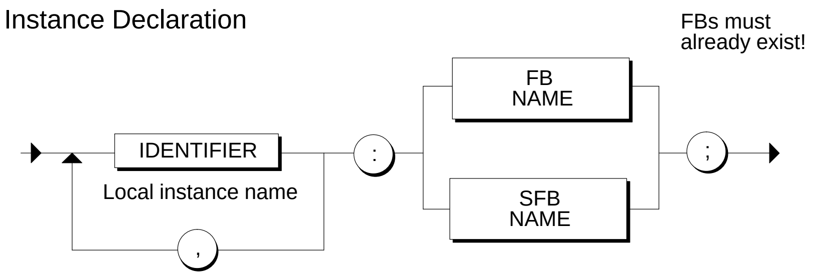


Figure 10-4 Syntax of Instance Declaration

Examples: The following are examples of correct syntax according to the syntax diagram in Figure 10-4:

```
Supply1     : FB10;
Supply2,Supply3,Supply4 : FB100;
Motor1      : Motor ;
```

// Motor is a symbol declared in the symbol table.

Symbol	Address	Data Type
MOTOR	FB20	FB20

Figure 10-5 Corresponding Symbol Table in STEP 7

Initialization

Local instance-specific initialization is not possible.

10.5 Static Variables

Overview

Static variables are local variables whose value is retained throughout all block cycles. They are used to store the values of a function block and are contained in a corresponding instance data block.

Static Variable Block

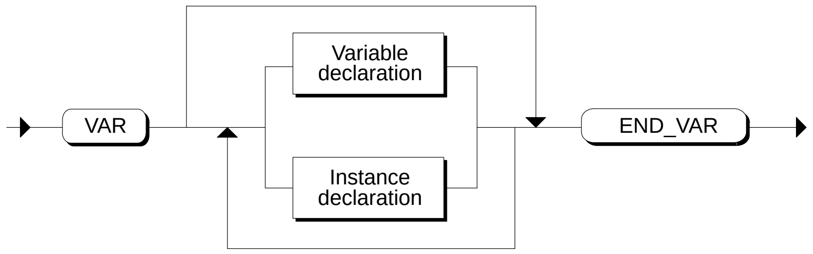


Figure 10-6 Syntax of a Static Variable Block

Declaration Subsection
VAR
END_VAR

The declaration subsection is a component of the FB declaration section. In it you can:

- Declare variable names and data types in a variable declaration with initialization if required (see Section 10.2)
- Insert existing variable declarations using an instance declaration (see Section 10.4).

After compilation, this subsection together with the subsections for the block parameters determines the structure of the assigned instance data block.

Example

Example 10-4 below illustrates the declaration of static variables.

```
VAR
  PASS            :INT;
  MEASBAND        :ARRAY[1..10] OF REAL;
  SWITCH          :BOOL;
  MOTOR_1,Motor_2 :FB100; // Instance declaration

END_VAR
```

Example 10-4 Declaration of Static Variables

Access

The variables are accessed from the code section as follows:

- **Internal access**: that is, from the code section of the function block in whose declaration section the variable is declared. This is explained in Chapter 14 (Value Assignments).
- **External access via the instance DB**: by way of the indexed variable *DBx.variable.* DBx is the data block name.

10.6 Temporary Variables

Overview

Temporary variables belong to a logic block locally and do **not** occupy any static memory area. They are located in the stack of the CPU. The value only exists while a block is being processed. Temporary variables **cannot** be accessed outside the block in which they are declared.

You should declare data as temporary data if you only require it to record interim results during the processing of your OB, FB or FC.

Temporary Variable Subsection

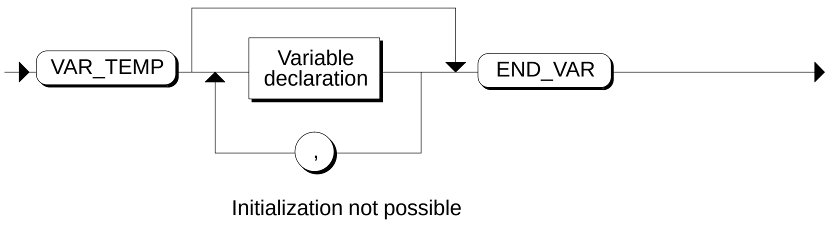


Figure 10-7 Syntax of a Temporary Variable Subsection

Declaration Subsection
VAR_TEMP
END_VAR

The declaration subsection is a component of an FB, FC, or OB. It is used to declare variable names and data types within the declaration section (see Section 10.2).

When an OB, FB or FC is first executed, the value of the temporary data has not been defined. Initialization is not possible.

Example

Example 10-5 below illustrates the declaration of block-temporary variables.

```
VAR_TEMP
       BUFFER_1      :ARRAY [1..10] OF INT;
       AUX1,AUX2     :REAL;
END_VAR
```

Example 10-5 Declaration of Block-Temporary Variables

Access

A variable is always accessed from the code section of the logic block in whose declaration section the **variable** is declared (**internal access**), see Chapter 14, Value Assignments.

10.7 Block Parameters

Overview

Block parameters are formal parameters of a function block or a function. When the function block or function is called, the actual parameters replace the formal parameters, thus forming a mechanism for exchange of data between the called block and the calling block.

- Formal input parameters are assigned the actual input values (inward flow of data)
- Formal output parameters are used to transfer output values (outward flow of data)
- Formal in/out parameters have the function of both an input and an output parameter.

For more detailed information about the use of parameters and the associated exchange of data, refer to Chapter 16.

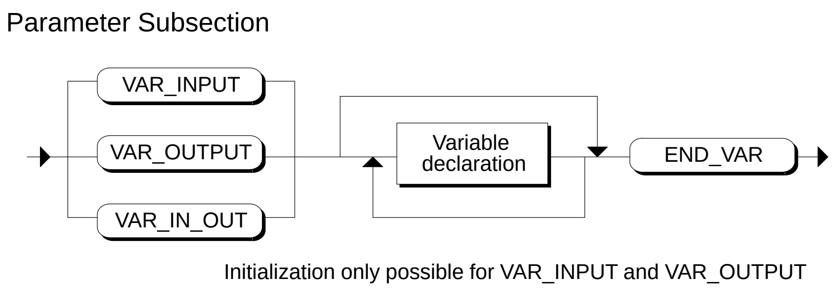


Figure 10-8 Syntax of Parameter Subsection

Declaration Subsection
`VAR_INPUT`
`VAR_OUTPUT`
`VAR_IN_OUT`

The declaration subsection is a component of an FB or FC. In it, the variable name and assigned data type are specified within the variable declaration see Section 10.2.

After compilation of an FB, these subsections together with the subsection delimited by `VAR and END_VAR` determine the structure of the assigned instance data block.

Example

Example 10-6 below illustrates the declaration of a parameter:

```
VAR_INPUT           //Input parameter
   CONTROLLER :DWORD;
   TIME       :TIME_OF_DAY;
END_VAR
```

```
VAR_OUTPUT          //Output parameter
   SETPOINTS: ARRAY [1..10] of INT;
END_VAR
```

```
VAR_IN_OUT          //In/out parameter
   EINSTELLUNG: INT;
END_VAR
```

Example 10-6 Declaration of Parameters

Access

Block parameters are accessed from the code section of a logic block as follows:

- **Internal access**: that is, from the code section of the block in whose declaration section the **parameter** is declared. This is explained in Chapter 14 (Value Assignments) and Chapter 13 (Expressions, Operators and Addresses).
- **External access by way of instance data block**. You can access block parameters of function blocks via the assigned instance DB (see Section 14.8).

10.8 Flags (OK Flag)

Description

The OK flag is used to indicate the correct or incorrect execution of a block. It is a global variable of the type BOOL identified by the keyword ”OK”.

If an error occurs when a block statement is being executed (for example overflow during multiplication), the OK flag is set to FALSE. When the block is quit, the value of the OK flag is saved in the implicitly defined output parameter ENO (Section 16.4) and can thus be read by the calling block.

When the block is first called, the OK flag has the value TRUE. It can be read or set to TRUE / FALSE at any point in the block by means of SCL statements.

Declaration

The OK flag is a system variable. Declaration is not necessary. However, you do have to select the compiler option ”OK Flag” before compiling the source file if you wish to use the OK flag in your application program.

Example

Example 10-7 below illustrates the use of the OK flag.

```
       // Set OK variable to TRUE
       // in order to be able to check
       // whether the operation below
       // is performed successfully
OK: = TRUE;
SUM: = SUM + IN;
IF OK THEN
       // Addition completed successfully
       :
       :
ELSE   // Addition not completed successfully
       :
END_IF;
```

Example 10-7 Use of the OK Variable

Declaring Constants and Jump Labels 11

Introduction

Constants are data elements that have a fixed value which can not be altered while the program is running. If the value of a constant is expressed by its format, it is termed a **literal constant.**

You do not have to declare constants. However, you have the option of assigning symbolic names for constants in the declaration section.

Jump labels represent the names of jump command destinations within the code section of the logic block.

Symbolic names of constants and jump labels are declared separately in their own declaration subsections.

Chapter Overview

11.1 Constants

Use of Constants

In value assignments and expressions, constants are also used in addition to variables and block parameters. Constants can be used as literal constants or they can have a symbolic name.

Declaration of Symbolic Names

Symbolic names for constants are declared within the `CONST` declaration subsection in the declaration section of your logic block (see Section 8.4).

Constant Subsection

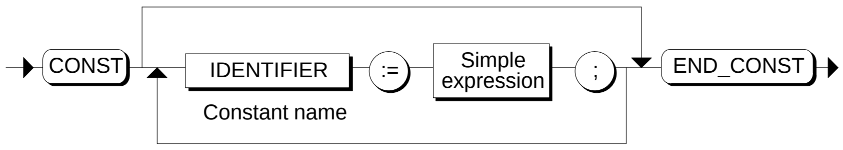


Figure 11-1 Syntax of Constant Subsection

’Simple expression’ in this case refers to mathematical expressions in which you can use using the basic operations +, –, *, /, DIV and MOD.

Example

Example 11-1 below illustrates the declaration of symbolic names.

```
CONST
 Figure     := 10 ;
 TIME1      := TIME#1D_1H_10M_22S.2MS ;
 NAME       := ’SIEMENS’ ;
 FIG2       := 2 * 5 + 10 * 4 ;
 FIG3       := 3 + NUMBER2 ;
 END_CONST
```

Example 11-1 Declaration of Symbolic Constants

Formats

SCL provides a number of different formats for entering or displaying constants. Those formats are known as literals. The sections which follow deal with the various types of literal.

11.2 Literals

Definition

A literal is a syntactical format for determining the type of a constant. There are the following groups of literals:

- Numeric literals
- Character literals
- Times

There is a specific format for the value of a constant according to its data type and data format.

15	VALUE 15	as integer in decimal notation
2#1111	Value 15	as integer in binary notation
16#F	Value 15	as integer in hexadecimal notation

Literal with different formats for the value 15

Assigning Data Types to Constants

A constant is assigned the data type whose value range is just sufficient to accommodate the constant without loss of data. When using constants in an expression (for example, in a value assignment), the data type of the target variable must incorporate the value of the constant. If, for example, an integer literal is specified whose value exceeds the integer range, it is assumed that it is a double integer. The compiler returns an error message if you assign this value to a variable of the type Integer.

11.3 Formats for Integer and Real Number Literals

Overview

SCL provides the following formats for numerical values:

- Integer literals for whole number values
- Real number literals for floating point numbers

In both of the above literals, you use a string of digits which must conform to the structure shown in Figure 11-2. This string of digits is referred to simply as a **decimal digit string** in the syntax diagrams below.

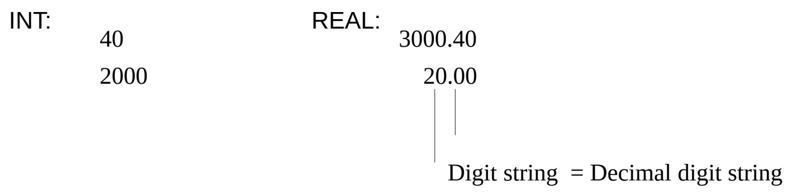


Figure 11-2 Digit String in a Literal

The decimal number in a literal consists of a string of digits which may also be separated by underscore characters. The underscores are used to improve readability in the case of long numbers.

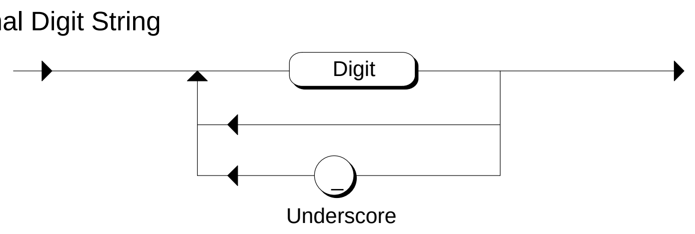


Figure 11-3 Syntax of Decimal Digit Strings

Below are some examples of valid formats for decimal digit strings in literals:

```
1000
1_120_200
666_999_400_311
```

Integer Literals

Integer literals are whole numbers. Depending on their length, they can be assigned in the SCL program to variables of the following data types:

`BOOL, BYTE, INT, DINT, WORD, DWORD.`

Figure 11-4 shows the syntax of an integer literal.

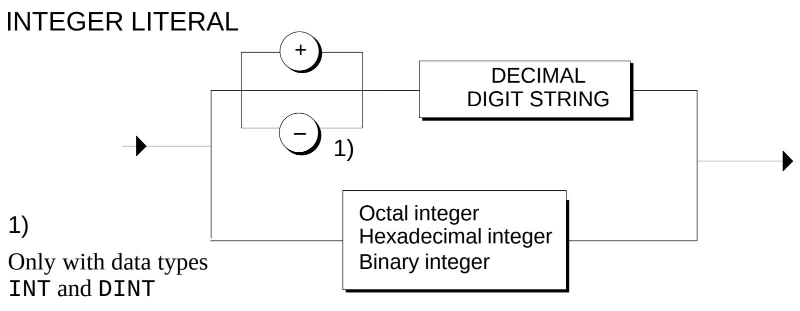


Figure 11-4 Syntax of an Integer Literal

Below are some examples of permissible formats for decimal digit strings in integer literals:

```
1000
+1_120_200
–666_999_400_311
```

Binary/Octal/Hexadecimal Values

You can specify an integer literal in a numeric system other than the decimal system by using the prefixes **2#, 8# or 16#** followed by the number in the notation of the selected system. You can use the underscore character within a number to make longer numbers easier to read.

The general format for an integer literal is illustrated in Figure 11-5 using the example of a digit string for an octal number.

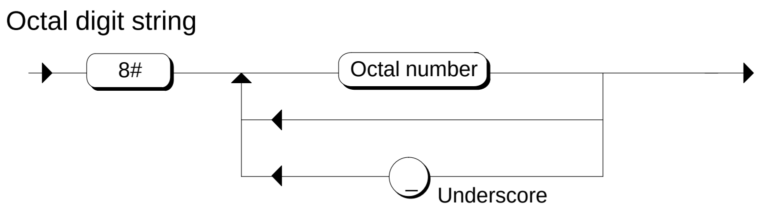


Figure 11-5 Syntax of an Octal Digit String

Below are some examples of permissible formats for integer literals:

```
Wert_2:=2#0101;// Binary number, decimal value 5
Wert_3:=8#17;  // Octal number, decimal value 15
Wert_4:=16#F;  // Hexadecimal number, decimal
               // value 15
```

Real Number Literals

Real number literals are values with decimal places. They can be assigned to variables of the data type REAL. The use of a plus or minus sign is optional. If no sign is specified, the number is assumed to be positive. Figure 11-7 shows the syntax for specifying an exponent. Figure 11-6 shows the syntax for a real number:

REAL NUMBER LITERAL

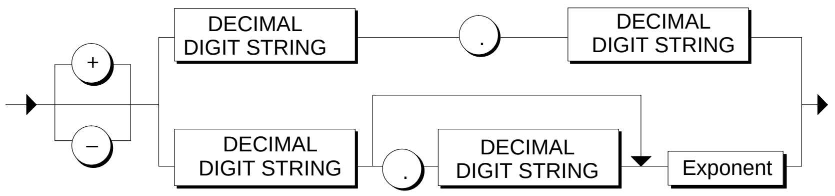


Figure 11-6 Syntax of a Real Number Literal

With exponential format, you can use an exponent to specify floating point numbers. The exponent is indicated by preceding the integer with the letter “E” or “e”, following a decimal digit string. Figure 11-7 shows the syntax for entering an exponent.

Exponent

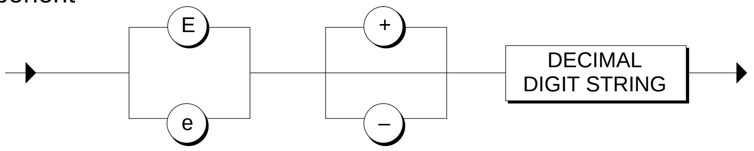


Figure 11-7 Exponent Syntax

Example:

The value 3 x 10 ^{10} can be represented by the following real numbers in SCL:

```
3.0E+10   3.0E10        3e+10       3E10
0.3E+11   0.3e11        30.0E+9     30e9
```

Examples

Example 11-2 summarizes the various alternatives once again:

```
// Integer literals
NUMBER1:= 10 ;
NUMBER2:= 2#1010 ;
NUMBER3:= 16#1A2B ;
// Real number literals
NUMBER4:= -3.4 ;
NUMBER5:= 4e2 ;
NUMBER6:= 40_123E10;
```

Example 11-2 Numeric Literals

11.4 Formats for Character and String Literals

Summary

SCL also provides the facility for entering and processing text data, for example a character string to be displayed as a message.

Calculations can not be performed on character literals, which means that character literals can **not** be used in expressions. A distinction is made between

- character literals, that is, single characters, and
- string literals, that is, a character string of up to 254 separate characters.

Character Literals (Single Characters)

A character literal, as shown in Figure 11-8, consists of a single character only. That character is enclosed in single inverted commas (′).

CHARACTER LITERAL

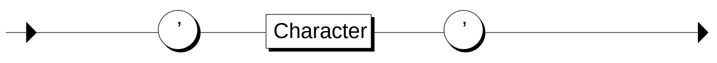


Figure 11-8 Character Literal Syntax

Example:

```
Char_1:=’B’;            // Letter B
```

String Literals

A string literal is a string of up to 254 characters (letters, numbers and special characters) enclosed in single inverted commas (′). Both upper and lower case letters can be used.

STRING LITERAL

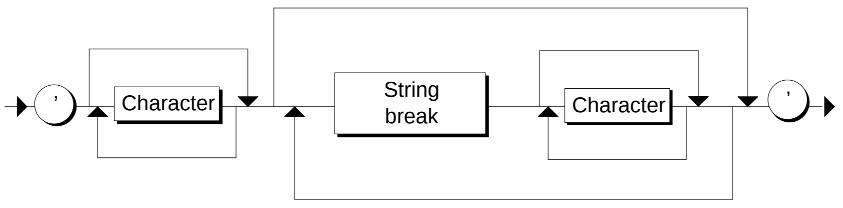


Figure 11-9 String Literal Syntax

The following are some examples of permissible string literals:

```
’RED’    ’7500 Karlsruhe’         ’270–32–3456’
’DM19.95’ ’The correct answer is:’
```

Please note that when assigning a string literal to a string variable, the maximum number of characters can be limited to less than 254.

The value assignment
`TEXT:STRING[20]:='SIEMENS␣KARLSRUHE␣Rheinbrückenstr.'`

will result in an error message and the information stored in the variable 'TEXT' will be as follows:

`'SIEMENS␣KARLSRUHE␣Rh'`

Special formatting characters, the inverted comma (') and the $ sign can be entered using the character $. A string literal can contain any number of **breaks**.

String Breaks

A string is located either on a single line of an SCL block or is spread over several lines by means of special identifiers. The identifier `'$>'` is used to break a string and the identifier `'$<'` to continue it on a subsequent line.

```
TEXT:STRING[20]:='The FB$>//Preliminary version
$<converts';
```

The space between the break and the continuation identifiers may extend over a number of lines and can only contain comments or spaces. A string literal can be broken and continued in this way (see also Figure 11-10) any number of times.

String Break Syntax

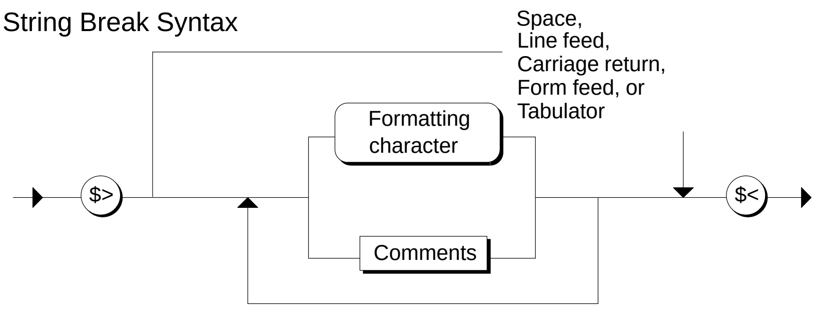


Figure 11-10 String Break Syntax

Printable Characters

All characters of the extended ASCII character set are permissible in a character or string literals. Special formatting characters and characters that cannot be directly represented (' and $) in a string can be entered using the alignment symbol $.

Characters

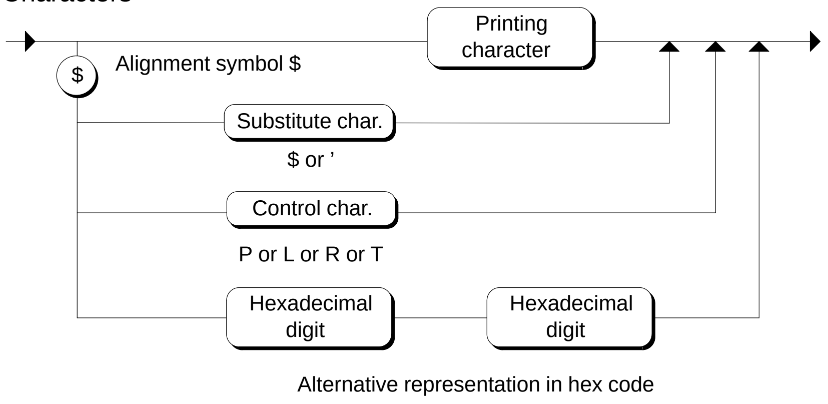


Figure 11-11 Character Syntax

Non-Printable Characters

In a character literal, you can also use all non-printing characters of the extended ASCII character set. To do this, you specify the substitute representation in hexadecimal code.

You type in an ASCII character in the form **$hh**, where hh represents the value of the ASCII character in hexadecimal notation.

Example:

```
CHAR_A      :=’$41’; //Represents the letter ’A’
Space       :=’$20’;.//Represents the character ␣
```

For more details of substitute and control characters, refer to Appendix A.

Examples

The following examples illustrate the formulation of character literals:

```
// Character literal
Char:= ’S’ ;

// String literal:
NAME:= ’SIEMENS’ ;

// Breaking a string literal:
MESSAGE1:= ’MOTOR $>
$< Control’ ;

 // String in hexadecimal notation:
 MESSAGE1:= ’$41$4E’ (*Character string AN*);
```

Example 11-3 Character Literals

11.5 Formats for Times

Different Types of Time Data

SCL provides various fomats for entering times and dates. The following types of time data are possible:

```
Date
Time period
Time of day
Date and time
```

Date

A date is introduced by the prefix DATE# or D# as shown in Figure 11-12.

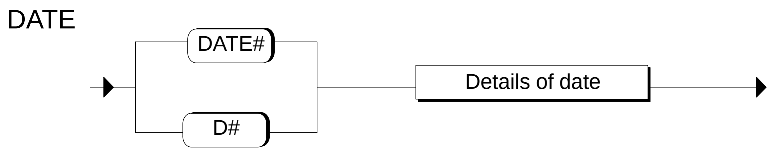


Figure 11-12 Date Syntax

The **date** is specified by means of integers for the year (4 digits), the month and the day, separated by hyphens.

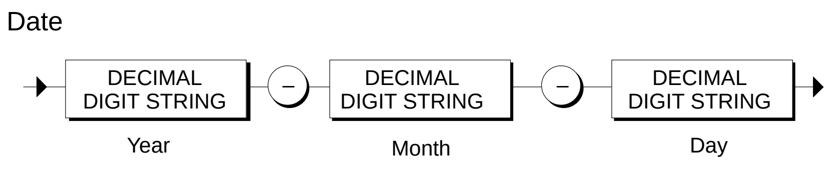


Figure 11-13 Date Entry Syntax

The following are examples of valid dates:

```
// Date
TIMEVARIABLE1:= DATE#1995-11-11;
TIMEVARIABLE2:= D#1995-05-05;
```

Time Period

A time period is introduced as shown in Figure 11-14 by the prefix TIME# or T#. The time period can be expressed in two possible ways:

- in simple time format
- in complex time format

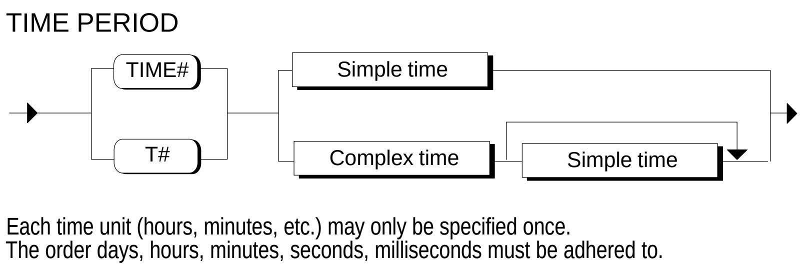

TIME PERIOD
TIME#
T#
Simple time
Complex time
Simple time
- Each time unit (hours, minutes, etc.) may only be specified once.
- The order days, hours, minutes, seconds, milliseconds must be adhered to.

Figure 11-14 Time Period Syntax

You use the **simple time format** if the time period has to be expressed in a single time unit (either days, hours, minutes, seconds or milliseconds).

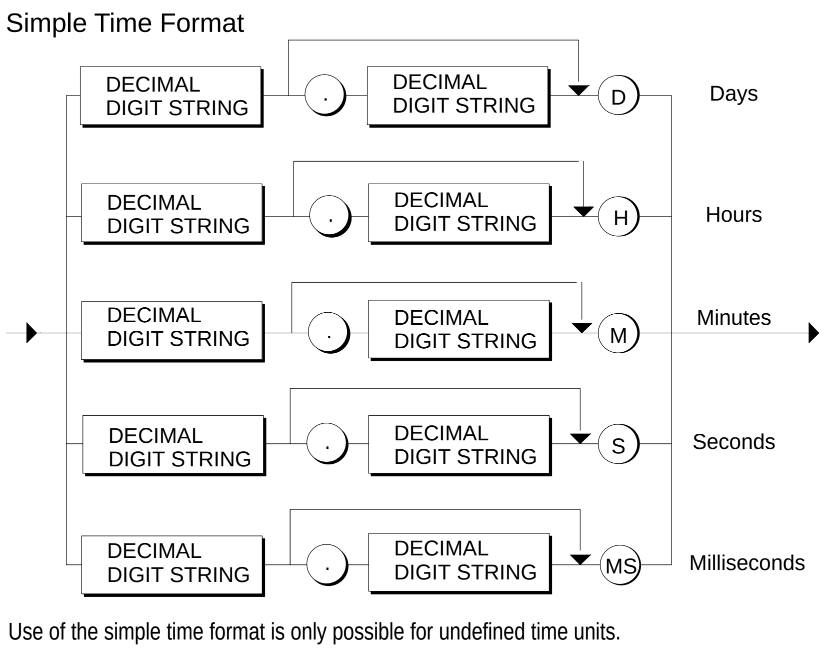


Figure 11-15 Syntax of Simple Time Format

Examples

The following are valid simple times:

```
TIME#20.5D                for 20,5  Days
TIME#45.12M               for 45,12 Minutes
T#300MS                   for 300 Milliseconds
```

The **complex time format** is used when you have to express the time period as a combination of more than one time unit (as a number of days, hours, minutes, seconds and milliseconds, see Figure 11-16). Individual components can be omitted. However, at least one time unit must be specified.

Complex Time Format

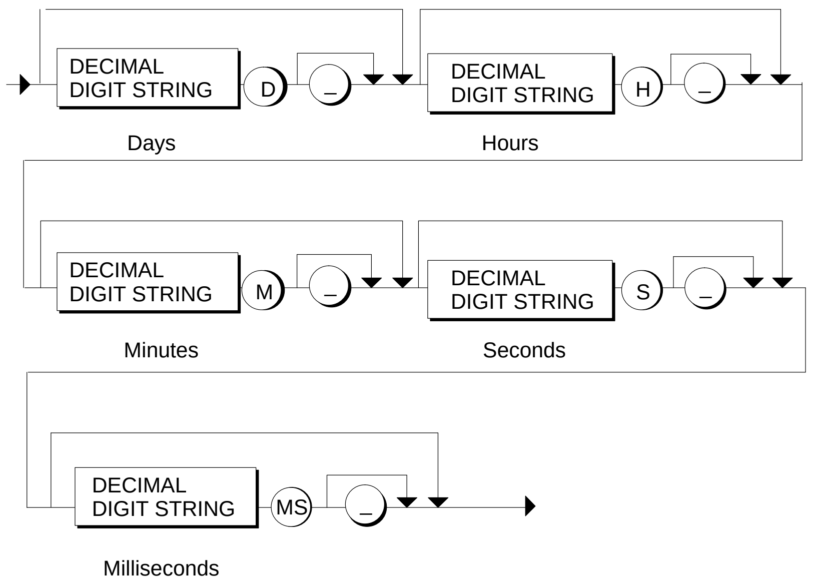


Figure 11-16 Complex Time Format Syntax

The following are valid complex times:

```
TIME#20D or TIME#20D_12H

TIME#20D_10H_25M_10s

TIME#200S_20MS
```

Time of Day

A time of day is introduced by the prefix TIME_OF_DAY# or TOD# as shown in Figure 11-17.

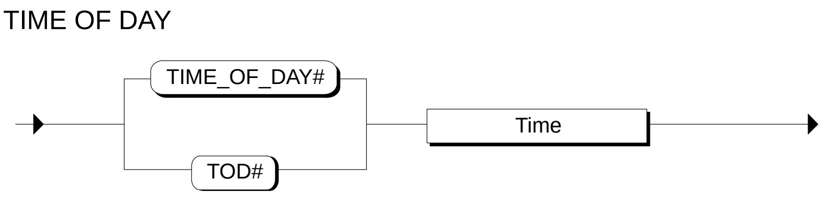


Figure 11-17 Time-of-Day Syntax

A time of day is indicated by specifying the number of hours, minutes and seconds separated by colons. Specifying the number of milliseconds is optional. The milliseconds are separated from the other numbers by a decimal point. Figure 11-18 shows the syntax for a time of day.

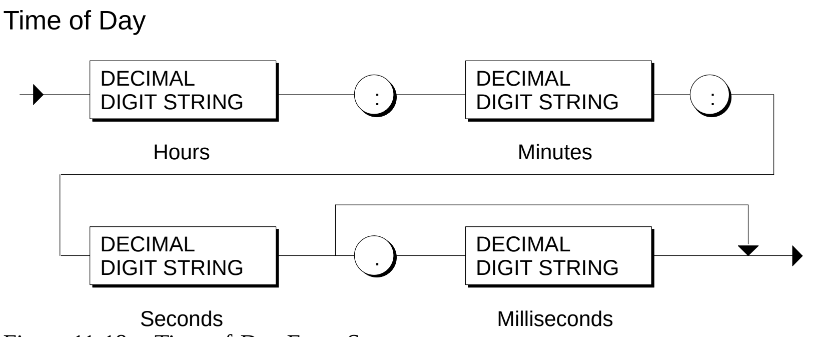


Figure 11-18 Time-of-Day Entry Syntax

The following are valid times of day:

```
//Time of day
TIME1:= TIME_OF_DAY#12:12:12.2;
TIME2:= TOD#11:11:11.7.200;
```

Date and Time

A date and time is introduced as shown in Fig. 11-19 by the prefix `DATE_AND_TIME#` or `DT#`. It is a literal made up of a date and a time of day.

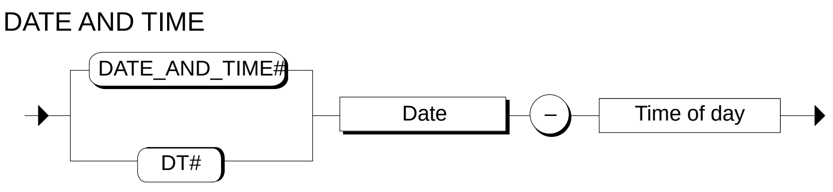


Figure 11-19 Date and Time Syntax

The example below illustrates the use of date and time:

```
// Time of day
TIME1:= DATE_AND_TIME#1995-01-01–12:12:12.2;
TIME2:= DT#1995-02-02–11:11:11;
```

11.6 Jump Labels

Description

Jump labels are used to define the destination of a GOTO statement (see Section 11-4).

Declaring Jump Labels

Jump labels are declared in the declaration section of a logic block together with their symbolic names (see Section 8.4) as follows:

Jump Label Subsection

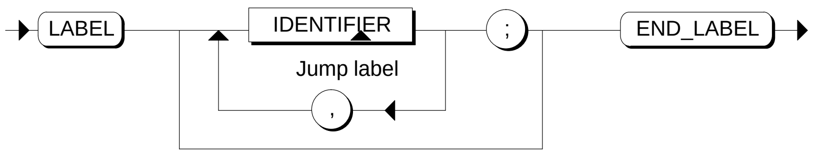


Figure 11-20 Syntax of a Jump Label Subsection

Example

The following example illustrates the declaration of jump labels:

```
LABEL
      LABEL1, LABEL2, LABEL3;
END_LABEL;
```

Example 11-4 Jump Labels

Declaring Global Data 12

Introduction

Global data can be used by any logic block (FC, FB or OB). These data can be accessed absolutely or symbolically. This chapter introduces you to the individual data areas and explains how the data can be accessed.

Chapter Overview

12.1 Overview

Global Data

In SCL you have the facility of accessing global data. There are two types of global data as follows:

- **CPU Memory Areas**

  These memory areas represent system data such as inputs, outputs and bit memory (see Section 7.5). The number of memory areas available is determined by your CPU.

- **Global User Data in the form of Loadable Data Blocks**

  These data areas are contained within data blocks. In order to be able to use them you must first have created the data blocks and declared the data within them. In the case of instance data blocks, they are derived from function blocks and automatically generated.

Types of Access

Global data can be accessed in the following ways:

- **absolute:** via address identifier and absolute address
- **symbolic:** via a symbol previously defined in the symbol table (see **/231/**).
- **indexed:** via address identifier and array index
- **structured:** via a variable

Table 12-1 Use of Types of Access to Global Data

Type of Access	CPU Memory Areas	Global User Data
absolute	yes	yes
symbolic	yes	yes
indexed	yes	yes
structured	no	yes

12.2 CPU Memory Areas

Definition

CPU memory areas are system data areas. For this reason, you do not have to declare them in your logic block.

Different Areas of Memory

Each CPU provides the following memory areas together with a separate address area for each:

- Inputs/outputs in the image memory
- Peripheral inputs/outputs
- Bit memory
- Timers, counters (see Chapter 17)

Syntax for Access

A CPU area is accessed by means of a value assignment in the code section of a logic block (see Section 14.3) using either

- a simple accessing operation which can be specified in absolute or symbolic terms, or
- an indexed accessing operation.

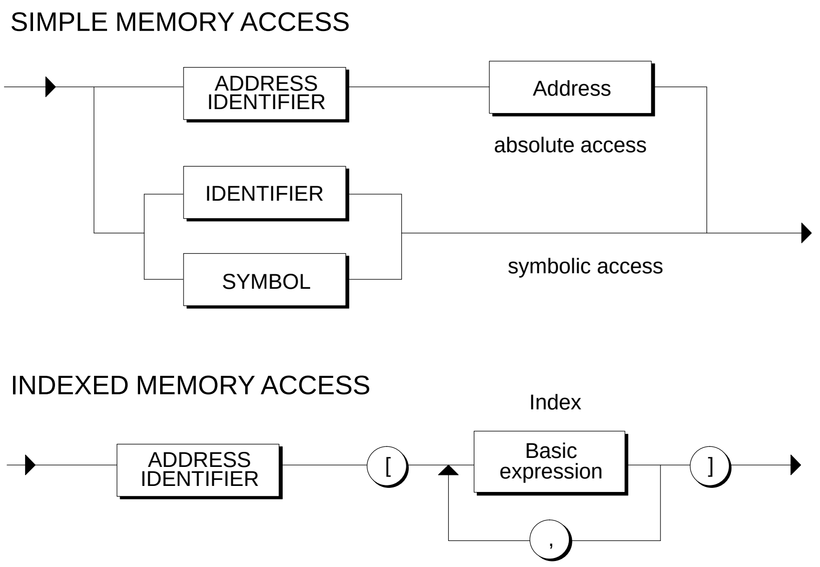
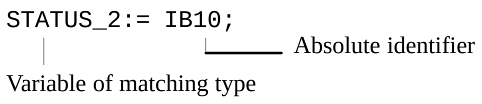


Figure 12-1 Syntax of Simple and Indexed Memory Access

12.3 Absolute Access to CPU Memory Areas

Basic Principle

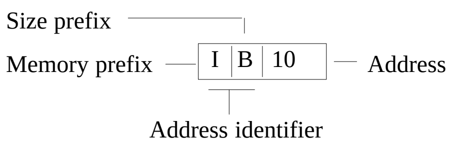
Absolute access to a memory area of the CPU is achieved by assigning an absolute identifier to a variable of the same type.



The absolute identifier indicates a memory area in the CPU. You specify this area by specifying the address identifier (in this case `IB`) followed by the address (in this case `10`).

Absolute Identifiers

The absolute identifier is made up of the address identifier, consisting of a memory and a size prefix, and an address.



Address Identifier

The combination of memory and size prefix makes the address identifier.

