

SIMATIC

S7-SCL V5.1 for S7-300/S7-400

Manual

This manual has the order number:
6ES7811-1CC04-8BA0

Preface, Contents	
Product Overview and Installation	1
Designing an SCL Program	2
Using SCL	3
Basic SCL Terms	4
SCL Program Structure	5
Data Types	6
Declaring Local Variables and Parameters	7
Declaring Constants and Jump Labels	8
Shared Data	9
Expressions, Operations and Addresses	10
Statements	11
Counters and Timers	12
SCL Standard Functions	13
Language Definition	14
Tips and Tricks	15
Glossary, Index	

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

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 descriptions, and only in connection with devices or components from other manufacturers which have been approved or recommended by Siemens.

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

Trademarks

SIMATIC®, SIMATIC HMI® and SIMATIC NET® are registered trademarks of SIEMENS AG.

Some of other designations used in these documents are also registered trademarks; the owner's rights may be violated if they are used by third parties for their own purposes.

Copyright © Siemens AG 2000 All rights reserved

The reproduction, transmission or use of this document or its contents is not permitted without express written authority. Offenders will be liable for damages. All rights, including rights created by patent grant or registration of a utility model or design, are reserved.

Disclaimer of Liability

We have checked the contents of this manual for agreement with the hardware and software described. Since deviations cannot be precluded entirely, we cannot guarantee full agreement. However, the data in this manual are reviewed regularly and any necessary corrections included in subsequent editions. Suggestions for improvement are welcomed.

Siemens AG
Bereich Automatisierungs- und Antriebstechnik
Geschäftsgebiet Industrie-Automatisierungssysteme
Postfach 4848, D- 90327 Nuernberg

Siemens Aktiengesellschaft

©Siemens AG 2000
Technical data subject to change.

6ES7811-1CC04-8BA0



Preface

Purpose of the Manual

This manual provides you with a complete overview of programming with S7-SCL. It supports you during the installation and setting up of the software. It includes explanations of how to create a program, the structure of user programs, and the individual language elements.

The manual is intended for programmers writing SCL programs and people involved in configuration, installation and service of programmable logic controllers. We recommend that you familiarize yourself with the example described in Chapter 2 "Designing an SCL Program". This will help you to get to know SCL quickly.

Required Experience

To understand the manual, you should have general experience of automation engineering.

You should also be familiar with working on computers or PC-type machines (for example programming devices with the Windows 95/98/2000 or NT operating systems. Since SCL uses the STEP 7 platform, you should also be familiar with working with the standard software described in the "Programming with STEP 7 V5.1" manual.

Scope of the Manual

The manual is valid for the S7-SCL V5.1 software package.

Documentation Packages for S7-SCL and the STEP 7 Standard Software

The following table provides you with an overview of the STEP 7 and SCL documentation:

Manuals	Purpose	Order Number
Basics of SCL and reference: <ul style="list-style-type: none"> S7-SCL for S7-300/400, Programming Blocks 	Basic and reference information explaining how to create a program, the structure of user programs and the individual language elements.	6ES7811-1CC04-8XA0
Basics of STEP 7: <ul style="list-style-type: none"> Getting Started and Exercises with STEP 7 V5.1 Programming with STEP 7 V5.1 Configuring Hardware and Connections with STEP 7 V5.1 Converting from S5 to S7 	The basics for technical personnel describing how to implement control tasks with STEP 7 and S7-300/400.	6ES7810-4CA05-8AA0
STEP 7 reference: <ul style="list-style-type: none"> LAD/FBD/STL manuals for S7-300/400 Standard and System Functions for S7-300/400 	Reference work describing the LAD, FBD and STL programming languages as well as standard and system functions as a supplement to the STEP 7 basics.	6ES7810-4CA05-8AR0

Online Help	Purpose	Order Number
Help on S7-SCL	Basics and reference for S7-SCL as online help	Part of the S7-SCL software package
Help on STEP 7	Basics on programming and configuring hardware with STEP 7 as online help	Part of the STEP 7 software package
Reference help on STL/LAD/FBD Reference help on SFBs/SFCs Reference help on organization blocks Reference help on IEC functions Reference help on system attributes	Context-sensitive reference	Part of the STEP 7 software package

Online Help

In addition to the manual, the online help integrated in the software provides you with detailed support when working with the software.

help system is integrated in the software with several interfaces:

- The **Help** menu provides numerous menu commands: **Contents** opens the contents of the SCL help system. **Introduction** provides an overview of programming with SCL. **Using Help** provides detailed instructions on working with the online help system.
- The context-sensitive help system provides information about the current context, for example help on an open dialog box or active window. This can be displayed by clicking the "Help" button or pressing the F1 key.
- The status bar is another form of context-sensitive help. A brief explanation of each menu command is displayed here when you position the mouse pointer on a menu command.
- A brief explanation of the buttons in the toolbar is also displayed if you position the mouse pointer briefly over a button.

If you prefer to have a printout of the information in the online help system, you can print individual topics, books or the entire help system.

This manual has the same content as the HTML help system of SCL. Since the manual and online help have the same structure, you can change easily between manual and online help.

SIMATIC Documentation on the Internet/Intranet

You will also find further information on the SIMATIC documentation on the Internet or SIEMENS Intranet.

- You will find up-to-date downloads of the documentation
 - on the **Internet** at http://www.ad.siemens.de/meta/html_00/support.shtml. Use the Knowledge Manager to find the documentation you require.
- You can send questions on the SIMATIC documentation to the following address. You will receive answers to your problems quickly.
 - On the **Internet** at <http://www4a.ad.siemens.de:8090/~SIMATIC/login>
- Or visit the home page of the SIMATIC documentation. Here you can find out about new products and innovations, send questions about the documentation and let us know if you have requests, suggestions, criticism or praise.
 - On the **Siemens Intranet** at http://intra1.khe.siemens.de/e8_doku/index.htm

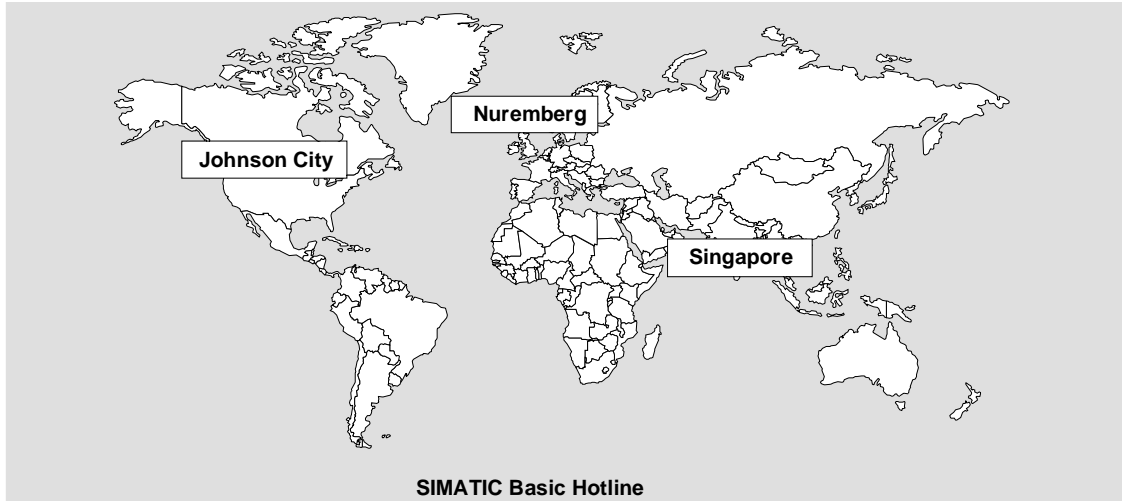
SIMATIC Training Center

To help you get to know the SIMATIC S7 automation system quickly, we offer various courses. Please contact your regional training center or the central training center in D 90327 Nuremberg, Germany.

Phone: +49 (911) 895-3200.

SIMATIC Customer Support Hotline

Available round the clock worldwide:



Worldwide (Nuremberg) Technical Support	Worldwide (Nuremberg) Technical Support	
(Free Contact) Local time: Mo.-Fr. 7:00 to 17:00 Phone: +49 (180) 5050 222 Fax: +49 (180) 5050 223 E-mail: techsupport@ ad.siemens.de GMT: +1:00	(charged, only with SIMATIC Card) Local time: Mo.-Fr. 0:00 to 24:00 Phone: +49 (911) 895-7777 Fax: +49 (911) 895-7001 GMT: +01:00	
Europe / Africa (Nuremberg) Authorization	America (Johnson City) Technical Support and Authorization	Asia / Australia (Singapore) Technical Support and Authorization
Local time: Mo.-Fr. 7:00 to 17:00 Phone: +49 (911) 895-7200 Fax: +49 (911) 895-7201 E-mail: authorization@ nbgm.siemens.de GMT: +1:00	Local time: Mo.-Fr. 8:00 to 19:00 Phone: +1 423 461-2522 Fax: +1 423 461-2289 E-mail: simatic.hotline@ sea.siemens.com GMT: -5:00	Local time: Mo.-Fr. 8:30 to 17:30 Phone: +65 740-7000 Fax: +65 740-7001 E-mail: simatic.hotline@ sae.siemens.com.sg GMT: +8:00

German and English are spoken on all the SIMATIC hotlines, French, Italian and Spanish are also spoken on the authorization hotline.

SIMATIC Customer Support Online Services

In its online services, SIMATIC Customer Support provides you with a wide range of additional information on SIMATIC products:

- You can obtain general up-to-the-minute information
 - on the Internet at <http://www.ad.siemens.de/simatic>
- Current product information bulletins and useful downloads:
 - on the **Internet** at <http://www.ad.siemens.de/simatic-cs>
 - From the **Bulletin Board System** (BBS) in Nuremberg (*SIMATIC Customer Support Mailbox*) at +49 (911) 895-7100.

To contact the mailbox, use a modem with up to V.34 (28.8 Kbauds) with the following parameter settings: 8, N, 1, ANSI, or dial via ISDN (x.75, 64 Kbps).

- You will find your local contact for Automation & Drives in our contacts database:
 - on the **Internet** at <http://www3.ad.siemens.de/partner/search.asp>

Contents

1 Product Overview and Installation

1.1	Overview of S7-SCL.....	1-1
1.2	What are the Advantages of S7-SCL?	1-3
1.3	Characteristics of the Development Environment.....	1-4
1.4	What's New in Version V5.1?	1-7
1.5	Installation and Authorization.....	1-9
1.6	Notes on Compatibility with DIN EN 61131-3.....	1-11

2 Designing an SCL Program

2.1	Welcome to "Measured Value Acquisition" - A Sample Program for First-Time Users	2-1
2.2	Task.....	2-2
2.3	Design of a Structured SCL Program.....	2-4
2.4	Defining the Subtasks	2-6
2.5	Defining the Interfaces Between Blocks.....	2-7
2.6	Defining the Input/Output Interface	2-10
2.7	Defining the Order of the Blocks in the Source File	2-11
2.8	Defining Symbols	2-12
2.9	Creating the SQUARE Function	2-13
2.9.1	Statement Section of the SQUARE Function	2-13
2.10	Creating the EVALUATE Function Block	2-14
2.10.1	Flow Chart for EVALUATE	2-14
2.10.2	Declaration Section of FB EVALUATE.....	2-15
2.10.3	Statement Section of FB EVALUATE	2-16
2.11	Creating the ACQUIRE Function Block.....	2-18
2.11.1	Flow Chart for ACQUIRE.....	2-18
2.11.2	Declaration Section of FB ACQUIRE	2-19
2.11.3	Statement Section of FB ACQUIRE.....	2-21
2.12	Creating the CYCLE Organization Block.....	2-24
2.13	Test Data	2-26

3 Using SCL

3.1	Starting the SCL Program	3-1
3.2	User Interface	3-2
3.3	Customizing the User Interface	3-3
3.4	Creating and Handling an SCL Source File.....	3-4
3.4.1	Creating a New SCL Source File	3-4
3.4.2	Opening an SCL Source File	3-5
3.4.3	Opening Blocks.....	3-6
3.4.4	Closing an SCL Source File	3-6
3.4.5	Specifying Object Properties	3-6
3.4.6	Creating Source Files with a Standard Editor.....	3-7
3.4.7	Block Protection	3-7

3.5	Guidelines for SCL Source Files.....	3-8
3.5.1	General Rules for SCL Source Files.....	3-8
3.5.2	Order of the Blocks	3-8
3.5.3	Using Symbolic Addresses.....	3-9
3.6	Editing in SCL Source Files.....	3-9
3.6.1	Undoing the Last Editing Action.....	3-9
3.6.2	Redoing an Editing Action	3-9
3.6.3	Finding and Replacing Text Objects.....	3-10
3.6.4	Selecting Text Objects	3-10
3.6.5	Copying Text Objects.....	3-10
3.6.6	Cutting Text Objects.....	3-11
3.6.7	Deleting Text Objects.....	3-11
3.6.8	Positioning the Cursor in a Specific Line.....	3-11
3.6.9	Syntactically Correct Indenting of Lines	3-12
3.6.10	Setting the Font Style and Color.....	3-12
3.6.11	Inserting Templates.....	3-13
3.7	Compiling an SCL Program.....	3-15
3.7.1	What You Should Know About Compiling.....	3-15
3.7.2	Customizing the Compiler	3-15
3.7.3	Compiling the Program	3-17
3.7.4	Creating a Compilation Control File.....	3-17
3.7.5	Debugging the Program After Compilation.....	3-18
3.8	Saving and Printing an SCL Source File.....	3-19
3.8.1	Saving an SCL Source File	3-19
3.8.2	Customizing the Page Format	3-19
3.8.3	Printing an SCL Source File	3-19
3.8.4	Setting the Print Options	3-20
3.9	Downloading the Created Programs.....	3-21
3.9.1	CPU Memory Reset	3-21
3.9.2	Downloading User Programs to the CPU.....	3-21
3.10	Debugging the Created Programs	3-23
3.10.1	The SCL Debugging Functions.....	3-23
3.10.2	The "Monitor" Debugging Function.....	3-24
3.10.3	Debugging with Breakpoints/Single Step Mode"	3-25
3.10.4	Steps in Monitoring	3-26
3.10.5	Steps for Debugging with Breakpoints.....	3-27
3.10.6	Using the STEP 7 Debugging Functions.....	3-29
3.11	Displaying and Modifying CPU Properties	3-31
3.11.1	Displaying and Modifying the CPU Operating Mode	3-31
3.11.2	Displaying and Setting the Date and Time on the CPU.....	3-31
3.11.3	Reading Out CPU Data.....	3-32
3.11.4	Reading Out the Diagnostic Buffer of the CPU	3-32
3.11.5	Displaying/Compressing the User Memory of the CPU	3-32
3.11.6	Displaying the Cycle Time of the CPU.....	3-33
3.11.7	Displaying the Time System of the CPU.....	3-33
3.11.8	Displaying the Blocks on the CPU	3-33
3.11.9	Displaying Information about Communication with the CPU.....	3-34
3.11.10	Displaying the Stacks of the CPU.....	3-34

4 Basic SCL Terms

4.1	Interpreting the Syntax Diagrams	4-1
4.2	Character Set.....	4-3
4.3	Reserved Words	4-4
4.4	Identifiers	4-5
4.5	Standard Identifiers	4-6
4.6	Block Identifier	4-6
4.7	Address Identifier	4-7
4.8	Timer Identifier	4-9
4.9	Counter Identifier	4-9
4.10	Numbers	4-10
4.11	Character Strings	4-12
4.12	Symbol.....	4-13
4.13	Comment Section.....	4-13
4.14	Line Comment.....	4-14
4.15	Variables.....	4-15

5 SCL Program Structure

5.1	Blocks in SCL Source Files	5-1
5.2	Order of the Blocks	5-2
5.3	General Structure of a Block.....	5-3
5.4	Block Start and End	5-3
5.5	Attributes for Blocks	5-5
5.6	Block Comment.....	5-7
5.7	System Attributes for Blocks.....	5-8
5.8	Declaration Section	5-9
5.9	System Attributes for Parameters	5-10
5.10	Statement Section.....	5-11
5.11	Statements.....	5-12
5.12	Structure of a Function Block (FB)	5-13
5.13	Structure of a Function (FC)	5-15
5.14	Structure of an Organization Block (OB)	5-17
5.15	Structure of a Data Block (DB)	5-18
5.16	Structure of a User-Defined Data Type.....	5-21

6 Data Types

6.1	Overview of the Data Types in SCL	6-1
6.2	Elementary Data Types	6-3
6.2.1	Bit Data Types	6-3
6.2.2	Character Types.....	6-3
6.2.3	Numeric Data Types.....	6-3
6.2.4	Time Types	6-4
6.3	Complex Data Types.....	6-5
6.3.1	DATE_AND_TIME Data Type	6-5
6.3.2	STRING Data Type	6-7
6.3.3	ARRAY Data Type	6-9
6.3.4	STRUCT Data Type	6-11
6.4	User-Defined Data Types	6-13
6.4.1	User-Defined Data Types (UDT).....	6-13
6.5	Data Types for Parameters.....	6-15
6.5.1	Data Types for Parameters.....	6-15
6.5.2	TIMER and COUNTER Data Types.....	6-15
6.5.3	BLOCK Data Types.....	6-16
6.5.4	POINTER Data Type.....	6-16

6.6	ANY Data Type.....	6-18
6.6.1	Example of the ANY Data Type.....	6-19
7	Declaring Local Variables and Parameters	
7.1	Local Variables and Block Parameters	7-1
7.2	General Syntax of a Variable or Parameter Declaration.....	7-3
7.3	Initialization.....	7-4
7.4	Declaring Views of Variable Ranges.....	7-6
7.5	Using Multiple Instances	7-8
7.6	Instance Declaration	7-8
7.7	Flags (OK Flag)	7-9
7.8	Declaration Subsections.....	7-10
7.8.1	Overview of the Declaration Subsections.....	7-10
7.8.2	Static Variables.....	7-11
7.8.3	Temporary Variables.....	7-12
7.8.4	Block Parameters.....	7-13
8	Declaring Constants and Jump Labels	
8.1	Constants	8-1
8.1.1	Declaring Symbolic Names for Constants.....	8-2
8.1.2	Data Types for Constants.....	8-3
8.1.3	Notation for Constants.....	8-4
8.2	Declaring Labels	8-17
8.2.1	Declaring Labels	8-17
9	Shared Data	
9.1	Overview of Shared Data	9-1
9.2	Memory Areas of the CPU	9-2
9.2.1	Overview of the Memory Areas of the CPU	9-2
9.2.2	Absolute Access to Memory Areas of the CPU	9-3
9.2.3	Symbolic Access to Memory Areas of the CPU	9-5
9.2.4	Indexed Access to Memory Areas of the CPU	9-6
9.3	Data Blocks.....	9-7
9.3.1	Overview of Data Blocks	9-7
9.3.2	Absolute Access to Data Blocks	9-8
9.3.3	Indexed Access to Data Blocks	9-10
9.3.4	Structured Access to Data Blocks.....	9-11
10	Expressions, Operations and Addresses	
10.1	Overview of Expressions, Operations and Addresses.....	10-1
10.2	Operations.....	10-2
10.3	Addresses.....	10-3
10.4	Syntax of an Expression.....	10-5
10.5	Simple Expression	10-7
10.6	Arithmetic Expressions.....	10-8
10.7	Logical Expressions	10-10
10.8	Comparison Expressions.....	10-12
11	Statements	
11.1	Value Assignments	11-1
11.1.1	Value Assignments with Variables of an Elementary Data Type	11-2
11.1.2	Value Assignments with Variables of the Type STRUCT and UDT	11-3
11.1.3	Value Assignments with Variables of the Type ARRAY	11-5
11.1.4	Value Assignments with Variables of the Data Type STRING	11-7
11.1.5	Value Assignments with Variables of the Type DATE_AND_TIME.....	11-8

11.1.6	Value Assignments with Absolute Variables for Memory Areas.....	11-9
11.1.7	Value Assignments with Shared Variables.....	11-10
11.2	Control Statements	11-12
11.2.1	Overview of Control Statements	11-12
11.2.2	Conditions.....	11-13
11.2.3	IF Statements.....	11-14
11.2.4	CASE Statement.....	11-16
11.2.5	FOR Statement.....	11-18
11.2.6	WHILE Statement	11-21
11.2.7	REPEAT Statement	11-22
11.2.8	CONTINUE Statement	11-23
11.2.9	EXIT Statement.....	11-24
11.2.10	GOTO Statement	11-25
11.2.11	RETURN Statement.....	11-26
11.3	Calling Functions and Function Blocks	11-27
11.3.1	Call and Parameter Transfer	11-27
11.3.2	Calling Function Blocks	11-28
11.3.3	Calling Functions.....	11-36
11.3.4	Implicitly Defined Parameters.....	11-42
12	Counters and Timers	
12.1	Counters	12-1
12.1.1	Counter Functions.....	12-1
12.1.2	Calling Counter Functions	12-1
12.1.3	Supplying Parameters for Counter Functions.....	12-3
12.1.4	Input and Evaluation of the Counter Value.....	12-4
12.1.5	Count Up (S_CU).....	12-5
12.1.6	Count Down (S_CD).....	12-5
12.1.7	Count Up/Down (S_CUD).....	12-6
12.1.8	Example of Counter Functions.....	12-7
12.2	Timers.....	12-8
12.2.1	Timer Functions	12-8
12.2.2	Calling Timer Functions.....	12-8
12.2.3	Supplying Parameters for Timer Functions	12-10
12.2.4	Input and Evaluation of a Time Value	12-12
12.2.5	Start Timer as Pulse Timer (S_PULSE)	12-14
12.2.6	Start Timer as Extended Pulse Timer (S_PEXT).....	12-15
12.2.7	Start Timer as On-Delay Timer (S_ODT)	12-16
12.2.8	Start Timer as Retentive On-Delay Timer (S_ODTS)	12-17
12.2.9	Start Timer as Off-Delay Timer (S_OFFDT).....	12-18
12.2.10	Example of Timer Functions.....	12-19
12.2.11	Selecting the Right Timer	12-20
13	SCL Standard Functions	
13.1	Data Type Conversion Functions.....	13-1
13.1.1	Converting Data Types.....	13-1
13.1.2	Implicit Data Type Conversion.....	13-2
13.1.3	Standard Functions for Explicit Data Type Conversion.....	13-4
13.2	Numeric Standard Functions	13-9
13.2.1	General Arithmetic Standard Functions	13-9
13.2.2	Logarithmic Functions	13-9
13.2.3	Trigonometric Functions.....	13-10
13.2.4	Examples of Numeric Standard Functions	13-10
13.3	Bit String Standard Functions.....	13-11
13.3.1	Examples of Bit String Standard Functions.....	13-12

13.4	Functions for Processing Character Strings	13-13
13.4.1	Functions for String Manipulation	13-13
13.4.2	Functions for Comparing Strings	13-17
13.4.3	Functions for Converting the Data Format	13-18
13.4.4	Example of Processing Character Strings.....	13-20
13.5	SFCs, SFBs and Standard Library	13-22
13.5.1	Transfer Interface to OBs	13-24
14	Language Definition	
14.1	Formal Language Definition	14-1
14.1.1	Overview of Syntax Diagrams	14-1
14.1.2	Rules	14-2
14.1.3	Terms Used in the Lexical Rules	14-4
14.1.4	Formatting Characters, Separators and Operations.....	14-6
14.1.5	Keywords and Predefined Identifiers	14-9
14.1.6	Address Identifiers and Block Keywords.....	14-12
14.1.7	Overview of Non Terms.....	14-13
14.1.8	Overview of Tokens	14-14
14.1.9	Identifiers.....	14-14
14.1.10	Assigning Names in SCL.....	14-16
14.1.11	Predefined Constants and Flags	14-18
14.2	Lexical Rules	14-19
14.2.1	Identifiers.....	14-19
14.2.2	Constants	14-21
14.2.3	Absolute Addressing	14-25
14.2.4	Comments	14-27
14.2.5	Block Attributes.....	14-28
14.3	Syntax Rules.....	14-29
14.3.1	Structure of SCL Source Files	14-29
14.3.2	Structure of the Declaration Sections.....	14-31
14.3.3	Data Types in SCL.....	14-35
14.3.4	Statement Section.....	14-37
14.3.5	Value Assignments	14-39
14.3.6	Calling Functions and Function Blocks	14-41
14.3.7	Control Statements	14-43
15	Tips and Tricks	
	Glossary	
	Index	

1 Product Overview and Installation

1.1 Overview of S7-SCL

Area of Application

Apart from their traditional control tasks, today's programmable controllers are increasingly expected to handle 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 DIN EN 61131-3.

SCL not only assists you with "normal" control tasks but also with extensive applications making it superior to the "traditional" programming languages in the following areas:

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

S7-SCL Programming Language

SCL (*Structured Control Language*) is a higher-level programming language oriented on PASCAL. It is based on a standard for PLCs (programmable logic controllers).

The DIN EN-61131-3 standard (int. IEC 1131-3) standardizes the programming languages for programmable logic controllers. The SCL programming language complies with the PLCopen Basis Level of the ST (structured text) language defined in this standard. In the NORM_TAB.WRI file, you will find an exact definition of the standard compliance according to the DIN EN-61131-3 standard.

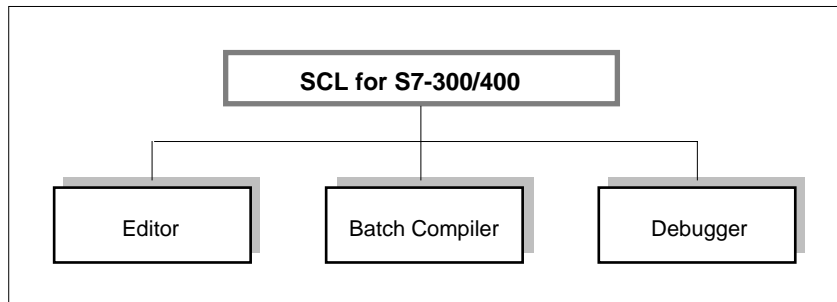
In addition to high-level language elements, SCL also includes language elements typical of PLCs such as inputs, outputs, timers, bit memory, block calls, etc. In other words, SCL complements and extends the STEP 7 programming software and its programming languages Ladder Logic, Function Block Diagram,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

Development Environment

For optimum use and practical application of SCL, there is a powerful development environment that 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). Programmers are supported in their tasks by powerful functions.
- A **Batch Compiler** to compile the edited program into MC7 machine code. The MC7 code generated will run on all S7-300/400 CPUs from CPU 314 upwards.
- A **Debugger** to search for logical programming errors in the compiled program. You debug at the source language level.

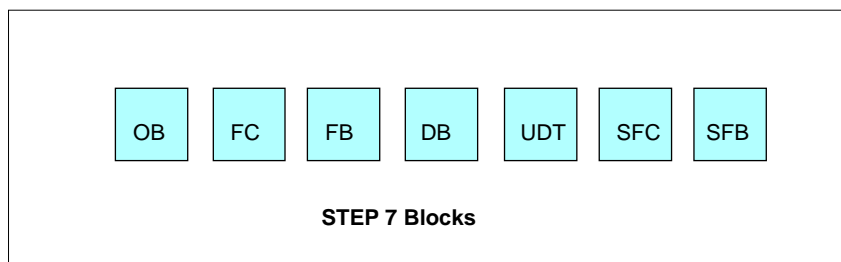
The individual components are simple and convenient to use since they run under Windows and make use of all the advantages of this operating system.



1.2 What are the Advantages of S7-SCL?

SCL offers you all the advantages of a high-level programming language. SCL also has a number of characteristics specifically designed to support structured programming, such as:

- SCL supports the block concept of STEP 7 and therefore allows standardized programming of blocks just as with Statement List (STL), Ladder Logic (LAD), and Function Block Diagram (FBD).

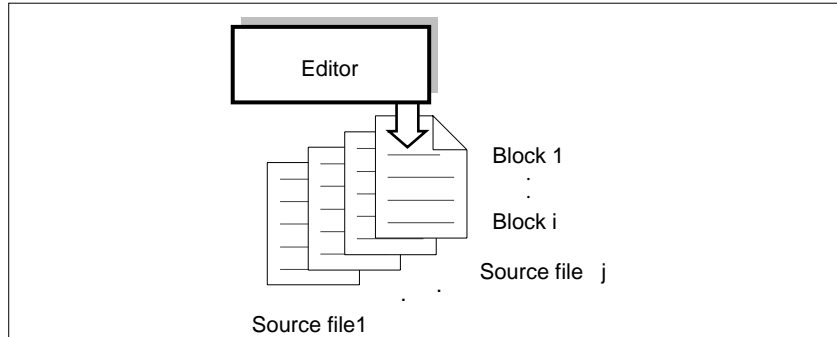


- You do not need to create every function yourself, but can use ready-made blocks such as system functions or system function blocks that already exist in the operating system of the CPU.
- You can use blocks programmed using SCL in combination with blocks programmed in Statement List (STL), Ladder Logic (LAD), and Function Block Diagram (FBD). 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 in STL, LAD, or FBD programs. The programming languages of STEP 7 and SCL (optional package) therefore complement one another perfectly.
- Source objects you create with SCL for STEP 5 are upwards compatible with one or two minor exceptions; in other words these programs can also be edited, compiled and debugged with S7 SCL.
- SCL blocks can be decompiled into the STEP 7 Statement List (STL) programming language. Recompilation from STL to SCL is not possible.
- With some experience of high-level programming languages, SCL can be learned quickly.
- When creating programs, the programmer is supported by powerful functions for processing the source text.
- When you compile your edited program, the blocks are created and can be executed on all the CPUs of the S7 300/400 programmable controllers with a CPU 314 or higher.
- With the SCL test and debugging functions, you can search for logical programming errors in the compiled program. You debug at the source language level.

1.3 Characteristics of the Development Environment

Editor

The SCL Editor is a text editor that can be used for editing any text files. Its main purpose is the creation and editing of source files for STEP 7 programs. You can program one or more blocks in a source file. The Editor does not check the syntax of text while it is being entered.

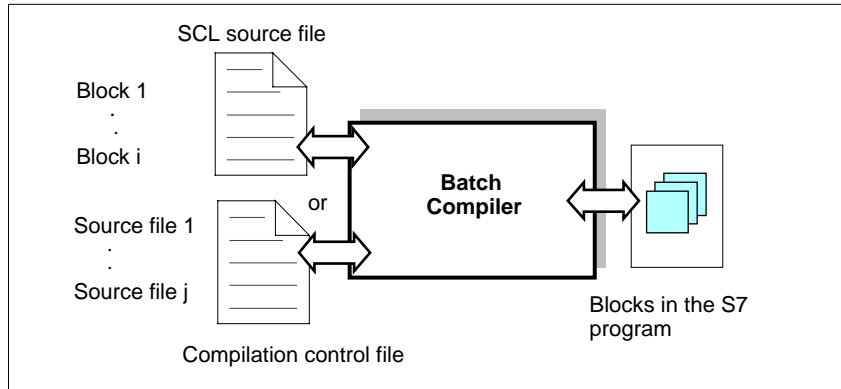


With the SCL Editor, you can:

- 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 that simplify the task of editing the source file, for example, search and replace.
- Customize the editor settings to meet your requirements, for example, by syntactically correct coloring of the various language elements.

Compiler

Once you have created your source files using the SCL Editor, you compile them into MC7 code.

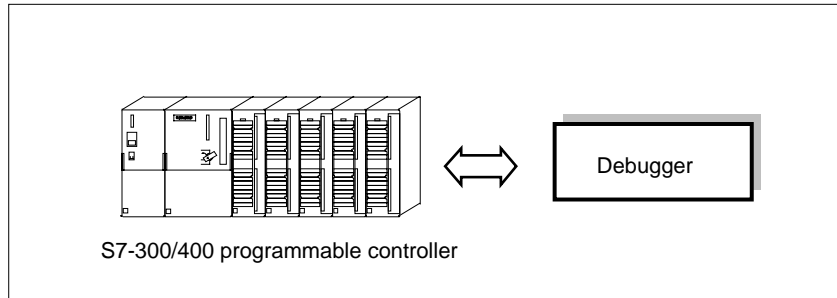


The SCL Compiler allows you to:

- Compile an SCL source file consisting of a number of blocks in a single compilation.
- Compile a series of SCL source files using a compilation control file containing the names of the source files.
- Compile selected blocks from a source file.
- Customize the compiler settings to suit your specific requirements.
- View all errors and warning messages that occur during the compilation process.
- Localize errors in the source file, if required with descriptions of the errors and instructions on rectifying them.

Debugger

With the SCL Debugger, you can check how a program will run on the PLC and identify any logical errors.



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 statement at a time and observe how the variable values change in a results window;
- **Continuous** monitoring - in this mode you can test out a group of statements within a block of the source file. During the test, the values of the variables and parameters are displayed in chronological sequence and (where possible) cyclically updated.

1.4 What's New in Version V5.1?

DIN EN 61131-1

From Version 5.0 onwards, S7-SCL complies with the PLCopen Basis Level for ST (structured text) of the DIN EN 61131-3 standard (previously IEC 1131-3).

Extended Language Range

- **Dynamic timer and counter calls**
When a timer or counter function is called, you can specify a variable of the INT data type instead of the absolute number. Each time the program is executed, you can assign a different number to these variables and make the function call dynamic.
- **Type-defined notation of constants**
Previously, a constant was given its data type only with the arithmetic or logical operation in which it was used. In the following statement, for example, '12345' is given the data type INT:
`Int1:=Int2 + 12345`

You can now assign data types to constants explicitly by using a type-defined notation for the constant as shown below:
`INT#12345`

- **Several views of a variable**
To access a declared variable with a different data type, you can define views of the variable or of areas within the variable.

Improved Editor Functions

- **Syntactically correct colors and styles**
Using different styles and colors for the various language elements lends your SCL source files a professional appearance.
- **Syntactically correct formatting of source files**
Automatic line indents increase the legibility of the SCL source files.
- **Undoing and redoing input step by step**
Using a menu command, several editing steps can be undone or redone.

Extended Print Functionality

- Selection of different fonts and styles for your printout
When you print out your SCL source file, you can select the styles that differ from those shown on screen.
- Printing with line numbers
You can also print out line numbers or insert a form feed before the start of each new block.

Selective Compilation and Downloading

- Selective compilation
With the "Compile Selected Blocks" function, you can compile individual blocks from an SCL source file so that you can make changes in the program more quickly.
- Selective Downloads
With the "Download Changes" function, you can download selected blocks of a source file.

1.5 Installation and Authorization

System Requirements

The S7-SCL V5.1 optional package can run on a programming device/PC with the following:

- Microsoft Windows 95/98/2000/NT operating system
- STEP 7 V5 standard package, service pack 3 or higher (any corrected versions of the standard package that are necessary are supplied).

Hardware Requirements

The requirements for S7-SCL are the same as those for the STEP 7 standard package. The extra hard disk space required by the S7-SCL V5.1 optional package can be found in the readme.wri file.

Starting the Installation Program

S7-SCL includes a Setup program that automatically installs the software. On-screen prompts that appear on the screen guide you step by step through the complete installation process.

Follow the steps outlined below:

1. Open the Control Panel in the Windows 95/98/2000/NT and double-click on the Add/Remove Programs icon.
2. Select Install...
3. Insert the CD and click "Next". Windows then automatically searches for the installation program "Setup.exe".
4. Follow the instructions displayed by the installation program.

Notes on Authorization

When you install the program, setup checks whether you have the authorization required to use the S7-SCL programming software on your hard disk. If no authorization is found, a message appears telling you that the software can only be used with the authorization. If you wish you can install the authorization at this point or continue the installation and install the authorization later. To install the authorization during installation, simply insert the authorization diskette when the prompt is displayed.

Authorization Diskette

To install the authorization you require the authorization diskette (note that authorizations cannot be copied with normal copy functions). The actual authorization itself is on this diskette. The "AuthorsW" program required for displaying, installing and uninstalling the authorization is on the same CD ROM as S7-SCL V5.1.

The number of possible user authorizations is specified by an authorization counter on the authorization diskette. Each time you install an authorization, the counter decrements by 1. Once it reaches zero, there are no more authorizations available on the diskette.



Caution

Read the instructions in the README.WRI file in the AuthorsW folder on the CD. If you do not keep to the instructions, the authorization may be irretrievably lost.

Loss of the Authorization

You could, for example, lose an authorization due to a defect on your hard disk that prevents you from uninstalling the authorization from the defective disk.

If you lose your authorization, you can fall back on the emergency authorization. This is also on the authorization diskette. With the emergency authorization, you can continue to use the software for a restricted period. In this case, when you start up, the time left until the validity expires is displayed. During this period, you should obtain a substitute for the lost authorization. To obtain a replacement for your authorization, please contact your SIEMENS representative.

Note

You will find further instructions and rules relating to installing and uninstalling software in the "Programming with STEP 7 V5.x" manual.

1.6 Notes on Compatibility with DIN EN 61131-3

From version 5.0 onwards, S7-SCL complies with the PLCopen Basis Level for ST (structured text) of the DIN EN 61131-3 standard (previously IEC 1131-3).

If you have an ST program, you can now import it as an ASCII file into the STEP 7 data management using the SIMATIC Manager or copy/insert it into the SCL editor.

Settings and Requirements

You require the following settings to create a system environment complying with the standard:

- Select the English mnemonics for the project in the SIMATIC Manager with **Options > Customize > Language**.
- In SCL, deselect the "Permit nested comments" option with **Options > Customize > Compiler**.
- Instead of the keywords FUNCTION_BLOCK / END_FUNCTION_BLOCK, the keywords PROGRAM / END_PROGRAM are also permitted.
- The name of the PROGRAM / FUNCTION_BLOCK or FUNCTION must be assigned a unique number in the symbol table.

Changes in the Syntax and Semantics

As a result of the compliance with the standard, the following changes have been made in the syntax and semantics of the SCL language Version 5.0:

- Symbols are no longer case-sensitive. For symbols from the symbol table, this applies from STEP 7 V4.02 onwards.
- The lines END_VAR, END_CONST, END_LABEL, and FUNCTION_BLOCK name, FUNCTION name etc. must not be completed by a semicolon. A semicolon is interpreted as an "empty" statement so that all following constructs are evaluated as statements.
- Value lists in the CASE statement no longer need to be sorted in ascending order. Only if you specify a range of values in the format "a .. b", then a <= b must be true.
- Addresses of the type INT or DINT are no longer automatically converted to the REAL data type in division (/). The data type of the result of division (/) is now determined by the data type of the most significant address. If, for example, two addresses of the data type INT are divided, the result is also of the data type INT (for example 10/3=3, whereas 10.0/3=3.33).

2 Designing an SCL Program

2.1 Welcome to "Measured Value Acquisition" - A Sample Program for First-Time Users

What You Will Learn

The sample program for first-time users shows you how to use SCL effectively. At first, you will probably have lots of questions, such as:

- How do I design a program written in SCL?
- Which SCL language functions are suitable for performing the task?
- What debugging functions are available?

These and other questions are answered in this section.

SCL language Elements Used

The sample program introduces the following SCL language functions:

- Structure and use of the various SCL block types
- Block calls with parameter passing and evaluation
- Various input and output formats
- Programming with elementary data types and arrays
- Initializing variables
- Program structures and the use of branches and loops

Required Hardware

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 so that you can test the program quickly using the switches on the input module and the displays on the output module. To run a thorough test, use the SCL debugging functions.

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

2.2 Task

Overview

Measured values will be acquired by an input module and then sorted and processed by an SCL program. The results will be displayed on an output module.

Acquire Measured Values

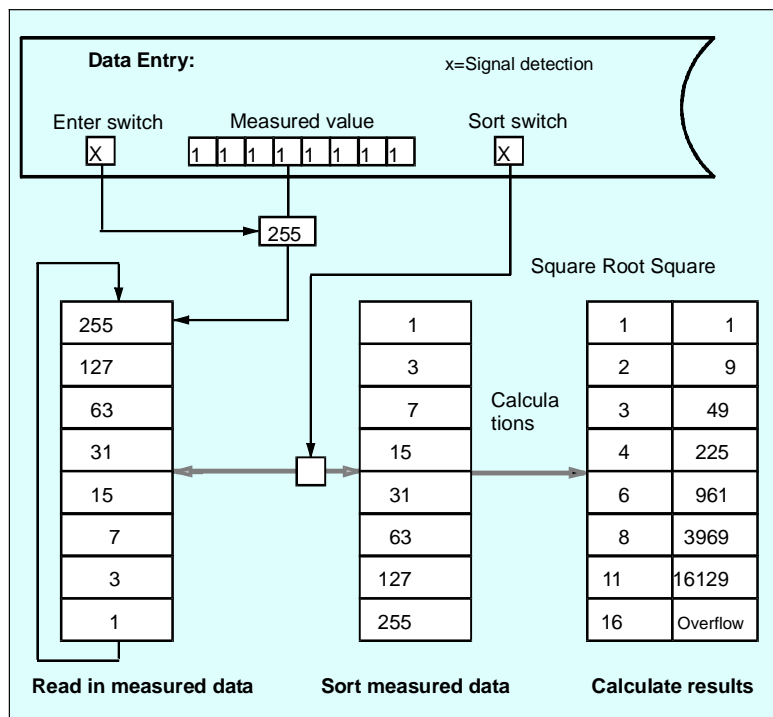
A measured value is set using the 8 input switches. This is then read into the measured value array in memory when an edge is detected at an input switch (see following diagram).

The range of the measured values is 0 to 255. One byte is therefore required for the input.

Processing Measured Values

The measured value array will be organized as a ring buffer with a maximum of eight entries.

When a signal is detected at the Sort switch, the values stored in the measured value array are arranged in ascending order. After that, the square root and the square of each number are calculated. One word is required for the processing functions.



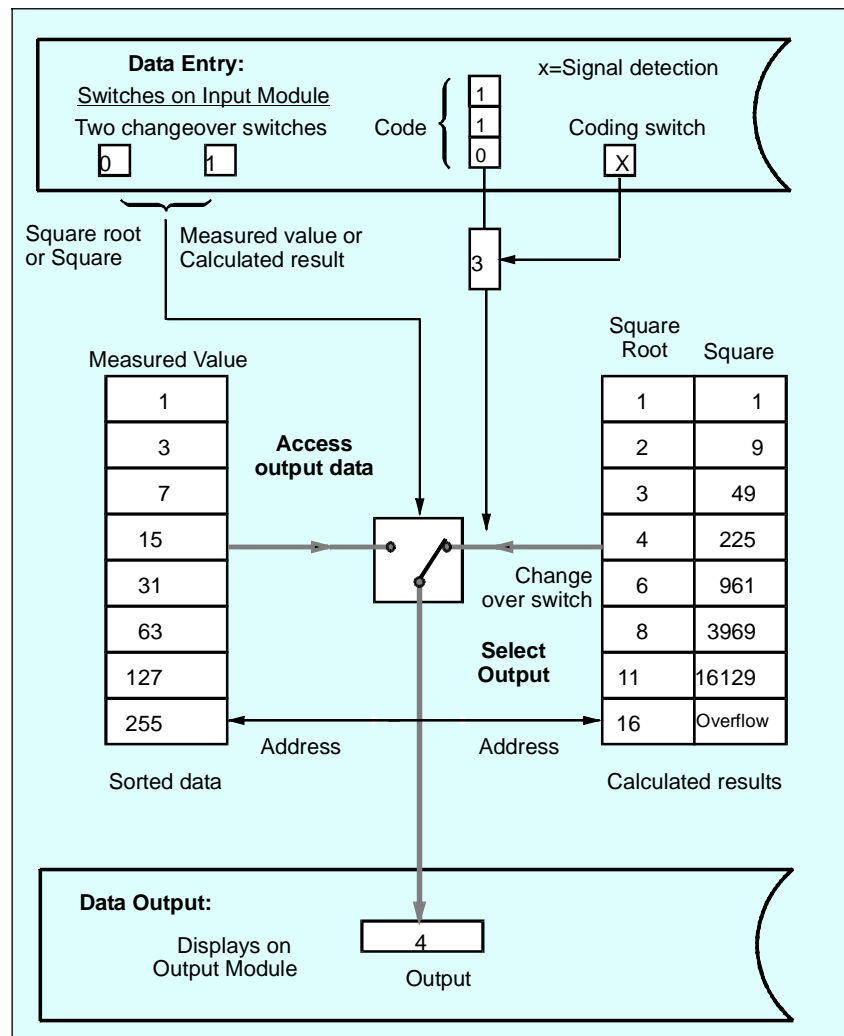
Selectable Outputs

Only one value can ever be displayed on the output module. The following selections can therefore be made:

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

The displayed value is selected as follows:

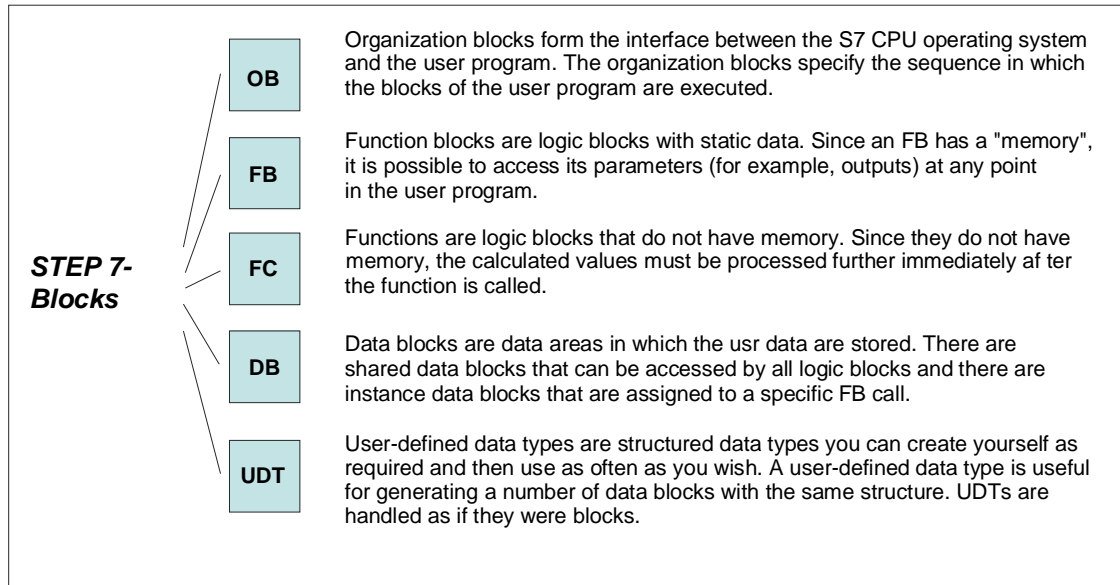
- Three switches are used to set a code that is copied if a signal is detected at a fourth switch, the Coding switch. From this, an address is calculated that is used to access the output.
- The same address identifies three values: the measured value, its square root and its square. To select one of these values, two selector switches are required.



2.3 Design of a Structured SCL Program

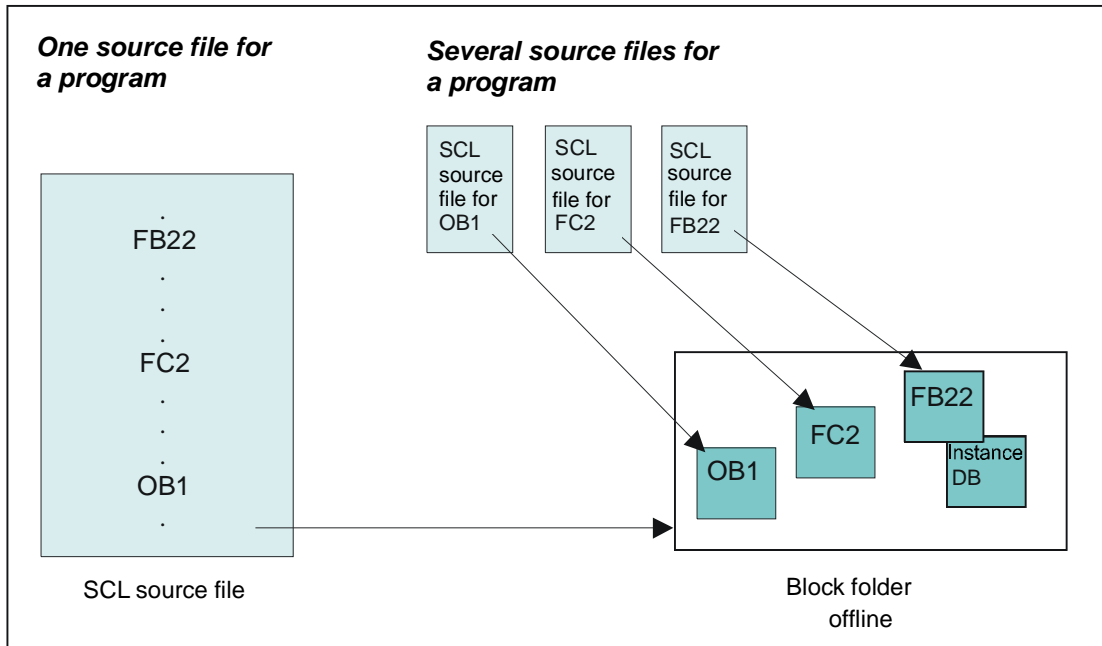
Block Types

The task defined above is best solved using a **structured SCL program**. This means using a modular design; in other words, the program is subdivided into a number of blocks, each responsible for a specific subtask. In SCL, as with the other programming languages in STEP 7, you have the following block types available.



Arrangement of Blocks in SCL Source Files

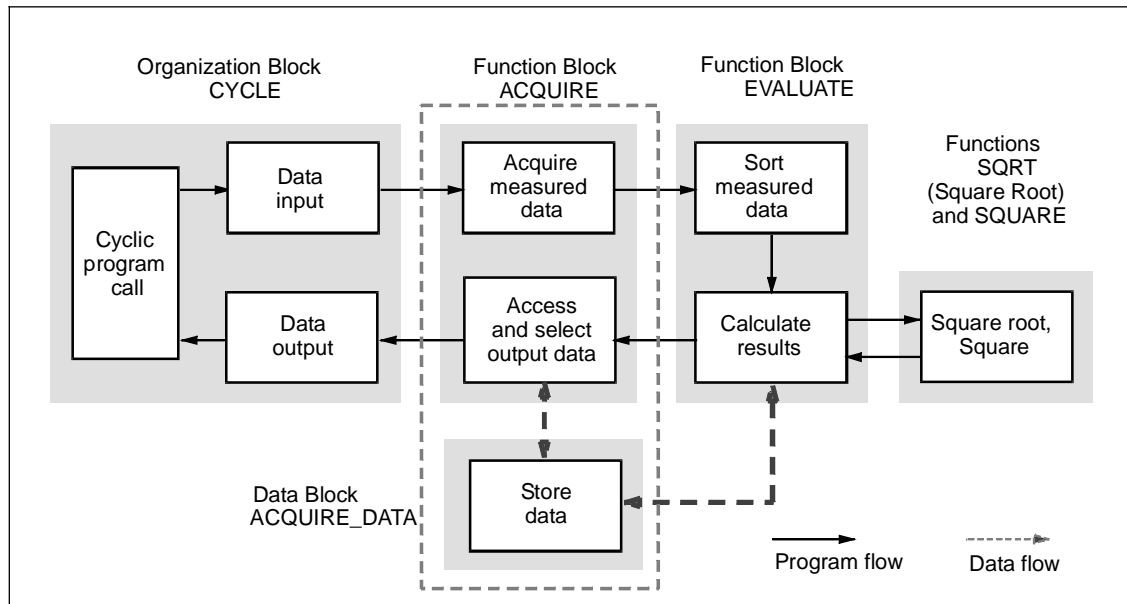
An SCL program consists of one or more SCL source files. A source file can contain a single block or a complete program consisting of various blocks.



2.4 Defining the Subtasks

Subtasks

The subtasks are shown in the figure below. The rectangular shaded areas represent the blocks. The arrangement of the logic blocks from left to right is also the order in which they are called.



Selecting and Assigning the Available Block Types

The individual blocks were selected according to the following criteria:

Function		Block Name
User programs can only be started in an OB. Since the measured values will be acquired cyclically, an OB for a <i>cyclic call</i> (OB1) is required. Part of the program - <i>data input</i> and <i>data output</i> - is programmed in the OB.	⇒	"Cycle" OB
The subtask " <i>acquire measured values</i> " requires a block with a memory; in other words, a function block (FB), since certain local block data (for example, the ring buffer) must be retained from one program cycle to the next. The location for <i>storing data</i> (memory) is the instance data block <code>ACQUIRE_DATA</code> . The same FB can also handle the <i>address and select output</i> subtask, since the data is available here.	⇒	"Acquire" FB

Function		Block Name
When selecting the type of block for the subtasks <i>sort measured values</i> and <i>calculate results</i> , remember that you need an output buffer containing the calculated results "square root" and "square" for each measured value. The only suitable block type is therefore an FB. Since this FB is called by an FB higher up in the call hierarchy, it does not require its own DB. Its instance data can be stored in the instance data block of the calling FB.	⇒	"Evaluate" FB
A function (FC) is best suited for the subtasks <i>calculate square root and square</i> since the result can be returned as a function value. Moreover, no data used in the calculation needs to be retained for more than one program cycle. The standard SCL function <code>SQRT</code> can be used to calculate the square root. A special function <code>SQUARE</code> will be created to calculate the square and this will also check that the value is within the permitted range.	⇒ ⇒	"SQRT" FC (square root) and "Square" FC

2.5 Defining the Interfaces Between Blocks

Overview

The interface of a block is formed by parameters that can be accessed by other blocks.

Parameters declared in the blocks are placeholders that have a value only when the block is actually used (called). These placeholders are known as formal parameters and the values assigned to them when the block is called are referred to as the actual parameters. 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 available for further processing. A function (FC) can pass on its result as a function value.