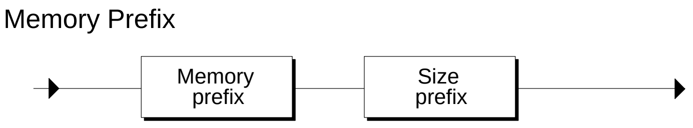


Figure 12-2 Syntax of Memory Address Identifiers

Memory Prefix

The memory prefix is used to specify the type of memory area to be accessed. Figure 12-3 below shows the various possible types of memory area. [1]

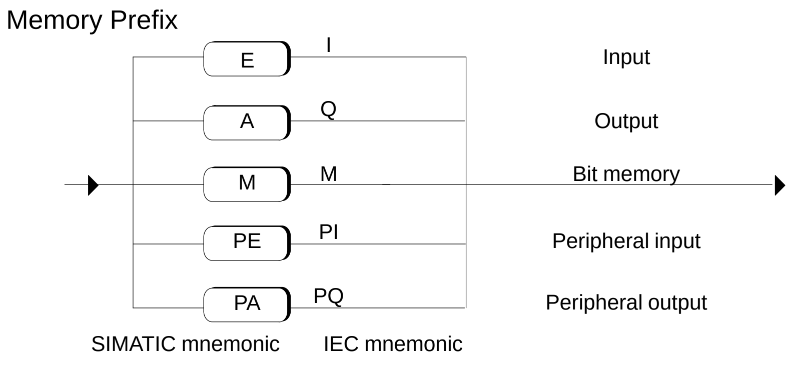


Figure 12-3 Syntax of Memory Prefix

1 *Depending on the language set in the SIMATIC Manager, either the SIMATIC or the IEC address identifiers have a reserved meaning. You can set the language and the mnemonics separately in the SIMATIC Manager.*

Size Prefix

The size prefix is used to specify the length or the type of the memory area (for example, a byte or a word) to be read from the peripheral I/Os. You can, for example read a byte or a word. Using the size prefix is optional if only one bit is specified. Figure 12-4 shows the syntax:

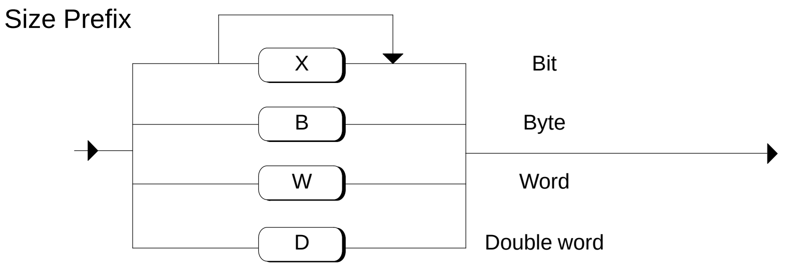


Figure 12-4 Syntax of Size Prefix

Address

When specifying an address depending on which size prefix you have used, you specify an absolute address that identifies a specific bit, byte, word or double word. Only if you have specified ”Bit” as the size can you specify an additional bit address (see Figure 12-5). The first number refers to the byte address and the second to the bit address.

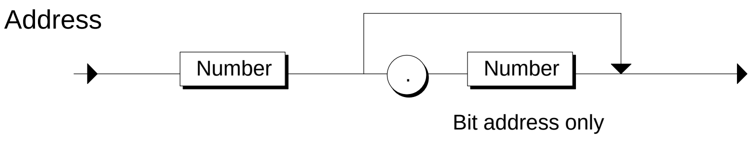


Figure 12-5 Syntax of Addresses

Examples

Below are some examples of absolute access:

```
STATUSBYTE    := IB10;
STATUS_3      := I1.1;
Measval       := IW20;
```

Example 12-1 Absolute Access

12.4 Symbolic Access to CPU Memory Areas

Basic Principle

When you program symbolically, instead of using the absolute address consisting of address identifier and address, you use a symbolic name to access a specific CPU memory area, as illustrated by the following examples:

Symbol	Absolute Address	Data Type	Comments
Motor_contact	I 1.7	BOOL	Contact switch 1 for Motor A 1
Input1	IW 10	INT	Status word
Input_byte1	IB 1	BYTE	Input byte
“Input 1.1”	I 1.1	BOOL	Photoelectric barrier
Meas_channels	MW 2	WORD	Meas. value buffer

The symbolic name is assigned to the address in your application program by creating a symbol table.

For the data type specification, you can use any elementary data type providing it can accept the specified data element size.

Accessing

You access a symbol, for example, by assigning a value to a variable of the same type using the symbol declared.

```
MEASVAL_1   := Motor_contact;
```

Creating the Symbol Table

The symbol table is created and values entered in it using STEP 7.

You can open the symbol table by means of the SIMATIC Manager or in SCL by selecting the menu command **Options** ▸ **Symbol Table**.

You can also import and edit symbol tables created with any text editor (for details, refer to /**231**/).

Examples

Below are some examples of symbolic access:

```
STATUSBYTE    := Input_byte1;
STATUS_3      := ”Input 1.1”;
Measval       := Meas_channels;
```

Example 12-2 Symbolic Access

12.5 Indexed Access to CPU Memory Areas

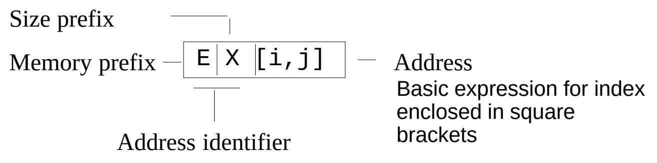
Basic Principle

You can also access memory areas of the CPU using an index. Compared with absolute addressing, the advantage of this method is that you can address dynamically using variable indices. For example, you can use the control variable of a FOR loop as the index.

Indexed access to a memory area is performed in a similar manner to the absolute method. It differs only by virtue of the address specification. Instead of the absolute address, an index is specified which can be a constant, a variable or a mathematical expression.

Absolute Identifier

The absolute identifier in the case of indexed access is made up of the address identifier and a basic expression for the indexing operation (as per Section 12.3).



Rules for Indexed Access

Indexing must conform to the following rules:

- When accessing data of the types `BYTE`, `WORD` or `DWORD`, you must use one index only. The index is interpreted as a byte address. The size of the data unit accessed is specified by the size prefix.
- When accessing data of the type `BOOL`, you must use two indices. The first index specifies the byte address, the second index the bit position within the byte.
- Each index must be a mathematical expression of the data type `INT.`

```
MEASWORD_1  := IW[COUNTER];

OUTMARKER   := I[BYTENUM, BITNUM];
```

Example 12-3 Indexed Access

12.6 Data Blocks

Summary

Within data blocks, you can store and process all the data for your application that is valid throughout the entire program or the entire project. Every logic block can read or write data from/to a data block.

Declaration

The syntax for the structure of data blocks is explained in Chapter 8. You should distinguish between two sorts of data block as follows:

- Data Blocks
- Instance data blocks

Accessing Data Blocks

The data in any data block can always be accessed in any of the following ways:

- Simple or absolute
- Indexed
- Structured

Figure 12-6 below summarizes the methods of access.

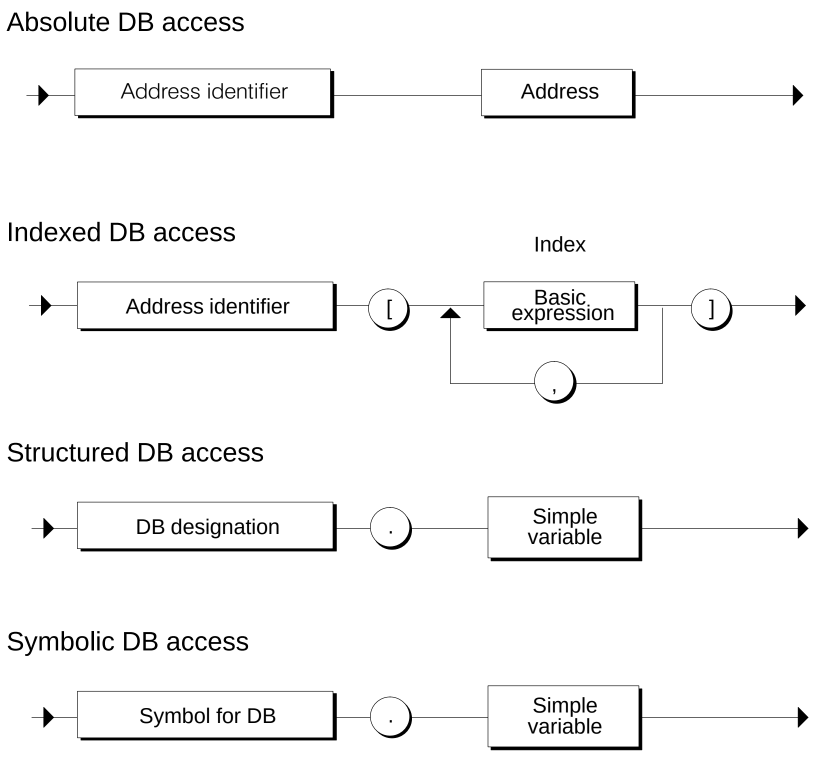
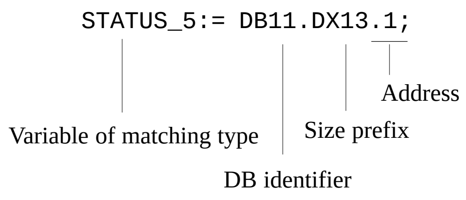


Figure 12-6 Syntax of Methods for Absolute, Indexed and Structured DB Access

12.7 Absolute Access to Data Blocks

Basic Principle

Absolute access to a data block is effected by assigning a value to a variable of a matching type in the same way as for CPU memory areas. You first specify the DB identifier followed by the keyword ″D″ and the size prefix (for example X for BIT) and the byte address (for example 13.1).



Accessing

Accessing is performed as shown in Figure 12-7 by specifying the DB identifier together with the size prefix and the address.

Absolute DB Access

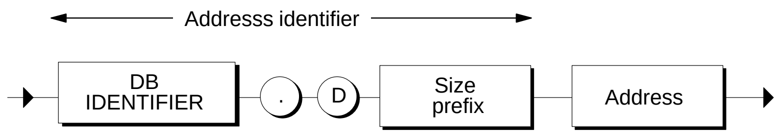


Figure 12-7 Syntax ofAbsolute DB Access

Size Prefix

Specifies the size of the memory area in the data block to be addressed; for example, one byte or one word. Specifying the size prefix is optional if you specify a bit address. Figure 12-8 shows the syntax for the size prefix.

Size Prefix

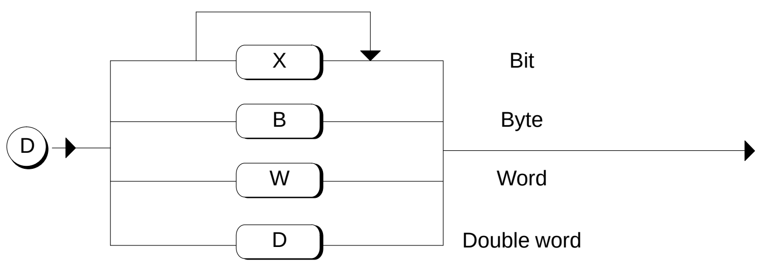


Figure 12-8 Syntax of Size Prefix

Address

When specifying the address as shown in Figure 12-9, you specify an absolute address that identifies a specific bit, byte, word or double word depending on the size prefix you have used. You can only specify an additional bit address if you have used the size prefix ”bit”. The first number represents the byte address and the second the bit address.

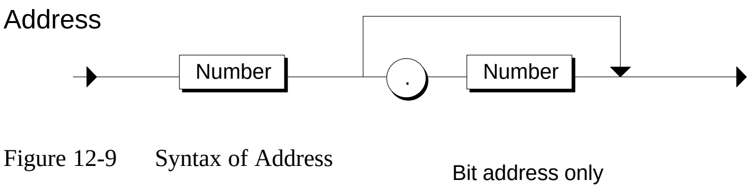


Figure 12-9 Syntax of Address

Examples

Below are some examples of data block accessing operations. The data block itself is specified in absolute terms in the first part and in symbolic terms in the second part.

```
STATUSBYTE    := DB101.DB10;
STATUS_3      := DB30.D1.1;
Measval       := DB25.DW20;

STATUSBYTE    := Statusdata.DB10;
STATUS_3      := ”New data” D1.1;
Measval       := Measdata.DW20;
STATUS_1      := WORD_TO_BLOCK_DB(INDEX).DW10;
```

Example 12-4 Absolute Access

12.8 Indexed Access to Data Blocks

Indexed Access

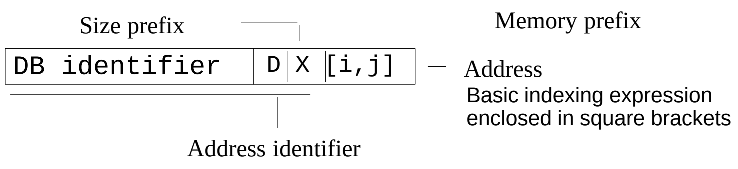
You can also access global data blocks using an index. Compared with absolute addressing, the advantage of this method is that by the use of variable indices you can address data dynamically. For example, you can use the control variable of a FOR loop as the index.

Indexed accessing of a data block is performed in a similar manner to absolute accessing. It differs only by virtue of the address.

Instead of the address, an index is specified which can be a constant, a variable or a mathematical expression.

Absolute Identifier

The absolute identifier in the case of indexed access is made up of the address identifer (as per Section 12.7) and a basic indexing expression.



Rules for Indexed Access

When using indices, the following rules must be adhered to:

- Each index must be a mathematical expression of the data type `INT`.
- When accessing data of the types `BYTE`, `WORD` or `DWORD`, you must use one index only. The index is interpreted as a byte address. The size of the data unit accessed is specified by the size prefix.
- When accessing data of the type `BOOL`, you must use two indices. The first index specifies the byte address, the second index the bit position within the byte.

```
STATUS_1:= DB11.DW[COUNTER];
STATUS_2:= DB12.DW[WNUM, BITNUM];

STATUS_1:= Database1.DW[COUNTER];
STATUS_2:= Database2.DW[WNUM, BITNUM];

STATUS_1:= WORD_TO_BLOCK_DB(INDEX).DW[COUNTER];
```

Example 12-5 Indexed Access

12.9 Structured Access to Data Blocks

Basic Principle

Structured access is effected by assigning a value to a variable of a matching type.

```
TIME_1:= DB11.TIME_OF_DAY ;
       |       |            |
       |       |            Simple variable
       |       DB identifier
  Variable of matching type
```

You identify the variable in the data block by specifying the DB name and the name of the simple variable separated by a full stop. The required syntax is detailed in Figure 12-6.

The simple variable stands for a variable to which you have assigned an elemetary or a complex data type in the declaration.

Examples

```
Declaration section of FB10:
VAR
Result:      STRUCT ERG1 : INT;
                    ERG2 : WORD;
             END_STRUCT
END_VAR
```

```
User-defined data type UDT1:
TYPE UDT1    STRUCT ERG1 : INT;
                    ERG2 : WORD;
             END_STRUCT
```

```
DB20 with user-defined data type:
DB20
UDT1
BEGIN ...
```

```
DB30 without user-defined data type:
DB30         STRUCT ERG1 : INT;
                    ERG2 : WORD;
             END_STRUCT
BEGIN ...
```

Example 12-6 Declaration of Data for Data Blocks

```
Function block showing accessing operations:
..
FB10.DB10();
ERGWORD_A    :=    DB10.Result.ERG2;
ERGWORD_B    :=    DB20.ERG2;
ERGWORD_C    :=    DB30.ERG2;
```

Example 12-7 Accessing Data Block Data

13 Expressions, Operators and Addresses

Introduction

An expression stands for a value that is calculated during compilation or when the program is running and consists of addresses (for example constants, variables or function values) and operators (for example *, /, + or –).

The data types of the addresses and the operators used determine the type of expression. SCL distinguishes:

- mathematical expressions
- exponential expressions
- comparative expressions
- logical expressions

Chapter Overview

13.1 Operators

Overview

Expressions consist of operators and addresses. Most SCL operators link two addresses and are therefore termed *binary* operators. The others work with only one address and are thus called *unary* operators.

Binary operators are placed between the addresses as in the expression 'A + B'. A unary operator always immediately precedes its address as in the expression '–B'.

The operator priority listed in Table 13-1 governs the order in which calculations are performed. '1' represents the highest priority.

Operator Classes

Table 13-1 Summary of Operators

Class	Operator	Symbol	Priority
Assignment operator *This operator assigns a value to a variable*	Assignment	: =	11
Mathematical Operators	Exponential	**	2
	Unary Operators		
	Unary plus	+	3
Required for mathematical calculations	Unary minus	-	3
	Basic Mathematical Operators		
	Multiplication Modulus Integer division Addition Subtraction	* MOD DIV + -	4 4 4 5 5
Comparative operators *These operators are required for formulating conditions*	Less than Greater than Less than or equal to Greater than or equal to Equal to Not equal to	< > <= >= = <>	6 6 6 6 7 7
Logical operators	Negation	NOT	3
	Basic Logical Operators		
These operators are required for logical expressions	And Exclusive or Or	AND or & XOR OR	8 9 10
Parentheses	(**Expression**)	( )	1

13.2 Syntax of Expressions

Overview

Expressions can be illustrated using the syntax diagram in Figure 13-1. Mathematical, logical and comparative expressions as well as exponential expressions have a number of special characteristics and are therefore treated individually in Sections 13.3 to 13.6.

Expression

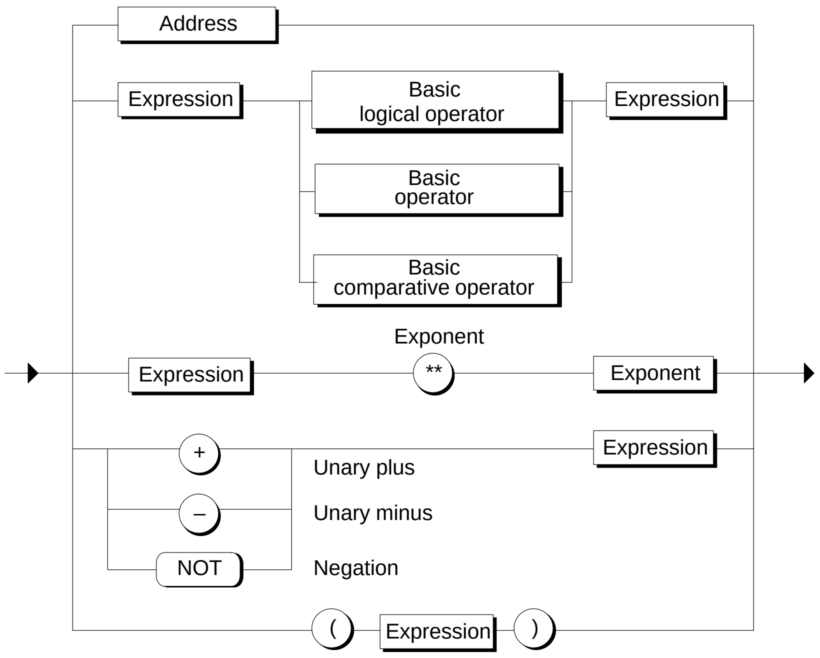


Figure 13-1 Syntax of Expressions

Result of an Expression

You can perform the following operations on the result of an expression:

- Assign it to a variable.
- Use it as the condition for a control instruction.
- Use it as a parameter for calling a function or a function block.

Sequence of Operations

The order in which the operations are performed is determined by:

- The priority of the operators involved
- The sequence from left to right
- The use of parentheses (if operators have the same priority).

Rules

Expressions are processed according to the following rules:

- An address between two operators of different priority is always attached to the higher-priority operator.
- Operators with the same priority are processed from left to right.
- Placing a minus sign before an identifier is the same as multiplying it by –1.
- Mathematical operators must not follow each other directly. The expression a * – b is invalid, whereas a * (–b) is permitted.
- Parentheses can be used to overcome operator priority, in other words parentheses have the highest priority.
- Expressions in parentheses are considered as a single address and always processed first.
- The number of left parentheses must match the number of right parentheses.
- Mathematical operators cannot be used with characters or logical data. Expressions such as ‘A’ +‘B’ and (n<=0) + (n<0) are thus not permissible.

Examples

Below are some examples of the structure of the various expressions:

```
IB10                       // address
A1 AND (A2)                // Logical expression
(A3) < (A4)                // Comparative expression
3+3*4/2                    // Mathematical expression


MEASVAL**2                 // Exponential expression
(DIFFERENCE)**DB10.EXPONENT
(SUM)**FC100(..)           // Exponential
                              expression
```

Example 13-1 Various Expressions

13.2.1 Addresses

Definition

Addresses are objects which can be used to construct expressions. The syntax of addresses is illustrated in Figure 13-2.

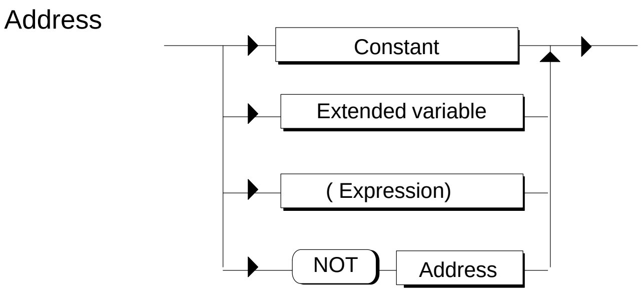


Figure 13-2 Syntax of Addresses

Constants

Constants can be a numerical value or a symbolic name or a character string.

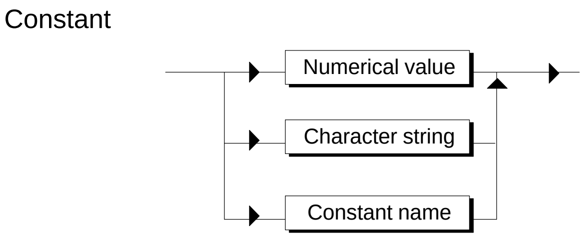


Figure 13-3 Syntax of Constants

The following are examples of valid constants:

```
4_711
4711
30.0
’CHARACTER’
FACTOR
```

Extended Variables

An extended variable is a generic term for a series of variables which are dealt with in more detail in Chapter 14.

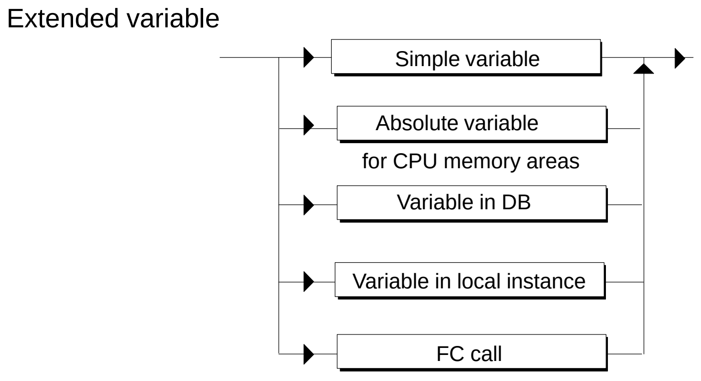


Figure 13-4 Syntax of Extended Variables

Examples of Extended Variables

The following are examples of valid variables:

```
SETPOINT          Simple variable
IW10              Absolute variable
I100.5            Absolute variable
DB100.DW[INDEX]   Variable in DB
MOTOR.SPEED       Variable in local instance
SQR(20)           Standard function
FC192 (SETPOINT)  Function call
```

Example 13-2 Extended variables in expressions

Note

In the case of a function call, the calculated result, the return value, is inserted in the expression in place of the function name. For that reason, VOID functions which do not give a return value are **not** permissible as addresses in an expression.

13.3 Mathematical Expressions

Definition

A mathematical expression is an expression formed using mathematical operators. These expressions allow numeric data types to be processed.

Basic mathematical operator

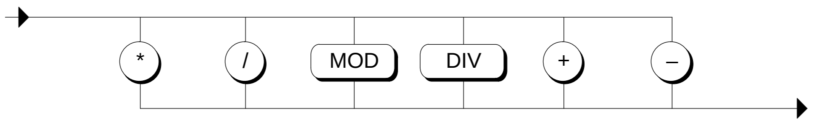


Figure 13-5 Syntax of Basic Mathematical Operators

Mathematical Operations

Table 13-2 below shows all the possible operations and indicates which type the result is assigned to depending on the operands. The abbreviations have the following meaning:

`ANY_INT`	for data types	`INT, DINT`
`ANY_NUM`	for data types	`ANY_INT and REAL`

Table 13-2 Mathematical Operators

Operation	Operator	1st Address	2nd Address	Result [1]	Priority
Exponent	**	ANY_NUM	INT	REAL	2
Unary plus	+	ANY_NUM	-	ANY_NUM	3
		TIME	-	TIME	
Unary minus	-	ANY_NUM	-	ANY_NUM	3
		TIME	-	TIME	
Multiplication	*	ANY_NUM	ANY_NUM	ANY_NUM	4
		TIME	ANY_INT	TIME	
Division	/	ANY_NUM	ANY_NUM	ANY_NUM	4
		TIME	ANY_INT	TIME	
Integer division	DIV	ANY_INT	ANY_INT	ANY_INT	4
		TIME	ANY_INT	TIME	
Modulus	MOD	ANY_INT	ANY_INT	ANY_INT	4
Addition	+	ANY_NUM	ANY_NUM	ANY_NUM	5
		TIME	TIME	TIME	
		TOD	TIME	TOD	
		DT	TIME	DT	
Subtraction	–	ANY_NUM	ANY_NUM	ANY_NUM	5
		TIME	TIME	TIME	
		TOD	TIME	TOD	
		DATE	DATE	TIME	
		TOD	TOD	TIME	
		DT	TIME	DT	
		DT	DT	TIME	

1) Remember that the result type is decided by the dominant address type.

Rules

The order in which operators are applied within a mathematical expression is based on their priority(see Table 13-2).

- It is advisable to place negative numbers in brackets for the sake of clarity even in cases where it is not necessary from a mathematical point of view.
- When dividing with two whole numbers of the type INT, the operators “DIV” and “/” produce the same result (see example 13-3).
- The division operators (‘/’, ‘MOD’ and ‘DIV’) require that the second address is not equal to zero.
- If one number is of the INT type (integer) and the other of the REAL type (real number), the result will always be of the REAL type.

Examples

The examples below illustrate the construction of mathematical expressions.

Let us assume that ‘i’ and ‘j’ are integer variables whose values are 11 and –3 respectively. Example 13-3 shows some integer expressions and their corresponding values.

```
Expression    Value
   i + j        8
   i – j       14
   i * j      –33
   i DIV j     –3
   i MOD j      2
   i/j         –3
```

Example 13-3 Mathematical Expressions

Let us assume that i and j are integer variables whose values are 3 and –5 respectively. In Example 13-4 the result of the mathematical expression shown, (that is, the integer value 7) is assigned to the variable VALUE.

```
VALUE:= i + i * 4 / 2 - (7+i) / (-j) ;
```

Example 13-4 Mathematical Expression

13.4 Exponential Expressions

Overview

Figure 13-6 illustrates the construction of the exponent in an exponential expression (see also Section 13.2). Remember, in particular, that the exponent expression can also be formed with extended variables.

Exponent

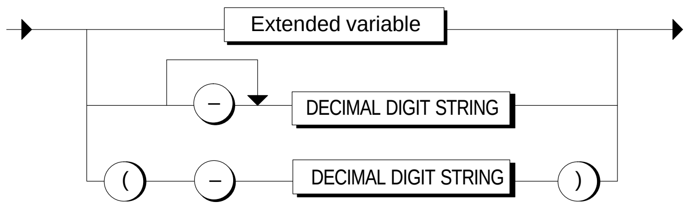


Figure 13-6 Syntax of an Exponent

```
MEASVAL**2               // Exponential expression
(DIFFERENCE)**DB10.EXPONENT//Exponential expression
(SUM)**FC100             // Exponential expression
```

Example 13-5 Exponential Expressions with Various Exponents

13.5 Comparative Expressions

Definition

A comparative expression is an expression of the type BOOL formed with comparative operators. These expressions are formed by combinations of addresses of the same type or type class with the operators shown in Table 13-7.

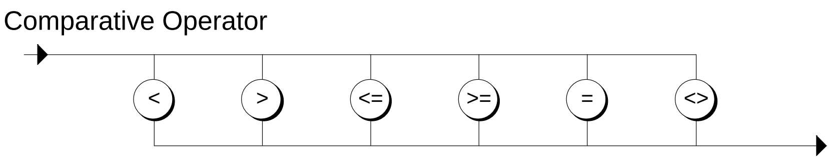


Figure 13-7 Syntax of Comparative Operators

Comparisons

The comparative operators compare the numerical value of two addresses.

`Address1 Operator Address2 ⇒ Boolean value`

The result obtained is a value that represents either the attribute TRUE or FALSE. The value is TRUE if the comparison condition is satisfied and FALSE if it is not.

Rules

The following rules must be adhered to when creating comparative expressions:

- Logical addresses should be enclosed in parentheses to ensure that the order in which the logical operations are to be performed is unambiguous.
- Logical expressions can be linked according to the rules of Boolean logic to create queries such as ”if a < b **and** b < c then ...”. Variables or constants of the type BOOL and comparative expressions can be used as the expression.
- Comparisons of all variables in the following type classes are permitted:
  - `INT, DINT, REAL`
  - `BOOL, BYTE, WORD, DWORD`
  - `CHAR, STRING`
- With the following time types, only variables of the same type can be compared:
  - `DATE, TIME, TOD, DT`
- When comparing characters (type CHAR), the operation follows the order of the ASCII character string.
- S5TIME variables can not be compared.
- If both addresses are of the type `DT` or `STRING`, you must use the appropriate IEC functions to compare them.

Examples

The examples below illustrate the construction of comparative expressions:

```
// The result of the comparative expression
// is negated.

      IF NOT (COUNTER > 5) THEN... ;
            //...
            //...
      END_IF;

// The result of the first comparative expression
// is negated and conjugated with the result
// of the second

      A:= NOT (COUNTER1 = 4) AND (COUNTER2 = 10) ;


// Disjunction of two comparative expressions
      WHILE (A >= 9) OR (QUERY <> ’n’) DO
            //...
            //...
      END_WHILE;
```

Example 13-6 Logical Expressions

13.6 Logical Expressions

Definition

A logical expression is an expression formed by logical operators. Using the operators `AND`, &, `XOR` and `OR`, logical addesses (type `BOOL`) or variables of the data type `BYTE`, `WORD` or `DWORD` can be combined to form logical expressions. The operator `NOT` is used to negate (that is, reverse) the value of a logical address.

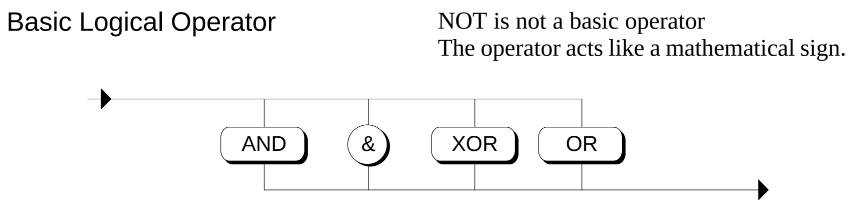

Basic Logical Operator
NOT is not a basic operator
The operator acts like a mathematical sign.
AND
&
XOR
OR

Figure 13-8 Syntax of Basic Logical Operators

Logic Operations

Table 13-3 below lists the available logical expressions and the data types for the results and addresses. The abbreviations have the following meaning:

`ANY_BIT`	for data types	`BOOL, BYTE, WORD, DWORD`

Table 13-3 Logical Operators

Operation	Operator	1st Address	2nd Address	Result	Priority
Negation	`NOT`	`ANY_BIT`	-	`ANY_BIT`	3
Conjunction	`AND`	`ANY_BIT`	`ANY_BIT`	`ANY_BIT`	8
Exclusive disjunction	`XOR`	`ANY_BIT`	`ANY_BIT`	`ANY_BIT`	9
Disjunction	`OR`	`ANY_BIT`	`ANY_BIT`	`ANY_BIT`	10

Results

The result of a logical expression is either

- 1 (*true*) or 0 (*false*) if Boolean operators are combined, or
- A bit pattern corresponding to the combination of the two addresses.

Examples

Let us assume that n is an integer variable with the value 10 and s is a character variable representing the character ‘A’. Some logical expressions using those variables could then be as follows:

Expression	Value
`( n>0 )  AND  ( n<20)`	`True`
`( n>0 )  AND  ( n<5 )`	`False`
`( n>0 )  OR   ( n<5 )`	`True`
`( n>0 )  XOR  ( n<20)`	`False`
`( n=10 ) AND  ( s=’A’)`	`True`
`( n<>5 ) OR   ( s>=’A’)`	`True`

Example 13-7 Logical Expressions

Value Assignments 14

Introduction

A value assignment is used to assign the value of an expression to a variable. The previous value of the variable is overwritten.

Further Information

In SCL there are simple and structured instructions. As well as value assignments, the simple instructions include operation calls and the GOTO instruction. For more detailed information, refer to Chapters 15 and 16.

The control instructions for a program branching operation or loop processing are structured instructions. A detailed explanation is given in Chapter 15.

14.1 Overview

Basic Principle

A value assignment replaces the current value of a variable with a new value specified by an expression. This expression can also contain identifiers for functions that it activates and which return corresponding values (return values).

As shown in syntax diagram 14-1, the expression on the right-hand side of the assignment operator is evaluated and the value obtained as the result is stored in the variable whose name is on the left-hand side of the assignment operator. The variables permitted for this function are shown in Figure 14-1.

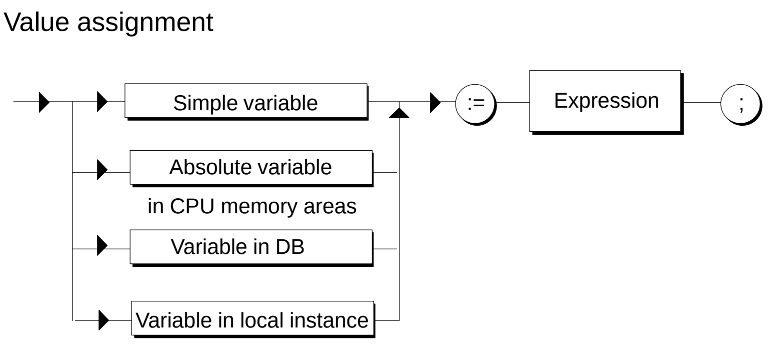


Figure 14-1 Syntax of Value Assignment

Results

The type of an assignment expression is the same as the type of the address on the left.

14.2 Value Assignments Using Variables of Elementary Data Types

Assignment

Any expression or variable of an elementary data type can be assigned to a different variable of the same type.

```
Identifier := expression ;
Identifier := variable of an elementary data type ;
```

Examples

The following are examples of valid value assignments:

```
FUNCTION_BLOCK FB10
VAR
      SWITCH_1    :INT;
      SWITCH_2    :INT;
      SETPOINT_1  :REAL;
      SETPOINT_2  :REAL;
      QUERY_1     :BOOL;
      TIME_1      :S5TIME;
      TIME_2      :TIME;
      DATE_1      :DATE;
      TIME_NOW_1  :TIME_OF_DAY;
END_VAR
BEGIN
// Assigning a constant to a variable
      SWITCH_1    := -17;
      SETPOINT_1  := 100.1;
      QUERY_1     := TRUE;
      TIME_1      :=TIME#1H_20M_10S_30MS;
      TIME_2      :=TIME#2D_1H_20M_10S_30MS;
      DATE_1      :=DATE#1996–01–10;
// Assigning a variable to a variable
      SETPOINT_1  := SETPOINT_2;
      SWITCH_2_   := SWITCH_1;
// Assigning an expression to a variable
      SWITCH_2:= SWITCH_1 * 3;
END_FUNCTION_BLOCK
```

Example 14-1 Value Assignments Using Elementary Data Types

14.3 Value Assignments Using Variables of the Types STRUCT or UDT

STRUCT and UDT Variables

Variables of the types STRUCT and UDT are structured variables which represent either a complete structure or a component of that structure.

The following are examples of valid structure variables:

```
Image           //Identifier for a structure
Image.element   //Identifier for a structure
                //component
Image.array     //Identifier for a single array
                //within a structure
Image.array[2,5] //Identifier for an array component
                //within a structure
```

Assigning a Complete Structure

An entire structure can only be assigned to another structure when the structure components match each other both in terms of data type and name. A valid assignment would be, for example:

```
  structname_1:=structname_2;
```

Assigning Structure Components

You can assign any structure component a variable of the same type, an expression of the same type or another structure component. The following assignments would be valid:

```
  structname_1.element1  := Value;
  structname_1.element1  := 20.0;
  structname_1.element1  := structname_2.element1;
  structname_1.arrayname1:= structname_2.arrayname2;
  structname_1.arrayname[10]:= 100;
```

Examples

The following examples illustrate value assignments for structure data.

```
FUNCTION_BLOCK FB10
VAR
      AUXVAR: REAL;
      MEASVALUE:  STRUCT  //destination structure
            VOLTAGE:REAL;
            RESISTANCE:REAL;
            SIMPLE_ARRAY:ARRAY[1..2,1..2] OF INT;
      END_STRUCT;

      PROCVALUE: STRUCT    //source structure
            VOLTAGE: REAL;
            RESISTANCE: REAL;
            SIMPLE_ARRAY:ARRAY[1..2,1..2] OF INT;
       END_STRUCT
END_VAR
BEGIN
//Assigning a complete structure to
//a complete structure
      MEASVALUE:= PROCVALUE;

//Assigning a structure component to a
//structure component
      MEASVALUE.VOLTAGE:= PROCVALUE.VOLTAGE

// Assigning a structure component to a
// variable of the same type
      AUXVAR:= PROCVALUE.RESISTANCE;

// Assigning a constant to a
// structure component
      MEASVALUE.RESISTANCE:= 4.5;

// Assigning a constant to a simple array
      MEASVALUE.SIMPLE_ARRAY[1,2]:= 4;
END_FUNCTION_BLOCK
```

Example 14-2 Value Assignments Using Variables of the Type STRUCT

14.4 Value Assignments Using Variables of the Type ARRAY

Array Variable

An array consists of one up to a maximum of six dimensions and contains elements that are all of the same type. There are two ways of assigning arrays to a variable as follows:

You can reference **complete** arrays or a **component** of an array. A complete array can be referenced by specifying the variable name of the array.

```
arrayname_1
```

A single component of an array is addressed using the array name followed by suitable index values in square brackets. An index is available for each dimension. These are separated by commas and also enclosed in square brackets. An index must be a mathematical expression of the data type `INT`.

```
arrayname_1[2]
arrayname_1[4,5]
```

Assigning a Complete Array

A complete array can be assigned to another array when both the data types of the components and the array limits (lowest and highest possible array indices) match. A valid assignment would be as follows:

```
arrayname_1 := arrayname_2 ;
```

Assigning an Array Component

A value assignment for a permissible array component is obtained by omitting indices in the square brackets after the name of the array, starting on the right. In this way, you address a subset of the array whose number of dimensions is equal to the number of indices omitted.

This means that you can reference ranges of lines and individual components of a matrix but not column ranges (that is, from ... to).

The following are examples of valid assignments

```
arrayname_1[ i ] := arrayname_2[ j ] ;
arrayname_1[ i ] := expression ;
identifier_1     := arrayname_1[ i ] ;
```

Examples

The examples below illustrate value assignments for arrays.

```
FUNCTION_BLOCK FB3
VAR
      SETPOINTS   :ARRAY [0..127] OF INT;
      PROCVALUES  :ARRAY [0..127] OF INT;
END_VAR

// Declaration of a matrix
// (=two-dimensional array)
// with 3 lines and 4 columns
      CTRLLR: ARRAY [1..3, 1..4] OF INT;

// Declaration of a vector
// (=one-dimensional array)
// with 4 components
      CTRLLR_1: ARRAY [1..4] OF INT;
END_VAR

BEGIN
// Assigning a complete array to an array
      SETPOINTS:= PROCVALUES;

// Assigning a vector to the second line
// of the CTRLLR ARRAY
      CTRLLR[2]:= CTRLLR_1;

//Assigning a component of an array to a
//component of the CTRLLR ARRAY
      CTRLLR [1,4]:= CTRLLR_1 [4];
END_FUNCTION_BLOCK
```

Example 14-3 Value Assignments Using Variables of the Type ARRAY

14.5 Value Assignments Using Variables of the Type STRING

STRING Variables

A variable of the data type STRING contains a character string with a maximum of 254 characters.

Assignment

Each variable of the data type STRING can be assigned another variable of the same type. Valid assignments would be as follows:

```
stringvariable_1 := Stringliteral ;
stringvariable_1 := stringvariable_2 ;
```

Example

The examples below illustrate value assignments using STRING variables:

```
FUNCTION_BLOCK FB3
VAR
      DISPLAY_1   : STRING[50] ;
      STRUCTURE1  : STRUCT
                    DISPLAY_2 : STRING[100] ;
                    DISPLAY_3 : STRING[50] ;
      END_STRUCT;
END_VAR
BEGIN
// Assigning a constant to a STRING
// variable

      DISPLAY_1 := ’error in module 1’ ;

// Assigning a structure component to a
// STRING variable.

      DISPLAY_1 := STRUCTURE1.DISPLAY_3 ;

// Assigning a STRING variable to
// a STRING variable

      If DISPLAY_1 <> DISPLAY_3 THEN
         DISPLAY_1 := DISPLAY_3 ;
      END_IF;
END_FUNCTION_BLOCK
```

Example 14-4 Value Assignments Using Variables of the Type STRING

14.6 Value Assignments Using Variables of the Type DATE_AND_TIME

DATE_AND_TIME Variables

The data type DATE_AND_TIME defines an area with 64 bits (8 bytes) for the date and time.

Assignment

Each variable of the data type DATE_AND_TIME can be assigned another variable of the same type or a constant. Valid assignments would be as follows:

```
dtvariable_1 := date and time literal ;
dtvariable_1 := dtvariable_2 ;
```

Example

The examples below illustrate value assignments using DATE_AND_TIME variables:

```
FUNCTION_BLOCK FB3
VAR
      TIME_1      : DATE_AND_TIME;
      STRUCTURE1  : STRUCT
              TIME_2 : DATE_AND_TIME ;
              TIME_3 : DATE_AND_TIME ;
      END_STRUCT;
END_VAR
BEGIN
// Assigning a constant to a
// DATE_AND_TIME variable
TIME_1 := DATE_AND_TIME#1995–01–01–12:12:12.2 ;
STRUCTURE.TIME_3 := DT#1995–02–02–11:11:11 ;

// Assigning a structure component to a
// DATE_AND_TIME variable.

TIME_1 := STRUCTURE1.TIME_2 ;


// Assigning a DATE_AND_TIME variable
// to a DATE_AND_TIME structure component
If TIME_1 < STRUCTURE1.TIME_3 THEN
   TIME_1 := STRUCTURE3.TIME_1 ;
END_IF;
END_FUNCTION_BLOCK
```

Example 14-5 Value Assignments Using DATE_AND_TIME Variables

14.7 Value Assignments using Absolute Variables for Memory Areas

Absolute Variables

An absolute variable references the globally valid memory areas of a CPU. You can assign values to these areas in three ways as described in Chapter 12.

Absolute Variable

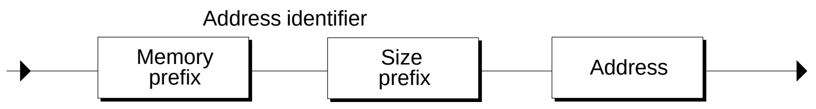


Figure 14-2 Syntax of Absolute Variables

Assignment

Any absolute variable with the exception of peripheral inputs and process image inputs can be assigned a variable or expression of the same type.

Example

The examples below illustrate value assignments using absolute variables:

```
VAR
      STATUSWORD1: WORD;
      STATUSWORD2: BOOL;
      STATUSWORD3: BYTE;
      STATUSWORD4: BOOL;
      ADDRESS: INT:= 10;
END_VAR
BEGIN
      // Assigning an input word to a
      // variable (simple access)
      STATUSWORD1:= IW4 ;

      // Assigning a variable to an
      // output bit (simple access)
      STATUSWORD2:= Q1.1 ;

      // Assigning an input byte to a
      // variable (indexed access)
      STATUSWORD3:= IB[ADDRESS];

      // Assigning an input bit to a
      // variable (indexed access)
      FOR ADDRESS:= 0 TO 7 BY 1 DO
      STATUSWORD4:= I[1,ADDRESS] ;
      END_FOR;
END_FUNCTION_BLOCK
```

Example 14-6 Value Assignments Using Absolute Variables

14.8 Value Assignments using Global Variables

Variables in DBs

You can also access global variables in data blocks by assigning a value to variables of the same type or vice-versa. You have the option of using structured, absolute or indexed access (see Chapter 12).

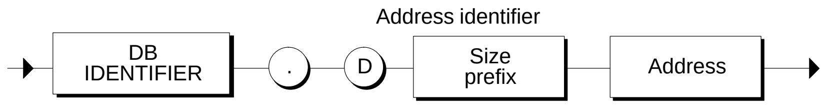


Figure 14-3 Syntax of DB Variables

Assignment

You can assign any global variable a variable or expression of the same type. The following are examples of valid assignments:

```
DB11.DW10:=20;
DB11.DW10:=Status;
```

Examples

The example below assumes that that a variable ″DIGIT″ of the data type INTEGER and a structure ”DIGIT1″ with the component ″DIGIT2″ of the type INTEGER have been declared in the data block DB11.

```
// Required data block DB11
DATA_BLOCK DB11
STRUCT
      DIGIT :     INT:=1;
      DIGIT1:     STRUCT
                  DIGIT2:INT := 256;
      END_STRUCT;
      WORD3 :     WORD:=W#16#aa;
      WORD4 :     WORD:=W#16#aa;
      WORD5 :     WORD:=W#16#aa;
      WORD6 :     WORD:=W#16#aa;
      WORD7 :     WORD:=W#16#aa;
      WORD8 :     WORD:=W#16#aa;
      WORD9 :     WORD:=W#16#aa;
      WORD10:     WORD;
END_STRUCT

BEGIN
WORD10:=W#16#bb;
END_DATA_BLOCK
```

Example 14-7 Value Assignments Using Global Variables

Data block DB11 could then be used as follows, for example:

```
VAR
    CONTROLLER_1: ARRAY [1..4] OF INT;
    STATUSWORD1   : WORD ;
    STATUSWORD2   : ARRAY [1..4] OF INT;
    STATUSWORD3   : INT ;
    ADDRESS : INT ;
END_VAR
BEGIN
    // Assignment of word 10 from DB11 to a
    // variable (simple access)
    STATUSWORD1:= DB11.DW10

    // The 1st array component is assigned
    // the variable
    // ”DIGIT” from DB11
    // (structured access):
    CONTROLLER_1[1]:= DB11.DIGIT;

     // Assignment of structure component ”DIGIT2”
     // of structure ”DIGIT1” to the variable
     // Statusword3
    STATUSWORD3:= DB11.DIGIT1.DIGIT2

    // Assignment of a word with index
      ADDRESS from
    // DB11 to a variable
    // (indexed access)
    FOR ADDRESS:= 1 TO 10 BY 1 DO
    STATUSWORD2[ADDRESS]:= DB11.DW[ADDRESS] ;
    END_FOR;
```

Example 14-8 Value Assignments Using the Global Variables of a Data Block

Control Statements 15

Introduction

Only on rare occasions is it possible to program blocks in such a way that all statements are processed one after the other from the beginning to the end of the block. It is usually the case that on the basis of specific conditions only certain statements (alternatives) are executed or are repeated a number of times over (loops). The programming tools used to bring about such effects are the control statements in an SCL block.

Chapter Overview

15.1 Overview

Selective Instructions

In programs, different instructions often have to be executed according to different conditions. A selective instruction enables you to direct the program progression into any number of alternative sequences of instructions.

Table 15-1 Types of Branch

Branch Type	Function
IF Statement	The IF statement enables you to direct the program progression into one of two alternative branches according to whether a specified condition is either TRUE of FALSE:
CASE Statement	The CASE statement enables you direct the program progression into 1 of n alternative branches by having a variable adopt a value from n alternatives.

Repetition Instructions

You can control loop processing by means of repetition instructions. A repetition instruction specifies which parts of a program should be repeated on the basis of specific conditions.

Table 15-2 Types of Statement for Loop Processing

Branch Type	Function
`FOR` Statement	Used to repeat a sequence of statements for as long as the control variable remains within the specified value range
`WHILE` Statement	Used to repeat a sequence of statements while an execution condition continues to be satisfied
`REPEAT` Statement	Used to repeat a sequence of statements until a break condition is met

Jump Statements

A jump statement causes the program to jump immediately to a specified jump destination and therefore to a different statement within the same block.

Table 15-3 Types of Jump Statement

Branch Type	Function
`CONTINUE` Statement	Used to stop processing of the current loop pass
`EXIT` Statement	Used to exit from a loop at any point regardless of whether the break condition is satisfied or not
`GOTO` Statement	Causes the program to jump immediately to a specified jump label
`RETURN` Statement	Causes the program to exit the block currently being processed

Conditions

A condition is either a comparative expression or a logical expression. The data type of a condition is `BOOL` and it can adopt either of the two values `TRUE` or `FAlSE`.

The following are examples of valid **comparative expressions**:

```
COUNTER<=100

SQR(A)>0.005

Answer = 0

BALANCE>=BALBFWD

ch1< 'T'
```

The following are examples of the use of comparative expressions with **logical** operators:

```
(COUNTER<=100) AND(CH1<'*')

(BALANCE<100.0) OR (STATUS ='R')

(Answer<0)OR((Answer>5.0) AND (Answer<10.0))
```

Note

Note that the logical addresses (in this case comparative expressions) are in brackets in order to prevent any ambiguity with regard to the order in which they are processed.

15.2 IF Statement

Basic Principle

The `IF` statement is a conditional statement. It provides one or more options and selects one (or none) of its statement components for execution.

IF Statement

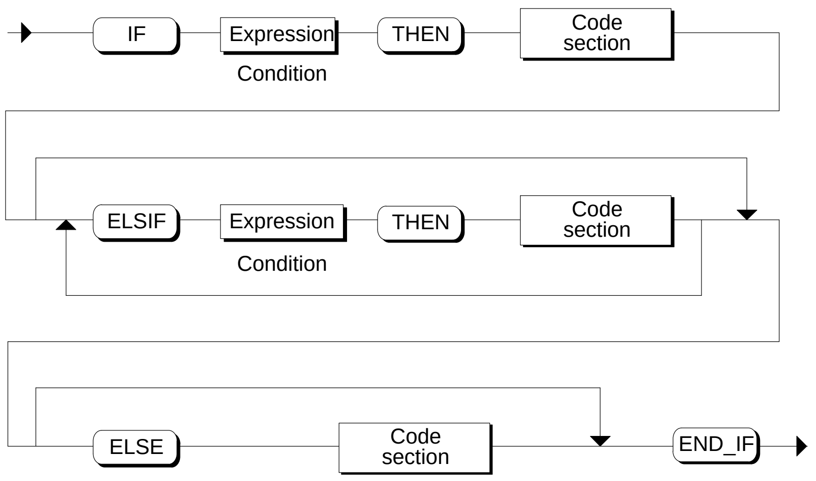


Figure 15-1 Syntax of the IF Statement

Execution of the conditional statement forces analysis of the specified logical expressions. If the value of an expression is TRUE then the condition is satisfied, if it is FALSE the condition is not satisfied.

Execution

An IF statement is processed according to the following rules:

1. If the value of the first expression is TRUE, the component of the statement which follows `THEN` is executed. Otherwise the statements in the `ELSIF` branches are processed.
2. If no Boolean expression in the `ELSIF` branches is TRUE, the sequence of statements following ELSE (or no sequence of statements if there is no `ELSE` branch) is executed.

Any number of ELSIF statements can be used.

It should be noted that the `ELSIF` branches and/or the `ELSE` branch can be omitted. In such cases, the program behaves as if those branches were present but contained no statements.

Note

Note that the statement `END_IF` must be concluded with a semicolon.

Note

Using one or more ELSIF branches has the advantage that the logical expressions following a valid expression are no longer evaluated in contrast to a sequence of IF statements. The runtime of a program can therefore be reduced.

Example

Example 15-1 below illustrates the use of the IF statement.

```
IF I1.1 THEN
      N:= 0;
      SUM:= 0;
      OK:= FALSE; // Set OK flag to FALSE
ELSIF START = TRUE THEN
      N:= N + 1;
      SUM:= SUM + N;
ELSE
      OK:= FALSE;
END_IF;
```

Example 15-1 IF Statements

15.3 CASE Statement

Basic Principle

The CASE statement selects one program section from a choice of n alternatives. That choice is based on the current value of a selection expression.

CASE Statement

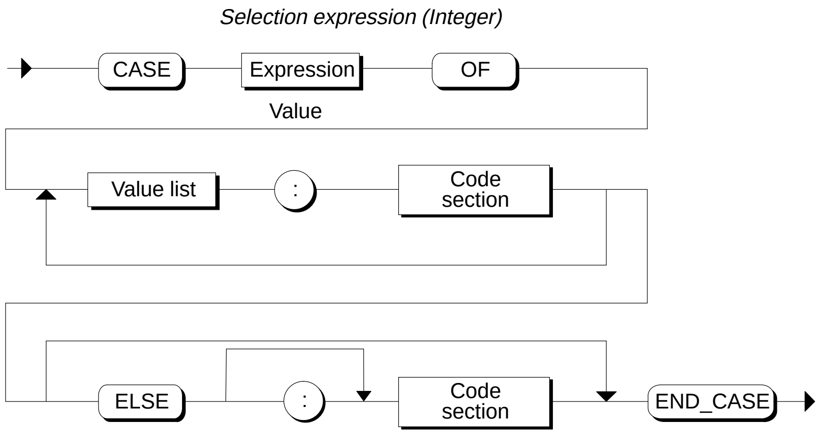


Figure 15-2 Syntax of the CASE Statement

Execution

The CASE statement is processed according to the following rules:

1. When a CASE statement is processed, the program checks whether the value of the selection expression is contained within a specified list of values. Each value in that list represents one of the permissible values for the selection expression. The selection expression must return a value of the type INTEGER.
2. If a match is found, the statement component assigned to the list is executed.
3. The ELSE branch is optional: it is executed if no match is found.

Note

Note that the statement END_CASE must be concluded with a semicolon.

Value List

This contains the permissible values for the selection expression

Value List

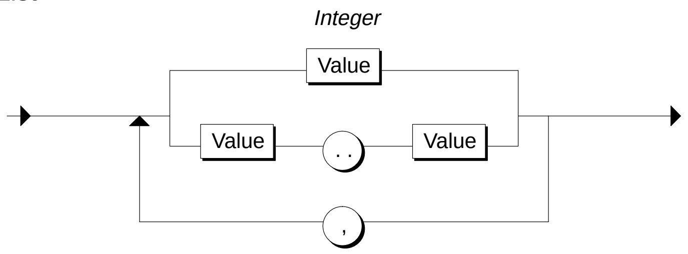


Figure 15-3 Syntax of Value List

Rules

When creating the value list you must observe the following rules:

- Each value list must begin with a constant, a list of constants or a range of constants.
- The values within the value list must be of the INTEGER type.
- Each value must only occur once.

Examples

Example 15-2 below illustrates the use of the CASE statement. The variable TW is usually of the INTEGER type.

```
CASE TW OF
      1:    DISPLAY   := OVEN_TEMP;
      2:    DISPLAY   := MOTOR_SPEED;
      3:    DISPLAY   := GROSS_TARE;
                  QW4 := 16#0003;
      4..10:DISPLAY   := INT_TO_DINT (TW);
                  QW4 := 16#0004;
      11,13,19:DISPLAY:= 99;
                  QW4 := 16#0005;
ELSE:       DISPLAY   := 0;
            TW_ERROR  := 1;
END_CASE;
```

Example 15-2 CASE Statement

Note

Take care to ensure that the running time of loops is not too long, otherwise the CPU will register a time-out error and switch to STOP mode.

15.4 FOR Statement

Basic Principle

A `FOR` statement is used to repeat a sequence of statements in a loop while a variable (the control variable) is continually assigned values. The control variable must be the identifier of a local variable of the type INT or DINT.

FOR Statement

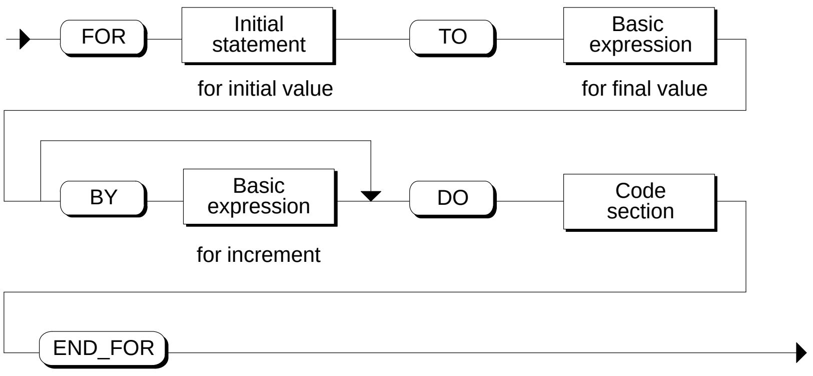


Figure 15-4 Syntax of FOR Statement

The definition of a loop using FOR includes the specification of an initial and a final value. Both values must be the same type as the control variable.

Execution

The FOR statement is processed according to the following rules:

1. At the start of the loop, the control variable is set to the initial value (initial assignment) and each time the loop is run through it is increased (positive increment) or decreased (negative increment) by the specified increment until the final value is reached.
2. Following each run through of the loop, the condition

   ```
   |initial value| <= |final value|
   ```

   is checked to establish whether or not it is satisfied. If the condition is satisfied, the sequence of statements is executed, otherwise the loop and thereby the sequence of statements is skipped.

Note

Note that the statement `END_FOR` must be concluded with a semicolon.

Initial Assignment

The initial assignment shown in Figure 15-5 can be used to create the initial value of the control variable.

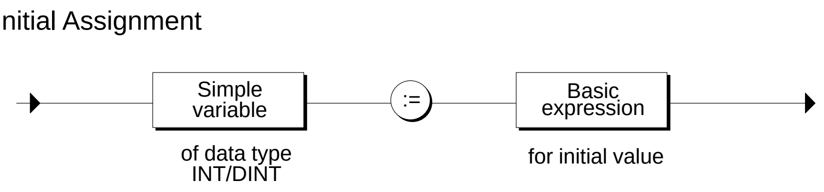


Figure 15-5 Syntax for Creating the Initial Value

Examples:

```
FOR I  := 1 TO 20
FOR I  := 1 TO (Init+J) DO
```

Final Value and Increment

You can create a basic expression for creating the final value and the required increment.

Rules

The following rules must be observed for the FOR statement:

- You can omit the statement *`BY [increment]`*. If no increment is specified, it is automatically assumed to be `+1`.
- Initial value, final value and increment are expressions (see Chapter 13). They are processed once only at the start of execution of the `FOR` statement.
- Alteration of the values for final value and increment is not permissible while the loop is being processed.

Example

Example 15-3 below illustrates the use of the FOR statement.

```
FUNCTION_BLOCK SEARCH
VAR
      INDEX       : INT;
      KEYWORD     : ARRAY [1..50] OF STRING;
END_VAR

BEGIN
FOR INDEX:= 1 TO 50 BY 2 DO
      IF KEYWORD [INDEX] = 'KEY' THEN
      EXIT;
      END_IF;
END_FOR;
END_FUNCTION_BLOCK
```

Example 15-3 FOR Statement

15.5 WHILE Statement

Basic Principle

The `WHILE` statement allows the repeated execution of a sequence of statements on the basis of an execution condition. The execution condition is formed according to the rules of a logical expression.

WHILE Statement

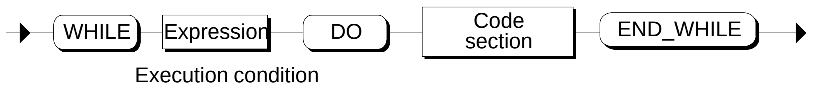


Figure 15-6 Syntax of the WHILE Statement

The statement component which follows `DO` is repeated as long as the value of the execution condition remains TRUE.

Execution

The `WHILE` statement is processed according to the following rules:

1. The execution condition is checked **before** each execution of the statement component.
2. If the value TRUE is returned, the statement component is executed.
3. If the value FALSE is returned, execution of the WHILE statement is terminated. It is possible for this to occur on the very first occasion the execution condition is checked.

Note

Note that the statement `END_WHILE` must be concluded with a semicolon.

Example

Example 15-4 below illustrates the use of the `WHILE` statement.

```
FUNCTION_BLOCK SEARCH
VAR
      INDEX       : INT;
      KEYWORD     : ARRAY [1..50] OF STRING;
END_VAR
BEGIN
INDEX:= 1;
WHILE INDEX <= 50 AND KEYWORD[INDEX] <> ’KEY’ DO
      INDEX:= INDEX + 2;
END_WHILE;
END_FUNCTION_BLOCK
```

Example 15-4 WHILE Statement

15.6 REPEAT Statement

Basic Principle

A REPEAT statement causes the repeated execution of a sequence of statements between REPEAT and UNTIL until a break condition occurs. The break condition is formed according to the rules of a logical expression.

REPEAT Statement

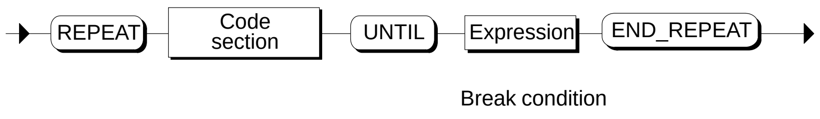


Figure 15-7 Syntax of the REPEAT Statement

The condition is checked **after** the loop has been executed. This means that the loop must be executed at least **once** even if the break condition is satisfied when the loop is started.

Note

Note that the statement END_REPEAT must be concluded with a semicolon.

Example

Example 15-5 below illustrates the use of the REPEAT statement

```
FUNCTION_BLOCK SEARCH
VAR
      INDEX       : INT;
      KEYWORD     : ARRAY [1..50] OF STRING;
END_VAR

BEGIN
INDEX:= 0;
REPEAT
      INDEX:= INDEX + 2;
UNTIL
      INDEX > 50 OR KEYWORD[INDEX] = ’KEY’
END_REPEAT;
END_FUNCTION_BLOCK
```

Example 15-5 REPEAT Statement

15.7 CONTINUE Statement

Basic Principle

A CONTINUE statement is used to terminate the execution of the current iteration of a loop (initiated by a FOR, WHILE or REPEAT statement) and to restart processing within the loop.

CONTINUE Statement

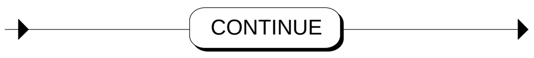


Figure 15-8 Syntax of the CONTINUE Statement

In a WHILE loop, the initial condition determines whether the sequence of statements is repeated and in a REPEAT loop the terminal condition.

In a FOR statement, the control variable is increased by the specified increment immediately after a CONTINUE statement.

Example

Example 15-6 below illustrates the use of the CONTINUE statement.

```
FUNCTION_BLOCK_CONTINUE
VAR
      INDEX :INT;
      ARRAY_1:ARRAY[1..100] OF INT;
END_VAR
BEGIN
INDEX:= 0;
WHILE INDEX <= 100 DO
      INDEX:= INDEX + 1;
      // If ARRAY_1[INDEX] equals INDEX,
      // then ARRAY_1 [INDEX] is not altered:
            IF ARRAY_1[INDEX] = INDEX THEN
                  CONTINUE;
            END_IF;
            ARRAY_1[INDEX]:= 0;
      // Other statements..
      //....
END_WHILE;
END_FUNCTION_BLOCK
```

Example 15-6 CONTINUE Statement

15.8 EXIT Statement

Basic Principle

An `EXIT` statement is used to exit a loop (`FOR`, `WHILE` or `REPEAT` loop) at any point regardless of whether the break condition is satisfied.

EXIT Statement

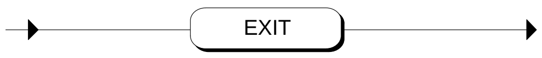


Figure 15-9 Syntax of the EXIT Statement

This statement causes the repetition statement immediately surrounding the exit statement to be exited immediately.

Execution of the program is continued after the end of the loop (for example after `END_FOR`).

Example

Example 15-7 below illustrates the use of the EXIT statement.

```
FUNCTION_BLOCK_EXIT
VAR
      INDEX_1      := INT;
      INDEX_2      := INT;
      INDEX_SEARCH:= INT;
      KEYWORD      : ARRAY[1..51] OF STRING;
END_VAR
BEGIN
INDEX_2     := 0;
FOR INDEX_1:= 1 TO 51 BY 2 DO
      // Exit the FOR loop if
      // KEYWORD[INDEX_1] equals ’KEY’:
      IF KEYWORD[INDEX_1] = ’KEY’ THEN
            INDEX_2:= INDEX_1;
            EXIT;
      END_IF;
END_FOR;
// The following value assignment is executed
// after execution of EXIT or after the
// normal termination of the FOR loop
INDEX_SEARCH:= INDEX_2;
END_FUNCTION_BLOCK
```

Example 15-7 EXIT Statement

15.9 GOTO Statement

Basic Principle

The **GOTO** statement is used to implement a program jump. It effects an immediate jump to the specified jump label and therefore to a different statement within the same block.

GOTO statements should only be used in special circumstances; for example, for error handling. According to the rules of structured programming, the GOTO statement should not be used.

GOTO Statement

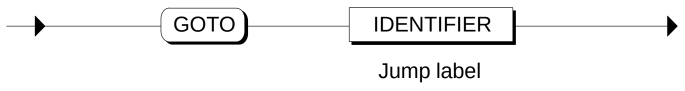


Figure 15-10 Syntax of the GOTO Statement

Jump label refers to a marker in the `LABEL / END_LABEL` declaration subsection. That marker precedes the statement which is to be next executed after the `GOTO` statement.

Rules

The following rules should be observed when using the GOTO statement:

- The destination of a GOTO statement must be within the same block.
- The destination of the jump must be unambiguous.
- Jumping to a loop is not permitted. Jumping from a loop is possible.

Example

Example 15-8 below illustrates the use of the GOTO statement.

```
FUNCTION_BLOCK FB3//GOTO_BSP
VAR
      INDEX : INT;
      A     : INT;
      B     : INT;
      C     : INT;
      KEYWORD: ARRAY[1..51] OF STRING;
END_VAR
LABEL
      LABEL1, LABEL2, LABEL3;
END_LABEL
BEGIN
IF A > B THEN GOTO LABEL1;
      ELSIF A > C THEN GOTO LABEL2;
END_IF;
//...
LABEL1      :     INDEX:= 1;
                  GOTO LABEL3;
LABEL2      :     INDEX:= 2;
//...
LABEL3      :     ;
//...
END_FUNCTION_BLOCK
```

Example 15-8 GOTO Jump Statement

15.10 RETURN Statement

Basic Principle

A RETURN statement causes the program to exit the block (OB, FB or FC) currently being processed and to return to the calling block or the operating system if the block being exited is an OB.

RETURN Instruction

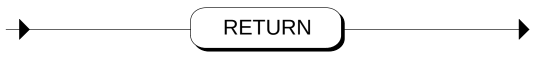


Figure 15-11 Syntax of the RETURN Statement

Note

A RETURN statement at the end of the code section of a logic block or the declaration section of a data block is redundant, since the operation is performed automatically at those points.

Calling Functions and Function Blocks 16

Introduction

An SCL block can call the following:

- Other functions (FCs) and function blocks (FBs) created in SCL
- Functions and function blocks programmed in another STEP 7 language (for example, Statement List or Ladder Logic)
- System functions (SFCs) and system function blocks (SFBs) in the operating system of the CPU you are using.

Chapter Overview

16.1 Calling and Transferring Parameters

Parameter Transfer

When functions or function blocks are called, data is exchanged between the calling and the called block. The parameters that are to be transferred must be specified in the function call in the form of a parameter list. The parameters are enclosed in brackets. A number of parameters are separated by commas.

Basic Principle

In the example of a function call below, an input parameter, an in/out parameter and an output parameter are specified.

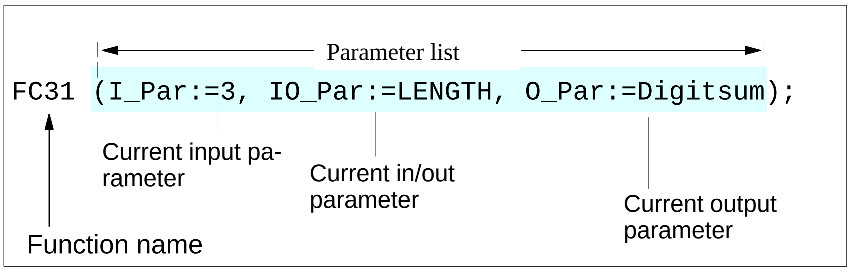


Figure 16-1 Basic Principle of Parameter Transfer

As is shown in Figure 16-2, specification of parameters takes the form of a value assignment. That value assignment assigns a value (actual parameter) to the parameters defined in the declaration section of the called block (formal parameters).

Formal Parameter		Actual Parameter
`I_Par`	⇐	`3`
`IO_Par`	⇐	`LENGTH`
`O_Par`	⇒	`Digitsum`

Figure 16-2 Value Assignment within the Parameter List

Formal Parameters

The formal parameters are those parameters expected by the block when invoked. They are merely “placeholders” for the actual parameters that are transferred to the block when called. Those parameters have been defined in the declaration section of a block (FB or FC).

Table 16-1 Permissible Declaration Subsections for Formal Parameters

Declaration Subsections	Data	Keyword

Parameter subsection	Input parameters	`VAR_INPUT`
		Declaration list
		`END_VAR`

	Output parameters	`VAR_OUTPUT`
		Declaration list
		`END_VAR`

	In/Out parameters	`VAR_IN_OUT`
		Declaration list
		`END_VAR`

16.2 Calling Function Blocks (FB or SFB)

Global and Local Instance

When you call a function block in SCL you can use

- Global instance data blocks, and
- Local instance areas of the active instance data block.

Calling an FB as a local instance differs from calling it as a global instance by virtue of the way in which the data is stored. In this case, the data is not stored in a special DB but is nested in the instance data block of the calling FB.

Function Block Call

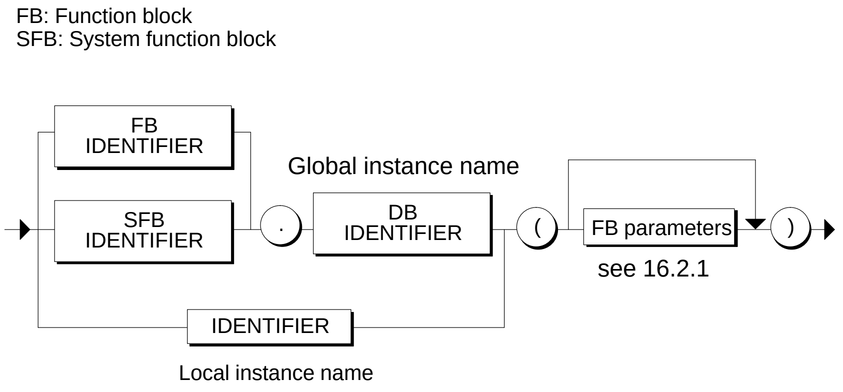


Figure 16-3 Syntax of an FB Call

Calling as Global Instance

The function call is made in a call instruction by specifying the following:

- the name of the function block or system function block (FB or SFB identifier),
- the instance data block (DB identifier),
- the parameter assignment (FB parameters)

A function call for a global instance can be either absolute or symbolic.

Absolute function call:

```
FB10.DB20 (X1:=5,X2:=78,......);
```

Parameter assignment

Symbolic function call:

```
DRIVE.ON (X1:=5,X2:=78,......);
```

Figure 16-4 Calling FB10 Using Instance Data Block DB20

Calling as Local Instance

The function call is made in a call instruction by specifying the following:

- the local instance name (IDENTIFIER),
- the parameter assignment (FB parameters).

A function call for a local instance is always symbolic, for example:

```
MOTOR (X1:=5,X2:=78,......);
                   ↑
                      Parameter assignment
```

Figure 16-5 Calling a Local Instance

16.2.1 FB Parameters

Basic Principle

When calling a function block – as a global or local instance – you must make a distinction in the parameter list between

- the input parameters and
- the in/out parameters

of an FB. In both cases, you use **value assignments** to assign the actual parameters to the formal parameters as illustrated below:

```
Formal Parameter        Actual Parameter
I_Par          ⇐     3       //Input assignment
IO_Par         ⇐     LENGTH //In/Out assignment
```

Figure 16-6 Value Assignment within the Parameters List

The output parameters do not have to be specified when an FB is called.

The syntax of the FB parameter specification is the same when calling both global and local instances.

FB Parameters

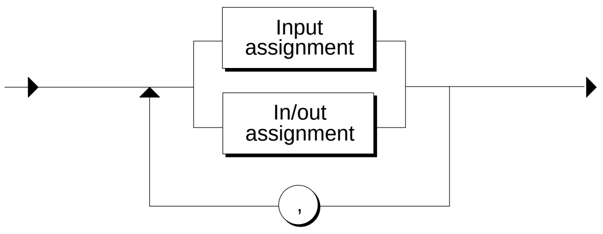


Figure 16-7 Syntax of FB Parameters

Example

A function call involving assignment of one input and one in/out parameter might be as follows:

```
FB31.DB77(I_Par:=3, IO_Par:=LENGTH);
```

Rules

The rules for assigning parameter values are as follows:

- The assignments can be in any order.
- Individual assignments are separated by commas.
- The data type of formal and actual parameters must match.
- Output assignments are not possible in FB calls. The value of a declared output parameter is stored in the instance data. From there it can be accessed by all FBs. To read an output parameter, you must define the access from within an FB (see Section 14.8).

Results of Function Call

When the block has been run through once:

- The actual parameters transferred are unchanged.
- The transferred and altered values of the in/out parameters have been updated; in/out parameters of an elementary data type are an exception to this rule (see Section 16.2.3).
- The output parameters can be read by the calling block from the global instance area or the local instance area. For more precise details, refer to Example 16-3.

16.2.2 Input Assignment (FB)

Basic Principle

Input assignments are used to assign actual parameters to the formal input parameters. The FB **cannot** change these actual parameters. The assignment of actual input parameters is optional. If no actual parameter is specified, the values of the last call are retained.

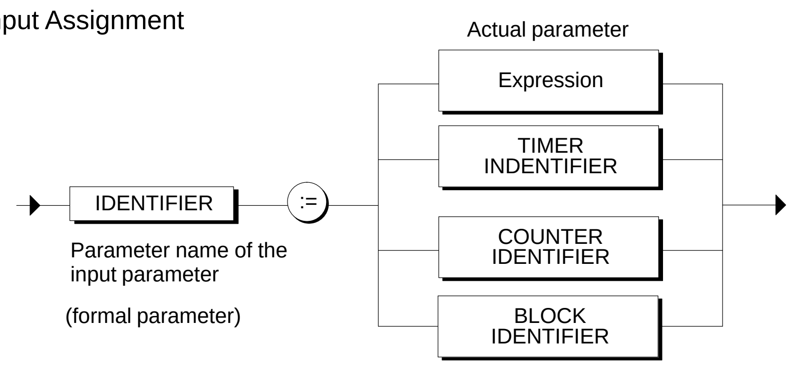


Figure 16-8 Syntax of an Input Assignment

Permissible Actual Parameters

The following actual parameters are permitted in input assignments:

Table 16-2 Actual Parameters in Input Assignments

Actual Parameter	Explanation
Expression	• Mathematical, logical or comparative expression • Constant • Extended variable
TIMER/COUNTER Identifier	Defines a specific timer or counter to be used when a block is processed (see also Chapter 17).
BLOCK Identifier	Defines a specific block to be used as an input parameter. The block type (FB, FC or DB) is specified in the input parameter declaration. When assigning parameter values you specify the block number. You can use either the absolute or symbolic number (see also Chapter 9).

16.2.3 In/Out Assignment (FB)

Basic Principle

In/out assignments are used to assign actual parameters to the formal in/out parameters of the FB that has been called.

In contrast to input parameters, the called FB can change the in/out parameters. The new value of a parameter that results from processing the FB is written back to the actual parameters. The original value is overwritten.

If in/out parameters are declared in the called FB, they must be assigned values the first time the block is called. After that, the specification of actual parameters is optional.

In/Out Assignment

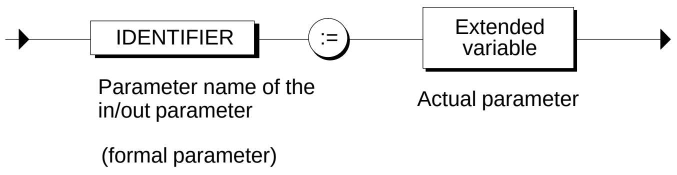


Figure 16-9 Syntax of an In/out Assignment

Actual Parameters of an In/out Assignment

Since the actual parameter assigned can be altered when the FB is processed as it is an in/out parameter, it has to be a variable. For that reason, input parameters can not be assigned by means of in/out assignments (the new value would not be capable of being written back).

Table 16-3 Actual Parameters in In/Out Assignments

Actual Parameter	Explanation
Extended variable	The following types of extended variable are possible: Simple variables and parameters Access to absolute variables Access to data blocks Function calls (see also Chapter 14).

Special Considerations

Note the following special considerations:

- When the block is processed, the altered value of the in/out parameter is updated. In/out parameters of an **elementary** data type are an exception to this rule. In the latter case, an update is only performed if an actual parameter is specified in the function call.
- The following can not be used as actual parameters for an in/out parameter of a **non elementary** data type:
  - FB in/out parameters
  - FC parameters
- **ANY parameters:** the aforesaid applies in this case, too. In addition, constants are **not** permissible as actual parameters.