Block parameters can be subdivided into the categories shown below:

Block Parameter	Explanation	Declaration
Input parameters	Input parameters accept the actual input values when the block is called. They are read-only.	<code>VAR_INPUT</code>
Output parameters	Output parameters transfer the current output values to the calling block. Data can be written to and read from them.	<code>VAR_OUTPUT</code>
In/out parameters	In/out parameters accept the actual value of a variable when the block is called, process the value, and write the result back to the original variable.	<code>VAR_IN_OUT</code>

Cycle OB

The `CYCLE` OB has no formal parameters itself. It calls the `ACQUIRE` FB and passes the measured value and the control data for its formal parameters to it.

Acquire FB

Parameter Name	Data Type	Declaration Type	Description
measval_in	INT	VAR_INPUT	Measured value
newval	BOOL	VAR_INPUT	Switch for entering measured value in ring buffer
resort	BOOL	VAR_INPUT	Switch for sorting and evaluating measured data
funct_sel	BOOL	VAR_INPUT	Selector switch for square root or square
selection	WORD	VAR_INPUT	Code for selecting output value
newsel	BOOL	VAR_INPUT	Switch for reading in code
result_out	DWORD	VAR_OUTPUT	Output of calculated result
measval_out	DWORD	VAR_OUTPUT	Output of measured value

Evaluate

The `ACQUIRE` FB calls the `EVALUATE` FB. The data they share is the measured value array that require sorting. This array is therefore declared as an in/out parameter. A structured array is created as an output parameter for the calculated results Square Root and Square. The following table shows the formal parameters:

Name	Data Type	Declaration Type	Description
sortbuffer	ARRAY[..] OF REAL	VAR_IN_OUT	Measured value array, corresponds to ring buffer
calcbuffer	ARRAY[..]OF STRUCT	VAR_OUTPUT	Array for results: Structure with "square root" and "square" components of type INT

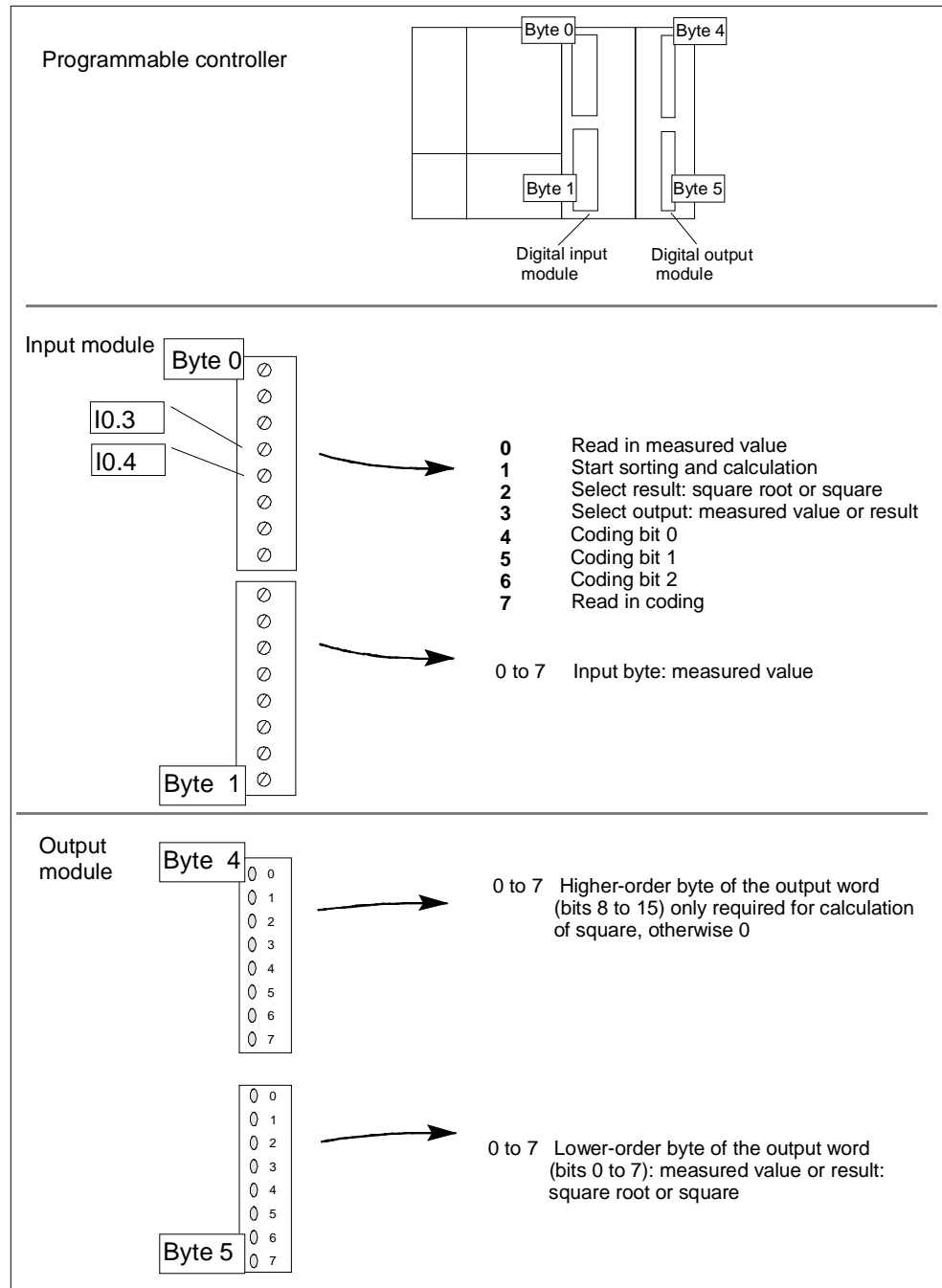
SQRT and Square

These functions are called by `EVALUATE`. They require an input value (argument) and return their results as a function value.

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

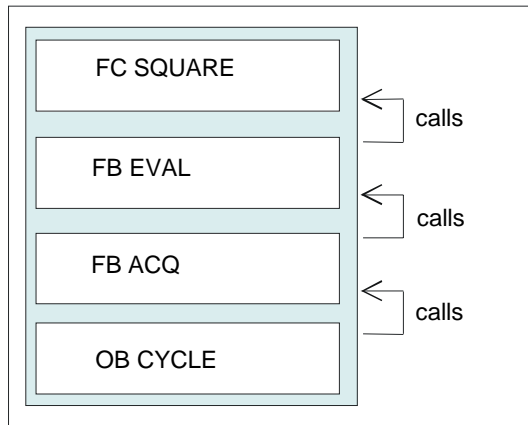
2.6 Defining the Input/Output Interface

The figure below shows the input/output interface. Note that when input/output is in bytes, the lower-order byte is at the top and the higher-order byte is at the bottom. If input/output is in words, on the other hand, the opposite is true.



2.7 Defining the Order of the Blocks in the Source File

When arranging the order of the blocks in the SCL source file, remember that a block must exist before you use it; in other words, before it is called by another block. This means that the blocks must be arranged in the SCL source file as shown below:



2.8 Defining Symbols

Using symbolic names for module addresses and blocks makes your program easier to follow. Before you can use these symbols, you must enter them in the symbol table.

The figure below shows the symbol table of the sample program. It describes the symbolic names that you declare in the symbol table so that the source file can be compiled free of errors:

	Symbol	Address	Data type	
1	Coding	IW 0	WORD	
2	Coding switch	I 0.7	BOOL	
3	CYCLE	OB 1	OB 1	
4	Entry	IB 1	BYTE	
5	EVALUATE	FB 20	FB 20	
6	Function switch	I 0.2	BOOL	
7	Input 0.0	I 0.0	BOOL	
8	Output	QW 4	INT	
9	Output switch	I 0.3	BOOL	
10	ACQUIRE	FB 10	FB 10	
11	ACQUIRE_DATA	DB 10	FB 10	
12	Sorting switch	I 0.1	BOOL	
13	SQUARE	FC 41	FC 41	

2.9 Creating the SQUARE Function

2.9.1 Statement Section of the SQUARE Function

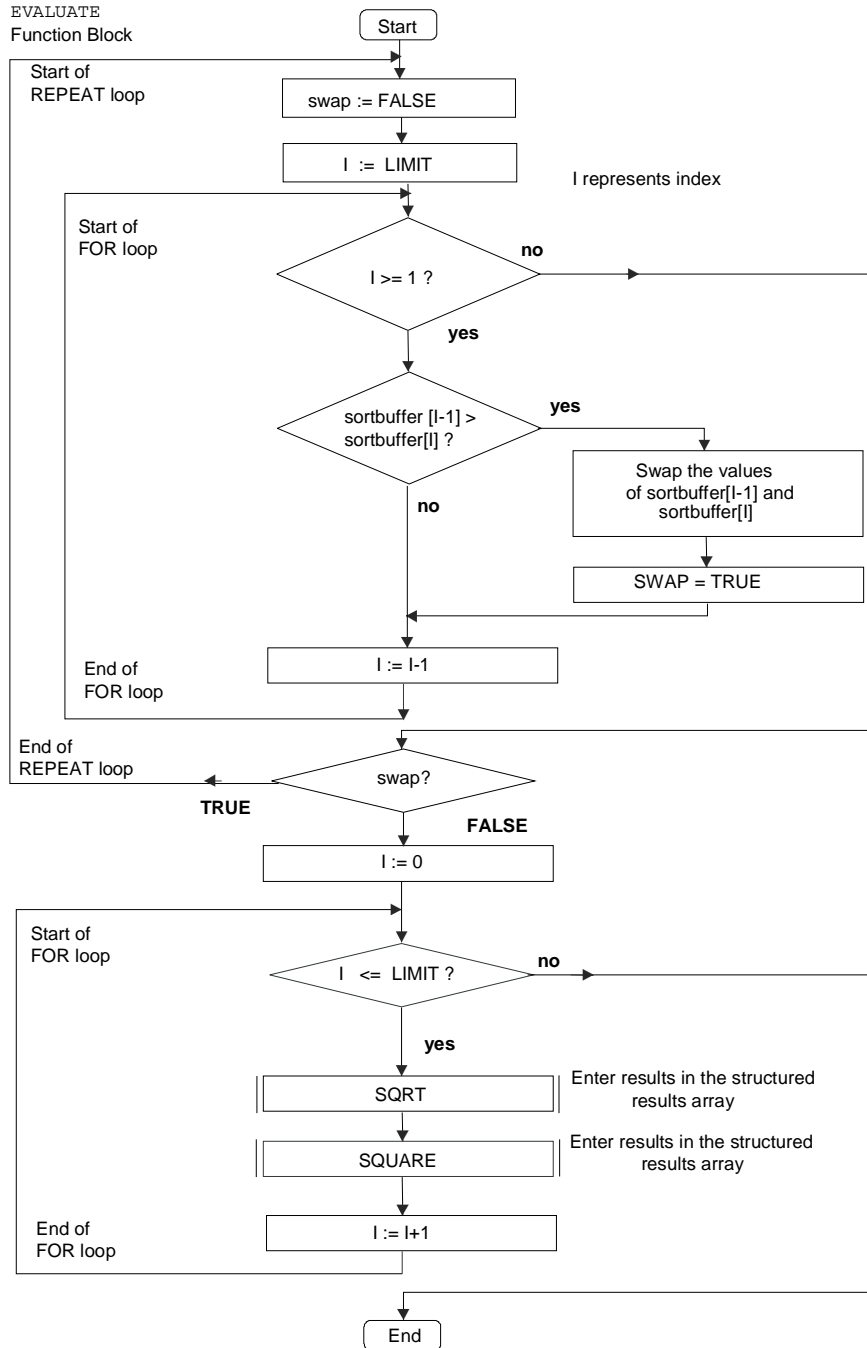
Statement Section

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

```
FUNCTION SQUARE : INT
(*****
This function returns as its function value the square of the
input value or if there is overflow, the maximum value that
can be represented as an integer.
*****)
VAR_INPUT
    value : INT;
END_VAR
BEGIN
IF value <= 181 THEN
    SQUARE := value * value; //Calculation of function
value
ELSE
    SQUARE := 32_767; // If overflow, set maximum value
END_IF;
END_FUNCTION
```

2.10 Creating the EVALUATE Function Block

2.10.1 Flow Chart for EVALUATE



2.10.2 Declaration Section of FB EVALUATE

Structure of the Declaration Section

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

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

```
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
```

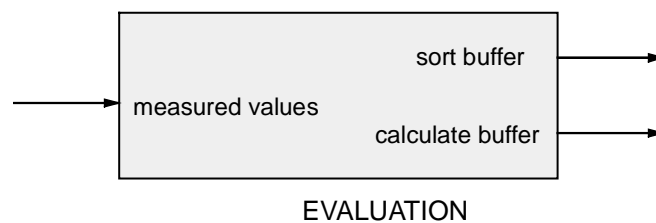
2.10.3 Statement Section of FB EVALUATE

Program Sequence

The in/out parameter "sortbuffer" is linked to the ring buffer "measvals" so that the original contents of the buffer are overwritten by the sorted measured values.

The new array "calcbuffer" is created as an output parameter for the calculated results. Its elements are structured so that they contain the square root and the square of each measured value.

The figure below shows you the relationship between the arrays.



This interface shows the heart of the data exchange for processing the measured values. The data is stored in the instance data block `ACQUIRE_DATA` since a local instance for `FB EVALUATE` was created in the calling `FB ACQUIRE`.

Statement Section of EVALUATE

First, the measured values in the ring buffer are sorted and then the calculations are made.

- **Sort algorithm**
The permanent exchange of values method is used to sort the measured value buffer. This means that consecutive values are compared and their order reversed until the final order is obtained throughout. The buffer used is the in/out parameter "sortbuffer".
- **Starting the calculation**
Once sorting is completed, a loop is executed in which the functions `SQUARE` for squaring and `SQRT` for extracting the square root are called. Their results are stored in the structured array "calcbuffer".

Statement Section of EVALUATE

The statement section of the logic block is as follows:

```

BEGIN
  (*****
  Part 1 Sorting : According to the "bubble sort" method: Swap
  pairs of values until the measured value buffer is 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 : Square root with standard function
  SQRT and squaring with the SQUARE function.
  *****)
  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

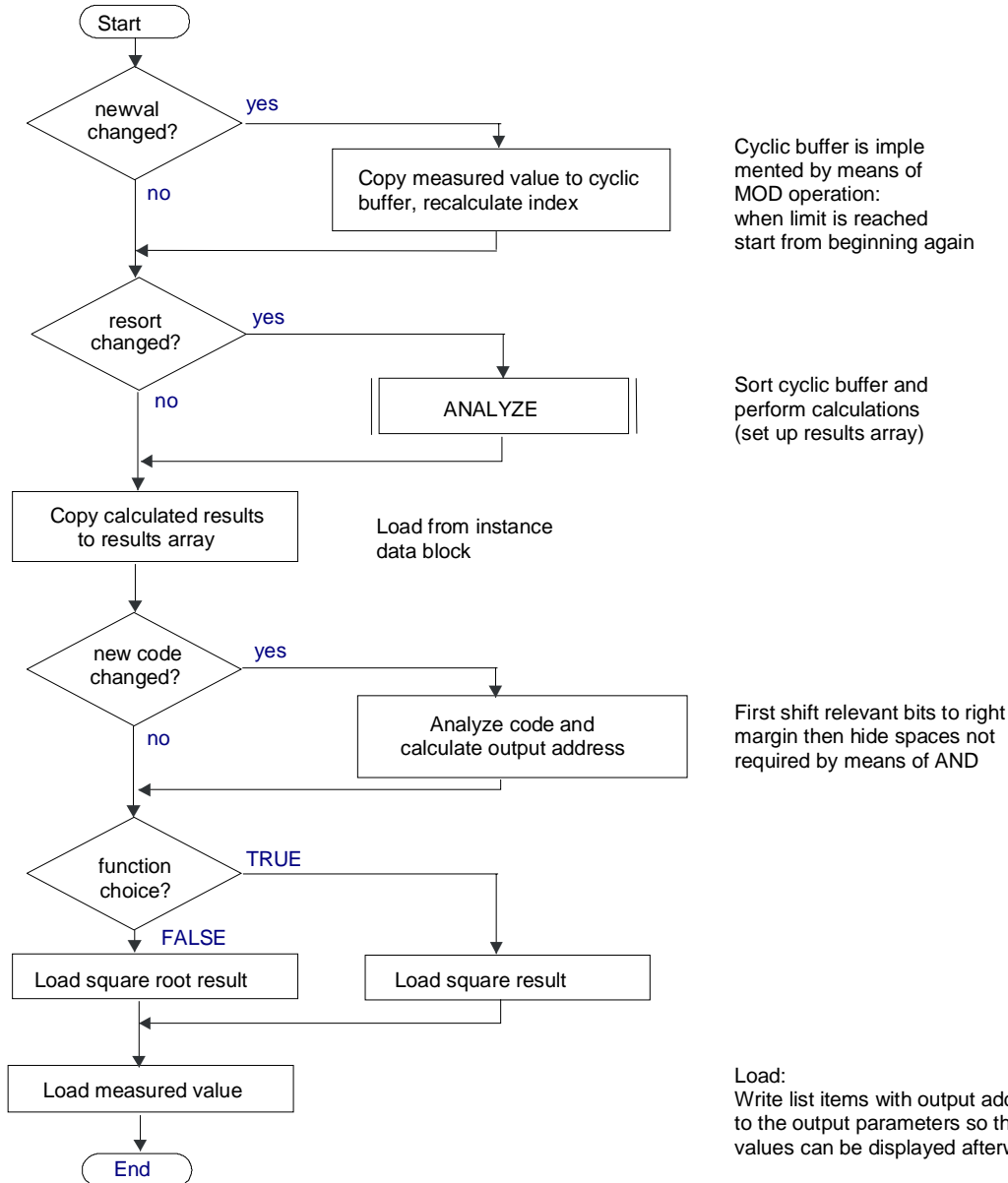
```

2.11 Creating the ACQUIRE Function Block

2.11.1 Flow Chart for ACQUIRE

The following figure shows the algorithm in the form of a flow chart:

RECORD
Function Block



2.11.2 Declaration Section of FB ACQUIRE

Structure of the Declaration Section

The declaration section in this block consists of the subsections:

- Constants: between CONST and END_CONST.
- Input parameters: between VAR_INPUT and END_VAR.
- Output parameters: between VAR_OUTPUT and END_VAR.
- Static variables: between VAR and END_VAR. This also includes declaration of the local instance for the EVALUATE block.

```

CONST
    LIMIT := 7;
    QUANTITY := LIMIT + 1;
END_CONST
VAR_INPUT
    measval_in : INT ;           // New measured value
    newval : BOOL; // Measured value in "measvals" ring buffer
    resort : BOOL; // Sort measured values
    funct_sel : BOOL; // Select calculation square
    root/square
    newssel : BOOL; // Take output address
    selection : WORD; // Output address
END_VAR
VAR_OUTPUT
    result_out : INT; // Calculated value
    measval_out : INT; // Corresponding measured value
END_VAR
VAR
    measvals : ARRAY[0..LIMIT] OF INT := 8(0);
    resultbuffer : ARRAY[0..LIMIT] OF
    STRUCT
        squareroot : INT;
        square : INT;
    END_STRUCT;
    pointer : INT := 0;
    oldval : BOOL := TRUE;
    oldsort : BOOL := TRUE;
    oldsel : BOOL := TRUE;
    address : INT := 0; //Converted output address
    outvalues_instance: EVALUATE; // Define local instance
END VAR

```

Static Variables

The FB block type was chosen because some data needs to be retained from one program cycle to the next. These are the static variables declared in the declaration subsection "VAR, END_VAR".

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.

Initializing Variables

Note the initialization values that are entered in the variables when the block is initialized (after being downloaded to the CPU). The local instance for the EVALUATE FB is also declared in the declaration subsection "VAR, END_VAR". This name is used subsequently for calling and accessing the output parameters. The shared instance ACQUIRE_DATA is used to store the data.

Name	Data Type	Initialization Value	Description
measvals	ARRAY [..] OF INT	8(0)	Ring buffer for measured values
resultbuffer	ARRAY [..] OF STRUCT	-	Array for structures with the components "square root" and "square" of the type INT
index	INT	0	Index for ring buffer identifying location for next measured value
oldval	BOOL	FALSE	Previous value for reading in measured value using "newval"
oldsort	BOOL	FALSE	Previous value for sorting using "resort"
oldsel	BOOL	FALSE	Previous value for reading in code using "newsel"
address	INT	0	Address for output of measured value or result
eval_instance	Local instance	-	Local instance for the EVALUATE FB

2.11.3 Statement Section of FB ACQUIRE

Structure of the Statement Section

The statement section of `ACQUIRE` is divided into three subsections:

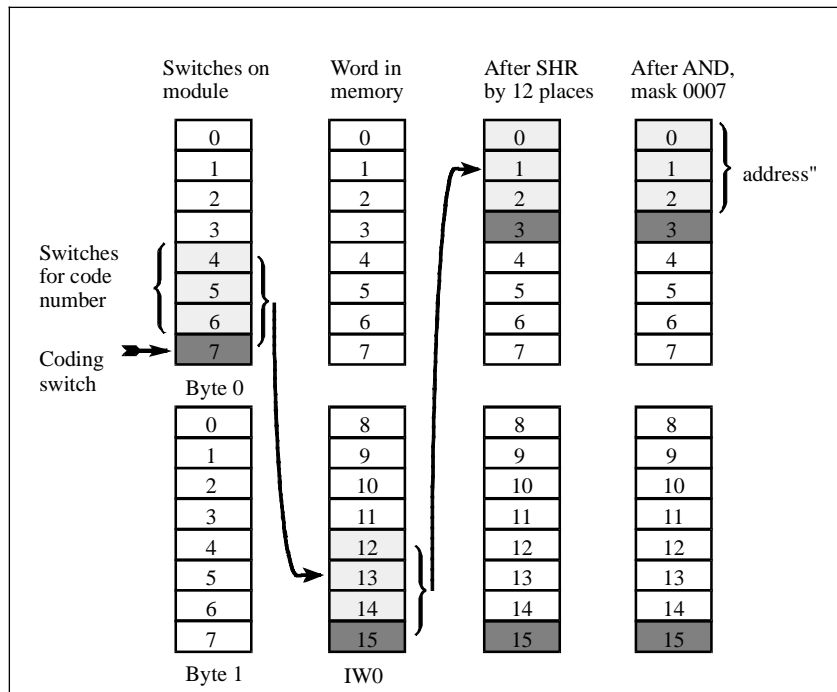
- **Acquire measured values:**
If the input parameter "newval" is different from the "oldval", a new measured value is read into the ring buffer.
- **Start sorting and calculation**
Sorting and calculation are started by calling the `EVALUATE` function block when the input parameter "resort" has changed compared with "oldsort".
- **Evaluating the coding and preparing output data**
The coding is read word by word. According to SIMATIC conventions, this means that the upper group of switches (byte 0) contains the higher-order eight bits of the input word and the lower group of switches (byte 1) the lower-order bits. The figure below shows the location of the coding switches.

Calculating the Address

The figure below shows how the address is calculated: Bits 12 to 14 of input word IW0 contain the coding that is read in when an edge is detected at the coding switch (bit 15). The "address" is obtained by shifting right using the standard function SHR and masking the relevant bits using an AND mask.

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 "funct_sel".

An edge at the coding switch is detected because "newsel" is different from "oldsel".



Statement Section

The statement section of the logic block is shown below:

```
BEGIN
  (*****
  Part 1 : Acquiring measured values. If "newval" changes, the
  measured value is entered. The MOD operation is used to
  implement a ring buffer for measured values.
  *****)
  IF newval <> oldval THEN
    pointer      := pointer MOD QUANTITY;
    measvals[pointer] := measval_in;
    pointer      := pointer + 1;
  END_IF;
  oldval := newval;
  (*****
  Part 2 : Start sorting and calculation
  if "resort" changes, start sorting the
  ring buffer and run calculations with the
  measured values. Results are stored in a new array called
  "calcbuffer".
  *****)
  IF resort <> oldsort THEN
    pointer := 0;           //Reset ring buffer pointer
    eval_instance(sortbuffer := measvals); //Call EVALUATE
  END_IF;
  oldsort := resort;
  resultbuffer := eval_instance.calcbuffer; //Square and square
  root

  (*****
  Part 3 : Evaluate coding and prepare output: If
  "newsel" changes, the coding for addressing the array element
  for output is recalculated: The relevant bits of "selection"
  are masked and converted to integer. Depending on the setting
  of
  the "funct_sel" switch, "squareroot" or "square" is selected
  for output.
  *****)
  IF newsel <> oldsel THEN
    address := WORD_TO_INT(SHR(IN := selection, N := 12) AND
    16#0007);
  END_IF;
  oldsel := newsel;
  IF funct_sel THEN
    result_out := resultbuffer[address].square;
  ELSE
    result_out := resultbuffer[address].squareroot;
  END_IF;
  measval_out := measvals[address]; //Measured value display
END FUNCTION BLOCK
```

2.12 Creating the CYCLE Organization Block

Tasks of the CYCLE OB

An OB1 was chosen because it is called cyclically. It performs the following tasks for the program:

- Calls and supplies the `ACQUIRE` function block with input and control data.
- Reads in the data returned by the `ACQUIRE` function block.
- Outputs the values to the display

At the beginning of the declaration section, there is the 20-byte temporary data array `"system data"`.

Program Code of the CYCLE OB

```

ORGANIZATION_BLOCK CYCLE
(*****
CYCLE is like an OB1, i.e. it is called cyclically by the S7
system.
Part 1 : Function block call and transfer of
the input values Part 2 : Reading in of the output values
and output
with output switchover
*****)
VAR_TEMP
    systemdata : ARRAY[0..20] OF BYTE; // Area for OB1
END_VAR
BEGIN
(* Part 1 :
*****)
ACQUIRE.ACQUIRE_DATA(
    measval_in:= WORD_TO_INT(input),
    newval      := "Input 0.0", //Input switch as signal
    identifier
    resort      := Sort_switch,
    funct_sel:= Function_switch,
    newsel      := Coding_switch,
    selection    := Coding);

(* Part 2 :
*****)
IF Output_switch THEN
//Output changeover
    Output      := ACQUIRE_DATA.result_out;
    //Square root or square
ELSE
    Output      := ACQUIRE_DATA.measval_out;    //Measured value
END_IF;
END ORGANIZATION_BLOCK

```

Data Type Conversion

The measured value is applied to the input as a BYTE data type. It must be converted to the INT data type. You will need to convert it from WORD to INT (the prior conversion from BYTE to WORD is made implicitly by the compiler). The output on the other hand requires no conversion, since this was declared as INT in the symbol table.

2.13 Test Data

Requirements

To perform the test, you require an input module with address 0 and an output module with address 4.

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 on the CPU, since the initial values of the variables must also be tested.

Test Procedure

Run the test as described in the table .

Test	Action	Result
1	Set the code to "111" (I0.4, I0.5 and I0.6) and enter this with the coding switch (I0.7).	All outputs on the output module (lower-order byte) are activated and the LEDs light up.
2	Display the corresponding square root by setting the output switch (I0.3) to "1".	The LEDs 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 LEDs on the output module light up. This indicates an overflow since the result of 255×255 is too high for the integer range.
4a	Reset the output switch (I0.3) back to "0".	The measured value is displayed again. All LEDs on the outputs of the lower-order output byte are set.
4b	Set the value 3 (binary "11") as the new measured value at the input.	The output does not change at this stage.
5a	Monitor reading in of the measured value: Set the code to "000" and enter it with coding switch (I0.7) so that you can later watch the value input.	The output module shows 0; i.e none of the LEDs lights up.
5b	Switch over the input switch "Input 0.0" (I0.0). This reads in the value set in test stage 4.	The output displays measured value 3, binary "11".
6	Start sorting and calculation by switching over the sort switch (I0.1).	The output again indicates 0 since the sorting process has moved the measured value to a higher position in the array.
7	Display the measured value after sorting: Set the code "110" (I0.6 = 1, I0.5 = 1, I0.4 = 0 of I0.0; corresponds to bit 14, bit 13 and bit 12 of IW0) and read it in by switching over the coding switch.	The output now indicates 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 the 7 th step.	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.

Additional Test

The following tables describe the switches on the input module and the examples for square and square root. These descriptions will help you to define your own tests:

- Input is made using switches. You can control the program with the top eight switches and you can set the measured value with the bottom 8 switches.
- Output is indicated by LEDs. The top group displays the higher-order output byte, the bottom group the lower-order byte.

Switch	Parameter Name	Description
Channel 0	Enter switch	Switch over to read in measured value
Channel 1	Sort switch	Switch over to start sorting/calculation
Channel 2	Function switch	Switch left ("0"): Square root, Switch right ("1"): Square
Channel 3	Output switch	Switch left ("0"): Measured value, Switch right ("1"): 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 enter code

The following table contains eight examples of measured values that have already been sorted.

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, start sorting and calculation by changing over the sort switch. You can then view the sorted values or the results (square root or 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!

3 Using SCL

3.1 Starting the SCL Program

Starting from the Windows Interface

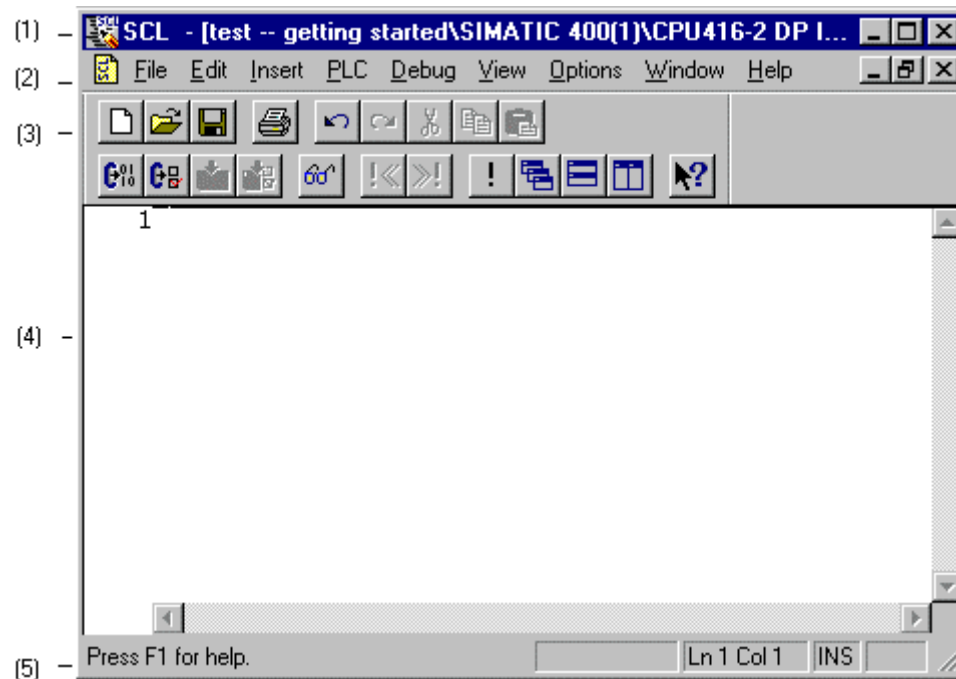
Once you have installed the SCL software on your programming device/PC, you can start SCL using the Start button in the Windows taskbar (entry under "SIMATIC / STEP7").

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.

3.2 User Interface

The SCL windows have the following standard components:



1. **Title bar:**
Contains the window title and window control buttons.
2. **Menu bar:**
Shows all menus available for the open window.
3. **Toolbar:**
Contains buttons for frequently used commands.
4. **Working area:**
Contains one or more windows in which you can edit program text or read compiler information or debugging data
5. **Status bar:**
Displays the status and other information on the active object

3.3 Customizing the User Interface

Customizing the Editor

To make the settings for the editor, select the menu command **Options > Customize** and click the "Editor" tab in the "Customize" dialog box. In this tab, you can make the following settings:

Options in the "Editor" Tab	Explanation
Fonts	Specifies the font for the entire source text.
Tabulator length	Specifies the column width of tabulators.
Display line numbers	Displays line numbers at the beginning of the lines.
Save before compiling	Before compiling, you are asked whether you want to save the source file.
Confirm before saving	Asks for confirmation before saving.

Adapting the Style and Color

To change the style and color of the various language elements, select the menu command **Options > Customize** and click the "Format" tab in the "Customize" dialog box. Here, you can make the following settings:

Options in the "Format" Tab	Explanation
Keywords in uppercase	Formats SCL keywords such as FOR, WHILE, FUNCTION_BLOCK, VAR or END_VAR as upper case characters when you are writing your program.
Indent keywords	Indents lines in the declaration sections and within the control statements IF, CASE, FOR, WHILE and REPEAT while you are writing your program.
Indent automatically	After a line break, the new line is automatically indented by the same amount as the previous line. This setting applies only to new lines.
Style/Color	You can select the style and color of the individual language elements.

The settings in this tab are only effective when you have selected the option "Use following formats" in the "Format" tab.

Toolbar, Breakpoint Bar, Status Bar

You can toggle the display of the toolbar, breakpoint bar and status bar on and off individually. Simply select or deselect the appropriate command in the **View** menu. When the function is activated, a check mark appears next to the command.

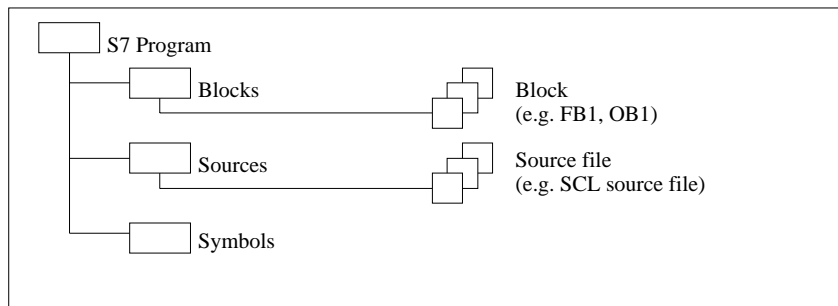
3.4 Creating and Handling an SCL Source File

3.4.1 Creating a New SCL Source File

Before you can write an SCL program, you must first create a new SCL source file. You create the source file in the source files folder in an S7 program.

Structure of an S7 Program in the SIMATIC Manager

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



Follow the steps outlined below:

1. Open the "New" dialog box as follows:
 - Click the "New" button in the toolbar or
 - Select the menu command **File > New**.
2. In the "New" dialog box, select
 - A project
 - The filter setting "SCL Source File" and
 - The source files folder within the S7 program
3. Type in the name of the source object in the text box. The name can be up to 24 characters long.
4. Confirm with "OK".

The source object is created under the name you have selected and is then displayed in a window where you can continue working with it.

Note

You can also create an SCL source file with the SIMATIC Manager by selecting a source file folder and the menu command **Insert > S7 Software > SCL Source File**.

3.4.2 Opening an SCL Source File

You can open an SCL source file so that you can compile or edit it.

Follow the steps outlined below:

1. Open the "Open" dialog box as follows:
 - Click the "Open" button in the toolbar or
 - Select the menu command **File > Open**.
2. Once the dialog box is open, select the following:
 - Required project
 - The required S7 program
 - The corresponding source files folder
3. Select the SCL source file.
4. Click the "OK" button.

Note

You can also open an SCL source file in the SIMATIC Manager by double-clicking its icon or using the menu command **Edit > Open Object** when the object is selected.

3.4.3 Opening Blocks

It is not possible to open blocks with the SCL application. Blocks created with SCL can, however, be opened with the LAD/STL/FBD editor and displayed and edited in the STL programming language. Do not make any modifications to the block in this STL representation for the following reasons:

- The displayed MC7 commands do not necessarily represent a valid STL block.
- An error-free compilation with the STL compiler involves modifications that require thorough knowledge of both STL and SCL.
- The block compiled with STL has the STL language identifier and no longer the SCL identifier.
- The SCL source file and the MC7 code are no longer consistent.

Further information is available in the STEP 7 online help.

Note

It is easier to maintain your CPU programs by making any changes you require in the SCL source files, and then compiling them again.

3.4.4 Closing an SCL Source File

Follow the steps outlined below:

- Select the menu command **File > Close**.
- Click on the "Close" symbol in the title bar of the window.

Note

If you have modified the source file, you will be asked whether or not you want to save any changes before you close the file. If you do not save changes they are lost.

3.4.5 Specifying Object Properties

You can specify the program properties by assigning attributes to the blocks. You can select the properties for the SCL source file (for example, the author) in the "Properties" dialog box.

Follow the steps outlined below:

1. Select the menu command **File > Properties**.
2. Enter the options you require in the "Properties" dialog box.
3. Confirm with "OK".

3.4.6 Creating Source Files with a Standard Editor

You can also use a standard ASCII editor for editing your SCL source file. If you choose this method, the helpful editing functions and integrated online help of SCL are not available.

Once you have created the source file and saved it, you must then import it into the source file folder of an S7 program using the SIMATIC Manager (see STEP 7 documentation). Following this, you can open the source file in SCL and continue working with it or compile it.

3.4.7 Block Protection

You can protect blocks by specifying the `KNOW_HOW_PROTECT` attribute when you program the block in the source file.

Result of Block Protection

- When you open a compiled block later with the incremental STL editor, the statements of the block are hidden.
- In the declarations of the block, only the variables of types `VAR_IN`, `VAR_OUT` and `VAR_IN_OUT` are displayed. The variables of the declaration fields `VAR` and `VAR_TEMP` remain hidden.

Rules for Using Block Protection

- The keyword is `KNOW_HOW_PROTECT`. Enter this before all other block attributes.
- OBs, FBs, FCs, and DBs can be protected in this way.

3.5 Guidelines for SCL Source Files

3.5.1 General Rules for SCL Source Files

SCL source files must comply with the following rules:

- Any number of logic blocks (FB, FC, OB), data blocks (DB), and user-defined data types (UDT) can be edited in an SCL source file.
- Each block type has a typical structure.
- Each statement and each variable declaration ends with a semicolon (;).
- No distinction is made between upper- and lowercase characters.
- Comments are only intended for documenting the program. They do not affect the running of the program.
- Instance data blocks are created automatically when a function block is called. They do not need to be edited.
- DB 0 has a special purpose. You cannot create a data block with this number.

3.5.2 Order of the Blocks

When creating the SCL source file, remember the following rules governing the order of the blocks:

- Called blocks must precede the calling blocks.
- User-defined data types (UDTs) must precede the blocks in which they are used.
- Data blocks that have been assigned a user-defined data type (UDT) are located after the UDT.
- Shared data blocks come before all blocks that access them.

3.5.3 Using Symbolic Addresses

In an SCL program, you work with addresses such as I/O signals, memory bits, counters, timers, and blocks. You can address these elements in your program using absolute addresses (for example, I1.1, M2.0, FB11), however the SCL source files are much easier to read if you use symbols (for example Motor_ON). The address can then be accessed in your user program using the symbol.

Local and Shared Symbols

- You use shared symbols for memory areas of the CPU and block identifiers. These are known throughout the entire application program and must therefore be identified uniquely. You can create the symbol table with STEP 7.
- Local symbols are only known in the block in whose declaration section they are defined. You can assign names for variables, parameters, constants, and jump labels and can use the same name for different purposes in different blocks.

Note

Make sure that the symbolic names are unique and are not identical to any of the keywords.

3.6 Editing in SCL Source Files

3.6.1 Undoing the Last Editing Action

With the menu command **Edit > Undo**, you can reverse one or more actions.

You cannot reverse all actions. As an example, the menu command **File > Save** cannot be reversed.

3.6.2 Redoing an Editing Action

After you have reversed one or more actions, you can restore the reversed actions with the menu command **Edit > Redo**.

3.6.3 Finding and Replacing Text Objects

If you want to edit or modify an SCL source file, you can often save valuable time by searching for text objects and replacing them. You can search, for example, for keywords, absolute identifiers, symbolic identifiers etc.

Follow the steps outlined below:

1. Select the menu command **Edit > Find and Replace....**
2. Enter the options in the "Find and Replace" dialog box.
3. Start the search as follows:
 - Click the "Find" button to find a text object and to mark it or
 - Click the "Replace" or "Replace All" button to find a text and replace it by the text entered in the "Replace with" text box.

3.6.4 Selecting Text Objects

You can select text objects by holding down the mouse button and dragging the mouse pointer over the selected area of text.

You can also:

- Select the complete source text by selecting the menu command **Edit > Select All**.
- Select a word by double-clicking on it.
- Select an entire line by clicking in the margin to the left of the line.

With the menu command **Edit > Undo Selection**, you can cancel the selection.

3.6.5 Copying Text Objects

With this function you can copy entire programs or sections of them. The copied text is placed on the clipboard and can then be pasted as often as you require at other points in the text.

Follow the steps outlined below:

1. Select the text object to be copied.
2. Copy the object as follows:
 - Click the "Copy" button in the toolbar or
 - Select the menu command **Edit > Copy**.
3. Move the cursor to the point at which you want to paste the object (in the same or in a different application).
4. Paste the object as follows:
 - Click the "Paste" button in the toolbar or
 - Select the menu command **Edit > Paste**.

3.6.6 Cutting Text Objects

With this function, you place the selected text on the clipboard. Normally, this menu command is used in conjunction with the menu command **Edit > Paste** that inserts the content of the clipboard at the position currently marked by the cursor.

Follow the steps outlined below:

1. Select the object you want to cut.
2. Cut the object as follows:
 - Click the "Cut" button in the toolbar or
 - Select the menu command **Edit > Cut**.

Note

- The selected object cannot be cut if the menu command **Edit > Cut** is not activated (on a gray background).
 - Using the menu command **Edit > Paste**, you can insert the text you have cut at any point (in the same or in a different program).
 - The content of the clipboard is retained until you use one of the menu commands **Edit > Cut** or **Edit > Copy** again.
-

3.6.7 Deleting Text Objects

You can delete a selected text object from the source text.

Follow the steps outlined below:

1. Select the text you want to delete.
2. Select the menu command **Edit > Delete**.

The deleted text is not copied to the clipboard. The deleted object can be retrieved with the menu command **Edit > Undo** or **Edit > Redo**.

3.6.8 Positioning the Cursor in a Specific Line

With this function you can position the cursor at the start of a particular line.

Follow the steps outlined below:

1. Open the "Go To" dialog box by selecting the menu command **Edit > Go To Line**.
2. Type in the line number in the "Go To" dialog box.
3. Confirm with "OK".

3.6.9 Syntactically Correct Indenting of Lines

The following functions allow you to structure SCL source files by indenting lines.

- **Automatic indent**
When this function is active, the new line following a line break is automatically indented by the same amount as the previous line.
- **Indenting keywords**
When this function is active, lines in the declaration sections and within the control structures IF, CASE, FOR, WHILE and REPEAT are indented.

Follow the steps outlined below:

1. Select the **Options > Customize** menu command.
2. Select the "Format" tab in the dialog displayed.
3. Make sure that the option "Use following formats" is active.
4. Activate the option "Indent automatically" or "Indent keywords".

3.6.10 Setting the Font Style and Color

The use of different styles and colors for the various language elements makes an SCL source file easier to read and lends it a more professional appearance. To format the source text, you can use the following functions:

- **Keywords in uppercase:**
When this function is active, defined keywords such as FOR, WHILE, FUNCTION_BLOCK, VAR or END_VAR are written in uppercase letters.
- **Defining the style and color:**
There are various default styles and colors for the various language elements such as operations, comments or constants. You can change these default settings.
The following colors are the defaults:

Font Color	Language Element	Example
Blue	Keywords	ORGANIZATION_BLOCK
	Predefined data types	INT
	Predefined identifiers	ENO
	Standard functions	BOOL_TO_WORD
Ochre	Operations	NOT
Pink	Constants	TRUE
Blue-green	Comments	//... or (*...*)
Violet	Shared symbols (symbol table) inside quotes	"Motor"
Black	Normal text	Variables

Follow the steps outlined below:

1. Select the **Options > Customize** menu command.
2. Select the "Format" tab in the dialog displayed.
3. Make sure that the option "Use following formats for printing:" is enabled.
4. You can now make the required settings. You can display detailed information about the dialog box by clicking the "Help" button while the dialog is open.

3.6.11 Inserting Templates

3.6.11.1 Inserting Block Calls

SCL supports you when programming block calls. You can call the following blocks:

- System function blocks (SFB) and system functions (SFC) from the SIMATIC libraries,
- Function blocks and functions created in SCL,
- Function blocks and functions created in other STEP 7 languages.

Follow the steps outlined below:

1. Select the **Insert > Block Call** menu command.
2. Select the required SFC, SFB, FC, or FB in the dialog box and confirm your selection with "OK".
SCL copies the called block automatically to the S7 program and enters the block call and the formal parameters of the block with the correct syntax into the source file.
3. If you call a function block, add the information about the instance DB.
4. Enter the actual parameters required by the block. To help you select an actual parameter, SCL indicates the required data type as a comment.

3.6.11.2 Inserting Block Templates

One SCL editing function allows you to insert block templates for OBs, FBs, FCs, instance DBs, DBs, DBs that reference UDTs, and UDTs. Using these block templates makes it easier to program and to keep to the required syntax.

Follow the steps outlined below:

1. Position the cursor at the point at which you want to insert the block template.
2. Select the menu command **Insert > Block Template > OB/FB/FC/DB/IDB/DB Referencing UDT/UDT**.

3.6.11.3 Inserting Templates for Comments

This SCL editing function allows you to insert templates for comments. Using these templates makes it easier to input your information and to keep to the required syntax.

Follow the steps outlined below:

1. Position the cursor after the block header of the required block.
2. Select the menu command **Insert > Block Template > Comment**.

3.6.11.4 Inserting Parameter Templates

One SCL editing function allows you to insert templates for the declarations of the parameters. Using these templates makes it easier to type in your program and to keep to the required syntax. You can declare parameters in function blocks and in functions.

Follow the steps outlined below:

1. Position the cursor in the declaration section of an FB or FC.
2. Select the menu command **Insert > Block Template > Parameter**.

3.6.11.5 Inserting Control Structures

This SCL editing function allows you to insert control structure templates for logic blocks. Using these templates makes it easier to input your information and to keep to the required syntax.

Follow the steps outlined below:

1. Position the cursor at the point at which you want to insert the template.
2. Select the menu command **Insert > Control Structure > IF/CASE/FOR/WHILE/REPEAT**.

3.7 Compiling an SCL Program

3.7.1 What You Should Know About Compiling

Before you can run or test your program, you must first compile it. Once you start compilation, the compiler is started automatically. The compiler has the following characteristics:

- You can compile an entire SCL source file in one compilation session or compile selected individual blocks from the source file.
- All syntax errors found by the compiler are displayed in a window.
- Each time a function block is called, a corresponding instance data block is created if it does not already exist.
- You can also compile several SCL source files together by creating an SCL compilation control file.
- Using the **Options > Customize** menu command, you can set options for the compiler.

Once you have created a user program that is free of errors and has been compiled, you can assume that the program is correct. Problems can, nevertheless, occur when the program is run on the PLC. Use the debugging functions of SCL to find errors of this type.

3.7.2 Customizing the Compiler

You can adapt the compilation to meet your own requirements.

Follow the steps outlined below:

1. Select the menu command **Options > Customize** to open the "Customize" dialog box.
2. Select the "Compiler" tab or "Create Block" tab.
3. Enter the options you require in the tab.

Options in the "Compiler" Tab

Create object code	With this option, you decide whether or not you want to create executable code. Compilation without this option serves simply as a syntax check.
Optimize object code	When you select this option, the blocks are optimized in terms of memory requirements and runtime on the PLC. It is advisable to keep this option permanently selected since the optimization has no disadvantages that affect the functionality of the block.
Monitor array limits	If you select this option, a check is made during the runtime of the S7 program to determine whether array indexes are within the permitted range according to the declaration for the ARRAY. If an array index exceeds the permitted range, the OK flag is set to FALSE.
Create debug info	This option allows you to run a test with the debugger after you have compiled the program and downloaded it to the CPU. The memory requirements of the program and the runtimes on the AS are, however, increased by this option.
Set OK flag	This option allows you to query the OK flag in your SCL source texts.
Permit nested comments	Select this option if you want to nest comments within other comments in your SCL source file.
Maximum string length:	Here, you can reduce the standard length of the STRING data type. The default is 254 characters. The setting affects all output and in/out parameters as well as the return values of functions. Note the value you set must not be smaller than the STRING variables actually used in the program.

Options in the "Create Block" Tab

Overwrite blocks	Overwrites existing blocks in the "Blocks" folder of an S7 program if blocks with the same identifier are created during compilation. Blocks with the same name that already exist on the target system are also overwritten when you download blocks. If you do not select this option, you are prompted for confirmation before a block is overwritten.
Display warnings	You can decide whether you also want warnings displayed in addition to errors following compilation.
Display errors before warnings	You can have errors listed before warnings in the display window.
Generate reference data	Select this option if you want reference data to be generated automatically when a block is created. With the menu command Options > Reference Data , you can also generate or update the reference data later.
Include system attribute 'S7_server'	Select this option if you want the "S7 server" system attribute for parameters to be taken into account when a block is created. You assign this attribute when the parameter is relevant to the configuration of connections or messages. It contains the connection or message number. This option extends the time required for compilation.

3.7.3 Compiling the Program

Before you can test a program or run it, it must first be compiled. 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" tab. The menu command **File > Compile** then implicitly saves the SCL source file.

Follow the Steps Outlined Below:

1. Save the SCL source file to be compiled.
2. To create an executable program, you must select the option "Create object code" in the "Compiler" tab of the "Customize" dialog box.
3. If required, modify other compiler settings.
4. Check whether the corresponding symbol table is in the same program folder.
5. You can start compilation in the following ways:
 - The menu command **File > Compile** compiles the entire source file.
 - The menu command **File > Compile Selected Blocks** opens a dialog box in which you can select individual blocks for compilation.
6. The "Errors and Warnings" dialog box displays all syntax errors and warnings that occurred while the program was being compiled. Correct any errors reported by the compiler and then repeat the procedure outlined above.

3.7.4 Creating a Compilation Control File

If you create a compilation control file, you can compile several SCL source files at one time within a source folder. In the compilation control file, you enter the name of the SCL source files in the order in which they are to be compiled.

Follow the Steps Outlined Below:

1. Open the "New" dialog box by selecting the menu command **File > New**.
2. In the "New" dialog box, select
 - a source file folder within an S7 program and
 - the filter "SCL Compilation Control File"
3. Enter the name of the control file in the corresponding box (max. 24 characters) and confirm with "OK".
4. The file is created and displayed in a working window for further editing. In the working window, enter the name of the SCL source files to be compiled in the required order and save the file.
5. Then start the compilation by selecting the menu command **File > Compile**.

3.7.5 Debugging the Program After Compilation

All the syntax errors and warnings that occur during compilation are displayed in the "Errors and Warnings" window. If an error occurs, the block cannot be compiled, whereas if only warnings occur, an executable block is compiled. You may still, nevertheless, encounter problems running the block on the PLC.

To correct an error:

1. Select the error and press the F1 key to display a description of the error and instructions on correcting the error.
2. If a line number and column number are displayed, you can locate the error in the source text as follows:
 - Click the error message in the "Errors and Warnings" window with the right mouse button and then select the **Display Errors** command.
 - Double-click the error message to position the cursor on the point reported (line, column).
3. Find out the correct syntax in the SCL Language Description.
4. Make the necessary corrections in the source text.
5. Save the source file.
6. Compile the source file again.

3.8 Saving and Printing an SCL Source File

3.8.1 Saving an SCL Source File

The term "saving" in SCL 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 folder. You can save an SCL source file in its current state at any time. The object is not compiled. Any syntax errors are also saved.

Follow the steps outlined below:

- Select the menu command **File > Save**, or click the "Save" button in the toolbar.
The SCL source file is updated.
- If you want 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.

3.8.2 Customizing the Page Format

You can modify the appearance of a printout as follows:

- The menu command **File > Page Setup** allows you to select the page format for your printout.
- You can set headers and footers for your documents in the SIMATIC Manager using the menu command **File > Headers and Footers**.
- You can also display and check the page layout before you print it using the menu command **File > Print Preview**.

3.8.3 Printing an SCL Source File

The SCL source file in the active editing window is printed; in other words, to print an SCL source file, this file must already be open.

Follow the steps outlined below:

1. Activate the editing window for the SCL source file you want to print.
2. Open the "Print" dialog box as follows:
 - Click the "Print" button in the toolbar or
 - Select the menu command **File > Print**.
3. Select the option you require in the "Print" dialog box, such as print range and number of copies.
4. Confirm with "OK".

3.8.4 Setting the Print Options

You can use the following functions to format your printout:

- **Form feed at start of block**
When this check box is enabled, each block is printed out on a new page.
- **Print line numbers**
When this check box is enabled, line numbers are printed out at the beginning of each line.
- **Define the font**
The default font for the entire text is Courier size 8. You can change this setting.
- **Define style**
You can define various styles for the various language elements. You can select the following elements:

Language Element	Example
Normal text	
Keyword	ORGANIZATION_BLOCK
Identifiers of predefined data types	INT
Predefined identifiers	ENO
Identifiers of standard functions	BOOL_TO_WORD
Operations	NOT
Constants	TRUE
Comment section	(* *)
Line comment	//...
Shared symbols (symbol table) inside quotes	"Motor"

Follow the steps outlined below:

1. Select the **Options > Customize** menu command.
2. Select the "Print" tab in the dialog displayed.
3. Make sure that the "Use following formats" check box is enabled.
4. Now make the required settings. You can display detailed information about the dialog box by clicking the "Help" button while the dialog is open.

3.9 Downloading the Created Programs

3.9.1 CPU Memory Reset

With the Clear/Reset function, you can delete the entire user program on a CPU online.

Follow the steps outlined below:

1. Select the menu command **PLC > Operating Mode** and switch the CPU to STOP.
2. Select the menu command **PLC > Clear/Reset**.
3. Confirm this action in the dialog box that is then displayed.



Warning

- The CPU is reset.
 - All user data are deleted.
 - The CPU terminates all existing connections.
 - If a memory card is inserted, the CPU copies the content of the memory card to the internal load memory after the memory reset.
-

3.9.2 Downloading User Programs to the CPU

Requirements

When you compile an SCL source file, blocks are created from the source file and are saved in the "Blocks" folder of the S7 program.

Blocks that are called at the first level in SCL blocks are automatically copied to the "Blocks" directory and entered in the load list.