16.2.4 Example of Calling a Global Instance

Basic Principle

An example of a function block with a FOR loop is shown in Example 16-1. The examples given assume that the symbol TEST has been declared in the symbol table for FB17.

```
FUNCTION_BLOCK TEST
      VAR_INPUT
            FINALVAL: INT; //Input parameter
      END_VAR
      VAR_IN_OUT
            IQ1: REAL; //In/Out parameter
      END_VAR
      VAR_OUTPUT
            CONTROL: BOOL;//Output parameter
      END_VAR
      VAR
            INDEX: INT;
      END_VAR
      BEGIN
            CONTROL:= FALSE;
            FOR INDEX:= 1 TO FINALVAL DO
            IQ1:= IQ1 * 2;
            IF IQ1 > 10000 THEN
            CONTROL:= TRUE;
            END_IF;
            END_FOR;
END_FUNCTION_BLOCK
```

Example 16-1 Example of an FB

Calling

To call the FB, you can choose one of the following options. It is assumed that VARIABLE1 has been declared in the calling block as a REAL variable.

```
  //Absolute function call, global instance:
FB17.DB10 (FINALVAL:=10, IQ1:= VARIABLE1);

  //Symbolic function call; global instance:
TEST.TEST_1 (FINALVAL:= 10, IQ1:= VARIABLE1) ;
```

Example 16-2 Example of FB Call Using an Instance Data Block

Result

After the block has been processed, the value calculated for the in/out parameter IQ1 can be accessed from VARIABLE1.

Reading the Output Value

The two examples below illustrate the two possible ways of reading the output parameter `CONTROL`.

```
//The output parameter is
//accessed by
RESULT:= DB10.CONTROL;
//However, you can also use the output parameter
//directly in another //FB call for assigning
//a value to an input parameter as follows:


FB17.DB12 (IN_1:= DB10.CONTROL);
```

Example 16-3 Result of FB Call with Instance Data Block

16.2.5 Example of Calling a Local Instance

Basic Principle

Example 16-1 illustrates how a function block with a simple FOR loop could be programmed assuming that the symbol `TEST` has been declared in the symbol table for `FB17`.

Calling

This FB can be invoked as shown below, assuming that `VARIABLE1` has been declared in the invoking block as a `REAL` variable.

```
// Call local instance:
TEST_L (FINALVAL:= 10, IQ1:= VARIABLE1) ;
```

Example 16-4 Example of FB Call as Local Instance

TEST_L must have been declared in the variable declaration as follows:

```
VAR
TEST_L : TEST;
END_VAR
```

Reading Output Parameters

The output parameter `CONTROL` can be read as follows:

```
// The output parameter is
// accessed by
RESULT:= TEST_L.CONTROL;
```

Example 16-5 Result of FB Call as Local Instance

16.3 Calling Functions

Return Value

In contrast to function blocks, functions always return a result known as the return value. For this reason, functions can be treated as addresses. Functions with a return value of the type `VOID` are an exception to this rule.

In the following value assignment, for example, the function DISTANCE is called with specific parameters:

```
LENGTH:= DISTANCE (X1:=–3, Y1:=2);
Return value is DISTANCE!
```

The function calculates the return value, which has the same name as the function, and returns it to the calling block. There, the value replaces the function call.

The return value can be used in the following elements of an FC or FB:

- a value assignment,
- a logical, mathematical or comparative expression or
- as a parameter for a further function block/function call.

Functions of the type VOID are an exception. They have no return value and can therefore not be used in expressions.

Figure 16-10 below illustrates the syntax of a function call.

Function Call

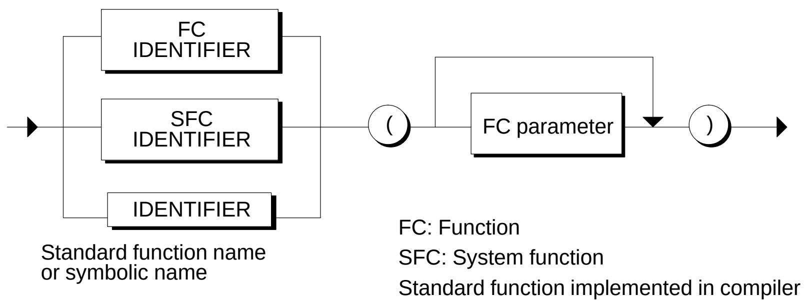


Figure 16-10 Syntax of Function Call

Note

If a function is called in SCL whose return value was not supplied, this can lead to incorrect execution of the user program.

In an SCL function, this situation can occur when the return value was supplied but the corresponding statement is not executed.
In an STL/LAD/FBD function, this situation can occur when the function was programmed without supplying the return value or the corresponding statement is not executed.

Calling

A function is called by specifying:

- the function name (FC IDENTIFIER, SFC IDENTIFIER, IDENTIFIER)
- the parameter list.

Example

The function name which identifies the return value can be specified in absolute or symbolic terms as shown in the following examples:

```
FC31        (X1:=5, Q1:= Digitsum)
DISTANCE    (X1:=5, Q1:= Digitsum)
```

Results of the Function Call

The results of a function call are available after execution of the call in the form of

- a return value or
- output or in/out parameters (actual parameters)

For more detailed information on this subject, refer to Chapter 18.

16.3.1 FC Parameters

Basic Principle

In contrast to function blocks, functions do not have any memory in which to store the values of parameters. Local data is only stored temporarily while the function is active. For this reason, all formal input, in/out and output parameters defined in the declaration section of a function must be assigned actual parameters as part of the function call.

Figure 16-11 below shows the syntax for FC parameter assignment.

FC Parameter

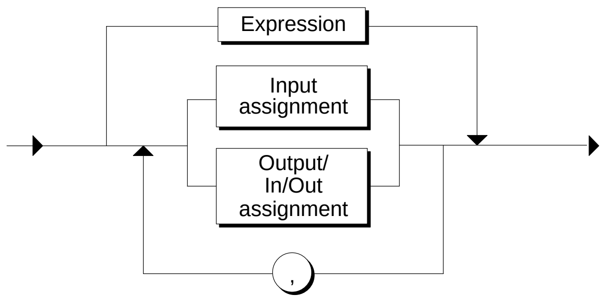


Figure 16-11 Syntax of an FC Parameter

The example below illustrates a function call involving assignment of an input parameter, an output parameter and an in/out parameter.

```
FC32 (I_Param1:=5,IO_Param1:=LENGTH,
      O_Param1:=Digitsum)
```

Rules

The rules for assigning values to parameters are as follows:

- The value assignments can be in any order.
- The data type of the formal and actual parameter in each case must match.
- The individual assignments must be separated by commas.

16.3.2 Input Assignment (FC)

Basic Principle

Input assignments assign values (actual parameters) to the formal input parameters of the called FC. The FC can work with these actual parameters but cannot change them. In contrast to an FB call, this assignment is **not optional** with an FC call. Input assignments have the follwing syntax:

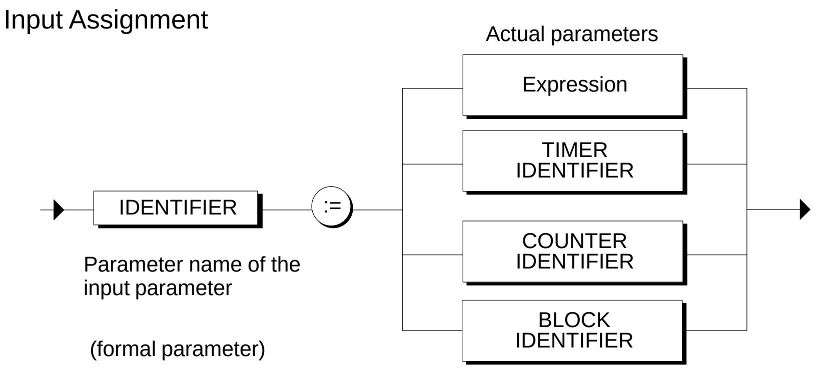


Figure 16-12 Syntax of an Input Assignment

Actual Parameters in Input Assignments

The following actual parameters can be assigned in input assignments:

Table 16-4 Actual Parameters in Input Assignments

Actual Parameter	Explanation
Expression	An expression represents a value and consists of addresses and operators. The following types of expression are possible: Mathematical, logical or comparative expressions Constants Extended variables
TIMER/COUNTER Identifier	Defines a specific timer or counter to be used when a block is processed (see also Chapter 17).
BLOCK Identifier	Defines a specific block to be used as an input parameter. The block type (FB, FC or DB) is specified in the declaration of the input parameter. When assigning parameters, you specify the block address. You can use either the absolute or the symbolic address (see also Chapter 9).

Special Consideration

Note that FB in/out parameters and FC parameters are not permissible as actual parameters for formal FC input parameters of a non-elementary data type.

16.3.3 Output and In/Out Assignment (FC)

Basic Principle

In an output assignment, you specify where the output values resulting from processing a function are to be written to. An in/out assignment is used to assign an actual value to an in/out parameter.

Figure 16-13 below shows the syntax of output and in/out assignments.

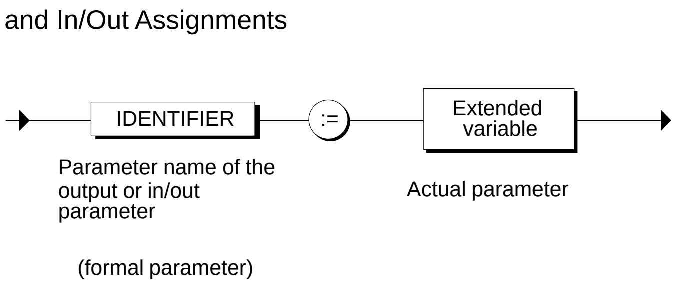

Output and In/Out Assignments
IDENTIFIER
:=
Extended variable
Parameter name of the output or in/out parameter
(formal parameter)
Actual parameter

Figure 16-13 Syntax of Output and In/Out Assignments

Actual Parameters in Output and In/Out Assignments

The actual parameters in output and in/out assignments must be variables since the FC writes values to the parameters. For this reason, input parameters can not be assigned in in/out assignments (the value could not be written).

Thus, only extended variables can be assigned in output and in/out assignments.

Table 16-5 Actual Parameters in Output and In/Out Parameters

Actual Parameter	Explanation
Extended variable	The following types of extended variable can be used: Simple variables and parameters Access to absolute variables Access to data blocks Function calls (see also Chapter 14).

Special Considerations

Note the following special considerations:

- After the block is processed, the altered value of the in/out parameter is updated.
- The following are not permitted as actual parameters for in/out parameters of a **non elemenatary** data type:
  - FB input parameters
  - FB in/out parameters and
  - FC parameters
- **ANY parameters:** The first point made above also applies here. The following are not permitted as actual parameters for in/out parameters of a **non elemenatary** data type:
  - FB input parameters
  - FC input parameters

    In addition, constants are **not** permitted as actual parameters.
    If the ANY type is declared as a function result (return value), the following also applies:
  - All ANY parameters must be supplied with addresses whose data types are within a type class. By type class is meant the number of numerical data types (INT, DNIT, REAL) or the number of bit data types (BOOL, BYTE, WORD, DWORD) is meant. The other data types each make up their own type class.
  - The SCL Compiler assumes that the data type of the current function result will be given as the highest-level type among the actual parameters which are assigned to the ANY parameters.
    With the function result, all operations are permitted which are defined for this data type.
- **POINTER-parameter:** The first point made above also applies here. The following are not permitted as actual parameters for in/out parameters of a **non elemenatary** data type:
  - FB input parameters
  - FC input parameters

16.3.4 Example of a Function Call

Basic Principle

A function `DISTANCE` for calculating the distance between two points (X1,Y1) and (X2,Y2) in the same plane using the Cartesian system of co-ordinates might take the following form (the examples assume that the symbol `DISTANCE` has been declared in a symbol table for `FC37`).

```
FUNCTION DISTANCE: REAL
      VAR_INPUT
            X1: REAL;
            X2: REAL;
            Y1: REAL;
            Y2: REAL;
      END_VAR
      VAR_OUTPUT
            Q2: REAL;
      END_VAR
      BEGIN
            DISTANCE:= SQRT
            ( (X2-X1)**2 + (Y2-Y1)**2 );
            Q2:= X1+X2+Y1+Y2;
END_FUNCTION
```

Example 16-6 Distance Calculation

The examples below show further options for subsequent use of a function value:

In a value assignment, for example

```
LENGTH:= DISTANCE (X1:=-3, Y1:=2, X2:=8.9,
Y2:=7.4, Q2:=Digitsum);
```

In a mathematical or logical expression, for example

```
RADIUS + DISTANCE (X1:=-3, Y1:=2, X2:=8.9,
Y2:=7.4, Q2:=Digitsum)
```

When assigning values to parameters in a called block, for example

```
FB32 (DIST:= DISTANCE (X1:=-3, Y1:=2, X2:=8.9,
Y2:=7.4, Q2:=Digitsum);
```

Example 16-7 Calculation of Values in an FC

16.4 Implicitly Defined Parameters

Overview

Implicitly defined parameters are parameters that you can use without having to declare them first in a block. SCL provides the following implicitly defined parameters:

- the input parameter `EN` and
- the output parameter `ENO`

Both parameters are of the data type `BOOL` and are stored in the temporary block data area.

Input Parameter EN

Every function block and every function has the implicitly defined input parameter EN. If EN is TRUE, the called block is executed. Otherwise it is not executed. Supplying a value for the parameter EN is optional.

Remember, however, that EN must not be declared in the declaration section of a block or function.

Since EN is an input parameter, you cannot change EN within a block.

Note

The return value of a function is not defined if the function is not called because `EN:=FALSE.`

Example

The following example illustrates the use of the parameter EN:

```
FUNCTION_BLOCK FB57
VAR
      RESULT      : REAL;
      MY_ENABLE   : BOOL;
END_VAR
...
BEGIN
MY_ENABLE:= FALSE;
// Function call
// in which the parameter EN is assigned a value:

RESULT:= FC85 (EN:= MY_ENABLE, PAR_1:= 27);
// FC85 not executed because MY_ENABLE
// is set to FALSE
//...
END_FUNCTION_BLOCK
```

Example 16-8 Use of EN

Output Parameter ENO

Every function block and every function has the implicitly defined output parameter `ENO` which is of the data type BOOL. When the execution of a block is completed, the current value of the `OK` variable is set in `ENO`.

Immediately after a block has been called you can check the value of `ENO` to see whether all the operations in the block ran correctly or whether errors occurred.

Example

The following example illustrates the use of the parameter EN0:

```
FUNCTION_BLOCK FB57
      //...
      //...
BEGIN
// Function block call:
FB30.DB30 (X1:=10, X2:=10.5);

// Check to see if all
// operations performed properly:

IF ENO THEN
      // Everything OK
      //...
ELSE
      // Error occurred,
      // therefore error handling
      //...
END_IF;
      //...
      //...
END_FUNCTION_BLOCK
```

Example 16-9 Use of ENO

Example

The following example shows the combination of EN and ENO:

```
// EN and ENO can also be combined
// as shown here:

FB30.DB30(X1:=10, X2:=10.5);

// The following function is only
// to be executed if FB30 is
// processed without errors

RESULT:= FC85 (EN:= ENO, PAR_1:= 27);
```

Example 16-10 Use of EN and ENO

17 Counters and Timers

Introduction

In SCL you can control the running of a program on the basis of a timer or counter reading.

STEP 7 provides standard counter and timer functions for this purpose which you can use in your SCL program without having to declare them beforehand.

Chapter Overview

17.1 Counter Functions

Overview

STEP 7 provides a series of standard counter functions. You can use these counters in your SCL program without needing to declare them previously. You must simply supply them with the required parameters. STEP 7 provides the following counter functions:

- Counter Up
- Counter Down
- Counter Up/Down

Calling

Counter functions are called just like functions. The function identifier can therefore be used as an address in an expression provided you make sure that the data type of the function result is compatible with the address replaced.

Table 17-1 Function Name of Counter Functions

Function Name	Description
S_CU	Counter Up
S_CD	Counter Down
S_CUD	Counter Up/Down

Function Value

The function value (return value) which is returned to the calling block is the current counter reading `(BCD format)` in data type `WORD.` For more information on this subject, refer to Section 17.1.1.

Function Call Parameters

The function call parameters for all three counter functions are listed in Table 17-2 together with their identifiers and descriptions. Basically, the following types of parameters should be distinguished:

- Control parameters (for example, set, reset, counting direction)
- Initialization value for a counter reading
- Status output (shows whether a counter limit has been reached).
- Counter reading in binary form

Table 17-2 Counter Function Call Parameters

Identifier	Parameter	Data Type	Description
C_NO		COUNTER	Counter number (COUNTER IDENTIFIER); the area depends on the CPU
CU	Input	BOOL	CU input: count up
CD	Input	BOOL	CD input: count down
S	Input	BOOL	Input for presetting the counter
PV	Input	WORD	Value in the range between 0 and 999 for initializing the counter (entered as 16#<value>, with the value in BCD format)
R	Input	BOOL	Reset input
Q	Output	BOOL	Status of the counter
CV	Output	WORD	Counter reading (binary)

Example

The counter function call shown in Example 17-1 below causes a global memory area of the type COUNTER with the name C12 to be reserved when the function is configured.

```
Counter_Reading:= S_CUD (C_NO :=C12,
                        CD    :=I.0,
                        CU    :=I.1,
                        S     :=I.2 & I.3,
                        PV    :=120,
                        R     :=FALSE,
                        CV    :=binVal,
                        Q     :=actFlag);
```

Example 17-1 Calling a Counter Down Function

Calling Dynamically

Instead of the absolute counter number (for example,`C_NO=C10`), you can also specify a variable of the data type COUNTER to call the function. The advantage of this method is that the counter function call can be made dynamic by assigning that variable a different absolute number in each function call.

```
Example:
Function_Block COUNTER;
Var_Input
 MyCounter: Counter;
End_Var
:
currVAL:=S_CD (C_NO:=MyCounter,........);
```

Rules

Since the parameter values (for example, CD:=I.0) are stored globally, under certain circumstances specifying those parameters is optional. The following general rules should be observed when supplying parameters with values:

- The parameter for the counter identifier C_NO must always be assigned a value when the function is called.
- Either the parameter CU (up counter) or the parameter CD (down counter) must be assigned a value, depending on the counter function required.
- The parameters PV (initialization value) and S (set) can be omitted as a pair.
- The result value in BCD format is always the function value.

Note

The names of the functions and parameters are the same in both SIMATIC und IEC mnemonics. Only the counter identifier is mnemonic-dependent, thus: *SIMATIC:* Z and *IEC:* C

Example of Counter Function Call

Example 17-2 below illustrates various counter function calls:

```
Function_block FB1


VAR
 currVal, binVal: word;
 actFlag: bool;
END_VAR

BEGIN
currVal   :=S_CD(C_NO:=C10, CD:=TRUE, S:=TRUE,
                PV:=100, R:=FALSE, CV:=binVal,
                Q:=actFlag);

currVal   :=S_CU(C_NO:=C11, CU:=M0.0, S:=M0,1,
                PV:=16#110, R:=M0.2, CV:=binVal,
                Q:=actFlag);

currVal   :=S_CUD(C_NO:=C12, CD:=E.0,
                CU:=I.1,S:=I.2 & I.3, PV:=120,
                R:=FALSE,CV:=binVal, Q:=actFlag);

currVal   :=S_CD(C_NO:=C10,CD:=FALSE,
                S:=FALSE,
                PV:=100, R:=TRUE, CV:=bVal,
                Q:=actFlag);


END_FUNCTION_BLOCK
```

Example 17-2 Counter Function Calls

17.1.1 Input and Evaluation of the Counter Reading

Overview

To input the initialization value or to evaluate the result of the function, you require the internal representation of the counter reading (see Figure 17-1).

When you set the counter (parameter S), the value you specify is written to the counter. The range of values is between 0 and 999. You can change the counter reading within this range by specifying the operations count up/down or count up and down

Format

Figure 17-1 below illustrates the bit configuration of the counter reading.

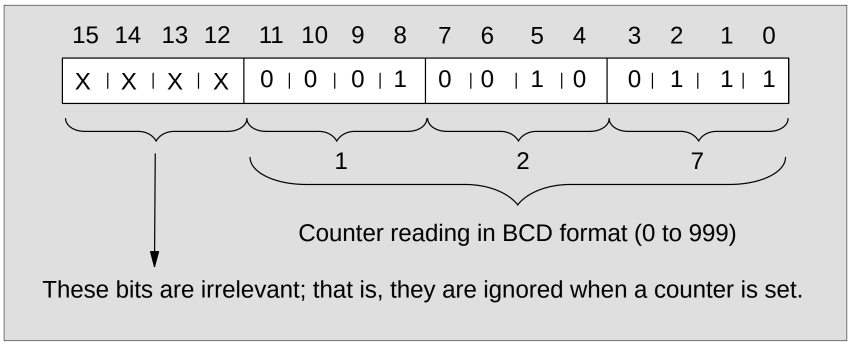

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X 0 0 0 1 0 0 1 0 0 1 1 1
1
2
7
Counter reading in BCD format (0 to 999)
These bits are irrelevant; that is, they are ignored when a counter is set.

Figure 17-1 Bit Configuration of Counter Reading

Input

You can load a predefined counter reading using the following formats:

- Decimal integer: for example 295 if that value corresponds to a valid BCD code
- BCD code (input as a hexadecimal constant): for example 16#127

Evaluation

You can evaluate the result in two different formats:

- As a function result (type WORD) in BCD format
- As the output parameter CV (type WORD) in binary code

17.1.2 Counter Up (CU)

Description

With the Counter Up function, you can only perform upward counting operations.

Method of Operation

Table 17-3 Counter Up Function

Operation	Explanation
Counter up	The counter reading is increased by ”1” when the signal status at input **CU** changes from ”0” to ”1” and the count value is less than 999.
Set counter	When the signal status at input **S** changes from ”0” to ”1”, the counter is set to the value of input **PV**. Such a signal change is always required to set a counter.
Reset	The counter is reset when input **R** = 1 is set. Resetting the counter sets the counter reading to ”0”.
Query counter	A signal status query at output Q returns ”1” if the counter reading is greater than ”0”. The query returns ”0” if the counter reading is equal to ”0”.

17.1.3 Counter Down (CD)

Description

With the Counter Down function, you can only execute downward counting operations.

Method of Operation

Table 17-4 Counter Down Function

Function	Explanation
Counter down	The counter reading is decreased by ”1” if the signal status at input **CD** changes from ”0” to ”1” and the count value is greater than ”0”.
Set counter	If the signal status at input **S** changes from ”0” to ”1”, the counter is set to the value of input **PV**. Such a signal change is always required to set a counter.
Reset	The counter is reset if input **R** = 1 is set. Resetting the counter sets the count value to ”0”.
Query counter	A signal status query at output **Q** returns ”1” if the counter reading is greater than ”0”. The query returns ”0” if the counter reading is equal to ”0”.

17.1.4 Counter Up/Down (CUD)

Description

With the Counter Up/Down function, you can execute both upward and downward counting operations. If up and down count pulses are received simultaneously, both operations are performed. The counter reading remains unchanged.

Method of Operation

Table 17-5 Up/Down Counter Function

Function	Function
Counter up	The counter reading is increased by ”1” if the signal status at input **CU** changes from ”0” to ”1” and the counter reading is less than 999.
Counter down	The counter reading is decreased by ”1” if the signal status at input **CD** changes from ”0” to ”1” and the counter reading is greater than ”0”.
Set counter	If the signal status at input **S** changes from ”0” to ”1”, the counter is set to the value of input **PV**. Such a signal change is always required to set a counter.
Reset	The counter is reset if input **R** = 1 is set. Resetting the counter sets the counter reading to ”0”.
Query counter	A signal status query at output Q returns ”1” if the counter reading is greater than ”0”. The query returns ”0” if the counter reading is equal to ”0”.

17.1.5 Example of the Function S_CD (Counter Down)

Parameter Assignment

Table 17-6 below illustrates parameter assignment for the function S_CD.

Table 17-6 Function Call Parameters

Parameter	Description
C_NO	`MyCounter`
CD	Input I0.0
S	`SET`
PV	`Initiliazation 16#0089`
R	`Reset`
Q	`Q0.7`
CV	`BIN_VAL`

Example

Example 17-3 illustrates use of the counter function S_CD:

```
FUNCTION_BLOCK COUNT
VAR_INPUT
 MYCOUNTER: COUNTER;
END_VAR
VAR_OUTPUT
 RESULT: INT;
END_VAR
VAR
 SET        : BOOL;
 RESET      : BOOL;
 BCD_VALUE        : WORD; //counter reading BCD
coded
 BIN_VALUE        : WORD; //counter reading
binary
 INITIALIZATION   : WORD;
END_VAR
BEGIN
 Q0.0:= 1;
 SET:= I0.2;
 RESET:= I0.3;
 INITIALIZATION:= 16#0089;
 BCD_VALUE:= S_CD
            (C_NO := MYCOUNTER,//COUNT UP.
            CD    := I.0,
            S     := SET,
            PV    := INITIALIZATION,
            R     := RESET,
            CV    := BIN_VALUE,
            Q     := Q0.7);
RESULT := WORD_TO_INT (BIN_VALUE);//further
                 //processing as an output
                 //parameter
QW4 := BCD_VALUE  //to output for display
END_FUNCTION_BLOCK
```

Example 17-3 Example of Counter Function

17.2 Timer Functions

Overview

Timers are functional elements in your program that perform and monitor timed processes. STEP 7 provides a series of standard timer functions which you can access using SCL. You can use timer operations to

- set delay periods
- enable monitoring periods
- generate pulses
- measure times

Calling

Timer functions are called in the same way as counter functions. The function identifier can be used in any expression in place of an address provided the data type of the function result is compatible with that of the address replaced.

Table 17-7 STEP 7 Timer Functions

Function Name	Description
S_PULSE	Pulse timer
S_PEXT	Extended pulse timer
S_ODT	On-delay timer
S_ODTS	Retentive on-delay timer
S_OFFDT	Off-delay timer

Function Value

The function value (return value) that is returned to the calling block is a time value of the data type `S5TIME`. For more information on this subject, refer to Section 17.2.1

Function Call Parameters

The parameters that have to be assigned values are listed in a table in the description of the standard function concerned. The function names and corresponding data types for all 5 timer functions are given in Table 17-8.

In general, the following types of parameter should be distinguished:

- Control parameters (for example, set, reset)
- Initialization value for start time
- Status output (indicates whether timer is running)
- Remaining time in binary form

Table 17-8 Function Call Parameters

Parameter	Data Type	Description
T_NO	TIMER	Identification number of the timer; the range depends on the CPU
S	BOOL	Start input
TV	S5TIME	Initialization of the timer reading (BCD format)
R	BOOL	Reset input
Q	BOOL	Status of the timer
BI	WORD	Time remaining (binary)

Example

The timer function call shown in Example 17-4 causes a global memory area of the type TIMER and with the name T10 to be reserved when the function is processed.

```
DELAY:=       S_ODT (T_NO :=    T10,
                     S    :=    TRUE,
                     TV   :=    T#1s,
                     R    :=    FALSE,
                     BI   :=    biVal,
                     Q    :=    actFlag
                     );
```

Example 17-4 Timer Function Call

Calling Dynamically

Instead of the absolute timer number (for example, T10), you can also specify a variable of the data type TIMER in the function call. The advantage of this is that the timer function call is made dynamic by assigning that variable a different absolute number in every function call.

```
Example:
FUNCTION_BLOCK TIMERUNIT
VAR_INPUT
MyTimer: timer;
END_VAR
:
currTime:=S_ODT (T_NO:=MyTimer,.........)
```

Rules

Since the parameter values are stored globally, under certain circumstances specifying those values is optional. The following general rules should be observed when assigning values to parameters:

- The parameter for the timer identifier T_NO must be assigned a value in symbolic or absolute form in the function call.
- The parameters TV (initialization value) and S (set) can be omitted as a pair.
- Reading of parameter values is optional. You can access Q and BI by means of a value assignment.
- The result in S5TIME format is always the function value.

Note

The names of the functions are the same in both SIMATIC and IEC mnemonics.

Example Timer Function Call

Example 17-5 below illustrates various timer function calls:

```
FUNCTION_BLOCK FB2

VAR
currTime: S5time;
biVal: word;
actFlag: bool;
END VAR

BEGIN
currTime:= S_ODT (T_NO:=T10, S:=TRUE, TV:=T#1s,
                  R:=FALSE, BI:=biVal,
                  Q:=actFlag);
```

```
currTime:= S_ODTS (T_NO:=T11, S:=M0,0, TV:=T#1s,
                  R:= M0.1, BI:=biVal,
                  Q:= actFlag);
```

```
currTime:=S_OFFDT (T_NO:=T12, S:=I0.1&actFlag,
                  TV:= T#1s,R:=FALSE,BI:=biVal,
                  Q:= actFlag);
```

```
currTime:= S_PEXT (T_NO:=T13, S:=TRUE,
                  TV:=T#1s,R:=I0.0, BI:=biVal,
                  Q:=actFlag);
```

```
currTime:= S_PULSE (T_NO:=T14, S:=TRUE,
                   TV:=T#1s,R:=FALSE, BI:=biVal,
                    Q:=actFlag);
END_FUNCTION_BLOCK
```

Example 17-5 Timer Function Calls

17.2.1 Input and Evaluation of the Timer Reading

Overview

To input the initialization value and to evaluate the function result in BCD code, you require the internal representation of the timer reading (see Figure 17-2).

Updating the time decreases the timer reading by 1 unit in 1 interval as specified by the time base. The timer reading is decreased until it reaches ”0”. The possible range of time is from 0 to 9,990 seconds.

Format

Figure 17-2 shows the internal representation of the timer reading.

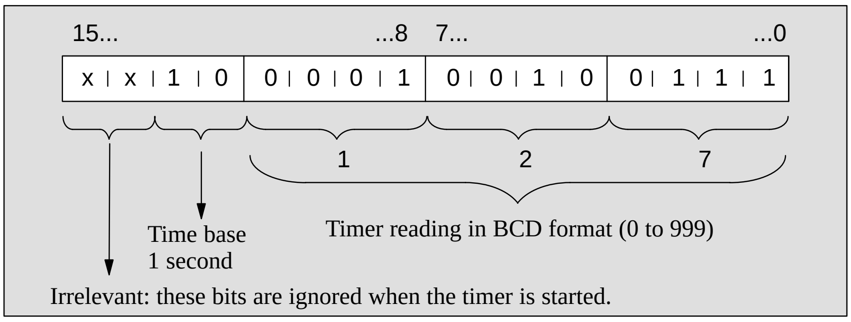

15...
...8
7...
...0
x x 1 0 0 0 0 1 0 0 1 0 0 1 1 1
1
2
7
Time base
1 second
Timer reading in BCD format (0 to 999)
Irrelevant: these bits are ignored when the timer is started.

Figure 17-2 Format of Timer Reading

Input

You can load a predefined timer reading using the following formats:

- In composite time format: `TIME#aH_bbM_ccS_dddMS`
- In simple format: `TIME#2.4H`

Evaluation

You can evaluate the result in two different formats:

- As a function result (type S5TIME): in BCD format
- As an output parameter (time without time base in data type `WORD`): in binary code

Time Base

Bits 12 and 13 of the timer word contain the time base in binary code. The time base defines the interval at which the time value is decreased by 1 unit (see Table 17-9 and Figure 17-2). The shortest time base is 10 ms; the longest is 10 s.

Table 17-9 Time Base and Binary Code

Time Base	Binary Code for Time Base
10 ms	00
100 ms	01
1 s	10
10 s	11

Note

Since timer readings can only be saved in one time interval, values that do not represent an exact multiple of the time interval are truncated.

Values with a resolution too high for the required range are rounded down so that the required range is achieved but **not** the required resolution.

17.2.2 Pulse Timer

Description

The maximum time for which the output signal remains set to ”1” is the same as the programmed timer reading.

If, during the runtime of the timer, the signal status 0 appears at the input, the timer is set to ”0”. This means a premature termination of the timer runtime.

Figure 17-3 shows how the ”pulse timer” function works:

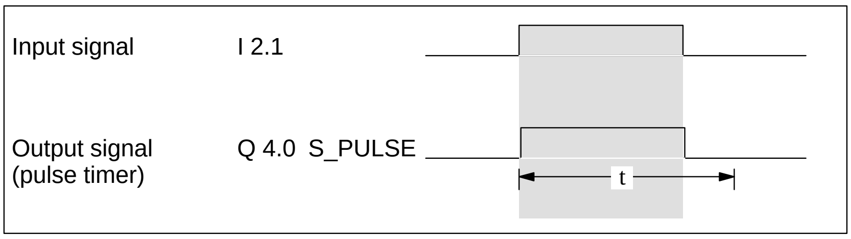


Figure 17-3 Pulse Timer

Method of Operation

Table 17-10 Method of Operation of Pulse Timer

Function	Explanation
Start time	The ”pulse timer” operation starts the specified timer when the signal status at the start input (**S**) changes from ”0” to ”1”. To enable the timer, a signal change is always required.
Specify runtime	The timer runs using the value at input **TV** until the programmed time expires and the input S = 1.
Abort runtime	If input **S** changes from ”1” to ”0” before the time has expired, the timer is stopped.
Reset	The time is reset if the reset input (**R**) changes from ”0” to ”1” while the timer is running. With this change, both the timer reading and the time base are reset to zero. The signal status ”1” at input R has no effect if the timer is not running.
Query signal status	As long as the timer is running, a signal status query following a ”1” at output **Q** produces the result ”1”. If the timer is aborted, a signal status query at output Q produces the result ”0”.
Query current timer reading	The current timer reading can be queried at output **BI** and using the function value S_PULSE.

17.2.3 Extended Pulse Timer

Description

The output signal remains set to ”1” for the programmed time (t) regardless of how long the input signal remains set to ”1”. Triggering the start pulse again restarts the counter time so that the output pulse is extended (retriggering).

Figure 17-4 shows how the ”extended pulse timer” function works:

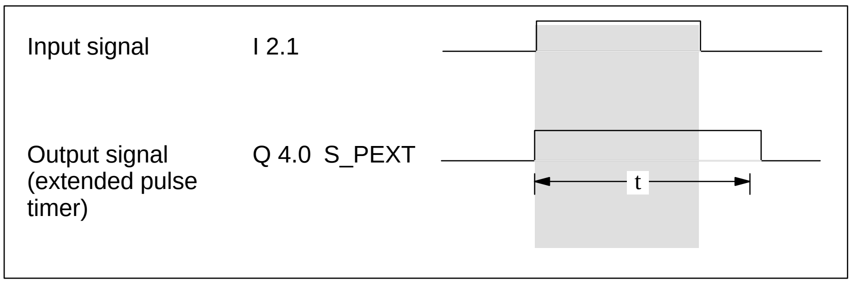


Figure 17-4 Extended pulse timer

Method of Operation

Table 17-11 Method of Operation of Extended Pulse Timer

Function	Explanation
Start time	The ”extended pulse timer” (S_PEXT) operation starts the specified time when the signal status at the start input (**S**) changes from ”0” to ”1”. To enable the timer, a signal change is always required.
Restart the counter time	If the signal status at input **S** changes to ”1” again while the timer is running, the timer is restarted with the specified timer reading.
Initialize runtime	The timer runs with the value at input **TV** until the programmed time has expired.
Reset	The time is reset if the reset input (**R**) changes from ”0” to ”1” while the timer is running. With this change, both the timer reading and the time base are reset to zero. The signal status ”1” at input R has no effect if the timer is not running.
Query signal status	As long as the timer is running, a signal status query following ”1” at output **Q** produces the result ”1” regardless of the length of the input signal.
Query current timer reading	The current timer reading can be queried at output **BI** and using the function value S_PEXT.

17.2.4 On-Delay Timer

Description

The output signal only changes from ”0” to ”1” if the programmed time has expired and the input signal is still ”1”. This means that the output is activated following a delay. Input signals that remain active for a time that is shorter than the programmed time do not appear at the output.

Figure 17-5 illustrates how the ”on-delay timer” function works.

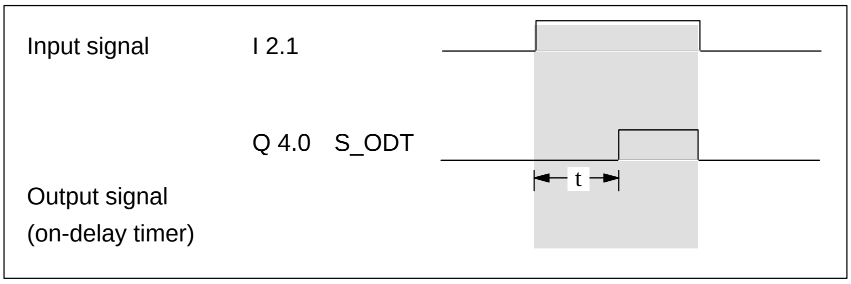


Figure 17-5 On-Delay Timer

Method of Operation

Table 17-12 Method of Operation of On-Delay Timer

Function	Explanation
Start time	The ”on-delay timer” starts a specified time if the signal status at the start input (**S**) changes from ”0” to ”1”. To enable the timer, a signal change is always required.
Stop timer	If the signal status at input S changes from ”1” to ”0” while the timer is running, it is stopped.
Specify the runtime	The timer continues to run with the value at input **TV** as long as the signal status at input S = 1.
Reset	The timer is reset if the reset input (**R**) changes from ”0” to ”1” while the timer is still running. With this signal change, the timer reading and the time base are reset to zero. The time is also reset if R = 1 is set when the timer is not running.
Query signal status	A signal status query following ”1” at output Q returns ”1” if the time has expired without an error occurring and input S is still set to ”1”. If the timer is stopped, a signal status query following ”1” always returns ”0”. A signal status query after ”1” at output Q also returns ”0” if the timer is not running and the RLO at input S is still ”1”.
Query current timer reading	The current timer reading can be queried at output **BI** and using the function value S_ODT.

17.2.5 Retentive On-Delay Timer

Description

The output signal only changes from ”0” to ”1” if the programmed time has expired regardless of how long the input signal remains set to ”1”.

Figure 17-6 shows how the ”retentive on-delay timer” function works.

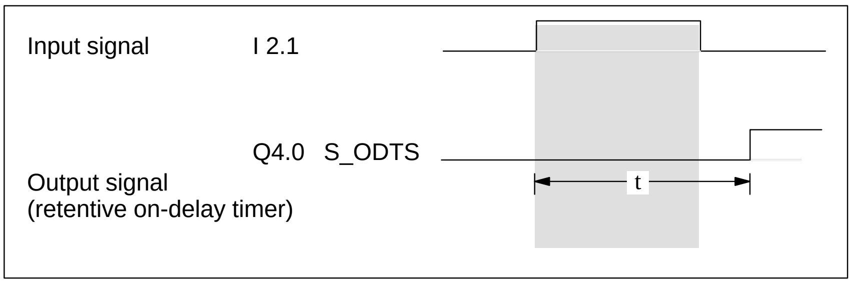


Figure 17-6 Retentive On-Delay Timer

Method of Operation

Table 17-13 Method of Operation of Retentive On-Delay Timer

Function	Explanation
Start time	The ”stored on-delay timer” function starts a specified timer if the signal status at the start input (**S**) changes from ”0” to ”1”. To enable the timer, a signal change is always required.
Restart timer	The timer is restarted with the specified value if input **S** changes from ”0” to ”1” while the timer is running.
Specify runtime	The timer continues to run with the value at input **TV** even if the signal status at input **S** changes to ”0” before the time has expired.
Reset	If the reset input (**R**) changes from ”0” to ”1”, the timer is reset regardless of the RLO at input S.
Query signal status	A signal status query following ”1” at output **Q** returns the result ”1” after the time has expired regardless of the signal status at input S.
Query current timer reading	The current timer reading can be queried at output BI and using the function value S_ODTS.

17.2.6 Off-Delay Timer

Description

With a signal status change from ”0” to ”1” at start input S, output Q is set to ”1”. If the start input changes from ”1” to ”0”, the timer is started. The output only returns to signal status ”0” after the time has expired. The output is therefore deactivated following a delay.

Figure 17-7 shows how the ”off-delay timer” function works.

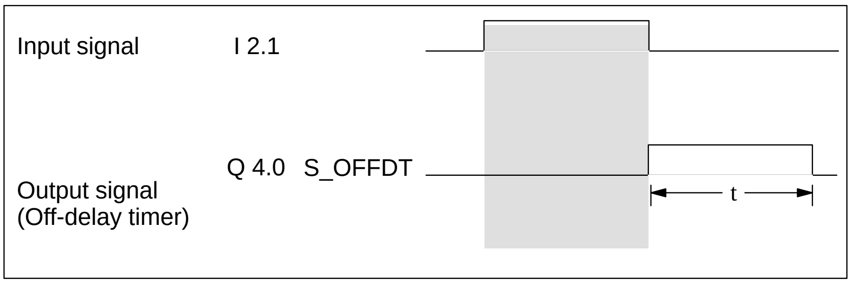


Figure 17-7 Off-Delay Timer

Method of Operation

Table 17-14 Method of Operation of Off-Delay Timer

Function	Explanation
Start time	The ”off-delay timer” operation starts the specified timer if the signal status at the start input (**S**) changes from ”1” to ”0”. A signal change is always required to enable the timer.
Restart timer	The timer is restarted if the signal status at input S changes from ”1” to ”0” again (for example following a reset).
Specify runtime	The timer runs with the value specified at input **TV**.
Reset	If the reset input (R) changes from ”0” to ”1” while the timer is running, the timer is reset.
Query signal status	A signal status query following ”1” at output Q produces ”1” if the signal status at input S = 1 or the timer is running.
Query current timer reading	The current timer reading can be queried at output **BI** and using the function value S_OFFDT.

17.2.7 Example of Program Using Extended Pulse Timer Function

Example of S_PEXT

Example 17-6 below illustrates a program using the extended pulse timer function.

```
FUNCTION_BLOCK TIMER
VAR_INPUT
 MYTIME: TIMER;
END_VAR
VAR_OUTPUT
 RESULT: S5TIME;
END_VAR
VAR
 SET        : BOOL;
 RESET      : BOOL;
 BCD_VALUE        : S5TIME;//time base and time
                         //remaining
                    //BCD coded
 BIN_VALUE          WORD;  //timer reading
binary
 INITIALIZATION   : S5TIME;
END_VAR
BEGIN
 Q0.0:= 1;
 SET:= I0.0;
 RESET:= I0.1;
 INITIALIZATION:= T#25S;
               ;
```

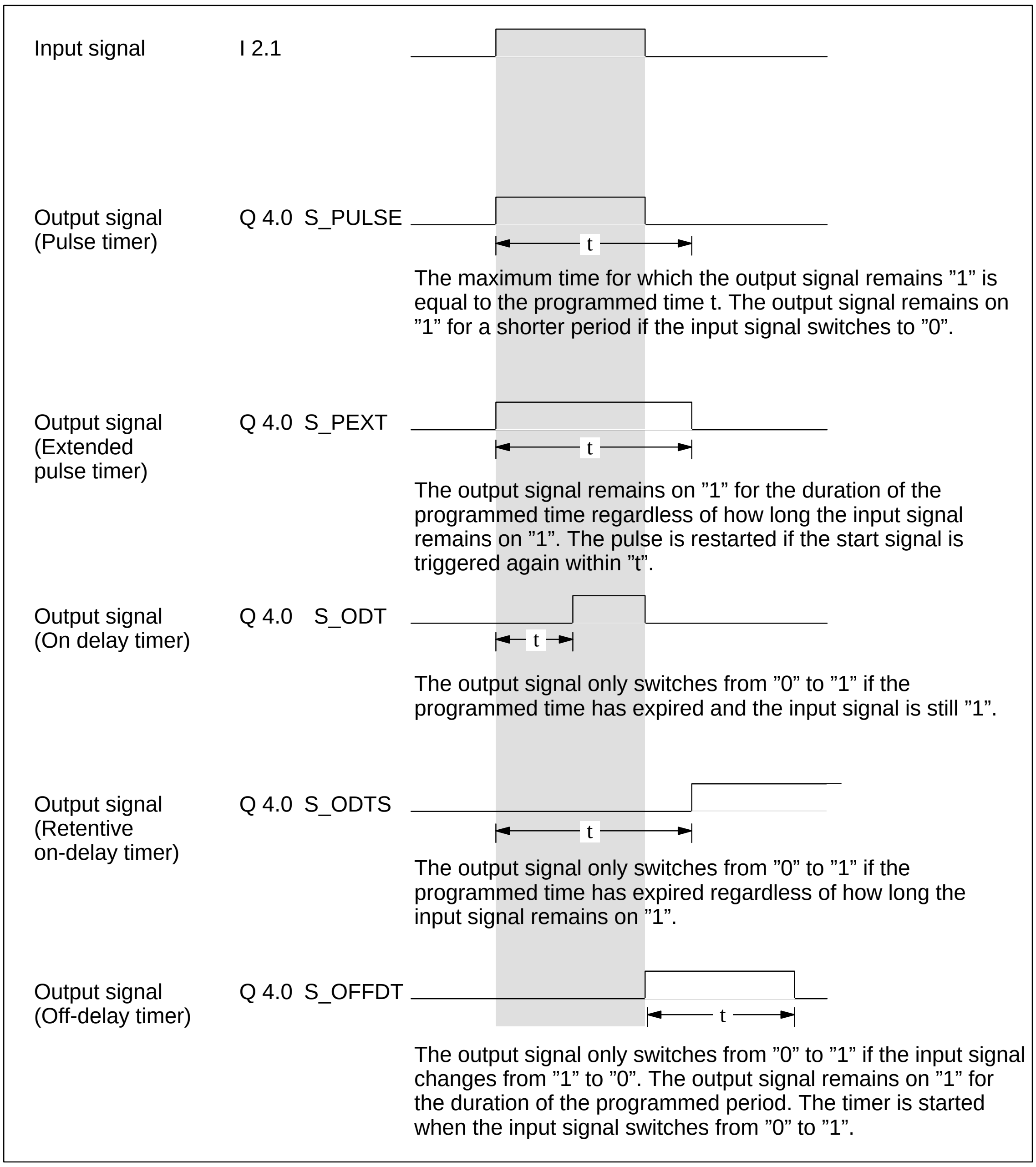
```
 BCD_VALUE:= S_PEXT(T_NO:= MYTIME,
                   S   := SET,
                   TV  := INITIALIZATION,
                   R   := RESET,
                   BI  := BIN_VALUE,
                   Q   := Q0.7);
```

```
RESULT:=BCD_VALUE; //Further processing
                   //as output parameter
QW4:= BIN_VALUE   //To output for display
END_FUNCTION_BLOCK
```

Example 17-6 Timer Function

17.2.8 Selecting the Right Timer Function

Figure 17-8 summarizes the five different timer functions described in this chapter. This summary is intended to assist you in selecting the timer function best suited to your particular purpose.



The maximum time for which the output signal remains ”1” is equal to the programmed time t. The output signal remains on ”1” for a shorter period if the input signal switches to ”0”.

The output signal remains on ”1” for the duration of the programmed time regardless of how long the input signal remains on ”1”. The pulse is restarted if the start signal is triggered again within ”t”.

The output signal only switches from ”0” to ”1” if the programmed time has expired and the input signal is still ”1”.

The output signal only switches from ”0” to ”1” if the programmed time has expired regardless of how long the input signal remains on ”1”.

The output signal only switches from ”0” to ”1” if the input signal changes from ”1” to ”0”. The output signal remains on ”1” for the duration of the programmed period. The timer is started when the input signal switches from ”0” to ”1”.

Figure 17-8 Selecting the Right Timer Function

SCL Standard Functions 18

Introduction

SCL provides a series of standard functions for performing common tasks which can be called by the SCL blocks you program.

Chapter Overview

18.1 Converting Data Types

Overview

When you link two addresses of differing data types or assign expressions to variables, you must check the mutual compatibility of the data types involved in each case. The following cases would produce incorrect results:

- a change to a different type class, for example, from a bit data type to a numeric data type;
- a change within a type class if the destination data type is of a lower order than the source data type.

Therefore, in such cases you must perform an **explicit** data type conversion. The necessary details are given in Section 18.2.

If neither of the above cases applies, the compiler forces automatic conversion to a common format. This type of conversion is referred to from now on as **implicit** data type conversion.

Implicit Data Type Conversions

Within the classes of auxiliary data type listed in Table 18-1, the compiler performs implicit data type conversions in the order indicated. The common format of two addresses is taken to be the lowest common standard type whose value range covers both addresses. Thus, the common format of Byte and Integer is Integer.

Please note also that in the case of data type conversion within the class `ANY_BIT`, leading bits are set to `0`.

Table 18-1 Order of Implicit Data Type Conversions

Class	Conversion Order
ANY_BIT	`BOOL` ⇒ `BYTE` ⇒ `WORD` ⇒ `DWORD`
ANY_NUM	`INT` ⇒ `DINT` ⇒ `REAL`

Example 18-1 illustrates implicit conversion of data types.

```
FUNCTION_BLOCK FB10
VAR
      PID_CONTROLLER_1:BYTE;
      PID_CONTROLLER_2:WORD;
END_VAR

BEGIN
IF (PID_CONTROLLER_1 <> PID_CONTROLLER_2) THEN...
(* In the condition for the above IF/THEN
instruction, PID_ CONTROLLER_1 is implicitly
converted to a variable of data type WORD *)
END_FUNCTION_BLOCK
```

Example 18-1 Implicit Data Type Conversion

18.2 Standard Functions for Data Type Conversions

Explicit Data Type Conversion

Explicit data type conversions are performed by means of standard functions. These standard functions are listed in Tables 18-2 and 18-3.

Function Call

For a detailed description of the function call, refer to Chapter 16.

- Input parameter:

Each function for converting a data type has one input parameter only. This parameter has the name `IN`. Since this is a function with only one parameter, you only need to specify the actual parameter.

- Function value

The result is always the function value. The two tables detail the rules according to which the data is converted. Table 18-3 also indicates whether or not the function affects the `OK` flag.

- Names of the functions

Since the data types of the input parameter and the function value are derived from the function name in each case, they are not separately itemized in Tables 18-2 and 18-3. For example, for the function `BOOL_TO_BYTE,` the data type of the input parameter is `BOOL` and the data type of the function value `BYTE`.

List of Conversion Functions (Class A)

Table 18-2 shows the data type conversion functions of Class A. These functions are performed implicitly by the compiler or you can specify them explicitly. The result is always defined.

Table 18-2 Data Type Conversion Functions, Class A

Function Name	Conversion Rule
BOOL_TO_BYTE	Adds leading zeros
BOOL_TO_DWORD	
BOOL_TO_WORD	
BYTE_TO_DWORD	
BYTE_TO_WORD	
CHAR_TO_STRING	Transformation to a string (of length 1) containing the same character.
DINT_TO_REAL	Transformation to `REAL` according to the IEEE standard. The value may change (due to the different resolution of `REAL`).
INT_TO_DINT	The higher-order word of the function value is padded with `16#FFFF` for a negative input parameter, otherwise it is padded with zeros. The value remains the same.
INT_TO_REAL	Transformation to `REAL` according to the IEEE standard. The value remains the same.
WORD_TO_DWORD	Adds leading zeros

List of Conversion Functions (Class B)

Table 18-3 shows the data type conversion functions of Class B. These functions must be specified explicitly. The result can also be undefined if the size of the destination type is insufficient.

You can check for this situation yourself by including a limit check or you can have the system make the check by selecting the ”OK flag” option prior to compilation. In situations where the result is undefined, the system then sets the OK variable to FALSE. Evaluation must be done by yourself.

Table 18-3 Data Type Conversion Functions, Class B

Function name	Conversion Rule	OK
BYTE_TO_BOOL	Copies the least significant bit	Y
BYTE_TO_CHAR	Copies the bit string	N
CHAR_TO_BYTE	Copies the bit string	N
CHAR_TO_INT	The bit string in the input parameter is entered in the lower-order byte of the function value. The higher-order byte is padded with zeros.	N
DATE_TO_DINT	Copies the bit string	N
DINT_TO_DATE	Copies the bit string	Y
DINT_TO_DWORD	Copies the bit string	N
DINT_TO_INT	Copies the bit for the sign. The value in the input parameter is interpreted in the data type `INT`. If the value is less than `–32_768` or greater than `32_767`, the OK variable is set to `FALSE`.	
DINT_TO_TIME	Copies the bit string	N
DINT_TO_TOD	Copies the bit string	Y
DWORD_TO_BOOL	Copies the least significant bit	Y
DWORD_TO_BYTE	Copies the 8 least significant bits	Y
DWORD_TO_DINT	Copies the bit string	N
DWORD_TO_REAL	Copies the bit string	N
DWORD_TO_WORD	Copies the 16 least significant bits	Y
INT_TO_CHAR	Copies the bit string	Y
INT_TO_WORD	Copies the bit string	N
REAL_TO_DINT	Rounds the IEEE `REAL` value to `DINT`. If the value is less than `–2_147_483_648` or greater than `2_147_483_647`, the `OK` variable is set to `FALSE`.	Y
REAL_TO_DWORD	Copies the bit string	N
REAL_TO_INT	Rounds the IEEE `REAL` value to `INT`. If the value is less than `–32_768` or greater than `32_767`, the `OK` variable is set to `FALSE`.	Y

Table 18-3 Data Type Conversion Functions, Class B

Function name	Conversion Rule	OK
STRING_TO_CHAR	Copies the first character of the string. If the STRING does not have a length of 1, the `OK` variable is set to `FALSE`.	Y
TIME_TO_DINT	Copies the bit string	N
TOD_TO_DINT	Copies the bit string	N
WORD_TO_BOOL	Copies the least significant bit	Y
WORD_TO_BYTE	Copies the least significant 8 bits	Y
WORD_TO_INT	Copies the bit string	N
WORD_TO_BLOCK_DB	The bit pattern of WORD is interpreted as the data block number	N
BLOCK_DB_TO_WORD	The data block number is interpreted as the bit pattern of WORD	N

Note

You also have the option of using **IEC functions** for data type conversion. In this case, you should copy the desired function from the STEP 7 library `STDLIBS\IEC` to your program directory. For details of individual IEC functions, refer to **/235/.**

Examples of Explicit Conversions

In Example 18-2 below, an explicit conversion is necessary since the destination data type is of a lower order than the source data type.

```
FUNCTION_BLOCK FB10
VAR
      SWITCH      : INT;
      CONTROLLER  : DINT;
END_VAR

BEGIN
SWITCH := DINT_TO_INT (CONTROLLER);
(* INT is of a lower order than DINT *)
//...
END_FUNCTION_BLOCK
```

Example 18-2 Target Data Type does not Match Source Data Type

In Example 18-3, an explicit data type conversion is necessary, since the data type REAL is not permissible for a mathematical expression with the MOD operator.

```
FUNCTION_BLOCK FB20
  VAR
      intval:INT:=17;
      CONV2 := INT;
  END_VAR

BEGIN
  CONV2 := intval MOD REAL_TO_INT (2.3);
  (* MOD may only be used for data of the types
  INT or DINT. *)
//...
END_FUNCTION_BLOCK
```

Example 18-3 Conversion due to Non-Permissible Data Type

In Example 18-4, conversion is necessary because the data type is incorrect for a logical operator. The NOT operator should only be used for data of the types BOOL, BYTE, WORD or DWORD.

```
FUNCTION_BLOCK FB30
  VAR
      intval:INT:=17;
      CONV1 :=WORD;
  END_VAR

BEGIN
  CONV1 := NOT INT_TO_WORD(intval);
  (* NOT may only be used for data
   of the type INT. *)
//...
END_FUNCTION_BLOCK
```

Example 18-4 Conversion due to Incorrect Data Type

Example 18-5 illustrates data type conversion in the case of peripheral inputs/outputs.

```
FUNCTION_BLOCK FB40
  VAR
      radius_on   : WORD;
      radius      : INT;
  END_VAR

BEGIN
  radius_on := IB0;
  radius    := WORD_TO_INT(radius_on);
(* Conversion due to change to different type
class. Value comes from input and is converted for
subsequent processing. *)
  radius    := Radius(area:= circledata.area);
  QB0       := WORD_TO_BYTE(INT_TO_WORD(radius));

(* Radius is recalculated from the area and is
present in integer format. For output purposes,
the value is first converted to a different type
class (INT_TO_WORD) and then to a lower-order type
(WORD_TO_BYTE). *)
//...
END_FUNCTION_BLOCK
```

Example 18-5 Conversion of Inputs and Outputs

Functions for Rounding and Truncating

The functions for rounding and truncating numbers are also classed as data type conversion functions. Table 18-4 shows the names, data types (for the input parameters and the function value) and purposes of these functions:

Table 18-4 Functions for Rounding and Truncating

Function Name	Data Type of Input Parameter	Data Type of Function Value	Purpose
ROUND	REAL	DINT	Rounds (forms a DINT number)
TRUNC	REAL	DINT	Truncates (forms a DINT number)

The differences in the way the various functions work are illustrated by the following examples:

-
  ```
  ROUND (3.14)     // Rounding down,
                   // Result: 3
  ```

-
  ```
  ROUND (3.56)     // Rounding up,
                   // Result: 4
  ```

-
  ```
  TRUNC (3.14)     // Truncating,
                   // Result: 3
  ```

-
  ```
  TRUNC (3.56)     // Truncating,
                   // Result: 3
  ```

18.3 Numeric Standard Functions

Function

Each numeric standard function has one input parameter. The result is always the function value. Each of the Tables 18-5, 18-6 and 18-7 details a group of numeric standard functions together with their function names and data types. The data type ANY_NUM stands for INT, DINT or REAL.

List of General Functions

General functions are for calculating the absolute amount, the square or the square root of an amount.

Table 18-5 General Functions

Function Name	Data Type of Input Parameter	Data Type of Function Value	Description
ABS	ANY_NUM[1]	ANY_NUM	Number
SQR	ANY_NUM[1]	REAL	Square
SQRT	ANY_NUM[1]	REAL	Square root

1 *Note that input parameters of the type ANY_NUM are converted internally into real variables.*

List of Logarithmic Functions

Logarithmic functions are for calculating an exponential value or the logarithm of a number.

Table 18-6 Logarithmic Functions

Function Name	Data Type of Input Parameter	Data Type of Function Value	Description
EXP	ANY_NUM[1]	REAL	e to the power IN
EXPD	ANY_NUM[1]	REAL	10 to the power IN
LN	ANY_NUM[1]	REAL	Natural logarithm
LOG	ANY_NUM[1]	REAL	Common logarithm

1 *Note that input parameters of the type ANY_NUM are converted internally into real variables.*

Note

You also have the option of using **IEC functions** as numeric standard functions. In that case, you should copy the desired function from the STEP 7 library STDLIBS\IEC to your program directory. For details of the individual IEC functions, refer to **/235/.**

List of Trigonometrical Functions

The trigonometrical functions listed in Table 18-7 expect and calculate angles in radians.

Table 18-7 Trigonometrical Functions

Function Name	Data Type of Input Parameter	Data Type of Function Value	Description
ACOS	ANY_NUM[1]	REAL	Arc cosine
ASIN	ANY_NUM[1]	REAL	Arc sine
ATAN	ANY_NUM[1]	REAL	Arc tangent
COS	ANY_NUM[1]	REAL	Cosine
SIN	ANY_NUM[1]	REAL	Sine
TAN	ANY_NUM[1]	REAL	Tangent

1 *Note that input parameters of the type* `ANY_NUM` *are converted internally into real variables.*

Examples

Table 18-8 shows possible function calls for standard functions and their various results:

Table 18-8 Calling Numeric Standard Functions

Function Call	Result
RESULT := ABS (-5);	5
RESULT := SQRT (81.0);	9
RESULT := SQR (23);	529
RESULT := EXP (4.1);	60.340 ...
RESULT := EXPD (3);	1_000
RESULT := LN (2.718_281);	1
RESULT := LOG (245);	2.389_166 ...

PI := 3. 141 592;
RESULT := SIN (PI / 6);
	0.5

RESULT := ACOS (0.5);	1.047_197 (=PI / 3)

18.4 Bit String Standard Functions

Function

Each bit string standard function has two input parameters identified by `IN` and `N`. The result is always the function value. Table 18-9 lists the function names and data types of the two input parameters in each case as well as the data type of the function value. Explanation of input parameters:

- Input parameter IN: buffer in which bit string operations are performed.
- Input parameter N: number of cycles of the cyclic buffer functions ROL and ROR or the number of places to be shifted in the case of SHL and SHR.

List of Functions

Table 18-9 shows the possible bit string standard functions.

Table 18-9 Bit String Standard Functions

Function Name	Data Type of Input Parameter `IN`	Data Type of Input Parameter `N`	Data Type of Function Value	Purpose
ROL	BOOL	INT	BOOL	The value in the parameter `IN` is rotated left by the number of bit places specified by the content of parameter N.
	BYTE	INT	BYTE	
	WORD	INT	WORD	
	DWORD	INT	DWORD	
ROR	BOOl	INT	BOOL	The value in the parameter `IN` is rotated right by the number of bit places specified by the content of parameter N.
	BYTE	INT	BYTE	
	WORD	INT	WORD	
	DWORD	INT	DWORD	
SHL	BOOL	INT	BOOL	The value in the parameter `IN` is shifted as many places left and as many bit places on the right-hand side replaced by 0 as are specified by the parameter N.
	BYTE	INT	BYTE	
	WORD	INT	WORD	
	DWORD	INT	DWORD	
SHR	BOOL	INT	BOOL	The value in the parameter `IN` is shifted as many places right and as many bit places on the left-hand side replaced by 0 as are specified by the parameter N.
	BYTE	INT	BYTE	
	WORD	INT	WORD	
	DWORD	INT	DWORD	

Note

You also have the option of using **IEC functions** for bit string operations. In that case you should copy the desired function from the STEP 7 library STDLIBS\IEC to your program directory. For details of individual IEC functions, refer to **/235/.**

Examples

Table 18-10 shows possible function calls for bit string standard functions and the results in each case.

Table 18-10 Calling Bit String Standard Functions

Function Call	**RESULT**

RESULT := ROL
(IN:=2#1101_0011, N:=5);
// IN := 211 decimal
	2#0111_1010
	(= 122 decimal)

RESULT := ROR
(IN:=2#1101_0011, N:=2);
// IN := 211 decimal
	2#1111_0100
	(= 244 decimal)

RESULT := SHL
(IN:=2#1101_0011, N:=3);
// IN := 211 decimal
	2#1001_1000
	(= 152 decimal)

RESULT := SHR
(IN:=2#1101_0011, N:=2);
// IN := 211 decimal
	2#0011_0100
	(= 52 decimal)

Function Call Interface 19

Introduction

S7 CPUs contain system and standard functions integrated in the operating system which you can make use of when programming in SCL. Specifically, those functions are the following:

- Organization blocks (OBs)
- System functions (SFCs)
- System function blocks (SFBs)

Chapter Overview

19.1 Function Call Interface

Overview

You can call blocks in symbolic or absolute terms. To do so, you require either the symbolic name, which must have been declared in the symbol table, or the number of the absolute identifier of the block.

In the function call, you must assign the **formal parameters**, whose names and data types have been specified when the configurable block was created, **actual parameters** with which the block works when the program is running.

All the information you require is given in **/235/**. This manual provides a general outline of the basic functions in S7 and, as reference information, detailed interface descriptions for use in your programs.

Example of SFC 31

The following command lines enable you to call the system function SFC 31 (query time of day interrupt):

```
FUNCTION_BLOCK FB20
  VAR
      Result:INT;
  END_VAR

BEGIN
  //...
  Result:= SFC 31 (OB_NR:= 10,STATUS:= MW100 );
  //...
  //...
END_FUNCTION_BLOCK
```

Example 19-1 Querying the Time-Of-Day Interrupt

Results

The data type of the function value is Integer. If its value is > = 0 this indicates that the block has been processed without errors. If the value is < 0, an error has occurred. After calling the function, you can check the implicitly defined output parameter ENO.

Conditional Function Call

For a conditional function call, you must set the predefined **input parameter EN** to 0 (foe example, via input I0.3). The block is then not called. If EN is set to 1, the function is called. The **output parameter ENO** is also set to ”1” in this case (otherwise ”0”) if no error occurs during processing of the block.

Note

In the case of function blocks or system function blocks, the information that can be passed over by means of the function value in the case of a function must be stored in output parameters. These are then subsequently read via the instance data block. For more detailed information, refer to Chapter 16.

19.2 Data Transfer Interface with OBs

Organization Blocks

Organization blocks form the interface between the CPU operating system and the application program. OBs can be used to execute specific program sections in the following situations:

- when the CPU is powered up
- as cyclic or timed operations
- at specific times or on specific days
- on expiry of a specified time period
- if errors occur
- if process or communications interrupts are triggered

Organization blocks are processed according to the priority they are assigned.

Available OBs

Not all CPUs can process all OBs provided by S7. Refer to the data sheets for your CPU to find out which OBs you can use.

Additional Information

Additional information can be obtained from the on-line help and the following manuals:

- **/70/** Manual: *S7-300 Programmable Controller, Hardware and Installation*
  This manual contains the data sheets which describe the performance specifications of the various S7-300 CPUs. This also includes the possible start events for each OB.
- **/100/** Manual: *S7-400/M7-400 Programmable Controllers,* Hardware and Installation
  This manual contains the data sheets which describe the performance specifications of the various S7-400 CPUs. This also includes the possible start events for each OB.

Appendix

Formal Description of Language A

Introduction

The basic tool for the description of the language in the various chapters of this manual is the syntax diagram. It provides a clear insight into the syntax (that is, grammatical structure) of SCL. The complete set of syntax diagrams and language elements is presented in Appendices B and C.

Chapter Overview

A.1 Overview

What is a Syntax Diagram?

The syntax diagram is a graphical representation of the structure of the language. That structure is defined by a series of rules. One rule may be based on others at a more fundamental level.

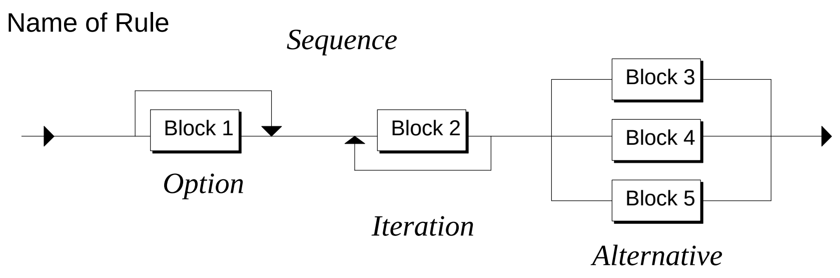


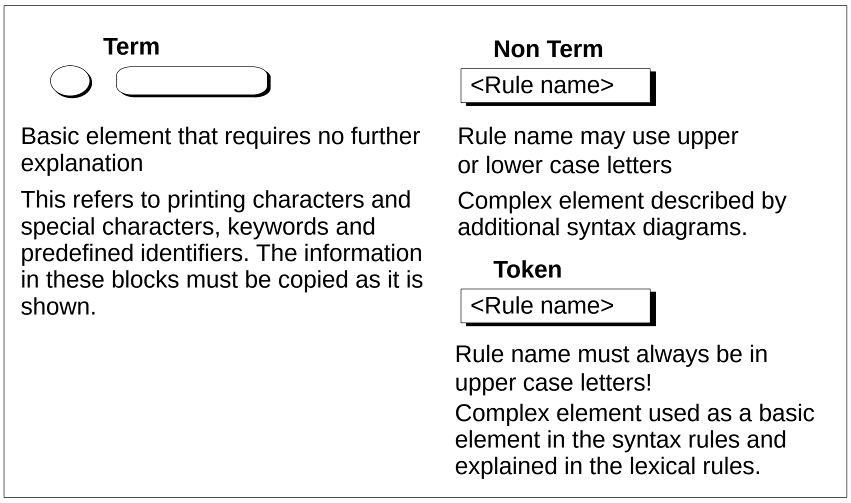
Figure A-1 Example of a Syntax Diagram

The syntax diagram is read from left to right and should conform to the following rule structures:

- Sequence: a sequence of blocks
- Option: a skippable branch
- Iteration: repetition of branches
- Alternative: a split into multiple branches

What Types of Blocks Are There?

A block is a basic element or an element made up of other blocks. The diagram below shows the symbols that represent the various types of block.



Basic element that requires no further explanation

This refers to printing characters and special characters, keywords and predefined identifiers. The information in these blocks must be copied as it is shown.

Rule name may use upper or lower case letters

Complex element described by additional syntax diagrams.

Rule name must always be in upper case letters!

Complex element used as a basic element in the syntax rules and explained in the lexical rules.

Figure A-2 Types of Symbols for Blocks

Rules

The rules which you apply to the structure of your SCL program are subdivided into the categories **lexical** and **syntax** rules.

Lexical Rules

The lexical rules describe the structure of the elements (tokens) processed during the lexical analysis performed by the Compiler. For this reason lexical rules are not free-format; that is, they must be strictly observed. In particular, this means that

- insertion of formatting characters is not permitted,
- insertion of remarks blocks and lines is not permitted,
- insertion of attributes for identifiers is not permitted.

IDENTIFIER

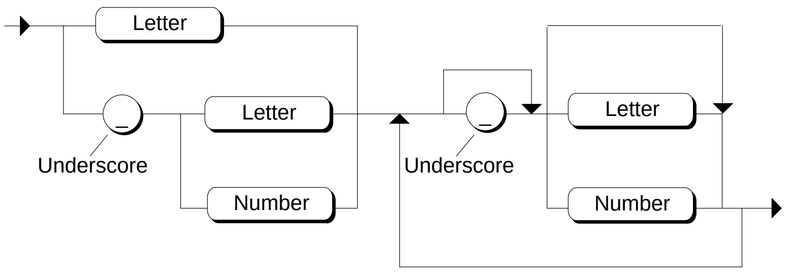


Figure A-3 Example of a Lexical Rule

The above example shows the lexical rule for IDENTIFIER. It defines the structure of an identifier (name), for example:

```
MEAS_ARRAY_12
SETPOINT_B_1
```

Syntax Rules

The syntax rules are built up from the lexical rules and define the structure of SCL. Within the limitations of those rules the structure of the your SCL program is free-format.

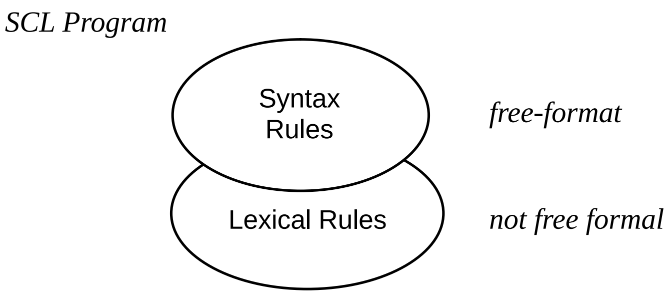


Figure A-4 Rule Categories and Format Restrictions

Formal Considerations

Each rule has a name which precedes the definition. If that rule is used in a higher-level rule, that name appears in the higher-level rule as a non term. If the rule name is written in upper case, it is a token that is described in the lexical rules.

Semantics

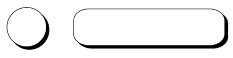
The rules can only represent the formal structure of the language. The meaning; that is, the semantics, is not always obvious from the rules. For this reason, where it is important, additional information is written next to the rule. The following are examples of such situations:

- Where there are elements of the same type with different meanings, an additional name is specified, for example, in the Date Specification rule the explanatory names Year, Month or Day are added to the element DECIMAL_DIGIT_STRING.
- Where there are important limitations, these are noted alongside the rule, for example, in the case of Symbol, the fact that it has to be defined in the symbol editor.

A.2 Overview of Terms

Definition

A term is a basic element that can not be explained by another rule but is represented verbally. In a syntax diagram, it is represented by the following symbol:



A term is represented by an oblong with rounded corners or a circle. The item is shown in literal terms or as a name (in upper case letters).

This defines the range of ASCII characters that can be used.

Figure A-5 Symbols for Terms

Summary

In Sections A.3 to A.4 the types of use for different characters are explained. The various types of character are as follows:

- letters, numbers, printing characters and special characters,
- formatting characters and delimiters in the lexical rules,
- prefixes for literals
- formatting characters and delimiters in the syntax rules
- operators

Sections A.5 and A.6 deal with keywords and predefined identifiers made up of character strings. The tables are arranged in alphabetical order. In the event of differences between SIMATIC and IEC mnemonics, the corresponding IEC mnemonic is shown as well.

- Keywords and predefined identifiers
- Address identifiers and block keywords

A.3 Lexical Rule Terms

Summary

The tables below define the terms on the basis of a range of characters from the ASCII character set.

Letters and Numbers

Letters and numbers are the characters most commonly used. An IDENTIFIER (see Section A.1), for example, can be made up of a combination of letters, numbers and the underscore character.

Table A-1 Letters and Numbers

Character	Subgroup	Character Set Range
Letter	Upper case letters	A.. Z
	Lower case letters	a.. z
Number	Decimal numbers	0.. 9
Octal number	Octal numbers	0.. 7
Hexadecimal number	Hexadecimal numbers	0.. 9, A.. F, a.. f
Bit	Binary numbers	0, 1

Printing Characters and Special Characters

The complete extended ASCII character set can be used in strings, comments and symbols.

Table A-2 Printing Characters and Special Characters

Character	Subgroup	Character Set Range
Printing character	Depends on the chracter code used. In the case of ASCII code, for example, upwards of decimal equivalent 31 excluding DEL and the following substitute characters:	All printing characters
Substitute characters	Dollar sign	$
	Apostrophe	’
Control characters	$P or $p	Page break (form feed, page feed)
	$L or $l	Line break (line feed)
	$R or $r	Carriage return
	$T or $t	Tabulator
Substitute representation in hexadecimal code	$hh	Any characters capable of representation in hexadecimal code (hh)

A.4 Formatting Characters, Delimiters and Operators

In Lexical Rules

Table A-3 below defines the use of individual characters in the ASCII character set as formatting characters and delimiters within lexical rules (see Appendix B).

Table A-3 Formatting Characters and Delimiters in Lexical Rules

Character	Description
:	Delimiter between hours, minutes and seconds Attribute
.	Delimiter for absolute addresses in real number or time period representation
’ ’	Characters and character strings
” ”	Introductory character for symbols according to symbol editor rules
_ Underscore	Delimiter for numbers in literals and can be used in IDENTIFIERS
$	Alignment symbol for specifying control characters or substitute characters
$> $<	String break, in case the string does not fit in one row, or if the comments are to be inserted.

For Literals

Table A-4 defines the use of individual characters and character strings in lexical rules. The table applies to SIMATIC and IEC versions.

Table A-4 Mnemonics for Literals in Alphabetical Order

Prefix	Represents	Lexical Rule
2#	INTEGER LITERAL	Binary digit string
8#	INTEGER LITERAL	Octal digit string
16#	INTEGER LITERAL	Hexadecimaldigit string
D#	Time specification	DATE
DATE#	Time specification	DATE
DATE_AND_TIME#	Time specification	DATE AND TIME
DT#	Time specification	DATE AND TIME
E	Delimiter for REAL NUMBER LITERAL	Exponent
e	Delimiter for REAL NUMBER LITERAL	Exponent
D	Delimiter for time unit (day)	Days (rule: complex format)
H	Delimiter for time unit (hours)	Hours: (rule: complex format)
M	Delimiter for time unit (minutes)	Minutes : (rule: complex format)
MS	Delimiter for time unit (milliseconds)	Milliseconds: (rule: complex format)
S	Delimiter for time unit (seconds)	Seconds: (rule: complex format)
T#	Time specification	TIME PERIOD
TIME#	Time specification	TIME PERIOD
TIME_OF_DAY#	Time specification	TIME OF DAY
TOD#	Time specification	TIME OF DAY

In Syntax Rules

The table below defines the use of individual characters as formatting characters and delimiters in the syntax rules and remarks and attributes (see Appendices B.2 and B.3).

Table A-5 Formatting Characters and Delimiters in Syntax Rules

Character	Description	Syntax Rule, Remarks or Attribute
:	Delimiter for type specification in statement after jump label	Variable declaration, instance declaration, function code section, CASE statement
;	Terminates a declaration or statement	Constant and variable declarations, code section, DB assignment section, constant subsection, jump label subsection, component declaration
,	Delimiter for lists and jump label subsection	Variable declaration, array data type specification, array initialization list, FB parameters, FC parameters, value list, instance declaration
..	Range specification	Array data type specification, value list
.	Delimiter for FB and DB name, absolute address	FB call, structure variables
( )	Function and function block calls bracketed in expressions Initialization list for arrays	Function call, FB call, expression, array initialization list, simple multiplication, exponential expression
[ ]	Array declaration, array structured variable section, indexing of global variables and strings	Array data type specification, STRING data type specification
(* *)	Block comment	see Appendix B
//	Line comment	see Appendix B
{ }	Attribute field	For specifying attributes
%	Introduction for direct descriptor	In order to program in agreement with IEC, you can use %M4.0 instead of M4.0.

Operators

Table A-6 details all SCL operators, keywords, for example, AND, and the usual operators as individual characters. The table applies for both SIMATIC and IEC mnemonics.

Table A-6 SCL Operators

Operator	Description	Example, Syntax Rule
:=	Assignment operator, initial assignment, data type initialization	Value assignment, DB assignment section, constant subsection, output and in/out assignments, input assignment
+, -	Mathematical operators: unary operators, plus and minus signs	Expression, simple expression, exponential expression
+, -, *, / MOD; DIV	Basic mathematical operators	Basic mathematical operator, simple multiplication
**	Mathematical operators, exponent operator	Expression
NOT	Logical operators; negation	Expression
AND, &, OR; XOR,	Basic logical operators	Basic logical operator, expression
<,>,<=,>=,=,<>	Comparator	Comparator

A.5 Keywords and Predefined Identifiers

Keywords and Predefined Identifiers

Table A-7 lists SCL keywords and predefined identifiers in alphabetical order. Alongside each one is a description and the syntax rule as per Appendix C in which they are used as a term. Keywords are generally independent of the mnemonics.