You can download further blocks from the user program with the SIMATIC Manager from the programming device to the CPU.

Before you can download blocks, a connection must exist between the programming device and CPU. An online user program must be assigned to the CPU module in the SIMATIC manager.

Procedure

Once you have compiled the source file, you can start the download in the following ways.

- The **File > Download** menu command downloads all blocks in the source file and all blocks that are called at the first level.
- The **File > Compile Selected Blocks** menu command opens a dialog box in which you can select individual blocks for compilation.

The blocks are transferred to the CPU. If a block already exists in the RAM of the CPU you will be asked to confirm whether or not you want to overwrite the block.

Note

It is advisable to download the user program in the STOP mode, since errors can occur if you overwrite an old program in the RUN mode.

3.10 Debugging the Created Programs

3.10.1 The SCL Debugging Functions

Using the SCL debugging functions, you can check the execution of a program on the CPU and locate any errors in the program. Syntax errors are indicated by the compiler. Runtime errors occurring during the execution of the program are also indicated, in this case, by system interrupts. You can locate logical programming errors using the debugging functions.

SCL Debugging Functions

In SCL, you can start the following test functions:

- **Monitor (S7-300/400-CPU)**
With this function, you can display the names and current values of variables in the SCL source file. During the test, the values of the variables and the parameters are displayed in chronological order and updated cyclically.
- **Debug with Breakpoints/Single Step (S7-400 CPUs only)**
With this function, you can set breakpoints and then debug in single steps. You can execute the program algorithm, for example statement by statement and can see how the values of the variables change.



Caution

Running a test while your plant is in operation can lead to serious injury to personnel or damage to equipment if the program contains errors!

Always make sure that no dangerous situations can occur before activating debugging functions.

Requirements for Debugging

- The program must be compiled with the options "Create object code" and "Create debug info". You can select the options in the "Compiler" tab of the "Customize" dialog box. You can display this dialog with the menu command **Options > Customize**.
- There must be an online connection from the programming device/PC to the CPU.
- The program must also be loaded on the CPU. You can do this with the menu command **PLC > Download**.

3.10.2 The "Monitor" Debugging Function

Using the continuous monitoring 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 obtained from consecutive cycles.

Values that have changed in the current cycle and values that have not changed can be distinguished by their color.

The range of statements that can be tested depends on the performance of the connected CPU. After compilation, the 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.

Remember the following restrictions for the "Monitor" function:

- Variables of a higher data type cannot be displayed in their entirety. The elements of these variables can be monitored providing they are not of an elementary data type.
- Variables of the type DATE_AND_TIME and STRING and parameters of the type BLOCK_FB, BLOCK_FC, BLOCK_DB, BLOCK_SDB, TIMER and COUNTER are not displayed.
- Access to data blocks with the format <symbol>.<absoluteaddress> are not displayed (for example data.DW4).

Querying this information 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 modes of operation.

Operating Mode	Explanation
Test Operation	In the "Test Operation" mode, the monitoring range is only limited by the performance of the connected CPU. All the debugging functions can be used without restrictions. The CPU cycle time can be extended considerably since the status of statements, for example, in programmed loops is queried in each iteration.
Process Operation	In the "Process Operation" mode, the SCL debugger restricts the maximum monitoring range so that the cycle times during testing do not exceed the real runtimes of the process or only insignificantly.

3.10.3 Debugging with Breakpoints/Single Step Mode"

If you test with breakpoints, the program is tested step by step. You can execute the program algorithm statement by statement and can 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 start step-by-step monitoring at this breakpoint.

Single Step Functions:

When the "Debugging with Breakpoints" function is active, you can use the following functions:

- Next Statement
The currently selected statement is executed.
- Resume
Resume until the next active breakpoint.
- To Cursor
Resume as far as the cursor position you have selected in the source file.
- Execute Call
Jump to or call an SCL block lower down the call hierarchy.

Breakpoints

You can define breakpoints at any point in the statement section of the source text.

The breakpoints are only transferred to the programmable controller and activated when you select the menu command **Debug > Breakpoints Active**. The program is then executed as far as the first breakpoint.

The maximum number of active breakpoints depends on the CPU.

CPU 416	maximum of 4 active breakpoints possible
CPU 414	maximum of 4 active breakpoints possible
S7-300 CPUs	no active breakpoints possible

Requirements:

The opened source file was not modified previously in the editor.

3.10.4 Steps in Monitoring

Once you have downloaded the compiled program to the programmable controller, you can test it in the "Monitor" mode.

Follow the steps outlined below:

1. Make sure that the requirements for debugging are satisfied and that the CPU is in the RUN or RUN-P mode.
2. Select the window containing the source file of the program to be tested.
3. If you want to change the default mode (process operation), select the menu command **Debug > Operation > Test Operation**.
4. Position the cursor in the line of the source text containing the first statement of the range to be tested.
5. Make sure that no dangerous situations can result from running the program and then select the menu command **Debug > Monitor**.
Result: The largest possible monitoring range is calculated 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 command **View > Symbolic Representation** to toggle the symbolic names from the symbol table on and off in your program.
7. Select the menu command **Options > Customize**, open the "Format" tab, and make the settings for the colors in which the values will be displayed.
8. Select the menu command **Debug > Monitor** to halt debugging.
9. Select the menu command **Debug > Finish Debugging** to quit the debugging function.

Note

The number of statements that you can debug (monitoring range) depends on the performance of the connected CPU.

3.10.5 Steps for Debugging with Breakpoints

3.10.5.1 Defining Breakpoints

To set and define breakpoints:

1. Open the source file you want to debug.
2. Display the toolbar for breakpoint editing with the menu command **View > Breakpoint Bar**.
3. Position the cursor at the required point and select the menu command **Test > Set Breakpoint** or the button in the breakpoint bar. The breakpoints are displayed at the left edge of the window as a red circle.
4. If required, select **Debug > Edit Breakpoints** and define a call environment. The call environment decides whether or not a breakpoint is only active when the block in which it is located
 - is called by a specific higher-level block and/or
 - is called with a specific data block.

3.10.5.2 Starting the Test with Breakpoints

Once you have downloaded the compiled program to the programmable controller and set breakpoints, you can debug it in the "Test with Breakpoints" mode.

Follow the steps outlined below:

1. Open the SCL source file of the program you want to debug.
2. Make sure that no dangerous situations can result from running the program and that the CPU is in the RUN-P mode. Select the menu command **Debug > Breakpoints Active** and then **Debug > Monitor**.
Result: The window is split vertically into two halves. The program is executed as far as the next breakpoint. When this is reached, the CPU changes to HOLD and the red breakpoint is marked by a yellow arrow.
3. Continue with one of the following commands:
 - Select the menu command **Debug > Resume** or click the "Resume" button.
 The CPU changes to the RUN mode. When the next active breakpoint is reached, it changes to hold again and displays the breakpoint in the right-hand window.
 - Select the menu command **Debug > Next Statement** or click the "Next Statement" button.
 The CPU changes to RUN. After processing the selected statement it changes to hold again and displays the contents of the currently processed variables in the right-hand window.
 - Select the menu command **Debug > To Cursor** or click the "To Cursor" button.
 The CPU changes to the RUN mode. When the selected point in the program is reached, it changes to hold again.

- Select the menu command **Debug > Execute call**, if the program stops in a line containing a block call.
If the lower block in the call hierarchy was created with SCL, it is opened and executed in the test mode. After it is processed, the program jumps back again to the call point.
If the block was created in another language, the call is skipped and the program line that follows is selected.

Note

The menu commands **Debug > Next Statement** or **Debug > To Cursor** set and activate a breakpoint implicitly. Make sure that you have not used the maximum number of active breakpoints for your particular CPU when you select these functions.

3.10.5.3 Stopping the Test with Breakpoints

To return to normal program execution:

- Deactivate the **Debug > Breakpoints Active** menu command to interrupt debugging or
- Select the menu command **Debug > Finish Debugging** to quit debugging.

3.10.5.4 Activating, Deactivating and Deleting Breakpoints

You can activate/deactivate and delete set breakpoints individually:

1. Select the menu command **Debug > Edit Breakpoints**.
2. In the dialog, you can
 - activate and deactivate selected breakpoints with a check mark.
 - delete individual breakpoints.

To delete all breakpoints, select the menu command **Debug > Delete All Breakpoints**.

3.10.5.5 Debugging in the Single Step Mode

Follow the steps outlined below:

1. Set a breakpoint before the statement line from which you want to debug your program in the single step mode.
2. Select the menu command **Debug > Breakpoints Active**.
3. Run the program until it reaches this breakpoint.
4. To execute the next statement, select the menu command **Debug > Next Statement** or click the button in the toolbar.
 - If the statement is a block call, the call is executed and the program jumps to the first statement after the block call.
 - With the **Debug > Execute Call** menu command, you jump to the block. Here, you can then continue debugging in the single step mode or you can set breakpoints. At the end of the block, you return to the statement after the block call.

3.10.6 Using the STEP 7 Debugging Functions

3.10.6.1 Creating and Displaying Reference Data

You can create and evaluate reference data to help you when debugging and modifying your user program.

You can display the following reference data:

- The user program structure
- The cross reference list
- The assignment list
- The list of unused addresses
- The list of addresses without symbols

Creating reference data

You can create reference data in the following ways:

- Using the menu command **Options > Reference Data > Display**, you can create or update and display the data as required.
- By filtering, you can restrict the amount of reference data displayed and speed up the creation and display of the data considerably. Select the **Options > Reference Data > Filter** menu command.
- Using menu command **Options > Customize**, you can decide whether or not the reference data are created 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" tab. Remember that creating reference data automatically will increase the time taken to compile your program.

3.10.6.2 Monitoring and Modifying Variables

When you test your program with the "monitoring and modifying variables" function, you can do the following:

- Display the current values of global data contained in your user program (monitor)
- Assign fixed values to the variables used in your user program (modify)

Follow the steps outlined below:

- Select the menu command **PLC > Monitor/Modify Variables**.
- Create the variable table (VAT) in the displayed window. If you want to modify variables, enter the new values for the variables.
- Specify the trigger points and conditions.



Caution

Running a test while your plant is in operation can lead to serious injury to personnel or damage to equipment if the program contains errors! Before running the debugging functions, make sure that no dangerous situations can occur!

3.11 Displaying and Modifying CPU Properties

3.11.1 Displaying and Modifying the CPU Operating Mode

You can query and modify the current operating mode of a CPU. There must be a connection to the CPU.

Follow the steps outlined below:

1. Select the menu command **PLC > Operating Mode**.
2. In the dialog box that is then displayed, select one of the following modes:
 - Warm restart
 - Cold restart
 - Hot restart
 - STOP



Warning

Modifying the operating mode while your plant is in operation can lead to serious injury to personnel or damage to equipment if the program contains errors!

Before running the debugging functions, make sure that no dangerous situations can occur!

3.11.2 Displaying and Setting the Date and Time on the CPU

You can query and modify the current time on a CPU. There must be a connection to the CPU.

Follow the steps outlined below:

1. Select the menu command **PLC > Set Date and Time**.
2. In the dialog box that appears, set the date and time for the real-time clock of the CPU.

If the CPU is not equipped with a real-time clock, the dialog box for the time displays "00:00:00" and the date box has the value "00.00.00". This means that you cannot make any changes.

3.11.3 Reading Out CPU Data

You can display the following information about a CPU:

- The system family, CPU type, order number, and version of the CPU.
- Size of the work memory and the load memory and the maximum possible configuration of the load memory.
- Number and address area of inputs and outputs, timers, counters, and memory bits.
- Number of local data with which this CPU can work.
- Whether or not a CPU is capable of multiprocessing (CPU-dependent).

There must be a connection to the CPU.

Follow the steps outlined below:

1. Select the menu command **PLC > Module Information**.
2. Select the "General" tab in the dialog box.

3.11.4 Reading Out the Diagnostic Buffer of the CPU

If you read out the diagnostic buffer, you can find out the cause of the STOP mode or back track the occurrence of diagnostic events.

There must be a connection to the CPU.

Follow the steps outlined below:

1. Select the menu command **PLC > Module Information**.
2. Select the "Diagnostic Buffer" tab in the next dialog box.

3.11.5 Displaying/Compressing the User Memory of the CPU

Using this function, you can display information about the memory load of the CPU and, if necessary, reorganize the CPU memory. This is necessary when the largest free continuous memory area is no longer large enough to take a new object downloaded onto the CPU from the PG.

There must be a connection to the CPU.

Follow the steps outlined below:

1. Select the menu command **PLC > Module Information**.
2. Select the "Memory" tab in the next dialog box.

3.11.6 Displaying the Cycle Time of the CPU

The following times are represented within the two selectable limit values:

- Duration of the longest cycle since the last change from STOP to RUN.
- Duration of the shortest cycle since the last change from STOP to RUN.
- Duration of the last cycle.

If the duration of the last cycle comes close to the watchdog time, it is possible that the watchdog time will be exceeded, and that the CPU will change to the STOP mode. The cycle time can be extended, for example, if you test blocks in the program status. To display the cycle times of your program, there must be a connection to the CPU.

Follow the steps outlined below:

1. Select the menu command **PLC > Module Information**.
2. Select the "Cycle Time" tab in the next dialog box.

3.11.7 Displaying the Time System of the CPU

The time system of the CPU includes information about the internal clock and the time synchronization between multiple CPUs.

There must be a connection to the CPU.

Follow the steps outlined below:

1. Select the menu command **PLC > Module Information**.
2. Select the "Time System" tab in the next dialog box.

3.11.8 Displaying the Blocks on the CPU

You can display the blocks available online for the CPU.

There must be a connection to the CPU.

Follow the steps outlined below:

1. Select the menu command **PLC > Module Information**.
2. In the next dialog box, select the "Performance Data/Blocks" tab.

3.11.9 Displaying Information about Communication with the CPU

For each CPU, you can display information online about the selected and maximum transmission rates, connections and the communications load.

There must be a connection to the CPU.

Follow the steps outlined below:

1. Select the menu command **PLC > Module Information**.
2. Select the "Communication" tab in the next dialog box.

3.11.10 Displaying the Stacks of the CPU

By selecting this tab, you can display information online about the content of the stacks of each CPU. The CPU must be in the STOP mode or must have reached a breakpoint.

Displaying the stacks is extremely useful to help you locate errors, for example when testing your blocks. If the CPU changes to STOP, you can display the interrupt point with the current status bits and accumulator contents in the interrupt stack (I stack) to find out the cause (for example of a programming error).

There must be a connection to the CPU.

Follow the steps outlined below:

1. Select the menu command **PLC > Module Information**.
2. Select the "Stacks" tab in the next dialog box.

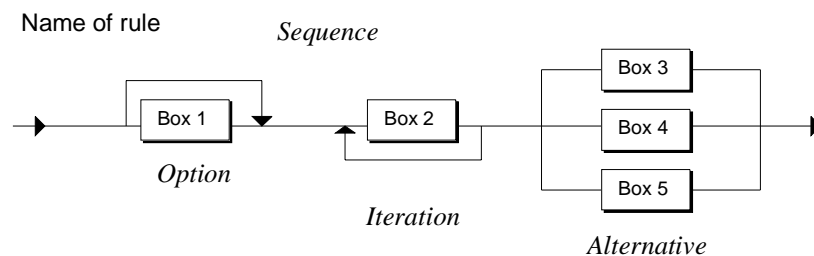
4 Basic SCL Terms

4.1 Interpreting the Syntax Diagrams

The basic tool for the description of the language in the various sections is the syntax diagram. It provides a clear insight into the structure of SCL syntax. The section entitled "Language Description" contains a collection of all the diagrams with the language elements.

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.

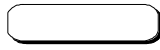


The syntax diagram is read from right to left. The following rule structures must be adhered to:

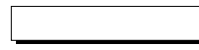
- Sequence: a sequence of boxes
- Option: a skippable branch
- Iteration: repetition of branches
- Alternative: multiple alternative branches

What Types of Boxes Are There?

A box is a basic element or an element made up of other objects. The diagram below shows the symbols that represent the various types of boxes.



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.

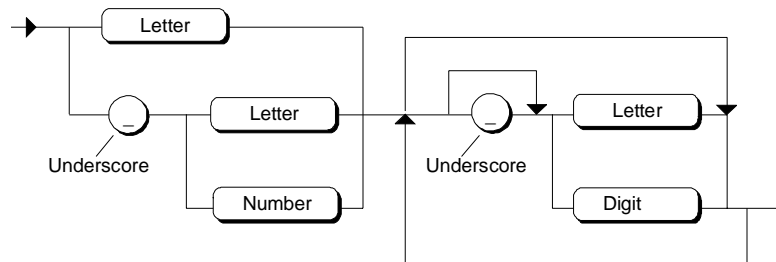
What Does Flexible Format Mean?

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 the section entitled "Language Description". Flexible format means that you can insert formatting characters such as spaces, tabs and page breaks as well as comments between the rule sections.

With lexical rules, there is no flexibility of format! When you apply a lexical rule, you must adopt the specifications exactly as set out.

Lexical Rule



The following examples keep to the above rule:

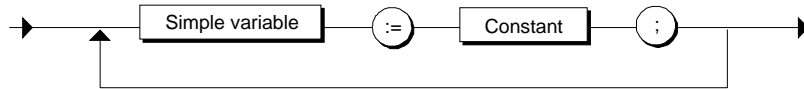
```
R_CONTROLLER3
_A_FIELD
_100_3_3_10
```

The following examples are invalid for the reasons listed above:

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

Syntax Rule

With syntax rules, the format is flexible.



The following examples keep to the above rule:

```

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

4.2 Character Set

Letters and Numeric Characters

SCL uses the following characters as a subset of the ASCII character set:

- The (upper- and lowercase) letters A to Z.
- The Arabic numbers 0 to 9.
- Blanks - the blank itself (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:

+	-	*	/	=	<	>	[]	()
:	;	\$	#	"	'	{	}	%	.	,

Note

In the section entitled "Language Description", you will find a detailed list of all the permitted characters and information on how the characters are interpreted in SCL.

4.3 Reserved Words

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

Keywords in SCL

AND	END_CASE	ORGANIZATION_BLOCK
ANY	END_CONST	POINTER
ARRAY	END_DATA_BLOCK	PROGRAM
AT	END_FOR	REAL
BEGIN	END_FUNCTION	REPEAT
BLOCK_DB	END_FUNCTION_BLOCK	RETURN
BLOCK_FB	END_IF	S5TIME
BLOCK_FC	END_LABEL	STRING
BLOCK_SDB	END_TYPE	STRUCT
BLOCK_SFB	END_ORGANIZATION_BLOCK	THEN
BLOCK_SFC	END_REPEAT	TIME
BOOL	END_STRUCT	TIMER
BY	END_VAR	TIME_OF_DAY
BYTE	END_WHILE	TO
CASE	ENO	TOD
CHAR	EXIT	TRUE
CONST	FALSE	TYPE
CONTINUE	FOR	VAR
COUNTER	FUNCTION	VAR_TEMP
DATA_BLOCK	FUNCTION_BLOCK	UNTIL
DATE	GOTO	VAR_INPUT
DATE_AND_TIME	IF	VAR_IN_OUT
DINT	INT	VAR_OUTPUT
DIV	LABEL	VOID
DO	MOD	WHILE
DT	NIL	WORD
DWORD	NOT	XOR
ELSE	OF	Names of the standard functions
ELSIF	OK	
EN	OR	

4.4 Identifiers

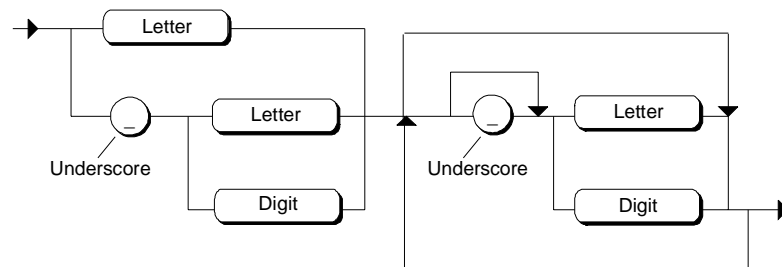
Definition

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

Rules

Identifiers can be made up of a maximum of 24 letters or numbers in any order but the first character must be either a letter or the underscore character. Both uppercase and lowercase letters are permitted. However, the identifiers are not case-sensitive (AnNa and AnnA, for example, are identical).

IDENTIFIER



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
S Value	//Blanks are not allowed (remember //that a blank is also a character).

Notes

- Make sure that the name is not already being used by keywords or standard identifiers.
- It is advisable to select unique names with a clear meaning to make the source text easier to understand.

4.5 Standard Identifiers

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

- The block identifiers,
- The address identifiers for addressing memory areas of the CPU,
- The timer identifiers and
- The counter identifiers.

4.6 Block Identifier

Definition

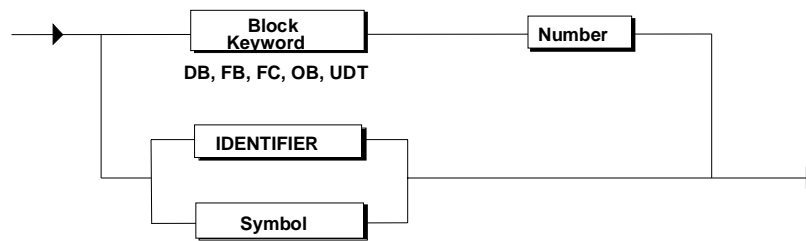
These standard identifiers are used for absolute addressing of blocks.

Rules

The table is sorted in the order of the German mnemonics and the corresponding international mnemonics are shown in the second column. The letter x is a placeholder for a number between 0 and 65533 or 0 and 65535 for timers and counters.

Mnemonic (SIMATIC)	Mnemonic (IEC)	Identifies
DBx	DBx	Data block. The block identifier DB0 is reserved for SCL.
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	User-defined data type
Zx	Cx	Counter

In SCL, there are several ways in which you can specify the block identifier. You can specify a whole decimal number as the number of the block.



Example

The following are examples of valid identifiers:

FB10
DB100
T141

4.7 Address Identifier

Definition

At any point in your program, you can address memory areas of a CPU using their address identifiers.

Rules

The table is sorted in the order of the German mnemonics and the corresponding international mnemonics are shown in the second column. The address identifiers for data blocks are only valid when the data block is also specified.

Mnemonic (German)	Mnemonic (Internat.)	Addresses	Data Type
Ax.y	Qx.y	Ouput (via the 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	Dx.y	Data block	Bit
DBx	DBx	Data block	Byte
DDx	DDx	Data block	Double word
DWx	DWx	Data block	Word
DXx.y	DXx.y	Data block	Bit
Ex.y	Ix.y	Input (via the process image)	Bit
EBx	IBx	Input (via process image)	Byte
EDx	IDx	Input (via process image)	Double word

Mnemonic (German)	Mnemonic (Internat.)	Addresses	Data Type
EWx	IWx	Input (via process image)	Word
EXx.y	IXx.y	Input (via process image)	Bit
Mx.y	Mx.y	Memory bit	Bit
MBx.y	MBx.y	Bit memory	Byte
MDx	MDx	Bit memory	Double word
MWx	MWx	Bit memory	Word
MXx	MXx	Bit memory	Bit
PABx	PQBx	Output (Direct to peripherals)	Byte
PADx	PQDx	Output (Direct to peripherals)	Double word
PAWx	PQWx	Output (Direct to peripherals)	Word
PEBx	PIBx	Input (Direct from peripherals)	Byte
PEDx	PIDx	Input (Direct from peripherals)	Double word
PEWx	PIWx	Input (Direct from peripherals)	Word

x = number between 0 and 65535 (absolute address)

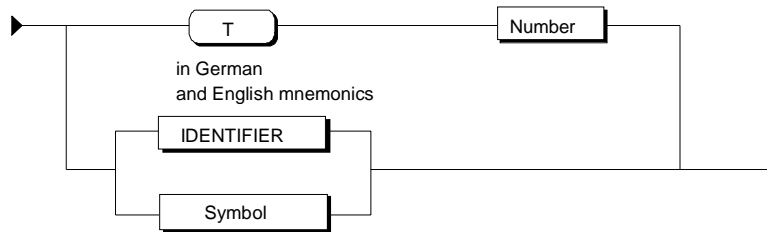
Example

I1.0 MW10 PQW5 DB20.DW3

4.8 Timer Identifier

Rules

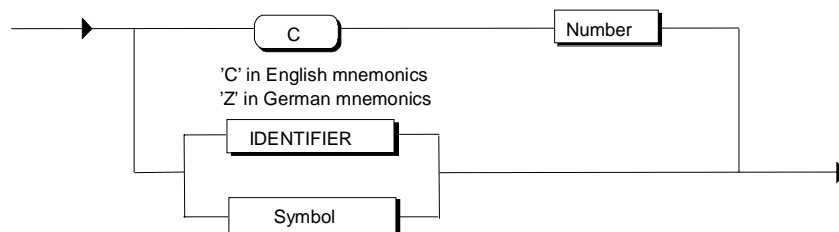
In SCL, there are several ways in which you can specify a timer. You can specify a whole decimal number as the number of the timer.



4.9 Counter Identifier

Rules

There are several ways of specifying a counter in SCL. You can specify a whole decimal number as the number of the counter.



4.10 Numbers

In SCL, there are various ways of writing numbers. The following rules apply to all numbers:

- A number can have an optional sign, a decimal point, and an exponent.
- A number must not contain commas or spaces.
- To improve legibility, the underscore (`_`) can be used as a separator.
- The number can be preceded if required by a plus (`+`) or minus (`-`). If the number is not preceded by a sign, it is assumed to be positive.
- Numbers must not exceed or fall below certain maximum and minimum values.

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` and `DINT`, each of which has a different range of possible values.

Examples of valid integers:

0	1	+1	-1
743	-5280	600_00	-32_211

The following integers are **incorrect** 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.

In SCL, you can represent integers in different numeric systems by preceding the integer with a keyword for the numeric 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 either contain a decimal point or an exponent (or both). A decimal point must be between two digits. 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 **incorrect**:

1.	There must be a digit on both sides of the decimal point.
1,000.0	Integers must not contain commas.
.3333	There must be a digit on both sides of the decimal point.

A real number can include an exponent 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 digit. The exponent itself must be either a positive or a negative integer. Base 10 is represented by the letter E.

The value 3×10 exponent 10 can be 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 **incorrect**:

3.E+10	There must be a digit on both sides of the decimal point.
8e2.3	The exponent must be an integer.
.333e-3	There must be a digit on both sides of the decimal point.
30 E10	Integers must not contain spaces.

4.11 Character Strings

Definition

A character string is a sequence of characters (in other words letters, numbers, and special characters) set in quotes.

Examples of valid character strings:

```
'RED'      '76181 Karlsruhe'      '270-32-3456'
'DM19.95'   'The correct answer is:'
```

Rules

You can enter special formatting characters, the quote (') or a \$ character with the escape symbol \$.

Source Text	After Compilation
'SIGNAL\$'RED\$' '	SIGNAL' RED'
'50.0\$\$'	50.0\$
'VALUE\$P'	VALUE page break
'RUL\$L'	RUL line feed
'CONTROLLER\$R	CONTROLLER carriage return
'STEP\$T'	STEP tabulator

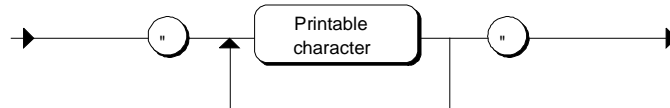
To enter nonprintable characters, type in the substitute representation in hexadecimal code in the form `$hh`, where `hh` stands for the value of the ASCII character expressed in hexadecimal.

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.

4.12 Symbol

You can enter symbols in SCL using the following syntax. The quotes are only necessary when the symbol does not adhere to the IDENTIFIER rule.

Syntax:



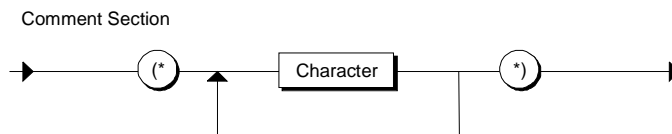
4.13 Comment Section

Rules

- The comment section can extend over a number of lines and is preceded by '(*' and terminated by '*)'.
- The default setting permits the nesting of comment sections. You can, however, change this setting and prevent the nesting of comment sections.
- 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.

Syntax

The comment section is represented formally by the following syntax diagram:



Example

```
(* This is an example of a comment section,
that can extend over several lines.*)
```

```
SWITCH := 3 ;
END_FUNCTION_BLOCK
```

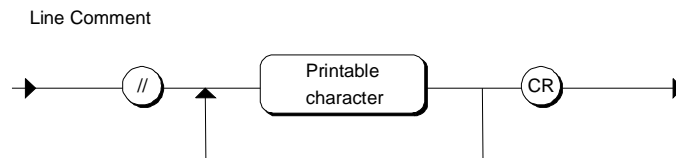
4.14 Line Comment

Rules

- The line comment is introduced by "//" and extends to the end of the line.
- The length of a comment is limited to a maximum of 254 characters including the introductory characters '//'.
- 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.

Syntax

The line comment is represented formally by the following syntax diagram:



Example

```
VAR
    SWITCH : INT ; // line comment
END_VAR
```

Notes

- Comments within the declaration section that begin with // are included in the interface of the block and can therefore also be displayed in the LAD/STL/CSF editor.
 - The printable characters are listed in the section entitled "Language Description".
 - Within the line comment, the pair of characters "(" and ")" have no significance.
-

4.15 Variables

An identifier whose value can change during the execution of a program is called a variable. Each variable must be individually declared 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 scope:

- Local data
- Shared user data
- Permitted predefined variables (CPU memory areas)

Local Data

Local data are declared in a logic block (FC, FB, OB) and have only that logic block as their scope. Specifically these are the following:

Variable	Explanation
Static Variables	Static variables are local variables whose value is retained both during and after execution of the block (block memory). They are used for storing values for a function block.
Temporary Variables	Temporary variables belong to a logic block locally and do not occupy any static memory area. Their values are only retained while the block concerned is running. Temporary variables cannot 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 the actual parameters specified when a block is called.

Shared User Data

These are data or data areas that can be accessed from any point in a program. To use shared 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.

Memory Areas of a CPU

You can access memory areas of a CPU directly using the address identifiers from any point in the program without having to declare these variables.

Remember also that you can always address these memory areas symbolically. The assignment of symbols in this situation is made globally using the symbol table in STEP 7.

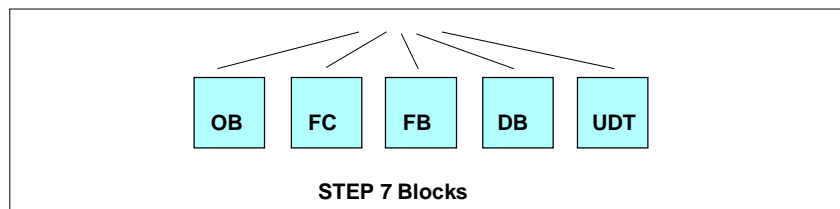
5 SCL Program Structure

5.1 Blocks in SCL Source Files

You can program any number of blocks in an SCL source file. STEP 7 blocks are subunits of a user program distinguished according to their function, their structure or their intended use.

Block Types

The following blocks are available:



Ready-Made Blocks

You do not have to program every function yourself. You can also make use of various ready-made blocks. These are available in the CPU operating system or libraries (*S7lib*) in the STEP 7 Standard Package and can be used, for example, to program communication functions.

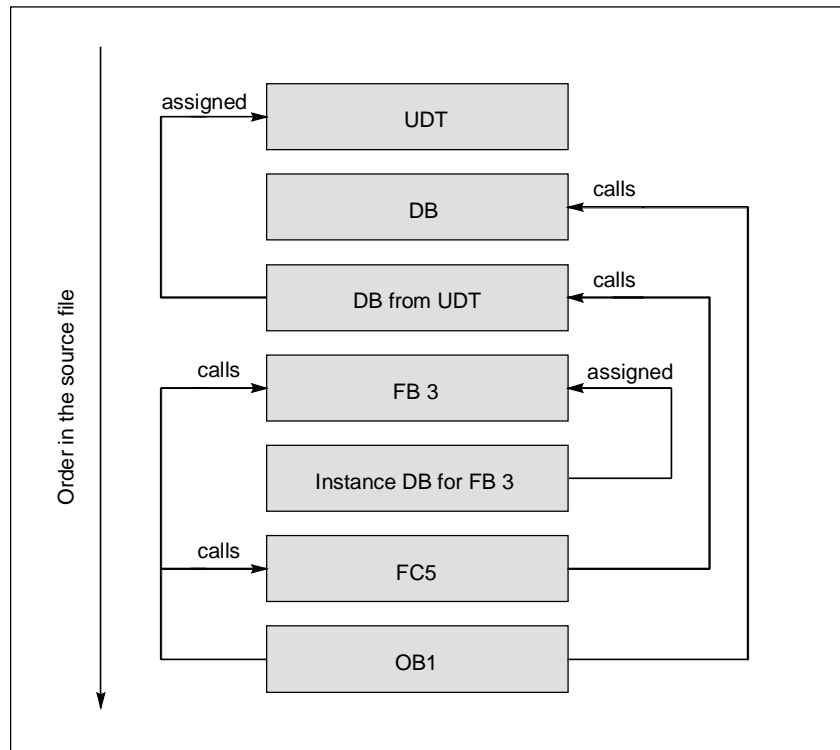
5.2 Order of the Blocks

The following general rule applies:

Called blocks are located before the calling blocks.

Specifically, this means the following:

- 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 that 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 that are called by blocks called in OB1 must precede the calling 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.



5.3 General Structure of a Block

A block consists of the following areas:

- Block start identified by a keyword and a block number or a symbolic block name, for example, "ORGANIZATION_BLOCK OB1" for an organization block. With functions, the function type is also specified. This decides the data type of the return value. If you want no value returned, specify the keyword VOID.
- Optional block title preceded by the keyword "TITLE =".
- Optional block comment. The block comment can extend over several lines, each line beginning with "//".
- Entry of the block attributes (optional)
- Entry of the system attributes for blocks (optional)
- Declaration section (depending on the block type)
- Statement section in logic blocks or assignment of actual values in data blocks (optional)
- In logic blocks: Statements
- Block end indicated by END_ORGANIZATION_BLOCK, END_FUNCTION_BLOCK or END_FUNCTION

5.4 Block Start and End

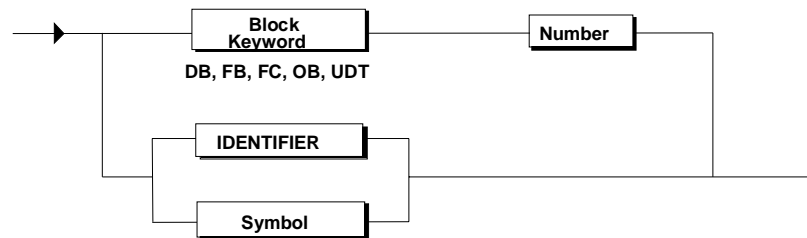
Depending on the type of block, the source text for a single block is introduced by a standard identifier for the start of the block and the block name. It is closed with a standard identifier for the end of the block.

The syntax for the various types of blocks can be seen in the following table:

Identifier	Block Type	Syntax
Function block	FB	FUNCTION_BLOCK fb_name ... END_FUNCTION_BLOCK
Function	FC	FUNCTION fc_name : function type ... END_FUNCTION
Organization block	OB	ORGANIZATION_BLOCK ob_name ... END_ORGANIZATION_BLOCK
Data block	DB	DATA_BLOCK db_name ... END_DATA_BLOCK
Shared data type	UDT	TYPE udt_name ... END_TYPE

Block Name

In the table, *xx_name* stands for the block name according to the following syntax:



The block number can be a value from 0 to 65533, the data block identifier DB0 is, however, reserved.

Please note also that you must define an identifier or a symbol in the STEP 7 symbol table.

Example

```
FUNCTION_BLOCK FB10
FUNCTION_BLOCK Controller Block
FUNCTION_BLOCK "Controller.B1&U2"
```

5.5 Attributes for Blocks

Definition

A block attribute is a block property that you can use, for example, to specify the block type, the version, the block protection or the author. You can display the properties in a STEP 7 properties page when you select blocks for your application.

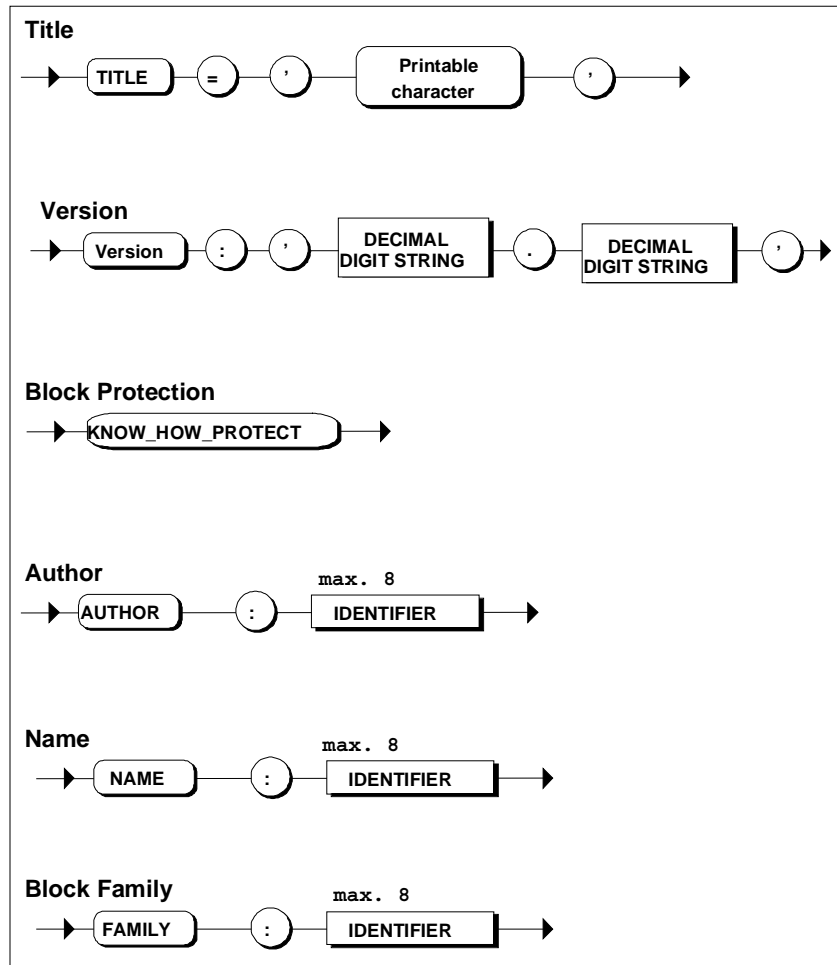
You can assign the following attributes:

Keyword/Attribute	Explanation	Examples
TITLE='printable characters'	Title of the block	TITLE='SORT'
VERSION : 'decimal digit string. decimal digit string'	Version number of the block (0 to 15) Note: With data blocks (DBs), the VERSION attribute is not specified in quotes.	VERSION : '3.1' //With a DB: VERSION : 3.1
KNOW_HOW_PROTECT	Block protection; a block compiled with this option cannot be opened with STEP 7.	KNOW_HOW_PROTECT
AUTHOR :	Name of the author: company name, department name or other name (IDENTIFIER)	AUTHOR : Siemens
NAME :	Block name (IDENTIFIER)	NAME : PID
FAMILY :	Name of the block family: for example motors. This saves the block in a group of blocks so that it can be found again more quickly (IDENTIFIER).	FAMILY : example

Rules

- You declare the block attributes using keywords directly after the statement for the block start.
- The identifier can be up to a maximum of 8 characters long.

The syntax for entering block attributes is shown below:



Examples

```
FUNCTION_BLOCK FB10
TITLE = 'Mean_Value'
VERSION : '2.1'
KNOW_HOW_PROTECT
AUTHOR : AUT_1
```

5.6 Block Comment

You can enter comments for the entire block in the block header after the "TITLE:" line. Here, you use the line comment notation. The comment can extend over several lines, each line beginning with "//".

The block comment is displayed, for example, in the Properties window of the block in the SIMATIC Manager or in the LAD/STL/FBD editor.

Example

```
FUNCTION_BLOCK FB15
TITLE=MY_BLOCK
//This is a block comment.
//It is entered as a series of line comments
//and can be displayed, for example, in the SIMATIC Manager.
AUTHOR...
FAMILY...
```

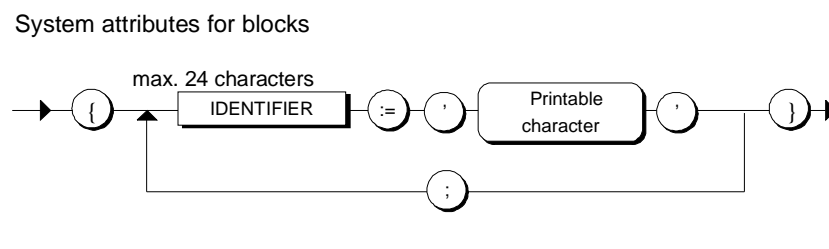
5.7 System Attributes for Blocks

Definition

System attributes are control system attributes valid beyond the scope of a single application. System attributes for blocks apply to the entire block.

Rules

- You specify system attributes immediately after the block start statement.
- The syntax for the entries is shown below:



Examples

```
FUNCTION_BLOCK FB10
{S7_m_c := 'true' ;
S7_blockview := 'big'}
```


5.8 Declaration Section

Definition

The declaration section is used for declarations of local variables, parameters, constants, and labels.

- The local variables, parameters, constants, and labels that must only be valid within a block are defined in the declaration section of the logic block.
- You define the data areas you want to be accessible to any logic block in the declaration section of data blocks.
- In the declaration section of a UDT, you specify the user-defined data type.

Structure

A declaration section is divided into different declaration subsections indicated by their own pair of keywords. Each subsection contains a declaration list for data of the same type. These subsections can be positioned in any order. The following table shows the possible declaration subsections:

Data	Syntax	FB	FC	OB	DB	UDT
Constants	CONST declaration list END CONST	X	X	X		
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 *)		X **)	X **)
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 list END VAR	X	X			

*) Although the declaration of variables between the keyword pair VAR and END_VAR is permitted in functions, the declarations are shifted to the temporary area when the source file is compiled.

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

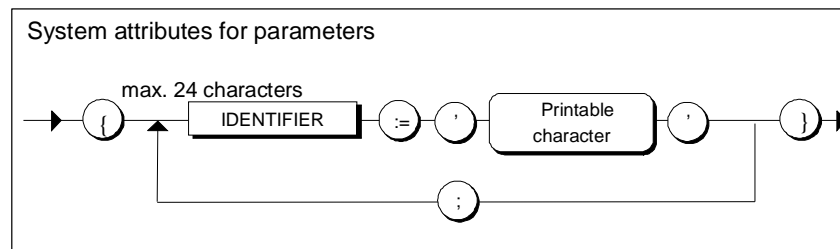
5.9 System Attributes for Parameters

Definition

System attributes are control system attributes valid beyond the scope of a single application. They are used, for example, for configuring messages or connections. System attributes for parameters apply only to the specific parameters that have been configured. You can assign system attributes to input, output and in/out parameters.

Rules

- You assign system attributes for parameters in the declaration fields input parameters, output parameters, or in/out parameters.
- An identifier can have up to a maximum of 24 characters.
- The syntax for the entries is shown below:



Example

```

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

5.10 Statement Section

Definition

The statement section contains statements that will be executed when a logic block is called. These statements are used for processing data or addresses.

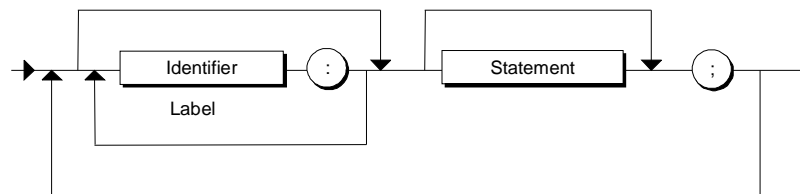
The statement section of a data block contains statements for initializing its variables.

Rules

- If you prefer, you can start the statement section with the BEGIN keyword. BEGIN is mandatory for data blocks. The statement section ends with the keyword for the end of the block.
- Each statement ends with a semicolon.
- Identifiers used in the statement section must already have been declared.
- If required, you can enter a label before each statement.

The syntax for the entries is shown below:

Statement Section



Example

```

BEGIN
    INITIAL_VALUE    :=0;
    FINAL_VALUE     :=200;
    .
    .
STORE:  RESULT      :=SETPOINT;
    .
    .
END_FUNCTION_BLOCK
  
```

5.11 Statements

Definition

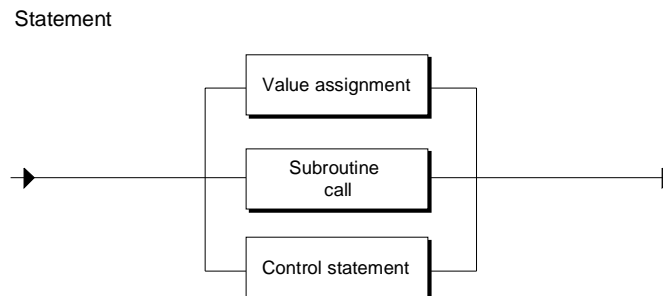
A statement is the smallest self-contained unit the user program. It represents an instruction to the processor to perform a specific operation.

The following types of statement can be used in SCL:

- 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.

Rules

The syntax for the entries is shown below:



Example

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) ;
```

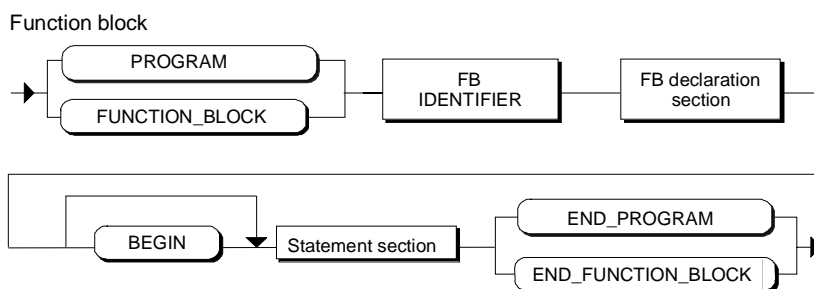
```
// Example of a control statement  
WHILE COUNTER < 10 DO..  
.  
.  
END_WHILE;
```

5.12 Structure of a Function Block (FB)

Definition

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

Syntax



FB Identifier

After the `FUNCTION_BLOCK` or `PROGRAM` keyword, enter the keyword `FB` as the FB identifier followed by the block number or the symbolic name of the FB. The block number can be a value from 0 to 65533.

Examples:

```

FUNCTION_BLOCK FB10
FUNCTION_BLOCK MOTOR1

```

FB Declaration Section

The FB declaration section is used to define the block-specific data. The possible declaration sections are described in detail in the section entitled "Declaration Section". Remember that the declaration section also determines the structure of the assigned instance DB.

Example

The example below 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
    FX1, FX2, FY1, FY2 : REAL ;

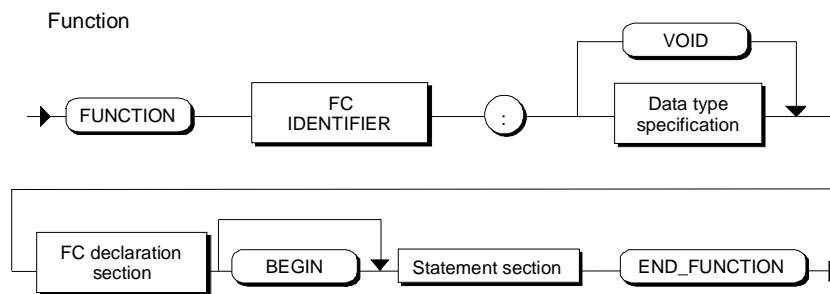
END_VAR
BEGIN
    IF V1 = 7 THEN
        FX1 := 1.5 ;
        FX2 := 2.3 ;
        FY1 := 3.1 ;
        FY2 := 5.4 ;
    //Call function FC11 and supply parameters
    //using the static variables.
        V2 := FC11 (X1:= FX1, X2 := FX2, Y1 := FY1, Y2 := FY2) ;
    END_IF ;
END_FUNCTION_BLOCK
```

5.13 Structure of a Function (FC)

Definition

A function (FC) is a logic block that is not assigned its own memory area. 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.

Syntax



FC Identifier

After the "FUNCTION" keyword, enter the keyword FC as the FC identifier followed by the block number or the symbolic name of the FC. The block number can be a value from 0 to 65533.

Example

```

FUNCTION FC17 : REAL
FUNCTION FC17 : VOID

```

Data Type Specification

The data type specification determines the data type of the return value. All data types are permitted except for STRUCT and ARRAY. No data type needs to be specified if you do not require a return value (using VOID).

FC Declaration Section

The FC declaration section is used to declare the local data (temporary variables, input parameters, output parameters, in/out parameters, constants, labels).

FC Code Section

The function name must be assigned the function result in the code section. This assignment is unnecessary with functions of the type VOID. The following is an example of a valid statement within a function with the name FC31:

```
FC31:= VALUE;
```

Example

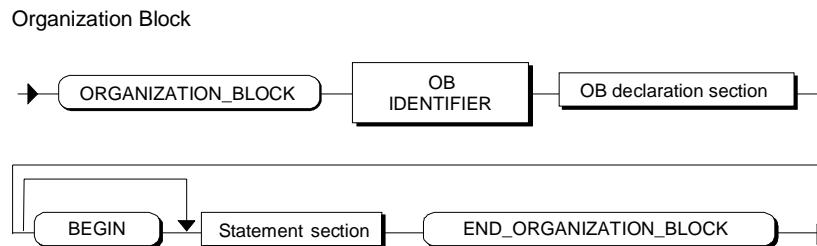
```
FUNCTION FC11: REAL
VAR_INPUT
    x1: REAL ;
    x2: REAL ;
    x3: REAL ;
    x4: REAL ;
END_VAR
VAR_OUTPUT
    Q2: REAL ;
END_VAR
BEGIN
    // Return value from function
    FC11:= SQRT( (x2 - x1)**2 + (x4 - x3) **2 ) ;
    Q2:= x1 ;
END_FUNCTION
```


5.14 Structure of an Organization Block (OB)

Definition

The organization block just like an FB or FC is part of the user program and is called cyclically or as a response to certain events by the operating system. It provides the interface between the user program and the operating system.

Syntax



OB Identifier

After the "ORGANIZATION_BLOCK" keyword, enter the keyword OB as the OB identifier followed by the block number or the symbolic name of the OB. The block number can be a value from 1 to 65533.

Examples

```

ORGANIZATION_BLOCK OB1
ORGANIZATION_BLOCK ALARM

```

OB Declaration Section

The OB declaration section is used to declare the local data (temporary variables, constants, labels).

To execute, each OB always requires 20 bytes of local data for the operating system. You must declare an array for this with an identifier. If you insert the block template for an OB, this declaration is already included.

Example

```

ORGANIZATION_BLOCK OB1
VAR_TEMP
    HEADER : ARRAY [1..20] OF BYTE ; //20 bytes for opsy
END_VAR
BEGIN
    FB17.DB10 (V1 := 7) ;
END_ORGANIZATION_BLOCK

```

5.15 Structure of a Data Block (DB)

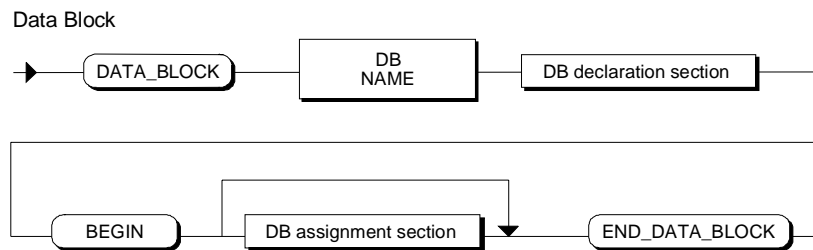
Definition

Shared user-specific data that can be accessed by all blocks in a program are contained in data blocks. Each FB, FC or OB can read or write these data blocks.

There are two types of data blocks:

- **Data blocks**
Data blocks that can be accessed by all logic blocks of the S7-program. Every FB, FC or OB can read or write the data contained in these data blocks.
- **Data blocks assigned to an FB (instance DB)**
Instance data blocks are data blocks that are assigned to a particular function block (FB). These contain the local data for the function block to which they are assigned. These data blocks are created automatically by the SCL compiler when the FB is called in the user program.

Syntax



DB Identifier

After the "DATA_BLOCK" keyword, enter the keyword DB as the DB identifier followed by the block number or the symbolic name of the DB. The block number can be a value from 1 to 65533.

Examples:

```
DATA_BLOCK DB20
```

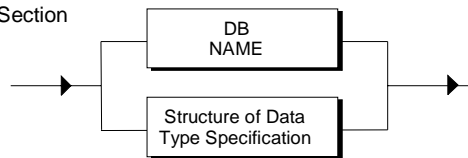
```
DATA_BLOCK MEASRANGE
```

DB Declaration Section

You define the data structure of the DB in the DB declaration section. There are two ways of doing this, as follows:

- **By assigning a user-defined data type**
Here, you can specify the identifier of a user-defined data type defined earlier in the program. The data block then takes the structure of this UDT. You can assign initial values for the variables in the assignment section of the DB.
- **By defining a STRUCT data type**
Within the STRUCT data type specification, you specify the data type for each variable to be stored in the DB and possibly also initial values.

DB Declaration Section



Example

```
DATA_BLOCK DB20
    STRUCT      // Declaration section
        VALUE:ARRAY [1..100] OF INT;
    END_STRUCT

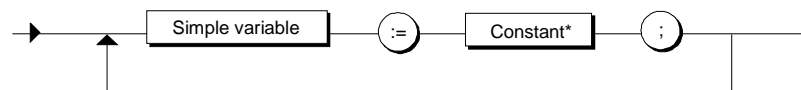
    BEGIN      // Start of assignment section
    :
    END_DATA_BLOCK // End of data block
```

DB Assignment Section

In the assignment section, you can adapt the data you declared in the declaration section so that they have DB specific values for your particular application.