Table A-7 SCL Keywords and Predefined Identifiers in Alphabetical Order

Keyword	Description	Syntax Rule
AND	Logical operator	Basic logical operator
ANY	Identifier for data type ANY	Parameter data type specification
ARRAY	Introduces the specification of an array and is followed by the index list enclosed in ”[” and ”]”.	Array data type specification
BEGIN	Introduces code section in logic blocks or initialization section in data blocks	Organization block, function, function block, data block
BLOCK_DB	Identifier for data type BLOCK_DB	Parameter data type specification
BLOCK_FB	Identifier for data type BLOCK_FB	Parameter data type specification
BLOCK_FC	Identifier for data type BLOCK_FC	Parameter data type specification
BLOCK_SDB	Identifier for data type BLOCK_SDB	Parameter data type specification
BOOL	Elementary data type for binary data	Bit data type
BY	Introduces increment specification	FOR statement
BYTE	Elementary data type	Bit data type
CASE	Introduces control statement for selection	CASE statement
CHAR	Elementary data type	Character type
CONST	Introduces definition of constants	constant subsection
CONTINUE	Control statement for FOR, WHILE and REPEAT loops	CONTINUE statement
COUNTER	Data type for counters, useable in parameter subsection only	Parameter data type specification
DATA_BLOCK	Introduces a data block	Data block
DATE	Elementary data type for dates	Time type
DATE_AND_TIME	Composite data type for date and time	see Table C-4
DINT	Elementary data type for whole numbers (integers), double resolution	Numeric data type
DIV	Operator for division	Basic mathematical operator, simple multiplication
DO	Introduces code section for FOR statement	FOR statement, WHILE statement
DT	Elementary data type for date and time	see Table C-4
DWORD	Elementary data type for double word	Bit data type
ELSE	Introduces instructions to be executed if condition is not satisfied	IF statement
ELSIF	Introduces alternative condition	IF statement
EN	Block clearance flag	

Table A-7 SCL Keywords and Predefined Identifiers in Alphabetical Order, continued

Keyword	Description	Syntax Rule
ENO	Block error flag	
END_CASE	Terminates CASE statement	CASE statement
END_CONST	Terminates definition of constants	constant subsection
END_DATA_BLOCK	Terminates data block	Data block
END_FOR	Terminates FOR statement	FOR statement
END_FUNCTION	Terminates function	Function
END_FUNCTION_BLOCK	Terminates function block	Function block
END_IF	Terminates IF statement	IF statement
END_LABEL	Terminates declaration of a jump label subsection	Jump label subsection
END_TYPE	Terminates UDT	User-defined data type
END_ORGANIZATION_BLOCK	Terminates organization block	Organization block
END_REPEAT	Terminates REPEAT statement	REPEAT statement
END_STRUCT	Terminates specification of a structure	Structure data type specification
END_VAR	Terminates declaration block	Temporary variables subsection, static variables ssubsection, parameter subsection
END_WHILE	Terminates WHILE statement	WHILE statement
EXIT	Executes immediate exit from loop	EXIT
FALSE	Predefined Boolean constant; logical condition not satisfied, value equals 0	
FOR	Introduces control statement for loop processing	FOR statement
FUNCTION	Introduces function	Function
FUNCTION_BLOCK	Introduces function block	Function block
GOTO	Instruction for executing a jump to a jump label	Program jump
IF	Introduces control statement for selection	IF statement
INT	Elementary data type for whole numbers (integers), single resolution	Numeric data type
LABEL	Introduces declaration of a jump label subsection	Jump label block
MOD	Mathematical operator for division remainder (modulus)	Basic mathematical operator, simple multiplication
NIL	Zero pointer	
NOT	Logical operator, one of the unary operators	Expression, address
OF	Introduces data type specification	Array data type specification, CASE statement
OK	Flag that indicates whether the instructions in a block have been processed without errors	
OR	Logical operator	Basic logical operator
ORGANIZATION_BLOCK	Introduces an organization block	Organization block

Table A-7 SCL Keywords and Predefined Identifiers in Alphabetical Order, continued

Keyword	Description	Syntax Rule
POINTER	Pointer data type, only allowed in parameter declarations in parameter subsection, not processed in SCL	See Chapter 10
REAL	Elementary data type	Numeric data type
REPEAT	Introduces control statement for loop processing	REPEAT statement
RETURN	Control statement which executes return from subroutine	RETURN statement
S5TIME	Elementary data type for time specification, special S5 format	Time type
STRING	Data type for character string	STRING data type specification
STRUCT	Introduces specification of a structure and is followed by a list of components	Structure data type specification
THEN	Introduces resulting actions if condition is satisfied	IF statement
TIME	Elementary data type for time specification	Time type
TIMER	Data type of timer, useable only in parameter subsection	Parameter data type specification
TIME_OF_DAY	Elementary data type for time of day	Time type
TO	Introduces the terminal value	FOR statement
TOD	Elementary data type for time of day	Time type
TRUE	Predefined Boolean constant; logical condition satisfied, value not equal to 0	
TYPE	Introduces UDT	User-defined data type
UNTIL	Introduces break condition for REPEAT statement	REPEAT statement
VAR	Introduces declaration subsection	Static variables subsection
VAR_INPUT	Introduces declaration subsection	Parameter subsection
VAR_IN_OUT	Introduces declaration subsection	Parameter subsection
VAR_OUTPUT	Introduces declaration subsection	Parameter subsection
VAR_TEMP	Introduces declaration subsection	Temporary variables subsection
WHILE	Introduces control statement for loop processing	WHILE statement
WORD	Elementary data type Word	Bit data type
VOID	No return value from a function call	See Chapter 8
XOR	Logical operator	Logical operator

A.6 Address Identifiers and Block Keywords

Global System Data

Table A-8 details the SIMATIC mnemonics of SCL address identifiers arranged in alphabetical order along with a description of each.

- Address identifier specification:
  Memory prefix (Q, I, M, PQ, PI) or data block (D)
- Data element size specification:
  Size prefix (optional or B, D, W, X)

The mnemonics represent a combination of the address identifier (memory prefix or D for data block) and the size prefix. Both are lexical rules. The table is arranged in order of SIMATIC mnemonics and the corresponding IEC mnemonics specified in the second column.

Table A-8 Address Identifiers for Global System Data

SIMATIC Mnemonics	IEC Mnemonics	Memory Prefix or Data Block	Size Prefix
A	Q	Output (via process image)	Bit
AB	QB	Output (via process image)	Byte
AD	QD	Output (via process image)	Double word
AW	QW	Output (via process image)	Word
AX	QX	Output (via process image)	Bit
D	D	Data block	Bit
DB	DB	Data block	Byte
DD	DD	Data block	Double word
DW	DW	Data block	Word
DX	DX	Data block	Bit
E	I	Input (via process image)	Bit
EB	IB	Input (via process image)	Byte
ED	ID	Input (via process image)	Double word
EW	IW	Input (via process image)	Word
EX	IX	Input (via process image)	Bit
M	M	Bit memory	Bit
MB	MB	Bit memory	Byte
MD	MD	Bit memory	Double word
MW	MW	Bit memory	Word
MX	MX	Bit memory	Bit
PAB	PQB	Output (Direct to peripherals)	Byte
PAD	PQD	Output (Direct to peripherals)	Double word
PAW	PQW	Output (Direct to peripherals)	Word
PEB	PIB	Input (Direct from peripherals)	Byte
PED	PID	Input (Direct from peripherals)	Double word
PEW	PIW	Input (Direct from peripherals)	Word

Block Keywords

Used for absolute addressing of blocks. The table is arranged in order of SIMATIC mnemonics and the corresponding IEC mnemonics given in the second column.

Table A-9 Block Keywords Plus Counters and Timers

SIMATIC Mnemonics	IEC Mnemonics	Memory Prefix or Data Block
DB	DB	Data block
FB	FB	Function block
FC	FC	Function
OB	OB	Organization block
SDB	SDB	System data block
SFC	SFC	System function
SFB	SFB	System function block
T	T	Timer
UDT	UDT	User-defined data type
Z	C	Counter

A.7 Overview of Non Terms

Definition

A non term is a complex element that is described by another rule. A non term is represented by an oblong box. The name in the box is the name of the more specific rule.

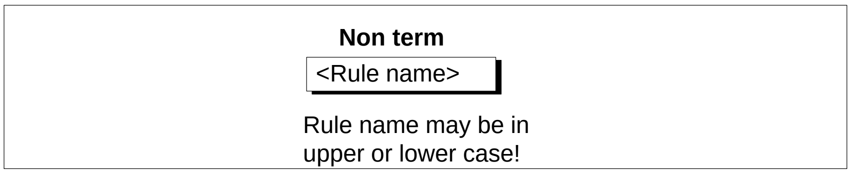


Figure A-6 Non Term

This element occurs in lexical and syntax rules.

A.8 Overview of Tokens

Definition

A token is a complex element used as a basic element in syntax rules and explained in the lexical rules. A token is represented by an oblong box. The NAME, written in upper case letters, is the name of the explanatory lexical rule (not shown inside a box).

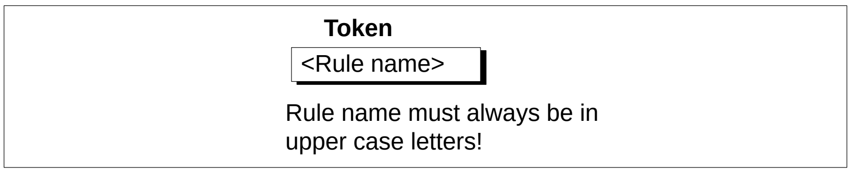


Figure A-7 Token

Summary

The defined tokens represent identifiers calculated as the result of lexical rules. Such tokens describe:

- Identifiers
- SCL names
- Predefined constants and flags

A.9 Identifiers

Identifiers in SCL

Identifiers are used to address SCL language objects. Table A-10 below details the classes of identifier.

Table A-10 Types of Identifier in SCL

Identifier Type	Comments, Examples
Keywords	For example, control statements `BEGIN, DO,WHILE`
Predefined names	Names of • standard data types (for example, BOOL, BYTE, INT) • PREDEFINED STANDARD FUNCTIONS E.G `ABS` • `STANDARD CONSTANTS TRUE and FALSE`
Absolute address identifiers	For global system data and data blocks: for example, `I1.2, MW10, FC20, T5, DB30, DB10.D4.5`
User-defined names based on the rule IDENTIFIER	Names of • declared variables • structure components • parameters • declared constants • jump labels
Symbol editor symbols	Conform either to the lexical rule IDENTIFIER or the lexical rule Symbol, that is, enclosed in inverted commas, for example, ”xyz”

Use of Upper and Lower Case

In the case of the keywords, use of upper and lower case is of no consequence. From SCL version 4.0 and higher, predefined names and user-defined names, for example, for variables, and symbols defined in the symbol table are no longer case-sensitive. Table A-11 summarises the requirements.

Table A-11 Significance of Use of Upper and Lower Case for Identifiers

Identifier Type	Case-Sensitive?
Keywords	No
Predefined names for standard data types	No
Names of standard functions	No
Predefined names for standard constants	No
Absolute address identifiers	No
User-defined names	No
Symbols in the symbol tyble	No

The names of standard functions, for example, BYTE_TO_WORD and ABS can also be written in lower case. The same applies to the parameters for timer and counter functions, for example, SE, se or CU, cu.

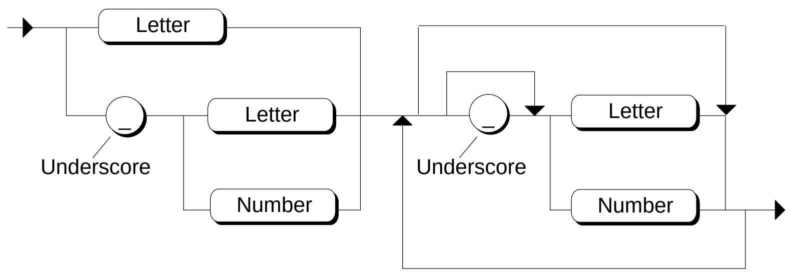
A.10 Naming Conventions in SCL

User-Defined Names

There are generally two options when creating user-defined names:

- You can assign names within SCL itself. Such names must conform to the rule IDENTIFIER (see Figure A-8). IDENTIFIER is the general term you can use for any name in SCL.
- Alternatively, you can assign the name via STEP 7 using the symbol table. The rule to be applied in this case is also IDENTIFIER or, as an additional option, Symbol. By putting your entry in inverted commas, you can write the symbol with all printable characters (for example, spaces).

IDENTIFIER



SYMBOL

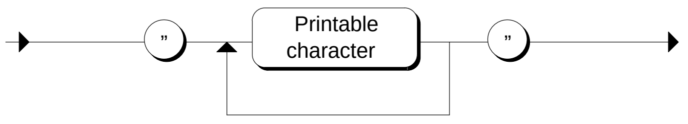


Figure A-8 Lexical Rules IDENTIFIER and Symbol

Naming Conventions

Please observe the following rules:

- Choose names that are unambiguous and self-explanatory and which enhance the comprehensibility of the program.
- Check that the name is not already in use by the system, for example as an identifier for a data type or standard function.
- Limits of applicability: names that apply globally are valid throughout the whole program, locally valid names on the other hand apply only within a specific block. This enables you to use the same names in different blocks. Table A-12 details the various options available.

Naming Restrictions

When assigning names, you must observe the following restrictions:

A name must be unique within the limits of its own applicability, that is, names already used within a particular block can not be used again within the same block. In addition, the following names reserved by the system may not be used:

- Names of keywords: for example, `CONST, END_CONST, BEGIN`
- Names of operators: for example, `AND, XOR`
- Names of predefined identifiers: e.g. names of data types such as `BOOL, STRING, INT`
- Names of the predefined constants `TRUE` and `FALSE`
- Names of standard functions: for example, `ABS, ACOS, ASIN, COS, LN`
- Names of absolute address identifiers for global system data: for example, `IB, IW, ID, QB, QW, QD MB, MD`

Use of IDENTIFIERS

Table A-12 shows in which situations you can use names that conform to the rule for IDENTIFIERS.

Table A-12 Occurrences of IDENTIFIER

IDENTIFIER	Description	Rule
Block name	Symbolic name for block	BLOCK IDENTIFIER, Function call
Name of timer or counter	Symbolic name for timer or counter	TIMER IDENTIFIER, COUNTER IDENTIFIER
Attribute name	Name of an attribute	Attribute assignment
Constant name	Declaration/use of symbolic constant	constant subsection Constant
Jump label	Declaration of jump label, use of jump label	Jump labels subsection code section GOTO statement
Variable name	Declaration of temporary or static variable	Variable declaration, simple variable, Structured variable
Local instance name	Declaration of local instance	Instance declaration, FB call name

BLOCK IDENTIFIERS

The rule BLOCK IDENTIFIER is a case in which you have the choice of using either an IDENTIFIER or a symbol.

BLOCK IDENTIFIER

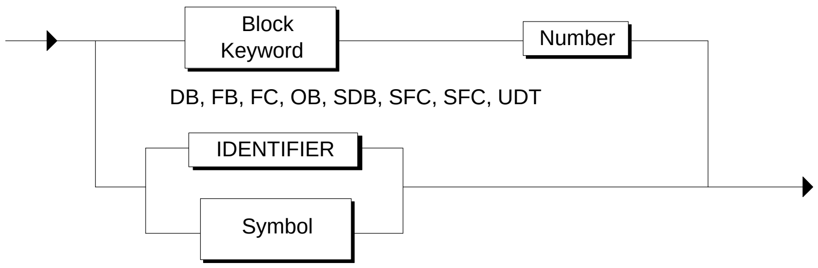


Figure A-9 Lexical Rule BLOCK IDENTIFIER

The same applies to the rules TIMER IDENTIFIER and COUNTER IDENTIFIER as with BLOCK IDENTIFIER.

A.11 Predefined Constants and Flags

Predefined Constants and Flags

The table applies for both SIMATIC and IEC mnemonics.

Table A-13 Predefined Constants

Mnemonic	Description
FALSE	Predefined Boolean constant (standard constant) with the value 0. Its logical meaning is that a condition has not been satisfied.
TRUE	Predefined Boolean constant (standard constant) with the value 1. Its logical meaning is that a condition has been satisfied.

Table A-14 Flags

Mnemonic	Description
EN	Block clearance flag
ENO	Block error flag
OK	Flag is set to FALSE if the statement has been incorrectly processed.

Lexical Rules

B

Chapter Overview

Lexical rules

The lexical rules describe the structure of the elements (tokens) processed during lexical analysis performed by the Compiler. For this reason lexical rules are not free-format; in other words, they must be strictly observed. In particular, this means that:

- Insertion of formatting characters is not permitted.
- Insertion of comment blocks and lines is not permitted.
- Insertion of attributes for identifiers is not permitted.

Categories

The lexical rules are subdivided into the following categories:

- Identifiers
- Literals
- Absolute addresses

B.1 Identifiers

Table B-1 Identifiers

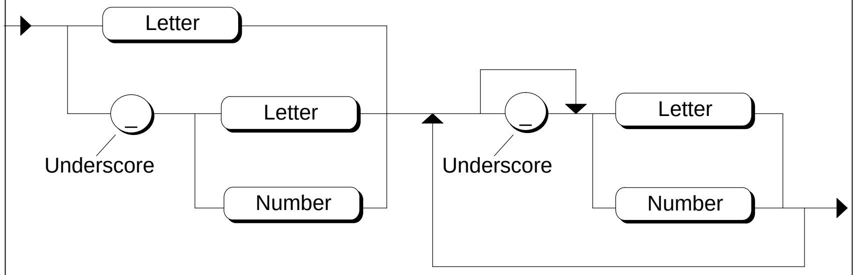
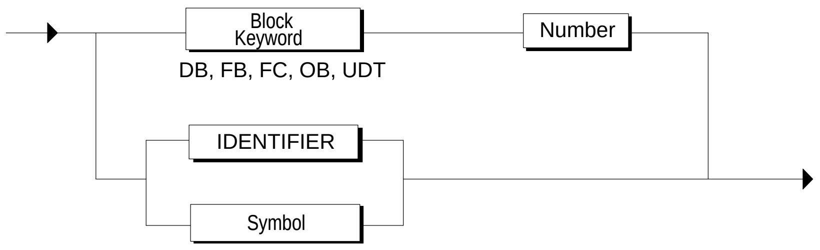
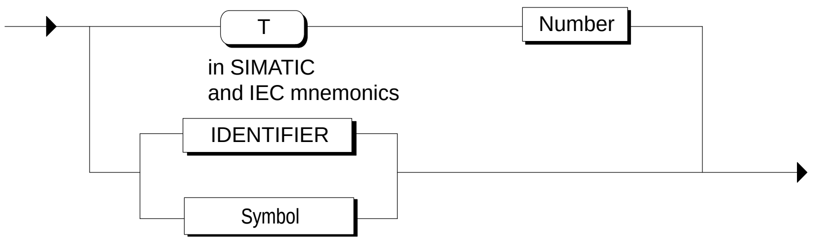
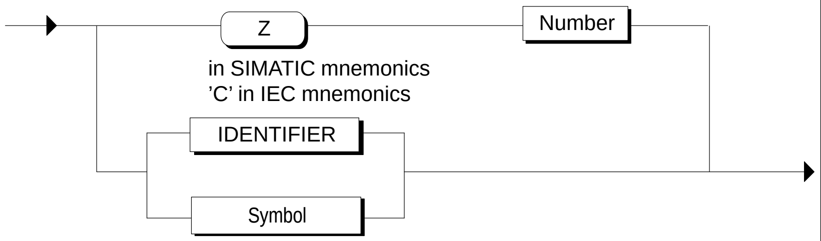
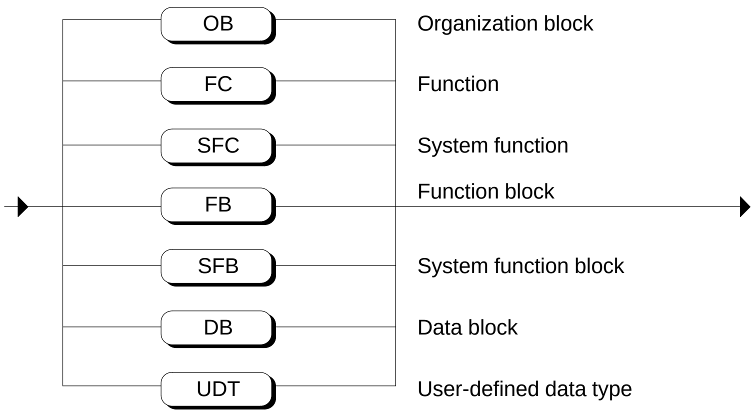
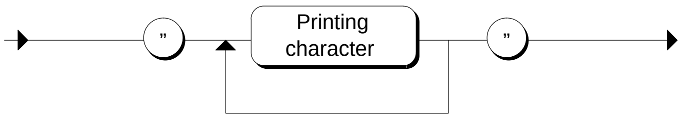
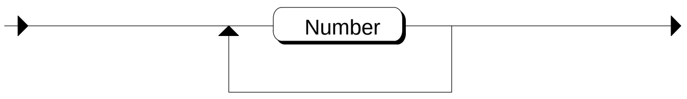
Rule	Syntax Diagram
IDENTIFIER	 Letter _ Underscore Letter Number _ Underscore Letter Number
BLOCK IDENTIFIER	The rule also applies to the following rule names: DB IDENTIFIER FB IDENTIFIER FC IDENTIFIER OB IDENTIFIER UDT IDENTIFIER  Block Keyword DB, FB, FC, OB, UDT Number IDENTIFIER Symbol
TIMER IDENTIFIER	 T in SIMATIC and IEC mnemonics Number IDENTIFIER Symbol

Table B-1 Identifiers, continued

Rule	Syntax Diagram
COUNTER IDENTIFIER	 Z in SIMATIC mnemonics 'C' in IEC mnemonics Number IDENTIFIER Symbol
Block Keyword	 OB Organization block FC Function SFC System function FB Function block SFB System function block DB Data block UDT User-defined data type
Symbol	
Number	

B.1.1 Literals

Table B-2 Literals

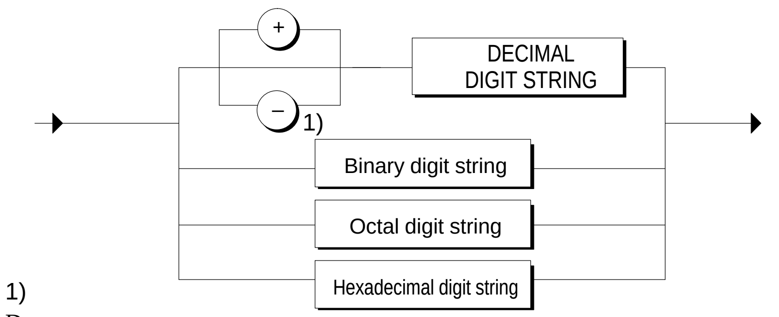
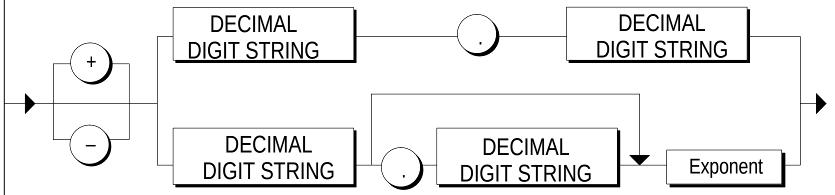
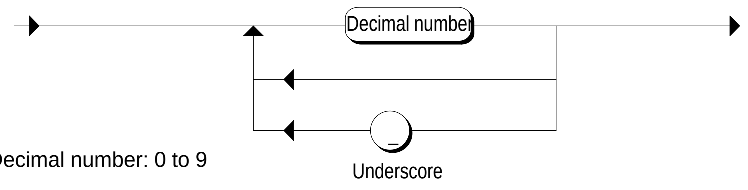
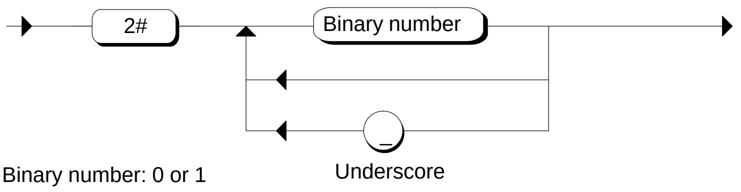
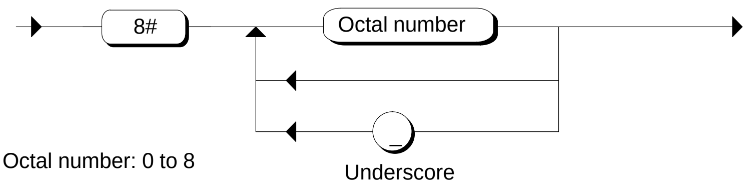
Rule	Syntax Diagram
INTEGER LITERAL	 + − 1) DECIMAL DIGIT STRING Binary digit string Octal digit string Hexadecimal digit string 1) Data types `INT` and `DINT only`
REAL NUMBER LITERAL	 + − DECIMAL DIGIT STRING . DECIMAL DIGIT STRING DECIMAL DIGIT STRING . DECIMAL DIGIT STRING Exponent
DECIMAL DIGIT STRING	 Decimal number: 0 to 9
Binary Digit String	 Binary number: 0 or 1
Octal Digit String	 Octal number: 0 to 8

Table B-2 Literals, continued

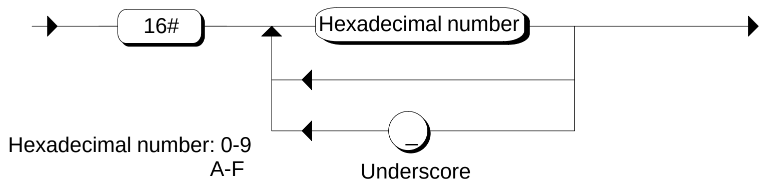
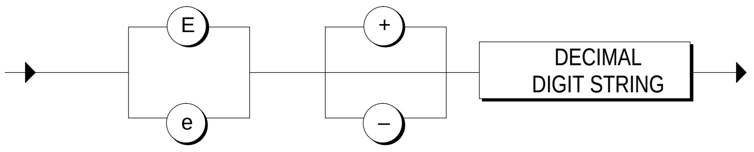
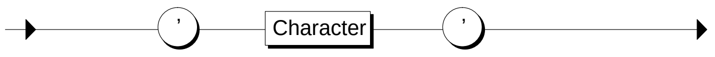
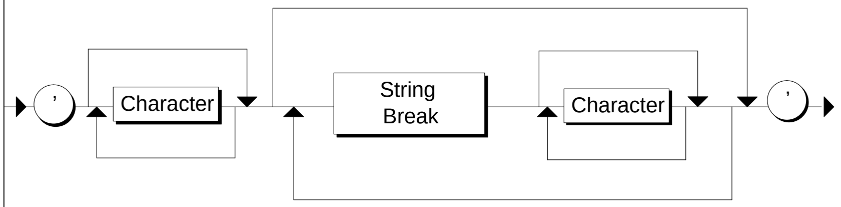
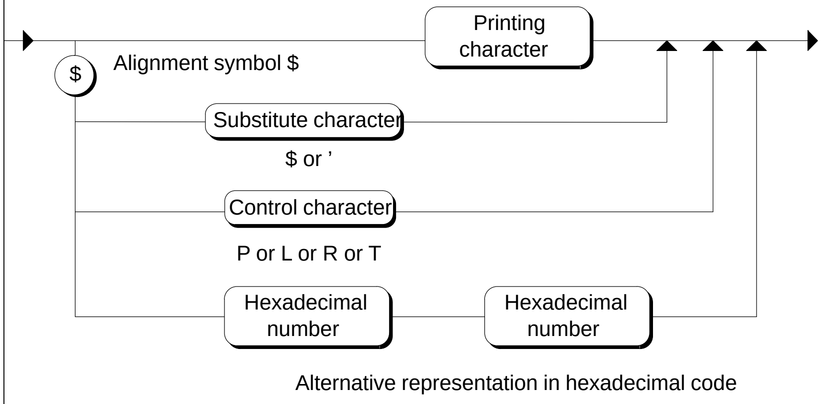
Rule	Syntax Diagram
Hexadecimal Digit String	 16# Hexadecimal number _ Hexadecimal number: 0-9 A-F Underscore
Exponent	
CHARACTER LITERAL	
STRING LITERAL	
Character	 Printing character $ Alignment symbol $ Substitute character $ or ' Control character P or L or R or T Hexadecimal number Hexadecimal number Alternative representation in hexadecimal code

Table B-2 Literals, continued

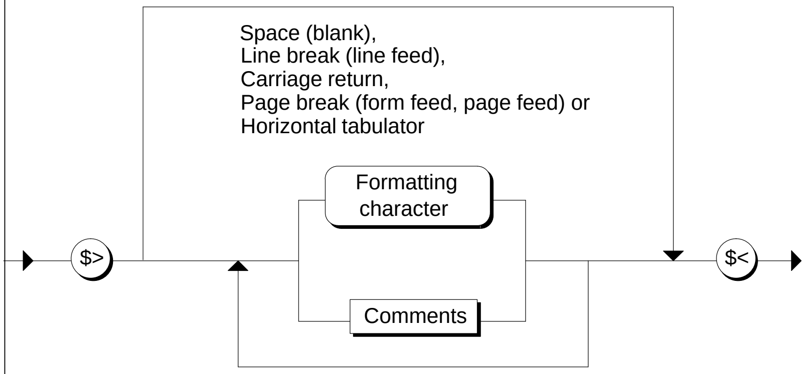
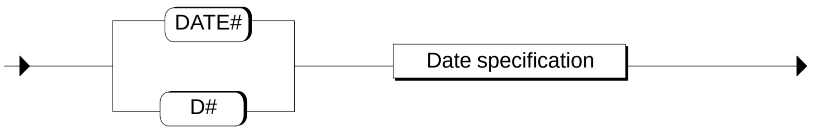
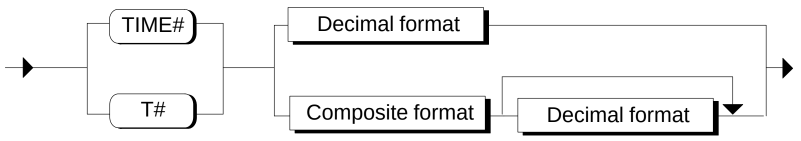
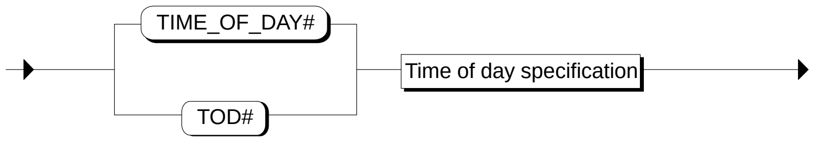
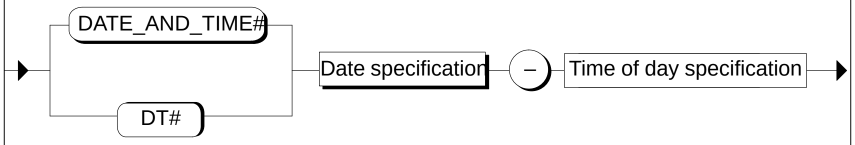
Rule	Syntax Diagram
String Break	 Space (blank), Line break (line feed), Carriage return, Page break (form feed, page feed) or Horizontal tabulator Formatting character Comments $> $<
DATE	
Time Period	 TIME# T# Decimal format Composite format Decimal format Each time unit (for example, hours, minutes) may only be specified once The order days, hours, minutes, seconds, milliseconds must be adhered to.
Time of Day	
Date and Time	 DATE_AND_TIME# DT# Date specification – Time of day specification

Table B-2 Literals, continued

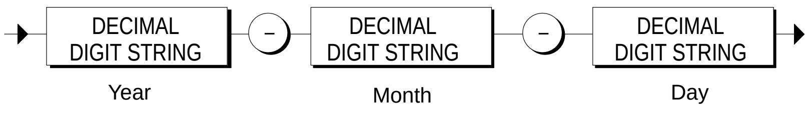
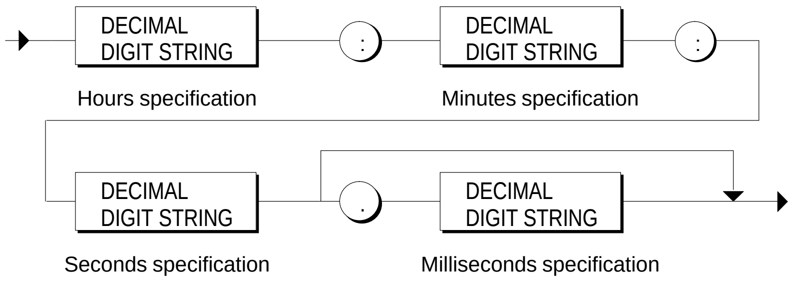
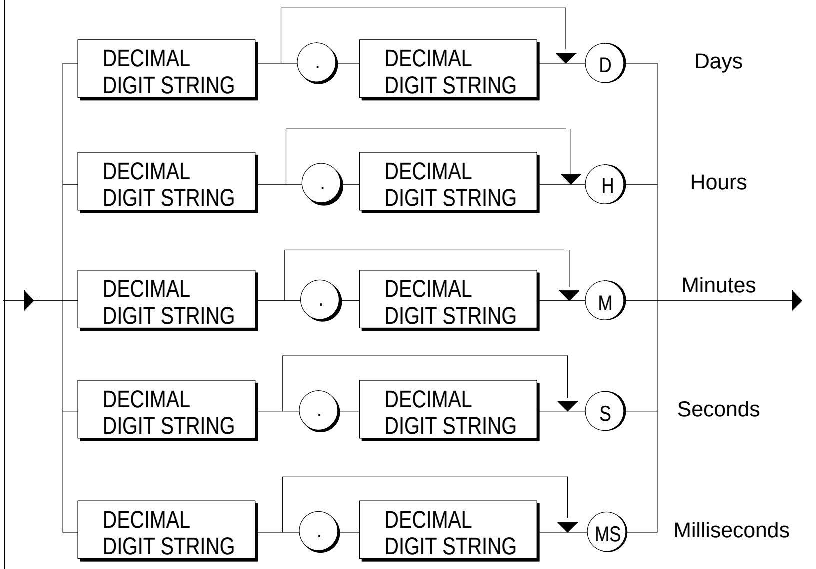
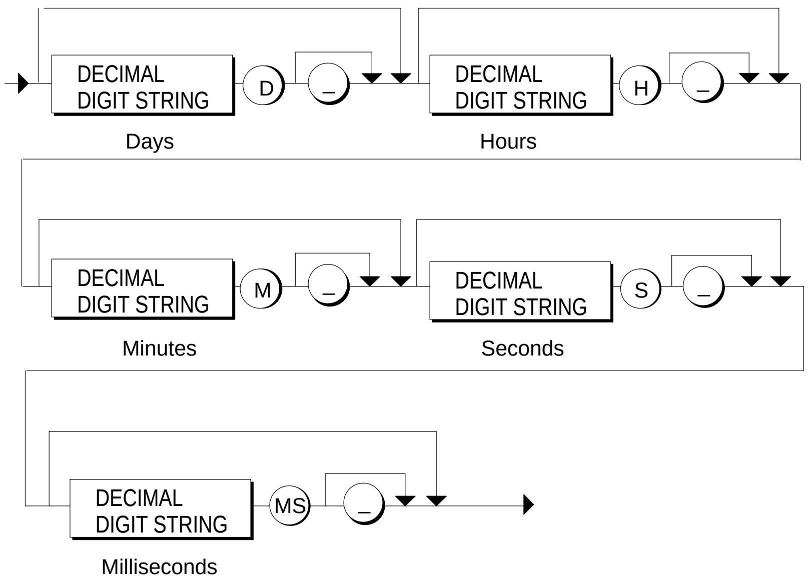
Rule	Syntax Diagram
Date Specification	 DECIMAL DIGIT STRING – DECIMAL DIGIT STRING – DECIMAL DIGIT STRING Year Month Day
Time of Day Specification	 DECIMAL DIGIT STRING : DECIMAL DIGIT STRING : Hours specification Minutes specification DECIMAL DIGIT STRING . DECIMAL DIGIT STRING Seconds specification Milliseconds specification

Table B-2 Literals, continued

Rule	Syntax Diagram
Decimal Format	 DECIMAL DIGIT STRING . DECIMAL DIGIT STRING D Days H Hours M Minutes S Seconds MS Milliseconds Use of decimal format is only possible in the case of previously undefined time units.
Complex Format	 DECIMAL DIGIT STRING D _ Days H Hours M Minutes S Seconds MS Milliseconds A value for at least one time unit must be specified!