The assignment section begins with the keyword BEGIN and then consists of a sequence of value assignments.

DB Assignment Section



* in STL notation

When assigning initial values (initialization), STL syntax applies to entering attributes and comments. For information on how to write constants, attributes and comments, use the STL online help or refer to the STEP 7 documentation.

Example

```
// Data block with STRUCT data type assigned DATA_BLOCK DB10
STRUCT // Date declaration with initial values
    VALUE      :      ARRAY [1..100] OF INT := 100 (1) ;
    SWITCH     :      BOOL           := TRUE ;
    S_WORD     :      WORD    := W#16#FFAA ;
    S_BYTE     :      BYTE    := B#16#FF ;
    S_TIME     :      S5TIME    := S5T#1h30m10s ;
END_STRUCT

BEGIN // Assignment section
    // Value assignment for specific array elements
    VALUE [1] := 5;
    VALUE [5] := -1 ;

END_DATA_BLOCK

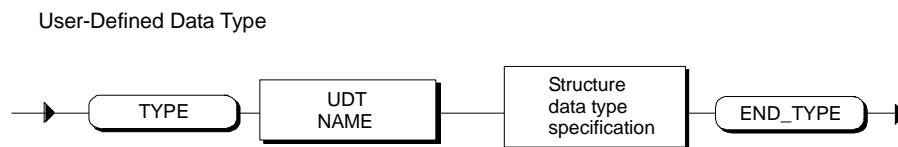
// Data block with user-defined data type assigned
DATA_BLOCK DB11
    UDT 51
BEGIN
END_DATA_BLOCK
```

5.16 Structure of a User-Defined Data Type

User-defined data types (UDTs) are special data structures that you create yourself. Since user-defined data types are assigned names they can be used many times over. Once they have been defined, they can be used at any point in the CPU program; in other words, they are shared data types. They can therefore be used:

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

When using user-defined data types, remember that they are located in the SCL source file before the blocks in which they are used.



UDT Identifier

After the `TYPE` keyword, enter the `UDT` keyword followed by a number or simply the symbolic name of the UDT. The block number can be a value from 0 to 65533.

Examples:

```

TYPE UDT10
TYPE SUPPLYBLOCK

```

Specifying the Data Type

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.

Example of a UDT Definition

```
TYPE MEASVALUES
STRUCT
// UDT definition with symbolic identifier
    BIPOL_1 : INT := 5;
    BIPOL_2 : WORD := W#16#FFAA ;
    BIPOL_3 : BYTE := B#16#F1 ;
    BIPOL_4 : WORD := B#(25,25) ;
    MEASURE : STRUCT
        BIPOLAR_10V : REAL ;
        UNIPOLAR_4_20MA : REAL ;
    END_STRUCT ;
END_STRUCT ;
END_TYPE

//Use of the UDT in an FB
FUNCTION_BLOCK FB10
VAR
    MEAS_RANGE : MEASVALUES;
END_VAR
BEGIN
    // . . .
    MEAS_RANGE.BIPOL_1 := -4 ;
    MEAS_RANGE.MEASURE.UNIPOLAR_4_20MA := 2.7 ;
    // . . .
END_FUNCTION_BLOCK
```

6 Data Types

6.1 Overview of the Data Types in SCL

The data types decide:

- the type and interpretation of the data elements,
- the permitted ranges for the data elements,
- the permitted operations that can be executed on an address of a data type
- the notation of the constants of the data type.

Elementary Data Types

Elementary data types define the structure of data elements that cannot be subdivided into smaller units. They correspond to the definition in the DIN EN 1131-3 standard. An elementary data type describes a memory area with a fixed length and stands for bit, integer, real, time period, time-of-day and character values. The following data types are predefined in SCL.

Group	Data Types	Explanation
Bit Data Types	BOOL BYTE WORD DWORD	Data elements of this type occupy either 1 bit, 8 bits, 16 bits or 32 bits
Character Types	CHAR	Data elements of this type occupy exactly 1 character in the ASCII character set
Numeric Types	INT DINT REAL	Data elements of this type are available for processing numeric values.
Time Types	TIME DATE TIME_OF_DAY S5TIME	Data elements of this type represent the various time and date values in STEP 7.

Complex Data Types

SCL supports the following complex data types:

Data Type	Explanation
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 consisting of structures and arrays.

User-Defined Data Types

You can create your own user-defined data types in the data type declaration. Each one is assigned a unique name and can be used any number of times. Once it has been defined, a user-defined data type can be used to generate a number of data blocks with the same structure.

Parameter Types

Parameter types are special data types for timers, counters and blocks that can be used as formal parameters.

Data Type	Explanation
TIMER	This is used to declare timer functions as parameters.
COUNTER	This is used to declare counter functions as parameters.
BLOCK_xx	This is used to declare FCs, FBs, DBs and SDBs as parameters.
ANY	This is used to allow an address of any data type as a parameter.
POINTER	This is used to allow a memory area as a parameter.

ANY Data Type

In SCL, you can use variables of the ANY data type as formal parameters of a block. You can also create temporary variables of this type and use them in value assignments.

6.2 Elementary Data Types

6.2.1 Bit Data Types

Data of this type are bit combinations occupying either 1 bit (data type BOOL), 8 bits, 16 bits or 32 bits. A numeric range of values cannot be specified for the data types: byte, word, and double word. These are bit combinations that can be used only to formulate Boolean expressions.

Type	Keyword	Bit Width	Alignment	Value Range
Bit	BOOL	1 bit	Begins at the least significant bit in the byte	0, 1 or FALSE, TRUE
Byte	BYTE	8 bits	Begins at the least significant byte in the word.	-
Word	WORD	16 bits	Begins at a WORD boundary.	-
Double word	DWORD	32 bits	Begins at a WORD boundary.	-

6.2.2 Character Types

Data elements of this type occupy exactly one character of the ASCII character set.

Type	Keyword	Bit Width	Value Range
Single character	CHAR	8	Extended ASCII character set

6.2.3 Numeric Data Types

These types are available for processing numeric values (for example for calculating arithmetic expressions).

Type	Keyword	Bit Width	Alignment	Value Range
Integer	INT	16	Begins at a WORD boundary.	-32_768 to 32_767
Double integer	DINT	32	Begins at a WORD boundary.	-2_147_483_648 to 2_147_483_647
Floating-point number (IEEE floating-point number)	REAL	32	Begins at a WORD boundary.	-3.402822E+38 to -1.175495E-38 +/- 0 1.175495E-38 to 3.402822E+38

6.2.4 Time Types

Data of this type represent the various time and date values within STEP 7 (for example for setting the date or for entering the time value for a time).

Type	Keyword	Bit Width	Alignment	Value Range
S5 time	S5TIME S5T	16	Begins at a WORD boundary.	T#0H_0M_0S_10MS to T#2H_46M_30S_0MS
Time period: IEC time in steps of 1 ms.	TIME T	32	Begins at a WORD boundary.	-T#24D_20H_31M_23S_647MS to T#24D_20H_31M_23S_647MS
Date IEC data in steps of 1 day	DATE D	16	Begins at a WORD boundary.	D#1990-01-01 to D#2168-12-31
Time of day time in steps of 1 ms.	TIME_OF_DAY TOD	32	Begins at a WORD boundary.	TOD#0:0:0.0 to TOD#23:59:59.999

If the set value is higher than the upper limit of the range, the upper limit value is used.

With variables of the data type S5TIME, the resolution is limited, in other words, only the time bases 0.01 s, 0.1 s, 1 s, 10 s are available. The compiler rounds the values accordingly. If the set value is higher than the upper limit of the range, the upper limit value is used.

6.3 Complex Data Types

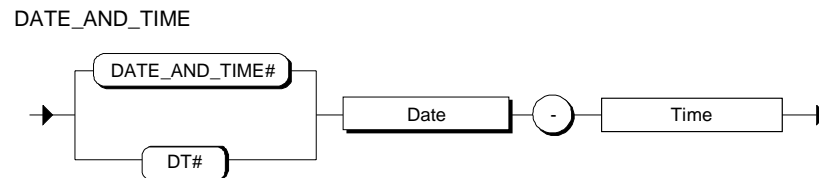
6.3.1 DATE_AND_TIME Data Type

Definition

This data type defines an area with 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.

Syntax



The exact syntax for specifying the date and time is described in "Declaring Constants".

Value Range

Type	Keyword	Bit Width	Alignment	Value Range
Date and time	DATE_AND_TIME DT	64	Begins and ends at a WORD boundary.	DT#1990-01-01-0:0:0.0 to DT#2089-12-31-23:59:59.999

The Date_And_Time data type is stored in BCD format:

Bytes	Content	Range
0	Year	1990 to 2089
1	Month	01 to 12
2	Day	1 to 31
3	Hour	0 to 23
4	Minute	0 to 59
5	Second	0 to 59
6	2 MSD (most significant decade) of ms	00 to 99
7 (4 MSB)	LSD (least significant decade) of ms	0 to 9
7 (4 LSB)	Weekday	1 to 7 (1 = Sunday)

Example

A valid definition for the date and time 20/Oct./1995 12:20:30 and 10 milliseconds is shown below:

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

Note

You can use standard functions (FCs) in the STEP 7 library to access the specific components DATE or TIME.

6.3.2 STRING Data Type

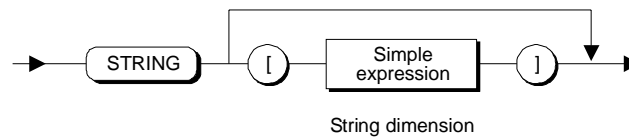
Definition

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 memory area is required to store 254 characters and a header of 2 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.

Syntax

STRING Data Type Specification



The simple expression stands for the maximum number of characters in the STRING. All the characters of the ASCII code are permitted in a character string. A string can also include special characters, for example, control characters and nonprintable characters. You can enter these using the syntax \$hh, where hh stands for the value of the ASCII character expressed in hexadecimal (example: '\$0D\$0AText')

When you declare the memory space for character strings, you can define the maximum number of characters that can be stored in the string. If you do not specify a maximum length, a string with a length of 254 is created.

Example

```

VAR
  Text1      : String [123];
  Text2      : String;
END_VAR
  
```

The constant "123" in the declaration of the variable "Text1" stands for the maximum number of characters in the string. For variable "Text2", a length of 254 characters is reserved.

Note

For output and in/out parameters and for return values of functions, you can reduce the default length (254) reserved for strings to make better use of the resources on your CPU. To reduce the default length, select the menu command **Options > Customize** and enter the required length in the "Maximum String Length" box in the "Compiler" tab. Remember that this setting affects all STRING variables in the source file. The value you set must therefore not be smaller than the STRING variables actually used in the program.

Initializing Character Strings

String variables, just like other variables, can be initialized in the declaration of the parameters of function blocks (FBs) with constant character strings. It is not possible to initialize parameters of functions (FCs).

If the initialized string is shorter than the declared maximum length, the remaining characters are not initialized. When the variable is processed in the program, only the currently occupied character locations are taken into account.

Example

```
x : STRING[7]:= 'Address';
```

If temporary variables of the STRING type are required, for example, for buffering results, they must always be initialized with a string constant either in the variable declaration or in a value assignment before they are used for the first time.

Note

If a function from a standard library returns a value of the STRING data type and if you want this value to be assigned to a temporary variable, the variable must first be initialized.

Example

```
FUNCTION Test : STRING[45]
VAR_TEMP
  x : STRING[45];
END_VAR
x := 'a';
x := concat (in1 := x, in2 := x);
Test := x;
END_FUNCTION
```

Without the initialization **x := 'a';**, the function would return an incorrect result.

Alignment

Variables of the STRING type begin and end at a WORD boundary.

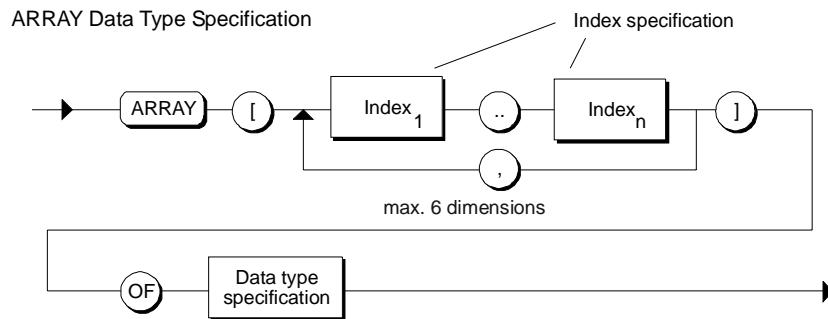
6.3.3 ARRAY Data Type

Definition

ARRAYs have a specified number of components of one data type. The following array types are possible in SCL:

- 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 6.

Syntax



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 periods. The individual index ranges must be separated by commas.
- The entire index specification is enclosed in square brackets.

Data Type Specification

With the data type specification, you declare the data type of the components. Apart from the ARRAY data type, all the data types are permitted for the specification. The data type of an array can, for example, also be a STRUCT type. Parameter types must not be used as the element type for an array.

Example

```
VAR
    CONTROLLER1 :
        ARRAY[1..3,1..4] OF INT:=  -54,   736,  -83,   77,
                                   -1289,10362, 385,    2,
                                   60,   -37,  -7,  103 ;
    CONTROLLER2 : ARRAY[1..10] OF REAL ;
END_VAR
```

Alignment

Variables of the ARRAY type are created row by row. Each dimension of a variable of the type BOOL, BYTE or CHAR ends at a BYTE boundary, all others at a WORD boundary.

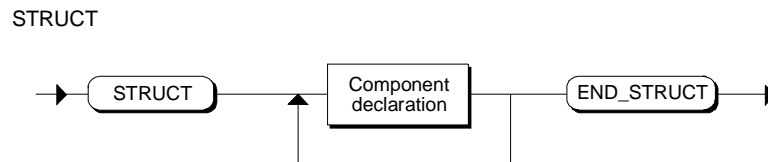
6.3.4 STRUCT Data Type

Definition

The 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 immediately following the STRUCT keyword in the component declaration.

The main feature of the STRUCT data type is that a data element can also be complex. This means that nesting of STRUCT data types is permitted.

Syntax



Component Declaration

The component declaration is a list of the various components of the STRUCT data type. It consists of the following:

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



The identifier is the name of a structure element to which the subsequent data type specification will apply.

All data types with the exception of parameter types are permitted for the data type specification.

You have the option of specifying an initial value for a specific structure element after the data type specification using a value assignment.

Example

```
VAR
    MOTOR : STRUCT
        DATA : STRUCT
            LOADCURR : REAL ;
            VOLTAGE   : INT  := 5 ;
        END_STRUCT ;
    END_STRUCT ;
END_VAR
```

Alignment

Variables of the STRUCT type begin and end at a WORD boundary.

6.4 User-Defined Data Types

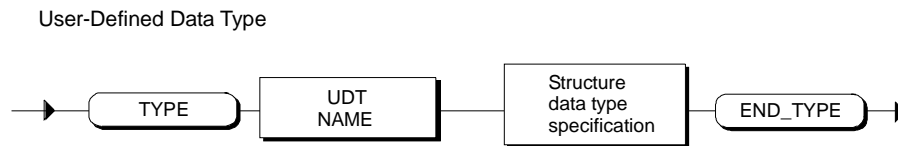
6.4.1 User-Defined Data Types (UDT)

Definition

You define a user-defined data type (UDT) as a block. Once it has been defined, it can be used throughout your user program; in other words, it is a shared data type. You can use these data types with their UDT identifier, UDTx (x represents a number), or with an assigned symbolic name defined in the declaration section of a logic block or data block.

The user-defined data type can be used to declare variables, parameters, data blocks, and other user-defined data types. Components of arrays or structures can also be declared with user-defined data types.

Syntax



UDT Identifier

The declaration of a user-defined data type starts with the TYPE keyword followed by the name of the user-defined data type (UDT identifier). The name of the user-defined data type can either be specified in absolute form; in other words, a standard name in the form UDTx (x stands for a number), or as a symbolic name.

Examples:

```

TYPE UDT10
TYPE MEASVALUES
  
```

Data Type Specification

The UDT identifier is followed by the data type specification. The only data type specification permitted in this case is STRUCT.

```
STRUCT
:
END_STRUCT
```

Note

The syntax of STL must be used within a user-defined data type. This applies, for example, to the notation for constants and the assignment of initial values (initialization). For information about the syntax of the constants, refer to the [STL online help](#).

Example

```
// UDT definition with a symbolic name
TYPE MEASVALUES:
  STRUCT
    BIPOL_1 : INT := 5;
    BIPOL_2 : WORD := W#16#FFAA ;
    BIPOL_3 : BYTE := B#16#F1 ;
    BIPOL_4 : WORD := W#16#1919 ;
    MEASURE : STRUCT
      BIPOLAR_10V : REAL ;
      UNIPOLAR_4_20MA : REAL ;
    END_STRUCT;
  END_STRUCT;
END_TYPE

//Use of the UDT in an FB
FUNCTION_BLOCK FB10
VAR
  MEAS_RANGE : MEASVALUES;
END_VAR
BEGIN
  // . . .
  MEAS_RANGE.BIPOL_1 := -4 ;
  MEAS_RANGE.MEASURE.UNIPOLAR_4_20MA := 2.7 ;
  // . . .
END_FUNCTION_BLOCK
```

6.5 Data Types for Parameters

6.5.1 Data Types for Parameters

To specify the formal block parameters of FBs and FCs, you can use parameter types in addition to the data types that have already been introduced.

Parameter	Size	Description
TIMER	2 bytes	Identifies a specific timer to be used by the program in the logic block called. Actual Parameter for example, T1
COUNTER	2 bytes	Identifies a specific counter to be used by the program in the logic block called. Actual Parameter for example, 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: for example, 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: for example, M50.0

6.5.2 TIMER and COUNTER Data Types

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

6.5.3 BLOCK Data Types

You specify a block that will be used as an input parameter. The declaration of the input parameter determines the block type (FB, FC, DB). For supplying parameters, you specify the block identifier. Both absolute and symbolic identifiers are permitted.

You can access the BLOCK_DB data type using absolute addressing (`myDB.dw10`). SCL does not provide any operations for the other block data types. Parameters of this type can only be supplied with values when the blocks are called. When using functions, input parameters cannot be passed on.

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

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

6.5.4 POINTER Data Type

You can assign variables to the POINTER data type that you have declared as formal parameters of a block. If you call such a block, these parameters can be supplied with addresses of any data type (except ANY).

SCL, however, only provides one statement for processing the POINTER data type, namely passing on to underlying blocks.

You can assign the following types of addresses as actual parameters:

- Absolute addresses
- Symbolic names
- Addresses of the POINTER data type
This is only possible when the address is a formal parameter with a compatible parameter type.
- NIL constant
You specify a nil pointer.

Restrictions

- The POINTER data type is permitted for formal input parameters, in/out parameters of FBs and FCs and for output parameters of FCs. Constants are not permitted as actual parameters (with the exception of the NIL constant).
- If you supply a formal parameter of the type POINTER with a temporary variable when an FB or FC is called, you cannot pass this parameter on to a further block. Temporary variables lose their validity when they are passed on.

Example

```
FUNCTION FC100 : VOID
VAR_IN_OUT
    N_out : INT;
    out   : POINTER;
END_VAR
VAR_TEMP
    ret   : INT;
END_VAR
BEGIN
    // ...
    ret := SFC79(N := N_out, SA := out);
    // ...
END_FUNCTION

FUNCTION_BLOCK FB100
VAR
    ii   : INT;
    aa : ARRAY[1..1000] OF REAL;
END_VAR

BEGIN
    // ...
    FC100( N_out := ii, out := aa);
    // ...
END_FUNCTION_BLOCK
```

6.6 ANY Data Type

In SCL, you can declare variables of the ANY data type as follows:

- As formal parameters of a block; these parameters can then be supplied with actual parameters of any data type when the block is called.
- As temporary variables; you can assign values of any data type to these variables.

You can use the following data as the actual parameters or as a value assignment on the right-hand side:

- Local and shared variables
- Variables in the DB (addressed absolutely or symbolically)
- Variables in the local instance (addressed absolutely or symbolically)
- NIL constant
You specify a nil pointer.
- ANY data type
- Timers, counters, and blocks
You specify the identifier (for example, T1, C20 or FB6).

Restrictions

- The ANY data type is permitted for formal input parameters, in/out parameters of FBs and FCs and for output parameters of FCs. Constants are not permitted as the actual parameters or on the right-hand side of a value assignment (with the exception of the NIL constant).
- If you supply a formal parameter of the type ANY with a temporary variable when an FB or FC is called, you cannot pass this parameter on to a further block. Temporary variables lose their validity when they are passed on.
- Variables of this type must not be used as a component type in a structure or as an element type for an array.

6.6.1 Example of the ANY Data Type

```
VAR_INPUT
    iANY : ANY;
END_VAR

VAR_TEMP
    pANY : ANY;
END_VAR

CASE ii OF
1:
    pANY := MW4;          // pANY contains the address of MW4

3..5:
    pANY:= aINT[ii];      //pANY contains the address of the ii th
                          // element of the aINT field;
100:
    pANY:= iANY; //pANY contains the value of the iANY input
    variable
ELSE
    pANY := NIL;          // pANY contains the value of the NIL
    pointer
END_CASE;

SFCxxx(IN := pANY);
```


7 Declaring Local Variables and Parameters

7.1 Local Variables and Block Parameters

Categories of Variables

The following table illustrates the categories of 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 save values of 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 value is only retained while the block concerned is running. Temporary variables cannot be accessed from outside the block in which they are declared.

Categories of Block Parameters

Block parameters are placeholders that are only assigned a specific value when the block is called. The placeholders in the block are known as 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 like local variables.

Block parameters can be subdivided into the categories shown below:

Block Parameter	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 adopt current input values when a block is called. After processing the value, they receive the result and return it to the calling block.

Flags (OK Flag)

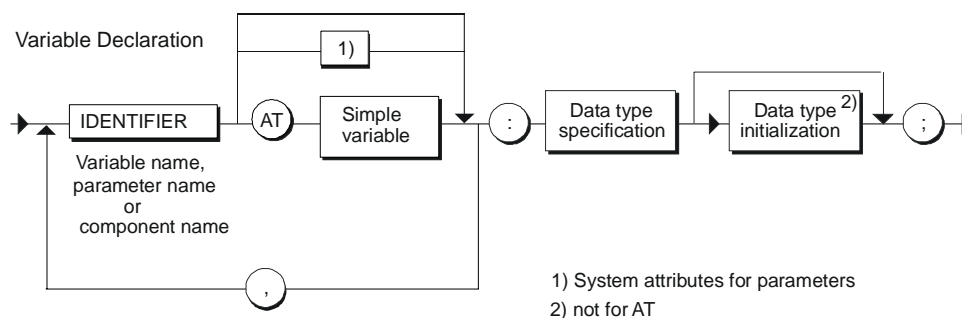
The SCL compiler provides a flag that can be used to detect errors when programs are running on the CPU. It is a local variable of the type BOOL with the predefined name "OK".

7.2 General Syntax of a Variable or Parameter Declaration

Variables and block parameters must be declared individually before they can be used within a logic block or data block. The declaration specifies that an identifier is used as a block parameter or variable and assigns it a data type.

A variable or parameter declaration consists of an identifier (named by the user) and a data type. The basic format is shown in the syntax diagram below.

Syntax of a Variable or Parameter Declaration



Examples

```
VALUE1 : REAL;
if there are several variables of the same type:

VALUE2, VALUE3, VALUE4, ....: INT;
ARR : ARRAY[1..100, 1..10] OF REAL;
SET : STRUCT
    MEASARR:ARRAY[1..20] OF REAL;
    SWITCH:BOOL;
END_STRUCT
```

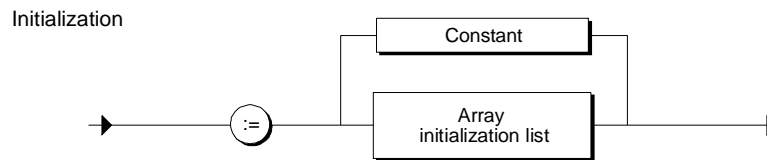
Note

If you want to use reserved words as identifiers, they must be preceded by the "#" character (for example, #FOR).

7.3 Initialization

Static variables as well as input and output parameters of an FB can be assigned an initial value when they are declared. In/out parameters can also be assigned an initial value, however, only if they are of an elementary data type. With simple variables, the initial value is assigned by assigning (:=) a constant after the data type specification.

Syntax



Example

```
VALUE      :REAL := 20.25;
```

Note

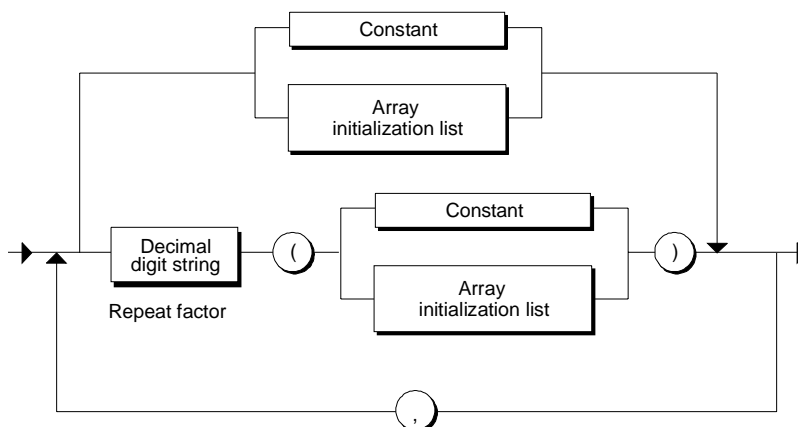
Initialization of a variable list (A1, A2, A3,...: INT:=...) is not possible. In such cases, the variables have to be initialized individually.

Array Initialization

To initialize ARRAYS, you can either specify a value for each component separated by a comma, or by specifying a repetition factor (integer) you can initialize several components with the same value.

Syntax of Array Initialization

Array Initialization List



Examples

```

VAR
// Initialization of static variables:
INDEX1 : INT := 3 ;
//Array initialization:
CONTROLLER1 : ARRAY [1..2, 1..2] OF INT := -54, 736, -83,
77;
CONTROLLER2 : ARRAY[1..10] OF REAL := 10(2.5);
//Structure initialization:
GENERATOR: STRUCT
    DAT1 :    REAL                := 100.5;
    A1 :     INT                  := 10 ;
    A2 :     STRING[6]            := 'FACTOR';
    A3 :     ARRAY[1..12] OF REAL := 0.0, 10(100.0), 1.0;
END_STRUCT ;
END_VAR

```

7.4 Declaring Views of Variable Ranges

To be able to access a declared variable with a different data type, you can define views of the variable or of ranges within the variables using the "AT" keyword. A view is visible only locally in the block; it is not included in the interface. A view can be used like any other variable in the block. It inherits all the properties of the variable that it references; only the data type is new.

Example

The following example makes several views of one input parameter possible:

```
VAR_INPUT
    Buffer : ARRAY[0..255] OF BYTE;
    Frame1 AT Buffer : UDT100 ;
    Frame2 AT Buffer : UDT200 ;
END_VAR
```

The calling block supplies the Buffer parameter, it does not see the names Frame1 and Frame2. The calling block now has three ways of interpreting the data, namely the array under the name buffer or with a different structure under Frame1 or Frame2.

Rules

- The declaration of a further view of a variable must be made following the declaration of the variable to which it points in the same declaration subsection.
- Initialization is not possible.
- The data type of the view must be compatible with the data type of the variable. The variable specifies the size of the memory area. The memory requirements of the view can be equal to this or smaller. The following rules for combining data types also apply:

		Data Type of the View	Data Type of the Variable		
			Elementary	Complex	ANY/POINTER
FB	Declaration of a view in VAR, VAR_TEMP, VAR_IN or VAR_OUT	Elementary Complex ANY/POINTER	x x	x x x (1)	x (1)
	Declaration of a view in VAR_IN_OUT	Elementary Complex ANY/POINTER	x	x	
FC	Declaration of a view in VAR or VAR_TEMP	Elementary Complex ANY/POINTER	x x	x x x	x
	Declaration of a view in VAR_IN, VAR_OUT or VAR_IN_OUT	Elementary Complex ANY/POINTER	x	x	

(1) ANY pointer not permitted in VAR_OUT.

Elementary = BOOL, BYTE, WORD, DWORD, INT, DINT, DATE, TIME, S5TIME, CHAR

Complex = ARRAY, STRUCT, DATE_AND_TIME, STRING

7.5 Using Multiple Instances

It is possible that you may want to or have to use a restricted number of data blocks for instance data owing to the performance (for example, memory capacity) of the S7 CPUs you are using. If other existing function blocks are called in an FB in your user program (call hierarchy of FBs), you can call these other function blocks without their own (additional) instance data blocks.

Use the following solution:

- Include the function blocks you want to call as static variables in the variable declaration of the calling function block.
- In this function block, call other function blocks without their own instance data block.
- This concentrates the instance data in one instance data block, allowing you to use the available number of data blocks more effectively.

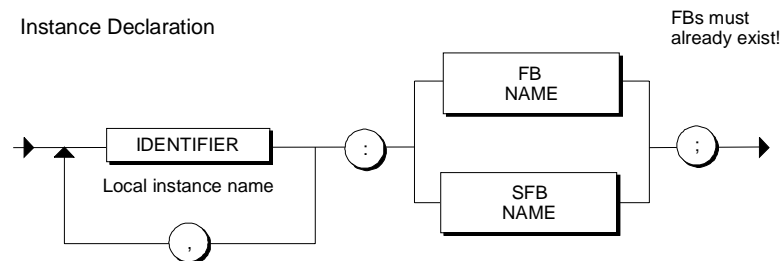
7.6 Instance Declaration

For function blocks, you can also declare variables of the type FB or SFB in the declaration subsection for static variables (VAR; END_VAR) in addition to the variables with elementary, complex or user-defined data types. 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. A local instance-specific initialization is not possible.

Blocks called as a local instance must not have the length 0. At least one static variable or a parameter must be declared in such blocks.

Syntax



Example

```

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

Where Motor is a symbol for an FB entered in the symbol table .

7.7 Flags (OK Flag)

The OK flag is used to indicate the correct or incorrect execution of a block. It is a local variable of the type BOOL with the predefined name "OK".

At the beginning of the program, the OK flag has the value TRUE. It can be queried at any point in the block or can be set to TRUE / FALSE using SCL statements. If an error occurs while an operation is being executed (for example division by zero), the OK flag is set to FALSE. When the block is exited, the value of the OK flag is saved in the output parameter ENO and can be evaluated by the calling block.

Declaration

The OK flag is a system variable. Declaration is not necessary. You must, however, select the compiler option "Set OK flag" before compilation if you want to use the OK flag in your user program.

Example

```
// Set OK flag to TRUE
// to check whether the
// action executes correctly.
OK:= TRUE;
Division:= 1 / IN;
IF OK THEN
    // Division was correct.

    // :
    // :

ELSE    // Division was not correct.

    // :

END_IF;
```

7.8 Declaration Subsections

7.8.1 Overview of the Declaration Subsections

Each category of local variables or parameters has its own declaration subsection identified by its own pair of keywords. Each subsection contains the declarations that are permitted for that particular declaration subsection. These subsections can be positioned in any order.

The following table shows which variables or types of parameter you can declare in the various logic blocks:

Data	Syntax	FB	FC	OB
Variable as: Static variable	VAR ... END_VAR	X	X *)	
Temporary variable	VAR_TEMP ... END_VAR	X	X	X
Block parameter as: Input parameter	VAR_INPUT ... END_VAR	X	X	
Output parameter	VAR_OUTPUT ... END_VAR	X	X	
In/out parameter	VAR_IN_OUT ... END_VAR	X	X	

*) Although the declaration of variables between the keyword pair VAR and END_VAR is permitted in functions, the declarations are created in the temporary area when the source file is compiled.

7.8.2 Static Variables

Static variables are local variables whose value is retained when the blocks are run. They are used to save the values of a function block and are contained in a corresponding instance data block.

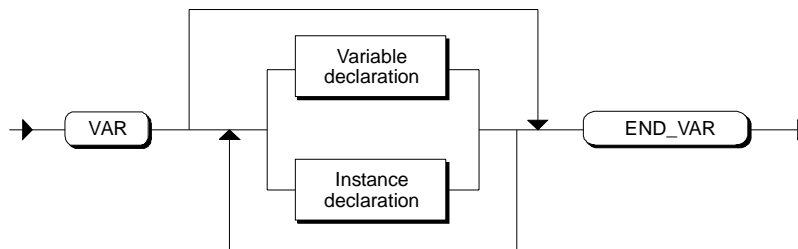
Syntax

Static variables are declared in the VAR / END_VAR declaration section. This declaration subsection is part of the FB declaration section. After compilation, this subsection and the subsections for the block parameters decide the structure of the assigned instance data block.

In this subsection you can:

- Create variables, assign data types to the variables and initialize the variables.
- Declare a called FB as a static variable if you want to call it in the current FB as a local instance.

Static Variable Block



Example

```
VAR
RUN           : INT;
MEASARR       : ARRAY [1..10] OF REAL;
SWITCH        : BOOL;
MOTOR_1,MOTOR_2 :FB100;  //Instance declaration

END_VAR
```

Access

The variables are accessed from the code section as follows:

- **Access within the block:** In the code section of the function block in which a variable was declared in the declaration section, you can access the variable. This is explained in detail in the section entitled "Value Assignment".
- **External access using the instance DB:** You can access the variable from other blocks using indexed access, for example *DBx.variable*.

7.8.3 Temporary Variables

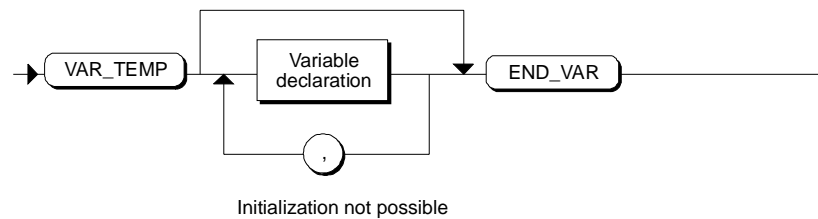
Temporary variables belong locally to a logic block and do not occupy a static memory area. They are located in the stack of the CPU. Their value is only retained while the block concerned is running. Temporary variables cannot be accessed from outside the block in which they are declared. When an OB, FB or FC is first executed, the value of the temporary data has not been defined. Initialization is not possible.

You should declare data as temporary data if you only require it to record interim results while your OB, FB or FC executes.

Syntax

Temporary variables are declared in the VAR_TEMP / END_VAR declaration section. This declaration subsection is part of an FB, FC, or OB. It is used to declare variable names and data types within the variable declaration.

Temporary Variable Subsection



Example

```

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

Access

A variable is always accessed from the code section of the logic block in which the variable is declared in the declaration section (internal access). Refer to the section entitled "Value Assignment".

7.8.4 Block Parameters

Parameters are placeholders that are only assigned a value when the block is actually called. The placeholders declared in the block are known as formal parameters that are assigned values as actual parameters. Parameters therefore provide a mechanism for the exchange of information between the blocks.

Types of Block Parameters

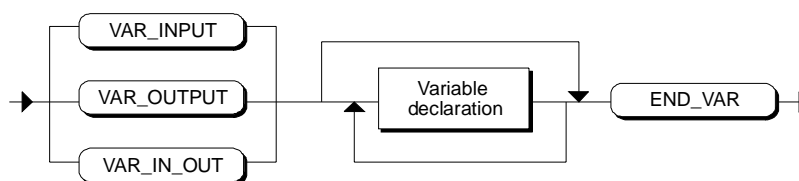
- Formal input parameters are assigned the actual input values (data flow into the block).
- Formal output parameters are used to transfer output values (data flow from the block to the outside)
- Formal in/out parameters have both the function of an input and an output parameter.

Syntax

The declaration of formal parameters is made in the declaration section of a function block or a function grouped according to parameter type in the three declaration subsections for parameters. Within the variable declaration, you specify the parameter name and the data type. Initialization is only possible for the input and output parameters of an FB.

When declaring formal parameters, you can use not only elementary, complex, and user-defined data types but also the data types for parameters.

Parameter Subsection



Initialization only possible for VAR_INPUT and VAR_OUTPUT

Example

```
VAR_INPUT                // Input parameters
    MY_DB      : BLOCK_DB ;
    CONTROLLER : DWORD  ;
    TIMEOFDAY  : TIME_OF_DAY ;
END_VAR

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

VAR_IN_OUT              // In_out parameters
    SETTING : INT ;
END_VAR
```

Access

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

- **Internal access:** Access from the code section of the block in whose declaration section the parameter is declared. This is explained in the sections entitled "Value Assignment" and "Expressions, Operations and Addresses".
- **External access using an instance data block:** You can access block parameters of function blocks using the assigned instance DB.

8 Declaring Constants and Jump Labels

8.1 Constants

Constants are data elements that have a fixed value that cannot change while the program is running.

The following groups of constants can be used in SCL.

- Bit constants
- Numeric constants
 - Integer constants
 - Real-number constants
- Character constants
 - Char constants
 - String constants
- Times
 - Date constants
 - Time period constants
 - Time-of-day constants
 - Date and time constants

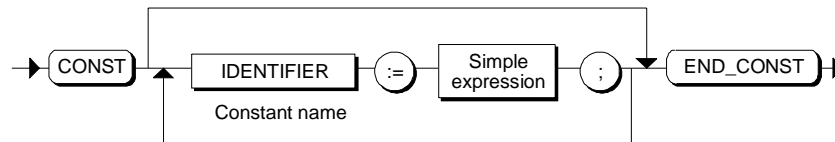
8.1.1 Declaring Symbolic Names for Constants

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

You can declare symbolic names for constants using the CONST statement in the declaration section of your logic block. This is advisable for all constants of a block. With this method, the block is easier to read and update if you want to make changes to constant values.

Syntax

Constant Subsection



In simple expressions, only the seven basic arithmetic operations are permitted (*, /, +, -, **, DIV, MOD).

Example

```

CONST
  Number           := 10 ;
  TIMEOFDAY1       := TIME#1D_1H_10M_22S_2MS ;
  NAME             := 'SIEMENS' ;
  NUMBER2          := 2 * 5 + 10 * 4 ;
  NUMBER3          := 3 + NUMBER2 ;
END_CONST
  
```


8.1.3 Notation for Constants

There is a specific notation or format for the value of a constant depending on its data type and data format. The type and value of a constant is decided directly by the notation and does not need to be declared.

Examples:

15	VALUE 15	as integer constant in decimal format
2#1111	VALUE 15	as integer constant in binary format
16#F	VALUE 15	as integer constant in hexadecimal format

Overview of the Possible Notations

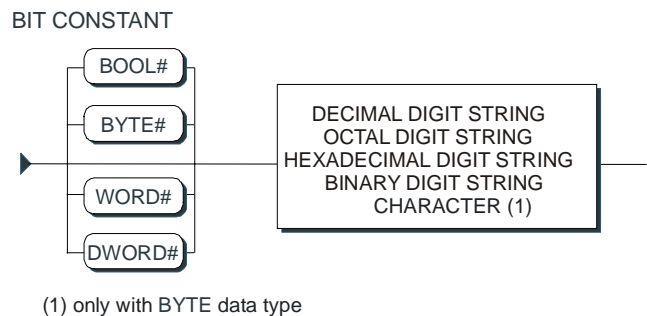
Data Type	Description	Example in SCL	Examples in STL (where different)
BOOL	Bit 1	FALSE TRUE BOOL#0 BOOL#1 BOOL#FALSE BOOL#TRUE	
BYTE	8-bit hexadecimal number	B#16#00 B#16#FF BYTE#0 B#2#101 Byte#'ä' b#16#f	
CHAR	8-bit (1 ASCII character)	'A' CHAR#49	
STRING	Maximum of 254 ASCII characters	'Address'	
WORD	16-bit hexadecimal number 16-bit octal number 16-bit binary number	W#16#0000 W#16#FFFF word#16# WORD#8#177777 8#177777 W#2#1001_0100 WORD#32768	
DWORD	32-bit hexadecimal number 32-bit octal number 32-bit binary number	DW#16#0000_0000 DW#16#FFFF_FFFF Dword#8#3777777777 8#3777777777 DW#2#1111_0000_1111_0000 dword#32768	
INT	16-bit	-32768	

Data Type	Description	Example in SCL	Examples in STL (where different)
	fixed-point number	+32767 INT#16#3f_ff int#-32768 Int#2#1111_0000 inT#8#77777	
DINT	32-bit fixed-point number	-2147483648 +2147483647 DINT#16#3fff_ffff dint#-1000_0000 Dint#2#1111_0000 dinT#8#1777777777	L#-2147483648 L#+2147483647
REAL	32-bit floating-point number	Decimal format 123.4567 REAL#1 real#1.5 Exponential format real#2e4 +1.234567E+02	
S5TIME	16-bit time value in SIMATIC format	T#0ms TIME#2h46m30s T#0.0s TIME#24.855134d	S5T#0ms S5TIME#2h46m30s
TIME	32-bit time value in IEC format	T#-24d20h31m23s647ms TIME#24d20h31m23s647ms T#0.0s TIME#24.855134d	
Date	16-bit date value	D#1990-01-01 DATE#2168-12-31	
TIME_OF_DAY	32-bit time of day	TOD#00:00:00 TIME_OF_DAY#23:59:59.999	
DATE_AND_ TIME	Date and time value	DT#95-01-01-12:12:12.2	

8.1.3.1 Bit Constants

Bit constants contain values with the length 1 bit, 8 bits, 16 bits or 32 bits. Depending on their length, these can be assigned to variables in the SCL program with the data types BOOL, BYTE, WORD or DWORD.

Syntax



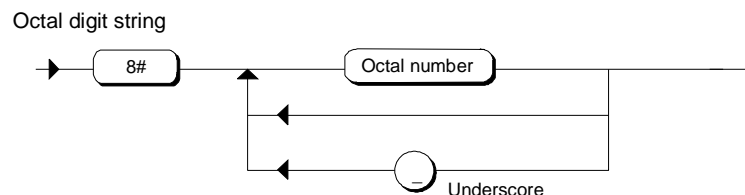
Decimal Digit String

The decimal number in a constant consists of a string of digits (if required, these can be separated by underscores). The underscores are used to improve readability in the case of long numbers. Examples of valid notations for decimal digit strings in constants are shown below:

```
DW#2#1111_0000_1111_0000
dword#32768
```

Binary, Octal and Hexadecimal Values

You can specify an integer constant 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. This is illustrated in the figure below based on the example of a digit string for an octal number:



Example

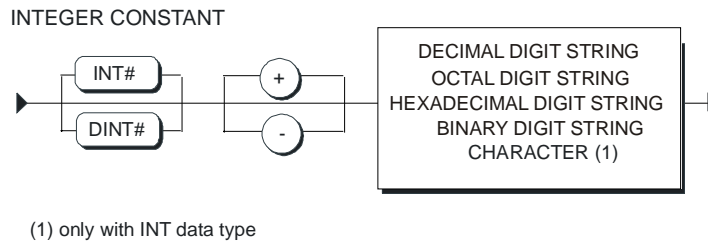
The following examples illustrate the notations for bit constants:

```
Bool#false
8#177777
DW#16#0000_0000
```

8.1.3.2 Integer Constants

Integer constants contain whole number values with a length of 16 bits or 32 bits. Depending on their length, these can be assigned to variables with the data types INT or DINT in the SCL program.

Syntax



Decimal Digit String

The decimal number in a constant consists of a string of digits (if required, these can be separated by underscores). The underscores are used to improve readability in the case of long numbers. Examples of valid notations for decimal digit strings in constants are shown below:

```
1000
1_120_200
666_999_400_311
```

Binary, Octal and Hexadecimal Values

You can specify an integer constant 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.

Example

The following examples illustrate the notations for integer constants:

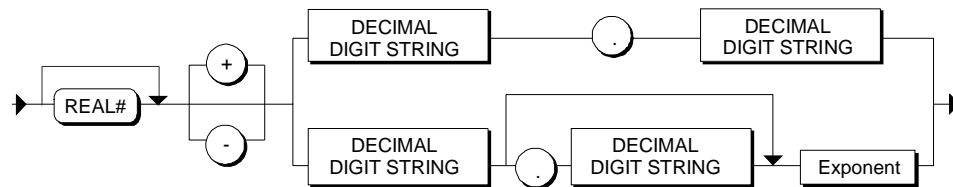
```
Value_2:=2#0101;      // Binary number, decimal value 5
Value_3:=8#17;        // Octal number, decimal value 14
Value_4:=16#F;        // Hexadecimal number, decimal value 15
Value_5:=INT#16#3f_ff // Hexadecimal number, type-defined
                        notation
```

8.1.3.3 Real Number Constants

Real number constants are values with decimal places. They can be assigned to variables with the data type REAL.

Syntax

REAL NUMBER CONSTANT



The use of a plus or minus sign is optional. If no sign is specified, the number is assumed to be positive.

The decimal number in a constant consists of a string of digits (if required, these can be separated by underscores). The underscores are used to improve readability in the case of long numbers. Examples of valid notations for decimal digit strings in constants are shown below:

```

1000
1_120_200
666_999_400_311
  
```

Exponent

When specifying floating-point numbers, you can use an exponent. The exponent is specified by the letter "E" or "e" followed by an integer value.

The value 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

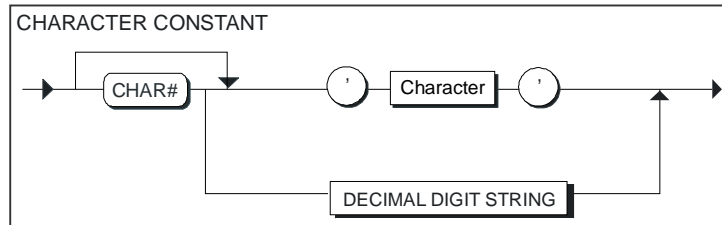
```

NUM4 := -3.4 ;
NUM5 := 4e2 ;
NUM6 := real#1.5;
  
```


8.1.3.4 Char Constants (Single Characters)

The char constant contains exactly one character. The character is enclosed in single quotes ('). Char constants cannot be used in expressions.

Syntax



Example

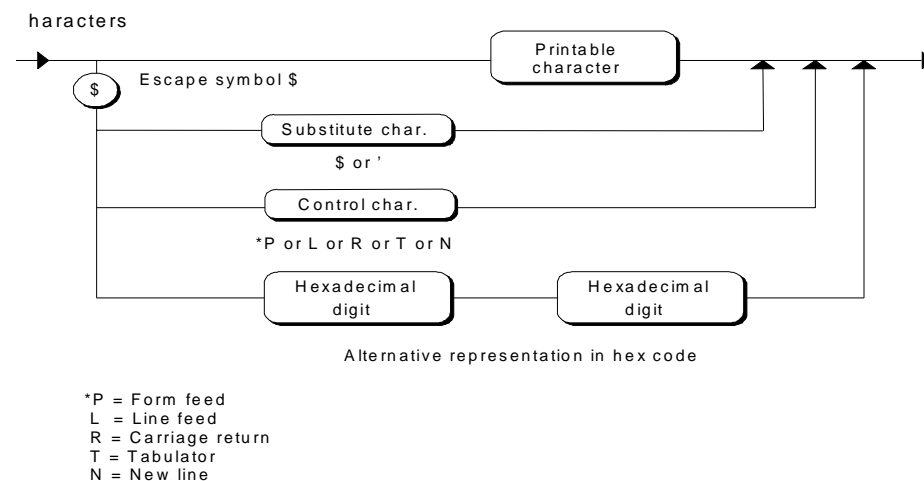
```

Charac_1 := 'B';
Charac_2 := char#43;
Charac_3 := char#'B';
  
```

Syntax of a Character

Any character in the complete, extended ASCII character set can be used. Special formatting characters, the quote (') or a \$ character can be entered using the escape symbol \$.

You can also use the nonprintable characters from the complete, extended ASCII character set. To do this, you specify the substitute representation in hexadecimal.



Example of a Character in Hexadecimal Code

```

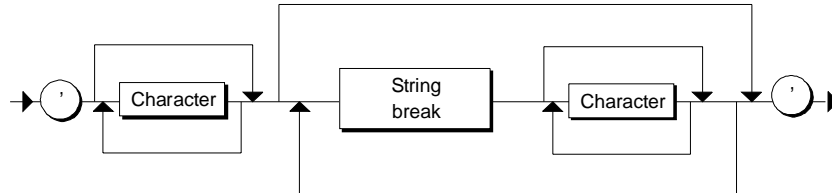
CHARACTER      := '$41' ; //Corresponds to the character 'A'
Blank          := '$20' ; //Corresponds to the character ' '
  
```

8.1.3.5 String Constants

A string constant is a character string with a maximum of 254 characters. The characters are enclosed in single quotes. String constants cannot be used in expressions.

Syntax

STRING CONSTANT

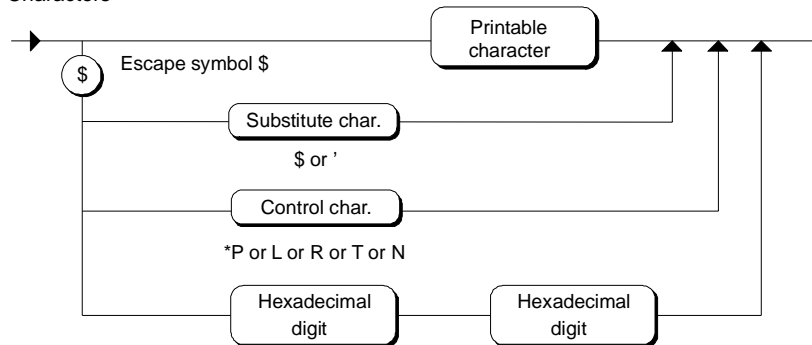


Syntax of a Character

Any character in the complete, extended ASCII character set can be used. Special formatting characters, the quote (') or a \$ character can be entered using the escape symbol \$.

You can also use the nonprintable characters from the complete, extended ASCII character set. To do this, you specify the substitute representation in hexadecimal code.

Characters



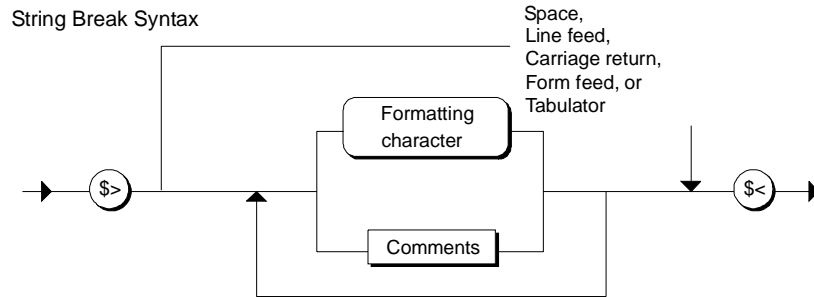
Alternative representation in hex code

*P = Form feed
 L = Line feed
 R = Carriage return
 T = Tabulator
 N = New line

Interrupting a String

You can interrupt and resume a string constant several times.

A string is located either in a line of an SCL block or is spread over several lines using special identifiers. To interrupt a string, you use the `$>` identifier and to continue it in a later line, you use the `$<` identifier. The space between the interrupt and the resume identifiers can extend over several lines and can contain either comments or blanks.



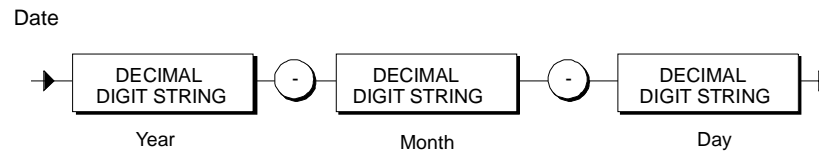
Examples

```
// String constant:
NAME:= 'SIEMENS';
//Interrupting a string constant
MESSAGE1:= 'MOTOR- $>
$< Controller';
// string in hexadecimal:
MESSAGE1:= '$41$4E' (*character string AN*);
```

8.1.3.6 Date Constants

A date is introduced by the prefixes DATE# or D# . The date is specified by integers for the year (4 digits), the month and the day, separated by dashes.

Syntax



Example

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

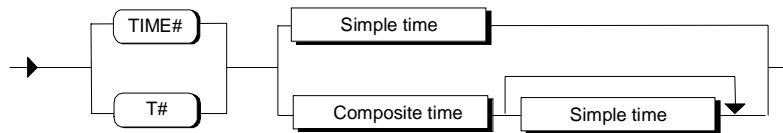
8.1.3.7 Time Period Constants

A period of time is introduced by the prefixes TIME# or T#. The time period can be expressed in two possible ways:

- Decimal format
- Composite Format

Syntax

TIME PERIOD



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

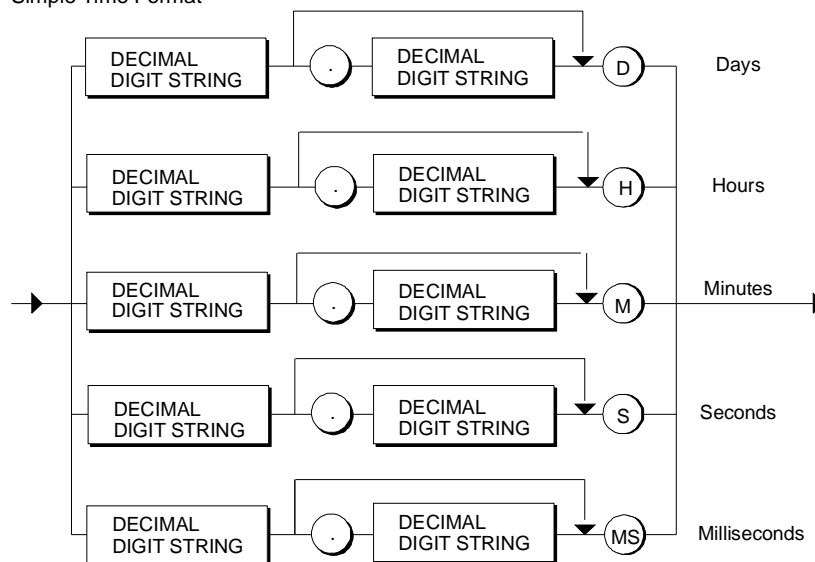
A change from composite format to decimal format is only possible when the time units have not yet been specified.

Following the introductory prefixes T# or TIME#, you must specify at least one time unit.

Decimal Format

You use the decimal format if you want to specify time components such as hours or minutes as a decimal number.

Simple Time Format

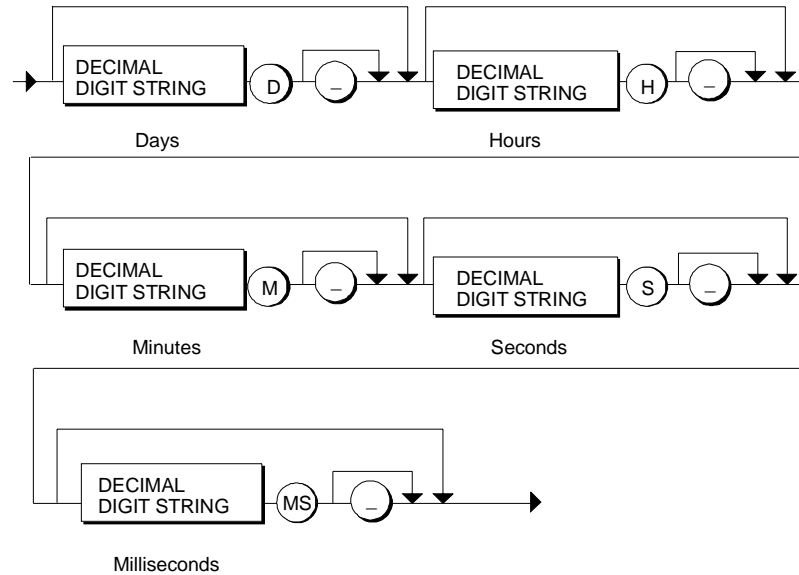


Use of the simple time format is only possible for undefined time units.

Composite format

The composite format is a sequence of individual time components. First days and then hours etc. are specified separated by the underscore character. You can, however, omit components from the sequence. However, at least one time unit must be specified.

Composite Time Format



Example

```
// Decimal format
Interval1:= TIME#10.5S ;

// Composite format
Interval2:= T#3D_2S_3MS ;

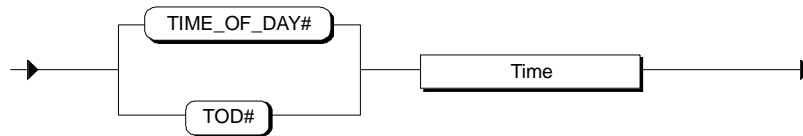
// Composite and decimal format
Interval3 := T#2D_2.3s ;
```

8.1.3.8 Time-of-Day Constants

A time of day is introduced by the prefixes TIME_OF_DAY# or TOD#.

Syntax

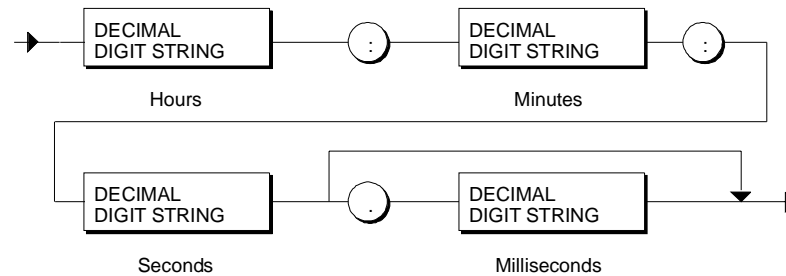
TIME OF DAY



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.

Specifying the Time of Day

Time of Day



Example

```

TIMEOFDAY1:= TIME_OF_DAY#12:12:12.2 ;
TIMEOFDAY2:= TOD#11:11:11 ;

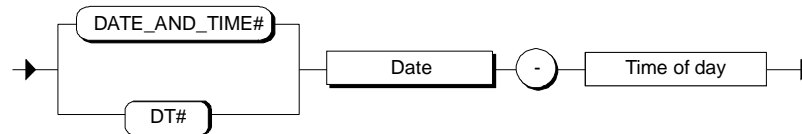
```

8.1.3.9 Date and Time Constants

A date and time are introduced by the prefixes DATE_AND_TIME# or DT#. This is constant formed by specifying a date and a time of day.

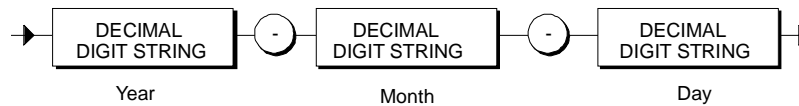
Syntax

DATE AND TIME



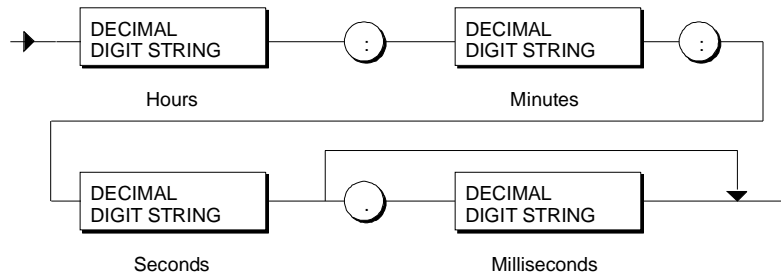
Date

Date



Time of Day

Time of Day



Example

```

TIMEOFDAY1 := DATE_AND_TIME#1995-01-01-12:12:12.2 ;
TIMEOFDAY2 := DT#1995-02-02-11:11:11;
  
```

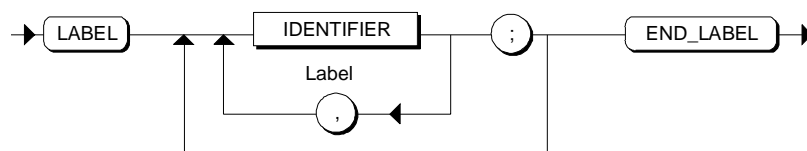

8.2 Declaring Labels

8.2.1 Declaring Labels

Labels are used to identify the destination of a GOTO statement. These are declared in the declaration section of a logic block with their symbolic names.

Syntax

Label Subsection



Example

```

LABEL
    LAB1, LAB2, LAB3;
END_LABEL
  
```


9 Shared Data

9.1 Overview of Shared Data

In SCL, you can access shared data. There are two types of shared data as follows:

- **CPU Memory Areas**

These memory areas contain system data, for example, inputs, outputs and bit memory. The number of memory areas available depends on the CPU you are using.

- **Shared User Data in the Form of Loadable Data Blocks**

These data areas are located within data blocks. To be able to use them, you must first create the data blocks and declare the data in them. Instance data blocks are based on specific function blocks and created automatically.

Access to Shared Data

You can access shared data as follows:

- **With absolute addressing:** Using the address identifier and the absolute address.
- **With symbolic addressing:** Specifying a symbol previously defined in the symbol table.
- **Indexed:** Using the address identifier and array index.
- **Structured:** Using a variable.

Type of Access	CPU Memory Areas	Shared User Data
Absolute	Yes	Yes
Symbolic	Yes	Yes
Indexed	Yes	Yes
Structured	No	Yes

9.2 Memory Areas of the CPU

9.2.1 Overview of the Memory Areas of the CPU

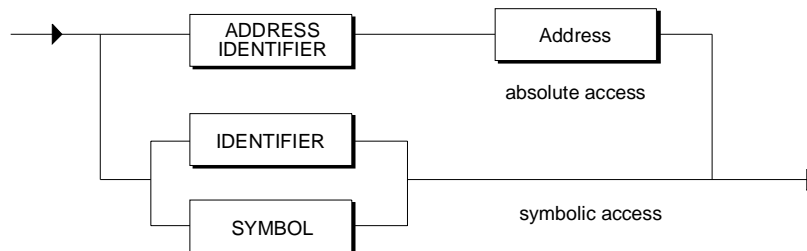
The memory areas of a CPU are areas declared throughout the system. For this reason, these areas do not need to be declared in your logic block. Every CPU provides the following memory areas with their own address ranges:

- Inputs/outputs in the process image (for example, Q1.0)
- Peripheral inputs/outputs (for example PQ1.0)
- Bit memory (for example M1.0)
- Timers, counters (C1)

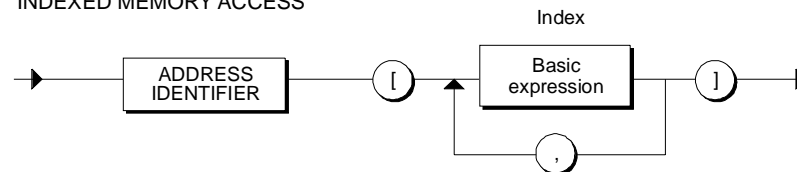
Syntax for Access

- You access a CPU memory area using a value assignment in the code section of a logic block, as follows: With simple access that you can specify as an absolute location or as a symbol, or
- Using indexed access.

SIMPLE MEMORY ACCESS

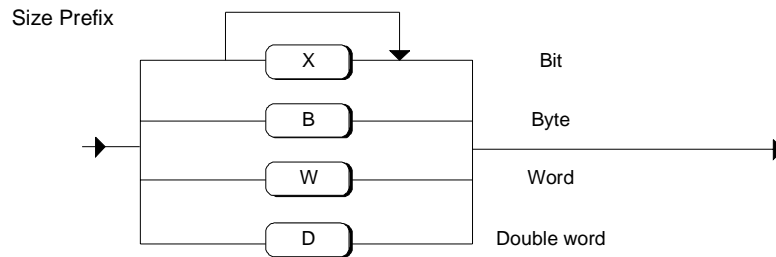


INDEXED MEMORY ACCESS



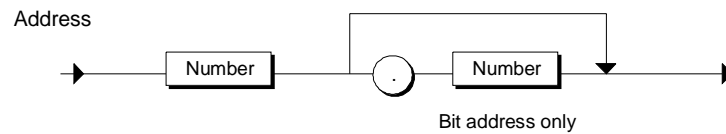
Size Prefix

With the size prefix, you specify the length of the memory area 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 you only want to specify one bit.



Address

For the address, you first specify the absolute byte address and then the bit address of the byte separated by a period. Specifying a bit address is optional.



Examples

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

9.2.3 Symbolic Access to Memory Areas of the CPU

With symbolic addressing, instead of an absolute identifier, you can use symbolic names to address the CPU memory areas.

You assign the symbolic names to the particular addresses in your user program by creating a symbol table. You can open this table at any time in SCL with the menu command **Options > Symbol Table** to add further symbols.

For the data type specification, you can use any elementary data type providing it can accept the specified data element size. The table below illustrates how a symbol table might appear.

Symbol	Absolute Address	Data Type	Comments
Motor_contact_1	I 1.7	BOOL	Contact switch 1 for Motor A
Input1	IW 10	INT	Status word

Access

The address is accessed by assigning a value to a variable of the same type with the declared symbol.

Example

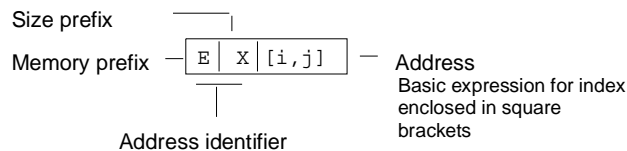
```
MEASVAL_1 := Motor_contact_1;  
Status_Motor1 := Input1 ;
```

9.2.4 Indexed Access to Memory Areas of the CPU

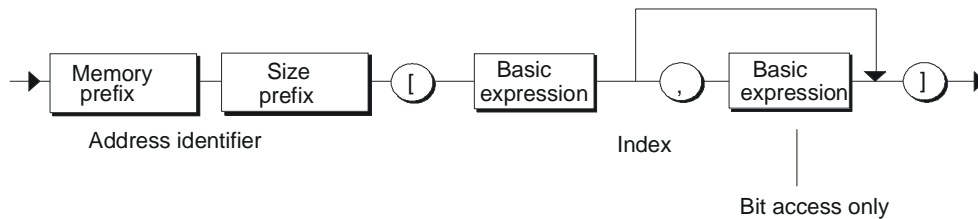
You can also access memory areas of the CPU using an index. Compared with absolute addressing the advantage of this method is that you can address dynamically using variable indexes. You can, for example, use the control variable of a FOR loop as the address.

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 an arithmetic expression.

For indexed access, the absolute identifier is made up of the address identifier (memory prefix and size prefix) and the basic expression for indexing.



Syntax of the Absolute Identifier



The Indexing (Base Expression) Must Adhere to the Following Rules:

- Each index must be an arithmetic 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 extent of the access is specified by the size prefix.
- When accessing data of the type BOOL, you must use two indexes. The first index specifies the byte address, the second index the bit position within the byte.

Example

```
MEASVAL_1      :=IW [COUNTER] ;
OUTLABEL :=I [BYTENO, BITNO] ;
```


9.3 Data Blocks

9.3.1 Overview of Data Blocks

Within data blocks, you can save and process all the data for your application whose scope is the entire program or the entire project. Each logic block can read or write shared user data.

Access

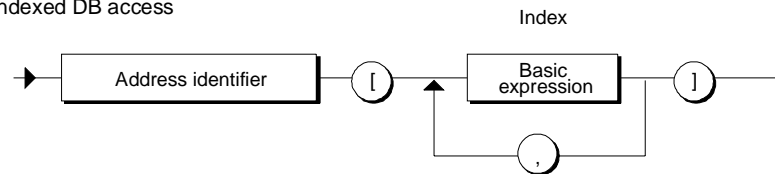
You can access the data of a shared data block in the following ways:

- absolute or simple,
- structured ,
- indexed.

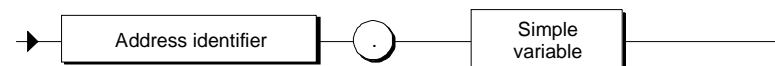
Absolute DB access



Indexed DB access

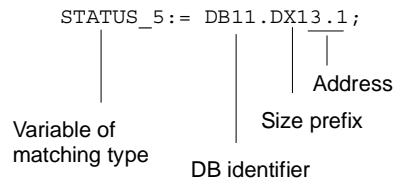


Structured DB access



9.3.2 Absolute Access to Data Blocks

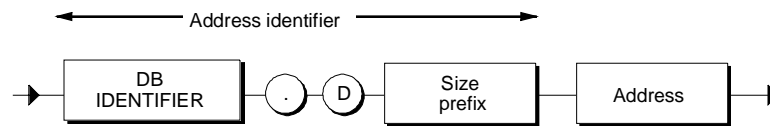
To program absolute access to a data block, you assign a value to a variable of the same type just as with the memory areas of the CPU. You first specify the DB identifier followed by the keyword "D" and the size prefix (for example X for bit) and the byte address (for example 13.1).



Syntax

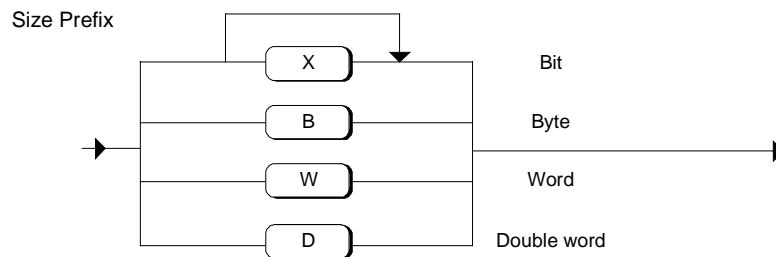
You define the access by specifying the DB identifier along with the size prefix and the address.

Absolute DB Access



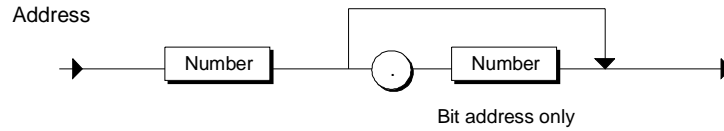
Size Prefix

The size prefix indicates the length of the memory area in the data block to be addressed. You can, for example, read a byte or a word from the DB. Using the size prefix is optional if you only want to specify one bit.



Address

When you specify the address, you first specify the absolute byte address and then the bit address (only with bit access) of the relevant byte separated by a period.



Example

Examples of absolute access to a data are shown below. 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      :=Status_data.DB10;
STATUS_3        :="New data".D1.1;
MEASVAL         :=Measdata.DW20.DW20;

STATUS_1        :=WORD_TO_BLOCK_DB (INDEX).DW10;

```

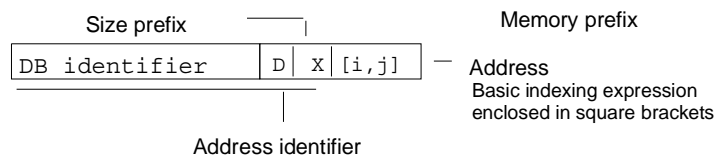
9.3.3 Indexed Access to Data Blocks

You can also access data blocks using an index. Compared with absolute addressing, this has the advantage of allowing you to address locations whose address is only decided during runtime. You can, for example, use the control variable of a FOR loop as the address.

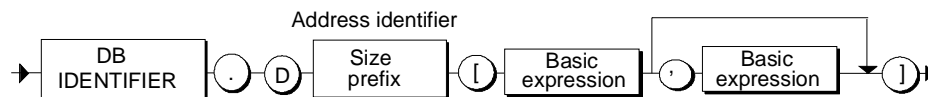
Indexed access to a data block is similar to absolute access. It differs only in the address specification.

Instead of the absolute address, an index is specified which can be a constant, a variable or an arithmetic expression.

Indexed access is made up of the DB identifier, the address identifier (keyword "D" and size prefix) and a basic expression for indexing.



Syntax



When using indexes, the following rules must be adhered to:

- 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 extent of the access is specified by the size prefix.
- When accessing data of the type BOOL, you must use two indexes. The first index specifies the byte address, the second index the bit position within the byte.
- Each index must be an arithmetic expression of the data type INT (0 - 32767).

Example

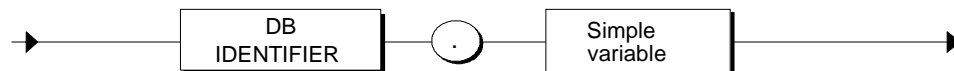
```
STATUS_1 := DB11.DW[COUNTER] ;
STATUS_2 := DB12.DX[WNO, BITNO] ;
STATUS_1 := Database1.DW[COUNTER] ;
STATUS_2 := Database2.DX[WNO, BITNO] ;
STATUS_1 := WORD_TO_BLOCK_DB(INDEX).DW[COUNTER] ;
```

9.3.4 Structured Access to Data Blocks

Structured access uses the identifier of the variables declared in the data block. You can assign the variable to any variable of the same type.

You reference the variable in the data block by specifying the DB name and the name of the simple variable separated by a period.

Syntax



The simple variable stands for a variable to which you assigned an elementary or complex data type in the declaration of the DB.

If a parameter of the type `BLOCK_DB` or the result of the conversion function `WORD_TO_BLOCK_DB` is used to initiate access to a data block, only absolute or indexed access is possible and structured access is not.

Example

In the declaration section of FB10:

```

VAR
Result:    STRUCT RES1 : INT;
RES2 : WORD;
END_STRUCT
END_VAR
  
```

```

User-defined data type UDT1
TYPE UDT1    STRUCT RES1 : INT;
RES2 : WORD;
END_STRUCT
  
```

DB20 with user-defined data type:

```

DB20
UDT1
BEGIN ...
  
```

DB30 without user-defined data type:

```

DB30    STRUCT RES1 : INT;
RES2 : WORD;
END_STRUCT
BEGIN ...
  
```

Function block with the following accesses:

```
..  
FB10.DB10();  
RESWORD_A   :=   DB10.Result.RES2;  
RESWORD_B   :=   DB20.RES2;  
RESWORD_C   :=   DB30.RES2;
```

10 Expressions, Operations and Addresses

10.1 Overview of Expressions, Operations and Addresses

An expression stands for a value that is calculated during compilation or during runtime and consists of addresses (for example constants, variables or function calls) and operations (for example *, /, + or -).

The data types of the addresses and the operations used determine the type of expression. The following expressions are possible in SCL:

- Arithmetic expressions
- Comparison expressions
- Logical expressions

An expression is evaluated in a specific order. This is decided by the following:

- the precedence of the operations involved and
- working from left to right or
- with operations having the same precedence by the parentheses.

You can do the following with the result of an expression:

- Assign it to a variable.
- Use it as a condition for control statement.
- Use it as a parameter for calling a function or a function block.

10.2 Operations

Expressions consist of operations and addresses. Most SCL operations combine two addresses and are therefore termed *binary* operators. The other operations involve only one address and are called *unary* operators.

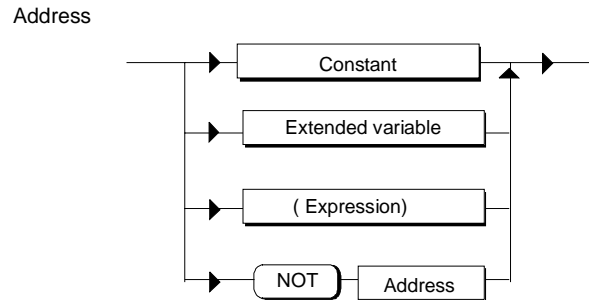
Binary operations are written between the addresses (for example, $A + B$). A unary operation always stands immediately before its address (for example, $-B$).

The precedence of the operations listed in the table below governs the order of evaluation. '1' represents the highest precedence.

Class	Operation	Symbol	Precedence
Assignment Operation:	Assignment	$:=$	11
Arithmetic Operations:	Power	$**$	2
	Unary Operations		
	Unary plus	$+$	3
	Unary minus	$-$	3
	Basic Arithmetic Operations		
	Multiplication	$*$	4
	Division	$/$	4
	Modulo function	MOD	4
	Integer division	DIV	4
	Addition	$+$	5
	Subtraction	$-$	5
Comparison Operations:	Less than	$<$	6
	Greater than	$>$	6
	Less than or equal to	$<=$	6
	Greater than or equal to	$>=$	6
	Equal to	$=$	7
	Not equal to	$<>$	7
Logical Operations:	Negation	NOT	3
	Basic Logical Operations		
	And	AND or $\&$	8
	Exclusive or	XOR	9
	Or	OR	10
Parentheses :	Parentheses	$()$	1

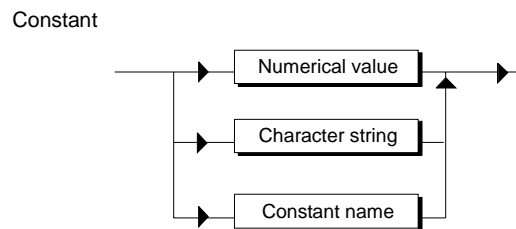
10.3 Addresses

Addresses are objects with which an expression can be formed. The following elements are permitted in addresses:



Constants

Constants can be a numerical value or a symbolic name or a character string.



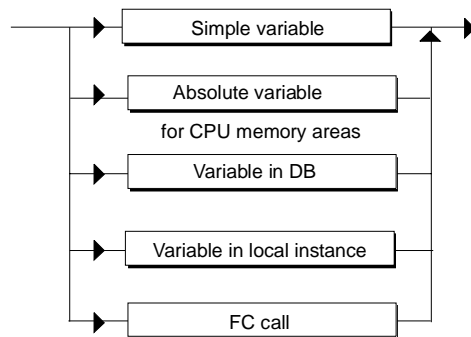
The following are examples of valid constants:

4_711
 4711
 30.0
 ' CHARACTER '
 FACTOR

Extended Variable

An extended variable is a generic term for a series of variables that are dealt with in more detail in the section entitled "Value Assignments".

Extended variable



Some examples of valid variables:

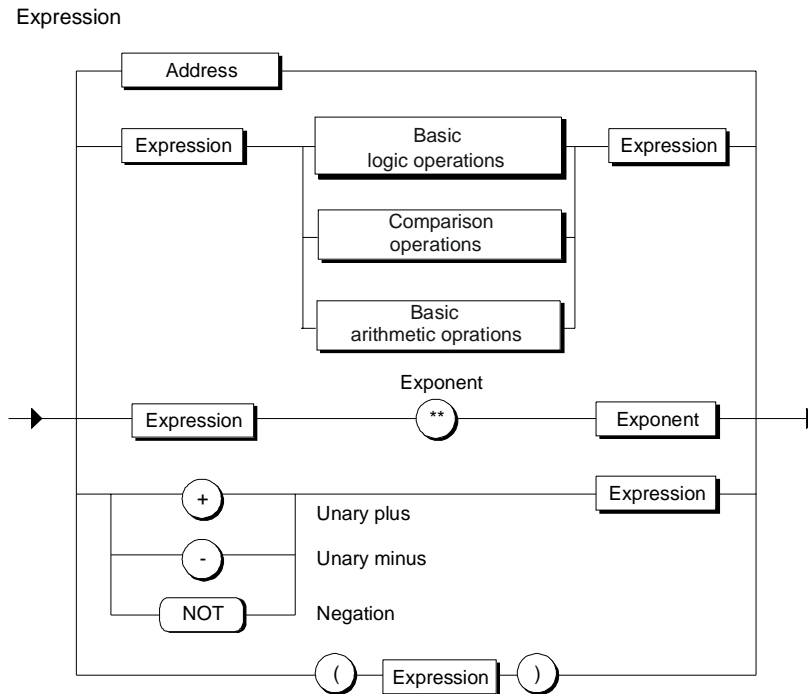
SETPOINT	simple variable
IW10	absolute variable
I100.5	absolute variable
DB100.DW [INDEX]	variable in the DB
MOTOR.SPEED	variable in a local instance
SQR (20)	standard function
FC192 (SETPOINT)	function call

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. VOID functions that do not return a value are therefore not allowed as addresses in an expression.

10.4 Syntax of an Expression

Syntax



Result of an Expression

You can do the following with 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.

Order of Evaluation

The order of evaluation of an expression depends on the following:

- The precedence of the operations involved
- The order from left to right
- The use of parentheses (if operations have the same precedence).

Rules

Expressions evaluate according to the following rules:

- An address between two operations with different precedence is always associated with the higher precedence operation.
- The operations are processed according to the hierarchical order.
- Operations with the same precedence are evaluated from left to right.
- Placing a minus sign before an identifier is the same as multiplying it by -1.
- Arithmetic operations must not follow each other directly. The expression $a * -b$ is invalid, whereas $a * (-b)$ is permitted.
- Parentheses can be used to overcome operation precedence; in other words, parentheses have the highest precedence.
- Expressions in parentheses are considered as a single address and always evaluated first.
- The number of left parentheses must match the number of right parentheses.
- Arithmetic operations cannot be used in conjunction with characters or logical data. Expressions such as 'A' + 'B' and $(n \leq 0) + (m > 0)$ are incorrect.

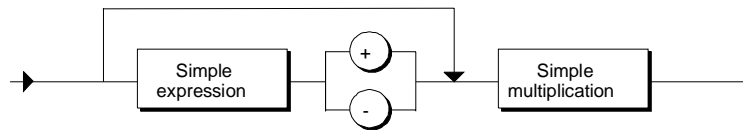
Examples of Expressions

IB10	// address
A1 AND (A2)	// logical expression
(A3) < (A4)	// comparison expression
3+3*4/2	//arithmetic expression

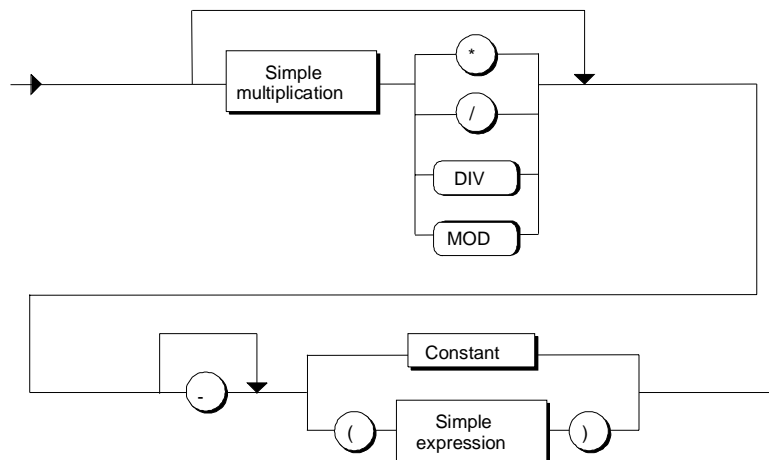
10.5 Simple Expression

In SCL, a simple expression means a simple arithmetic expression. You can multiply or divide constant values in pairs and add or subtract these pairs.

Syntax of a Simple Expression



Syntax of Simple Multiplication



Example

SIMP_EXPRESSION= A * B + D / C - 3 * VALUE1;

10.6 Arithmetic Expressions

An arithmetic expression is an expression formed with arithmetic operations. These expressions allow numeric data types to be processed.

The following table shows all the possible *operations* and indicates the type to which the result belongs depending on the addresses. The following abbreviations are used:

ANY_INT	for data types	INT, DINT
ANY_NUM	for data types	ANY_INT and REAL

Operation	Identifier	1st Address	2nd Address	Result	Precedence
Power	**	ANY_NUM	ANY_NUM	REAL	2
Unary plus	+	ANY_NUM	-	ANY_NUM	3
		TIME	-	TIME	3
Unary minus	-	ANY_NUM	-	ANY_NUM	3
		TIME	-	TIME	3
Multiplication	*	ANY_NUM	ANY_NUM	ANY_NUM	4
		TIME	ANY_INT	TIME	4
Division	/	ANY_NUM	ANY_NUM	ANY_NUM	4
		TIME	ANY_INT	TIME	4
Integer division	DIV	ANY_INT	ANY_INT	ANY_INT	4
		TIME	ANY_INT	TIME	4
Modulo division	MOD	ANY_INT	ANY_INT	ANY_INT	4
Addition	+	ANY_NUM	ANY_NUM	ANY_NUM	5
		TIME	TIME	TIME	5
		TOD	TIME	TOD	5
		DT	TIME	DT	5
Subtraction	-	ANY_NUM	ANY_NUM	ANY_NUM	5
		TIME	TIME	TIME	5
		TOD	TIME	TOD	5
		DATE	DATE	TIME	5
		TOD	TOD	TIME	5
		DT	TIME	DT	5
		DT	DT	TIME	5

Note

Remember that when you specify an address type (for example ANY_NUM), the addresses depend on the type of the result. If the result of the expression is, for example, of the type INTEGER, you must not use addresses of the type REAL.

Rules

Operations in arithmetic expressions are handled in the order of their precedence.

- It is advisable to place negative numbers in brackets for the sake of clarity even in cases where it is not syntactically necessary.
- When dividing with two whole numbers of the type INT, the operations "DIV" and "/" produce the same result (see example below).
- In the division operations ('/', 'MOD' and 'DIV'), the second address must not be 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

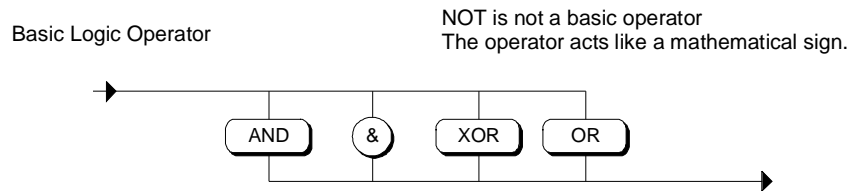
```
// The result (11) of the arithmetic expression is
// assigned to the variable "VALUE"
VALUE1 := 3 + 3 * 4 / 2 - (7+3) / (-5) ;
// The VALUE of the following expression is 1
VALUE2 := 9 MOD 2 ;
```

10.7 Logical Expressions

A logical expression is an expression formed by logic operations.

Basic Logic Operations

Using the operations AND, &, XOR and OR, logical addresses (BOOL type) or variables of the data type BYTE, WORD or DWORD can be combined to form logical expressions. To negate a logical address, the NOT operation is used.



Logic Operations:

The result of the expression is either TRUE or FALSE following a logic operation on Boolean addresses or it is a bit pattern after logic operation on the bits of two addresses.

The following table lists the available logical expressions and data types for the result and addresses. The following abbreviations are used:

ANY_BIT

for data types

BOOL, BYTE, WORD, DWORD

Operation	Identifier	1st Address	2nd Address	Result	Precedence
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

Result:

The result of a logic expression is either

- 1 (*true*) or 0 (*false*) if Boolean addresses are combined, or
- A bit pattern corresponding to the combination of the two addresses.

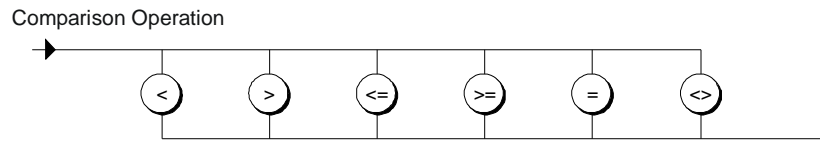
Examples

```
// The result of the comparison expression is negated.  
    IF NOT (COUNTER > 5) THEN . . . ;  
// The result of the first comparison expression is negated  
    and  
// combined with the result of the second  
    A := NOT (COUNTER1 = 4) AND (COUNTER2 = 10) ;  
// Disjunction of two comparison expressions  
    WHILE (A >= 9) OR (SCAN <> "n") DO.... ;  
// Masking an input byte (bit operation)  
    Result := IB10 AND 2#11110000 ;
```

10.8 Comparison Expressions

The comparison operations compare the values of two addresses and evaluate to a Boolean value. The result is TRUE if the comparison condition is true and FALSE if it fails.

Syntax



Rules

The following rules apply to comparison expressions:

- 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:
 - DT, TIME, DATE, TOD
- When comparing characters (CHAR type), the operation uses the order of the ASCII character set.
- S5 TIME variables are not permitted in comparison operations. S5TIME format must be converted explicitly to TIME using IEC functions.
- Comparison expressions can be combined according to the rules of Boolean logic to implement statements such as "if a < b and b < c then ...".
(Example: Value_A > 20 AND Value_B < 20)
The operations are evaluated in the order of their precedence. The precedence can be changed by parentheses.

Example:

A<>(B AND C)

Examples

```
// Compare 3 LESS THAN OR EQUAL TO 4. The result
// is "TRUE"
    A := 3 <= 4
// Compare 7 NOT EQUAL TO 7. The result
// is "FALSE"
    7 <> 7
// Evaluation of a comparison expression in
// an IF statement
    IF COUNTER < 5 THEN ....
```

11 Statements

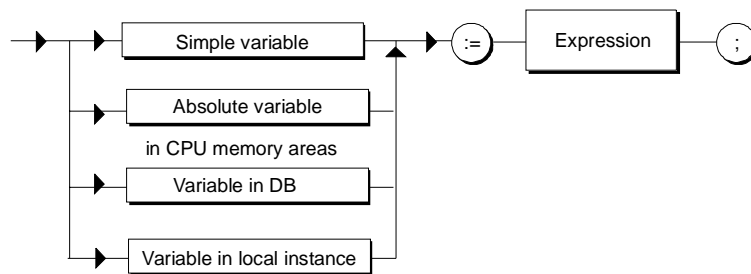
11.1 Value Assignments

When a value is assigned, the current value of a variable is replaced by a new value specified by an expression. This expression can also contain identifiers for functions that are activated by the statement and then return corresponding values (return value).

As shown in the diagram below, the expression on the right-hand side of the assignment operator is evaluated and the value obtained as the result is set in the variable whose name is on the left-hand side of the assignment operator. The variables permitted for this function are shown in the figure.

Syntax of a Value Assignment

Value Assignment



The type of an assignment expression is the type of the left address. The simple variable can be a variable of an elementary or complex data type.

11.1.1 Value Assignments with Variables of an Elementary Data Type

Every expression and every variable of an elementary data type can be assigned the value of a different variable of the same type.

```
Identifier := Expression ;  
Identifier := Variable ;
```

Example

```
FUNCTION_BLOCK FB12  
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 ;  
    TIMEOFDAY_1   : TIME_OF_DAY ;  
END_VAR  
BEGIN  
  
    // Assignment of a constant to a variable  
    SWITCH_1      := -17 ;  
    SETPOINT_1    := 100.1 ;  
    QUERY_1       := TRUE ;  
    TIME_1        := T#1H_20M_10S_30MS ;  
    TIME_2        := T#2D_1H_20M_10S_30MS ;  
    DATE_1        := D#1996-01-10 ;  
  
    // Assignment of a variable to a variable  
    SETPOINT_1    := SETPOINT_2 ;  
    SWITCH_2      := SWITCH_1 ;  
    // Assignment of an expression to a variable  
  
    SWITCH_2      := SWITCH_1 * 3 ;  
END_FUNCTION_BLOCK
```

11.1.2 Value Assignments with Variables of the Type STRUCT and UDT

Variables of the types STRUCT and UDT are structured variables that represent either a complete structure or a component of the structure.

The following are examples of valid structure variables:

```
Image           //Identifier for a structure
Image.element   //Identifier for a structure component
Image.arr       //Identifier for a single array
               //within a structure
Image.arr[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. The following assignments would be valid:

```
structname_1 := structname_2 ;
```

Assigning Structure Components

You can assign a variable of the same type, an expression of the same type or another structure component to any structure component.

You can reference a structure component by specifying the identifier of the structure and the identifier of the structure component separated by a period. The following assignments would be valid:

```
structname_1.element1      := Value ;
structname_1.element1      := 20.0 ;
structname_1.element1      := structname_2.element1 ;
structname_1.arrname1       := structname_2.arrname2 ;
structname_1.arrname[10]    := 100 ;
```

Example

```

FUNCTION_BLOCK FB3
VAR
    AUXVAR : REAL ;
    MEASVAL : STRUCT    //Target structure
        VOLTAGE:REAL ;
        RESISTANCE:REAL ;
        SIMPLEARR : ARRAY [1..2, 1..2] OF INT ;
    END_STRUCT ;
    PROCVAL : STRUCT    //Source structure
        VOLTAGE : REAL ;
        RESISTANCE : REAL ;
        SIMPLEARR : ARRAY [1..2, 1..2] OF INT ;
    END_STRUCT ;
END_VAR

BEGIN
    //Assignment of a complete structure to a complete structure
    MEASVAL := PROCVAL ;
    //Assignment of a structure component to a structure
    component
        MEASVAL.VOLTAGE := PROCVAL.VOLTAGE ;
    //Assignment of a structure component to a variable of the
    same type
        AUXVAR := PROCVAL.RESISTANCE ;
    //Assignment of a constant to a structure component
        MEASVAL.RESISTANCE := 4.5;
    //Assignment of a constant to a single array element
        MEASVAL.SIMPLEARR[1,2] := 4;
END_FUNCTION_BLOCK

```

11.1.3 Value Assignments with Variables of the Type ARRAY

An array consists of one up to a maximum of six dimensions and contains elements that are all of the same type. To assign arrays to a variable there are two access variants. You can reference complete arrays or a component of an array.

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 indexes) match. If this is the case, specify the identifier of the array after the assignment operator. The following assignments would be valid:

```
arrname_1 := arrname_2 ;
```

Assigning a Component of an Array

A single component of an array is addressed using the array name followed by suitable index values in square braces. An index is available for each dimension. These are separated by commas and also enclosed in square brackets. An index must be an arithmetic expression of the data type INT.

To obtain a value assignment for a permitted component, you omit indexes starting at the right in the square braces after the name of the array. In this way, you address a subset of the array whose number of dimensions is equal to the number of indexes omitted. The following assignments would be valid:

```
arrname_1[ i ] := arrname_2[ j ] ;  
arrname_1[ i ] := expression ;  
identifier_1   := arrname_1[ i ] ;
```

Example

```

FUNCTION_BLOCK FB3
VAR
    SETPOINTS      :ARRAY [0..127] OF INT ;
    PROCVALS       :ARRAY [0..127] OF INT ;
    // Declaration of a matrix (=two-dimensional array)
    // with 3 rows and 4 columns
    CRTLLR : ARRAY [1..3, 1..4] OF INT ;
    // Declaration of a vector (=one-dimensional array) with 4
    components
    CRTLLR_1 : ARRAY [1..4] OF INT ;
END_VAR

BEGIN
    // Assignment of a complete array to an array
    SETPOINTS := PROCVALS ;
    // Assignment of a vector to the second row of the CRTLLR
    //array
    CRTLLR[2] := CRTLLR_1 ;
    //Assignment of a component of an array to a component of the
    //CTRLLR array
    CRTLLR [1,4] := CRTLLR_1 [4] ;
END_FUNCTION_BLOCK

```


11.1.4 Value Assignments with Variables of the Data Type STRING

A variable of the data type STRING contains a character string with a maximum of 254 characters. Each variable of the STRING data type can be assigned another variable of the same type. The following assignments would be valid:

```
stringvariable_1 := stringconstant;  
stringvariable_1 := stringvariable_2 ;
```

Example

```
FUNCTION_BLOCK FB3  
VAR  
    DISPLAY_1    : STRING[50] ;  
    STRUCTURE1   : STRUCT  
        DISPLAY_2      : STRING[100] ;  
        DISPLAY_3      : STRING[50] ;  
    END_STRUCT ;  
END_VAR  
  
BEGIN  
    // Assignment of a constant to a STRING variable  
        DISPLAY_1 := 'Error in module 1' ;  
    // Assignment of a structure component to a STRING variable.  
        DISPLAY_1 := STRUCTURE1.DISPLAY_3 ;  
    // Assignment of a STRING variable to a STRING variable  
        If DISPLAY_1 <> STRUCTURE1.DISPLAY_3 THEN  
            DISPLAY_1 := STRUCTURE1.DISPLAY_3 ;  
        END_IF;  
END_FUNCTION_BLOCK
```

11.1.5 Value Assignments with Variables of the Type DATE_AND_TIME

The data type DATE_AND_TIME defines an area with 64 bits (8 bytes) for the date and time. Each variable of the data type DATE_AND_TIME can be assigned another variable of the same type or a constant. The following assignments would be valid:

```
dtvariable_1 := date and time constant;  
dtvariable_1 := dtvariable_2 ;
```

Example

```
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  
    // Assignment of a constant to a DATE_AND_TIME variable  
    TIME_1 := DATE_AND_TIME#1995-01-01-12:12:12.2 ;  
    STRUCTURE1.TIME_3 := DT#1995-02-02-11:11:11 ;  
    // Assignment of a structure component to a DATE_AND_TIME  
    // variable.  
    TIME_1 := STRUCTURE1.TIME_2 ;  
    // Assignment of a DATE_AND_TIME variable to a DATE_AND_TIME  
    // variable  
    If TIME_1 < STRUCTURE1.TIME_3 THEN  
        TIME_1 := STRUCTURE1.TIME_3 ;  
    END_IF ;  
END_FUNCTION_BLOCK
```

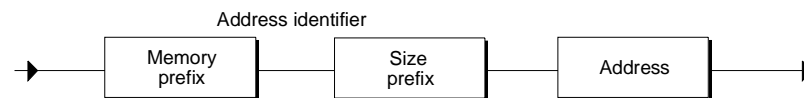
11.1.6 Value Assignments with Absolute Variables for Memory Areas

An absolute variable references the memory areas of a CPU with global scope. You can access these areas in three ways:

- Absolute Access
- Indexed Access
- Symbolic Access

Syntax of Absolute Variables

Absolute Variable



Example

```

FUNCTION_BLOCK FB3
VAR
  STATUSWORD1 : WORD ;
  STATUSWORD2 : BOOL ;
  STATUSWORD3 : BYTE ;
  STATUSWORD4 : BOOL ;
  ADDRESS      : INT ;
END_VAR
BEGIN
  ADDRESS := 10 ;
  // Assignment of an input word to a variable (simple access)
  STATUSWORD1 := IW4 ;
  // Assignment of a variable to an output bit (simple access)
  a1.1 := STATUSWORD2 ;
  // Assignment of an input byte to a variable (indexed access)
  STATUSWORD3 := IB[ADDRESS] ;
  // Assignment of an input bit to a variable (indexed access)
  FOR ADDRESS := 0 TO 7 BY 1 DO
    STATUSWORD4 := e[1, ADDRESS] ;
  END_FOR ;
END_FUNCTION_BLOCK
  
```

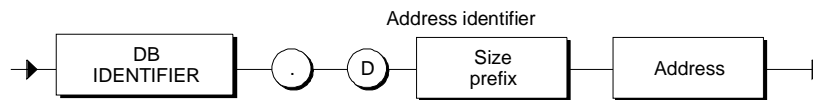
11.1.7 Value Assignments with Shared Variables

You also access data in data blocks by assigning a value to variables of the same type or vice-versa. You can assign any global variable a variable or expression of the same type. There are several ways in which you can access this data:

- Structured Access
- Absolute Access
- Indexed Access

Syntax of the DB Variable

DB Variable



Example

```

FUNCTION_BLOCK FB3
VAR
    CRTLLR_1      : ARRAY [1..4] OF INT ;
    STATUSWORD1   : WORD ;
    STATUSWORD2   : ARRAY [0..10] OF WORD ;
    STATUSWORD3   : INT ;
    STATUSWORD4   : WORD ;
    ADDRESS       : INT ;
END_VAR
VAR_INPUT
    ADDRESSWORD   : WORD ;
END_VAR
BEGIN
    // Assignment of word 1 from DB11
    //to a variable (simple access)
    STATUSWORD1 := DB11.DW1 ;
    // The array component in the 1st row and
    // 1st column of the matrix is assigned the value
    // of the "NUMBER" variable (structured access):
    CRTLLR_1[1] := DB11.NUMBER ;
    // Assignment of structure component "NUMBER2"
    // of structure "NUMBER1" to the variable status word3
    STATUSWORD3 := DB11.NUMBER1.NUMBER2 ;
    // 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] ;
            // Here the input parameter ADDRESSWORD as number of the
            //DB and the index ADDRESS are used to specify the word
            //address within the DB.
            STATUSWORD4 :=
WORD_TO_BLOCK_DB(ADDRESSWORD).DW[ADDRESS] ;
        END_FOR ;
    END_FUNCTION_BLOCK

```

11.2 Control Statements

11.2.1 Overview of Control Statements

Selective Statements

A selective statement enables you to direct program execution into alternative sequences of statements.

Types of Branch	Function
IF Statement	The IF statement enables you to direct program execution into one of two alternative branches depending on a condition being TRUE or FALSE:
CASE Statement	The CASE statement enables you direct program execution into 1 of n alternative branches based on the value of a variable.

Loops

You can control loop execution using iteration statements. An iteration statement specifies which parts of a program should be iterated depending on certain conditions.

Types of Branch	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 terminate condition is met

Program Jump

A program jump means an immediate jump to a specified jump destination and therefore to a different statement within the same block.

Types of Branch	Function
CONTINUE Statement	Used to stop execution of the current loop iteration.
EXIT Statement	Used to exit a loop at any point regardless of whether the terminate condition is satisfied or not
GOTO Statement	Causes the program to jump immediately to a specified label
RETURN Statement	Causes the program to exit the block currently being executed

11.2.2 Conditions

The condition is either a comparison expression, a logical expression or an arithmetic expression. It is of the type BOOL and can have the values TRUE or FALSE. Arithmetic expressions are TRUE if they result in a value other than 0 and FALSE when they result in the value 0. The table below shows examples of conditions:

Type	Example
Comparison expression	TEMP > 50 COUNTER <= 100 CHAR1 < 'S'
Comparison and logical expression	(ALPHA <> 12) AND NOT BETA
Boolean address	I 1.1
Arithmetic expression	ALPHA = (5 + BETA)

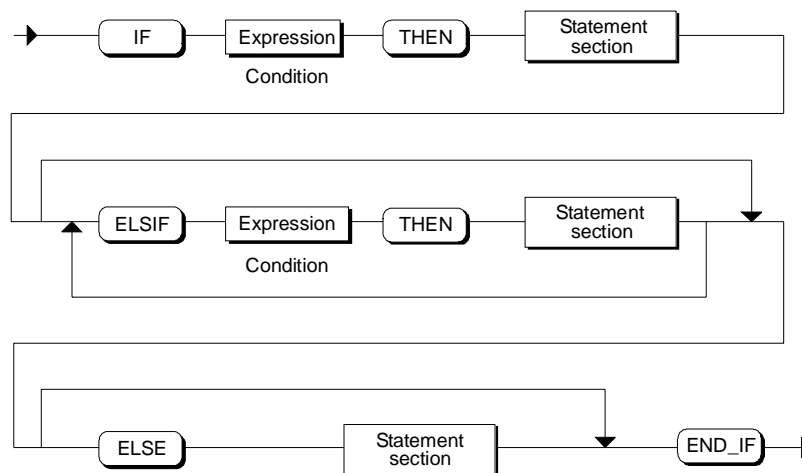
11.2.3 IF Statements

The IF statement is a conditional statement. It provides one or more options and selects one (or none) of its statement components for execution.

Execution of the conditional statement evaluates 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.

Syntax

IF Statement



An IF statement is executed according to the following rules:

- The first sequence of statements whose logical expression = TRUE is executed. The remaining sequences of statements are not executed.
- If no Boolean expression = TRUE, the sequence of statements introduced by ELSE is executed (or no sequence of statements if the ELSE branch does not exist).
- Any number of ELSIF statements can exist.

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

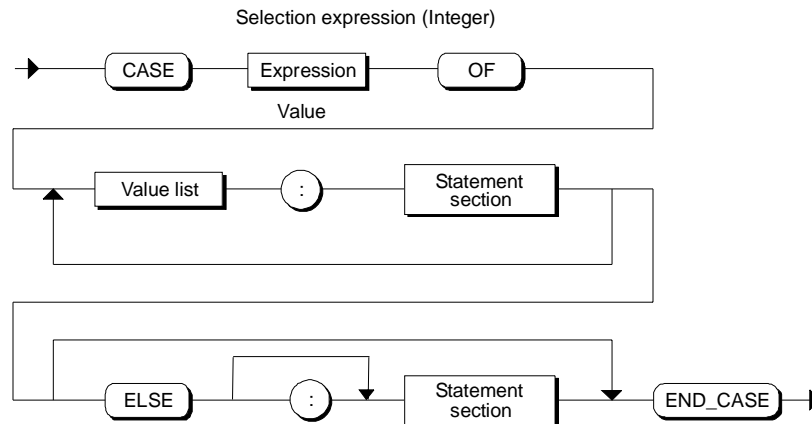
```
IF I1.1 THEN
    N := 0 ;
    SUM := 0 ;
    OK := FALSE ; // Set OK flag to FALSE
ELSIF START = TRUE THEN
    N := N + 1 ;
    SUM := SUM + N ;
ELSE
    OK := FALSE ;
END_IF ;
```

11.2.4 CASE Statement

The CASE statement is used to select one of several alternative program sections. This choice is based on the current value of a selection expression.

Syntax

CASE Statement



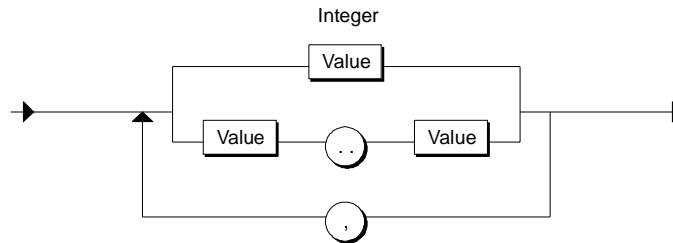
The CASE statement is executed according to the following rules:

- The selection expression must return a value of the type INTEGER.
- When a CASE statement is processed, the program checks whether the value of the selection expression is contained within a specified list of values. If a match is found, the statement component assigned to the list is executed.
- If no match is found, the program section following ELSE is executed or no statement is executed if the ELSE branch does not exist.

Value List

This contains the values permitted for the selection expression

Value List

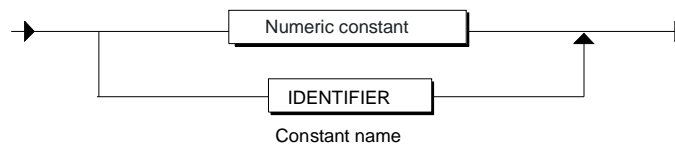


The following rules apply to the value list:

- Each value list begins with a constant, a list of constants or a range of constants.
- The values within the value list must be of the type INTEGER.
- Each value must only occur once.

Value

The value has the syntax shown below:



Example

```

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 ;

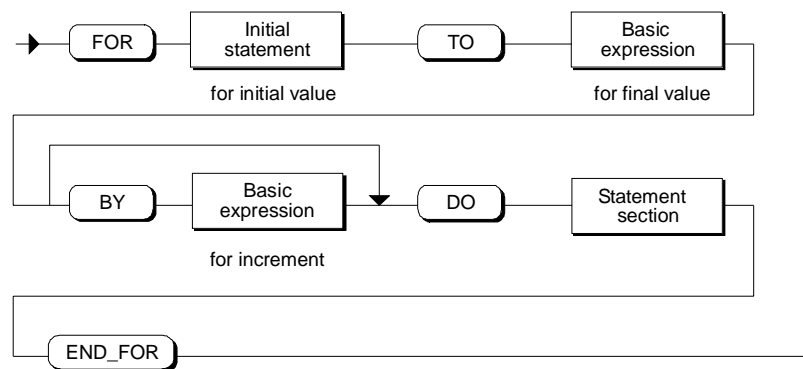
```

11.2.5 FOR Statement

A FOR statement is used to repeat a sequence of statements as long as a control variable is within the specified range of values. The control variable must be the identifier of a local variable of the type INT or DINT. The definition of a loop with FOR includes the specification of an initial and an end value. Both values must be the same type as the control variable.

Syntax

FOR Statement



The FOR statement executes as follows:

- At the start of the loop, the control variable is set to the initial value (initial assignment) and each time the loop iterates, it is incremented by the specified increment (positive increment) or decremented (negative increment) until the final value is reached.
- Following each run through of the loop, the condition is checked (final value reached) to establish whether or not it is satisfied. If the condition is satisfied, the sequence of statements is executed, otherwise the loop and with it the sequence of statements is skipped.

Rules

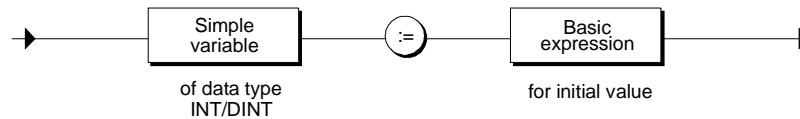
Rules for formulating FOR statements

- The control variable may only be of the data type INT or DINT.
- You can omit the statement BY [increment]. If no increment is specified, it is automatically assumed to be +1.

Initial Assignment

The initial value of the control variable must have the following syntax. The simple variable on the left of the assignment must be data type INT or DINT.

Initial Assignment

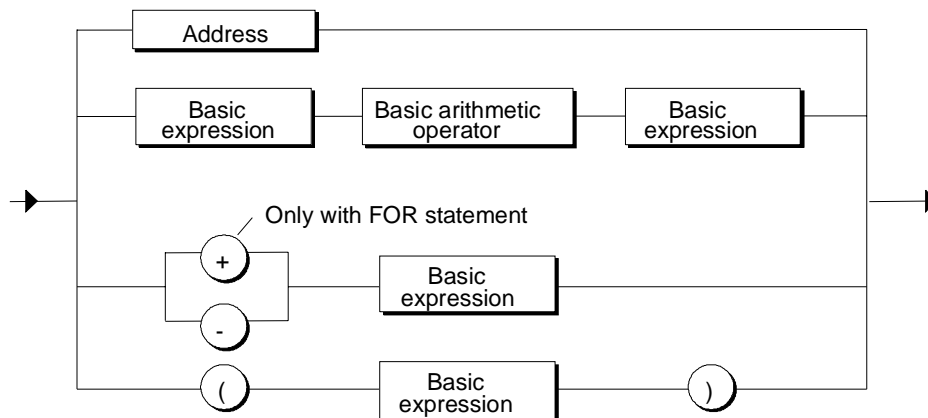


Examples of valid initial assignments:

```
FOR I := 1 TO 20
FOR I := 1 TO (START + J)
```

Final Value and Increment

You can write a basic expression for the final value and the required increment. This basic expression must have the following syntax:



- You can omit the statement BY [increment]. If no increment is specified, it is automatically assumed to be +1.
- The initial value, final value and increment are expressions (see "Expressions, Operations and Addresses"). It is evaluated once at the start when the FOR statement is executed.
- Alteration of the values for final value and increment is not permitted while the loop is executing.

Example

```
FUNCTION_BLOCK FOR_EXA
VAR
    INDEX: INT ;
    IDWORD: ARRAY [1..50] OF STRING;
END_VAR
BEGIN
    FOR INDEX := 1 TO 50 BY 2 DO
        IF IDWORD [INDEX] = 'KEY' THEN
            EXIT;
        END_IF;
    END_FOR;

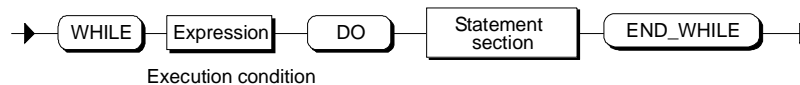
END_FUNCTION_BLOCK
```

11.2.6 WHILE Statement

The WHILE statement allows the repeated execution of a sequence of statements controlled by an execution condition. The execution condition is formed according to the rules for logical expressions.

Syntax

WHILE Statement



The WHILE statement executes according to the following rules:

- Prior to each iteration of the loop body, the execution condition is evaluated.
- The loop body following DO iterates as long as the execution condition has the value TRUE.
- Once the value FALSE occurs, the loop is skipped and the statement following the loop is executed.

Example

```

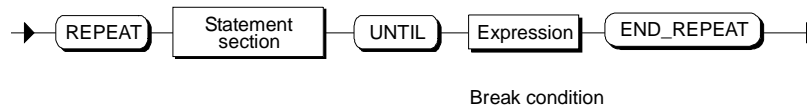
FUNCTION_BLOCK WHILE_EXA
VAR
    INDEX: INT ;
    IDWORD: ARRAY [1..50] OF STRING ;
END_VAR
BEGIN
    INDEX := 1 ;
    WHILE INDEX <= 50 AND IDWORD[INDEX] <> 'KEY' DO
        INDEX := INDEX + 2 ;
    END_WHILE ;
END_FUNCTION_BLOCK
  
```

11.2.7 REPEAT Statement

A REPEAT statement causes the repeated execution of a sequence of statements between REPEAT and UNTIL until a terminate condition occurs. The terminate condition is formed according to the rules for logical expressions.

Syntax

REPEAT Statement



The condition is evaluated after the loop body has been executed. This means that the loop body must be executed at least once even if the termination condition is satisfied when the loop is started.

Example

```
FUNCTION_BLOCK REPEAT_EXA
VAR
    INDEX: INT ;
    IDWORD: ARRAY [1..50] OF STRING ;
END_VAR

BEGIN
    INDEX := 0 ;
    REPEAT
        INDEX := INDEX + 2 ;
    UNTIL INDEX > 50 OR IDWORD[INDEX] = 'KEY'
    END_REPEAT ;

END_FUNCTION_BLOCK
```


11.2.8 CONTINUE Statement

A CONTINUE statement is used to terminate the execution of the current iteration of a loop statement (FOR, WHILE or REPEAT).

Syntax

CONTINUE Statement



The CONTINUE statement executes according to the following rules:

- This statement immediately terminates execution of a loop body.
- Depending on whether the condition for repeating the loop is satisfied or not the body is executed again or the iteration statement is exited and the statement immediately following is executed.
- In a FOR statement, the control variable is incremented by the specified increment immediately after a CONTINUE statement.

Example

```

FUNCTION_BLOCK CONTINUE_EXA
VAR
  INDEX :INT ;
  ARRAY :ARRAY[1..100] OF INT ;
END_VAR

BEGIN
  INDEX := 0 ;
  WHILE INDEX <= 100 DO
    INDEX := INDEX + 1 ;
    // If ARRAY[INDEX] is equal to INDEX,
    // then ARRAY [INDEX] is not changed:
    IF ARRAY[INDEX] = INDEX THEN
      CONTINUE ;

      END_IF ;
      ARRAY[INDEX] := 0 ;
      // Further statements
    END_WHILE ;
  END_FUNCTION_BLOCK
  
```

11.2.9 EXIT Statement

An EXIT statement is used to exit a loop (FOR, WHILE or REPEAT) at any point regardless of whether the terminate condition is satisfied.

Syntax

EXIT Statement



The EXIT statement executes according to the following rules:

- 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

```
FUNCTION_BLOCK EXIT_EXA
VAR
    INDEX_1      : INT ;
    INDEX_2      : INT ;
    INDEX_SEARCH : INT ;
    IDWORD : 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
        // IDWORD[INDEX_1] is equal to 'KEY':
        IF IDWORD[INDEX_1] = 'KEY' THEN
            INDEX_2 := INDEX_1 ;
            EXIT ;
        END_IF ;
    END_FOR ;
    // The following value assignment is made
    // after executing EXIT or after the
    // regular end of the FOR loop:
    INDEX_SEARCH := INDEX_2 ;
END_FUNCTION_BLOCK
```

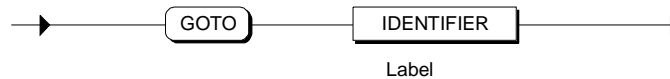
11.2.10 GOTO Statement

You can implement a program jump using the GOTO statement. This causes an immediate jump to the specified label and therefore to a different statement within the same block.

GOTO statements should only be used in special situations, for example in error handling. According to the rules of structured programming, the GOTO statement should not be used.

Syntax

GOTO Statement



Here, label is a label in the LABEL/END_LABEL declaration section. This label precedes the statement that will be executed next after the GOTO statement.

If you use the GOTO statement, remember the following rules:

- The destination of the jump must be in the same block.
- The destination of the jump must be uniquely identified.
- It is not possible to jump to a loop section. It is possible to jump from within a loop.

Example

```

FUNCTION_BLOCK GOTO_EXA
VAR
  INDEX : INT ;
  A      : INT ;
  B      : INT ;
  C      : INT ;
  IDWORD : ARRAY[1..51] OF STRING ;
END_VAR
LABEL
  LAB1, LAB2, LAB3 ;
END_LABEL

BEGIN
  IF A > B THEN
    GOTO LAB1 ;
  ELSIF A > C THEN
    GOTO LAB2 ;
  END_IF ;
  // . . .
  LAB1: INDEX := 1 ;
        GOTO LAB3 ;
  LAB2: INDEX := 2 ;
  // . . .
  LAB3:
  // . . .

```

11.2.11 RETURN Statement

A RETURN statement exits the currently active block (OB, FB, FC) and returns to the calling block or to the operating system, when an OB is exited.

Syntax

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 this is automatically executed.

11.3 Calling Functions and Function Blocks

11.3.1 Call and Parameter Transfer

Calling FCs and FBs

To make it easier to read and update user programs, the functions of the program are divided into smaller individual tasks that are performed by function blocks (FBs) and functions (FCs). You can call other FCs and FBs from within an SCL block. You can call the following blocks:

- Function blocks and functions created in SCL
- Function blocks and functions created in other STEP 7 languages (LAD, FBD, STL)
- System functions (SFCs) and system function blocks (SFBs) available in the operating system of the CPU.

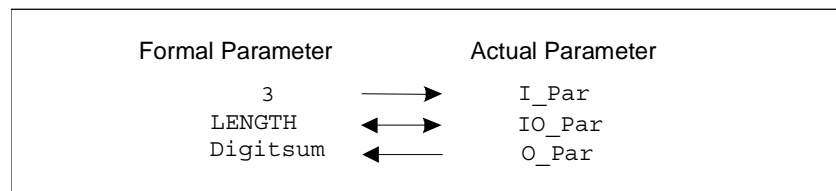
Basic Principle of Parameter Transfer

When functions or function blocks are called, data is exchanged between the calling and the called block. Parameters are defined in the interface of the called block with which the block works. These parameters are known as formal parameters. They are merely "placeholders" for the parameters that are passed to the block when it is called. The parameters passed to the block are known as actual parameters.

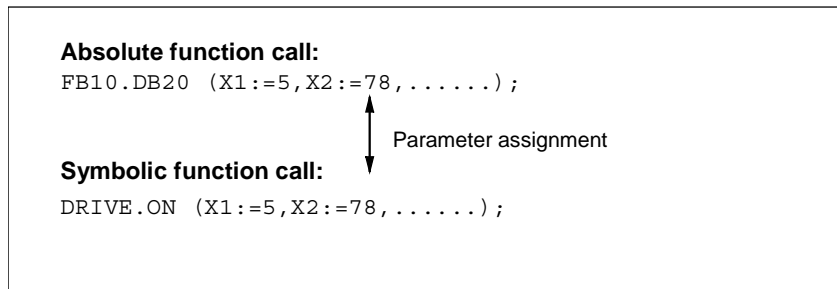
Syntax of Parameter Transfer

The parameters that are to be transferred must be specified in the call in the form of a parameter list. The parameters are enclosed in brackets. A number of parameters are separated by commas.

In the example of a function call below, an input parameter, an in/out parameter and an output parameter are specified.



The parameters are specified in 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).



11.3.2 Calling Function Blocks

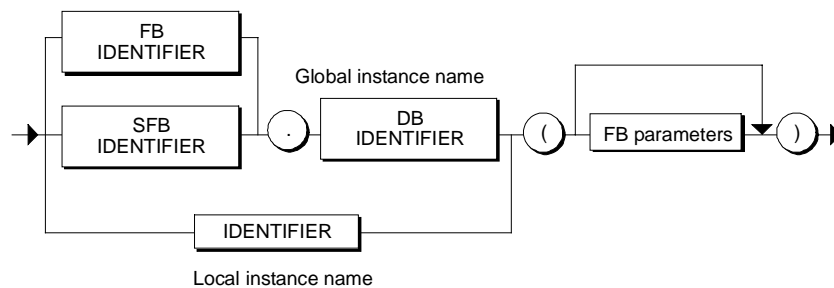
When you call a function block, you can use both shared instance data blocks and local instance areas of the currently active instance data block.

Calling an FB as a local instance differs from the call as a shared instance in the way in which the data are stored. Here, the data are not saved in a special DB but in the instance data block of the calling FB.

Syntax

Function Block Call

FB: Function block
 SFB: System function block



Call as a Shared Instance

The call is made in a call statement 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 supply (FB parameter).

A function call for a shared instance can be either absolute or symbolic.

Absolute function call:

```
FB10.DB20 (X1:=5,X2:=78,.....);
```

Symbolic function call:

```
DRIVE.ON (X1:=5,X2:=78,.....);
```

Parameter assignment

Call as a Local Instance

The call is made in a call statement by specifying the following:

- The local instance name (IDENTIFIER)
- The parameter supply (FB parameters)

A call for a local instance is always symbolic. You must declare the symbolic name in the declaration section of the calling block.

```
MOTOR (X1:=5,X2:=78,.....);
```

Parameter assignment

11.3.2.1 Supplying FB Parameters

When calling a function block (as a shared or local instance) you must supply the following parameters:

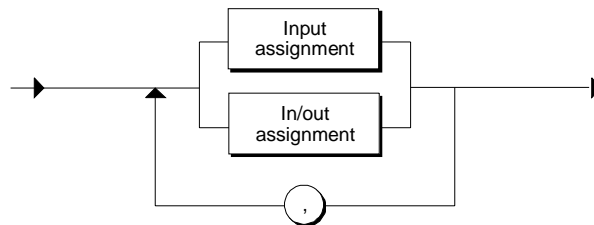
- Input Parameters
- In/out parameters

The output parameters do not have to be specified when an FB is called.

Syntax of a Value Assignment for Defining FB Parameters

The syntax of the FB parameter specification is the same when calling shared or local instances.

FB Parameters



The following rules apply when supplying parameters:

- The assignments can be in any order.
- The data types of formal and actual parameters must match.
- The assignments are separated by commas.
- 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.
- Remember the special features for parameters of the ANY data type and POINTER data type.

Result after Executing the Block

After executing the block:

- The actual parameters transferred are unchanged.
- The transferred and modified values of the in/out parameters have been updated; In/out parameters of an elementary data type are an exception to this rule.
- The output parameters can be read by the calling block from the shared instance data block or the local instance area.

Example

A call with an assignment for an input and an in/out parameter could, for example, appear as follows:

```
FB31.DB77(I_Par:=3, IO_Par:=LENGTH);
```

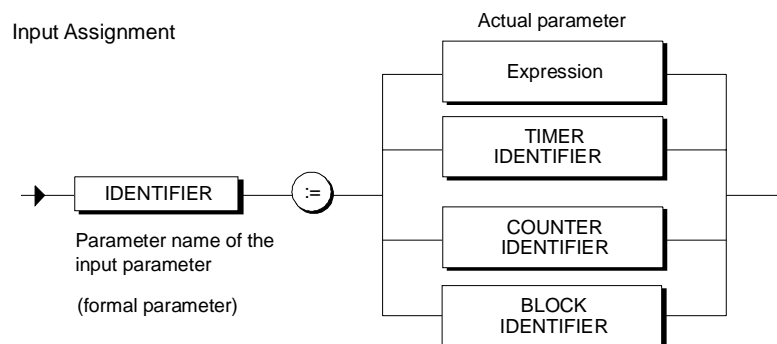
11.3.2.2 Input Assignment (FB)

The input assignments assign actual parameters to the formal 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.

Possible actual parameters are shown below:

Actual Parameter	Explanation
Expression	<ul style="list-style-type: none"> Arithmetic, logical or comparison expression Constant Extended variable
TIMER/COUNTER identifier	Defines a specific timer or counter to be used when a block is processed
BLOCK identifier	<p>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.</p> <p>When assigning parameter values you specify the block number. You can specify this in absolute or symbolic form.</p>

Syntax



11.3.2.3 In/Out Assignment (FB)

In/out assignments are used to assign actual parameters to the formal in/out parameters. The called FB can modify 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.

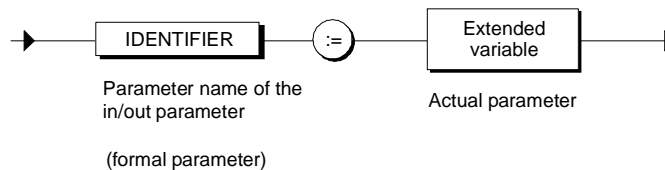
If in/out parameters are declared in the called FB, they must be supplied with values when the block is called the first time. When it is executed again, specifying actual parameters is optional. With in/out parameters of an elementary data type, there is no updating of the actual parameter if the formal parameter is not supplied with a value when the block is called.

Since the assigned actual parameter as an in/out parameter can be changed while the FB is being executed, it must be a variable.

Actual Parameter	Explanation
Extended Variable	The following types of extended variable are possible: <ul style="list-style-type: none">• Simple variables and parameters• Access to absolute variables• Access to data blocks• Function calls

Syntax

In/Out Assignment



Note

- Special rules apply to supplying values for the data types ANY and POINTER.
 - The following cannot be used as actual parameters for an in/out parameter of a non elementary data type:
 - FB in/out parameters
 - FC parameters
-

11.3.2.4 Reading Output Values (FB Call)

After the called block has been executed, the output parameters can be read from the shared instance block or the local instance area using a value assignment.

Example

```
RESULT:= DB10.CONTROL;
```

11.3.2.5 Example of a Call as a Shared Instance

An example of a function block with a FOR loop might appear as shown in the following examples. These examples assume that the symbol `TEST` has been declared in the symbol table for `FB17`.

Function Block

```
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
```

Call

To call the FB, you can choose one of the following variants. It is assumed that VARIABLE1 has been declared in the calling block as a REAL variable.

```
//Absolute function call, shared instance:  
FB17.DB10 (FINALVAL:=10, IQ1:=VARIABLE1);  
  
//Symbolic call, shared instance:  
TEST.TEST_1 (FINALVAL:=10, IQ1:= VARIABLE1);
```

Result:

After the block has executed, the value calculated for the in/out parameter IQ1 is available in VARIABLE1 .

Reading an 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;  
  
//You can also use the output parameter  
//directly in another FB call to  
//supply an input parameter:  
    FB17.DB12 (INP_1:=DB10.CONTROL);
```

11.3.2.6 Example of a Call as a Local Instance

A function block with a simple FOR loop could be programmed as in the example "Call as a Shared Instance" assuming that the symbol `TEST` is declared in the symbol table for `FB17`.

This FB can be called as shown below, assuming that `VARIABLE1` has been declared in the calling block as a `REAL` variable.

Call

```
FUNCTION_BLOCK CALL
VAR
  // Local instance declaration
  TEST_L : TEST ;
  VARIABLE1 : REAL ;
  RESULT : BOOL ;
END_VAR
BEGIN
  . . .

  // Call local instance:
  TEST_L (FINALVAL:= 10, IQ1:= VARIABLE1) ;
```

Reading an Output Value

The `CONTROL` output parameter can be read as follows:

```
// The output parameter is accessed
//by:
RESULT := TEST_L.CONTROL ;
END_FUNCTION_BLOCK
```

11.3.3 Calling Functions

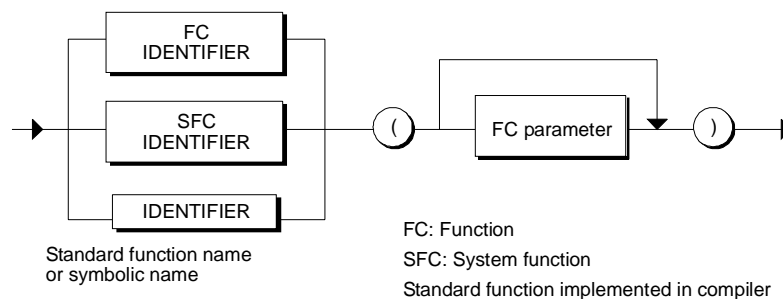
You call a function by specifying the function name (FC, SFC IDENTIFIER) and the parameter list. You can specify the function name that identifies the return value in absolute or symbolic form:

```
FC31 (X1:=5, Q1:=Checksum) ;    // Absolute  
DISTANCE (X1:=5, Q1:=Checksum) ;    // Symbolic
```

After the call, the results of the function are available as a return value or as output and in/out parameters (actual parameters).

Syntax

Function Call



Note

If a function is called in SCL and its return value was not supplied, this can lead to incorrect execution of the user program:

- This can occur with a function programmed in SCL when the return value was supplied but the corresponding statement was not executed.
 - This can occur in a function programmed in STL/LAD/FBD, if the function was programmed without the supply of the return value or the corresponding statement was not executed.
-

11.3.3.1 Return Value (FC)

In contrast to function blocks, functions supply a result known as the return value. For this reason, functions can be treated as addresses (exception: functions of the type VOID).

The function calculates the return value that has the same name as the function and returns it to the calling block. There, the value replaces the function call.

In the following value assignment, for example, the `DISTANCE` function is called and the result assigned to the `LENGTH` variable:

```
LENGTH:= DISTANCE (X1:=-3, Y1:=2);
```

The return value can be used in the following elements of an FC or FB:

- in a value assignment,
- in a logic, arithmetic or comparison expression or
- as a parameter for a further function block or function call.

Note

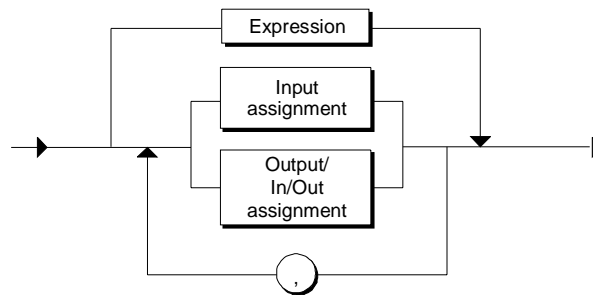
- If functions have the return value ANY, at least one input or in/out parameter must also be of the type ANY. If more than one ANY parameter is defined, you must supply them with actual parameters of the same type class (for example, INT, DINT and REAL). The return value is then automatically of the largest used data type in this type class.
 - The maximum length of the data type STRING can be reduced from 254 characters to any length.
-

11.3.3.2 FC Parameters

In contrast to function blocks, functions do not have any memory in which they could save the values of the parameters. Local data is only stored temporarily while the function is active. For this reason, when you call a function, all formal input, in/out and output parameters defined in the declaration section of a function must be assigned actual parameters.

Syntax

FC Parameter



Rules

Rules for supplying parameters

- The assignments can be in any order.
- The data types of formal and actual parameters must match.
- The data type of formal and actual parameters must match.

Example

A call with an assignment for an input, output and an in/out parameter could, for example appear as follows:

```
FC32 (E_Param1:=5,D_Param1:=LENGTH,  
      A_Param1:=Checksum)
```

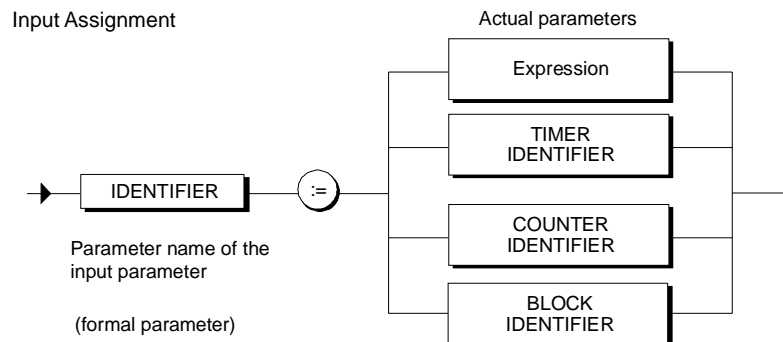

11.3.3.3 Input Assignment (FC)

Using input assignments, the formal input parameters of the called FC are assigned values (actual parameters). 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.

The following actual parameters can be assigned in input assignments:

Actual Parameter	Explanation
Expression	An expression represents a value and consists of addresses and operations. The following types of expression are possible: <ul style="list-style-type: none"> • Arithmetic, logical or comparison expression • Constant • Extended Variable
TIMER/COUNTER Name	Defines a specific timer or counter to be used when a block is processed
BLOCK Name	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 parameters, you specify the block address. You can use either the absolute or the symbolic address.

Syntax



Note

With formal input parameters of a non-elementary type, FB in/out parameters and FC parameters are not permitted as actual parameters. Remember the special features for the data types ANY and POINTER.

11.3.3.4 Output and In/Out Assignment (FC)

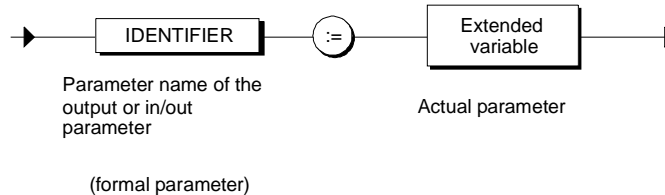
In an output assignment, you specify the variable of the calling block into which the output values resulting from executing a function will be written. An in/out assignment is used to assign an actual value to an in/out parameter.

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 cannot be assigned in in/out assignments (the value could not be written). This means that only extended variables can be assigned in output and in/out assignments.

Actual Parameter	Explanation
Extended Variable	The following types of extended variable are possible: <ul style="list-style-type: none">• Simple variables and parameters• Access to absolute variables• Access to data blocks• Function calls

Syntax

Output and In/Out Assignments



Note

The following actual parameters are not permitted with formal output or in/out parameters:

- FC/FB input parameters
 - FB in/out parameters of a non-elementary data type
 - FC in/out parameters and output parameters of a non-elementary data type
 - Remember the special features for the data types ANY and POINTER.
 - The maximum length of the data type STRING can be reduced from 254 characters to any length.
-

11.3.3.5 Example of a Function Call

Function to be Called

A function DISTANCE for calculating the distance between two points (X1,Y1) and (X2,Y2) in the same plane using the Cartesian system of coordinates might take the following form (the examples assume that the symbol DISTANCE has been declared in a symbol table for FC37).

```
FUNCTION DISTANCE: REAL // symbolic
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
```

Calling Block

The examples below show more options for further use of a function value:

```
FUNCTION_BLOCK CALL
VAR
  LENGTH : REAL ;
  CHECKSUM : REAL ;
  RADIUS : REAL;
  Y : REAL;
END_VAR
BEGIN
  . . .
  // Call in a value assignment:
  LENGTH := DISTANCE (X1:=3, Y1:=2, X2:=8.9, Y2:= 7.4,
    Q2:=CHECKSUM) ;

  //Call in an arithmetic or logic expression, for example,
  Y := RADIUS + DISTANCE (X1:=-3, Y1:=2, X2:=8.9, Y2:=7.4,
    Q2:=Checksum)

  //Use in the parameter supply of a further called block
  FB32.DB32 (DIST:= DISTANCE (X1:=-3, Y1:=2, X2:=8.9, Y2:=7.4),
    Q2:=Checksum)
  . . .
END_FUNCTION_BLOCK
```

11.3.4 Implicitly Defined Parameters

11.3.4.1 Input Parameter EN

Every function block and every function has the implicitly defined input parameter EN. EN is of the data type BOOL and is stored in the temporary block data area. 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 it 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 it was not called (EN : FALSE).

Example

```
FUNCTION_BLOCK FB57
VAR
    MY_ENABLE: BOOL ;
    Result : REAL;
END_VAR
// . . .
BEGIN
    // . . .
    MY_ENABLE:= FALSE ;

    // Calling a function and supplying the EN parameter:
    Result := FC85 (EN:= MY_ENABLE, PAR_1:= 27) ;
    // FC85 was not executed since MY_ENABLE above was set to
    // FALSE

END_FUNCTION_BLOCK
```

11.3.4.2 Output Parameter ENO

Every function block and every function has the implicitly defined output parameter ENO which is of the data type BOOL. It is stored in the temporary block data. Once a block has executed, the current value of the OK flag is entered 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

```
// Function block call:
FB30.DB30 ([Parameter supply]);

// Check whether everything ran correctly in the called
// block:
IF ENO THEN
// Everything OK
// . . .
ELSE
// Error occurred, so error handling required
// . . .
END_IF;
```


12 Counters and Timers

12.1 Counters

12.1.1 Counter Functions

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 Function	Explanation
S_CU	Count Up
S_CD	Count Down
S_CUD	Count Up Down

12.1.2 Calling Counter Functions

Counter functions are called like functions. The function identifier can therefore be used anywhere instead of an address in an expression as long as the type of the function value is compatible with that of the replaced address.

The function value (return value) returned to the calling block is the current count value (BCD format) in data type WORD.

Absolute or Dynamic Call

For the call, you can enter an absolute value as the counter number (for example C_NO:=C10). Such values can, however, no longer be modified during runtime.

Instead of the absolute counter number, you can also specify a variable or constant of the INT data type. The advantage of this method is that the counter call can be made dynamic by assigning the variable or constant a different number in each call.

To achieve a dynamic call, you can also specify a variable of the COUNTER data type.

Examples

```
//Example of an absolute call:
S_CUD (C_NO:=C12,
      CD:=I0.0,
      CU:=I0.1,
      S:=I0.2 & I0.3,
      PV:=120,
      R:=FALSE,
      CV:=binVal,
      Q:=actFlag);

//Example of a dynamic call: In each iteration of a
//FOR loop, a different counter is called:
FUNCTION_BLOCK COUNT
VAR_INPUT
    Count: ARRAY [1..4] of STRUCT
        C_NO: INT;
        PV  : WORD;
    END_STRUCT;
.
.
END_VAR
.
.
FOR I:= 1 TO 4 DO
    S_CD(C_NO:=Count[I].C_NO, S:=true, PV:= Count[I].PV);
END_FOR;

//Example of a dynamic call using a variable of the
//COUNTER data type:
FUNCTION_BLOCK COUNTER
VAR_INPUT
    MYCounter:COUNTER;
END_VAR
.
.
CurrVal:=S_CD (C_NO:=MyCounter,.....);
```

Note

The names of the functions and parameters are the same in both the German and English mnemonics. Only the counter identifier differs (German: Z, English: C).

12.1.3 Supplying Parameters for Counter Functions

The following table provides you with an overview of the parameters for counter functions.

Parameter	Data Type	Description
C_NO	COUNTER INT	Counter number (COUNTER IDENTIFIER); the area depends on the CPU
CD	BOOL	CD input: Count down
CU	BOOL	CU input: Count up
S	BOOL	Input for presetting the counter
PV	WORD	Value in the range between 0 and 999 for initializing the counter (entered as 16#<value>, with the value in BCD format)
R	BOOL	Reset input
Q	BOOL	Output: Status of the counter
CV	WORD	Output: Count value binary

Rules

Since the parameter values (for example, CD:=I0.0) are stored globally, it is not necessary to specify them in certain situations. The following general rules should be observed when supplying parameters with values:

- The parameter for the counter identifier C_NO must be supplied when the function is called. Instead of the absolute counter number (for example, C12), you can also specify a variable or a constant with the INT data type or an input parameter of the COUNTER data type in the call.
- Either the parameter CU (count up) or the parameter CD (count down) must be supplied.
- 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.

Example

```
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:=I0.0, CU:=I0.1, S:=I0.2
        &I0.3, PV:=120, R:=FALSE, CV:=binVal, Q:=actFlag);
    CurrVal := S_CD (C_NO: C10, CD:=FALSE, S:=FALSE, PV:=100,
        R:=TRUE, CV:=binVal, Q:=actFlag);
END_FUNCTION_BLOCK
```

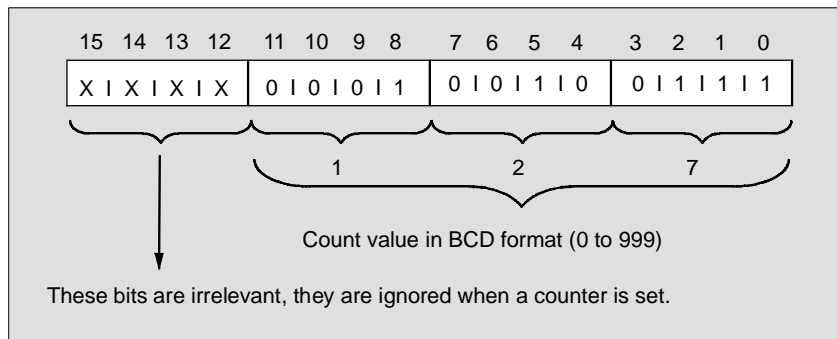
12.1.4 Input and Evaluation of the Counter Value

To input the initialization value or to evaluate the result of the function you require the internal representation of the count value. The count value is of the data type WORD in which bits 0 to 11 contain the count value in BCD code. Bits 12-15 are not used.

When you set the counter, the value you have specified is written to the counter. The range of values is between 0 and 999. You can change the count value within this range by specifying the operations count up/down (S_CUD), count up (S_CU) and count down (S_CD).

Format

The figure below illustrates the bit configuration of the count value.



Input

- Decimal as an integer value: For example, 295, assuming that this value corresponds to a valid BCD format.
- In BCD format (input as a hexadecimal constant): for example 16#127

Evaluation

- As a function result (type WORD): in BCD format
- As the output parameter CV (type WORD): in binary code

12.1.5 Count Up (S_CU)

With the count up (S_CU) function, you can only execute incrementing counter operations. The table illustrates how the counter works.

Operation	Explanation
Count up	The count value is increased by "1" if the signal status at input CU changes from "0" to "1" and the count value is less than 999.
Set counter	When the signal state at input S changes from "0" to "1", the counter is set with 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 count value to "0".
Query counter	A signal status query at output Q returns "1" if the count value is greater than "0". The query returns "0" if the count value is equal to "0".

12.1.6 Count Down (S_CD)

With the count down (S_CD) function, you can only execute decrementing counter operations. The table illustrates how the counter works.

Operation	Explanation
Count down	The value of the counter is decremented by "1" when the signal state at input CD changes from "0" to "1" and the count value is greater than "0".
Set counter	When the signal state at input S changes from "0" to "1", the counter is set with 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 count value to "0".
Query counter	A signal state query at output Q produces "1" when the count value is greater than "0". The query returns "0" if the count value is equal to "0".

12.1.7 Count Up/Down (S_CUD)

With the count up/down (S_CUD) function, you can execute both up and down counter operations. If up and down count pulses are received simultaneously, both operations are executed. The count value remains unchanged. The table illustrates how the counter works.

Operation	Explanation
Count up	The value of the counter is incremented by "1" when the signal state at input CU changes from "0" to "1" and the count value is less than 999.
Count down	The value of the counter is decremented by "1" when the signal state at input CD changes from "0" to "1" and the count value is greater than "0".
Set counter	When the signal state at input S changes from "0" to "1", the counter is set with 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 count value to "0".
Query counter	A signal status query at output Q returns "1" if the count value is greater than "0". The query returns "0" if the count value is equal to "0".

12.1.8 Example of Counter Functions

Parameter Assignment

The table below illustrates the parameter assignment for the function S_CD.

Parameter	Description
C_NO	MYCOUNTER:
CD	INPUT I0.0
S	SET
PV	INITIALVALUE 16#0089
R	RESET
Q	Q0.7
CV	BIN VALUE

Example

```

FUNCTION_BLOCK COUNT
VAR_INPUT
    MYCOUNTER    : COUNTER ;
END_VAR
VAR_OUTPUT
    RESULT : INT ;
END_VAR
VAR
    SET          : BOOL ;
    RESET        : BOOL ;
    BCD_VALUE     : WORD ;      // Count value BCD coded
    BIN_VALUE     : WORD ;      // Count value binary
    INITIALVALUE : WORD ;
END_VAR
BEGIN
    Q0.0          := 1 ;
    SET           := I0.2 ;
    RESET         := I0.3 ;
    INITIALVALUE  := 16#0089 ;
    //Count down
    BCD_VALUE := S_CD (C_NO := MYCOUNTER,
        CD      := I0.0 ,
        S       := SET ,
        PV      := INITIALVALUE,
        R       := RESET ,
        CV      := BIN_VALUE ,
        Q       := Q0.7) ;
    //Further processing as output parameter
    RESULT      := WORD_TO_INT (BIN_VALUE) ;
    QW4         := BCD_VALUE ;
END_FUNCTION_BLOCK

```

12.2 Timers

12.2.1 Timer Functions

Timers are function elements in your program, that execute and monitor time-controlled functions. STEP 7 provides a series of standard timer functions that you can use in your SCL program.

Timer Function	Explanation
S_PULSE	Start timer as pulse timer
S_PEXT	Start timer as extended pulse timer
S_ODT	Start timer as on-delay timer
S_ODTS	Start timer as retentive on-delay timer
S_OFFDT	Start timer as off-delay timer

12.2.2 Calling Timer Functions

Timer functions are called like functions. The function identifier can therefore be used anywhere instead of an address in an expression as long as the type of the function result is compatible with the first replaced address.

The function value (return value) that is returned to the calling block is a time value of the data type S5TIME.

Absolute or Dynamic Call

In the call, you can enter an absolute value, (for example T_NO:=T10) of the TIMER data type as the number of the timer function. Such values can, however, no longer be modified during runtime.

Instead of the absolute number, you can also specify a variable or constant of the INT data type. The advantage of this method is that the call can be made dynamic by assigning the variable or constant a different number in each call.

To achieve a dynamic call, you can also specify a variable of the TIMER data type.

Examples

```
//Example of an absolute call:
S_ODT (T_NO:=T10,
      S:=TRUE,
      TV:=T#1s,
      R:=FALSE,
      BI:=biVal,
      Q:=actFlag);

//Example of a dynamic call: In each iteration of a
//FOR loop, a different timer function is called:
FUNCTION_BLOCK TIMER
VAR_INPUT
  MY_TIMER: ARRAY [1..4] of STRUCT
    T_NO: INT;
    TV  : WORD;
  END_STRUCT;
.
.
END_VAR
.
.
FOR I:= 1 TO 4 DO
  S_ODT(T_NO:=MY_TIMER[I].T_NO, S:=true,
    TV:= MY_TIMER[I].TV);
END_FOR;

//Example of a dynamic call using a variable of the
//TIMER data type:
FUNCTION_BLOCK TIMER
VAR_INPUT
  mytimer:TIMER;
END_VAR
.
.
CurrTime:=S_ODT (T_NO:=mytimer,.....);
```

Note

The names of the functions are the same in both the German and English mnemonics.

12.2.3 Supplying Parameters for Timer Functions

The following table shows an overview of the parameters of the timer functions:

Parameter	data type	Description
T_NO	TIMER INTEGER	Identification number of the timer; the range depends on the CPU
S	BOOL	Start input
TV	S5TIME	Initialization of the timer value (BCD format)
R	BOOL	Reset input
Q	BOOL	Status of the timer
BI	WORD	Time remaining (binary)

Rules

Since the parameter values are stored globally, it is not necessary to specify these value in certain situations. The following general rules should be observed when assigning values to parameters:

- The parameters for the timer identifier T_NO must be supplied when the function is called. Instead of the absolute timer number (for example, T10), you can also specify a variable of the INT data type or an input parameter of the TIMER data type in the call.
- The parameters PV (initialization value) and S (set) can be omitted as a pair.
- The result value in S5TIME format is always the function value.

Example

```
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
```

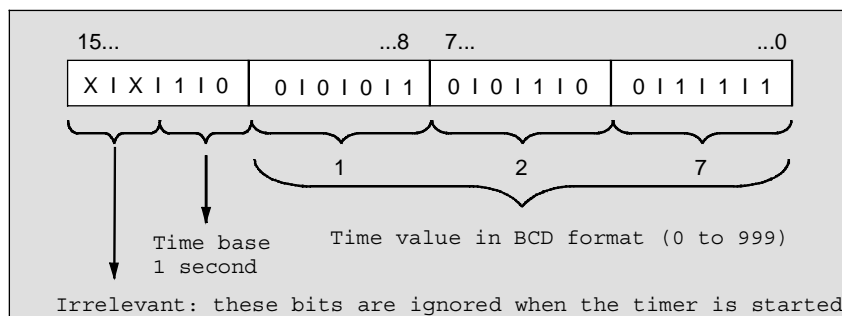
12.2.4 Input and Evaluation of a Time Value

To input the initial value and to evaluate the function result in BCD code, you require the internal representation of the time value. The time value is of the WORD data type, where bits 0 to 11 contain the time value in BCD format and bits 12 and 13 the time base. Bits 14 and 15 are not used.

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 9990 seconds.

Format

The figure below illustrates the bit configuration of the time value.



Input

You can load a predefined time value with the following formats:

- In composite time format
- In decimal time format

The time base is selected automatically in both cases and the value is rounded down to the next number in this time base.

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 for Time Values

To input and evaluate the time value, you require a time base (bits 12 and 13 of the timer word). The time base defines the interval at which the time value is decremented by one unit (see table). The smallest time base is 10 ms; the largest 10 s.

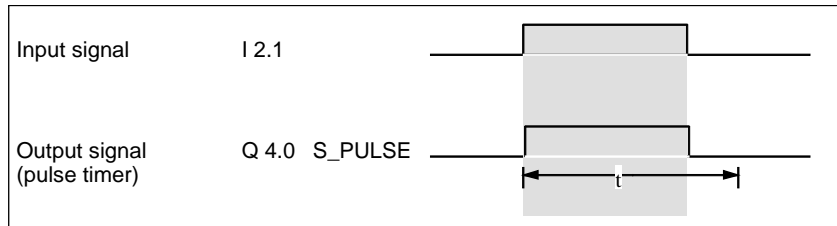
Time Base	Binary Code for Time Base
10 ms	00
100 ms	01
1 s	10
10 s	11

Note

Since time values are only saved at intervals, values that are not 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 but not the required resolution is achieved.

12.2.5 Start Timer as Pulse Timer (S_PULSE)

The maximum time for which the output signal remains set to "1" is the same as the programmed time value. If, during the run time of the timer, the signal state 0 appears at the input, the timer is set to "0". This means a premature termination of the timer runtime.



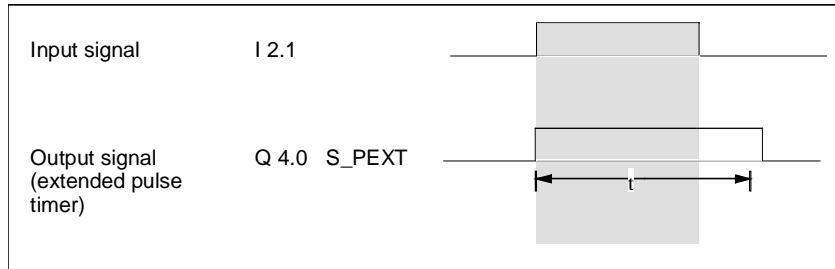
How the Timer Functions

The table shows how the "pulse timer" function works:

Operation	Explanation
Start time	The "pulse timer" operation starts the specified time when the signal state 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 is set.
Abort runtime	If input S changes from "1" to "0" before the time has expired, the timer is stopped.
Reset	The time is reset when 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 state "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 state query following a "1" at output Q produces the result "1". If the timer is aborted, a signal state 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 .

12.2.6 Start Timer as Extended Pulse Timer (S_PEXT)

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 time so that the output pulse is extended (retriggering).



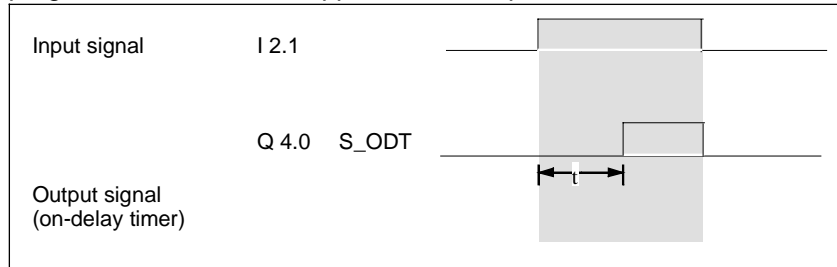
How the Timer Functions

The table shows how the "extended pulse timer" function works.:

Operation	Explanation
Start time	The "extended pulse timer" (S_PEXT) operation starts the specified time when the signal state 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 state at input S changes to "1" again while the timer is running, the timer is restarted with the specified time value.
Initialize runtime	The timer runs with the value at input TV until the programmed time has expired.
Reset	The time is reset when 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 state "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 state query following "1" at output Q produces the result "1" regardless of the length of the input signal.
Query current timer reading	The current time value can be queried at output BI and using the function value S_PEXT.

12.2.7 Start Timer as On-Delay Timer (S_ODT)

The output signal only changes from "0" to "1" when 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.



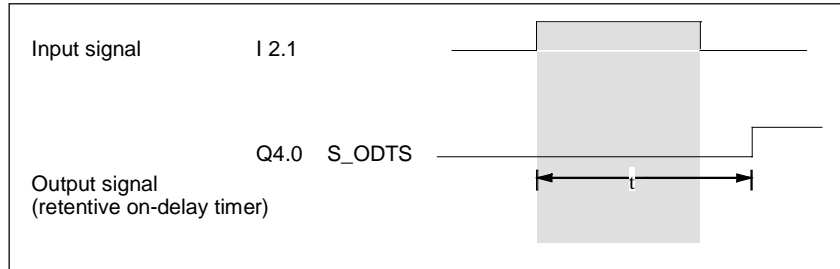
How the Timer Functions

The table illustrates how the "on delay timer" function works.

Operation	Explanation
Start time	The "on delay timer" starts a specified time when the signal state 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 state at input S changes from "1" to "0" while the timer is running, it is stopped.
Specify runtime	The timer continues to run with the value at input TV as long as the signal state at input S = 1 is set.
Reset	The time is reset when 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 time is also reset when R = 1 is set although the timer is not running.
Query signal status	A signal state query following "1" at output Q produces "1" when the time expired without an error occurring and input S is still set to "1". If the timer was stopped, a signal status query following "1" always produces "0". A signal state query after "1" at output Q also produces "0" when the timer is not running and the signal state at input S is still "1".
Query current timer reading	The current time value can be queried at output BI and using the function value S_ODT.

12.2.8 Start Timer as Retentive On-Delay Timer (S_ODTS)

The output signal only changes from "0" to "1" when the programmed time has expired regardless of how long the input signal remains set to "1".



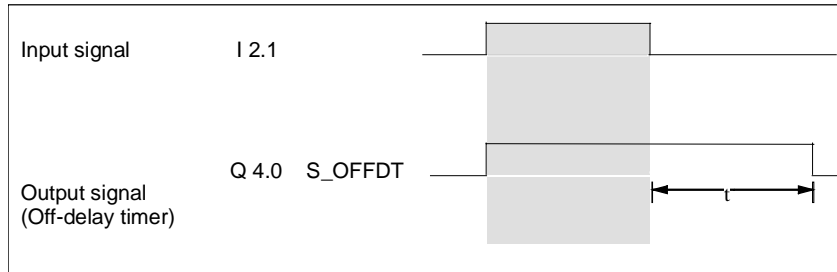
How the Timer Functions

The table shows how the "retentive on delay timer" function works.

Operation	Explanation
Start time	The "retentive on delay timer" starts a specified time when the signal state 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 when 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 state 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 signal state at input S .
Query signal status	A signal state query following "1" at output Q produces the result "1" after the time has expired regardless of the signal state at input S .
Query current timer reading	The current time value can be queried at output BI and using the function value S_ODTS.

12.2.9 Start Timer as Off-Delay Timer (S_OFFDT)

With a signal state change from "0" to "1" at start input S, a "1" is set at output Q. 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.



How the Timer Functions

The table shows how the "off delay timer" function works.

Operation	Explanation
Start time	The "off delay timer" operation starts the specified time when the signal state 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 when the signal state 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 state query following "1" at output Q produces "1" if the signal state at input S = 1 is set or the timer is running.
Query current timer reading	The current time value can be queried at output BI and using the function value S_OFFDT.

12.2.10 Example of Timer Functions

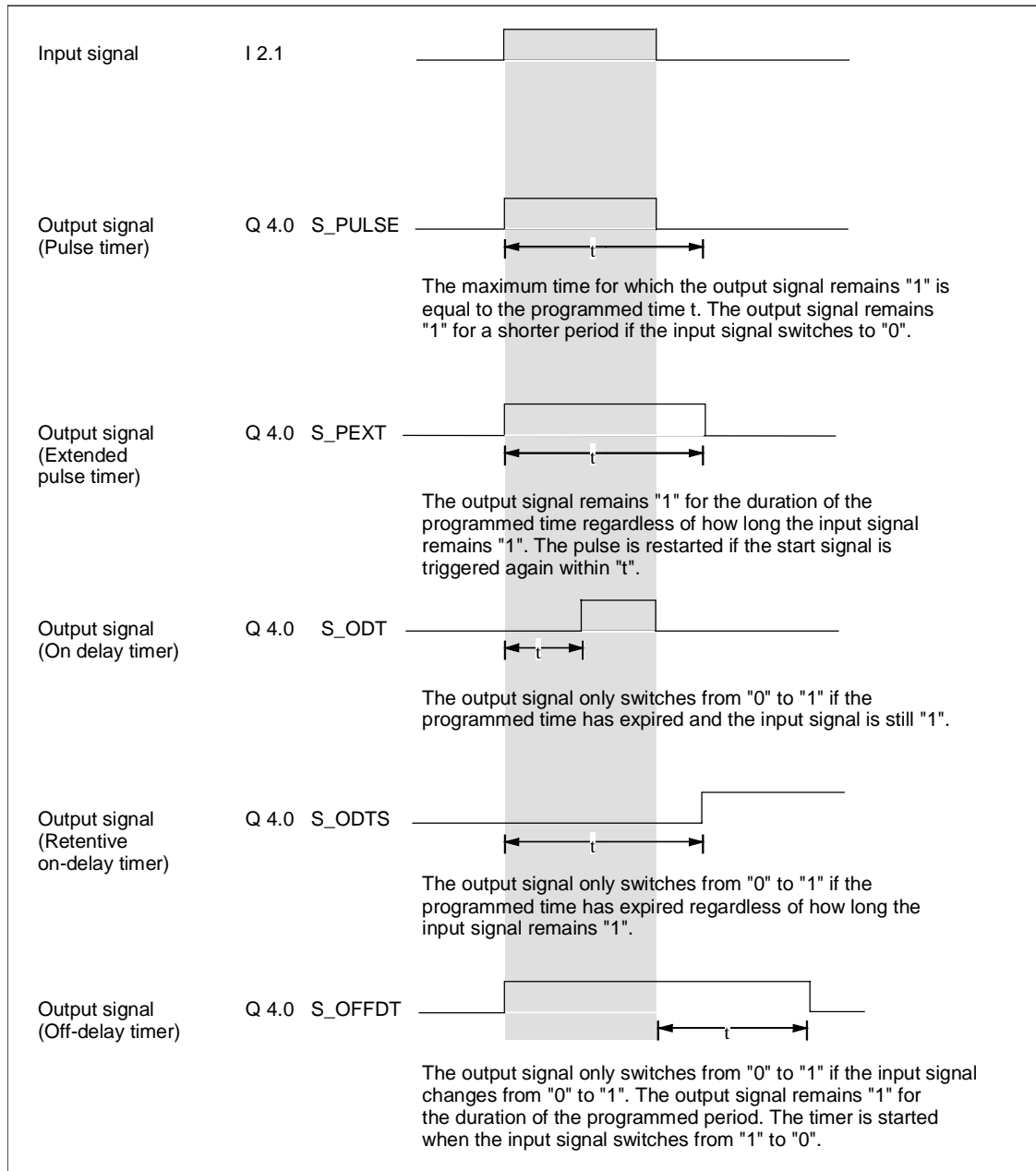
```

FUNCTION_BLOCK TIMER
VAR_INPUT
    mytime : TIMER ;
END_VAR
VAR_OUTPUT
    result : S5TIME ;
END_VAR
VAR
    set      : BOOL ;
    reset    : BOOL ;
    bcdvalue : S5TIME ; //Time base and time remaining in BCD
    binvalue : WORD ; //Time value in binary
    initialvalue : S5TIME ;
END_VAR
BEGIN
    Q0.0 := 1;
    set   := I0.0 ;
    reset := I0.1;
    initialvalue := T#25S ;
    bcdvalue := S_PEXT (T_NO := mytime ,
                        S      := set ,
                        TV     := initialvalue ,
                        R      := reset ,
                        BI     := binvalue ,
                        Q       := Q0.7) ;
    //Further processing as output parameter
    result := bcdvalue ;
    //To output for display
    QW4 := binvalue ;
END_FUNCTION_BLOCK

```

12.2.11 Selecting the Right Timer

The following figure provides an overview of the five different timer functions described in this section. This overview will help you to select the timer function best suited to your particular purpose.



13 SCL Standard Functions

13.1 Data Type Conversion Functions

13.1.1 Converting Data Types

If you use two addresses in a logic operation, you must make sure that the data types of the addresses are compatible. If the addresses are of different data types, a data type conversion is necessary. The following data type conversions are possible in SCL:

- Implicit Data Type Conversion

The data types are grouped in classes. If the addresses fall within the same class, SCL makes an implicit data type conversion. The functions used by the compiler are grouped in "Conversion Functions Class A".

- Explicit Data Type Conversion

If the addresses do not belong to the same class, you must start a conversion function yourself. To allow explicit data type conversion, SCL provides numerous standard functions grouped in the following classes:

- Conversion Functions Class B
- Functions for Rounding and Truncating

13.1.2 Implicit Data Type Conversion

13.1.2.1 Implicit Data Type Conversion

Within the classes of data types defined in the table, the compiler makes an implicit data type conversion in the order shown. The common format of two addresses is always the smallest data type with a range that accommodates both addresses - the common format of BYTE and INTEGER, for example, is INTEGER.

Remember that a data type conversion in the ANY_BIT class results in leading bits being set to 0.

Classes	Conversion Order
ANY_BIT	BOOL > BYTE > WORD > DWORD
ANY_NUM	INT > DINT > REAL

Example of Implicit Data Type Conversion

```
VAR
    PID_CTRLER_1 : BYTE ;
    PID_CTRLER_2 : WORD ;
END_VAR
BEGIN
    IF (PID_CTRLER_1 <> PID_CTRLER_2) THEN ...
    (* In the IF statement above, PID_CTRLER_1 is converted
    implicitly from BYTE to WORD. *)
```

13.1.2.2 Conversion Functions Class A

The table shows the data type conversion functions of Class A. These functions are executed implicitly by the compiler, however, you can also specify them explicitly if you prefer. The result is always defined.

Function Name	Conversion Rule
BOOL_TO_BYTE	Adds leading zeros
BOOL_TO_DWORD	Adds leading zeros
BOOL_TO_WORD	Adds leading zeros
BYTE_TO_DWORD	Adds leading zeros
BYTE_TO_WORD	Adds leading zeros
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 accuracy 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

13.1.3 Standard Functions for Explicit Data Type Conversion

You will find a general description of the function call in "Calling Functions".

Remember the following points when calling the conversion functions:

- **Input parameters:**
Each function for converting a data type has exactly one input parameter with the name IN. Since it is a function with only one parameter, this does not need to be specified.
- **Function value**
The result is always the function value.
- **Names of the functions**
The data types of the input parameter and the function value can clearly be recognized in the function name in the overview of classes A and B. 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`.

13.1.3.1 Conversion Functions Class B

The table shows the data type conversion functions of Class B. These functions must be specified explicitly. The result can also be undefined if the destination type is not large enough.

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 sets the OK flag to `FALSE`.

Function Name	Conversion Rule	OK
<code>BYTE_TO_BOOL</code>	Copies the least significant bit	Y
<code>BYTE_TO_CHAR</code>	Copies the bit string	N
<code>CHAR_TO_BYTE</code>	Copies the bit string	N
<code>CHAR_TO_INT</code>	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
<code>DATE_TO_DINT</code>	Copies the bit string	N
<code>DINT_TO_DATE</code>	Copies the bit string	Y
<code>DINT_TO_DWORD</code>	Copies the bit string	N
<code>DINT_TO_INT</code>	Copies the bit for the sign. The value in the input parameter is interpreted in the data type <code>INT</code> . If the value is less than <code>-32_768</code> or greater than <code>32_767</code> , the OK variable is set to <code>FALSE</code> .	Y
<code>DINT_TO_TIME</code>	Copies the bit string	N
<code>DINT_TO_TOD</code>	Copies the bit string	Y
<code>DWORD_TO_BOOL</code>	Copies the least significant bit	Y
<code>DWORD_TO_BYTE</code>	Copies the 8 least significant bits	Y
<code>DWORD_TO_DINT</code>	Copies the bit string	N
<code>DWORD_TO_REAL</code>	Copies the bit string	N
<code>DWORD_TO_WORD</code>	Copies the 16 least significant bits	Y
<code>INT_TO_CHAR</code>	Copies the bit string	Y
<code>INT_TO_WORD</code>	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
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 8 least significant 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 further IEC functions for data type conversion. For information about the functions, refer to the reference manual "System Software for S7-300 and S7-400, System and Standard Functions".

13.1.3.2 Functions for Rounding and Truncating

The functions for rounding and truncating numbers also belong to the data type conversion functions. The table shows the names, data types (for the input parameters and the function value) and tasks of these functions:

Function Name	Data Type Input Parameter	Data Type Function Value	Task
ROUND	REAL	DINT	Rounds (forming a DINT number) In compliance with DIN EN 61131-3, the function always rounds to the next even integer value; in other words, 1.5 returns 2 and 2.5 also returns 2.
TRUNC	REAL	DINT	Truncates (forming a DINT number)

Note

You also have the option of using further IEC functions for data type conversion. For information about the functions, refer to the reference manual "System Software for S7-300 and S7-400, System and Standard Functions".

Example

```
// Rounding down (result: 3)
  ROUND (3.14) ;

// Rounding up (result: 4)
  ROUND (3.56) ;

// Truncating (result: 3)
  TRUNC (3.14) ;

// Truncating (result: 3)
  TRUNC (3.56) ;
```


13.1.3.3 Examples of Converting with Standard Functions

In the example 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;
    CTRLER : DINT;
END_VAR

(* INT is lower order than DINT *)
SWITCH := DINT_TO_INT (CTRLER) ;
// . . .
END_FUNCTION_BLOCK
```

In the following example, an explicit data type conversion is necessary, since the data type REAL is not allowed for an arithmetic expression with the MOD operation.

```
FUNCTION_BLOCK FB20
VAR
    SWITCH      : REAL
    INTVALUE    : INT := 17;
    CONV2       : INT ;
END_VAR

(* MOD can only be used with data of the INT or DINT type *)
CONV2 := INTVALUE MOD REAL_TO_INT (SWITCH);
// . . .
END_FUNCTION_BLOCK
```

In the following example, conversion is necessary because the data type is incorrect for a logical operation. The NOT operation can only be used for data of the types BOOL, BYTE, WORD or DWORD.

```
FUNCTION_BLOCK FB30
VAR
    INTVALUE : INT := 17;
    CONV1    : WORD ;
END_VAR

(* NOT must not be applied to data of the INT type *)
CONV1 := NOT INT_TO_WORD (INTVALUE);
// . . .
END_FUNCTION_BLOCK
```

The following example illustrates data type conversion for peripheral inputs/outputs.

```
FUNCTION_BLOCK FB40
VAR
    Radius_in      : WORD ;
    Radius          : INT;
END_VAR

    Radius_in      := %IB0;
    Radius          := WORD_TO_INT (radius_in);
(* Conversion when changing to a different type class. Value
comes from input and is converted for further processing.*)

    Radius          := Radius (area:= circledata.area)
    %QB0           :=WORD_TO_BYTE (INT_TO_WORD(RADIUS));
(*Radius is recalculated from the area and is then as an
integer. For output, the value is first converted to a
different type class (INT_TO_WORD) and then into a lower
order type (WORD_TO_BYTE).*)
// . . .
END_FUNCTION_BLOCK
```

13.2 Numeric Standard Functions

13.2.1 General Arithmetic Standard Functions

These are the functions for calculating the absolute value, the square or the square root of a value.

The data type ANY_NUM stands for INT, DINT or REAL. Note that input parameters of the INT or DINT type are converted internally to REAL variables if the function value is of the REAL type.

Function Name	Data Type Input Parameter	Data Type Function Value	Description
ABS	ANY_NUM	ANY_NUM	Number
SQR	ANY_NUM	REAL	Square
SQRT	ANY_NUM	REAL	Square Root

Note

You also have the option of using further IEC functions for data type conversion. For information about the functions, refer to the reference manual "System Software for S7-300 and S7-400, System and Standard Functions".

13.2.2 Logarithmic Functions

These are functions for calculating an exponential value or a logarithm of a value.

The data type ANY_NUM stands for INT, DINT or REAL. Note that input parameters of the type ANY_NUM are converted internally into real variables.

Function Name	Data Type Input Parameter	Data Type Function Value	Description
EXP	ANY_NUM	REAL	e to the power IN
EXPD	ANY_NUM	REAL	10 to the power IN
LN	ANY_NUM	REAL	Natural logarithm
LOG	ANY_NUM	REAL	Common logarithm

Note

You also have the option of using further IEC functions for data type conversion. For information about the functions, refer to the reference manual "System Software for S7-300 and S7-400, System and Standard Functions".

13.2.3 Trigonometric Functions

The trigonometric functions represented in the table calculate values of angles in radians.

The data type ANY_NUM stands for INT, DINT or REAL. Note that input parameters of the type ANY_NUM are converted internally into real variables.

Function Name	Data Type Input Parameter	Data Type Function Value	Description
ACOS	ANY_NUM	REAL	Arc cosine
ASIN	ANY_NUM	REAL	Arc sine
ATAN	ANY_NUM	REAL	Arc tangent
COS	ANY_NUM	REAL	Cosine
SIN	ANY_NUM	REAL	Sine
TAN	ANY_NUM	REAL	Tangent

Note

You also have the option of using further IEC functions for data type conversion. For information about the functions, refer to the reference manual "System Software for S7-300 and S7-400, System and Standard Functions".

13.2.4 Examples of Numeric Standard Functions

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)

13.3 Bit String Standard Functions

Every bit string standard function has two input parameters identified by IN and N. The result is always the function value. The following table lists the function names and data types of the two input parameters and the function value. Explanation of input parameters:

- Input parameter IN: buffer in which bit string operations are performed. The data type of this input parameter decides the data type of the function value.
- 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.

The table shows the possible bit string standard functions.

Function Name	Data Type Input Parameter IN	Data Type Input Parameter N	Data Type Function Value	Task
ROL	BOOL BYTE WORD DWORD	INT INT INT INT	BOOL BYTE WORD DWORD	The value in the parameter IN is rotated left by the number of bit places specified by the content of parameter N.
ROR	BOOL BYTE WORD DWORD	INT INT INT INT	BOOL BYTE WORD DWORD	The value in the parameter IN is rotated right by the number of bit places specified by the content of parameter N.
SHL	BOOL BYTE WORD DWORD	INT INT INT INT	BOOL BYTE WORD DWORD	The value in the parameter IN is shifted as many places left and as many bit places on the right-hand side are replaced by 0 as specified by the parameter N.
SHR	BOOL BYTE WORD DWORD	INT INT INT INT	BOOL BYTE WORD DWORD	The value in the parameter IN is shifted as many places right and as many bit places on the left-hand side are replaced by 0 as specified by the parameter N.

Note

You also have the option of using further IEC functions for data type conversion. For information about the functions, refer to the reference manual "System Software for S7-300 and S7-400, System and Standard Functions".

13.3.1 Examples of Bit String Standard Functions

Call	Result
RESULT := ROL (IN:=BYTE#2#1101_0011, N:=5);	//2#0111_1010 //(= 122 decimal)
RESULT := ROR (IN:=BYTE#2#1101_0011, N:=2);	//2#1111_0100 //(= 244 decimal)
RESULT := SHL (IN:=BYTE#2#1101_0011, N:=3);	//2#1001_1000 //(= 152 decimal)
RESULT := SHR (IN:=BYTE#2#1101_0011, N:=2);	//2#0011_0100 //(= 52 decimal)

13.4 Functions for Processing Character Strings

13.4.1 Functions for String Manipulation

LEN

The LEN function (FC21) returns the current length of a string (number of valid characters). An empty string (") has zero length. The function does not report errors.

Example `LEN (S:= 'XYZ')`

Parameter	Declaration	Data Type	Memory Area	Description
S	INPUT	STRING	D, L	Input variable in the STRING format
Return value		INT	I, Q, M, D, L	Current number of characters

CONCAT

The CONCAT function (FC2) combines two STRING variables to form a string. If the resulting string is longer than the variable at the output parameter, the resulting string is limited to the maximum length. Errors can be queried at the OK flag.

Example `CONCAT (IN1:= 'Valve', IN2:= ' open')`

Parameter	Declaration	Data Type	Memory Area	Description
IN1	INPUT	STRING	D, L	Input variable in the STRING format
IN2	INPUT	STRING	D, L	Input variable in the STRING format
Return value		STRING	D, L	Resulting string

LEFT or RIGHT

The LEFT and RIGHT functions (FC20 and FC32) return the first or last L characters of a string. If L is higher than the current length of the STRING variable, the complete string is returned. If L = 0, an empty string is returned. If L is negative, an empty string is output and the OK flag is set to "0".

Example LEFT (IN:= 'Valve', L:= 4)

Parameter	Declaration	Data Type	Memory Area	Description
IN	INPUT	STRING	D, L	Input variable in the STRING format
L	INPUT	INT	I, Q, M, D, L, const.	Length of the left string
Return value		STRING	D, L	Output variable in the STRING format

MID

The MID function (FC26) returns part of a string. L is the length of the string that will be read out, P is the position of the first character to be read out. If the sum of L and (P-1) is longer than the current length of the STRING variable, a string is returned that starts at the character indicated by P and extends up to the end of the input value. In all other situations (P is outside the current length, P and/or L equal to zero or negative), an empty string is output and the OK flag is set to "0".

Example MID (IN:= 'Temperature', L:= 2, P:= 3)

Parameter	Declaration	Data Type	Memory Area	Description
IN	INPUT	STRING	D, L	Input variable in the STRING format
L	INPUT	INT	I, Q, M, D, L, const.	Length of the mid string section
P	INPUT	INT	I, Q, M, D, L, const.	Position of the first character
Return value		STRING	D, L	Output variable in the STRING format

INSERT

The INSERT function (FC17) inserts the character string at parameter IN2 into the string at parameter IN1 after the character identified by P. If P equals zero, the second string is inserted before the first string. If P is higher than the current length of the first string, the second string is appended to the first string. If P is negative, an empty string is output and the OK flag is set to "0". The OK flag is also set to "0" when the resulting string is longer than the variable specified at the output parameter; in this case, the resulting string is limited to the configured maximum length.

Example INSERT (IN1:= 'Participant arrived', IN2:='Miller':= 2, P:= 11)

Parameter	Declaration	Data Type	Memory Area	Description
IN1	INPUT	STRING	D, L	STRING variable into which string will be inserted
IN2	INPUT	STRING	D, L	STRING variable to be inserted
P	INPUT	INT	I, Q, M, D, L, const.	Insert position
Return value		STRING	D, L	Resulting string

DELETE

The DELETE function (FC 4) deletes L characters in a string starting at the character identified by P (inclusive). If L and/or P equals zero or if P is higher than the current length of the input string, the input string is returned. If the sum of L and P is higher than the input string length, the string is deleted up to the end. If L and/or P is negative, an empty string is output and the OK flag set to "0".

Example: DELETE (IN:= 'Temperature ok', L:= 6, P:= 5)

Parameter	Declaration	Data Type	Memory Area	Description
IN	INPUT	STRING	D, L	STRING variable in which characters will be deleted
L	INPUT	INT	I, Q, M, D, L, const.	Number of characters to be deleted
P	INPUT	INT	I, Q, M, D, L, const.	Position of the first character to be deleted
Return value		STRING	D, L	Resulting string

REPLACE

The REPLACE function (FC31) replaces L characters of the first string (IN1) starting at the character identified by P (inclusive) with the second string (IN2). If L equals zero, the first string is returned. If P equals zero or one, the characters are replaced starting at the first character (inclusive). If P is outside the first string, the second string is appended to the first string. If L and/or P is negative, an empty string is output and the OK flag set to "0". The OK flag is also set to "0" when the resulting string is longer than the variable specified at the output parameter; in this case, the resulting string is limited to the configured maximum length.

Example REPLACE (IN1:= 'Temperature', IN2:= ' high' L:= 6, P:= 5)

Parameter	Declaration	Data Type	Memory Area	Description
IN1	INPUT	STRING	D, L	STRING variable in which characters will be replaced
IN2	INPUT	STRING	D, L	STRING variable to be inserted
L	INPUT	INT	I, Q, M, D, L, const.	Number of characters to be replaced
P	INPUT	INT	I, Q, M, D, L, const.	Position of the first replaced character
Return value		STRING	D, L	Resulting string

FIND

The FIND function (FC11) returns the position of the second string (IN2) within the first string (IN1). The search begins at the left; the first occurrence of the string is reported. If the second string does not occur in the first string, zero is returned. The function does not report errors.

Example FIND (IN1:= 'Processingstation', IN2:='station')

Parameter	Declaration	Data Type	Memory Area	Description
IN1	INPUT	STRING	D, L	STRING variable to search
IN2	INPUT	STRING	D, L	STRING variable to search for
Return value		INT	I, Q, M, D, L	Position of the located string

13.4.2 Functions for Comparing Strings

You can compare strings using the SCL comparison functions ==, <>, <, >, <= and >=. The compiler includes the required function call automatically. The following functions are listed simply to provide you with a complete picture.

EQ_STRNG and NE_STRNG

The EQ_STRNG (FC10) and NE_STRNG (FC29) functions compare the contents of two variables in the STRING format for equality (FC10) or inequality (FC29) and return the result of the comparison. The return value has signal state "1" if the string of parameter S1 equals (does not equal) the string of parameter S2. The function does not report errors.

Parameter	Declaration	Data Type	Memory Area	Description
S1	INPUT	STRING	D, L	Input variable in the STRING format
S2	INPUT	STRING	D, L	Input variable in the STRING format
Return value		BOOL	I, Q, M, D, L	Comparison result

GE_STRNG and LE_STRNG

The GE_STRNG (FC13) and LE_STRNG (FC19) functions compare the contents of two variables in the STRING format for greater than (less than) or equal to and return the result of the comparison. The return value has signal state "1" if the string of parameter S1 is greater than (less than) or equal to the string of parameter S2. The characters are compared starting from the left using their ASCII coding (for example, 'a' is greater than 'A'). The first character to differ, decides the result of the comparison. If the left part of the longer string is identical to the shorter string, the longer string is considered to be greater. The function does not report errors.

Parameter	Declaration	Data Type	Memory Area	Description
S1	INPUT	STRING	D, L	Input variable in the STRING format
S2	INPUT	STRING	D, L	Input variable in the STRING format
Return value		BOOL	I, Q, M, D, L	Comparison result

GT_STRNG and LT_STRNG

The GT_STRNG (FC15) and LT_STRNG (FC24) functions compare the values of two variables in the STRING format for greater than (less than) and return the value of the comparison. The return value has signal state "1" if the string of parameter S1 is greater than (less than) the string of parameter S2. The characters are compared starting from the left using their ASCII coding (for example, 'a' is greater than 'A'). The first character to differ, decides the result of the comparison. If the left part of the longer string is identical to the shorter string, the longer string is considered to be greater. The function does not report errors.

Parameter	Declaration	Data Type	Memory Area	Description
S1	INPUT	STRING	D, L	Input variable in the STRING format
S2	INPUT	STRING	D, L	Input variable in the STRING format
RET_VAL		BOOL	I, Q, M, D, L	Comparison result

13.4.3 Functions for Converting the Data Format**I_STRNG and STRNG_I**

The functions I_STRNG (FC16) and STRNG_I (FC38) convert a variable in the INT format into a character string or a string into an INT variable. The string is represented with a leading sign. If the variable specified at the return parameter is too short, no conversion is made and the OK flag is set to "0".

Parameter	Declaration	Data Type	Memory Area	Description
I_STRNG				
I	INPUT	INT	I, Q, M, D, L, const.	Input value
Return value		STRING	D, L	Resulting string
STRNG_I				
S	INPUT	STRING	D, L	Input string
Return value		INT	I, Q, M, D, L	Result

DI_STRNG and STRNG_DI

The functions DI_STRNG (FC5) and STRNG_DI (FC37) convert a variable in the DINT format into a character string or a string into a DINT variable. The string is represented with a leading sign. If the variable specified at the return parameter is too short, no conversion is made and the OK flag is set to "0".

Parameter	Declaration	Data Type	Memory Area	Description
DI_STRNG				
I	INPUT	DINT	I, Q, M, D, L, const.	Input value
Return value		STRING	D, L	Resulting string
STRNG_DI				
S	INPUT	STRING	D, L	Input string
Return value		DINT	I, Q, M, D, L	Result

R_STRNG and STRNG_R

The functions R_STRNG (FC30) and STRNG_R (FC39) convert a variable in the REAL format into a character string or a string into a REAL variable. The string is represented with 14 places:

$\pm v.nnnnnnnE\pm xx$

If the variable specified at the return parameter is too short or if there is no valid floating-point number at the IN parameter, no conversion is made and the OK flag is set to "0".

Parameter	Declaration	Data Type	Memory Area	Description
R_STRNG				
IN	INPUT	REAL	I, Q, M, D, L, const.	Input value
Return value		STRING	D, L	Resulting string
STRNG_R				
S	INPUT	STRING	D, L	Input string
Return value		REAL	I, Q, M, D, L	Result

13.4.4 Example of Processing Character Strings

Putting together message texts

```
//Put message texts together controlled by the process and store
//them
////////////////////////////////////
//The block contains the necessary message texts and the last 20
//messages generated
////////////////////////////////////

DATA_BLOCK Messagetexts

STRUCT
  Index      : int;
  textbuffer : array [0..19] of string[34];
  HW         : array [1..5] of string[16]; //5 different devices
  statuses   : array [1..5] of string[12]; // 5 different statuses
END_STRUCT
BEGIN
  Index := 0;
  HW[1] := 'Motor ';
  HW[2] := 'Valve ';
  HW[3] := 'Press ';
  HW[4] := 'Weldingstation ';
  HW[5] := 'Burner ';
  Statuses[1] := ' problem';
  Statuses[2] := ' started';
  Statuses[3] := ' temperature';
  Statuses[4] := ' repaired';
  Statuses[5] := ' maintained';
END_DATA_BLOCK

////////////////////////////////////
//The function puts message texts together and enters them in
//the DB message texts. The message texts are stored in a
//circulating buffer. The next free index of the text buffer is
//also in the DB message texts and is updated by the function.
////////////////////////////////////

FUNCTION Textgenerator : bool
VAR_INPUT
  unit      : int;      //Index of the device text
  no        : int;      // ID no. of the device
  status    : int;
  value     : int;
END_VAR
VAR_TEMP
  text      : string[34];
  i         : int;
END_VAR
//initialization of the temporary variables
text := '';
```

```

Textgenerator := true;
Case unit of
  1..5 : case status of
    1..5 : text := concat( in1 := Messagetexts.HW[unit],
                          in2 := right(l:=2,in:=I_STRNG(no)));
    text := concat( in1 := text,
                    in2 := Messagetexts.statuses[status]);
    if value <> 0 then
      text := concat( in1 := text,
                      in2 := I_STRNG(value));
    end if;
  else Textgenerator := false;
end case;
else Textgenerator := false;
end case;
i := Messagetexts.index;
Messagetexts.textbuffer[i] := text;
Messagetexts.index := (i+1) mod 20;
END_FUNCTION

/////////////////////////////////////////////////////////////////
//The function is called in the cyclic program at an edge change
//in %M10.0 and a message is entered once if a parameter
//changes.
/////////////////////////////////////////////////////////////////

Organization_block Cycle
Var_temp
  Opsy_ifx : array [0..20] of byte;
  error: BOOL;
End_var;

/////////////////////////////////////////////////////////////////
//The following call enters the message "Motor 12 started" in
//text buffer of DB message texts, with %MW0 supplying a 1, %IW2
//the 12 and %MW2 a 2. *)
/////////////////////////////////////////////////////////////////

if %M10.0 <> %M10.1 then
  error := Textgenerator (unit := word_to_int(%MW0),
                          no := word_to_int(%IW2),
                          status := word_to_int(%MW2),
                          value := 0);
  %M10.1:=M10.0;
end if;
end_organization_block

```

13.5 SFCs, SFBs and Standard Library

The S7 CPUs have integrated system and standard functions in their operating systems that you can use when programming in SCL. Specifically these are the following:

- Organization blocks(OBs)
- System functions (SFCs)
- System function blocks (SFBs)

Call Interface (SFC/SFB)

You can address blocks in symbolic or absolute form. You require either the symbolic name that must be declared in the symbol table or the number of the absolute identifier of the block.

When these functions and blocks are called, you must assign the actual parameters (with the values used by the block when your program runs) to the formal parameters whose names and data types were specified when the configurable block was created.

SCL searches the following folders and libraries for the block to be called:

- The "Programs" folder
- The Simatic standard libraries
- The IEC standard library

If SCL finds a block, it is copied to the user program. The exceptions to this are the blocks that must be called with the notation (" ... ") due to their names and those that are called using the absolute identifier. SCL then searches for these names only in the symbol table of the user program. You must copy these functions into your user program yourself with the SIMATIC Manager. This affects the following IEC functions.

- "DATE and TOD to DT"
- "DT to DATE"
- "DT to DAY"
- "DT to TOD"

Conditional Call (SFB/SFC)

For a conditional function call, you must set the predefined input parameter EN to 0 (for example, via input I0.3). The block is then not called. If EN has the value 1, the function is called. The output parameter ENO is also set to "1" in this case (otherwise to "0") if no error occurred during the execution of the block.

Conditional SFC calls are not recommended since the variable that should normally receive the return value of the function is undefined if the function is not called.

Note

If you use the following operations for the data types TIME, DATE_AND_TIME and STRING in your program, SCL implicitly calls the corresponding standard blocks.

The symbols and block numbers of these standard blocks are therefore reserved and must not be used for other blocks. If you break this rule, it will not always be detected by SCL and can lead to a compiler error.

The following table contains an overview of the IEC standard functions used implicitly by SCL.

Operation	DATE_AND_TIME	STRING
==	EQ_DT (FC9)	EQ_STRING (FC10)
<>	NE_DT (FC28)	NE_STRING (FC29)
>	GT_DT (FC14)	GT_STRING (FC15)
>=	GE_DT (FC12)	GE_STRING (FC13)
<=	LE_DT (FC18)	LE_STRING (FC19)
<	LT_DT (FC23)	LT_STRING (FC24)
DATE_AND_TIME + TIME	AD_DT_TM (FC1)	
DATE_AND_TIME + TIME	SB_DT_TM (FC35)	
DATE_AND_TIME + DATE_AND_TIME	SB_DT_DT (FC34)	
TIME_TO_S5TIME(TIME)	TIM_S5TI (FC40)	
S5TIME_TO_TIME(S5TIME)	S5TI_TIM (FC33)	

For detailed information about available SFBs, SFCs and OBs and a detailed interface description, refer to the reference manual "System Software for S7-300 and S7-400, System and Standard Functions".

13.5.1 Transfer Interface to OBs

Organization Blocks

Organization blocks form the interface between the CPU operating system and the user 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 hardware or communications interrupts are triggered

Organization blocks are processed according to the priority they are assigned.

Available OBs

Not all CPUs can execute all OBs provided by S7. Refer to the data sheets for your CPU to find out which OBs you can use.

14 Language Definition

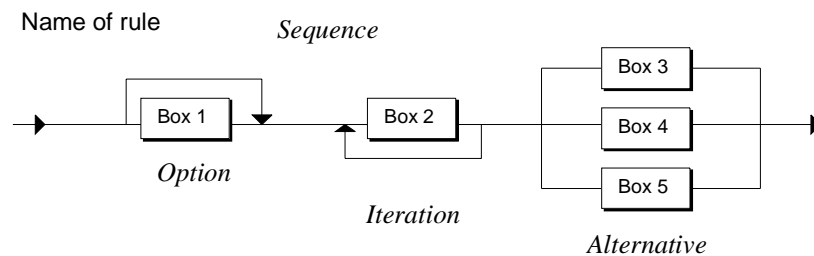
14.1 Formal Language Definition

14.1.1 Overview of Syntax Diagrams

The basic tool for the description of the language in the various sections is the syntax diagram. It provides a clear insight into the structure of SCL syntax. You will find a complete collection of all the diagrams with the language elements in the sections entitled "Lexical Rules" and "Syntax Rules".

What is a Syntax Diagram?

The syntax diagram is a graphic representation of the structure of the language. The structure is defined by a series of rules. One rule may be based on others at a more fundamental level.

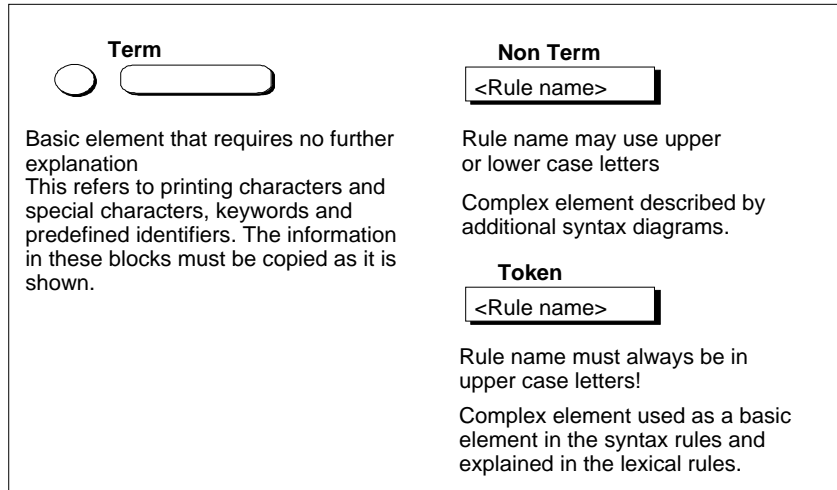


The syntax diagram is read from right to left. The following rule structures must be adhered to:

- Sequence: a sequence of boxes
- Option: a skippable branch
- Iteration: repetition of branches
- Alternative: multiple alternative branches

What Types of Boxes Are There?

A box is a basic element or an element made up of other objects. The diagram below shows the symbols that represent the various types of boxes



14.1.2 Rules

The rules you apply to the structure of your SCL program are subdivided into the categories **lexical** and **syntax** rules.

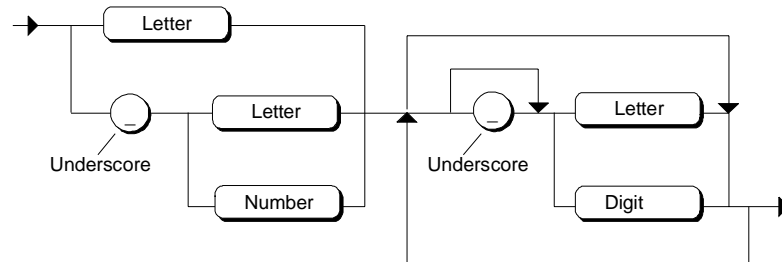
Lexical Rules

The lexical rules describe the structure of the elements (tokens) processed during the lexical analysis performed by the Compiler. For this reason lexical rules do not allow a flexible format and must be strictly observed. In particular, this means that

- Inserting formatting characters is not permitted,
- Section and line comments cannot be inserted.
- Inserting attributes for identifiers is not permitted.

The above example shows the lexical rule for IDENTIFIER. It defines the structure of an identifier (name), for example:

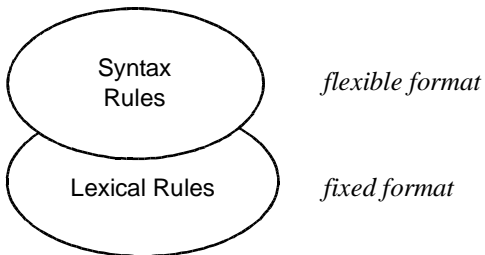
```
MEAS_FIELD_12
SETPOINT_B_1
```



Syntax Rules

The syntax rules are built up from the lexical rules and define the structure of SCL. Within the limitations of these rules, the structure of the your SCL program has a flexible format.

SCL Program



Formal Aspects

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

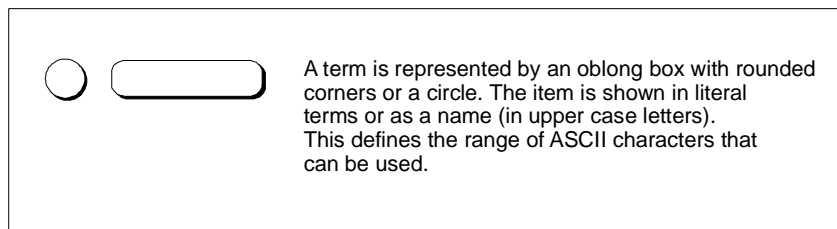
The rules can only represent the formal structure of the language. The meaning (semantics) is not always obvious from the rules. For this reason, where it is important, additional information is written next to the rule. The following are examples of such situations:

- When elements of the same type have a different meaning, an additional name is given: For example in the data specification rule for DECIMALDIGITSTRING year, month or day. The name indicates the usage.
- Important restrictions are noted alongside the rules: For example, you will find a note with the symbol rule telling you that a symbol must be defined in the symbol table.

14.1.3 Terms Used in the Lexical Rules

Definition

A term is a basic element that cannot be explained by another rule but only verbally. In a syntax diagram, it is represented by the following symbol:



The tables below define the terms on the basis of a range of characters from the ASCII character set.

Letters and Numeric Characters

Letters and numeric characters are the main characters used. The IDENTIFIER, for example, consists of letters, numeric characters and the underscore.

Character	Subgroup	Character Set Range
Letter	Uppercase Lowercase	A to Z a to z
Digit	Decimal digits	0.. 9
Octal digit	Octal digits	0.. 7
Hexadecimal digit	Hexadecimal digits	0 to 9, A to F, a to f
Bit	Binary digits	0, 1

Printable Characters and Special Characters

The complete, extended ASCII character set can be used in strings, comments and symbols.

Character	Subgroup	Character Set Range
Printable character	Depends on the character code used. In ASCII code, for example, characters starting at the decimal equivalent of 32 without DEL and without the following substitute characters:	All printing characters
Substitute characters	Dollar sign Quote	\$,
Control characters	\$P or \$p \$L or \$l \$R or \$r \$T or \$t \$N or \$n	form feed line feed carriage return tabulator new line
Substitute representation in hexadecimal code	\$hh	Any characters capable of representation in hexadecimal code (hh)

14.1.4 Formatting Characters, Separators and Operations

Used in the Lexical Rules

The following table shows the characters of the ASCII character set used as formatting characters and separators in the lexical rules.

Character	Description
:	Delimiter between hours, minutes and seconds Attribute
.	Separator for absolute addresses in real number or time period representation
' '	Characters and character strings
" "	Introductory character for symbols according to symbol editor rules
_ Underscore	Separator for numbers in constants and can be used in IDENTIFIERS
\$	Escape symbol for specifying control characters or substitute characters
\$> \$<	String break, in case the string does not fit in one line, or if the comments are to be inserted.

For Constants

The following table shows the use of individual characters and character strings for constants in the lexical rules. The table applies to both the English and German mnemonics.

Prefix	Represents	Lexical Rule
BOOL#	Type-defined constant of type BOOL	BIT constant
BYTE#	Type-defined constant of type BYTE	BIT constant
WORD#	Type-defined constant of type WORD	BIT constant
DWORD#	Type-defined constant of type DWORD	BIT constant
INT#	Type-defined constant of type INT	Integer constant
DINT#	Type-defined constant of type DINT	Integer constant
REAL#	Type-defined constant of type REAL	REAL constant
CHAR#	Type-defined constant of type CHAR	CHAR constant
2#	Numeric constant	Binary digit string
8#	Numeric constant	Octal digit string
16#	Numeric constant	Hexadecimal digit string
D#	Times	DATE
DATE#	Times	DATE
DATE_AND_TIME#	Times	DATE AND TIME

Prefix	Represents	Lexical Rule
DT#	Times	DATE AND TIME
E	Separator for REAL number constant	Exponent
e	Separator for REAL number constant	Exponent
D	Separator for time unit (day)	Days (rule: complex format)
H	Separator for time unit (hours)	Hours: (rule: complex format)
M	Separator for time unit (minutes)	Minutes : (rule: complex format)
MS	Separator for time unit (milliseconds)	Milliseconds: (rule: complex format)
S	Separator for time unit (seconds)	Seconds: (rule: complex format)
T#	Times	TIME PERIOD
TIME#	Times	TIME PERIOD
TIME_OF_DAY#	Times	TIME OF DAY
TOD#	Times	TIME OF DAY

In the Syntax Rules

The following table shows the use of individual characters as formatting characters and separators in the syntax rules and in comments and attributes.

Character	Description	Syntax Rule, Remarks or Attribute
:	Delimiter for type specification in statement after 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, label subsection, component declaration
,	Delimiter for lists and 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, structured 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 shared variables and strings	Array data type specification, STRING data type specification
(* *)	Comment section	See "Lexical Rules"
//	Line comment	See "Lexical Rules"
{ }	Attribute field	For specifying attributes
%	Introduces a direct identifier	To program in conformity with IEC, %M4.0 can be used instead of M4.0.
#	Introduces a non-keyword	Indicates that an identifier is not a keyword, for example, #FOR.

Operations

The following table lists all SCL operations, keywords, for example AND and the common operations as single characters. This table applies to both the English and German mnemonics.

Operation	Description	Example, Syntax Rule
:=	Assignment operation, initial assignment, data type initialization	Value assignment, DB assignment section, constant subsection, output and in/out assignments, input assignment, in/out assignment
+, -	Arithmetic operations: unary operations, sign	Expression, simple expression, exponential expression
+, -, *, / MOD; DIV	Basic arithmetic operations	Basic arithmetic operation, simple multiplication
**	Arithmetic operations: exponential operation	Expression
NOT	Logical operations: negation	Expression
AND, &, OR; XOR,	Basic logic operations	Basic logic operation
<, >, <=, >=, =, <>	Comparison operation	Comparison operation

14.1.5 Keywords and Predefined Identifiers

The following table list the keywords in SCL and the predefined identifiers in alphabetical order. Alongside each one is a description and the syntax rule in which they are used as a term. Keywords are generally independent of the mnemonics.

Keywords	Description	Example, Syntax Rule
AND	Logic operation	Basic logic operation
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
AT	Declares a view of a variable	Variable Declaration
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	DATE_AND_TIME
DINT	Elementary data type for whole numbers (integers), double resolution	Numeric data type
DIV	Operation for division	Basic arithmetic operation, simple multiplication
DO	Introduces statement section for FOR statement	FOR statement, WHILE statement
DT	Elementary data type for date and time	DATE_AND_TIME
DWORD	Elementary data type for double	Bit data type

Keywords	Description	Example, Syntax Rule
	word	
ELSE	Introduces instructions to be executed if condition is not satisfied	IF statement CASE statement
ELSIF	Introduces alternative condition	IF statement
EN	Block clearance flag	
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 label subsection	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 subsection, parameter subsection
END_WHILE	Terminates WHILE statement	WHILE Statement
EXIT	Executes immediate exit from loop	EXIT
FALSE	Predefined Boolean constant: logic condition not true, 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 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 label subsection	Label subsection
MOD	Arithmetic operation for division remainder	Basic arithmetic operation, simple multiplication
NIL	Zero pointer	
NOT	Logic operation, belongs to the unary operations	Expression
OF	Introduces data type specification	Array data type specification, CASE statement
OK	Flag that indicates whether the instructions in a block have been	

Keywords	Description	Example, Syntax Rule
	processed without errors	
OR	Logic operation	Basic logic operation
ORGANIZATION_BLOCK	Introduces an organization block	Organization block
POINTER	Pointer data type, only allowed in parameter declarations in parameter subsection, not processed in SCL	see also Chapter 10
PROGRAM	Introduces the statement section of an FB (synonymous with FUNCTION_BLOCK)	Function block
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	STRUCT 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: Logic condition met, value does not equal 0	
TYPE	Introduces UDT	User-defined data type
VAR	Introduces declaration subsection	Static variables subsection
VAR_TEMP	Introduces declaration subsection	Temporary variables subsection
UNTIL	Introduces terminate condition for REPEAT statement	REPEAT Statement
VAR_INPUT	Introduces declaration subsection	Parameter subsection
VAR_IN_OUT	Introduces declaration subsection	Parameter subsection
VAR_OUTPUT	Introduces declaration subsection	Parameter 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	Function
XOR	Logic operation	Basic logic operation

14.1.6 Address Identifiers and Block Keywords

Shared System Data

The following table lists 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 sorted in the order of the German mnemonics and the corresponding English mnemonics are shown in the second column.

German Mnemonic	English Mnemonic	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

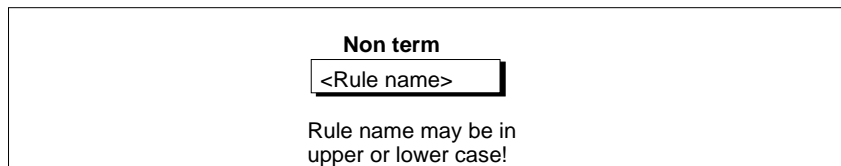
Block Keywords

Used for absolute addressing of blocks. The table is sorted in the order of the German mnemonics and the corresponding English mnemonics are shown in the second column.

German Mnemonic	English Mnemonic	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

14.1.7 Overview of Non Terms

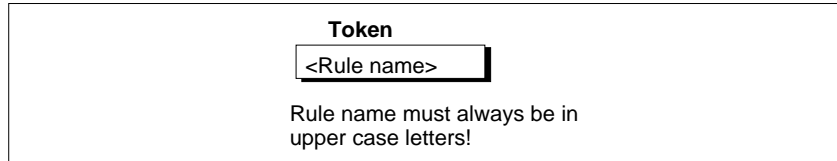
A non term is a complex element that is described by another rule. A non term is represented by an oblong box. The name in the box is the name of the more specific rule.



This element occurs in lexical and syntax rules.

14.1.8 Overview of Tokens

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).



The defined tokens represent identifiers obtained on the basis of lexical rules. Such tokens describe:

- Identifiers
- SCL Naming Conventions
- Predefined constants and flags

14.1.9 Identifiers

Identifier

You can access language objects of SCL using identifiers. The following table shows the classes of identifiers.

Identifier Type	Comments, Examples
Keywords	For example, control statements BEGIN, DO, WHILE
Predefined names	Names of <ul style="list-style-type: none">• Standard data types (for example, BOOL, BYTE, INT)• Predefined standard functions, for example ABS• Standard constants TRUE and FALSE
Absolute address identifiers	For shared 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 <ul style="list-style-type: none">• declared variables• structure components• parameters• declared constants• labels
Symbol editor symbols	Conform either to the lexical rule IDENTIFIER or the lexical rule Symbol; in other words, enclosed in quotes, for example, "xyz"

Upper- and Lowercase

Upper- and lowercase notation is not relevant for the keywords. Since S7 SCL version 4.0, the notation of predefined names and the freely selectable names, such as for variables and symbols from the symbol table is no longer case sensitive. The following table provides an overview.

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
Symbol editor symbols	No

The names of standard functions, such as `BYTE_TO_WORD` and `ABS` can also be written in lowercase characters. This also applies to parameters for timer and counter functions, for example, `SV`, `se` or `CV`.

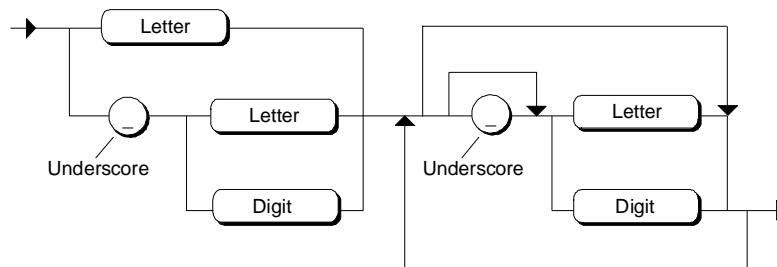
14.1.10 Assigning Names in SCL

Assigning Selectable Names

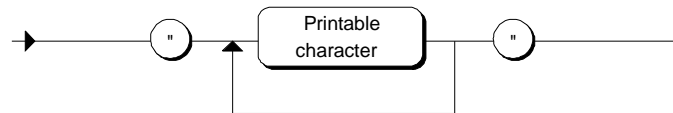
You can assign names in two basic ways:

- You can assign names within SCL itself. These names must conform to the rule IDENTIFIER (see Figure). IDENTIFIER is the general term you can use for any name in SCL.
- Alternatively, you can assign the name in 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



Symbols must be defined in the symbol table.

Rules for Assigning Names

Please remember the following points:

- Choose names that are unambiguous and self-explanatory and which enhance the comprehensibility of the program.
- Check whether the name is already being used by the system, for example by identifiers for data types or standard functions.
- Scope: If you use names with a global scope, the scope covers the entire program. Names with a local scope are valid only within a block. This enables you to use the same names in different blocks. The following table lists the various options available.

Restrictions

When assigning names, remember 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 operations: For example, AND, XOR
- Names of predefined identifiers: For example, 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 shared system data: For example, IB, IW, ID, QB, QW, QD MB, MD

Using IDENTIFIERS

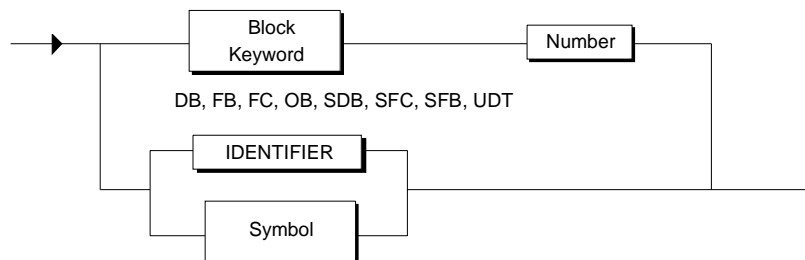
The following table shows the situations in which you can use names that conform to the rule for IDENTIFIERS.

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
Label	Declaration of label, use of label	Labels subsection statement 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 IDENTIFIER

In the BLOCK IDENTIFIER rule, you can use IDENTIFIERS or symbols:

BLOCK IDENTIFIER



The TIMER IDENTIFIER and COUNTER IDENTIFIER rules are analogous to the BLOCK IDENTIFIER rule.

14.1.11 Predefined Constants and Flags

The tables apply to both the German and English mnemonics.

Constants

Mnemonics	Description
FALSE	Predefined Boolean constants (standard constants) with the value 0. The logical meaning is that a condition is not satisfied.
TRUE	Predefined Boolean constants (standard constants) with the value 1. The logical meaning is that a condition is satisfied.

Flags

Mnemonics	Description
EN	Block enable flag
ENO	Block error flag
OK	Flag is set to FALSE if the statement has been incorrectly executed.

14.2 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 do not have a flexible format and must be strictly observed. In particular, this means that

- Inserting formatting characters is not permitted,
- Section and line comments cannot be inserted.
- Inserting attributes for identifiers is not permitted.

14.2.1 Identifiers

Rule	Syntax Diagram
Identifier	<p>IDENTIFIER</p>
Block identifier	<p>BLOCK IDENTIFIER</p>
Timer identifier	

Rule	Syntax Diagram
Counter identifier	
Block keyword	
Symbol	
Number	

Rule	Syntax Diagram
Bit constant	<p>BIT CONSTANT</p> <p>(1) only with BYTE</p>
Integer constant	<p>INTEGER CONSTANT</p> <p>(1) only with INT data type</p>
Real number constant	<p>REAL NUMBER CONSTANT</p>
Decimal digit string	<p>Decimal Digit String</p> <p>Decimal digits: 0-9</p>
Binary digit string	<p>Binary Digit String</p> <p>Binary digits: 0 or 1</p>

Rule	Syntax Diagram
Octal digit string	<p>Octal digit string</p>
Hexadecimal digit string	<p>HEXADECIMAL DIGIT STRING</p> <p>Hexadecimal digit: 0-9 A-F</p>
Exponent	<p>Exponent</p>
Character constant	<p>CHARACTER CONSTANT</p>
String constant	<p>STRING CONSTANT</p>

Rule	Syntax Diagram
Characters	<div>Characters</div> <p>Alternative representation in hex code</p> <p>*P = Form feed L = Line feed R = Carriage return T = Tabulator N = New line</p>
String Break	<div>String Break Syntax</div> <p>Space, Line feed, Carriage return, Form feed, or Tabulator</p>
Date	<div>DATE</div> <p>Details of date</p>
Time period	<div>TIME PERIOD</div> <p>- Each time unit (hours, minutes, etc.) may only be specified once. - The order days, hours, minutes, seconds, milliseconds must be adhered to.</p>
Time of day	<div>TIME OF DAY</div> <p>Time</p>

Rule	Syntax Diagram
Date and time	<p>DATE AND TIME</p> <pre> graph LR Entry(()) --> Choice(()) Choice --> DATE_AND_TIME#(DATE_AND_TIME#) Choice --> DT#(DT#) DATE_AND_TIME# --> Date[Date] DT# --> Date Date --> Hyphen[-] Hyphen --> TimeOfDay[Time of day] TimeOfDay --> Exit(()) </pre>
Date	<p>Date</p> <pre> graph LR Entry(()) --> Year[DECIMAL DIGIT STRING] Year --> Hyphen1[-] Hyphen1 --> Month[DECIMAL DIGIT STRING] Month --> Hyphen2[-] Hyphen2 --> Day[DECIMAL DIGIT STRING] Day --> Exit(()) Year --- YearLabel[Year] Month --- MonthLabel[Month] Day --- DayLabel[Day] </pre>
Time of day	<p>Time of Day</p> <pre> graph LR Entry(()) --> Hours[DECIMAL DIGIT STRING] Hours --> Colon(:) Colon --> Minutes[DECIMAL DIGIT STRING] Minutes --> Choice(()) Choice --> Seconds[DECIMAL DIGIT STRING] Choice --> Milliseconds[DECIMAL DIGIT STRING] Seconds --> Hyphen[-] Hyphen --> Exit(()) Hours --- LongLine[] Minutes --- LongLine LongLine --> Seconds LongLine --> Milliseconds Hours --- HoursLabel[Hours] Minutes --- MinutesLabel[Minutes] Seconds --- SecondsLabel[Seconds] Milliseconds --- MillisecondsLabel[Milliseconds] </pre>
Decimal representation	<p>Simple Time Format</p> <pre> graph LR Entry(()) --> Choice(()) Choice --> D[DECIMAL DIGIT STRING] Choice --> H[DECIMAL DIGIT STRING] Choice --> M[DECIMAL DIGIT STRING] Choice --> S[DECIMAL DIGIT STRING] Choice --> MS[DECIMAL DIGIT STRING] D --> Hyphen1[-] Hyphen1 --> D2[DECIMAL DIGIT STRING] D2 --> DUnit((D)) H --> Hyphen2[-] Hyphen2 --> H2[DECIMAL DIGIT STRING] H2 --> HUnit((H)) M --> Hyphen3[-] Hyphen3 --> M2[DECIMAL DIGIT STRING] M2 --> MUnit((M)) S --> Hyphen4[-] Hyphen4 --> S2[DECIMAL DIGIT STRING] S2 --> SUnit((S)) MS --> Hyphen5[-] Hyphen5 --> MS2[DECIMAL DIGIT STRING] MS2 --> MSUnit((MS)) DUnit --- LongLine[] HUnit --- LongLine MUnit --- LongLine SUnit --- LongLine MSUnit --- LongLine LongLine --> Exit(()) D --- DLabel[Days] H --- HLabel[Hours] M --- MLabel[Minutes] S --- SLabel[Seconds] MS --- MSLabel[Milliseconds] </pre> <p>Use of the simple time format is only possible for undefined time units.</p>

Rule	Syntax Diagram
Composite time format	<p>Composite Time Format</p> <p>Days Hours</p> <p>Minutes Seconds</p> <p>Milliseconds</p>

14.2.3 Absolute Addressing

Rule	Syntax Diagram
Simple memory access	<p>ADDRESS IDENTIFIER</p> <p>Address</p> <p>absolute access</p> <p>IDENTIFIER</p> <p>SYMBOL</p> <p>symbolic access</p>
Indexed memory access	<p>Memory prefix</p> <p>Size prefix</p> <p>[</p> <p>Basic expression</p> <p>,</p> <p>Basic expression</p> <p>]</p> <p>Address identifier</p> <p>Index</p> <p>Bit access only</p>
Address identifier for memory	<p>Memory Prefix</p> <p>Memory prefix</p> <p>Size prefix</p>

Rule	Syntax Diagram
Absolute DB access	<p>Absolute access</p>
Indexed DB access	<p>in the case of bit access</p>
Structured DB access	
Address identifier DB	
Memory prefix	<p>Memory Prefix</p> <p>German mnemonics English mnemonics</p>
Size prefix for memory and DB	<p>Size Prefix</p>
Address for memory and DB	<p>Address</p> <p>Bit address only</p>

Rule	Syntax Diagram
Access to local instance	<pre>graph LR; Start(()) --> IDENTIFIER[IDENTIFIER]; IDENTIFIER --> Dot((.)); Dot --> SimpleVariable[Simple variable]; SimpleVariable --> End(())</pre> <p>Local instance name</p>

14.2.4 Comments

- The following points are the most important things to remember when inserting comments:
- Nesting of comments is permitted if the "Allow nested comments" is activated.
 - They can be inserted at any point in the syntax rules but not in the lexical rules.

Rule	Syntax Diagram
Comment	<pre>graph LR; Start(()) --> Choice(()); Choice --> LineComment[Line comment]; Choice --> CommentSection[Comment section]; LineComment --> End(()); CommentSection --> End(())</pre>
Line comment	<pre>graph LR; Start(()) --> SlashSlash((//)); SlashSlash --> PrintableCharacter[Printable character]; PrintableCharacter --> CR((CR)); CR --> End(())</pre> <p>Line Comment</p>
Comment section	<pre>graph LR; Start(()) --> StarOpen(((*)); StarOpen --> Character[Character]; Character --> StarClose((*))</pre> <p>Comment Section</p>

14.2.5 Block Attributes

Block attributes can be placed after the BLOCK IDENTIFIER and before the declaration of the first variable or parameter subsection using the syntax shown here.

Rule	Syntax Diagram
Title	<pre> graph LR TITLE[TITLE] --> EQ[=] EQ --> Q1[''] Q1 --> PC[Printable character] PC --> Q2[''] Q2 --> End(()) </pre>
Version	<pre> graph LR VERSION[VERSION] --> COLON[:] COLON --> Q1[''] Q1 --> DDS1[DECIMAL DIGIT STRING] DDS1 --> PERIOD[.] PERIOD --> DDS2[DECIMAL DIGIT STRING] DDS2 --> Q2[''] DDS1 --- L1[0 - 15] DDS2 --- L2[0 - 15] Q2 --> End(()) </pre>
Block protection	<pre> graph LR KHP[KNOW_HOW_PROTECT] --> End(()) </pre>
Author	<pre> graph LR AUTHOR[AUTHOR] --> COLON[:] COLON --> IDENTIFIER[IDENTIFIER] IDENTIFIER --> End(()) IDENTIFIER --- L1[max. 8 characters] </pre>
Parameter name	<pre> graph LR NAME[NAME] --> COLON[:] COLON --> IDENTIFIER[IDENTIFIER] IDENTIFIER --> End(()) IDENTIFIER --- L1[max. 8 characters] </pre>
Block family	<pre> graph LR FAMILY[FAMILY] --> COLON[:] COLON --> IDENTIFIER[IDENTIFIER] IDENTIFIER --> End(()) IDENTIFIER --- L1[max. 8 characters] </pre>
System attributes for blocks	<pre> graph LR LBR[{ }] --> IDENTIFIER[IDENTIFIER] IDENTIFIER --> EQ[=] EQ --> Q1[''] Q1 --> PC[Printable character] PC --> Q2[''] Q2 --> RBR[}] IDENTIFIER --- L1[max. 24 characters] RBR --> SEM[;] SEM --> LBR </pre>

14.3 Syntax Rules

The syntax rules are built up from the lexical rules and define the structure of SCL. Within the limitations of these rules, the structure of the your SCL program is flexible.

Each rule has a name that precedes the definition. If the rule is used in a higher-level rule, the name appears in the higher-level rule as a non term.

If the name in the oblong box is in upper case letters, this means it is a token that is described in the lexical rules.

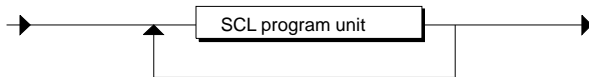
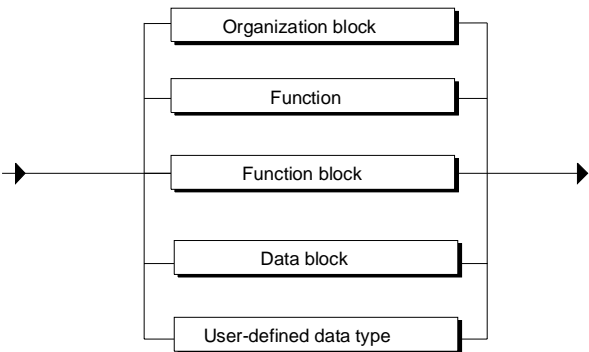
You will find information on rule names in rounded or circular boxes in the section entitled "Formal Language Description".

Flexible Format

Flexible format means:

- You can insert formatting characters at any point.
- You can insert comment lines and comment sections

14.3.1 Structure of SCL Source Files

Rule	Syntax Diagram
SCL program	
SCL program unit	

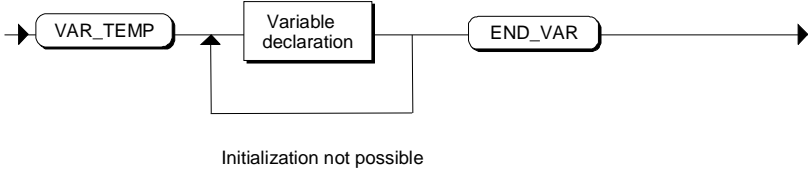
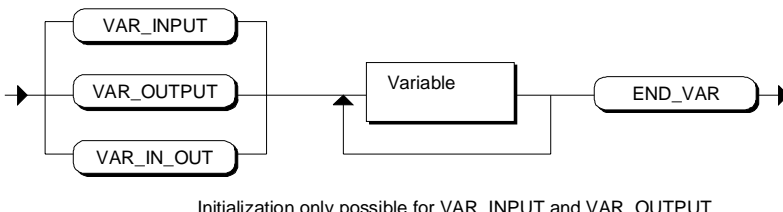
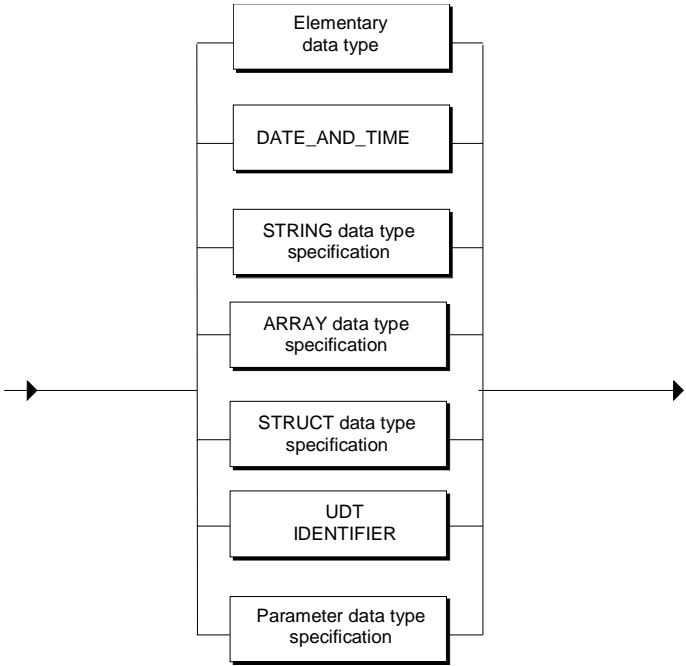
Rule	Syntax Diagram
Organization block	<p>Organization Block</p>
<p>Function</p> <p>Note that if functions do not have VOID in the code section, the return value must be assigned to the function name.</p>	<p>Function</p>
Function block	<p>Function block</p>
Data block	<p>Data Block</p>
User-defined data type	<p>User-Defined Data Type</p>

14.3.2 Structure of the Declaration Sections

Rule	Syntax Diagram
OB declaration section	<p>The diagram shows a sequence of three subsections: 'Constants subsection', 'Labels subsection', and 'Temporary variables subsection'. An entry arrow on the left points to the start of the 'Constants subsection', and an exit arrow on the right points away from the end of the 'Temporary variables subsection'. A feedback loop arrow connects the exit point back to the entry point.</p>
FC declaration section	<p>The diagram shows a sequence of four subsections: 'Constants subsection', 'Labels subsection', 'Temporary variables subsection', and 'Parameters subsection'. An entry arrow on the left points to the start of the 'Constants subsection', and an exit arrow on the right points away from the end of the 'Parameters subsection'. A feedback loop arrow connects the exit point back to the entry point. The label 'Interface' is positioned to the right of the 'Parameters subsection'.</p>
FB declaration section	<p>The diagram shows a sequence of five subsections: 'Constants subsection', 'Labels subsection', 'Temporary variables subsection', 'Static variables subsection', and 'Parameters subsection'. An entry arrow on the left points to the start of the 'Constants subsection', and an exit arrow on the right points away from the end of the 'Parameters subsection'. A feedback loop arrow connects the exit point back to the entry point. The label 'Interface' is positioned to the right of the 'Parameters subsection'.</p>
DB declaration section	<p>The diagram shows two components: 'UDT IDENTIFIER' and 'Structure data type specification'. An entry arrow on the left points to the start of 'UDT IDENTIFIER', and an exit arrow on the right points away from the end of 'Structure data type specification'. A feedback loop arrow connects the exit point back to the entry point.</p>

Rule	Syntax Diagram
DB assignment section	<p>DB Assignment Section</p> <p>* in STL notation</p>
constant subsection	<p>Constant Subsection</p>
Label subsection	<p>Label Subsection</p>
Static variable subsection	<p>Static Variable Section</p> <p>* only with FB</p>

Rule	Syntax Diagram
Variable declaration	<p>Variable name, Parameter name, or Component name</p> <p>Component name within structures</p> <p>Not during initialization</p> <p>1) System attributes for parameters</p> <p>max. 24 characters</p> <p>Printable character</p>
Data type initialization	<p>Initialization</p> <p>Constant</p> <p>Array initialization list</p>
Array initialization list	<p>Array Initialization List</p> <p>Constant</p> <p>Array initialization list</p> <p>Decimal digit string</p> <p>Repeat factor</p>
Instance declaration (possible only in the VAR section of an FB)	<p>Instance Declaration</p> <p>FB NAME</p> <p>SFB NAME</p> <p>FBs must already exist!</p> <p>Local instance name</p>

Rule	Syntax Diagram
Temporary variable subsection	<p>Temporary Variable Subsection</p>  <pre> graph LR Start(()) --> VAR_TEMP([VAR_TEMP]) VAR_TEMP --> Decl[Variable declaration] Decl --> End_VAR([END_VAR]) End_VAR --> Exit(()) Decl -- "Initialization not possible" --> VAR_TEMP </pre>
Parameter subsection	 <pre> graph LR Start(()) --> Group subgraph Group direction TB VAR_INPUT([VAR_INPUT]) VAR_OUTPUT([VAR_OUTPUT]) VAR_IN_OUT([VAR_IN_OUT]) end Group --> Var[Variable] Var --> End_VAR([END_VAR]) End_VAR --> Exit(()) Var -- "Initialization only possible for VAR_INPUT and VAR_OUTPUT" --> Group </pre>
Data type specification	 <pre> graph LR Start(()) --> Stack subgraph Stack direction TB E[Elementary data type] D[DATE_AND_TIME] S[STRING data type specification] A[ARRAY data type specification] ST[STRUCT data type specification] U[UDT IDENTIFIER] P[Parameter data type specification] end Stack --> Exit(()) </pre>

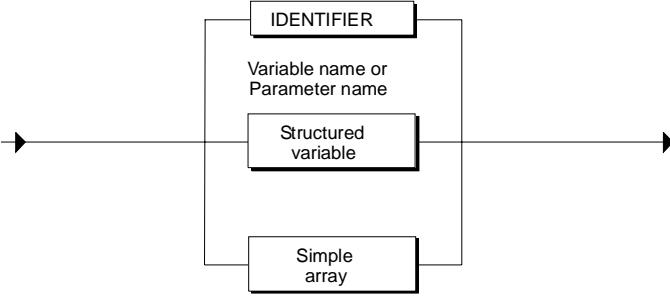
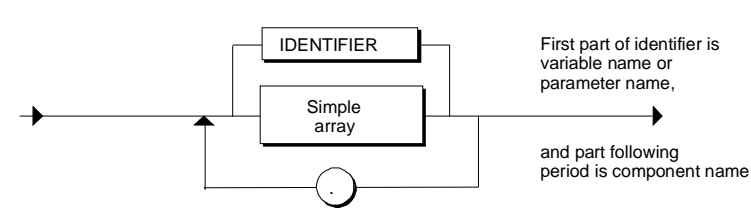
14.3.3 Data Types in SCL

Rule	Syntax Diagram
Elementary data type	
Bit data type	
Character type	
STRING data type specification	<p>STRING Data Type Specification</p>
Numeric data type	
Time type	

Rule	Syntax Diagram
DATE_AND_TIME	<p>DATE_AND_TIME</p>
ARRAY data type specification	<p>ARRAY Data Type Specification</p>
STRUCT data type specification Remember that the END_STRUCT keyword must be terminated by a semicolon.	<p>STRUCT</p>
Component declaration	
Parameter type specification	

14.3.4 Statement Section

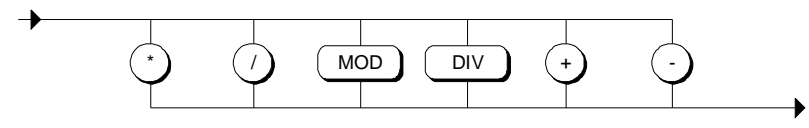
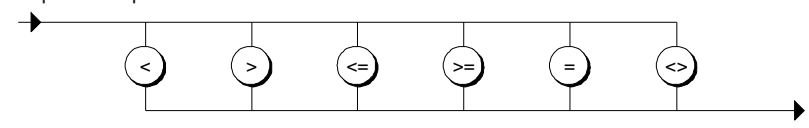
Rule	Syntax Diagram
Statement section	<p>Statement Section</p>
Statement	<p>Statement</p>
Value assignment	<p>Value Assignment</p>
Extended variable	<p>Extended variable</p>

Rule	Syntax Diagram
Simple variable	 <p>The diagram shows a horizontal arrow entering from the left and splitting into three parallel paths. The top path goes through a box labeled 'IDENTIFIER' with the text 'Variable name or Parameter name' below it. The middle path goes through a box labeled 'Structured variable'. The bottom path goes through a box labeled 'Simple array'. The three paths then merge back into a single horizontal arrow exiting to the right.</p>
Structured variable	 <p>The diagram shows a horizontal arrow entering from the left and splitting into two paths. The top path goes through a box labeled 'IDENTIFIER'. The bottom path goes through a box labeled 'Simple array'. After the 'Simple array' box, there is a circular loop symbol. An arrow from the loop goes back to the start of the 'Simple array' box, indicating a loop. Another arrow from the loop goes to the right, leading to the text: 'First part of identifier is variable name or parameter name, and part following period is component name'. The main horizontal arrow continues to the right after the loop.</p>

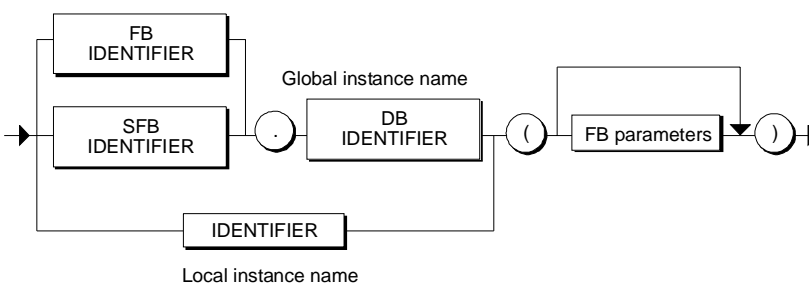
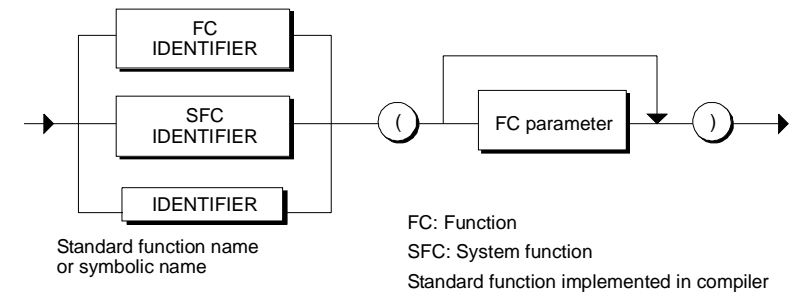
14.3.5 Value Assignments

Rule	Syntax Diagram
Expression	<p>Expression</p> <p>Address</p> <p>Expression</p> <p>Basic logic operations</p> <p>Comparison operations</p> <p>Basic arithmetic operations</p> <p>Exponent</p> <p>Expression</p> <p>Exponent</p> <p>Expression</p> <p>+</p> <p>Unary plus</p> <p>-</p> <p>Unary minus</p> <p>NOT</p> <p>Negation</p> <p>(Expression)</p>
Simple expression	<p>Simple expression</p> <p>+</p> <p>-</p> <p>Simple multiplication</p>
Simple multiplication	<p>Simple multiplication</p> <p>*</p> <p>/</p> <p>DIV</p> <p>MOD</p> <p>Constant</p> <p>Simple expression</p> <p>-</p> <p>(Expression)</p>

Rule	Syntax Diagram
Address	
Extended Variable	<p>Extended variable</p>
Constant	<p>Constant</p>
Exponent	<p>Exponent</p>
Basic logic operation	

Rule	Syntax Diagram
Basic arithmetic operation	<p>Basic Arithmetic Operation</p> 
Comparison operation	<p>Comparison Operation</p> 

14.3.6 Calling Functions and Function Blocks

Rule	Syntax Diagram
FB call	<p>Function Block Call</p> <p>FB: Function block SFB: System function block</p> 
Function call	<p>Function Call</p>  <p>FC: Function SFC: System function Standard function implemented in compiler</p>

Rule	Syntax Diagram
FB parameter	<p>FB Parameters</p>
FC parameters	<p>FC Parameter</p>
Input assignment	<p>Input Assignment</p>
Output or in/out assignment	<p>Output and In/Out Assignments</p>

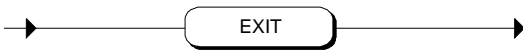
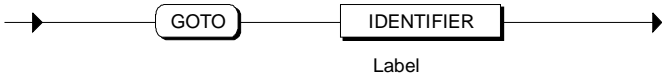
Rule	Syntax Diagram
In/out assignment	<div><p>In/Out Assignment</p><pre>graph LR; Start(()) --> ID[IDENTIFIER]; ID --> OP((:=)); OP --> EV[Extended variable]; EV --> End(())</pre><p>Parameter name of the in/out parameter (formal parameter)</p><p>Actual parameter</p></div>

14.3.7 Control Statements

Rule	Syntax Diagram
<div><p>IF statement</p><p>Remember that the END_IF keyword must be terminated by a semicolon.</p></div>	<div><p>IF Statement</p><pre>graph LR; Start(()) --> IF[IF]; IF --> E1[Expression
Condition]; E1 --> T1[THEN]; T1 --> SS1[Statement section]; SS1 --> E2[ELSIF]; E2 --> E3[Expression
Condition]; E3 --> T2[THEN]; T2 --> SS2[Statement section]; SS2 --> E4[ELSE]; E4 --> SS3[Statement section]; SS3 --> END_IF[END_IF]; END_IF --> End(())</pre></div>
<div><p>CASE statement</p><p>Remember that the END_CASE keyword must be terminated by a semicolon.</p></div>	<div><p>CASE Statement</p><p>Selection expression (Integer)</p><pre>graph LR; Start(()) --> CASE[CASE]; CASE --> E[Expression
Value]; E --> OF[OF]; OF --> VL[Value list]; VL --> C1((:)); C1 --> SS1[Statement section]; SS1 --> ELSE[ELSE]; ELSE --> C2((:)); C2 --> SS2[Statement section]; SS2 --> END_CASE[END_CASE]; END_CASE --> End(())</pre></div>

Rule	Syntax Diagram
Value list	<p>Value List</p> <pre> graph LR Start(()) --> V1[Value] V1 --> Dots((...)) Dots --> V2[Value] V2 --> Comma((,)) Comma --> V1 V2 --> End(()) </pre> <p>Integer</p> <p>Value</p> <p>Value</p> <p>Value</p>
Value	<pre> graph LR Start(()) --> NC[Numeric constant] Start --> ID[IDENTIFIER] ID -- Constant name --> NC NC --> End(()) </pre> <p>Numeric constant</p> <p>IDENTIFIER</p> <p>Constant name</p>
Iteration and jump statements	<pre> graph LR Start(()) --> FOR[FOR statement] FOR --> WHILE[WHILE statement] WHILE --> REPEAT[REPEAT statement] REPEAT --> CONTINUE[CONTINUE statement] CONTINUE --> EXIT[EXIT statement] EXIT --> RETURN[RETURN statement] RETURN --> GOTO[GOTO statement] GOTO --> CONTINUE CONTINUE --> End(()) </pre> <p>FOR statement</p> <p>WHILE statement</p> <p>REPEAT statement</p> <p>CONTINUE statement</p> <p>EXIT statement</p> <p>RETURN statement</p> <p>GOTO statement</p>

Rule	Syntax Diagram
FOR statement Remember that the END_FOR keyword must be terminated by a semicolon.	FOR Statement
Initial assignment	Initial Assignment
WHILE statement Remember that the END_WHILE keyword must be terminated by a semicolon.	WHILE Statement
REPEAT statement Remember that the END_REPEAT keyword must be terminated by a semicolon.	REPEAT Statement
CONTINUE statement	CONTINUE Statement
RETURN statement	RETURN Statement

Rule	Syntax Diagram
EXIT statement	<p>EXIT Statement</p>  <pre> graph LR Start(()) --> EXIT(EXIT) EXIT --> End(()) </pre>
Program jump	<p>GOTO Statement</p>  <pre> graph LR Start(()) --> GOTO(GOTO) GOTO --> IDENTIFIER[IDENTIFIER Label] IDENTIFIER --> End(()) </pre>

15 Tips and Tricks

Division of two Integer Values with the Result in the REAL Format

You program the following statement in SCL:

```
Fraction:=Dividend/Divisor;
```

Where `Fraction` is a real value, while `Dividend` and `Divisor` are integer values.

Remember that when the SCL compiler detects such operations, it makes an implicit data type conversion and compiles the statement above as follows:

```
Fraction:=INT_TO_REAL(Dividend/Divisor);
```

This means that the division always returns a rounded value as the result, for example, $1/3 = 0$ or $3/2 = 1$.

Runtime Optimized Code when Accessing Structures in Data Blocks

If you need to access a structure more than once in a data block, the following method can be recommended:

1. Create a local variable with the type of the structure.
2. Assign the structure from the data block to the variable once.
3. You can then use the variable more than once in the code without having to access the DP again.

Example

```
DB100.array[i].value :=  
DB100.array[i].value1 * DB100.array[i].value2 /  
DB100.array[i].value3 ;
```

This example requires less memory and has a shorter runtime if you program it as follows:

```
VAR_TEMP  
  tmp : STRUCT  
    value : REAL;  
    value1 : REAL;  
    value2 : REAL;  
    value3 : REAL;  
  END_STRUCT;  
END_VAR  
tmp := DB100.array[i];  
DB100.array[i].value := tmp.value1 * tmp.value2 / tmp.value3;
```

Note

With VAR_TEMP, you store variables in the stack of the CPU. With smaller CPUs, this can lead to a stack overflow. You should therefore use temporary variables sparingly!

Problems Allocating the L Stack with Small CPUs

Problems allocating the L stack are due to the small stack size of the smaller CPUs. In most cases, the problem can be avoided by taking the measures outlined below:

- Use temporary variables sparingly (VAR_TEMP or VAR section).
- Do not declare any variables of a higher data type and reduce the number of variables of an elementary data type to a minimum.
- Use static variables:
 - When you program an FB, you can use the VAR section instead of VAR_TEMP.
 - When you program an OB or FC, make use of a shared data block or bit memory.
- Avoid complicated expressions. When it processes complicated expressions, the compiler stores interim results on the stack. Depending on the type and number of interim results, the available stack size might be exceeded.
Remedy:
Break down your expression into several smaller expressions and assign the interim results explicitly to variables.

Output of REAL Numbers during Monitoring

The "Monitoring" test function can produce the following patterns when displaying nonprintable REAL numbers:

Value	Symbol
+ infinity	1.#INFrandom-digits
- infinity	-1.#INFrandom-digits
Indefinite	digit.#INDrandom-digits
NaN digit.	#NANrandom-digits

Displaying SCL Programs in STL Representation

You can open an SCL block with the STL/LAD/FBD editor and display the compiled MC7 commands. Do not make any changes in STL for the following reasons:

- The displayed MC7 command do not necessarily represent a valid STL block.
- An error-free compilation with the STL compiler normally requires modifications to be made that require thorough knowledge of both STL and SCL.
- The block compiled with STL then has the STL language identifier and no longer the SCL identifier.
- The SCL source file and the MC7 code are no longer consistent.

Handling the Time Stamp of Interface and Code

If you create new blocks (FBs, FCs and OBs), the interface (block parameters) and code have the time stamp of time at which they are compiled.

If a block already exists, the code always has the time stamp of the time at which it is compiled. The interface may retain its old time stamp. The time stamp of an interface is changed only when the structure of the interface is modified; in other words,

- the time stamp is retained if modifications are made in the code section, in the attributes, in the comment, in the section VAR_TEMP (with FCs also VAR) or in the notation of the names of parameters or variables. This also applies to underlying interfaces.
- The time stamp of an interface is updated when the data type or any initialization of a parameter or variable is modified or when parameters are removed or added and if the name of the FB changes when multiple instances are involved. This also applies to underlying interfaces.

Return Value of STEP 7 Standard and System Functions

Many STEP 7 standard and system functions have a function value of the type INT that contains the error code. In the reference manual for these functions, the possible error codes are specified as WORD constants of the type "W#16#8093".

S7 SCL is a language that is strict in its rules regarding mixing of types, so that INT and WORD cannot be mixed. The following query, for example, does not produce the required result.

```
IF SFCxx(..) = 16#8093 THEN ...
```

You can, however, tell the S7 SCL compiler that a WORD constant should be considered as INT as follows.

- By type-defining the constant. In this case, the query above appears as follows:

```
IF SFCxx(..) = INT#16#8093 THEN ...
```
- By converting WORD_TO_INT(). You would then formulate the query above as follows:

```
IF SFCxx(..) = WORD_TO_INT(16#8093) THEN ...
```

Rewiring Blocks

You can no longer rewire the block calls in the SCL blocks using the **Options > Rewire** SIMATIC Manager function. You must edit the calls in the SCL source file of the blocks affected manually.

Recommendation:

- Define symbolic names for the blocks in the symbol table and call the blocks using their symbolic names.
- Define symbolic names for absolute addresses (I, M, Q etc.) in the symbol table and use the symbolic names in your program.

If you want to rewire a block later, you only need to change the assignment in the symbol table and do not need to make changes in the SCL source file.

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 "I3.6".

Address

An address is the part of a statement that specifies the data on which an operation is to be performed. It can be addressed in both absolute and symbolic terms.

Address Identifier

An address identifier is that part of an address of an operation that 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).

Addressing

Assignment of a memory location in the user program. Memory locations can be assigned specific addresses or address areas (examples: input I 12.1, memory word MW25)

Addressing, Absolute

With absolute addressing, you specify the memory location of the address to be processed. Example: The address Q4.0 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 or using a symbol file.

Array

An array is a complex data type consisting of a number of data elements of the same type. These data elements in turn can be elementary or complex.

Assignment

Mechanism for assigning a value to a variable.

Attribute

An attribute is a characteristic that can be attached, for example, to a block identifier or variable name. In SCL there are attributes for the following items of information: block title, release version, block protection, author, block name, block family.

B

BCD

Binary-coded decimal. In STEP 7, internal coding of timers and counters in the CPU is in BCD format only.

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 to store interim results.

Block

Blocks are subunits of a user program and are distinguished by 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 starts 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;

Block Comment

You can enter additional information about a block (for example, to describe the automated process). These comments are not loaded into the work memory of SIMATIC S7 programmable controllers.

Block Protection

Using block protection, you can protect individual blocks from being decompiled. You enable this protection by assigning the keyword "KNOW_HOW_PROTECTED" when the block source file is compiled.

Block Type

The block architecture of STEP 7 includes 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.

Breakpoint

This function can be used to switch the CPU to HOLD at specific points in the program. When the program reaches a breakpoint, debugging functions such as single-step instruction processing or controlling/monitoring variables are possible.

C**Call Hierarchy**

Blocks must be called before they can be processed. The order and nesting sequence of these block calls is known as the 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, these 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.

Comments

Language construction with which you can include explanatory text in a program and that has no influence on the running of the program.

Compilation

The process of generating an executable user program from a source file.

Compilation, Source-Oriented

In source-oriented input, the source is compiled into an executable user program only when all the instructions have been entered. The compiler checks for input errors.

Constant

Placeholders for constant values in logic blocks. Constants are used for improving the legibility of a program. Example: Instead of specifying a value (for example, 10), the placeholder "Max_loop_iterations" is specified. When the block is called, the value of the constant (for example, 10) replaces the placeholder.

Constant, (symbolic)

Constants with symbolic names are placeholders for constant values in logic blocks. Symbolic constants are used for improving the legibility of a program.

CONTINUE Statement

A CONTINUE statement is used in SCL to terminate the execution of the current iteration of a loop statement (FOR, WHILE or REPEAT).

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, count up) and to execute it (for example, start).

D

Data Block (DB)

Data blocks are blocks containing the data and parameters with which the user program operates. In contrast to all other types of blocks, they contain no instructions.

Data, Static

Static data are local data of a function block that are stored in the instance data block and are therefore retained until the next time the function block is processed.

Data, Temporary

Temporary data are the local data for a block that are entered in the local stack (L stack) while the block is executing. Once a block has executed, this data is no longer available.

Data Type

Data types determine the following:

- The type and interpretation of data elements
- The permitted memory and value ranges of data elements
- The set of operations that can be performed on an address of a data type
- The notation of data elements

Data Type, Complex

Complex data types made up of data elements of elementary data types. A distinction is made between structures and arrays. The data types STRING and DATE_AND_TIME are also complex data types.

Data Type Conversion

A data type conversion is necessary when an operation is required on two variables of different 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 integer variable.

Data Type, User-defined

User-defined data types (UDTs) are data types you can create yourself using the data type declaration. Each one is assigned a unique name and can be used any number of times. A user-defined data type is useful for generating a number of data blocks with the same structure (for example, controller).

Declaration

A mechanism for defining a language element. A declaration involves the linking of an identifier with the language element and the assignment of attributes and data types.

Declaration Section

The variable declaration of a block is divided into various declaration sections for declaring the various block parameters. The declaration section IN contains, for example, the declaration of the input parameters, the declaration section OUT contains the declaration of the output parameters.

Declaration Section

You declare the local data of a logic block in the declaration section if you write your program with a text editor.

Download

The transfer of loadable objects (for example, logic blocks) from a programming device to the load memory of a CPU.

E

Enable (EN)

In STEP 7, each function block and each function has the implicitly defined input parameter "Enable" (EN) that can be set when the block is called. If EN is TRUE, the called block is executed. Otherwise it is not executed.

Enable Output (ENO)

In STEP 7 every block has an "Enable Output" (ENO). When the execution of a block is completed the current value of the OK flag 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.

EXIT Statement

Language construction within a program used to exit a loop at any point regardless of conditions.

Expression

In SCL, an expression is a means of processing data. A distinction is made between arithmetic, logical and comparison expressions.

F

FOR Statement

Language construction within a program. A FOR statement is used to execute a sequence of statements in a loop while a control variable is continuously assigned values.

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.

Function (FC)

According to the International Electrotechnical Commission's IEC 1131-3 standard, functions are logic blocks that do not have static data. 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 with static data (Data, Static). Since an FB has a "memory" (instance data block), it is possible to access its parameters (for example, outputs) at any time and at any point in the user program.

G

GOTO Statement

Language construction within a program. A GOTO statement causes the program to jump immediately to a specified label and therefore to a different statement within the same block.

H

HOLD

The CPU changes to the HOLD state from the RUN mode following a request from the programming device. Special test functions are possible in this mode.

I

Identifier

Combination of letters, numbers and underscores that identify a language element.

Initial Value

Value assigned to a variable when the system starts up.

In/Out Parameter

In/out parameters exist 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 Parameters

Input parameters exist only in functions and function blocks. 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.

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.

Integer (INT)

Integer (INT) is one of the elementary data types. Its values are all 16-bit whole numbers.

K

Keyword

A reserved word that characterizes a language element, for example, "IF".

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 flexible formats; in other words, the addition of spaces and control characters is not permitted.

Literal

Formal notation that determines the value and type of a constant.

Local Data

Local data is data assigned to a specific logic block that is declared in its declaration section or in its variable declaration. Depending on the particular block, it consists of the formal parameters, static data and temporary data.

Logic Block

A logic block in SIMATIC 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.

Mnemonics

Mnemonics are the abbreviated representation of the addresses and programming operations in the program. STEP 7 supports English representation (in which, for example, "I" stands for input) and German representation (where, for example, "E" stands for input (Eingang in German)).

Monitoring

By monitoring a program, you can check how the program is executed on the CPU. During monitoring, for example, names and actual values of variables and parameters are displayed in chronological order and updated cyclically.

Multiple Instance

When multiple instances are used, the instance data block holds the data for a series of function blocks within a call hierarchy.

N

Non Term

A non term is a complex element in a syntactical description that is described by another lexical or syntax rule.

O

Offline

Offline is the operating mode in which the programming device is not connected (physically or logically) to the PLC.

OK Flag

The OK flag is used to indicate the correct or incorrect execution of a sequence of commands in a block. It is a shared variable of the type BOOL.

Online

Online is the operating mode in which the programming device is connected (physically or logically) with the PLC.

Online Help

When working with STEP 7 programming software, you can display context-sensitive help on the screen.

Operation

An operation is the part of a statement specifying what action the processor is to perform.

Organization Block (OB)

Organization blocks form the interface between the S7 CPU operating system and the user program. The organization blocks specify the sequence in which the blocks of the user program are executed.

Output Parameter

The output parameters of a block in the user program are used to pass results to the calling block.

P**Parameter Type**

A 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 on 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 make it easier to implement 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 SCL programming language allows you to use symbolic character strings instead of addresses: For example, the address Q1.1 can be replaced by "valve_17". The symbol table creates the link between the address and its assigned symbolic character string.

Project

A folder for storing all objects relating to a particular automation solution regardless of the number of stations, modules or how they are networked.

R

Real Number

A real number is a positive or negative number representing a decimal value, for example 0.339 or -11.1.

REPEAT Statement

Language construction within a program used to repeat a sequence of statements until a termination condition is reached.

RETURN Statement

Language construction within a program with which you can exit the current block.

Return Value (RET_VAL)

In contrast to function blocks, functions produce a result known as the return value.

RUN

In the RUN mode the user program is processed and the process image is updated cyclically. All digital outputs are enabled.

RUN-P

The RUN-P operating mode 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

A folder for blocks that are downloaded to a programmable S7 module (for example CPU or FM) and are capable of being run on the module as part of the program controlling a system or a process.

Scan Cycle Monitoring Time

If the time taken to execute the user program exceeds the set scan cycle monitoring time, the operating system generates an error message and the CPU switches to STOP mode.

Scan Cycle Time

The scan cycle time is the time required by the CPU to execute the user program once.

SCL

PASCAL-based high-level language that conforms to the standard DIN EN-61131-3 (international IEC 1131-3) and is used to program 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 compiled blocks are stored in the "Blocks" folder in the S7 program.

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 designed for use with SCL with which you create 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 later translated into machine code by the SCL Compiler.

Semantics

Relationship between the symbolic elements of a programming language and their meaning, interpretation and application.

Shared Data

Shared data is data that can be accessed by 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.

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

Part of a program created with a graphics or textual editor from which an executable user program can be compiled.

Statement

A statement 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.

Status Word

The status word is a component of the CPU registers. 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)

Complex data type consisting of any data elements of different data types. The data types within structures can be elementary or more complex.

Symbol

A symbol is a name defined by the user that adheres to certain syntax rules. 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), I1.7 (Address)Controller (Symbol), SFB24 (Block)

Syntax Rule

The higher level of rules in the formal SCL language description consists of the syntactical rules. When they are used they are not subject to formatting restrictions; in other words, spaces and control characters can be added.

System Data Block (SDB)

System data blocks are data areas in the S7 CPU that contain system settings and module parameters. System data blocks are created and edited using the STEP 7 standard software.

System Function (SFC)

A system function (SFC) is a function integrated in the CPU operating system that can be called in the STEP 7 user program when required.

System Function Block (SFB)

A system function block (SFB) is a function block integrated in the CPU operating system that can be called in the STEP 7 user program when required.

System Memory (System Area)

The system memory is integrated in the S7 CPU and is implemented as 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 might be a keyword or even a single character.

Timers

Timers are components of the system memory of the CPU. The contents of these timers are 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 its execution (Start).

U

UDT

See: Data Type, User-defined

User Data

User data are exchanged between a CPU and a signal module, function module and communications modules via the process image or by 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 assigned 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 that can be used in the STEP 7 user program. A variable consists of an address (for example, M3.1) 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

The variable declaration includes the specification of a symbolic name, a data type and, if required, an initialization value and a comment.

Variable Table

The variable table is used to collect together the variables including their format information that you want to monitor and modify.

View

To be able to access a declared variable with a different data type, you can define views of the variable or of areas within the variables. A view can be used like any other variable in the block. It inherits all the properties of the variable that it references; only the data type is new.

Index

-	10-8	Bit Data Types	6-3
*	10-9	Bit Memory	9-2
**	10-8	Bit String Standard Functions	13-11
/	10-8	Block Attributes	5-5, 5-8
+	10-8	Definition	5-5
<	10-12	Lexical Rules	14-28
<=	10-12	System Attributes for Blocks	5-8
<>	10-12	Block Call	3-13
=	10-12	BLOCK Data Types	6-16
>	10-12	Block End	5-3
>=	10-12	Block Identifier	4-6
ABS	13-9	Block Name	5-3
Absolute Access to Data Blocks	9-8	Block Parameters	4-15, 7-13
Absolute Access to Memory Areas		Block Protection	3-7
of the CPU	9-3	Block Start	5-3
Absolute Addressing		Block Structure	5-3
Lexical Rules	14-25	Block Templates	3-14
ACOS	13-10	Block Types	1-3
Actual Parameters	7-1	BLOCK_DB_TO_WORD	13-4
Definition	7-1	Blocks	2-4, 3-6, 5-1
Input Assignment	11-39	Boolean Expression	10-12
Addition	10-2	Boxes in Syntax Diagrams	4-1
Address Identifier	4-7	Breakpoints	3-27–3-29
Addresses	9-2, 10-3	BYTE	6-3
And	10-2	BYTE_TO_BOOL	13-4
AND	10-10	BYTE_TO_CHAR	13-4
ANY	6-18	Calling Blocks	3-13
ANY Data Type	6-18	Calling Counter Functions	12-1
Arithmetic Expressions	10-8	Calling Function Blocks (FB or SFB)	11-28
ARRAY	6-9 , 7-4, 11-5	Call as Local Instance	11-28
Initialization	7-4	Call as Shared Instance	11-28
Value Assignment with Variables		In/Out Assignment	11-32
of the Data Type ARRAY	11-5	Input Assignment	11-31
ARRAY Data Type	6-9	Procedure	11-28
Arrays	6-9	Reading Output Values	11-33
ASIN	13-10	Supplying FB Parameters	11-30
AT	7-6	Syntax	11-28
ATAN	13-10	Calling Functions (FC)	11-36
Attributes	5-6, 5-8, 5-10	Input Assignment	11-39
Authorization	1-9	Input Parameter EN	11-42
Authorization Diskette	1-9	Output or In/Out Assignment	11-40
AUTHORS.EXE	1-9	Output Parameter ENO	11-43
Basic Terms in SCL	4-3–4-12	Parameter Supply	11-38
BIT	6-3	Procedure	11-36
Bit Constants	8-6	Return Value	11-37

Syntax	11-36	Count Down (S_CD)	12-5
Calling Timer Functions	12-8	Count Up (S_CU)	12-5
CASE Statement	11-12, 11-16	Count Up/Down (S_CUD)	12-6
CHAR	6-3	COUNTER	6-15, 12-1
Char Constants	8-9	COUNTER Data Type	6-15
CHAR_TO_BYTE	13-4	Counters	12-1–12-7
CHAR_TO_INT	13-4	Calling Counter Functions	12-1
Character Set	4-3	Count Down (S_CD)	12-5
Character Strings	4-12	Count Up (S_CU)	12-5
Character Types	6-3	Count Up/Down (S_CUD)	12-6
Closing an SCL Source File	3-6	Example of Counter Functions	12-7
Code blocks	2-4, 3-8, 5-1	Input and Evaluation of the	
Color and Font Style of the		Counter Value	12-4
Source Text	3-12 , 3-20	Parameter Supply for	
Comment		Counter Functions	12-3
Comment Section	4-13	CPU Communication	3-34
Inserting Templates for Comments	3-14	CPU Memory Areas	4-7
Lexical Rules	14-27	CPU Memory Reset	3-21
Comment Section	4-13	Creating a Compilation Control File	3-17
Comments		Creating a New SCL Source File	3-4
Line Comment	4-14	Creating Source Files with a	
Comparison Expressions	10-12	Standard Editor	3-7
Compilation Control File	3-17	Customizing	3-3, 3-15
Compiler	3-15	Customizing the Page Format	3-19
Customizing the Compiler	3-15	Cutting Text Objects	3-11
Development Environment	1-1, 1-4	Cycle Time	3-33
Compiling	3-15–3-18	Data Blocks	5-18, 9-7–9-11
Complex Data Types	6-1, 6-5, 6-7	Data Type Conversion Functions	13-4, 13-6
Compliance with Standard	1-1	Data Type POINTER	6-16
CONCAT	13-13	Data Type STRUCT	6-11
Conditions	11-13	Data Type UDT	6-13
Constants	8-2, 8-6–8-16	Data Types	6-1–6-13
CONTINUE Statement	11-12, 11-23	Complex	6-2
Continuous Monitoring	3-24	Description	6-1
Control File for Compilation	3-17	Elementary	6-1, 6-2
Control Statements	3-14, 5-12, 11-14	Data Types for Parameters	6-15
CASE Statement	11-16	DATE	6-4
CONTINUE Statement	11-23	Date Constant	8-12
EXIT Statement	11-24	DATE_AND_TIME	6-5
FOR Statement	11-18	DATE_AND_TIME Data Type	6-6
GOTO Statement	11-25	DATE_TO_DINT	13-4
IF Statement	11-14	Debugger	1-6
Inserting Control Statements	3-14	Development Environment	1-4
REPEAT Statement	11-22	Debugging Functions in STEP 7	3-29
Statements	5-12	Debugging Functions of STEP 7	3-30
Syntax Rules	14-43	Debugging the Program After Compilation	3-18
WHILE Statement	11-21	Debugging with Breakpoints	3-25
Conversion Functions	13-4	Declaration	5-9
Class B	13-4	Declaration of Static Variables	7-3
Copying Text Objects	3-10	Declaration Section	5-9, 7-4, 7-10–7-14
Correct Syntax Formatting		Block Parameters	7-14
the Source Text	3-12	Definition	5-9
COS	13-10	Initialization	7-4

Overview of the Declaration	
Subsections	7-10
Static Variables	7-11
Structure	5-9
Syntax Rules	14-31
Temporary Variables	7-12
Decompile STL	
SCL Block	1-3
DELETE	13-15
Deleting Text Objects	3-11
Designing SCL Programs	2-1
Development Environment	1-2
Batch Compiler	1-1
Debugger	1-1
Editor	1-1
DI_STRNG	13-19
Diagnostic Buffer	3-32
DIN Standard EN-61131-3	1-1
DINT	6-3
DINT_TO_DATE	13-4
DINT_TO_DWORD	13-4
DINT_TO_INT	13-4
DINT_TO_TIME	13-4
DINT_TO_TOD	13-4
Displaying and Modifying the	
CPU Operating Mode	3-31
Displaying and Setting the	
Date and Time on the CPU	3-31
Displaying Information about	
Communication with the CPU	3-34
Displaying the Blocks on the CPU	3-33
Displaying the Cycle Time of the CPU	3-33
Displaying the Stacks of the CPU	3-34
Displaying the Time System of the CPU	3-33
Displaying/Compressing the	
User Memory of the CPU	3-32
DIV	10-8
Division	10-2
Double Word	6-3
Downloading	3-21
Downloading User Programs	3-21
DWORD	6-3
DWORD_TO_BOOL	13-4
DWORD_TO_BYTE	13-4
DWORD_TO_DINT	13-4
DWORD_TO_REAL 1)	13-4
DWORD_TO_WORD	13-4
Editing an SCL Source File	3-9–3-14
Editor	
Development Environment	1-4
Elementary Data Types	6-1–6-4
Emergency Authorization	1-10
EN	11-42
ENO	11-43
EQ_STRNG	13-17
Equality	10-2
Examples	6-19, 11-33–11-35, 11-41, 12-7, 13-7, 13-10, 13-12
Exclusive Or	10-2
EXIT Statement	11-12, 11-24
EXP	13-9
EXPD	13-9
Expressions	10-2, 10-6–10-12
Extended Variable	10-4
FB Parameters	11-30–11-33
FC	5-15 , 11-27, 11-36
FC Parameters	11-38–11-40
FIND	13-16
Finding Text Objects	3-10
Flags (OK Flag)	7-9
Flexible Format	4-2
Flow Chart for ACQUIRE	2-18
Font Style and Color	3-12, 3-20
FOR Statement	11-12, 11-18
Formal Language Description	14-1
Formal Parameters	7-1
Function (FC)	5-15 , 11-27, 11-36
Function Block (FB)	5-13 , 11-27–11-30
Functions for Rounding and Truncating	13-6
GE_STRNG	13-17
Generating and Displaying	
Reference Data	3-29
Go To	3-11
GOTO Statement	11-25
GT_STRNG	13-18
I_STRNG	13-18
Identifiers	4-5, 14-16–14-18
Definition	4-5
Examples	4-5
Formal Language Description	14-14, 14-16
Lexical Rules	14-19
Rules	4-5
IF Statement	11-12
IF Statements	11-14
In/Out Assignment	11-32, 11-40
In/Out Assignment (FB/SFB)	11-32
In/Out Assignment (FC)	11-40
In/Out Parameters	7-1 , 11-32
Indent Automatically	3-12
Indexed Access to Data Blocks	9-10
Indexed Access to Memory Areas	
of the CPU	9-6
Inequality	10-2
Initial Values	7-4
Initialization	7-4
Input Parameters	7-1, 11-31, 11-39, 11-42

Definition.....	7-1	Negation	10-2
Input Assignment (FB)	11-31	NIL Pointer	6-18
Input Assignment (FC).....	11-39	Non Terms (in syntax diagrams).....	14-13
Input Parameter EN.....	11-42	Nonprintable Characters	8-9
INSERT	13-15	NOT	10-10
Inserting Block Calls.....	3-13	Numbers	4-10
Inserting Block Templates	3-14	Numeric Data Types	6-3
Inserting Control Structures	3-14	Numeric Standard Functions	13-9
Inserting Parameter Templates.....	3-14	OB	5-17
Inserting Templates for Comments	3-14	OK Flag.....	7-1, 7-9
Installation	1-9	Opening an SCL Source File.....	3-5
Instance Declaration	7-8	Opening Blocks	3-6
INT	6-3	Operating Mode.....	3-31
INT_TO_CHAR.....	13-4	Operations	14-8
INT_TO_WORD.....	13-4	Alphabetic List	14-6
Integer Constant	8-7	Or	10-2
Integer Division	10-2	OR	10-10
Jump Statements.....	11-12	Order of the Blocks.....	3-8
Keywords.....	4-4, 14-9	Organization Block	5-17
Labels	8-17	Output Parameter ENO.....	11-43
Language Description	4-1, 14-1	Output Parameters	7-1
LE_STRNG	13-17	Page Break	3-20
LEFT	13-14, 13-16	Page Format	3-19
LEN.....	13-13	Parameter	7-13
Lexical Rules	14-19	Parameter Supply.....	11-27
Line Comment	4-14	Parameter Supply for Counter Functions	12-3
Line Indent.....	3-12	Parameter Templates	3-14
Line Numbers	3-3, 3-20	Parameters	5-10, 6-15, 7-1, 7-10, 11-30, 11-38–11-40
Literals	8-6–8-16	Parenthesis	10-2
LN	13-9	Peripheral Inputs/Outputs	9-2
Local Data	4-15, 7-1–7-4, 7-11	POINTER.....	6-16
Local Instance.....	11-28, 11-29, 11-35	Positioning the Cursor in a Specific Line.....	3-11
LOG	13-9	Power.....	10-2
Logarithmic Functions	13-9	Predefined Constants and Flags	
Logic Blocks	2-4, 3-8, 5-1	Formal Language Description.....	14-18
Loops	11-12	Printable Characters	8-9
LT_STRNG.....	13-18	Printing an SCL Source File	3-19
Math Standard Functions	13-9	Process Image of the Inputs and Outputs	9-2
Memory Areas of the CPU.....	9-1–9-6	Program Branch	11-12
Memory Location of Variables	7-6	Program Design	2-1
Menu Bar.....	3-2	Program Jump.....	11-12
MID	13-14	Programming Structured	1-3
MOD.....	10-9	Programming Language	
Modulo Function	10-2	Higher	1-1
Monitoring.....	3-24, 3-26	Higher-Level	1-3
Multiple Instances	7-8	Programming with Symbols.....	3-9
Multiplication.....	10-2	R_STRNG	13-19
Names.....	4-5	Reading Out CPU Data.....	3-32
Definition.....	4-5	Reading Out the Diagnostic Buffer	
Examples.....	4-5	of the CPU.....	3-32
Formal Language Description	14-14, 14-16	Reading Output Values	11-33
Rules	4-5	Output Assignment at an FC Call.....	11-40
NE_STRNG.....	13-17		

- Output Assignment in an FB Call 11-33
- REAL 6-3
- Real Number Constant 8-8
- REAL_TO_DINT 13-4
- REAL_TO_DWORD 2) 13-4
- REAL_TO_INT 13-4
- Redoing an Editing Action 3-9
- Reference Data 3-29
- REPEAT Statement 11-12, **11-22**
- REPLACE 13-16
- Replacing Text Objects 3-10
- Reserved Words 4-4
- RETURN Statement 11-12, **11-26**
- Return Value 11-37
- Return Value (FC) 11-37
- RIGHT 13-14
- ROL 13-11
- ROR 13-11
- ROUND 13-6
- Rule Structures 4-1
- Rules for SCL Sources 3-8
- S_CD 12-5
- S_CU 12-5
- S_CUD 12-6
- S_ODT 12-16
- S_ODTS 12-17
- S_OFFDT 12-18
- S_PEXT 12-15
- S_PULSE 12-14
- S5 Time 12-12
- S5TIME 6-4
- Sample Program
 - "Measured Value Acquisition" 2-1
- Sample Program for First-Time Users 2-1
- Saving an SCL Source File 3-19
- SCL Debugging Functions 3-23–3-25
- SCL User Interface 3-2
- Selecting Text Objects 3-10
- Selecting the Right Timer 12-20
- Selective Statement 11-12
- Setting the Date 3-31
- Setting the Time 3-31
- SFCs/SFBs 13-22
- Shared Data 9-1
 - Overview of Shared Data 9-2
- Shared Instance 11-28, 11-33
- SHL 13-11
- SHR 13-11
- Simple Expression 10-7
- SIN 13-10
- Single Step 3-25
- Source File 3-4–3-8, 3-19, 5-11, 5-21
- Specifying Object Properties 3-6
- SQR 13-9
- SQRT 13-9
- Stacks 3-34
- Standard Functions 13-4, 13-6, 13-9–13-11
- Standard Identifier 4-6
- Start
 - SCL 3-1
- Statement Section
 - Structure 5-11
 - Syntax Rules 14-37
- Statements 11-1, 11-14–11-18, 11-21–11-26
- Static Variables 4-15, 7-1, 7-3, 7-8, **7-11**
- Status Bar 3-2
- STEP 7 Block Concept 1-3
- STEP 7 Debugging Functions 3-29
- STL
 - Extension SCL 1-1
 - SCL Block Decompile 1-3
- STRING 6-7, 8-9, 13-15, 13-13–13-19
- STRING Data Type 6-7
- STRING_TO_CHAR 13-4
- STRNG_DI 13-19
- STRNG_I 13-18
- STRNG_R 13-19
- STRUCT 6-11
- Structure of a Data Block (DB) 5-18
- Structure of a Function (FC) 5-15
- Structure of a Function Block (FB) 5-13
- Structure of an Organization Block (OB) 5-17
- Structure of an SCL Source File 5-9–5-18
- Structure of the Declaration Section 5-9
- Structured Access to Data Blocks 9-11
- Structured Programming 2-4, 2-6
- Structures 6-11
- Subroutine Call 5-12
- Subtraction 10-2
- Supplying Parameters for
 - Timer Functions 12-10
- Symbolic Access the
 - Memory Areas of the CPU 9-5
- Symbolic Constants 8-2
- Syntax Diagrams 4-1, 14-1
- Syntax Rules 14-29
- System Attributes 5-8, 5-10
 - for Blocks 5-8
 - for Parameters 5-10
- System Functions/Function Blocks
 - and the Standard Library 13-22
- TAN 13-10
- Templates 3-14
 - for Blocks 3-14

for Comments	3-14	User Interface.....	3-2
for Control Structures	3-14	User Memory.....	3-32
for Parameters	3-14	User Program.....	2-4, 5-1
Temporary Variables	4-15, 7-1, 7-12	User-Defined	
Terminate Condition.....	11-22, 11-24	Data Types (UDT).....	5-21, 6-13 , 11-3
Terms Used in the Lexical Rules		Value Assignment.....	11-2–11-10
(Syntax Diagrams).....	14-4	Syntax Rules	14-39
Testing in Single Steps.....	3-25	Value Assignment with	
TIME	6-4	Absolute Variables for Memory Areas ..	11-9
Time Period Constant	8-13	Value Assignment with Variables	
Time System.....	3-33	of the Type DATE_AND_TIME	11-8
Time Value	12-12	Value Assignment with Variables	
TIME_OF_DAY.....	6-4	of the Type STRUCT and UDT	11-3
TIME_TO_DINT.....	13-4	Value Assignments with	
Time-of-Day Constant	8-15	Shared Variables	11-10
TIMER (data type).....	6-15	Value Assignments with Variables	
TIMER Data Type	6-15	of an Elementary Data Type	11-2
Timer Functions	12-8	Value Assignments with	
Timers	12-8–12-19	Variables of the Type STRING.....	11-7
Calling Timer Functions.....	12-8	Value Assignments.....	5-12
Examples.....	12-19	VAR	7-10
Input and Evaluation of a Time Value....	12-12	VAR_IN_OUT.....	7-10
Start Timer as Extended Pulse Timer		VAR_INPUT	7-10
(S_PEXT).....	12-15	VAR_OUTPUT	7-10
Start Timer as Off-Delay Timer		VAR_TEMP.....	7-10
(S_OFFDT)	12-18	Variables	
Start Timer as On-Delay Timer		General Syntax of a	
(S_ODT)	12-16	Variable or Parameter Declaration	7-3
Start Timer as Pulse Timer		Initialization.....	7-4
(S_PULSE)	12-14	Instance Declaration	7-8
Start Timer as Retentive		Local Variables and Block Parameters.....	7-1
On-Delay Timer (S_ODTS).....	12-17	Monitoring/Modifying Variables.....	3-30
Supplying Parameters for		Overview of the	
Timer Functions	12-10	Declaration Subsections.....	7-10
Title Bar.....	3-2	Static Variables.....	4-15, 7-1 , 7-11
TOD_TO_DINT.....	13-4	Temporary Variables.....	4-15, 7-1 , 7-10
Toolbar.....	3-2	Views of Variable Ranges	7-6
Trigonometric Functions.....	13-10	Warnings.....	3-18
TRUNC.....	13-6	What's New?	1-7
UDT.....	6-13	WHILE Statement.....	11-12, 11-21
Call.....	5-21	WORD	6-3
Definition.....	5-22	WORD_TO_BLOCK_DB.....	13-4
Elements.....	5-21	WORD_TO_BOOL	13-4
Unary Minus	10-2	WORD_TO_BYTE	13-4
Unary Plus.....	10-2	WORD_TO_INT	13-4
Undoing the Last Editing Action	3-9	Working Area.....	3-2
User Authorization	1-9	Working with an SCL Source File	3-5, 3-19
User Data	9-1	XOR.....	10-10
Shared.....	9-1		

Siemens AG
A&D AS E 81
Oestliche Rheinbrueckenstr. 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:

- | | |
|--|---|
| <input type="checkbox"/> Automotive | <input type="checkbox"/> Pharmaceutical |
| <input type="checkbox"/> Chemical | <input type="checkbox"/> Plastic |
| <input type="checkbox"/> Electrical Machinery | <input type="checkbox"/> Pulp and Paper |
| <input type="checkbox"/> Food | <input type="checkbox"/> Textiles |
| <input type="checkbox"/> Instrument and Control | <input type="checkbox"/> Transportation |
| <input type="checkbox"/> Nonelectrical Machinery | <input type="checkbox"/> Other |
| <input type="checkbox"/> 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:

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....