B.1.2 Absolute Addresses

Table B-3 Absolute Addresses

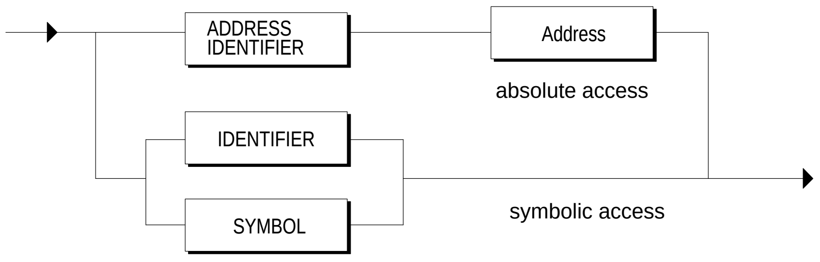
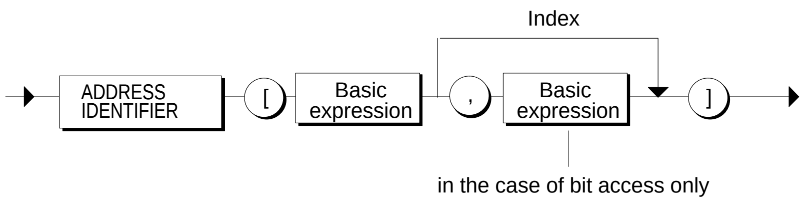
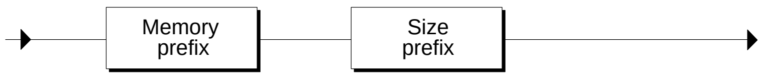
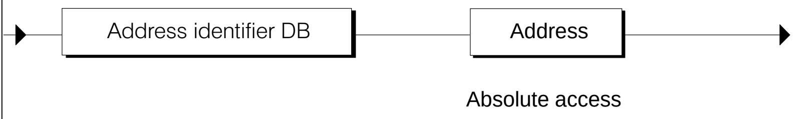
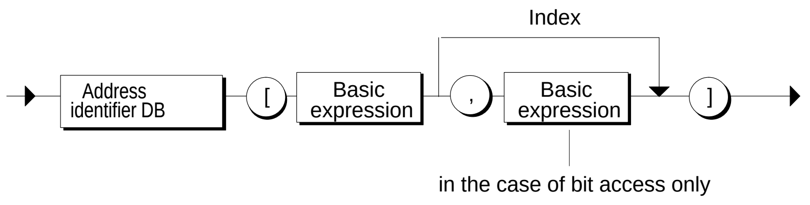
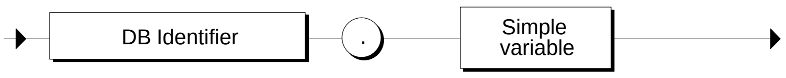
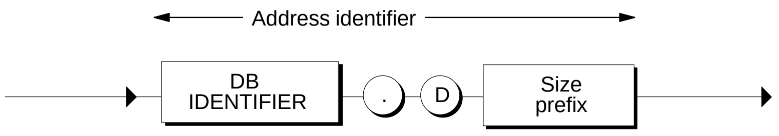
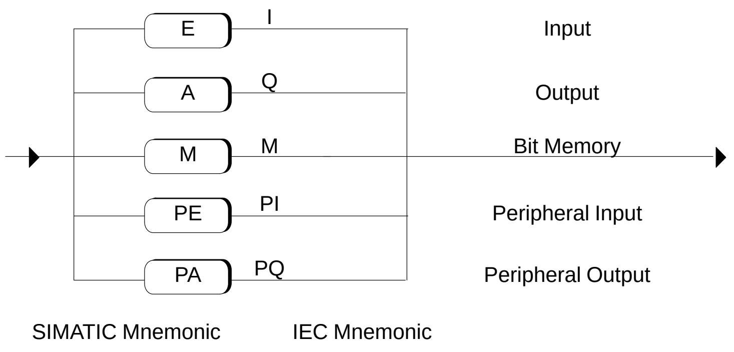
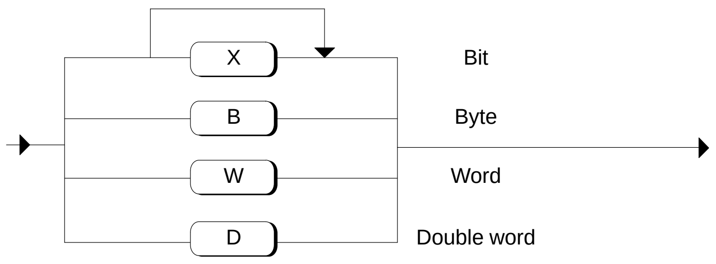
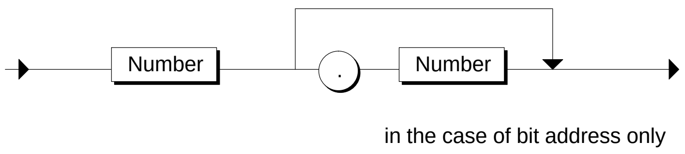
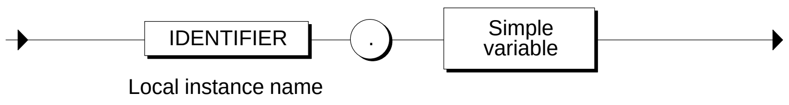
Rule	Syntax Diagram
SIMPLE MEMORY ACCESS	 ADDRESS IDENTIFIER Address absolute access IDENTIFIER SYMBOL symbolic access
INDEXED MEMORY ACCESS	 Index ADDRESS IDENTIFIER [Basic expression , Basic expression] in the case of bit access only
ADDRESS IDENTIFIER FOR MEMORY	
ABSOLUTE DB ACCESS	
INDEXED DB ACCESS	 Index Address identifier DB [Basic expression , Basic expression] in the case of bit access only
STRUCTURED DB ACCESS	

Table B-3 Absolute Addresses, continued

Rule	Syntax Diagram
Address Identifier DB	 Address identifier DB IDENTIFIER . D Size prefix
Memory Prefix	 E I Input A Q Output M M Bit Memory PE PI Peripheral Input PA PQ Peripheral Output SIMATIC Mnemonic IEC Mnemonic
Size Prefix for Memory and DB	
Address for Memory and DB	
Access to Local Instance	

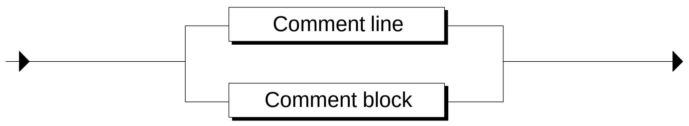
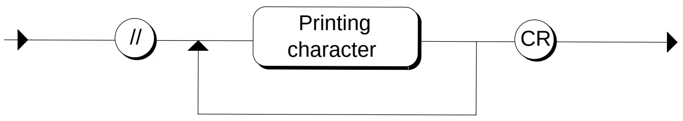
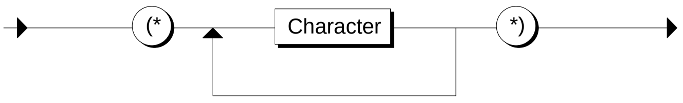
B.2 Remarks

Points to Note

The following are the most important points to be observed when inserting remarks:

- Nesting of comments is not permitted
- They can be inserted at any point in the syntax rules but not in the lexical rules.

Table B-4 Remarks

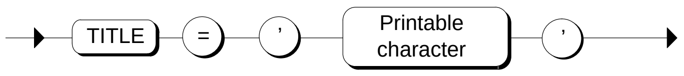
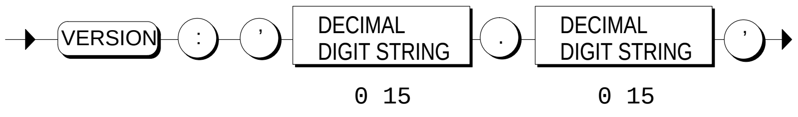
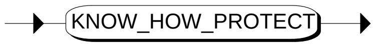
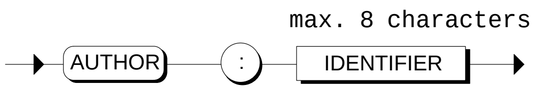
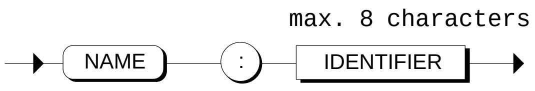
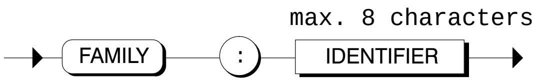
Rule	Syntax Diagram
COMMENTS	
COMMENT LINE	
COMMENT BLOCK	

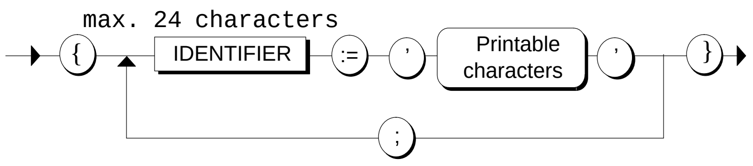
B.3 Block Attributes

Points to Note

Block attributes can be placed after the BLOCK IDENTIFIER and before the declaration of the first variables or parameters subsection using the syntax indicated.

Table B-5 Attributes

Rule	Syntax Diagram
TITLE	
VERSION	 VERSION : ' DECIMAL DIGIT STRING 0 15 . DECIMAL DIGIT STRING 0 15 '
BLOCK PROTECTION	
AUTHOR	
NAME	
BLOCK FAMILY	

System attributes for blocks	 max. 24 characters
	{
	IDENTIFIER
	:=
	'
	Printable characters
	'
	}
	;

Syntax Rules C

Definition

The syntax rules develop from the lexical rules and describe the structure of SCL. Within the framework of these rules, you can create your SCL program without format restrictions.

Chapter Overview

Formal Considerations

Each rule has a name which precedes it. If a rule is used in a higher-level rule, its name appears in an oblong box.

If the name in the oblong box is written in upper case letters, this means it is a token, which is described in the lexical rules.

In Appendix A you will find information about rule names which appear in a box with rounded corners or a circle.

Points to Note

The free-format characteristic means the following:

- You can insert formatting characters at any point.
- You can insert comment blocks and lines (see Section 7.6).

C.1 Subunits of SCL Source Files

Table C-1 Syntax of SCL Source Files

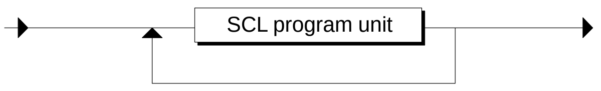
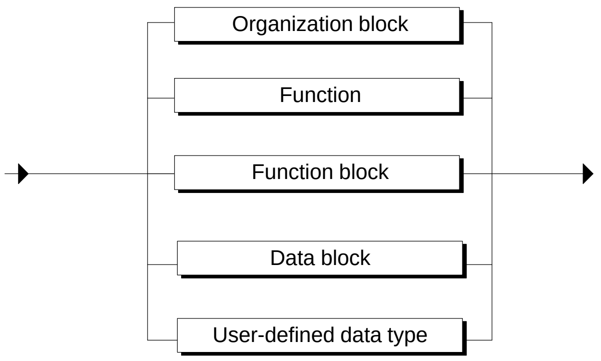
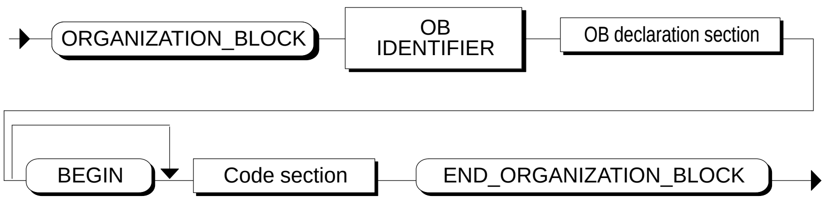
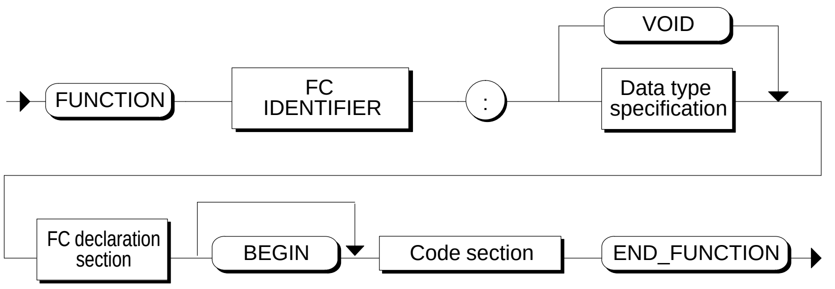
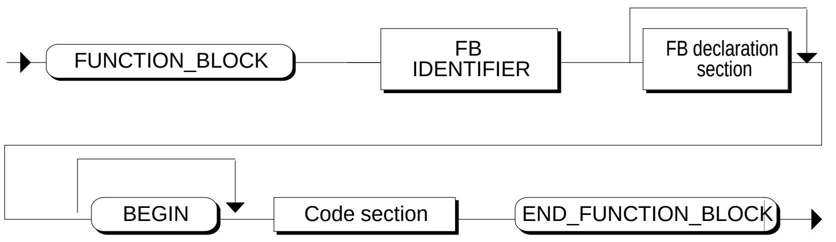
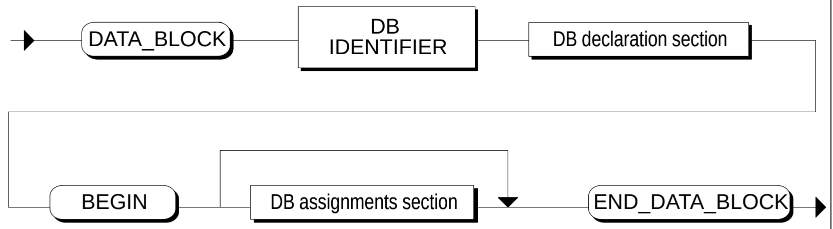
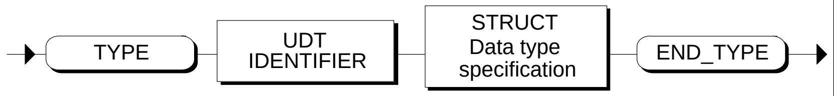
Rule	Syntax Diagram
SCL Program	
SCL Program Unit	 Organization block Function Function block Data block User-defined data type
Organization Block	 ORGANIZATION_BLOCK OB IDENTIFIER OB declaration section BEGIN Code section END_ORGANIZATION_BLOCK
Function Note that in the case of functions without VOID in the code section the return value must be assigned to the function name.	 VOID FUNCTION FC IDENTIFIER : Data type specification FC declaration section BEGIN Code section END_FUNCTION
Function Block	 FUNCTION_BLOCK FB IDENTIFIER FB declaration section BEGIN Code section END_FUNCTION_BLOCK

Table C-1 Syntax of SCL Source Files, continued

Rule	Syntax Diagram
Data Block	 DATA_BLOCK DB IDENTIFIER DB declaration section BEGIN DB assignments section END_DATA_BLOCK
User-Defined Data Type	 TYPE UDT IDENTIFIER STRUCT Data type specification END_TYPE

C.2 Structure of Declaration Sections

Table C-2 Syntax of Declaration Section

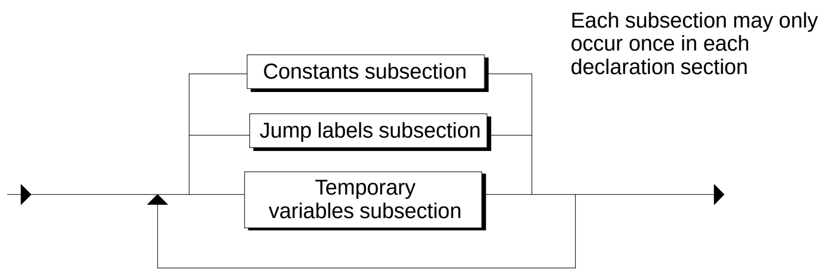
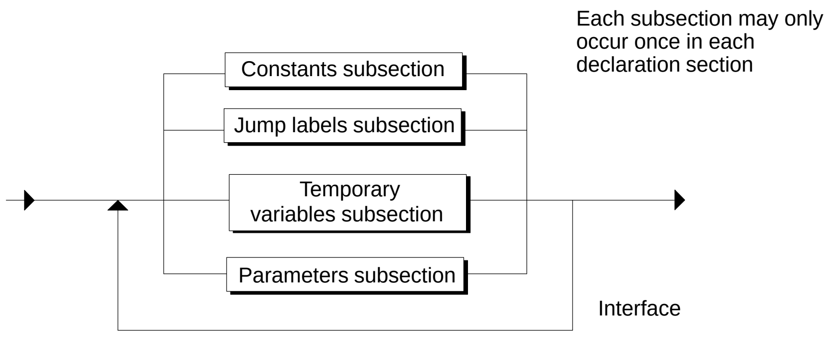
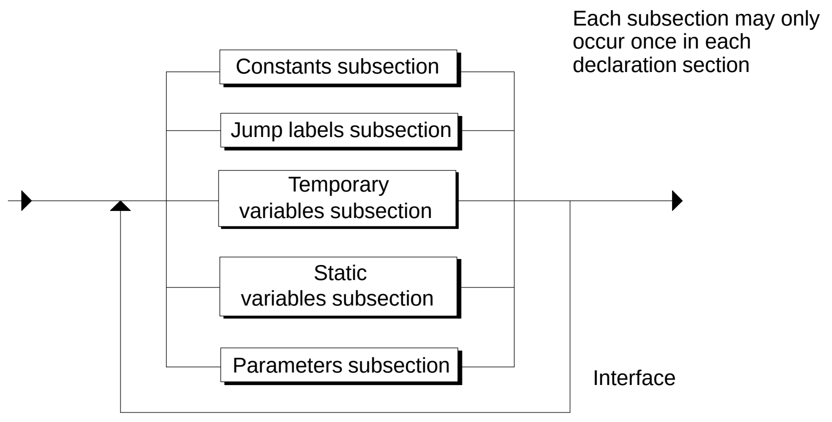
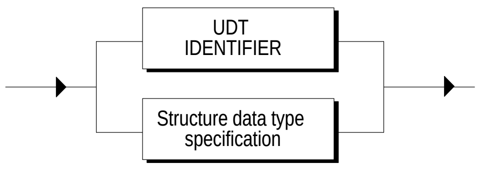
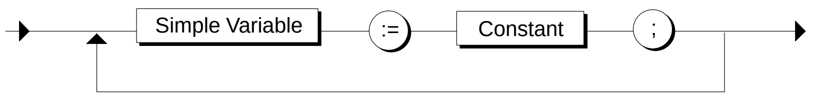
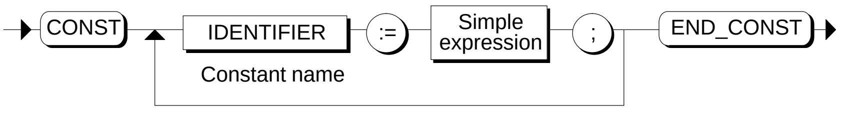
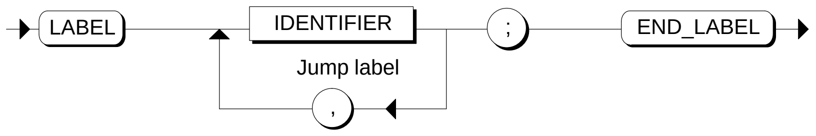
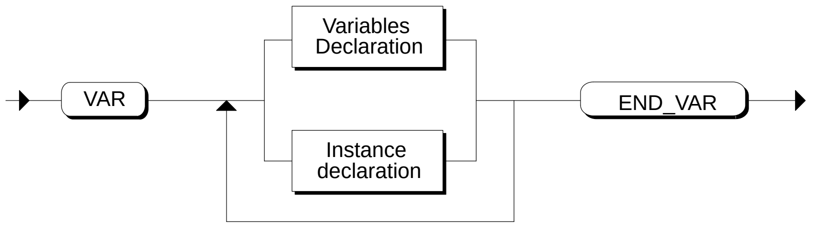
Rule	Syntax Diagram
OB Declaration Section	 Each subsection may only occur once in each declaration section Constants subsection Jump labels subsection Temporary variables subsection
FC Declaration Section	 Each subsection may only occur once in each declaration section Constants subsection Jump labels subsection Temporary variables subsection Parameters subsection Interface
FB Declaration Section	 Each subsection may only occur once in each declaration section Constants subsection Jump labels subsection Temporary variables subsection Static variables subsection Parameters subsection Interface
DB Declaration Section	

Table C-3 Syntax of Declaration Subsections

Rule	Syntax Diagram
DB Assignment Section	
Constant Subsection	 CONST IDENTIFIER Constant name := Simple expression ; END_CONST
Jump Label Subsection	
Static Variable Subsection	 VAR Variables Declaration Instance declaration END_VAR

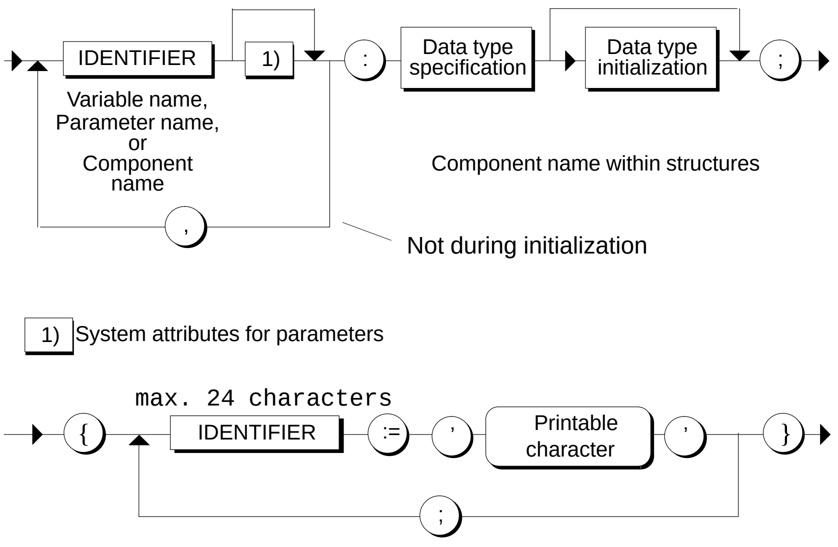
Variable Declaration	 IDENTIFIER
	1)
	:
	Data type specification
	Data type initialization
	;
	Variable name, Parameter name, or Component name
	,
	Component name within structures
	Not during initialization
	1) System attributes for parameters
	max. 24 characters
	{
	IDENTIFIER
	:=
	'
	Printable character
	'
	}
	;

Table C-3 Syntax of Declaration Subsections, continued

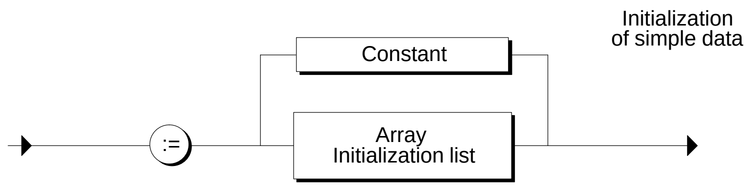
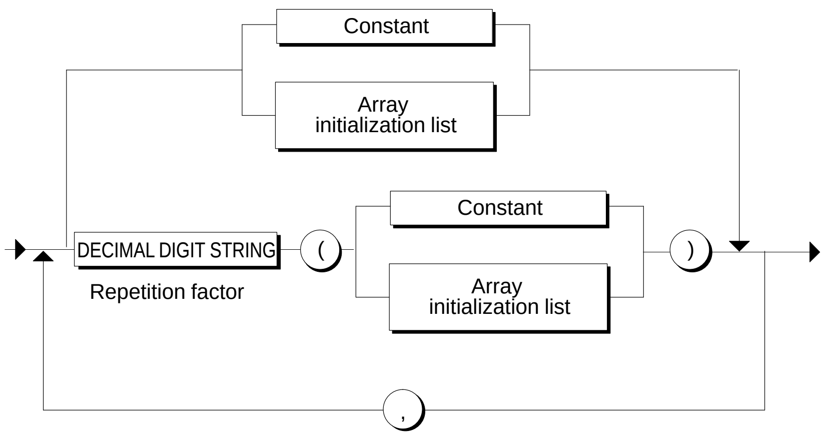
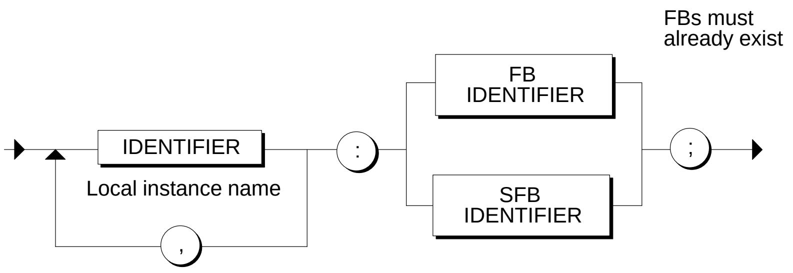
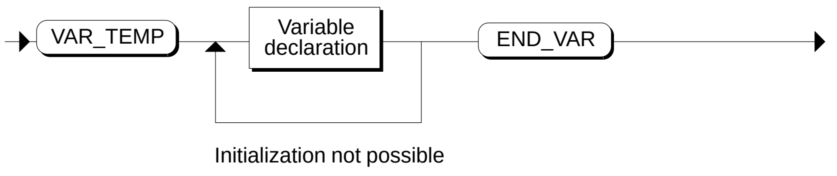
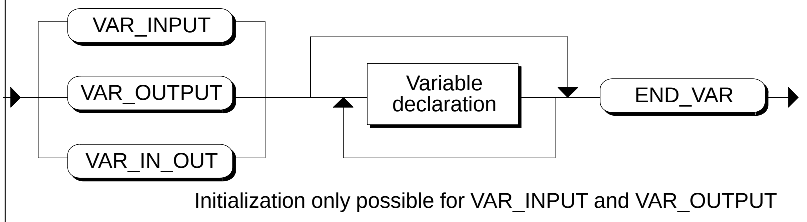
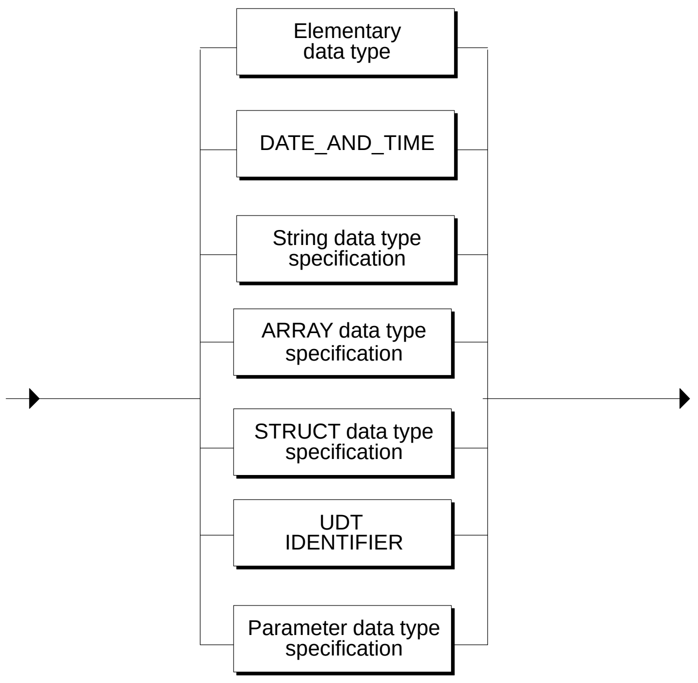
Rule	Syntax Diagram
Data Type Initialization	 Initialization of simple data := Constant Array Initialization list
Array Initialization List	 Constant Array initialization list DECIMAL DIGIT STRING Repetition factor (Constant Array initialization list) ,
Instance Declaration	 FBs must already exist IDENTIFIER Local instance name , : FB IDENTIFIER SFB IDENTIFIER ;
Temporary Variable Subsection	 VAR_TEMP Variable declaration END_VAR Initialization not possible

Table C-3 Syntax of Declaration Subsections, continued

Rule	Syntax Diagram
Parameter Subsection	 VAR_INPUT VAR_OUTPUT VAR_IN_OUT Variable declaration END_VAR Initialization only possible for VAR_INPUT and VAR_OUTPUT
Data Type Specification	 Elementary data type DATE_AND_TIME String data type specification ARRAY data type specification STRUCT data type specification UDT IDENTIFIER Parameter data type specification

C.3 Data Types in SCL

Table C-4 Syntax of Data Types in Declaration Section

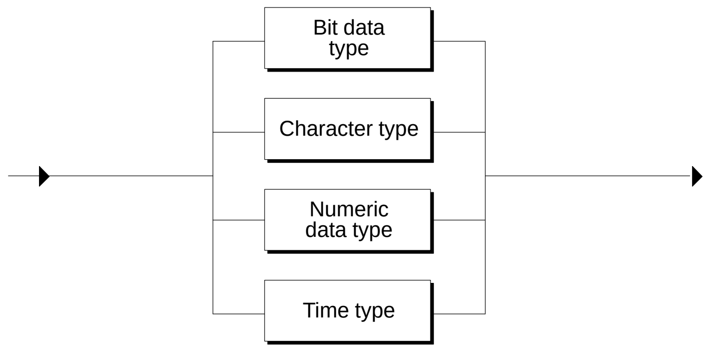
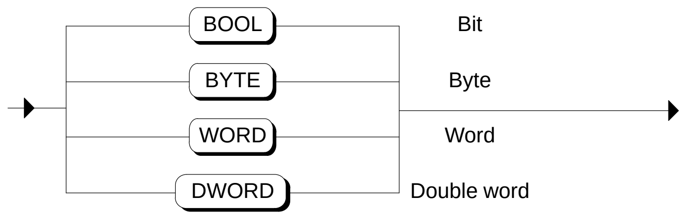
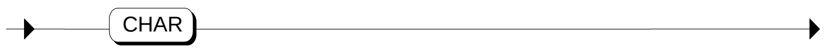
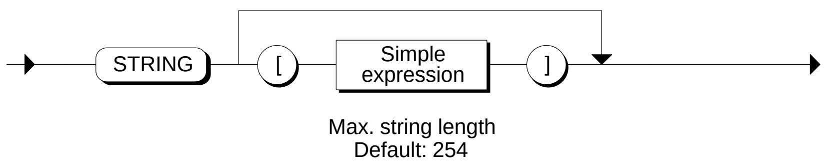
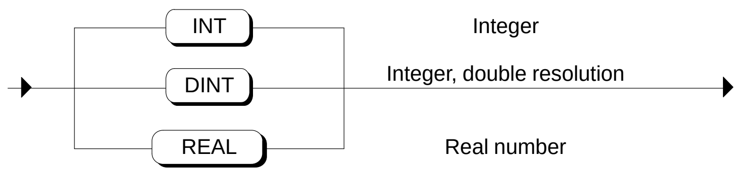
Rule	Syntax Diagram
Elementary Data Type	 Bit data type Character type Numeric data type Time type
Bit Data Type	 BOOL Bit BYTE Byte WORD Word DWORD Double word
Character Type	
STRING Data Type Specification	 STRING [Simple expression] Max. string length Default: 254
Numeric Data Type	 INT Integer DINT Integer, double resolution REAL Real number

Table C-4 Syntax of Data Types in Declaration Section, continued

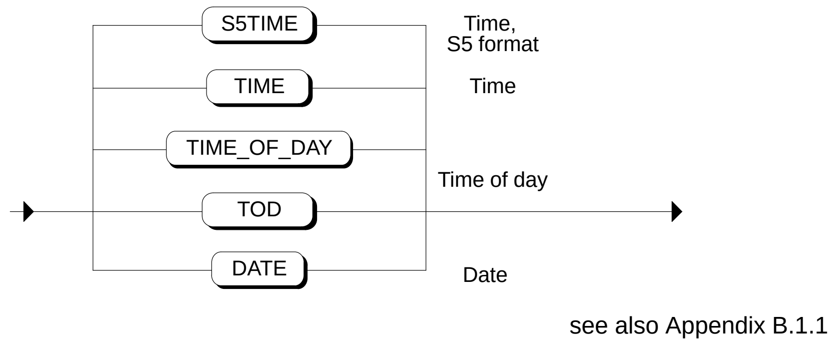
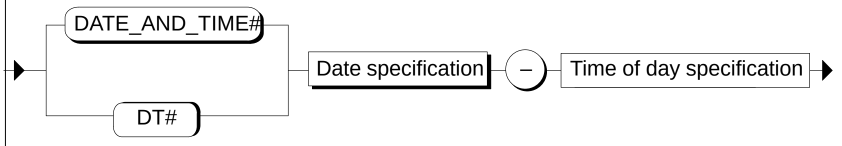
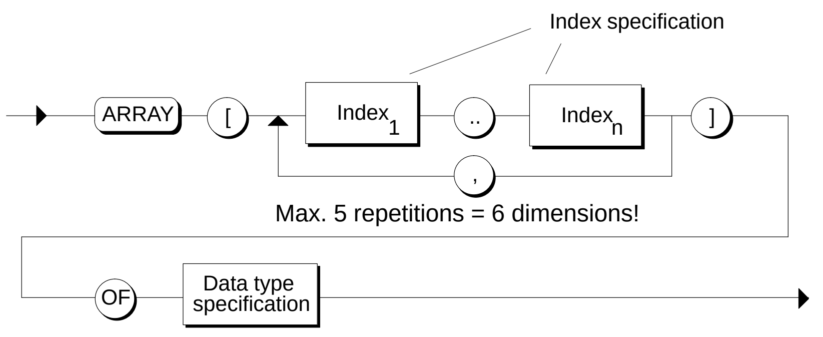
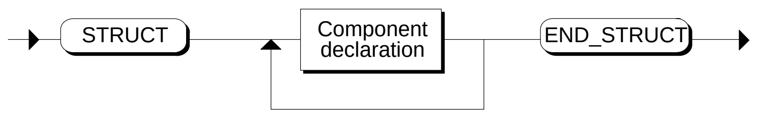
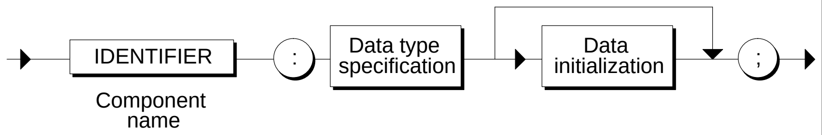
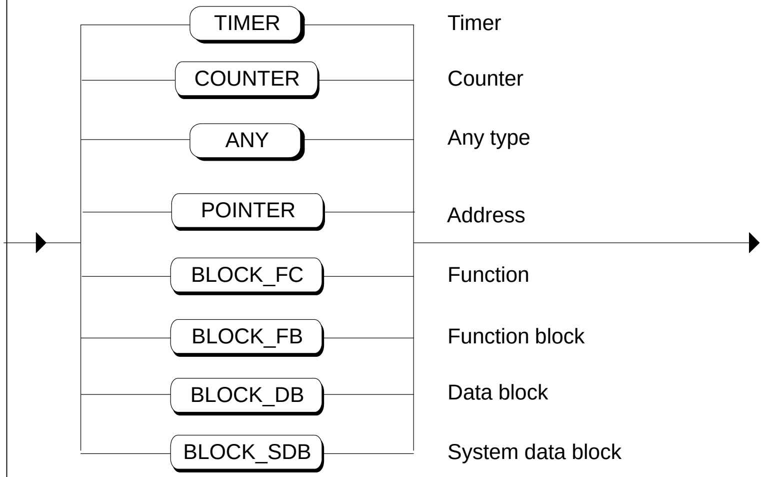
Rule	Syntax Diagram
Time Type	 S5TIME Time, S5 format TIME Time TIME_OF_DAY Time of day TOD DATE Date see also Appendix B.1.1
DATE_AND_TIME	 DATE_AND_TIME# DT# Date specification – Time of day specification
ARRAY Data Type Specification	 Index specification ARRAY [Index1 .. Indexn] , Max. 5 repetitions = 6 dimensions! OF Data type specification
STRUCT Data Type Specification Remember that the keyword END_STRUCT must be terminated by a semicolon.	

Table C-4 Syntax of Data Types in Declaration Section, continued

Rule	Syntax Diagram
Component Declaration	 IDENTIFIER Component name : Data type specification Data initialization ;
Parameter Data Type Specification	 TIMER Timer COUNTER Counter ANY Any type POINTER Address BLOCK_FC Function BLOCK_FB Function block BLOCK_DB Data block BLOCK_SDB System data block

C.4 Code section

Table C-5 Syntax of Code Section

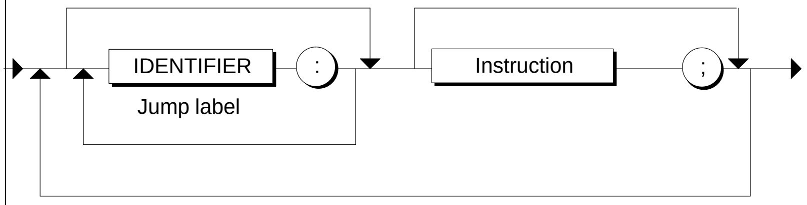
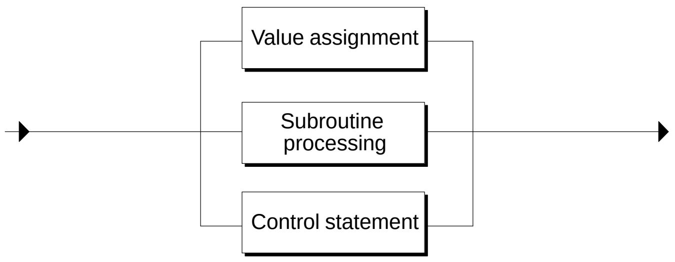
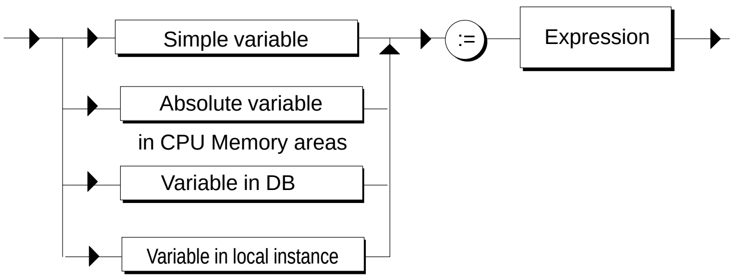
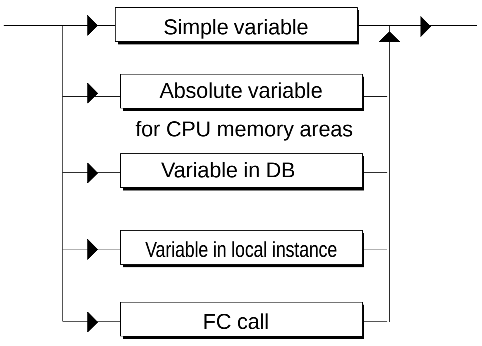
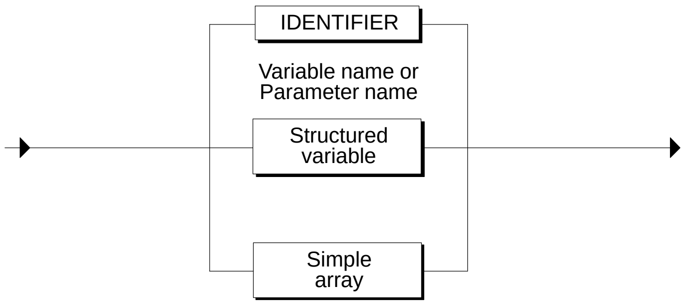
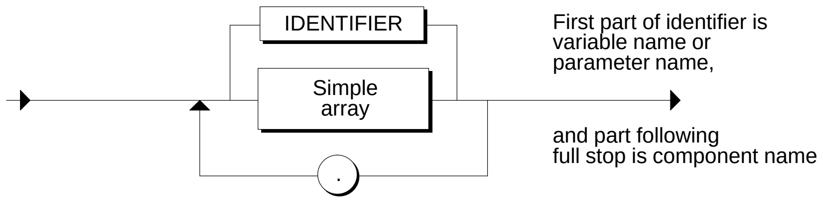
Rule	Syntax Diagram
Code Section	
Statement	 Value assignment Subroutine processing Control statement
Value Assignment	 Simple variable := Expression Absolute variable in CPU Memory areas Variable in DB Variable in local instance
Extended Variable	 Simple variable Absolute variable for CPU memory areas Variable in DB Variable in local instance FC call

Table C-5 Syntax of Code Section, continued

Rule	Syntax Diagram
Simple Variable	 IDENTIFIER Variable name or Parameter name Structured variable Simple array
Structured Variable	 IDENTIFIER Simple array . First part of identifier is variable name or parameter name, and part following full stop is component name

C.5 Value Assignments

Table C-6 Syntax of Value Assignments

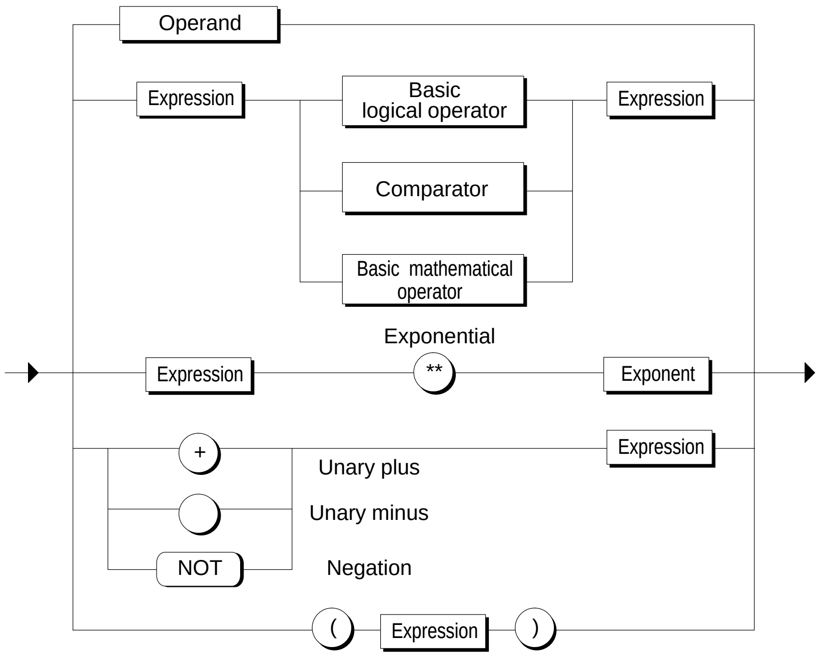
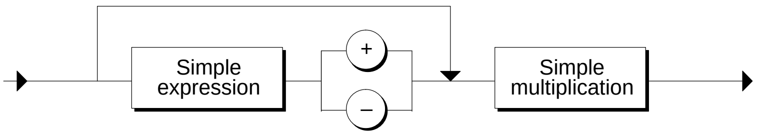
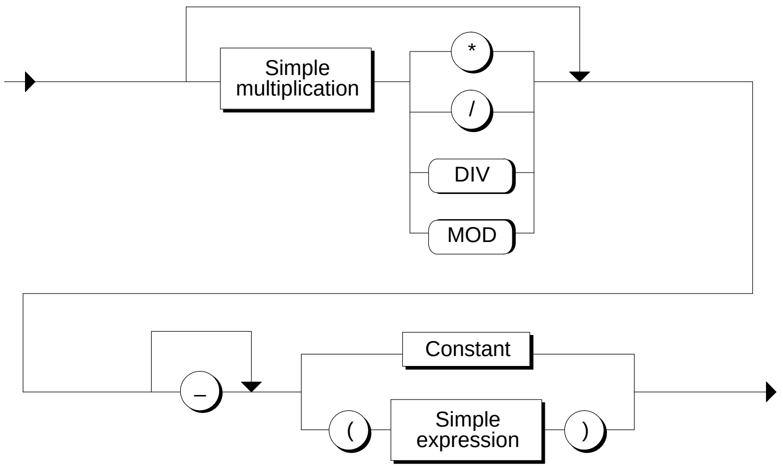
Rule	Syntax Diagram
Expression	 Operand Expression Basic logical operator Comparator Basic mathematical operator Expression Exponential Expression ** Exponent + Unary plus Unary minus NOT Negation Expression (Expression)
Simple Expression	
Simple Multiplication	 Simple multiplication * / DIV MOD – Constant (Simple expression)

Table C-6 Syntax of Value Assignments, continued

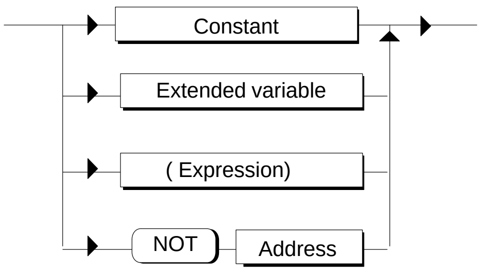
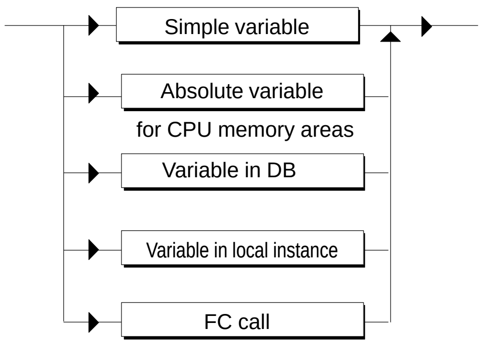
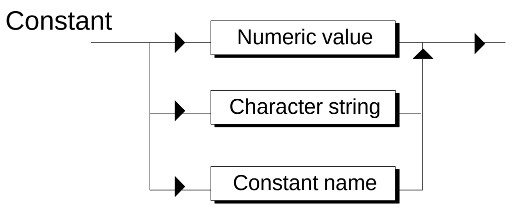
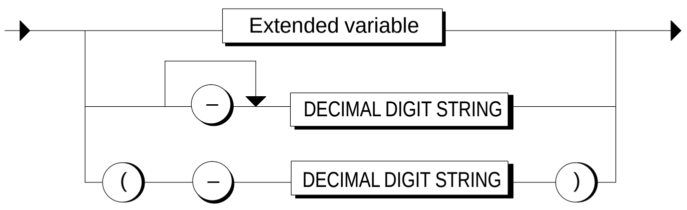
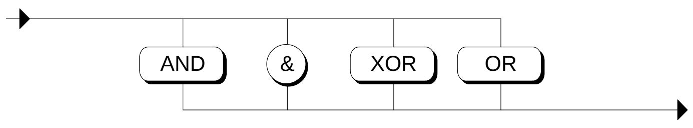
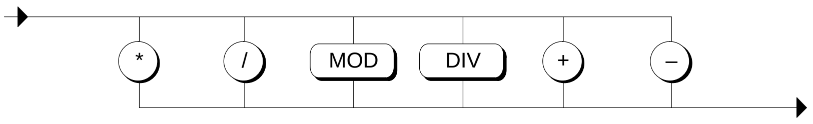
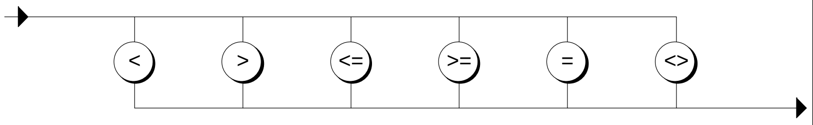
Rule	Syntax Diagram
Address	 Constant Extended variable (Expression) NOT Address
Extended Variable	 Simple variable Absolute variable for CPU memory areas Variable in DB Variable in local instance FC call
Constant	 Constant Numeric value Character string Constant name
Exponential Expression	 Extended variable – DECIMAL DIGIT STRING (– DECIMAL DIGIT STRING)
Basic Logical Operator	

Table C-6 Syntax of Value Assignments, continued

Rule	Syntax Diagram
Basic Mathematical Operator	
Comparator	

C.6 Function and Function Block Calls

Table C-7 Syntax of Function and Function Block Calls

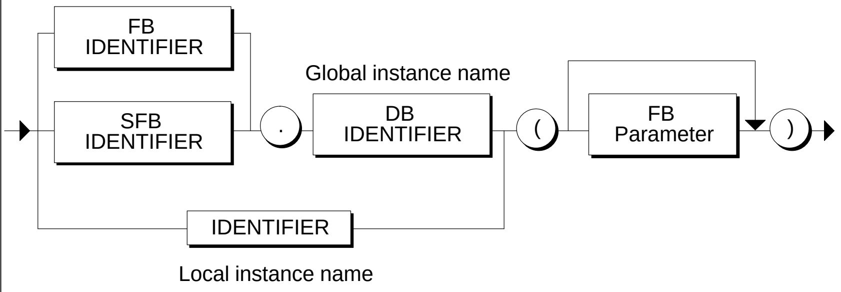
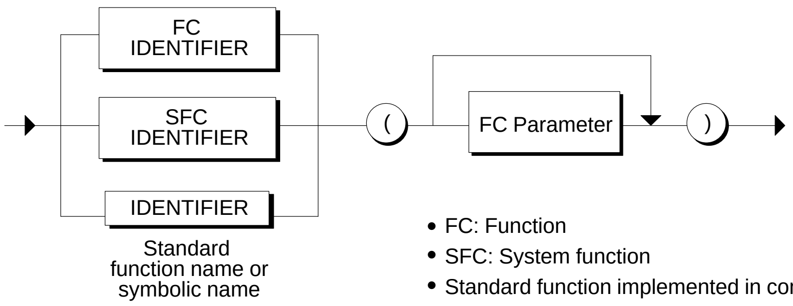
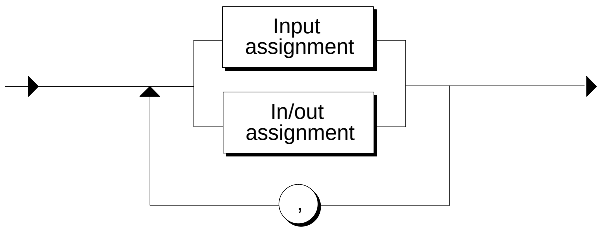
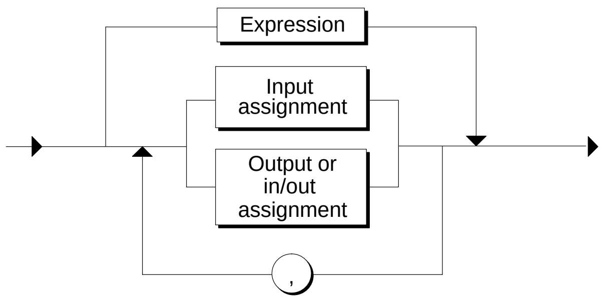
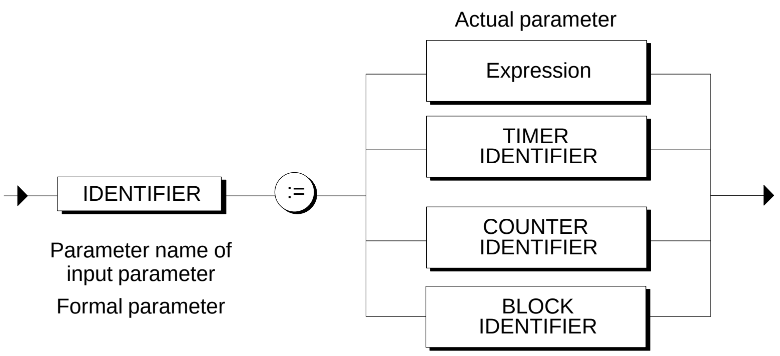
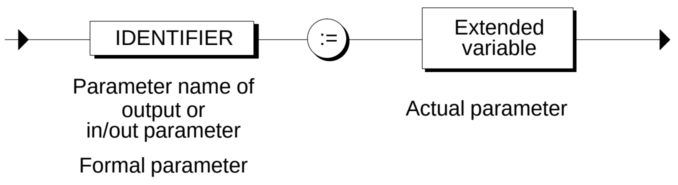
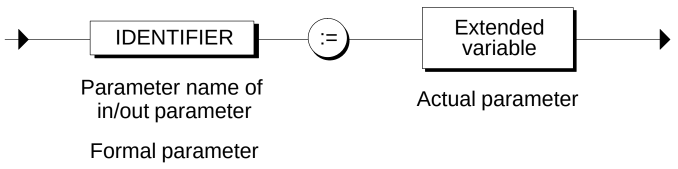
Rule	Syntax Diagram
FB Call	FB: Function block SFB: System function block  FB IDENTIFIER SFB IDENTIFIER . Global instance name DB IDENTIFIER (FB Parameter) IDENTIFIER Local instance name
Function Call	 FC IDENTIFIER SFC IDENTIFIER IDENTIFIER Standard function name or symbolic name (FC Parameter) • FC: Function • SFC: System function • Standard function implemented in compil
FB Parameter	
FC Parameter	 Expression Input assignment Output or in/out assignment ,

Table C-7 Syntax of Function and Function Block Calls, continued

Rule	Syntax Diagram
Input Assignment	 Actual parameter Expression TIMER IDENTIFIER COUNTER IDENTIFIER BLOCK IDENTIFIER IDENTIFIER := Parameter name of input parameter Formal parameter
Output or In/Out Assignment	 IDENTIFIER := Extended variable Parameter name of output or in/out parameter Formal parameter Actual parameter
In/Out Assignment	 IDENTIFIER := Extended variable Parameter name of in/out parameter Formal parameter Actual parameter

C.7 Control Statements

Table C-8 Syntax of Control Statements

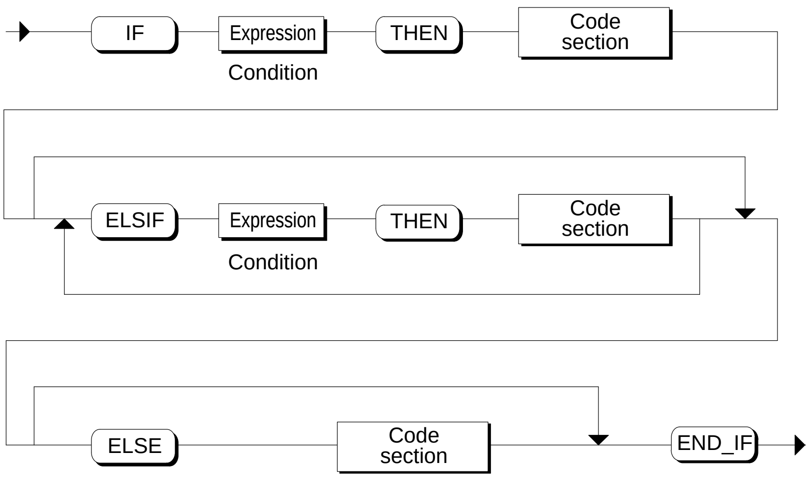
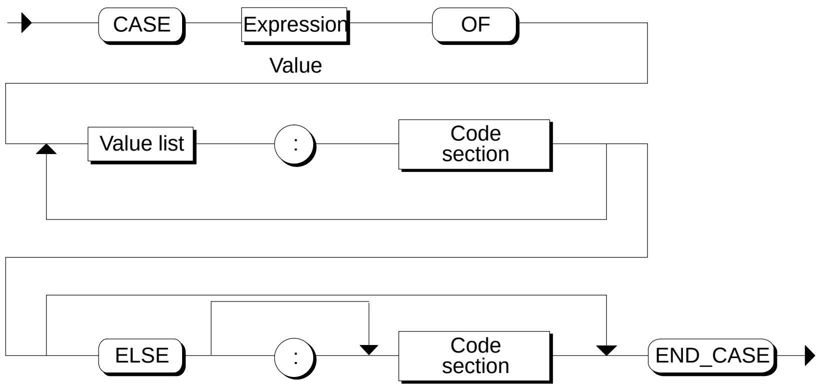
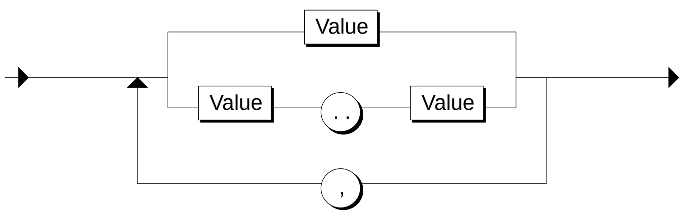
Rule	Syntax Diagram
IF Statement Do not forget that the keyword END_IF must be terminated by a semicolon.	 IF Expression Condition THEN Code section ELSIF Expression Condition THEN Code section ELSE Code section END_IF
CASE Statement Do not forget that the keyword END_CASE must be terminated by a semicolon.	 CASE Expression Value OF Value list : Code section ELSE : Code section END_CASE
Value List	

Table C-8 Syntax of Control Statements, continued

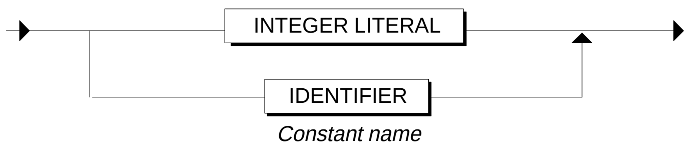
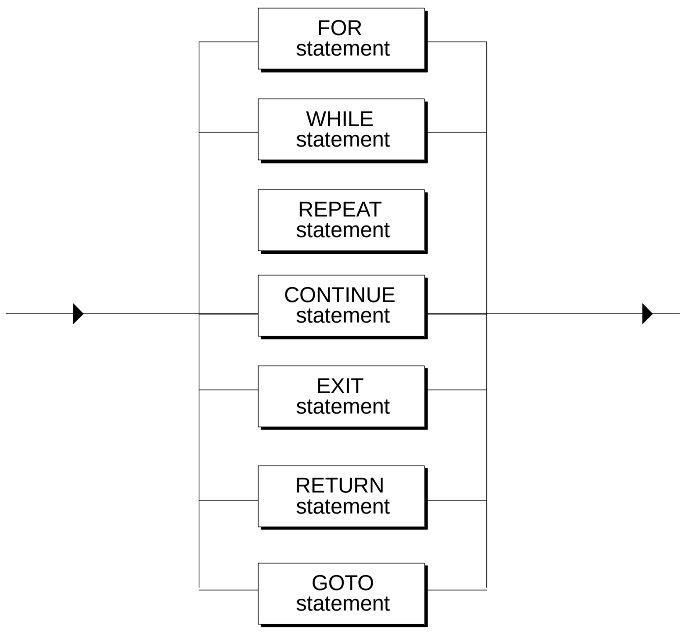
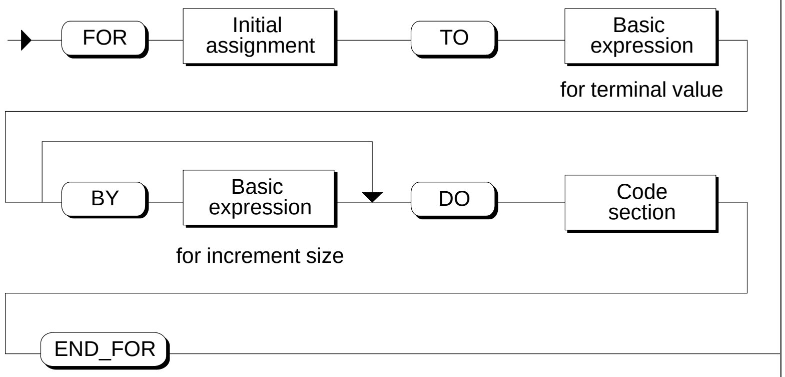
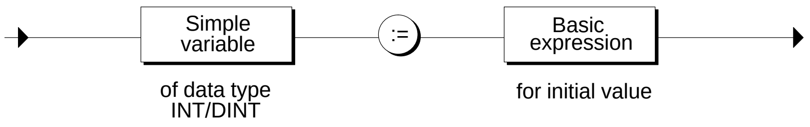
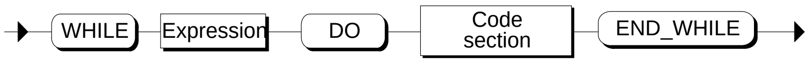
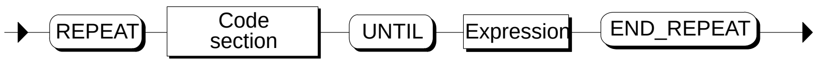
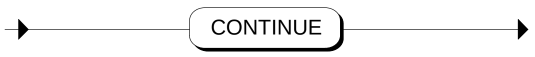
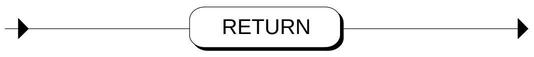
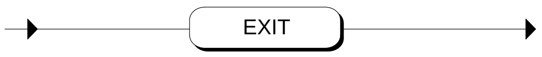
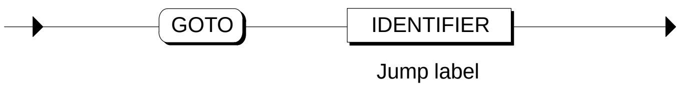
Rule	Syntax Diagram
Value	
Iteration and Jump Instructions	 FOR statement WHILE statement REPEAT statement CONTINUE statement EXIT statement RETURN statement GOTO statement
FOR Statement Do not forget that the keyword END_FOR must be terminated by a semicolon.	 FOR Initial assignment TO Basic expression for terminal value BY Basic expression for increment size DO Code section END_FOR

Table C-8 Syntax of Control Statements, continued

Rule	Syntax Diagram
Initial Assignment	 Simple variable := Basic expression of data type INT/DINT for initial value
WHILE Statement Do not forget that the keyword END_WHILE must be terminated by a semicolon.	
REPEAT Statement Do not forget that the keyword END_REPEAT must be terminated by a semicolon.	
CONTINUE Statement	
RETURN Statement	
EXIT Statement	
Program Jump	

References D

/12/ Technical Overview: *S7-300 Programmable Controller*, Configuration and Application

/13/ Technical Overview: *S7-400 Programmable Controller*, Configuration and Application

/14/ Technical Overview: *M7-300/M7-400 Programmable Controllers*, Configuration and Application

/20/ Technical Overview: *S7-300/S7-400 Programmable Controllers*, Programming

/25/ Technical Overview: *M7 Programmable Controller*, Programming

/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*

/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

/234/ Programming Manual: *System Software for S7-300 and S7-400* Program Design

/235/ Reference Manual: *System Software for S7-300 and S7-400* System and Standard Functions

/236/ Manual: FBD *for S7-300 and 400,* Programming

/237/ *Master Index*, STEP 7

/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

/253/ Manual: *C Programming for S7-300 and S7-400,*
Writing C Programs

/254/ Manual: *Continuous Function Charts (CFC) for S7 and M7,*
Programming Continuous Function Charts

/290/ User Manual: *ProC/C++ for M7-300 and M7-400,*
Writing C Programs

/291/ User Manual: *ProC/C++ for M7-300 and M7-400,*
Debugging C Programs

/800/ *DOCPRO*
Creating Wiring Diagrams (CD only)

/803/ Reference Manual: *Standard Software for S7-300/400*
STEP 7 Standard Functions, Part 2 (CD only)

Glossary

A

Actual Parameter	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 a component of an instruction that specifies the data on which an operation is to be performed. It can be addressed in both absolute and symbolic terms.
Addressing, Absolute	With absolute addressing, the memory location of the address to be processed is given. Example: The address Q 4.0 describes bit 0 in byte 4 of the process-image output area.
Addressing, Symbolic	Using symbolic addressing, the address to be processed is entered as a symbol and not as an address. The assignment of a symbol to an address is made in the symbol table.
Address Identifier	An address identifier is that part of an address of an operation which contains information, for example, the details of the memory area where the operation can access a value (data object) with which it is to perform a logic operation, or the value of a variable (data object) with which it is to perform a logic operation. In the instruction ”Value := IB10”, ”IB” is the address identifier (”I” designates the input area of the memory and ”B” stands for a byte in that area).
Array	An array is a complex data type consisting of a number of data elements of the same type. Those data elements in turn can be elementary or complex.

Attribute	An attribute is a characteristic which can be attached to a block identifier or variable name, for example. In SCL there are attributes for the following items of information: block title, release version, block protection, author, block name, block family.

B

BCD Format	In STEP 7, internal specification of timers and counters is is done in BCD format only. BCD stands for binary coded decimal.
Bit Memory (M)	A memory area in the system memory of a SIMATIC S7 CPU. This area can be accessed using write or read access (bit, byte, word, and double word). The bit memory area can be used by the user to store interim results.
Block	Blocks are subunits of a user program delimited according to their function, their structure or their purpose. In STEP 7 there are logic blocks (FBs, FCs, OBs, SFCs and SFBs), data blocks (DBs and SDBs) and user-defined data types (UDTs).
Block Call	A block call calls a block in a STEP 7 user program. Organization blocks are only called by the operating system; all other blocks are called by the STEP 7 user program.
Block Class	Blocks are subdivided according to the type of information they contain into the following two classes: Logic blocks and data blocks; user-defined data types (UDTs) can be categorized as data blocks.
Block Protection	Block protection refers to the facility of protecting individual blocks against decompilation. This is done by employing the keyword ”KNOW_HOW_PROTECTED” when the block source file is compiled.
Block Comment	Additional information about a block (for example, explanatory information about the automated process) which can not be loaded into the RAM of the SIMATIC S7 programmable controllers.
Block Status	⇒ Continuous Monitoring
Block Type	The block architecture of STEP 7 recognizes the following block types: organization blocks, function blocks, functions, data blocks as well as system function blocks, system functions, system data blocks and user-defined data types. ⇒ Block

Breakpoint	This function can be used to switch the CPU to STOP mode at specific points in the program. When the program reaches a breakpoint, debugging functions such as single-step instruction processing or controlling/monitoring variables can be performed.

C

Call Hierarchy	Any block has to be called before it can be processed. The order and nesting sequence of the operation calls by which blocks are called is referred to as the operation call hierarchy.
Call Interface	The call interface is defined by the input, output and in/out parameters (formal parameters) of a block in the STEP 7 user program. When the block is called, those parameters are replaced by the actual parameters.
CASE Statement	This statement is a selective branching statement. It is used to select a specific program branch from a choice of n branches on the basis of the value of a selection expression.
Compilation	The process of generating a user program from a source file.
Compilation, Incremental	When using incremental input, the program is not checked for possible input errors until it is compiled. Executable code is not generated until no more errors are found.
Constant, Literal	Constants with symbolic names are placeholders for constant values in logic blocks. Symbolic constants are used for improving the legibility of a program.
Constant, Symbolic	Constants whose value and type are determined by their formal format. A distinction is made between literals, character literals and time literals.
Container	Object in the SIMATIC Manager user interface which can be opened and can contain other folders and objects.
CONTINUE Statement	Exits a control loop and restarts it using the next value for that control variable.
Continuous Monitoring	SCL debugging mode. When debugging a program in continuous monitoring mode, you can test out a series of instructions. This series of instructions is referred to as the monitoring range.

Conversion, Explicit	Explicit conversion refers to inserting a conversion function in the source file. When two addresses of differing data types are linked, the programmer must perform an explicit conversion in the following cases: if data is being changed into a different type class, for example, from a bit data type to a numeric data type, and – if the destination data type is of a lower order than the source data type – if data is changed to another type of the same class.
Conversion, Implicit	Implicit conversion refers to a conversion function being inserted automatically by the compiler. When two addresses of differing data types are linked, automatic conversion takes place if the operation does not involve a change of type class and if the destination data type is not of a lower order than the source data type.
Counter	Counters are components of the system memory of the CPU. The contents of a counter are updated by the operating system asynchronously with the user program. STEP 7 instructions are used to define the precise function of a counter (for example, up counter) and to activate it (for example, start).

D

Data, Global	Global data refer to memory areas of the CPU that can be accessed from any point in the program (for example, bit memory).
Data, Static	Static data are local data of a function block which are stored in the instance data block and are thus retained until the next time the function block is processed.
Data, Temporary	Temporary data are assigned to a logic block at local level and do **not** occupy any static memory areas since they are stored in the CPU stack. Their value is only retained while the block concerned is running.
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.
Data Type	Data types are used to specify how the value of a variable or constant is to be used in the user program. In SCL there are three classes of data type available to the user, as follows: • Elementary data types (data type, elementary) • Complex data types (data type, complex) • User-defined data types (UDTs).

Data Type, User-defined	User-defined data types (UDTs) are created by the user in the data type declaration. Each one is assigned a unique name and can be used any number of times. Thus, a user-defined data type can be used to generate a number of data blocks with the same structure (for example, controller).
Data Type Declaration	The data type declaration is where the user declares user-defined data types.
Data Type, Elementary	Elementary data types are predefined data types in accordance with IEC 1131-3. Examples: the data type ”`BOOL`” defines a binary variable (”`Bit`”); the data type ”`INT`” defines a 16-bit fixed point variable.
Data Type, Complex	A distinction is made between structures and arrays. ”Structures” are made up of various different data types (for example, elementary data types). ”Arrays” consist of a number of identical elements of a single data type. The data types `STRING` and `DATE_AND_TIME` are also complex data types.
Declaration Section	This is where the local data of a logic block are declared.
Declaration Type	The declaration type specifies how a parameter or a local variable is to be used by a block. There are input parameters, output parameters and in/out parameters as well as static and termporary variables.
Decompilation	Decompilation to STL enables a block downloaded to the CPU to be opened and viewed on any programming device or PC. Certain components of the block such as symbols and comments may be missing.
Download to PLC	Transfer of loadable objects (for example, logic blocks) from the programming device to the working memory of a programmable module. This can be done either via a programming device directly connected to the CPU or; for example via PROFIBUS.

E

Enable (EN)	In STEP 7 every block has an ”Enable” input (EN) that can be set when a block is called. If the signal present at EN is 1, the block is called. If the signal is 0, it is not.

Enable Out (ENO)

In STEP 7 every block has an ”Enable Output” (ENO). Within the block, the programmer can link the input ”Enable” with an internal value (UND). The result is automatically assigned to the output ENO. ENO enables the processing of succeeding blocks in block call sequences to be made dependent on correct processing of the preceding block.

EXIT Statement

Exits a control loop.

Expression

In SCL, an expression is a means of processing data. A distinction is made between mathematical, logical and comparative expressions.

F

FOR Statement

A FOR instruction is used to repeat a sequence of instructions for as long as a control variable remains within a specified range.

Formal Parameter

A formal parameter is a placeholder for the ”actual” parameter in configurable logic blocks. In the case of FBs and FCs, the formal parameters are declared by the programmer, in the case of SFBs and SFCs they already exist. When a block is called, the formal parameters are assigned actual parameters with the result that the called block works with the actual values. The formal parameters count as local block data and are subdivided into input, output and in/out parameters.

Free-Edit Mode

The free-edit mode is possible when programming with SCL. A program can be written with the aid of any text editor. The actual program code is generated only when the source file is compiled. At that point any errors are detected as well. This mode is suited to symbolic programming of standard programs.

In free-edit mode, the blocks or the complete user program are edited in text file form. The syntax is not checked until the source file is compiled. SCL uses free-edit mode.

Function (FC)

According to the International Electrotechnical Commission’s IEC 1131–3 standard, functions are logic blocks that do not reference an instance data block, meaning they do not have a ’memory’. A function allows you to pass parameters in the user program, which means they are suitable for programming complex functions that are required frequently, for example, calculations.

Function Block (FB)

According to the International Electrotechnical Commission’s IEC 1131–3 standard, function blocks are logic blocks that reference an instance data block, meaning they have static data. 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.

G

Global Data

Global data is data that can be accessed from any logic block (FC, FB or OB). Specifically it includes bit memory (M), inputs (I), outputs (O), timers, counters and elements of data blocks (DBs). Global data can be addressed in either absolute or symbolic terms.

GOTO Statement

A GOTO statement executes an immediate jump to a specified label.

H

HOLD Mode

The HOLD state is reached from the RUN mode via a request from the programming device. Special test functions are possible in this mode.

I

Identifier

Identifiers are used to address SCL language objects. There are the following classes of identifier: standard identifiers, predefined names and keywords, absolute identifiers (or address identifiers), user-defined names, for example, for variables and jump labels or symbolic names generated by a symbol table.

In/Out Parameter

In/out parameters are used in functions and function blocks. In/out parameters are used to transfer data to the called block, where they are processed, and to return the result to the original variable from the called block.

Input Parameter

Only functions and function blocks have input parameters. Input parameters are used to transfer data to the called block for processing.

Instance

The term ”instance” refers to a function block call. The function block concerned is assigned an instance data block or a local instance. If a function block in a STEP 7 user program is called n times, each time using different parameters and a different instance data block name, then there are n instances.

```
FB13.DB3 (P3:=...), FB13.DB4 (P4:=...),
FB13.DB5 (P5:=...), .....FB13.DBn (Pn:=...).
```

Instance Data Block (Instance DB)

An instance data block stores the formal parameters and static local data for a function block. An instance data block can be assigned to an FB call or a function block call hierarchy. It is generated automatically in SCL.

Instance, Local

A local instance is defined in the static variable section of a function block. Instead of a complete instance data block, only a local section is used as the data area for the function block which is called using the local instance name.

Instruction

An instruction is a component of a statement specifying what action the processor is to perform.

Integer (INT)

Integer (INT) is an elementary data type. Its format is 16-bit whole number.

K

Keyword

Keywords are used in SCL to mark the beginning of a block, to mark subsections in the declaration section and to identify instructions. They are also used for attributes and comments.

L

Lexical Rule

The lower level of rules in the formal language description of SCL consists of the lexical rules. When applied, they do not permit unrestricted format; that is, addition of spaces and control characters is not permitted.

Local Data

Local data refers to the data assigned to a specific logic block which is declared in its declaration section. It consists of (depending on the particular block) the formal parameters, static data and temporary data.

Logic Block	A logic block in SIMAT IC S7 is a block that contains a section of a STEP 7 user program. In contrast, a data block contains only data. There are the following types of logic blocks: organization blocks (OBs), function blocks (FBs), functions (FCs), system function blocks (SFBs) and system functions (SFCs).

M

Memory Area	A SIMATIC S7 CPU has three memory areas - the load area, the working area and the system area.
Mnemonic	A mnemonic is an abbreviation for an address or a programming operation used in the program (for example, ”I” stands for input). STEP 7 supports IEC mnemonics (which are based on English terms) and SIMATIC mnemonics (which are based on the German names of operations and the SIMATIC addressing conventions).
Multiple Instance	When multiple instances are used, the instance data block holds the data for a series of function blocks within a function call hierarchy.

N

Non Term	A non term is a complex element that is described by another lexical or syntax rule.

O

Off-Line	Off-line designates the operating mode whereby the programming device is not connected (physically or logically) to the PLC.
OK Variable	The OK variable is used to indicate whether a sequence of block statements has been executed correctly or not. It is a global variable of the type BOOL.
On-Line	On-line designates the operating mode whereby the programming device is connected (physically or logically) with the PLC.
On-Line Help	STEP 7 provides the facility of obtaining context-sensitive help on the screen while working with the programming software.

Organization Block (OB)	Organization blocks form the interface between the CPU operating system and the user program. The organization blocks specify the sequence in which the user program is to be processed.
Output Parameter	The output parameters of a block in a STEP 7 user program are used to transfer results to the calling block.

P

Parameter	In SCL, a parameter is a variable of a logic block (actual parameter or formal parameter).
Parameter Type	Parameter type is a special data type for timers, counters and blocks. It can be used for input parameters of function blocks **and** functions, and for in/out parameters of function blocks only in order to transfer timer and counter readings and blocks to the called block.
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).
Process-Image Input Table (PII)	The process image of the inputs is read in from the input modules by the operating system before the user program is processed.
Process-Image Output Table (PIQ)	The process image of the outputs is transferred to the output modules at the end of the user program by the operating system.
Programming, Structured	To facilitate the implementation of complex automation tasks, a user program is subdivided into separate, self-contained subunits (blocks). Subdivision of a user program is based on functional considerations or the technological structure of the system.
Programming, Symbolic	The programming language SCL permits the use of symbolic character strings in place of addresses, for example, the address A1.1 might be replaced by ”Valve_17”. The symbol table in STEP 7 creates the link between the address and its assigned symbolic character string.
Project	A folder for storing all objects relating to a particular automation concept regardless of the number of stations, modules or how they are networked.

R

Real Number	A real number, also called a floating point number, is a positive or negative number which contains a decimal fraction, for example, 0.339 or -11.1.
REPEAT Statement	A REPEAT statement is used to repeat a sequence of statements until a break condition is satisfied.
RETURN Statement	This statement causes the program to exit the active block.
RUN Mode	In the RUN mode the user program is processed and the process image is updated cyclically. In addition, all digital outputs are enabled.
RUN-P Mode	The operating mode RUN-P is the same as RUN operating mode except that in RUN-P mode, all programming device functions are permitted without restriction.

S

S7 User Program	The S7 user program is located in the “Blocks” folder. It contains blocks that are uploaded to a programmable S7 module (for example CPU) and are capable of being run on the module as part of the program controlling a system or a process.
Scan Cycle Time	The scan cycle time is the time required by the CPU to process the user program once.
Scan Cycle Monitoring Time	If the user program processing time exceeds the set scan cycle monitoring time, the operating system generates an error message and the CPU switches to STOP mode.
SCL	PASCAL-based high-level language which conforms to the standard DIN EN-61131-3 (IEC 1131-3) and is intended for programming complex operations on a PLC, for example, algorithms and data processing tasks. Abbreviation for ”Structured Control Language”.
SCL Compiler	The SCL Compiler is a batch compiler which is used to translate a program written using a text editor (SCL source file) into M7 machine code. The blocks generated by the process are stored in the “Blocks” folder.

SCL Debugger	The SCL Debugger is a high-level language debugger used for finding logical programming errors in user programs created with SCL.
SCL Editor	The SCL Editor is a text editor specially adapted for use with SCL for creating SCL source files.
SCL Source File	An SCL source file is a file in which a program is written in SCL. The SCL source file is translated into machine code by the SCL Compiler after it has been written.
Single Step	A single step is a step in a debugging operation carried out by the SCL Debugger. In single-step debugging mode you can execute a program one instruction at a time and view the results of each step in the Results window.
Source File	A source file (text file) contains source code (ASCII text) that can be created with any text editor. A source file is translated into a user program file by a compiler (STL, SCL). Source files are stored in the “Source Files” folder under the name of the S7 program.
Statement	An instruction is the smallest indivisible unit of a user program written in a text-based language. It represents an instruction to the processor to perform a specific operation.
Statement List	Statement List is a low-level text-based programming language.
Status Word	The status word is a component of the CPU register. The status word contains status information and error information in connection with the processing of STEP 7 commands. The status bits can be read and written by the programmer. The error bits can only be read.
Structure (STRUCT)	A structure is a complex data type consisting of data elements of differing types. Those data elements can be elementary or complex.
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)

Syntax Rule

The higher level of rules in the formal description of SCL consists of the syntax rules. When they are used they are not subject to format restrictions; that is, spaces and control characters can be added.

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 Data Block (SDB)

System data blocks are data areas in the CPU which contain system settings and module parameters. System data blocks are generated and edited by the STEP 7 standard software.

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.

T

Term

A term is a basic element of a lexical or syntax rule that can not be explained by another rule but is represented in literal terms. A term can be a keyword or a single character, for example.

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

UDT	⇒ Data Type, User-Defined
Upload to PC	Transfer loadable objects (for example, logic blocks) from the load memory of a programmable module to a programming device. This can be done either via a programming device with a direct connection or, for example, via PROFIBUS.
User Data	User data are exchanged between a CPU and a signal module, function module and communications modules via the process image or direct access. Examples of user data are: digital and analog input/output signals from signal modules, control and status data from function modules.
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, TAPE_ON): Variables are declared in the declaration section.
Variable Declaration	Variable declaration involves the specification of a symbolic name, a data type and, if required, an initialization value, address and comments.
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
A&D AS E46

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: