SIEMENS

SIMOTION

SIMOTION ST Structured Text

Programming and Operating Manual

08/2008

Safety Guidelines

This manual contains notices you have to observe in order to ensure your personal safety, as well as to prevent damage to property. The notices referring to your personal safety are highlighted in the manual by a safety alert symbol, notices referring only to property damage have no safety alert symbol. These notices shown below are graded according to the degree of danger.

⚠DANGER

indicates that death or severe personal injury **will** result if proper precautions are not taken.

⚠WARNING

indicates that death or severe personal injury **may** result if proper precautions are not taken.

⚠CAUTION

with a safety alert symbol, indicates that minor personal injury can result if proper precautions are not taken.

CAUTION

without a safety alert symbol, indicates that property damage can result if proper precautions are not taken.

NOTICE

indicates that an unintended result or situation can occur if the corresponding information is not taken into account.

If more than one degree of danger is present, the warning notice representing the highest degree of danger will be used. A notice warning of injury to persons with a safety alert symbol may also include a warning relating to property damage.

Qualified Personnel

The device/system may only be set up and used in conjunction with this documentation. Commissioning and operation of a device/system may only be performed by **qualified personnel**. Within the context of the safety notes in this documentation qualified persons are defined as persons who are authorized to commission, ground and label devices, systems and circuits in accordance with established safety practices and standards.

Prescribed Usage

Note the following:

⚠WARNING

This device may only be used for the applications described in the catalog or the technical description and only in connection with devices or components from other manufacturers which have been approved or recommended by Siemens. Correct, reliable operation of the product requires proper transport, storage, positioning and assembly as well as careful operation and maintenance.

Trademarks

All names identified by ® are registered trademarks of the Siemens AG. The remaining trademarks in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owner.

Disclaimer of Liability

We have reviewed the contents of this publication to ensure consistency with the hardware and software described. Since variance cannot be precluded entirely, we cannot guarantee full consistency. However, the information in this publication is reviewed regularly and any necessary corrections are included in subsequent editions.

Siemens AG
Industry Sector
Postfach 48 48
90327 NÜRNBERG
GERMANY

Preface

Scope

This document is part of the SIMOTION Programming documentation package.

This document is valid for product version V4.1 Service Pack 2 of SIMOTION SCOUT (the engineering system of the SIMOTION product family) in conjunction with:

- a SIMOTION device with the following versions of the SIMOTION kernel:
  - V4.1 SP2
  - V4.1 SP1
  - V4.0
  - V3.2
  - V3.1
  - V3.0
- The relevant version of the following SIMOTION Technology Packages, depending on the kernel
  - Cam
  - Path (kernel V4.1 and higher)
  - Cam_ext (kernel V3.2 and higher)
  - TControl
  - Gear, Position and Basic MC (only for kernel V3.0).

This document describes the syntax and implementation of the SIMOTION ST Structured Text programming language for this version of SIMOTION SCOUT. It also includes information on the following topics:

- ST Editor and Compiler with program example
- Data storage and data management on SIMOTION devices
- Options for diagnosis and troubleshooting

The scope of the SIMOTION ST programming language may contain new syntax elements compared to earlier versions. These have only been tested using the current version of the SIMOTION kernel and are released only for this kernel version or higher versions.

Conversion of existing projects to the current SIMOTION SCOUT version

It is possible to upgrade existing projects to the current version of SIMOTION SCOUT and the SIMOTION ST programming language. In some cases, recompilation using the current version of the compiler can change the version identifiers in the data storage areas of the programs, thus resulting in deletion and initialization of all retentive and non-retentive data on the SIMOTION device. In exceptional cases, minor changes to the program source files may also be required.

If new syntax elements of the SIMOTION ST programming language are used on a SIMOTION device with an older version of the SIMOTION kernel, the compiler issues a warning (version V3.2.1 and higher of the SIMOTION kernel). If these syntax elements are used anyway, the project can be stored in the old project format, but can no longer be converted using the compiler of an older version of SIMOTION SCOUT.

Information in this manual

The following is a list of chapters included in this manual along with a description of the information presented in each chapter.

- Introduction (Chapter 1)
- Getting Started with ST (Chapter 2)

  Requirements for creating programs and a sample program
- ST Basics (Chapter 3)

  Elements of the ST programming language, variable and data type declarations, statements
- Functions, Function Blocks and Programs (Chapter 4)

  Programming and call of the program organization units (POU)
- Integration of ST in SIMOTION SCOUT (Chapter 5)

  Behavior of variables, access to inputs and outputs, libraries, preprocessor
- Error Sources and Program Test (Chapter 6)

  Information on error sources, efficient programming, and program testing
- Appendices
  - Formal Language Description (Appendix A.1)
  - Compiler Error Messages and Remedies (Appendix A.2)
  - Template for Example Unit (Appendix A.3)
- Index

If you want to get started immediately, begin by working through Chapter 2.

SIMOTION Documentation

An overview of the SIMOTION documentation can be found in a separate list of references.

This documentation is included as electronic documentation with the supplied SIMOTION SCOUT.

The SIMOTION documentation consists of 9 documentation packages containing approximately 80 SIMOTION documents and documents on related systems (e.g. SINAMICS).

The following documentation packages are available for SIMOTION V4.1 SP2:

- SIMOTION Engineering System
- SIMOTION System and Function Descriptions
- SIMOTION Diagnostics
- SIMOTION Programming
- SIMOTION Programming - References
- SIMOTION C
- SIMOTION P350
- SIMOTION D4xx
- SIMOTION Supplementary Documentation

Hotline and Internet addresses

Technical support

If you have any technical questions, please contact our hotline:

	Europe / Africa
Phone	+49 180 5050 222 (subject to charge)
Fax	+49 180 5050 223
Internet	http://www.siemens.com/automation/support-request

	Americas
Phone	+1 423 262 2522
Fax	+1 423 262 2200
E-mail	mailto:techsupport.sea@siemens.com

	Asia / Pacific
Phone	+86 1064 719 990
Fax	+86 1064 747 474
E-mail	mailto:adsupport.asia@siemens.com

Note

Country-specific telephone numbers for technical support are provided under the following Internet address:

http://www.siemens.com/automation/service&support

Calls are subject to charge, e.g. 0.14 €/min. on the German landline network. Tariffs of other phone companies may differ.

Questions about this documentation

If you have any questions (suggestions, corrections) regarding this documentation, please fax or e-mail us at:

Fax	+49 9131- 98 63315
E-mail	mailto:docu.motioncontrol@siemens.com

Siemens Internet address

The latest information about SIMOTION products, product support, and FAQs can be found on the Internet at:

- General information:
  - http://www.siemens.de/simotion (German)
  - http://www.siemens.com/simotion (international)
- Product support:
  - http://support.automation.siemens.com/WW/view/en/10805436

Additional support

We also offer introductory courses to help you familiarize yourself with SIMOTION.

Please contact your regional training center or our main training center at D-90027 Nuremberg, phone +49 (911) 895 3202.

Information about training courses on offer can be found at:

www.sitrain.com

Contents

Introduction 1

In addition to conventional open and closed-loop control tasks, today's automation systems are increasingly required to handle data management functions and complex mathematical calculations. ST (Structured Text) is specially designed for these tasks. Standardized to IEC 61131-3 (German standard DIN EN-61131-3), this programming language makes your job as a programmer easier.

1.1 High-level programming language

ST is a high-level, PASCAL-based programming language. This language is based on the IEC 61131-3 standard, which standardizes programming languages for programmable controllers (PLC). ST is based on the *Structured Text* part of this standard.

Using a high-level language like ST to program control systems offers the user a wide range of possibilities, for example:

- Data management
- Process optimization
- Mathematical/statistical calculations

1.2 Programming language with technology commands

In addition to IEC 61131-3 compliance, the SIMOTION ST programming language also contains commands for SIMOTION devices, motion control and technology.

Technology objects represent a technological functionality, e.g. positioning an axis or assigning parameters for an output cam. Technology commands are language commands provided by the technology objects. Such commands may be used, for example, to activate camming or to control motion sequences, for example, in order to position an axis.

1.3 Execution levels

The SIMOTION execution system provides different execution levels (cyclic, synchronous, time-controlled, alarm-controlled and sequential) for optimal support of the various tasks involved in creating user programs.

SIMOTION SCOUT is the engineering system of the SIMOTION product family. ST is the high-level language for creating user programs; in ST, you can develop user programs for the various execution levels.

The execution of user programs can be time-driven if you want them to run synchronously with the system clock or a defined time cycle. They can be interrupt-driven if they are to start and run once in response to a particular event. Alternatively, they can run sequentially or cyclically at the round robin execution level.

1.4 ST editor with tools for writing and testing programs

An easy-to-use text editor is provided for creating programs.

The ST compiler converts the edited program into executable code and indicates any syntax errors, specifying the program line and the cause of the error.

SIMOTION SCOUT provides test functions for testing ST programs. You can test and visualize your programs online.

Getting Started with ST 2

This chapter uses a simple example to describe how to write a program, compile it into executable code, run it, and test it.

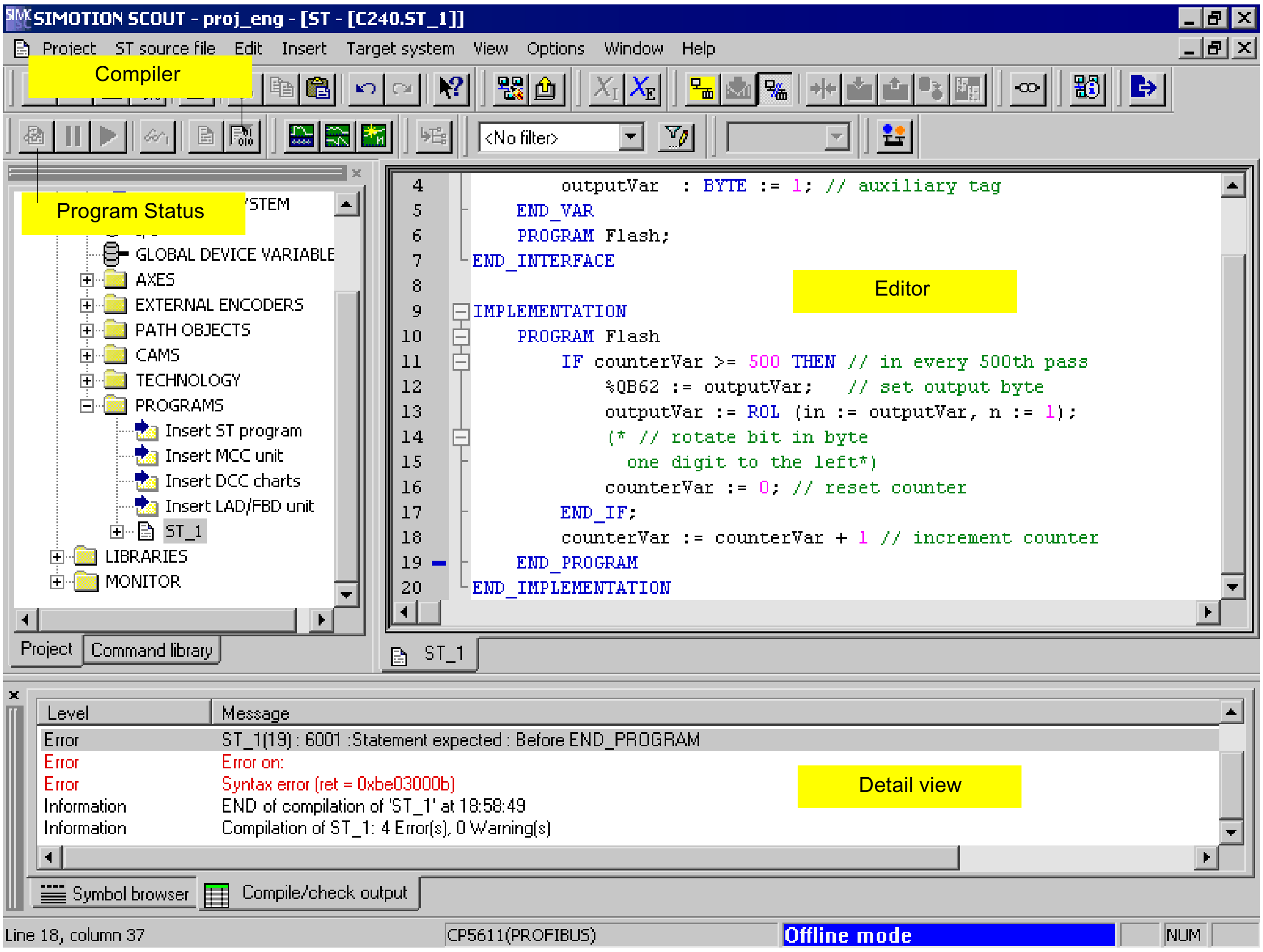
2.1 Integration of ST in SCOUT

The program environment for ST comprises the following components:

- An **editor** for creating programs, consisting of functions (FC), function blocks (FB), and user-defined data types (UDT), etc.
- A **compiler** for compiling the previously edited ST program into executable machine code
- The **program status** for assisting your search for logical program errors in the running program
- A **detail view**, in which, for example, error messages of the compiler are displayed. An important tab of the detail view is the **Symbol browser**, where you can monitor and change variables.

The individual components are easy to use. They are integrated directly in the SIMOTION SCOUT workbench.

For more information about the operation of the workbench and its tools, refer to the SIMOTION SCOUT Configuration Manual.



```
 4          outputVar  : BYTE := 1; // auxiliary tag
 5      END_VAR
 6      PROGRAM Flash;
 7  END_INTERFACE
 8
 9  IMPLEMENTATION
10      PROGRAM Flash
11          IF counterVar >= 500 THEN // in every 500th pass
12              %QB62 := outputVar;   // set output byte
13              outputVar := ROL (in := outputVar, n := 1);
14              (* // rotate bit in byte
15                one digit to the left*)
16              counterVar := 0; // reset counter
17          END_IF;
18          counterVar := counterVar + 1 // increment counter
19      END_PROGRAM
20  END_IMPLEMENTATION
```

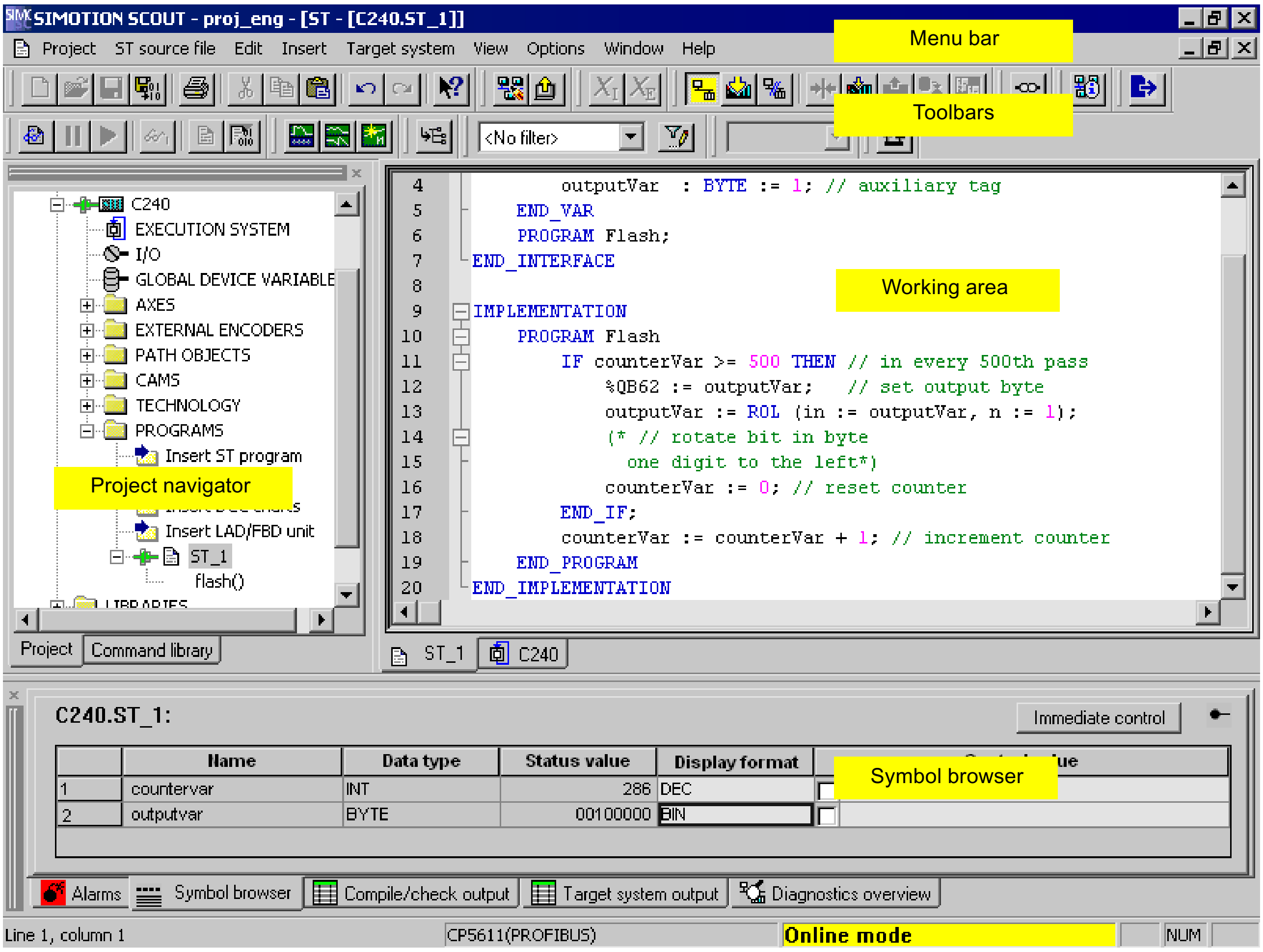
Level	Message
Error	ST_1(19) : 6001 :Statement expected : Before END_PROGRAM
Error	Error on:
Error	Syntax error (ret = 0xbe03000b)
Information	END of compilation of 'ST_1' at 18:58:49
Information	Compilation of ST_1: 4 Error(s), 0 Warning(s)

Line 18, column 37 CP5611(PROFIBUS) Offline mode NUM

Figure 2-1 Development environment of ST

2.1.1 Getting to know the elements of the workbench

The workbench represents the framework for SIMOTION SCOUT. Its tools allow you to perform all the steps necessary to configure, optimize and program a machine for your application.



```
        outputVar  : BYTE := 1; // auxiliary tag
    END_VAR
    PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
    PROGRAM Flash
        IF counterVar >= 500 THEN // in every 500th pass
            %QB62 := outputVar;   // set output byte
            outputVar := ROL (in := outputVar, n := 1);
            (* // rotate bit in byte
              one digit to the left*)
            counterVar := 0; // reset counter
        END_IF;
        counterVar := counterVar + 1; // increment counter
    END_PROGRAM
END_IMPLEMENTATION
```

C240.ST_1:

	Name	Data type	Status value	Display format	Control value
1	countervar	INT	286	DEC	
2	outputvar	BYTE	00100000	BIN	

Figure 2-2 Workbench elements

The workbench contains the following elements:

- Menus

Menus contain menu commands with which you can control the workbench and call tools, etc.

- Toolbars

You can execute many of the available menu commands by clicking the corresponding button in one of the toolbars.

- Project navigator

The project navigator displays the entire project and its elements (e.g. CPU, axes, programs, cams) in a tree structure.

- Work Area

This window allows you to perform specific tasks either independently (by programming) or using wizards (by configuring).

- Detailed view

The detail view displays additional information about the elements selected in the project navigator, e.g. all global variables for a program or the **Compile/Test Output** window.

2.2 Requirements for program creation

This section describes the general conditions you will need to meet before writing a program. You will find detailed information in the SIMOTION SCOUT Configuring Manual and the SIMOTION Motion Control function descriptions.

Add or open a project

The project is the highest level in the data management hierarchy. SIMOTION SCOUT saves all data which belongs, for example, to a production machine, in the project directory.

This means that the project therefore brackets together all SIMOTION devices, drives, etc. belonging to one machine.

Once you have created a project, you can:

- Configure hardware
- Insert and configure technology objects

Configuring hardware

Within the project, the hardware used must be made known to the system, including:

- SIMOTION device
- Centralized I/O (with I/O addresses)
- Distributed I/O (with I/O addresses)

A SIMOTION device must be configured before you can insert and edit ST source files.

Insert and configure technology objects

The functionality of axes, output cams, etc. is represented in SIMOTION by technology objects (TOs).

You cannot program technology objects using system functions and access their system variables until you have inserted and configured them.

2.3 Working with the ST editor and the compiler

In this section, you will learn how to use the ST editor and the compiler.

2.3.1 Insert ST source file

ST source files are assigned to the SIMOTION device on which they are to run.

Proceed as follows

1. Open the appropriate SIMOTION device in the project navigator.
2. Select the **PROGRAMS** folder.
3. Select the menu **Insert > Program > ST source file**.
4. Enter the name of the ST source file.

   Names for program source files must satisfy the rules for identifiers: They are made up of letters (A … Z, a … z), digits (0 … 9) or single underscores (_) in any order, whereby the first character must be a letter or underscore. No distinction is made between upper and lower case letters.

   The permissible length of the name depends on the SIMOTION Kernel version:

   – As of Version V4.1 of the SIMOTION Kernel: maximum 128 characters.

   – Up to Version V4.0 of the SIMOTION Kernel: maximum 8 characters.

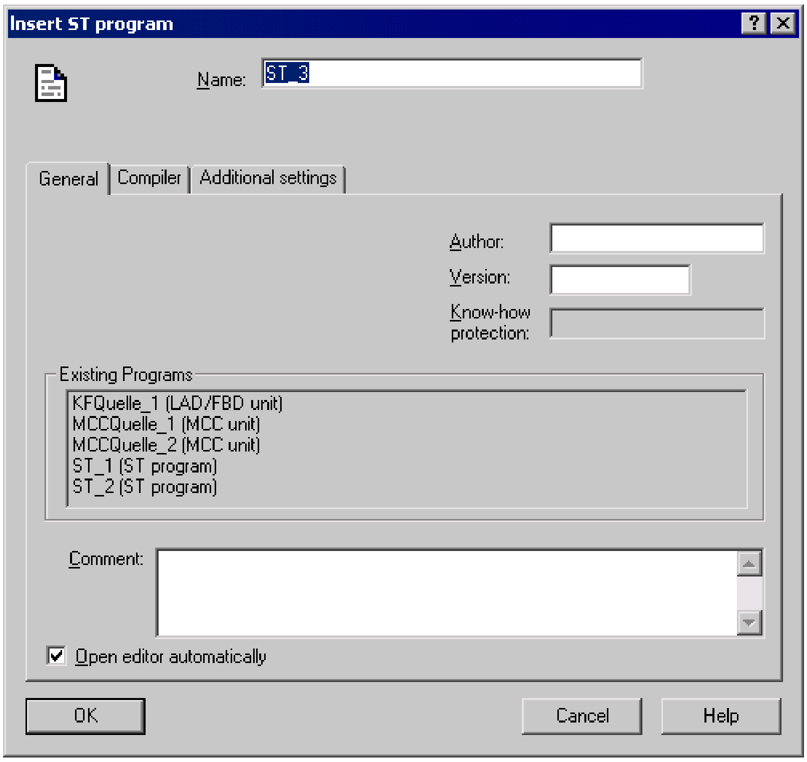
   Names must be unique within the SIMOTION device.

   Protected or reserved identifiers (Page 75) are not allowed.

   Existing program sources (e.g. ST source files, MCC units) are displayed.
5. If necessary, select further tabs to make local settings (only valid for this ST source file):

   – **Compiler** tab: Local settings of the compiler (Page 46) for code generation and message display.

   – **Additional settings** tab: Definitions for preprocessor (Page 51)
6. Select the **Open editor automatically** checkbox.
7. Confirm with **OK**.



Insert ST program

Name: ST_3

Author:

Version:

Know-how protection:

Existing Programs

KFQuelle_1 (LAD/FBD unit)
MCCQuelle_1 (MCC unit)
MCCQuelle_2 (MCC unit)
ST_1 (ST program)
ST_2 (ST program)

Comment:

☑ Open editor automatically

Figure 2-3 Insert ST source file

NOTICE
With versions of the SIMOTION Kernel up to V4.0, a violation of the permissible length of the program source file name may not be detected until a consistency check or a download of the program source file is performed!

2.3.2 Opening an existing ST source file

Proceed as follows

1. Open the subtree of the appropriate SIMOTION device in the project navigator.
2. Open the **PROGRAMS** folder.
3. Select the desired ST source file.
4. Select the **Edit > Open object** menu command.
5. Only for ST source files with know-how protection:

   If the user with the login assigned to the ST source file has not yet logged on:

   – Enter the corresponding password for the displayed login.

   You can now open additional ST source files to which the same login is assigned without having to re-enter the password.

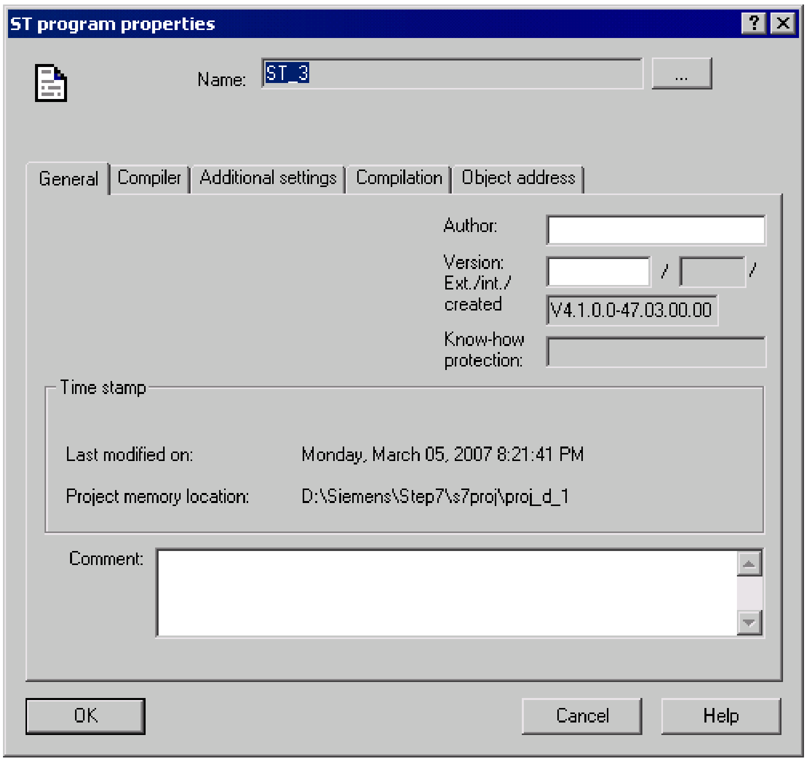
Note

You can also double-click the required ST source file to open it.

2.3.3 Changing the properties of an ST source file

Proceed as follows

1. Under the SIMOTION device, open the **PROGRAMS** folder.
2. Select the desired ST source file.
3. Select the **Edit > Object Properties** menu command.
4. If necessary, select further tabs to make local settings (only valid for this ST source file):
   - **General** tab: General details for the ST source, e.g. timestamp of the last change and the storage location of the project (see figure).
   - **Compiler** tab: Local settings of the compiler (Page 46) for code generation and message display.
   - **Additional settings** tab: Definitions for the preprocessor (Page 51) and display the compiler options (Page 49) as specified for the current settings of the compiler.
   - **Compilation** tab: Display of the compiler options (Page 49) for the last compilation of the ST source.
   - **Object address** tab: Set the internal object address of the ST source. The object addresses of the other program sources are displayed.



Name: ST_3

Author:

Version: Ext./int./ created / / V4.1.0.0-47.03.00.00

Know-how protection:

Time stamp

Last modified on: Monday, March 05, 2007 8:21:41 PM

Project memory location: D:\Siemens\Step7\s7proj\proj_d_1

Comment:

Figure 2-4 Properties of an ST source file

Changing the name of an ST source file

You can also change the names of the ST source file here. To do this, click the [...] button.

Names for program source files must satisfy the rules for identifiers: They are made up of letters (A … Z, a … z), numbers (0 … 9) or single underscores (_) in any order, whereby the first character must be a letter or underscore. No distinction is made between upper and lower case letters.

The permissible length of the name depends on the SIMOTION Kernel version:

- As of Version V4.1 of the SIMOTION Kernel: maximum 128 characters.
- Up to Version V4.0 of the SIMOTION Kernel: maximum 8 characters.

Names must be unique within the SIMOTION device.

Protected or reserved identifiers (Page 75) are not allowed.

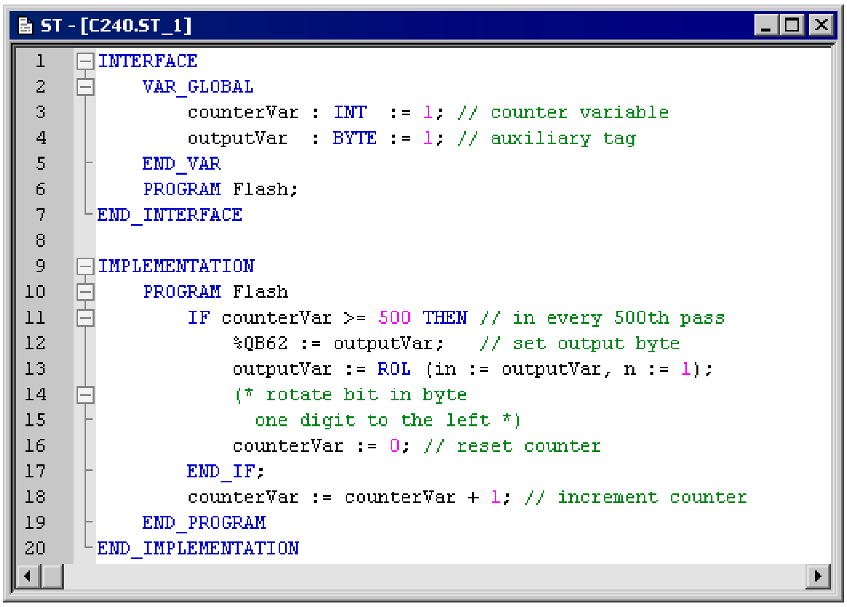
Existing program sources (e.g. ST source files, MCC units) are displayed.

NOTICE
With versions of the SIMOTION Kernel up to V4.0, a violation of the permissible length of the program source file name may not be detected until a consistency check or a download of the program source file is performed!

2.3.4 Working with the ST editor

The ST editor makes it easier for you to work with the ST source file, variables and technology objects through the following operator controls:

- Syntax coloring
- Drag&drop
- Menu commands and shortcuts



```
INTERFACE
    VAR_GLOBAL
        counterVar : INT  := 1; // counter variable
        outputVar  : BYTE := 1; // auxiliary tag
    END_VAR
    PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
    PROGRAM Flash
        IF counterVar >= 500 THEN // in every 500th pass
            %QB62 := outputVar;   // set output byte
            outputVar := ROL (in := outputVar, n := 1);
            (* rotate bit in byte
              one digit to the left *)
            counterVar := 0; // reset counter
        END_IF;
        counterVar := counterVar + 1; // increment counter
    END_PROGRAM
END_IMPLEMENTATION
```

Figure 2-5 Opened ST source file in the ST editor

See also

Shortcuts (Page 27)

2.3.4.1 Syntax coloring

The ST editor represents language elements in different colors:

- Blue: Keywords and compiler built-in functions
- Magenta: Numbers, values
- Green: Comments
- Black: Technology objects, user code, variables

2.3.4.2 Drag&drop

Drag&drop

A drag-and-drop operation (dragging while keeping the left mouse button pressed) enables you to:

- Move selected text areas within an ST source file or to another opened ST source file.
- Copy names of variables from the symbol browser to the ST source file.
- Copy names (e.g. of technology objects, functions or function blocks) from the project navigator to the ST source file.
- Copy system functions from the command library to the ST source file.

To copy names of variables from the symbol browser to the ST source file:

1. Select the entire line of the desired variable in the symbol browser. To do this, click the line number at the start of the line.
2. Press the left mouse button and drag the line number to the desired position in the ST source file.

   The name of the selected variable is inserted in the ST source file.

To copy the name of an element (e.g. a technology object, a function or a function block) from the project navigator to the ST source file:

1. Select the **Project** tab in the project navigator.
2. Select the element in the project navigator.
3. Press the left mouse button and drag the element to the desired position in the ST source file.

   The name of the selected element is inserted in the ST source file.

To copy a system function from the command library to the ST source file:

1. Select the **Command Library** tab in the project navigator.
2. Select the system function in the command library.
3. Press the left mouse button and drag the system function to the desired position in the ST source file.

   The system function is inserted in the ST source file with its parameters.

2.3.4.3 Shortcuts

The ST editor also provides keyboard shortcuts. Commands can currently also be called via the **Edit** and **ST editor** menus:

Table 2-1 ST Editor keyboard shortcuts

Shortcuts	Description
DEL	Delete the selected area (Menu **Edit > Delete**)
F2	Jump to the next bookmark
Arrow key	Move the cursor
SHIFT+F2	Jump to the previous bookmark
SHIFT+Arrow key	Select line of text
CTRL+A	Select all text (Menu **Edit > Select All**)
CTRL+B	Save and compile ST source file (menu **ST source > Accept and compile**)
CTRL+C	Copy the selected area to the clipboard (Menu **Edit > Copy**)
CTRL+D	Duplicate Row
CTRL+F	Find text in ST source file (Menu **Edit > Find**)
CTRL+H	Replace text in ST source file (Menu **Edit > Replace**)
CTRL+L	Copy line
CTRL+V	Paste clipboard contents (Menu **Edit > Paste**)
CTRL+X	Cut the selected area (Menu **Edit > Cut**)
CTRL+Y	Redo the last action (Menu **Edit > Redo**)
CTRL+Z	Undo the last action (Menu **Edit > Undo**)
CTRL+space	Automatic completion
CTRL+F2	Set or delete bookmarks
CTRL+F4	Close ST source (Menu **ST source > Close**)
CTRL+F7	Activation and deactivation of the Program Status function (menu **ST source > Program Status on/off**)
CTRL+SHIFT+F2	Delete all bookmarks in the ST source code
CTRL+SHIFT+F3	Arrange windows, tile horizontally
CTRL+SHIFT+F5	Arrange windows, tile vertically
CTRL+SHIFT+F8	Format selected area
CTRL+SHIFT+F9	Move cursor to the start of the current or higher-level block
CTRL+SHIFT+F10	Move cursor to the end of the current block
CTRL+SHIFT+F11	Move cursor to the start of the higher-level block, 1st level
CTRL+SHIFT+F12	Move cursor to the start of the higher-level block, 2nd level
CTRL+ALT+B	Display bracket pairs in the current ST source file
CTRL+ALT+C	Folding: Hide all blocks of the current ST source file
CTRL+ALT+D	Folding: Display all blocks of the current ST source file
CTRL+ALT+F	Folding: Display or hide folding information in the current ST source file
CTRL+ALT+I	Display indentation level in the current ST source file
CTRL+ALT+L	Display or hide line numbers in the current ST source file.
CTRL+ALT+R	Folding: Display all subordinate blocks

Shortcuts	Description
CTRL+ALT+T	Folding: Display/hide block
CTRL+ALT+V	Folding: Hide all subordinate blocks
CTRL+ALT+W	Display or hide spaces and tabs in the current ST source file
CTRL+ADD (numeric keypad)	Increase font size in the current ST source file
CTRL+MINUS (numeric keypad)	Decrease font size in the current ST source file
CTRL+DIV (numeric keypad)	Change font size in the current ST source file to 100%
ALT+SHIFT+Arrow key	Select text by column
ALT+SHIFT+L	Change selected text to upper case
ALT+SHIFT+U	Change selected text to lower case

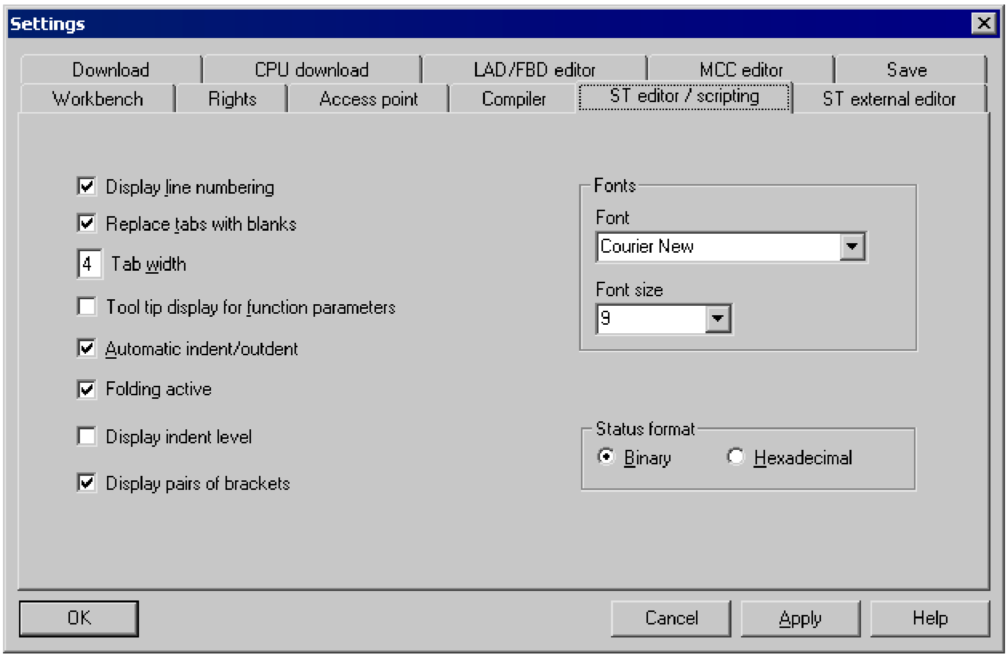
Table 2-2 Combined keyboard and mouse actions

Keyboard	Mouse	Description
	Single left click in text	Set cursor
	Double left click in text	Select word
	Press left button and drag mouse	Select line of text
	Single left click on line number	Select line
SHIFT	Single left click in text	Select line of text
CTRL	Single left click on line number	Select all text (Menu **Edit > Select All**)
CTRL	Single left click in bookmark column	Set bookmarks
CTRL	Turn mouse wheel	Change font size
ALT	Press left button and drag mouse	Select text by column
ALT+SHIFT	Single left click in text	Select text by column

2.3.4.4 Settings of the ST editor

Proceed as follows:

1. Select the menu **Tools > Settings**.
2. Select the **ST editor / Scripting** tab.
3. Enter the settings.
4. Click **OK** or **Accept** to confirm.



☑ Display line numbering
☑ Replace tabs with blanks
4 Tab width
□ Tool tip display for function parameters
☑ Automatic indent/outdent
☑ Folding active
□ Display indent level
☑ Display pairs of brackets

Fonts
Font: Courier New
Font size: 9

Status format
● Binary ○ Hexadecimal

Figure 2-6 ST Editor / Scripting

The settings also apply to the script editor.

The table below contains a description of the individual parameters.

Table 2-3 Parameter settings ST Editor / Scripting

Parameter	Description
Display line numbering	If active, the line numbers are displayed. See: Other ST editor tools (Page 42).
Replace tabs with blanks	You select here how text indentation is performed (for the automatic indentation or by pressing the Tab key): • If active: By adding the appropriate number of space characters ($20). • If inactive: By adding the tab character ($09). See: Indentations and tabs (Page 30).
Tab width	Number of characters skipped by a tab. See: Indentations and tabs (Page 30).

Parameter	Description
Tooltip display for function parameters	When active, the parameters are displayed as tooltips for the functions.
Automatic indent/outdent	If active, for the text input, source file sections and blocks are indented automatically by the set tab width. See: Indentations and tabs (Page 30).
Folding active	If active, the column with the folding information is displayed at the left-hand side next to the edit area. You can then hide blocks in an ST source file so that only the first line of the block remains visible. See: Fold (show and hide blocks) (Page 32)
Display indentation level	If active, you can optically highlight the indent and outdent for blocks using vertical help lines (in accordance with the set tab size). See: Indentations and tabs (Page 30).
Display bracket pairs	If active, the associated bracket of the pair that belongs to the bracket where the cursor is located will be found and optically highlighted. See: Other ST editor tools (Page 42).
Font	Font for the display of the text in the ST editor. All non-proportionally spaced fonts installed on the PC are available for selection.
Font size	Font size (in pt) for the display of the text in the ST editor. See: Change the font size in the ST editor (Page 36).
Status format	Format in which the variable values are displayed for the program status (for ST editor only). See: Properties of the program status (Page 265).

2.3.4.5 Indentations and tabs

Specify tab width

The standard tab width for all ST sources is specified in the settings of the ST editor (Page 29).

This setting is used for all ST source files opened subsequently.

Indent using tabs or spaces

You can select in the settings of the ST editor (Page 29) how the text will be indented (e.g. with the automatic indent and outdent when the Tab key is pressed):

- By adding the appropriate number of space characters ($20).
- By adding the tab character ($09).

This setting is used for all ST source files opened subsequently.

Automatically indent and outdent blocks

The ST editor recognizes blocks introduced with a keyword and terminated with another keyword, e.g.:

- INTERFACE / END_INTERFACE
- IMPLEMENTATION / END_IMPLEMENTATION
- Declaration blocks (e.g. TYPE / END_TYPE, VAR / END_VAR)
- Program organization units (e.g. PROGRAM / END_PROGRAM)
- Control statements (e.g. IF / END_IF, FOR / END_FOR)

During the text input, the ST editor can automatically indent text within blocks by the tab size. The end line of the block will be outdented automatically.

This function is activated in the settings of the ST editor (Page 29).

Note

This setting affects only the behavior during the text input. It does not have any effect on existing text in the ST sources.

Format selection

You can use this function to force the blocks (see above) in an existing text to be indented by the tab size in accordance with their hierarchy. The number of the leading spaces or tabs will be changed:

- As specified by the current tab size of the ST source file.
- As specified by the current setting for the type of the indent (with tabs or spaces).

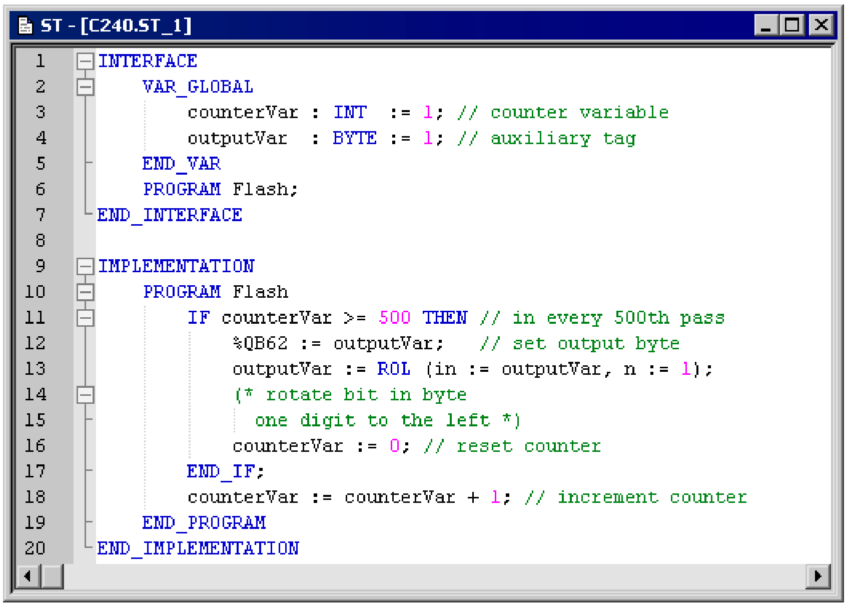
Follow these steps:

1. Select the text area in the ST editor that you want to format (see Select text (Page 37)).
2. Press the **CTRL+SHIFT+F8** key combination.

NOTICE
Leading tabs or spaces will be replaced in a line only when the formatting changes their number.

Display indentation level

You can optically highlight the indent and outdent for blocks using vertical help lines (in accordance with the set tab size).



```
INTERFACE
    VAR_GLOBAL
        counterVar : INT  := 1; // counter variable
        outputVar  : BYTE := 1; // auxiliary tag
    END_VAR
    PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
    PROGRAM Flash
        IF counterVar >= 500 THEN // in every 500th pass
            %QB62 := outputVar;   // set output byte
            outputVar := ROL (in := outputVar, n := 1);
            (* rotate bit in byte
              one digit to the left *)
            counterVar := 0; // reset counter
        END_IF;
        counterVar := counterVar + 1; // increment counter
    END_PROGRAM
END_IMPLEMENTATION
```

Figure 2-7 ST source with visible indent aid

You can activate or deactivate this function:

- For the active ST source
  – Press the **CTRL+ALT+I** key combination.
- For all open ST sources:
  – Activate or deactivate the **Display indentation level** checkbox in the ST editor settings (Page 29).

2.3.4.6 Folds (show and hide blocks)

You can hide blocks in an ST source file so that only the first line of the block remains visible. This increases the legibility during the editing or reading of an ST source file.

A block is introduced with a keyword and terminated with another keyword, e.g.:

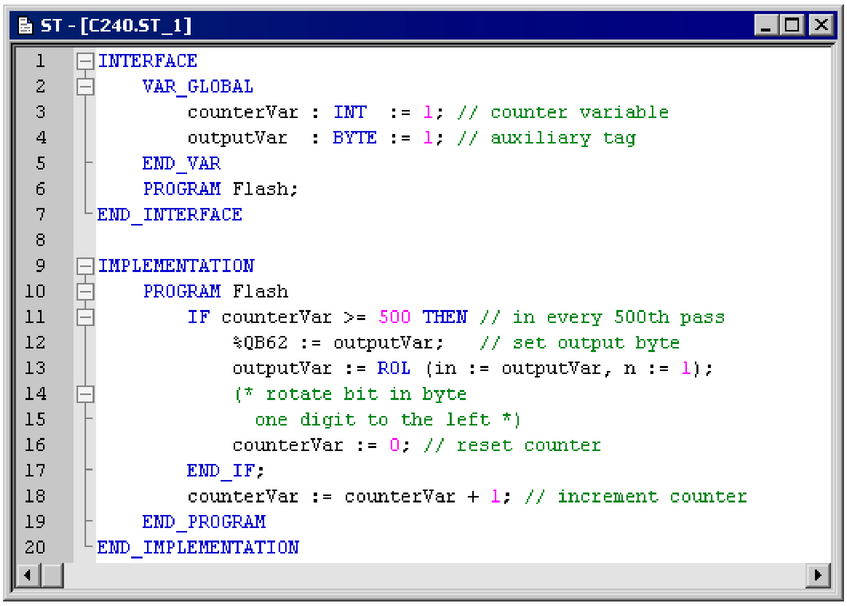
- INTERFACE / END_INTERFACE
- IMPLEMENTATION / END_IMPLEMENTATION
- Declaration blocks (e.g. TYPE / END_TYPE, VAR / END_VAR)
- Program organization units (e.g. PROGRAM / END_PROGRAM)
- Control statements (e.g. IF / END_IF, FOR / END_FOR)
- Block comment (* / *)

How to recognize that a block is displayed:

- When the column is shown with the fold information (at the left-hand side next to the editing area), a **minus** character appears next to the first line of the block.

How to recognize that a block is hidden:

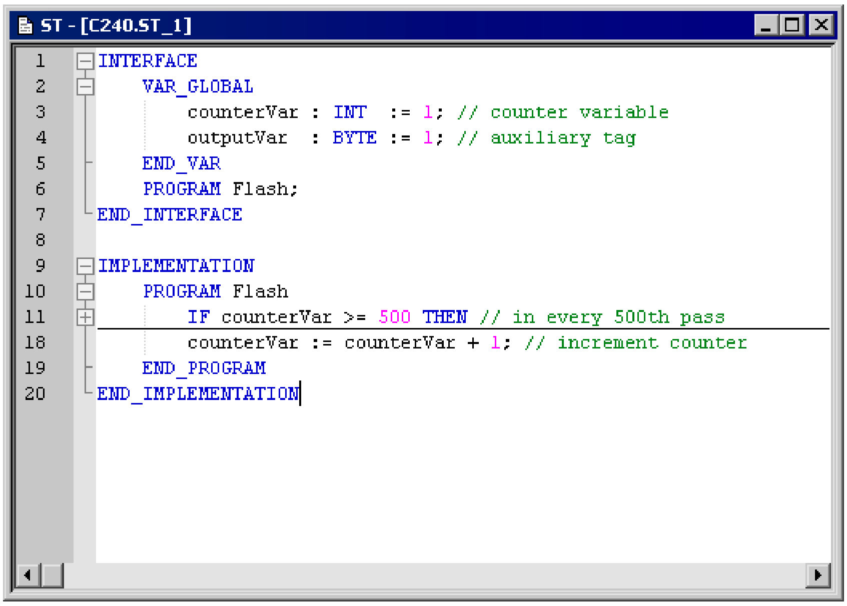
- When the column is shown with the fold information (at the left-hand side next to the editing area), a **plus** character appears next to the first line of the block.
- A hyphen is displayed below this line.



```
INTERFACE
    VAR_GLOBAL
        counterVar : INT  := 1; // counter variable
        outputVar  : BYTE := 1; // auxiliary tag
    END_VAR
    PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
    PROGRAM Flash
        IF counterVar >= 500 THEN // in every 500th pass
            %QB62 := outputVar;   // set output byte
            outputVar := ROL (in := outputVar, n := 1);
            (* rotate bit in byte
              one digit to the left *)
            counterVar := 0; // reset counter
        END_IF;
        counterVar := counterVar + 1; // increment counter
    END_PROGRAM
END_IMPLEMENTATION
```

Figure 2-8 ST source for which all blocks are shown



```
INTERFACE
    VAR_GLOBAL
        counterVar : INT  := 1; // counter variable
        outputVar  : BYTE := 1; // auxiliary tag
    END_VAR
    PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
    PROGRAM Flash
        IF counterVar >= 500 THEN // in every 500th pass
        counterVar := counterVar + 1; // increment counter
    END_PROGRAM
END_IMPLEMENTATION
```

Figure 2-9 ST source with hidden IF block (including block comment)

Proceed as follows:

How to show or hide the column with the fold information (at the left-hand side of the editing area):

- For the active ST source:
  - Press the **CTRL+ALT+F** key combination.
- For all open ST sources:
  - Activate or deactivate the **Folding active** checkbox in the settings of the ST editor (Page 29).

How to hide a block:

- Click on the **minus** character in the column with the fold information.

  Only the first line of the block remains visible. All subsequent lines of the block (including lower-level blocks) will be hidden.

How to show a block:

- Click on the **plus** character in the column with the fold information.

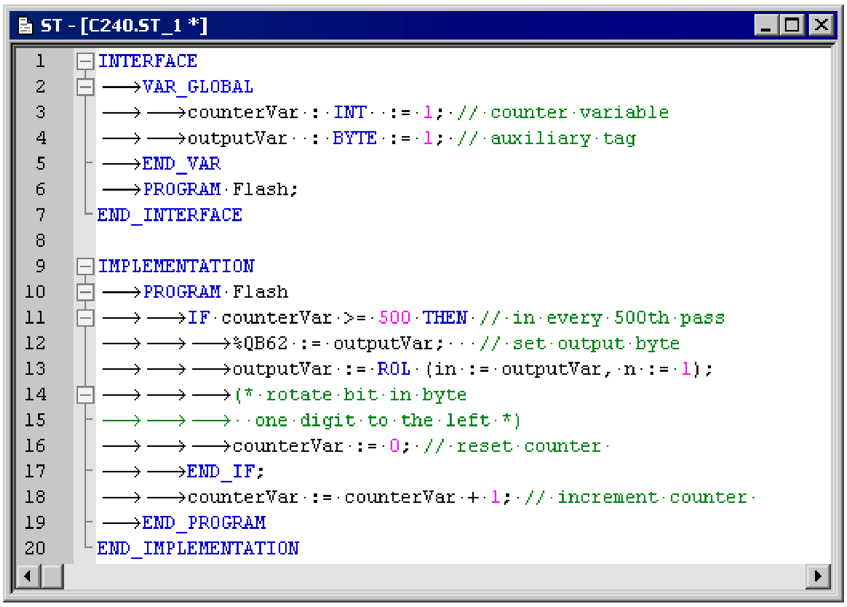
  All subsequent lines of the block will be shown. Lower-level blocks will be displayed in the state they had when they were hidden.

Note

After opening an ST source in the editor, all lines of the ST source are visible. All blocks are shown.

2.3.4.7 Display spaces and tabs

You can display spaces and tabs in the ST source files.



```
INTERFACE
→VAR_GLOBAL
→→counterVar·:·INT··:=·1;·//·counter·variable
→→outputVar··:·BYTE·:=·1;·//·auxiliary·tag
→END_VAR
→PROGRAM·Flash;
END_INTERFACE

IMPLEMENTATION
→PROGRAM·Flash
→→IF·counterVar·>=·500·THEN·//·in·every·500th·pass
→→→%QB62·:=·outputVar;···//·set·output·byte
→→→outputVar·:=·ROL·(in·:=·outputVar,·n·:=·1);
→→→(*·rotate·bit·in·byte
→→→··one·digit·to·the·left·*)
→→→counterVar·:=·0;·//·reset·counter·
→→END_IF;
→→counterVar·:=·counterVar·+·1;·//·increment·counter·
→END_PROGRAM
END_IMPLEMENTATION
```

Figure 2-10 ST source file with visible spaces and tabs

Proceed as follows

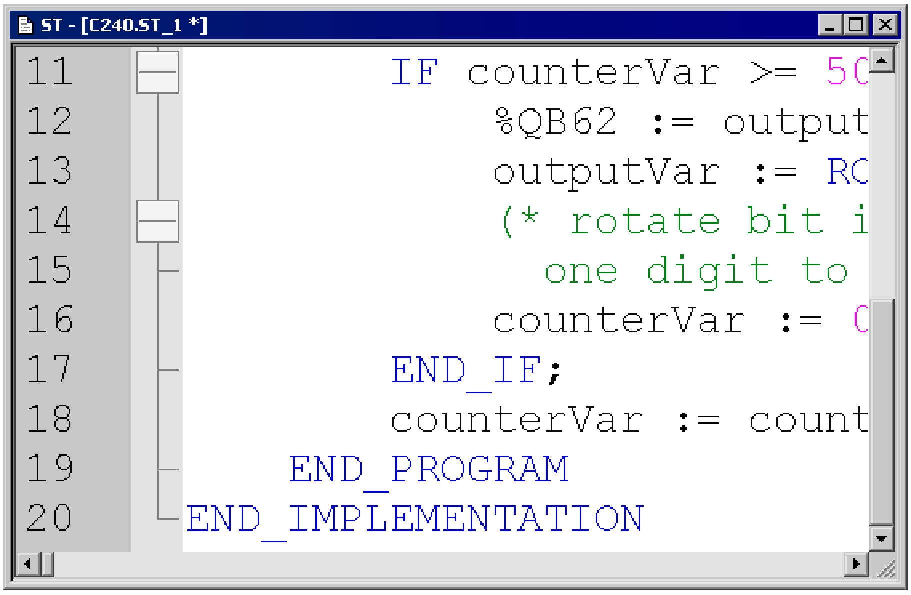
How to specify whether spaces and tabs are displayed in the active ST source file:

1. Set the cursor in the opened ST source.
2. Press the **CTRL+ALT+W** key combination.

This setting is not saved when the ST source is closed.

2.3.4.8 Changing the font size in the ST editor

You can change the font size of the ST source in the editor. The font size of the line numbers and the size of other display elements (e.g. fold marks, bookmarks) will also be changed.



```
    IF counterVar >= 50
      %QB62 := output
      outputVar := RO
      (* rotate bit i
       one digit to
      counterVar := 0
    END_IF;
    counterVar := count
  END_PROGRAM
END_IMPLEMENTATION
```

Figure 2-11 Increased size display of the ST source

Proceed as follows

You can change the font size:

- For the current ST source
- For ST source files to be opened subsequently

How to change the font size for the current ST source (alternative):

- Press the **CTRL** key while moving the mouse wheel
- Press concurrently the **CTRL** key and one of the following keys on the numeric block:
  - **ADD** (+) to increase,
  - **MINUS** (-) to reduce,
  - **DIV** for 100%.

How to change the font size for ST sources to be opened subsequently:

1. Open the settings for the ST editor (see Settings of the ST editor (Page 29)).
2. Enter the required font size.

This setting will used for all ST sources that will be opened subsequently. It does not affect the currently opened ST sources.

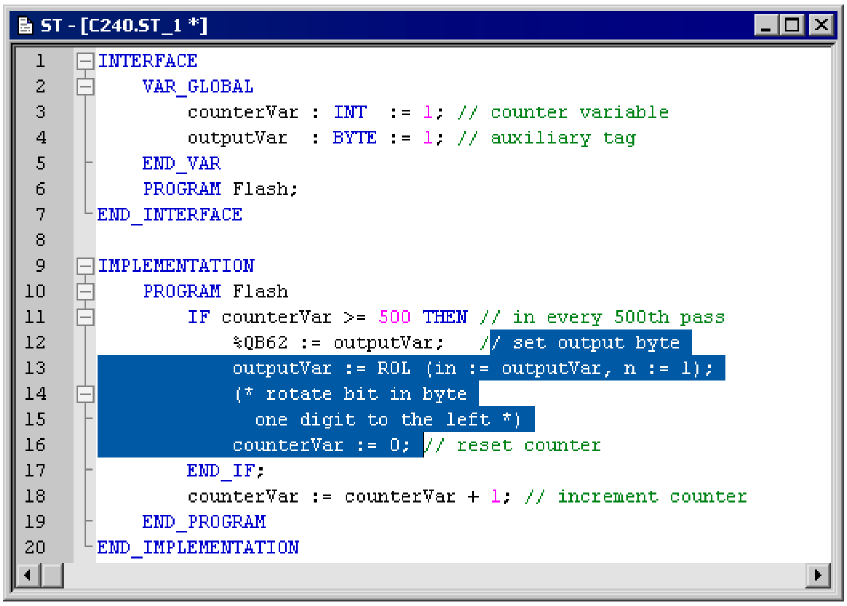
2.3.4.9 Select text

Selecting lines of text

How to select lines of text:

- With the mouse:
  – With pressed left mouse button, scan the text to be selected.

  or
- With the keyboard or the mouse:
  – Place the cursor with the arrow keys of the keyboard or with the mouse at the start of the text to be selected.
  – Press the **Shift** key while placing the cursor at the end of the text to be selected.



```
INTERFACE
    VAR_GLOBAL
        counterVar : INT  := 1; // counter variable
        outputVar  : BYTE := 1; // auxiliary tag
    END_VAR
    PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
    PROGRAM Flash
        IF counterVar >= 500 THEN // in every 500th pass
            %QB62 := outputVar;    // set output byte
            outputVar := ROL (in := outputVar, n := 1);
            (* rotate bit in byte
              one digit to the left *)
            counterVar := 0; // reset counter
        END_IF;
        counterVar := counterVar + 1; // increment counter
    END_PROGRAM
END_IMPLEMENTATION
```

Figure 2-12 ST source with selected lines of text

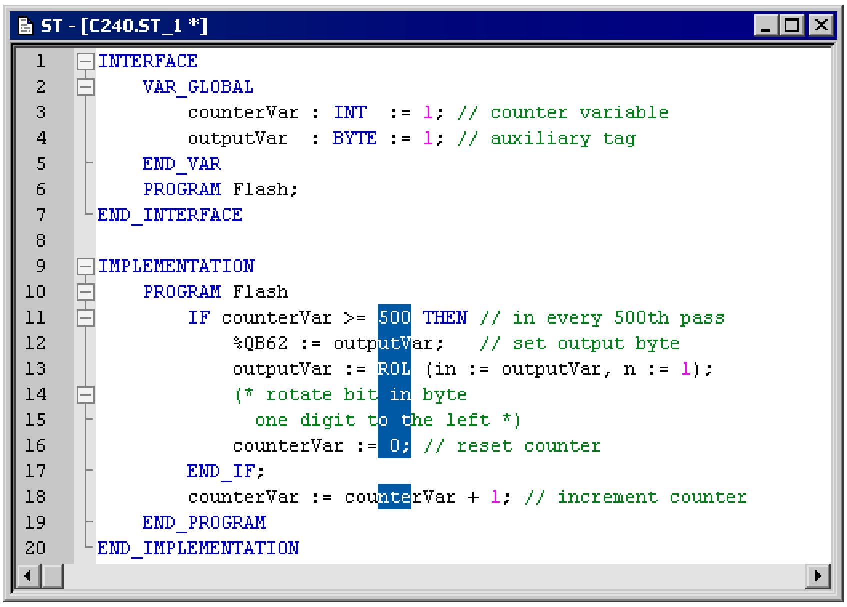
Selecting columns of text

How to select columns of text:

- With the mouse:
  – Press the **Alt** key while keeping the left mouse button pressed, scan the text to be selected.

  or

- With the keyboard or the mouse:
  - Place the cursor with the arrow keys of the keyboard or with the mouse at the start of the text to be selected.
  - Press the **ALT+SHIFT** key combination while placing the cursor at the end of the text to be selected.



```
INTERFACE
    VAR_GLOBAL
        counterVar : INT  := 1; // counter variable
        outputVar  : BYTE := 1; // auxiliary tag
    END_VAR
    PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
    PROGRAM Flash
        IF counterVar >= 500 THEN // in every 500th pass
            %QB62 := outputVar;   // set output byte
            outputVar := ROL (in := outputVar, n := 1);
            (* rotate bit in byte
              one digit to the left *)
            counterVar := 0; // reset counter
        END_IF;
        counterVar := counterVar + 1; // increment counter
    END_PROGRAM
END_IMPLEMENTATION
```

Figure 2-13 ST source with selected columns of text

Selecting a single line

How to select a single line:

- Click with the left mouse button next to the line number of the appropriate line.

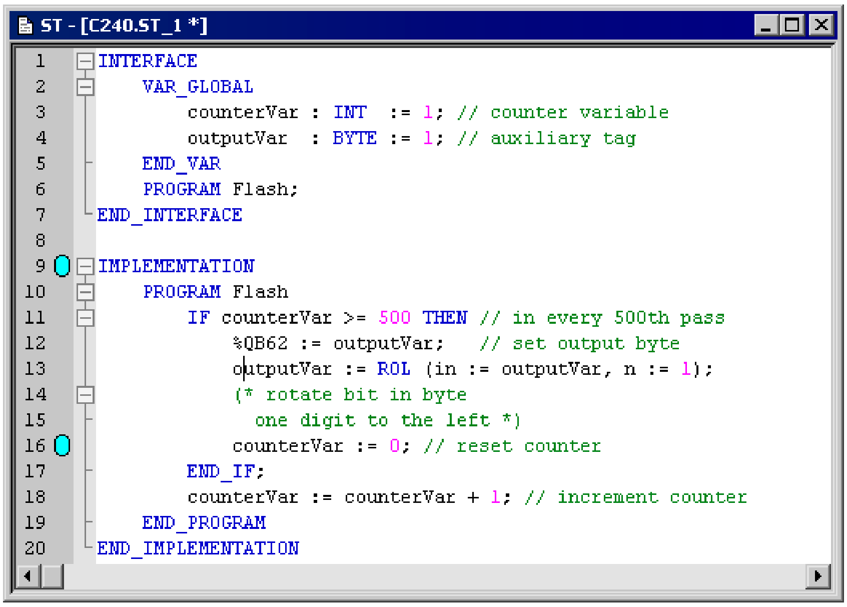
Selecting the complete text

How to select the complete text (alternatives):

- Press the **CTRL** key while clicking with the left mouse button in the column with the line numbers.
- Press the **CTRL+A** key combination.

2.3.4.10 Use bookmarks

You can set bookmarks in the ST editor. This allows you to jump to specific selected lines within the ST source file.



```
INTERFACE
    VAR_GLOBAL
        counterVar : INT  := 1; // counter variable
        outputVar  : BYTE := 1; // auxiliary tag
    END_VAR
    PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
    PROGRAM Flash
        IF counterVar >= 500 THEN // in every 500th pass
            %QB62 := outputVar;   // set output byte
            outputVar := ROL (in := outputVar, n := 1);
            (* rotate bit in byte
              one digit to the left *)
            counterVar := 0; // reset counter
        END_IF;
        counterVar := counterVar + 1; // increment counter
    END_PROGRAM
END_IMPLEMENTATION
```

Figure 2-14 ST source with bookmarks

Setting and deleting bookmarks

How to set a bookmark for a line of the active ST source file or to delete an existing bookmark:

- With the keyboard and the mouse:
  - Press the **Ctrl** key.
  - Simultaneously, click with the left mouse button at the right-hand side next to the line number of the appropriate line.
- With the keyboard:
  - Set the cursor in the appropriate line of the ST source.
  - Press the **CTRL+F2** key combination.

NOTICE
Bookmarks are not saved when the ST source is closed.

Jump to bookmark

How to jump to the next bookmark within the ST source:

- Press the **F2** key.

How to jump to the previous bookmark within the ST source:

- Press the **SHIFT+F2** key combination.

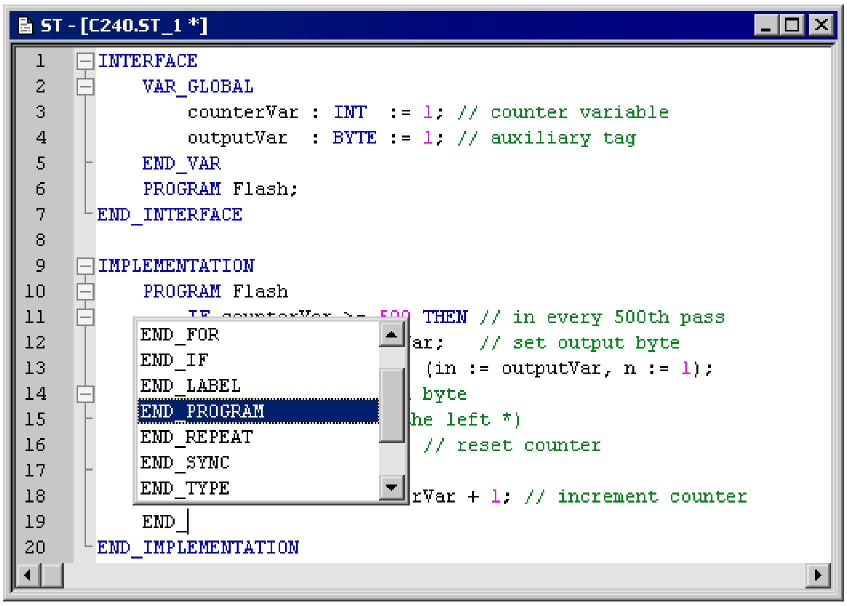
Delete all bookmarks

How to delete all bookmarks in an ST source:

- Press the **CTRL+SHIFT+F2** key combination.

2.3.4.11 Automatic completion

In the ST editor, you can automatically complete identifiers. A selection list with identifiers that begin with the previously entered characters will be displayed.



```
INTERFACE
    VAR_GLOBAL
        counterVar : INT  := 1; // counter variable
        outputVar  : BYTE := 1; // auxiliary tag
    END_VAR
    PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
    PROGRAM Flash
        IF counterVar >= 500 THEN // in every 500th pass
                              ar;    // set output byte
                               (in := outputVar, n := 1);
                              . byte
                              he left *)
                               // reset counter

                              rVar + 1; // increment counter
    END_
END_IMPLEMENTATION
```

Figure 2-15 ST editor, automatic completion of an identifier (e.g. END_)

Proceed as follows

How to automatically complete an identifier:

1. Write the first characters of the identifier (e.g. the letters of a word).
2. Press the **Ctrl+space** key combination.

   The selection possibilities are displayed in a window.
3. Select the required identifier.

Note

If only a single identifier is offered for selection, the selection window will not be opened and the identifier completed immediately.

Functional description

The following identifiers that begin with the specified character will be offered:

- Keywords of the Structured Text language
- Identifiers from the command library
- For technology objects including their system variables and configuration data
- Identifiers of the own ST source:
  - Program organization units (POU)
  - Data types
  - Variables and constants
  - Structure elements
- Identifiers from imported program sources

Note

Identifiers from the own ST source and from imported program sources will be displayed correctly only when the corresponding program source has been compiled.

The display is made context-sensitive, only those types of identifiers that are appropriate at the associated location of the ST source will be offered:

- Within a declaration block, only data types and keywords
- Within a program organization unit (POU), no data types
- For a structure (e.g. var_struct.xx), only structure components

---

2.3.4.12 Other help for the ST editor

Display bracket pairs

The two brackets of a bracket-pair can be optically highlighted.

To do this, place the cursor next to a bracket. The editor attempts to find the associated brackets of the pair and possibly displays both brackets red. This simplifies the recognition of bracket pairs, in particular for nesting.

How to switch this function on or off:

- For the active ST source:
  - Press the **CTRL+ALT+B** key combination.
- For all open ST sources:
  - Activate or deactivate the **Display bracket pairs** checkbox in the ST editor settings (Page 29).

  This setting is also used for all ST source files opened subsequently.

Show and hide line numbers

Line numbers can be displayed in the ST editor:

How to switch this function on or off:

- For the active ST source file:
  - Press the **CTRL+ALT+L** key combination.
- For all open ST sources:
  - Activate or deactivate the **Display line numbers** checkbox in the ST editor settings (Page 29).

  This setting is also used for all ST source files opened subsequently.

2.3.4.13 Using the command library

The command library is a tab in the project navigator. It contains the available system functions, system function blocks, and operators.

You can drag these elements from the command library to the ST editor window with drag&drop.

2.3.4.14 ST editor toolbar

This toolbar contains important operating actions for programming:

Table 2-4 ST editor toolbar

Symbol	Meaning
	Program status Click this icon to start the **program status** test mode. During the program execution, you can monitor the values of the variables marked in the ST source. The following prerequisites are necessary: 1. The program must be compiled with the appropriate compiler option. 2. The project and the program must be loaded into the target system. 3. An online connection to the target system must have been established. Reclick this icon to end the **program status**. See: Using the program status (Page 266).
⏸	Stop monitoring of the program variables Click this icon in the **program status** test mode to stop the monitoring of the program variables. See Using the program status (Page 266).
▶	Continue monitoring of the program variables Click this icon in the **program status** test mode to continue the monitoring of the program variables. See: Using the program status (Page 266).
	Refresh Click this icon in the **program status** test mode to force the updating of the displayed values. The monitoring of the program variables must have been activated. See: Using the program status (Page 266).
📄	Insert ST source file Click this icon to create a new ST source file. The icon is active only when the **PROGRAMS** folder where the ST source file is to be saved is selected in the project navigator. See: Insert ST source file (Page 21).
	Accept and compile Click this icon to transfer the current ST source file to the project and compile into executable code. See: Starting the compiler (Page 44).

2.3.5 Starting the compiler

Requirement

The ST source file has been opened with the ST editor.

Proceed as follows

1. Click in the window with the ST editor. The dynamic ST source file menu appears.
2. Select the **ST source file** > **Accept and compile** menu command.

Note

The ST source file menu is dynamic. It only appears if the window of the ST editor is active.

The compiler checks the syntax of the ST source file. The "Compile/check output" tab of the detail view displays the successful compilation of the source text or compiler errors. The error details include: The name of the ST source file, the number of the line in which the error occurred, the error number and the error description.

2.3.5.1 Help for the error correction

To obtain help during error correction:

- Double-click the error message in the **Compile/check output** tab of the detail view.

The cursor is placed at the relevant line in the ST source file.

2.3.6 Making settings for the compiler

You can define the compiler settings (compiler options) as follows:

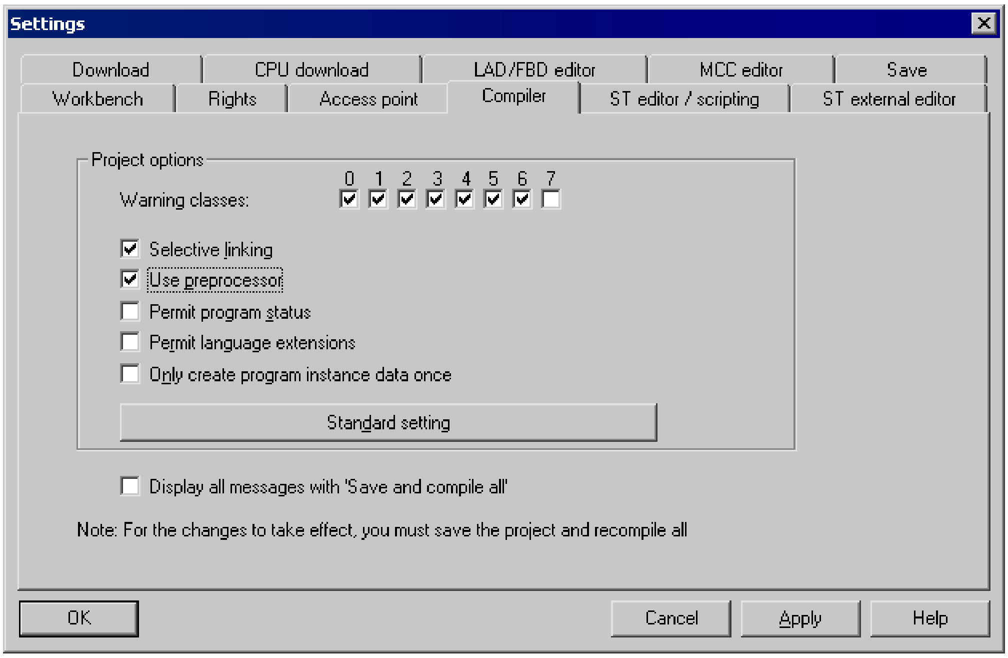
- Globally for the SIMOTION project, valid for all programming languages, seeGlobal settings of the compiler (Page 45)
- Locally for an individual ST source within the SIMOTION project, see Local settings of the compiler (Page 46)

2.3.6.1 Global compiler settings

The global setting are valid for all programming languages within the SIMOTION project.

Proceed as follows

1. Select the menu **Tools > Settings.**
2. Select the **Compiler** tab.
3. Define the settings according to the following table.
4. Confirm with **OK**.



Warning classes: 0 ☑ 1 ☑ 2 ☑ 3 ☑ 4 ☑ 5 ☑ 6 ☑ 7 □

☑ Selective linking
☑ Use preprocessor
□ Permit program status
□ Permit language extensions
□ Only create program instance data once

□ Display all messages with 'Save and compile all'

Note: For the changes to take effect, you must save the project and recompile all

Figure 2-16 Global compiler settings

Parameter

Table 2-5 Parameters for global compiler settings

Parameter	Description
Warning classes[1]	**Active**: In addition to the error messages, the compiler outputs warning messages of the selected classes. **Inactive**: The compiler suppresses the warning messages of the respective class. See also For meanings of the warning classes (Page 49).
Selective linking[1]	**Active** (standard): Unused code is removed from the executable program. **Inactive**: Unused code is retained in the executable program.
Use preprocessor[1]	**Active**: Preprocessor is used (see Control preprocessor (Page 243)). **Inactive** (standard): Preprocessor is not used.

Parameter	Description
Enable program status[1]	**Active**: Additional program code is generated to enable monitoring of program variables (including local variables). **Inactive** (standard): Program status not possible. See Properties of the program status (Page 265).
Permit language extensions[1]	**Active**: Language elements are permitted that do not comply with IEC 61131-3. **Inactive** (standard): Only language elements that comply with IEC 61131-3 are permitted.
Only create program instance data once[1]	**Active**: The local variables of a program are only stored once in the user memory of the unit. The setting is required when a further program is to be called within a program. **Inactive** (standard): The local variables of a program are stored according to the task assignment in the user memory of the respective task. See Memory ranges of the variable types (Page 194).
Display all messages with *Save and compile all*[2]	Here, you can control the scope of the error log that will be displayed in the workbench's detail view when you call the **Save and compile all** command in SIMOTION SCOUT. **Active**: A detailed log is created that is similar to that for single compilation of an ST source file. **Inactive**: A compressed error log is created.
[1] Local setting also possible, see Local settings of the compiler (Page 46). [2]User-specific settings. Valid for all SIMOTION projects that the user processes.	

NOTICE
You may have to recompile the project for the settings to take effect.

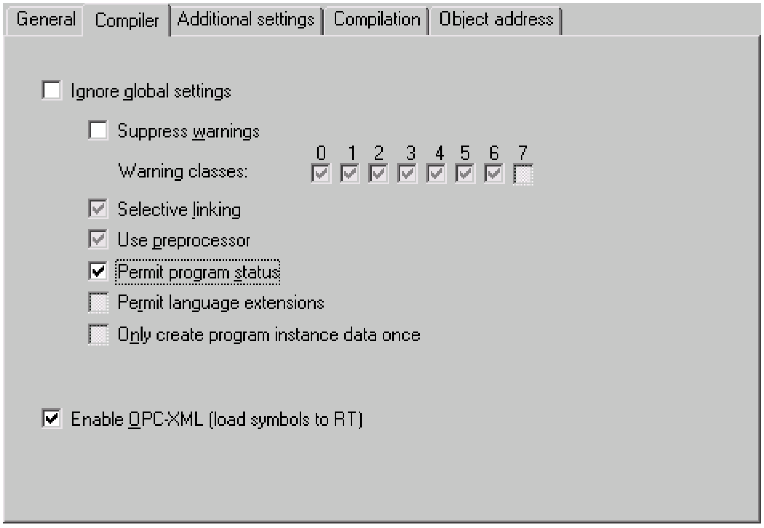
2.3.6.2 Local compiler settings

Local settings are configured individually for each ST source file; local settings overwrite global settings.

Proceed as follows

1. Open the Properties window for the ST source file, see Changing the properties of an ST source (Page 23):

   Select the ST source file in the project navigator and select the **Edit > Object properties** menu command.
2. Select the **Compiler** tab.
3. Define the settings according to the following table.
4. Confirm with **OK**.



□ Ignore global settings

□ Suppress warnings

Warning classes: 0 ☑ 1 ☑ 2 ☑ 3 ☑ 4 ☑ 5 ☑ 6 ☑ 7 □

☑ Selective linking

☑ Use preprocessor

☑ Permit program status

□ Permit language extensions

□ Only create program instance data once

☑ Enable OPC-XML (load symbols to RT)

Figure 2-17 Local compiler settings for the ST source file

Parameter

Table 2-6 Parameters for the local compiler settings for the ST source file

Parameter	Description
Ignore global settings	Affects: • Warning classes • Selective linking • Use preprocessor • Enable program status • Permit language extensions • Only create program instance data once **Active**: Only the selected local settings apply. The global settings are ignored. **Inactive**: The respective global setting can be adopted. The corresponding checkbox is grayed out.
Suppress warnings	In addition to error messages, the compiler can output warnings. You can set the scope of the output warning messages: **Active**: The compiler outputs the warning messages according to the selection in the global settings of the warning classes. The checkboxes of the warning classes can no longer be selected. **Inactive**: The compiler outputs the warning messages according to the following selection of the warning classes.
Warning classes[1]	Only for Suppress warnings = inactive. **Active**: The compiler outputs warning messages of the selected class. **Inactive**: The compiler suppresses warning messages of the respective class. **Grey background**: The displayed global setting is adopted (only for Ignore global settings = inactive). See also For meanings of the warning classes (Page 49).

Parameter	Description
Selective linking[1]	**Active**: Unused code is removed from the executable program. **Inactive**: Unused code is retained in the executable program. **Grey background**: The displayed global setting is adopted (only for Ignore global settings = inactive).
Use preprocessor[1]	**Active**: Preprocessor is used. **Inactive**: Preprocessor is not used. **Grey background**: The displayed global setting is adopted (only for Ignore global settings = inactive). See Controlling the preprocessor (Page 243).
Enable program status[1]	**Active**: Additional program code is generated to enable monitoring of program variables (including local variables). **Inactive**: Program status not possible. **Grey background**: The displayed global setting is adopted (only for Ignore global settings = inactive). See Properties of the program status (Page 265).
Permit language extensions[1]	**Active**: Language elements are permitted that do not comply with IEC 61131-3. **Inactive**: Only language elements are permitted that comply with IEC 61131-3. **Grey background**: The displayed global setting is adopted (only for Ignore global settings = inactive).
Only create program instance data once[1]	**Active**: The local variables of a program are only stored once in the user memory of the unit. The setting is required when a further program is to be called within a program. **Inactive**: The local variables of a program are stored according to the task assignment in the user memory of the respective task. **Grey background**: The displayed global setting is adopted (only for Ignore global settings = inactive). See Memory ranges of the variable types (Page 194).
Enable OPC-XML	**Active:** Symbol information for the unit variables of the ST source is available in the SIMOTION device (required for the *_exportUnitDataSet* and *_importUnitDataSet* functions, see the *SIMOTION Basic Functions* Function Manual. **Inactive:** Icon information is not created,
[1] Global setting also possible, see Global settings of the compiler (Page 45).	

2.3.6.3 Meaning of warning classes

The table lists the warning classes and their meanings.

Table 2-7 Meaning of warning classes

Warning class	Meaning
0	Warnings for unreferenced or unused code sections and data
1	Warnings for hidden identifiers
2	Warnings for data type conversion, e.g. for data change
3	Warnings about set compiler options
4	Warnings about semaphores (potentially faulty functions)
5	Warnings about alarm functions
6	Warnings about constructs in libraries (unit variables declared)
7	Messages of the preprocessor

For the detailed description of the compiler error messages, specify which warning classes are assigned to the individual warnings (Page 361) and information (Page 365).

2.3.6.4 Display of the compiler options

You can view for a program source the following:

- The current compiler options using the global or local settings of the compiler.
- The compiler options used for the last compilation of the program source.

Requirement

The Properties window of the program source (Page 23) is open.

Proceed as follows

To display the current compiler options using the global or local settings of the compiler (Page 44):

- Select the **Additional settings** tab.

  The current compiler options for the program source are displayed. They are valid for a future compilation.

To display the compiler options used for the last compilation of the program source:

- Select the **Compiler** tab.

  The following are displayed for the last compilation of the program source:

  – The version of the used compiler.

  – The used compiler options.

Meaning of the compiler options

Compiler option	Meaning
-c[2]	Do not create debug and symbol information.
-C lang_ext -C lang_iec	"Permit language extensions"[1] active. "Permit language extensions" inactive.
-C opcsym -C no_opcsym	"Permit OPC-XML"[1] active. "Permit OPC-XML" inactive.
-C opcsym -C no_preproc	"Use preprocessor"[1] active. "Use preprocessor" inactive.
-C prog_once -C prog_multi	"Create program instance data only once"[1] active. "Create program instance data only once" inactive.
-D *text*	Preprocessor definition (Page 51).
-e local[2] -e user[2]	Only local settings act. Only global settings act. No details (default): Global settings will be augmented with local settings.
-I[2]	Accept the package settings from device or library.
-l sel -l no_sel	"Selective linking"[1] active. "Selective linking" inactive.
-s -s_off	"Enable program status"[1] active. "Enable program status" inactive.
-w no_warn -w all_warn[2] -w *n*_off -w *n*_on	"Suppress warnings"[1] active. Display all warnings. Warning class *n* active[1]. Warning class *n* inactive[1].
Further options	Internal options of the SIMOTION compiler.
[1] Meaning of the compiler option: see "Local compiler settings (Page 46)". [2] Only when the compiler is called from the command line, e.g. using scripting.	

Note

The compiler options can also be specified when the compiler is called from the command line, e.g. using scripting.

2.3.7 Know-how protection for ST source files

You can protect ST source files from access by unauthorized third parties. Protected ST source files can only be opened or exported as plain text files by entering a password.

For information about how to apply know-how protection, refer to the online help.

Note

If you export in XML format, the ST source files are exported in an encrypted form. When importing the encoded XML files, the know-how protection, including login and password, remains in place.

See also

Know-how protection for libraries (Page 229)

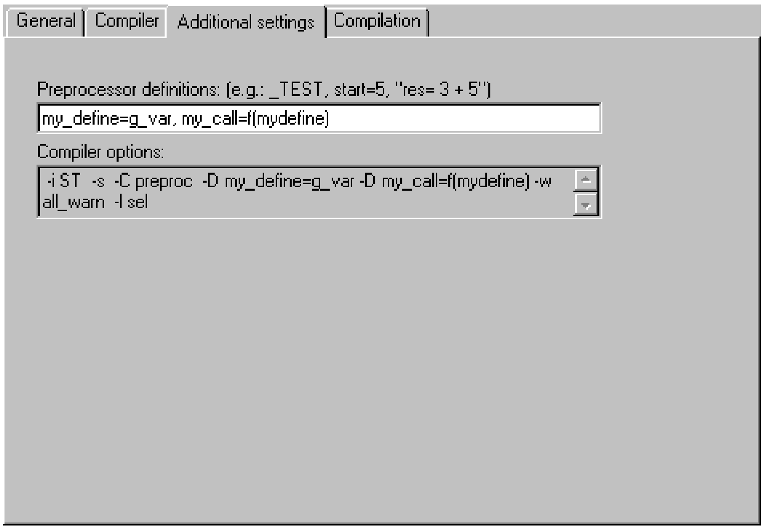
2.3.8 Making preprocessor definitions

You can make definitions for the preprocessor (see Control preprocessor (Page 243)) in the Properties dialog box of the ST source file. This enables you also to control the preprocessor with ST source files with know-how protection (see Know-how protection for ST sources (Page 51)).

Making preprocessor definitions in the Properties dialog box

1. Open the Properties window for the ST source file
   (see Changing the properties of an ST source (Page 23)):

   Select the ST source file in the project navigator and select the **Edit > Object properties** menu command.
2. Select the **Additional settings** tab.
3. Enter the preprocessor definitions (syntax as shown in the following table).
4. Confirm with **OK**.



Preprocessor definitions: (e.g.: _TEST, start=5, "res= 3 + 5")

my_define=g_var, my_call=f(mydefine)

Compiler options:

```
 -i ST  -s  -C preproc  -D my_define=g_var -D my_call=f(mydefine) -w
all_warn  -I sel
```

Figure 2-18 Preprocessor definitions

Table 2-8 Syntax of the preprocessor definitions

Syntax	Meaning
Identifier=text	The specified *identifier* is defined and replaced in the ST source file by the specified *text*. Permissible characters: See table footnote. If the expression contains blanks (e.g. in the text), the syntax *"Identifier=text"* must be used.
'Identifier=text'	
"Identifier=text"	
Identifier	The specified *identifier* is defined and replaced in the ST source file by blank text. Permissible characters: See table footnote.
Multiple preprocessor definitions are separated by commas: *Definition_1, Definition_2, …* Permissible characters: • For *identifier*: In accordance with the rules for identifiers: Series of letters (A … Z, a … z), digits (0 … 9) or single underscores (_) in any order, whereby the first character must be a letter or underscore. No distinction is made between upper and lower case letters. • For *text*: Sequence of any characters other than \ (backslash), ' (single quote) and " (double quote). The keywords USES, USELIB and USEPACKAGE are not permitted.	

Note

Preprocessor definitions made within an ST source file with pragmas, overwrite the definitions in the Properties dialog box.

Note the information for preprocessor statement (Page 244)!

2.3.9 Exporting, importing and printing an ST source file

An overview is provided here of the export, import and printing of an ST source file.

2.3.9.1 Exporting an ST source file as a text file (ASCII)

To export an ST source file as an ASCII file:

1. Open the ST source file (Page 23), if necessary entering the password (for ST source files with know-how protection (Page 51)).
2. Make sure that the cursor is in the ST editor.
3. Select the **ST source file > Export** menu command.
4. Enter the path and file name for the ASCII file and click **Save** to confirm.

The ST source file is saved as an ASCII file; the file name is given the default extension *.st.

Alternatively, you can also proceed as follows:

1. Select the ST source file in the project navigator.
2. Select **Export** from the context menu.
3. Only for ST source files with know-how protection (Page 51):

   If the user with the login assigned to the ST source file has not yet logged on:

   – Enter the corresponding password for the displayed login.

   You can now export or open additional ST source files to which the same login is assigned, without having to re-enter the password.
4. Enter the path and file name for the ASCII file and click **Save** to confirm.

2.3.9.2 Exporting an ST source file in XML format

Follow these steps to export an ST source file in XML format:

1. Select the ST source file in the project navigator.
2. Select the context menu **Expert > Save project and export object**.
3. Specify the path for the XML export, and confirm with **OK**.

An XML file with the ST source file name and a folder with additional associated XML files are saved in the specified path.

Note

Know-how-protected ST source files can also be exported in XML format. The ST source files are exported encrypted. When importing the encoded XML files, the know-how protection, including login and password, remains in place.

2.3.9.3 Importing a text file (ASCII) as an ST source file

To import an ASCII file as an ST source file:

1. Select the PROGRAMS folder under the appropriate SIMOTION device in the project navigator.
2. Select the menu Insert > External source > ST source file.
3. Select the ASCII file to be imported, and click Open to confirm.

   The dialog box for inserting an ST source file is displayed.
4. Enter the name of the ST source file and select the additional options (see Insert ST source file (Page 21)).

The ASCII file is incorporated into the current project directory as an ST source file and can be opened.

2.3.9.4 Importing XML data into ST source files

Follow these steps to import XML data into an ST source file:

1. If applicable, insert a new ST source file (see Insert ST source file (Page 21)).
2. Select the ST source file in the project navigator.
3. Select the context menu Expert > Import object.
4. Select the XML data to be imported.

The imported XML data overwrites existing data in the selected ST source file. The entire project is saved and recompiled.

Note

Note that if the XML data to be imported were exported from an ST source file that was know-how protected, the know-how protection, including login and password, remains in place while importing the encoded XML files.

2.3.9.5 Printing an ST source file

To print an ST source file:

1. Open the ST source file.
2. Make sure that the cursor is in the ST editor.
3. Select the menu Project > Print.

The program is printed with the name and date.

2.3.10 Using an external editor

What external editors can be used?

As an alternative to the default ST editor, you can use any other ASCII editor that supports the following function:

- External programs (for example, compiler) can be called and run on the active window.

In addition, the editor should be capable of highlighting certain text passages of the ST source file in color (syntax coloring).

Note

If you use an external editor, the dynamic ST source file menu and its entries are not available. The corresponding toolbar is also inactive.

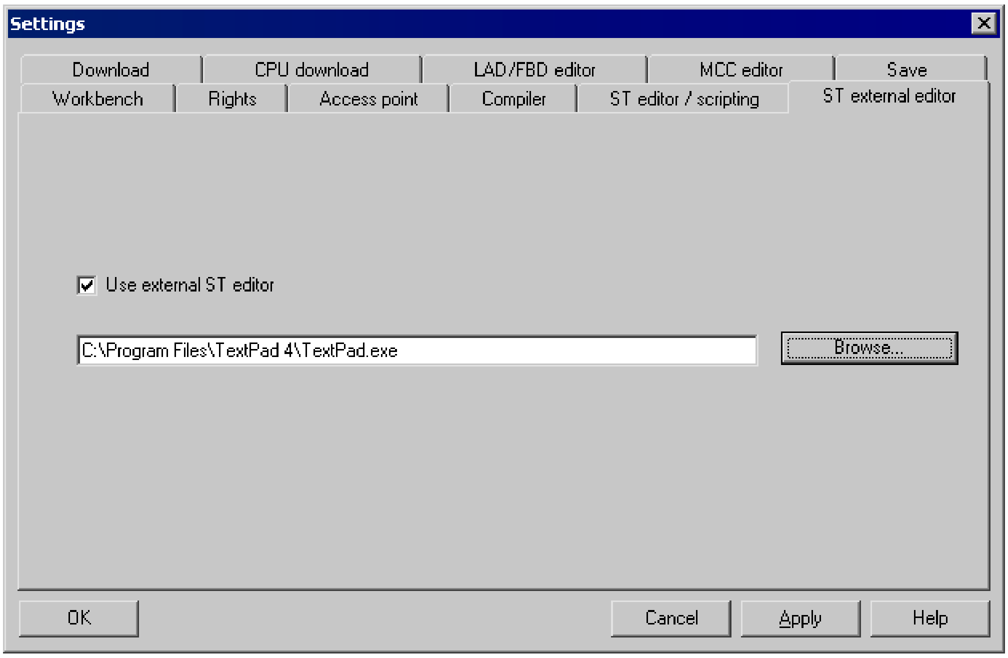
It must be possible to start compilation of the ST source file from the external editor.

Status Program continues with the ST editor.

Settings for the use of an external editor

The settings are made in the SCOUT workbench:

1. Select the menu **Tools > Settings**.
2. Select the **ST external editor** tab (see figure).
3. Activate the **Use external ST editor** checkbox.
4. Enter the path of the external editor:
   - Click **Browse...** and select the path and file name of the editor.



☑ Use external ST editor

C:\Program Files\TextPad 4\TextPad.exe

Figure 2-19 Settings for the use of an external editor

Making settings in the external editor

The following notes are of a general nature. Compare the operator instructions of the external editor.

1. Configure the path to the ST compiler in the external editor. The compiler is located in the STEP7 installation directory s7bin\u7wstcax.exe.
2. Syntax files are supplied for various editors. These enable the editor to highlight text passages in color (syntax coloring). Copy the syntax file to the relevant directory and configure the editor accordingly.

Note the following when using an external editor:

⚠ CAUTION

Close all windows of the external editor before you close a project or exit SIMOTION SCOUT. Failure to do so could result in loss of data!

2.3.11 ST source file menus

2.3.11.1 ST source file menu

Depending on the active application/editor or the mode (ONLINE/OFFLINE), certain commands are not displayed or cannot be selected. The menu is only displayed if the ST editor is active in the working area.

You can select the following functions:

Table 2-9 ST Source File Menu

Function	Meaning/Note
Close	Select this command to close the active ST source file. In the event of changes, you can decide whether you want to transfer the changed source file to the project.
Characteristics	Select this command to display the properties of the active ST source file. Several tabs are provided to make local settings for this source. See: Changing the properties of an ST source file (Page 23).
Accept and compile	Choose this command to transfer the current ST source file to the project and compile into executable code. See: Starting the compiler (Page 44).
Use preprocessor	As an option, the preprocessor scans an ST source file before compiling and can, for example, replace character strings in the file, which will then be taken into account during the compilation. You can specifically execute the preprocessor statements with this menu command.
Export	Select this command to export the active ST source file as text file (ASCII). See: Exporting an ST source file as a text file (ASCII) (Page 53).
Program status On/Off	**Program status On/Off** monitors the current status of the active ST source file. During the status, the program variable values can be monitored during a program run. The ST editor is divided into two windows. The right pane displays the current values of the program variables selected in the ST source file (left pane). **The project and the program must be available in the target system and an ONLINE connection to the target system must be active.**

2.3.11.2 ST source file context menu

Depending on the active application/editor or the mode (ONLINE/OFFLINE), certain commands are not displayed or cannot be selected.

You can select the following functions:

Table 2-10 ST source file context menu

Function	Meaning/Note
Close	**Close** closes the active ST source file.
Cut	Select **Cut** to remove the selected object and save it to the clipboard.
Copy	Select **Copy** to copy the selected object. It is stored in the clipboard.
Inserting	Select **Paste** to insert the contents of the clipboard at the current cursor position.
Deleting	Use **Delete** to delete the current St source file. All data from the ST source file is permanently deleted.
Rename	Use **Rename** to rename the current ST source file. Please note that with name changes, it is not possible to change the referencing to this name and that the new name must comply with the ST conventions
Save variables	You can save retain, unit and global variables with this menu command. You can save these variables from the RAM/ROM memory of the target device and store them on a data medium as XML file. When these variables are restored, they can be written from the data medium to the RAM/ROM memory of the target device.
Restore variables	You can restore retain, unit and global variables from the previously exported variables with this menu command. When these variables are restored, they can be written from the data medium to the RAM/ROM memory of the target device.
Expert	
Import object	**Import object** imports the data of an ST source file from another project which was previously created with a selective XML export.
Save project and export object	Use **Save project and export object** to export selected data of the ST source file in XML format. This data export can then be reimported into other projects.
Accept and compile	Use **Accept and compile** to save and compile the selected ST source file.
Run preprocessor	As an option, the preprocessor scans an ST source file before compiling and can, for example, replace character strings in the file, which will then be taken into account during the compilation. You can specifically execute the preprocessor statements with this menu command.
Program status On/Off	**Program status On/Off** monitors the current status of the active ST source file. During the status, the program variable values can be monitored during a program run. The ST editor is divided into two windows. The right pane displays the current values of the program variables selected in the ST source file (left pane). The project and the program must be available in the target system and an ONLINE connection to the target system must be active.
Export	**Export** exports the active ST source file as a file in ST format, e.g. to import the program to other projects.
Know-how Protection	
Set	Use **Set know-how protection** to set know-how protection for ST sources files. The protected sources can only be opened and modified with the specified log-on and password.
Deleting	**Delete know-how protection** releases the protected ST source files so that they can be opened and read without entering a password.
Reference data	

Function	Meaning/Note
Create	Select **Create reference data** to create the reference data of the selected ST source file. The reference data contains information about the designators used, with details of their declaration, application, function calls and the nesting of these.
Display	
Cross references	**Display cross references** displays the cross reference list of the selected ST source file in the detail view. The reference data must be created before the cross references can be displayed.
Program structure	Select **Display program structure** to display the program structure of the selected ST source file in the detail view. Before the program structure can be displayed, the reference data must first be created.
Print	Use **Print** to print the selected ST source file.
Print preview	Select **Print preview** to preview the page to be printed.
Characteristics	Use **Properties** to display the properties of the active ST source file. This window displays the name, change date and the storage location.

2.4 Creating a sample program

In this section, we create a short program to illustrate the steps involved, including starting and testing. Testing is described in Program test (Page 252).

Function

The *Flash* program sets a bit in an output byte of your target system and rotates it within this byte. This causes each bit of the output byte to be set and reset in succession. After the last bit of the byte, the first bit is to be set again. You can observe the result of the program at the outputs of your target system.

2.4.1 Requirements

To create the sample program, you need

- A SIMOTION project and
- A SIMOTION device (e.g. SIMOTION C240) within the project whose output is configured at address 62.

2.4.2 Opening or creating a project

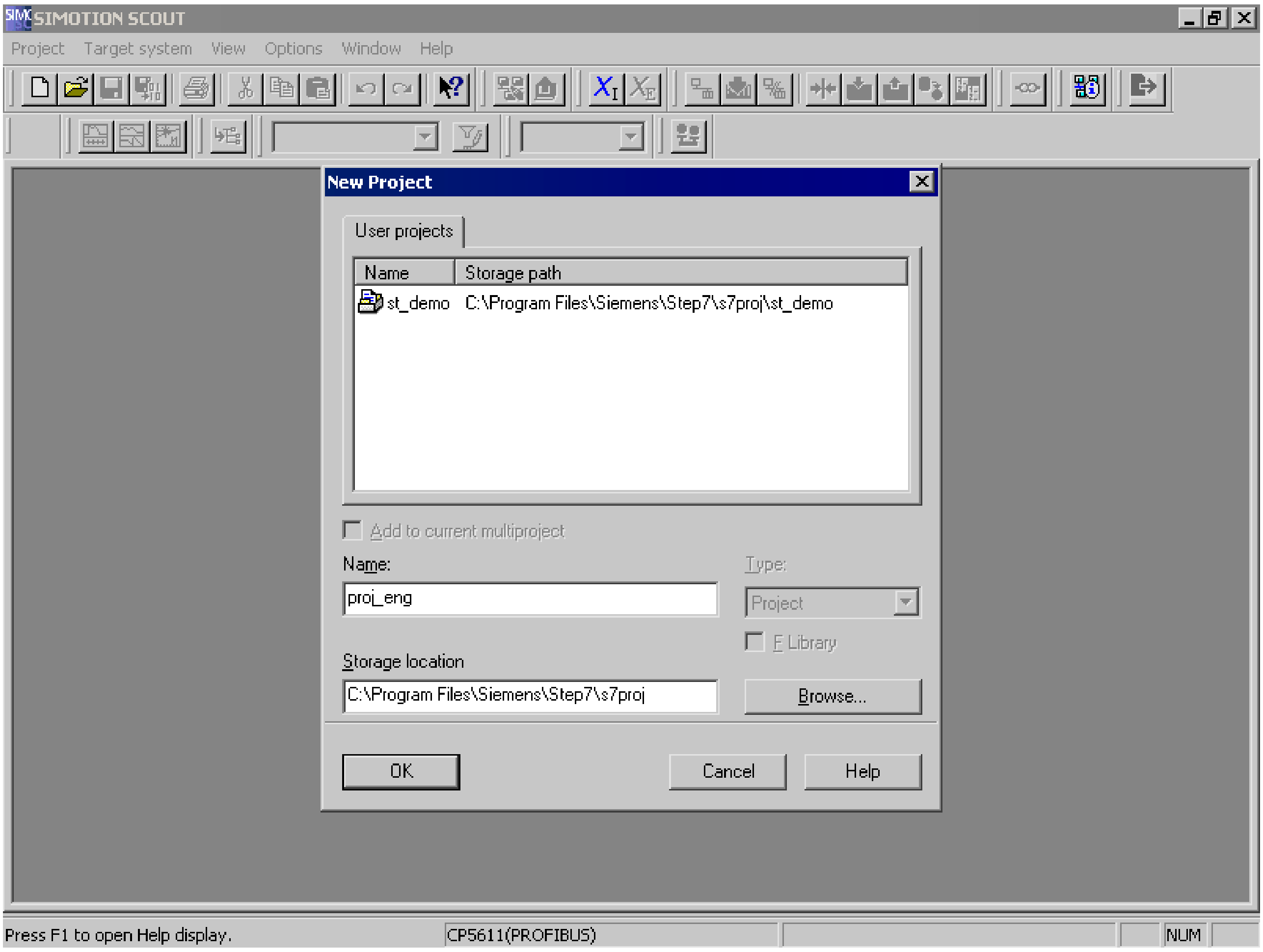
Projects contain all the information about the hardware and configuration. This includes the programs you use to control the hardware.

Proceed as follows

If a project does not yet exist, proceed as follows:

1. Select **Project** in the menu bar.
2. Select **New** or **Open**.
3. Specify a name for a new project, and click **OK** to confirm.

For details, see the online help.



Name	Storage path
st_demo	C:\Program Files\Siemens\Step7\s7proj\st_demo

□ Add to current multiproject

Name: proj_eng

Type: Project

□ F Library

Storage location: C:\Program Files\Siemens\Step7\s7proj

Press F1 to open Help display.

CP5611(PROFIBUS)

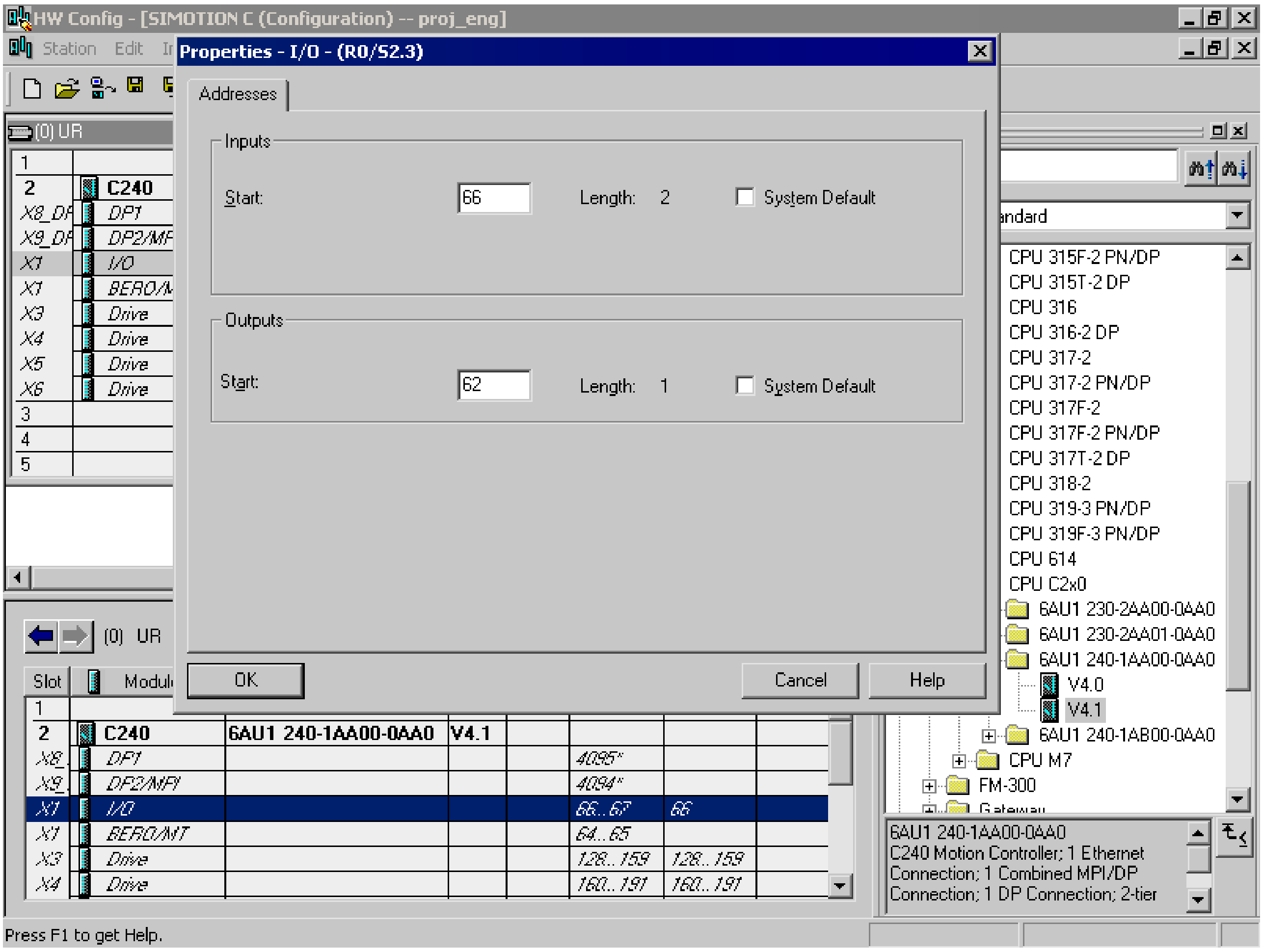
Figure 2-20 Creating a new project

2.4.3 Making the hardware known

The steps are as follows:

1. Create and configure a new SIMOTION device (e.g. C240 V4.1).
2. Configure an output in HW Config at Address 62.

For more details on steps 1 and 2, refer to the online help.



HW Config - [SIMOTION C (Configuration) -- proj_eng]

Properties - I/O - (R0/S2.3)

Addresses

Inputs

Start: 66 Length: 2 □ System Default

Outputs

Start: 62 Length: 1 □ System Default

OK Cancel Help

(0) UR

Slot	Module					
1						
2	C240	6AU1 240-1AA00-0AA0	V4.1			
X8	DP1				4095*	
X9	DP2/MPI				4094*	
X1	I/O				66..67	66
X1	BERO/NT				64..65	
X3	Drive				128..159	128..159
X4	Drive				160..191	160..191

6AU1 240-1AA00-0AA0
C240 Motion Controller; 1 Ethernet Connection; 1 Combined MPI/DP Connection; 1 DP Connection; 2-tier

Press F1 to get Help.

Figure 2-21 Change in HW Config

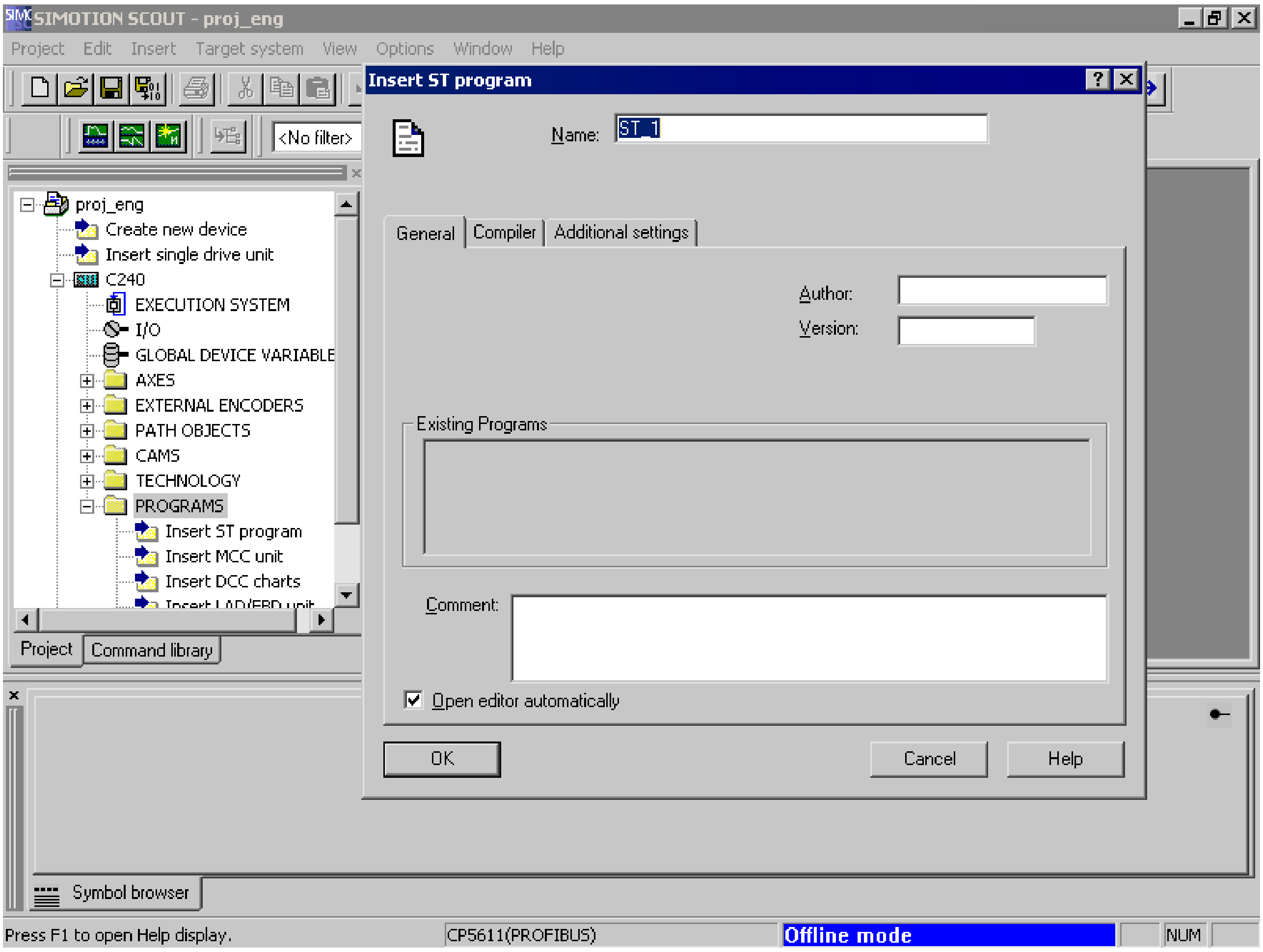
2.4.4 Entering source text with the ST editor

Proceed as follows

1. In the project navigator, open the tree for your SIMOTION device (programs are assigned to the SIMOTION device on which they are to run).
2. Select the **PROGRAMS** folder and choose **Insert > Program > ST source file**.
3. Enter a name for the ST source file consisting of up to 128 characters (see figure), e.g. **ST_1**, and click **OK** to confirm the entries.

   The ST editor appears in the working area. The ST source file **ST_1** is inserted in the navigator.
4. Enter the source text from Source text of the sample program (Page 64), preferably with indented lines. To do this, press the TAB key.

   The features of the ST editor are described in Working with the ST editor (Page 25); the structure of an ST source file is described in detail in Structure of the ST source file (Page 86) and in Source file sections (Page 169).
5. Use comments as often as possible. Enter your comment after the **//** characters if the comment fits on one line of text. If the comment extends across several lines, insert it between character pairs (***** and *****).
6. Save the complete project with **Project > Save**.



Name: ST_1

Author:

Version:

Comment:

☑ Open editor automatically

Press F1 to open Help display.

Figure 2-22 Naming the ST source file

2.4.4.1 Functions of the editor

In addition to simple text input, the ST editor provides the following advanced/convenience functions for documenting the functionality of your source text:

- Standard Windows user features (for example, Undo with Ctrl+Z or Redo with Ctrl+Y)
- Syntax coloring (different colors for different language elements)
- Source file printout in an appropriate layout with page number, source file name and printing date
- Export/import of the source file
- Source file archiving (via the project)

A detailed description of the functions is contained in Working with the ST editor (Page 25) and in Making settings for the compiler (Page 44).

2.4.4.2 Source text of the sample program

The table shows the source code of the sample program. You need to enter it in the same way to create executable code.

Table 2-11 Flash sample program

```
INTERFACE
    VAR_GLOBAL
        counterVar : INT  := 1; // counter variable
        outputVar  : BYTE := 1; // auxiliary tag
    END_VAR
    PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
    PROGRAM Flash
        IF counterVar >= 500 THEN // in every 500th pass
            %QB62 := outputVar;   // set output byte
            outputVar := ROL (in := outputVar, n := 1);
            (* // rotate bit in byte
              one digit to the left*)
            counterVar := 0; // reset counter
        END_IF;
        counterVar := counterVar + 1; // increment counter
    END_PROGRAM
END_IMPLEMENTATION
```

2.4.5 Compiling a sample program

Before you can run or test your program, you must compile it into executable machine code. The compiler performs this task.

2.4.5.1 Starting the compiler

Before you can run or test your program, you must compile it into executable machine code. The ST compiler performs this task.

Start the compiler as follows:

1. Click in the window with the ST editor to display the **ST source file** menu. This menu is a dynamic menu and is only displayed if the window of the ST editor is active.
2. Start the compiler by selecting the **ST source file > Accept and compile** menu command.

2.4.5.2 Correcting errors

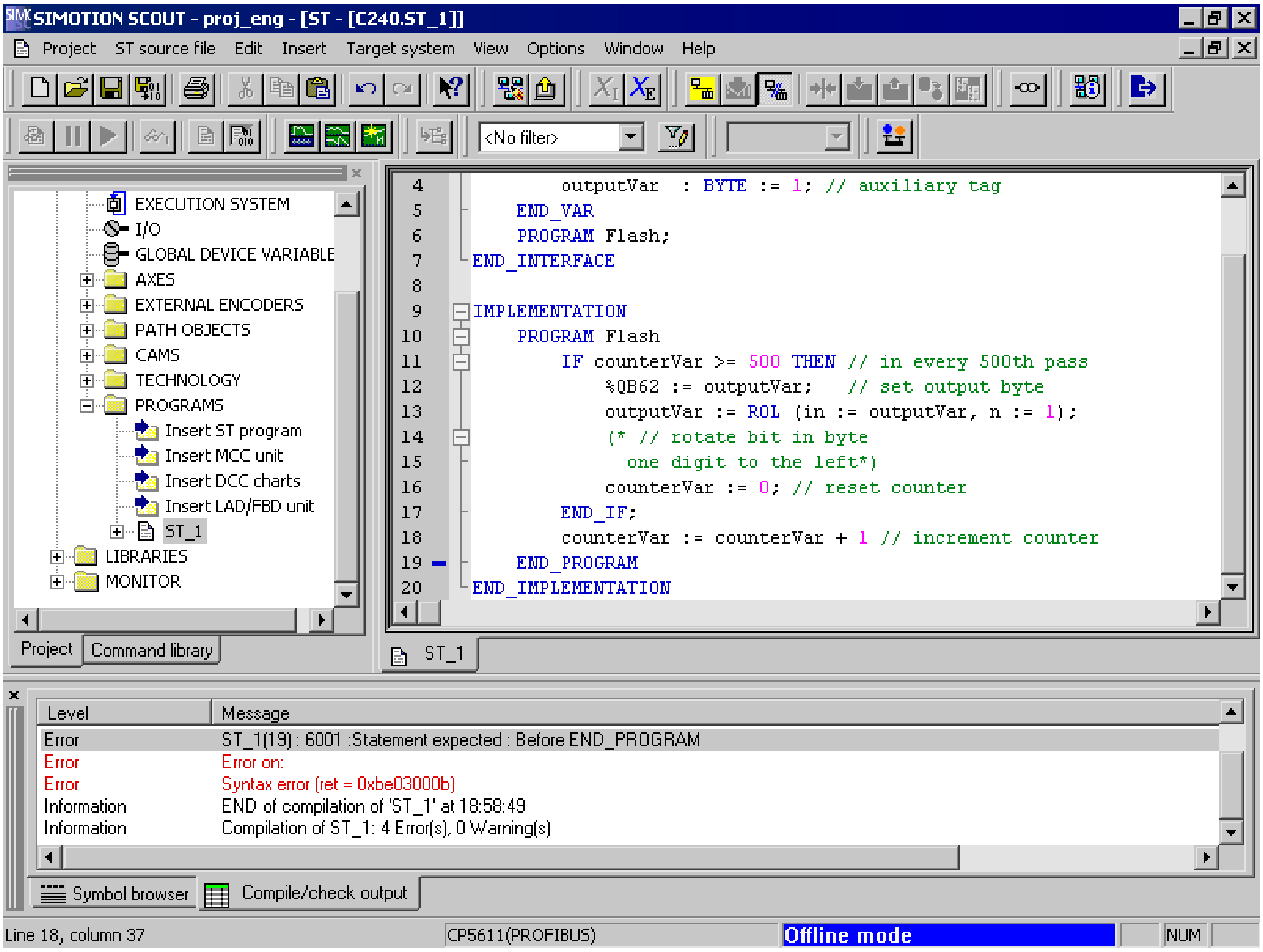
The compiler checks the syntax of the ST source file. The **Compile/check output** tab of the detail view displays the successful compilation of the source text or compiler errors. The error details include: Name of the ST source file, the line number where the error occurred, the error number and an error description.

Proceed as follows to correct an error in the sample program:

1. Double-click the error message. The cursor is placed at the relevant line in the ST source file. See Example for error messages (Page 65).
2. Start debugging the first error.
3. Start the compilation operation again.
4. Repeat the entire operation until no more errors are displayed (**0 errors**).

After a successful compilation, you will have created an application program with the name **flash**. This program is displayed in the project navigator below the **ST_1** program source file.

2.4.5.3 Example of error messages



```
          outputVar   : BYTE := 1; // auxiliary tag
      END_VAR
      PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
      PROGRAM Flash
          IF counterVar >= 500 THEN // in every 500th pass
              %QB62 := outputVar;   // set output byte
              outputVar := ROL (in := outputVar, n := 1);
              (* // rotate bit in byte
                one digit to the left*)
              counterVar := 0; // reset counter
          END_IF;
          counterVar := counterVar + 1 // increment counter
      END_PROGRAM
END_IMPLEMENTATION
```

```
Level          Message
Error          ST_1(19) : 6001 :Statement expected : Before END_PROGRAM
Error          Error on:
Error          Syntax error (ret = 0xbe03000b)
Information    END of compilation of 'ST_1' at 18:58:49
Information    Compilation of ST_1: 4 Error(s), 0 Warning(s)
```

Figure 2-23 Error messages during ST source file compilation

The figure shows an example of compiling the ST source file ST_1 (see Source text of the sample program (Page 64), in which the following change has been made: The semicolon is missing in the statement "counterVar := counterVar + 1" at the end of line 18.

The compiler does not detect the error until Line19, because it continues with the compilation after the missing semicolon.

Once the missing semicolon is added, the ST source file is compiled without errors.

A detailed list of all compiler error messages can be found in Compiler error messages and their correction (Page 350).

2.4.6 Running the sample program

Before you can run the program, you must assign it to an execution level or task. When you have done this, you can establish the connection to the target system, download the program to the target system and then start it.

2.4.6.1 Assigning a sample program to an execution level

The execution levels specify the order in which the programs run. Each execution level contains one or more tasks to which you can assign programs.

The assignment of a program to a task can only be performed after compilation and before the program is loaded onto the target system.

Assign the sample program to the *BackgroundTask*. The *BackgroundTask* is provided for the programming of cyclic sequences without a fixed time frame. It is executed cyclically in the round robin execution level, which means it will be automatically restarted on completion.

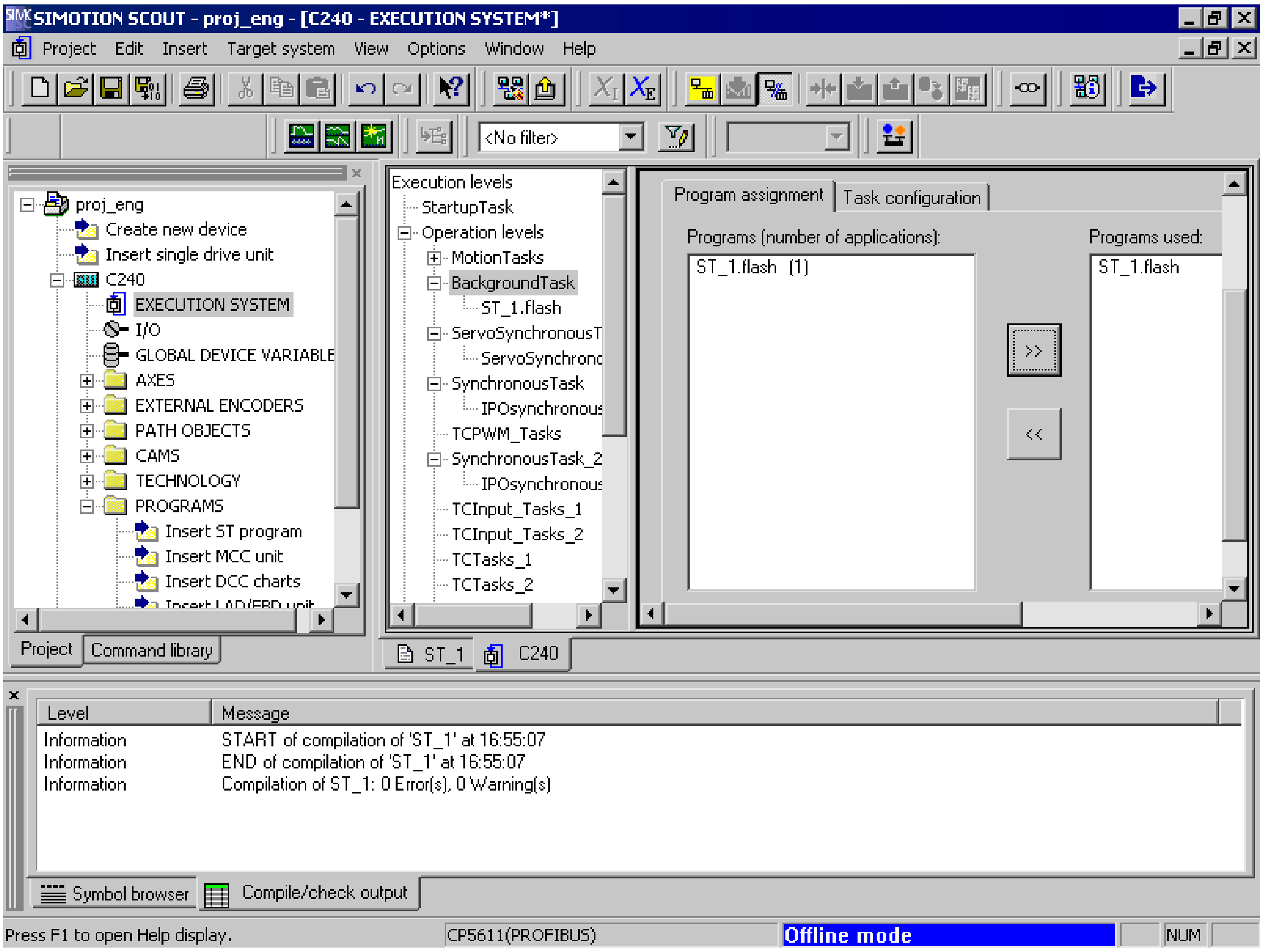
How to assign the sample program to the *BackgroundTask*:

1. When you double-click the **Execution system** element in the project navigator, the window containing the execution system and the program assignment appears in the working area.
2. Click **BackgroundTask** to select it for the program assignment.

   The program assignment on the left side of the window shows you all the compiled programs that can be assigned to tasks.
3. In the **Programs** list, click sample program **ST_1.flash**. Then, click the **>>** button to assign the program to the BackgroundTask.

   The result is shown in the following figure. The program **ST_1.flash** is displayed in the **Programs used** list box.

For more information on the execution system and assignment of programs to tasks, see *SIMOTION Motion Control Basic Functions* Function Description.



Level	Message
Information	START of compilation of 'ST_1' at 16:55:07
Information	END of compilation of 'ST_1' at 16:55:07
Information	Compilation of ST_1: 0 Error(s), 0 Warning(s)

Press F1 to open Help display.

Figure 2-24 Assigning the sample program to the BackgroundTask

2.4.6.2 Establishing a connection to the target system

Before a connection to the target system can be set up, the PC interface card must be configured and connected to the target system.

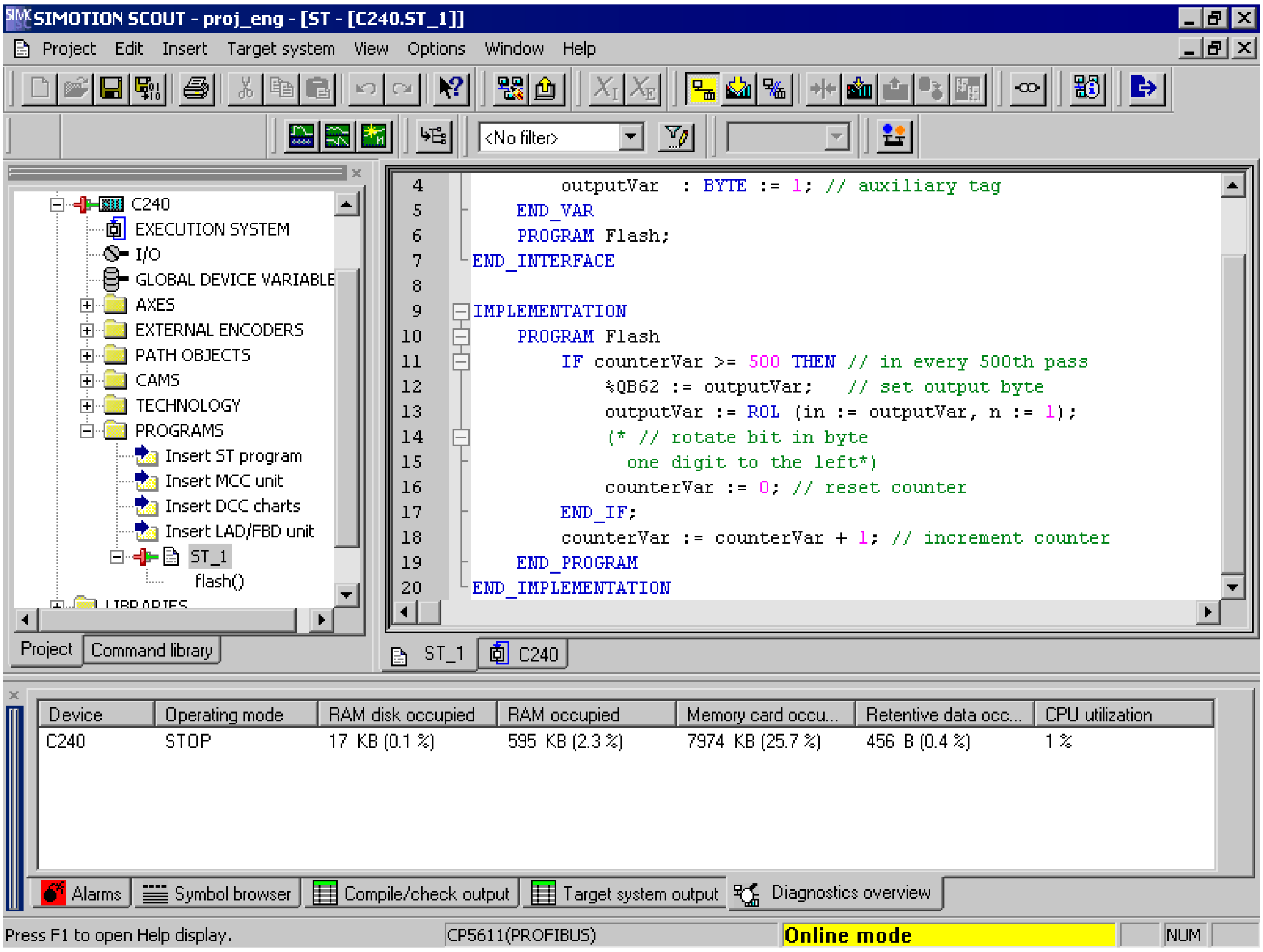
Proceed as follows to connect to the target system:

1. Select the **Project > Connect to target system** menu command.

   The **Diagnostics overview** tab is opened in the detail view. The diagnostics overview shows you the operating state, memory allocation and CPU utilization for the device you are connected to. You can see at the lower right edge of the screen that you are connected to the target system.

   Note

   For more detailed information, refer to the SIMOTION SCOUT Configuring Manual and SIMOTION SCOUT online help.



```
      outputVar  : BYTE := 1; // auxiliary tag
    END_VAR
    PROGRAM Flash;
  END_INTERFACE

  IMPLEMENTATION
    PROGRAM Flash
      IF counterVar >= 500 THEN // in every 500th pass
        %QB62 := outputVar;   // set output byte
        outputVar := ROL (in := outputVar, n := 1);
        (* // rotate bit in byte
          one digit to the left*)
        counterVar := 0; // reset counter
      END_IF;
      counterVar := counterVar + 1; // increment counter
    END_PROGRAM
  END_IMPLEMENTATION
```

Device	Operating mode	RAM disk occupied	RAM occupied	Memory card occu...	Retentive data occ...	CPU utilization
C240	STOP	17 KB (0.1 %)	595 KB (2.3 %)	7974 KB (25.7 %)	456 B (0.4 %)	1 %

Press F1 to open Help display. CP5611(PROFIBUS) **Online mode** NUM

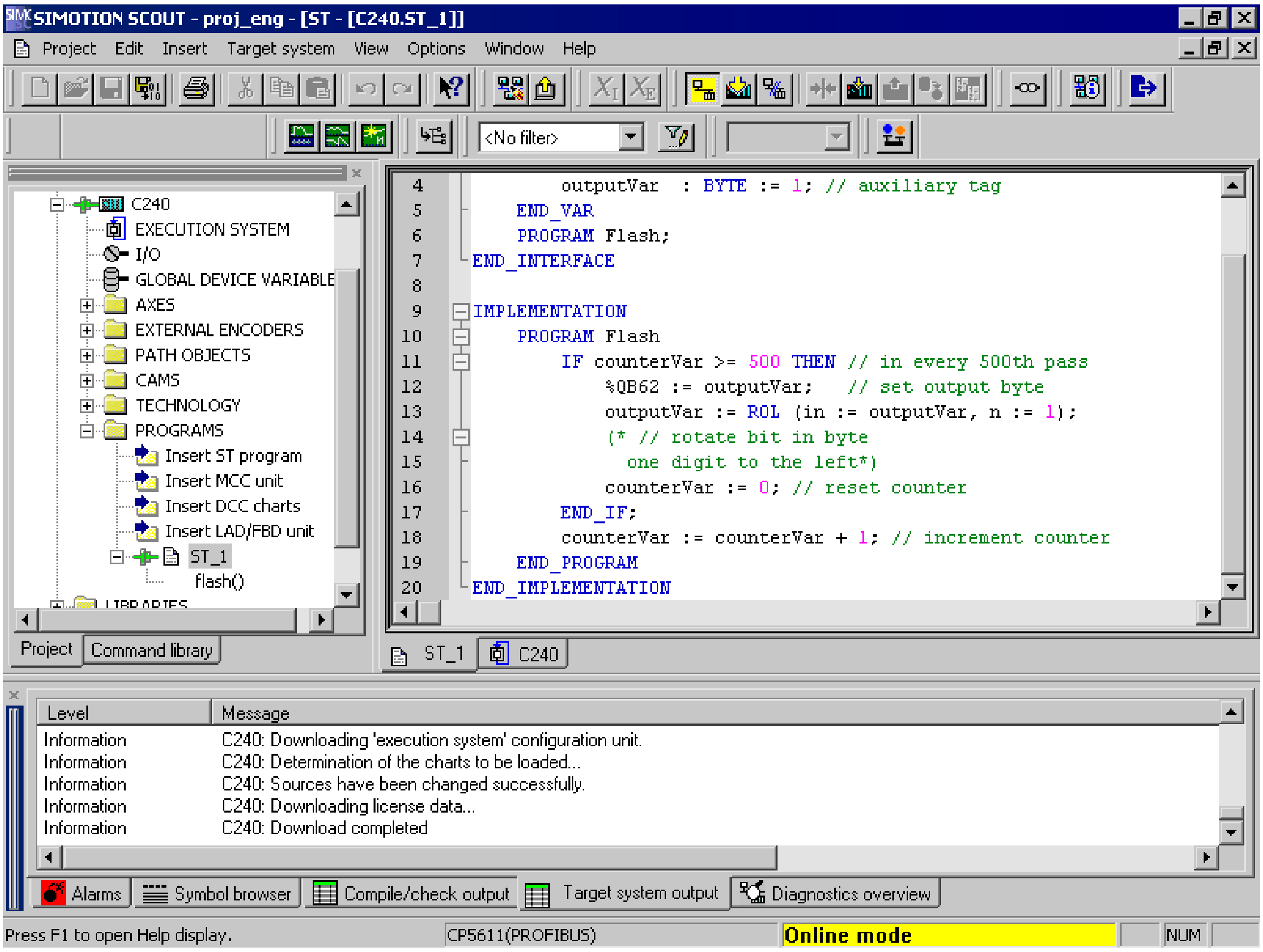
Figure 2-25 Establishing a connection to the target system

2.4.6.3 Downloading the sample program to the target system

Proceed as follows to download the sample program to the target system:

1. Switch the target system to **STOP**.
2. Select the **Target system > Download > Project to target system** menu command.
3. Confirm all further queries.

The Target system output window in the detail view opens and displays the result of the download.



```
        outputVar   : BYTE := 1; // auxiliary tag
    END_VAR
    PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
    PROGRAM Flash
        IF counterVar >= 500 THEN // in every 500th pass
            %QB62 := outputVar;   // set output byte
            outputVar := ROL (in := outputVar, n := 1);
            (* // rotate bit in byte
              one digit to the left*)
            counterVar := 0; // reset counter
        END_IF;
        counterVar := counterVar + 1; // increment counter
    END_PROGRAM
END_IMPLEMENTATION
```

Level	Message
Information	C240: Downloading 'execution system' configuration unit.
Information	C240: Determination of the charts to be loaded...
Information	C240: Sources have been changed successfully.
Information	C240: Downloading license data...
Information	C240: Download completed

Figure 2-26 Downloading the sample program to the target system

2.4.6.4 Starting and testing the sample program

Starting sample program

Proceed as follows to start the sample program:

- Switch your target system to RUN (see hardware description).

The lamps flash in sequence at the outputs of your target system.

Testing a sample program

See Program test (Page 252).

ST Fundamentals 3

This section describes the language resources available in ST and how to use them. Please note that functions, function blocks and the task control system are described in the following chapters. For a complete formal language description containing all the syntax diagrams, see Appendix Rules (Page 307).

3.1 Language description resources

Syntax diagrams are used as a basis for the language description in the following sections of the manual. They provide you with an invaluable insight into the syntactic (i.e. grammatical) structure of ST.

3.1.1 Syntax diagram

The syntax diagram is a graphical representation of the language structure. The structure is described by a sequence of rules. A rule can build on existing rules.

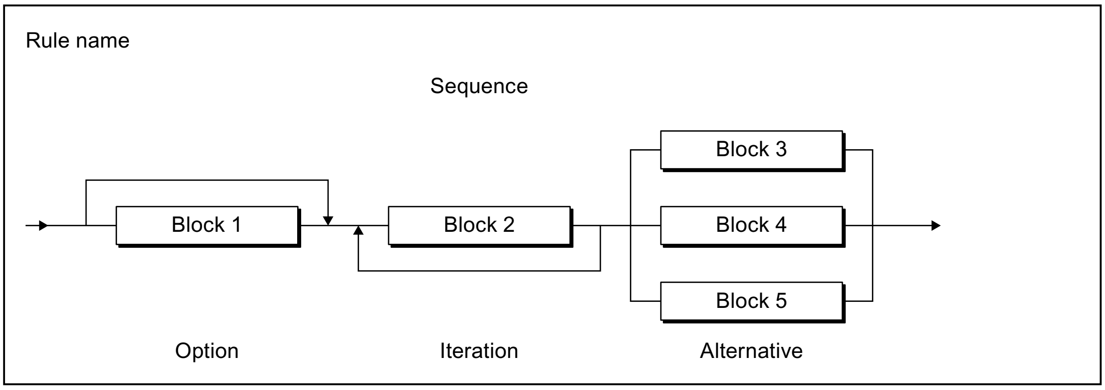


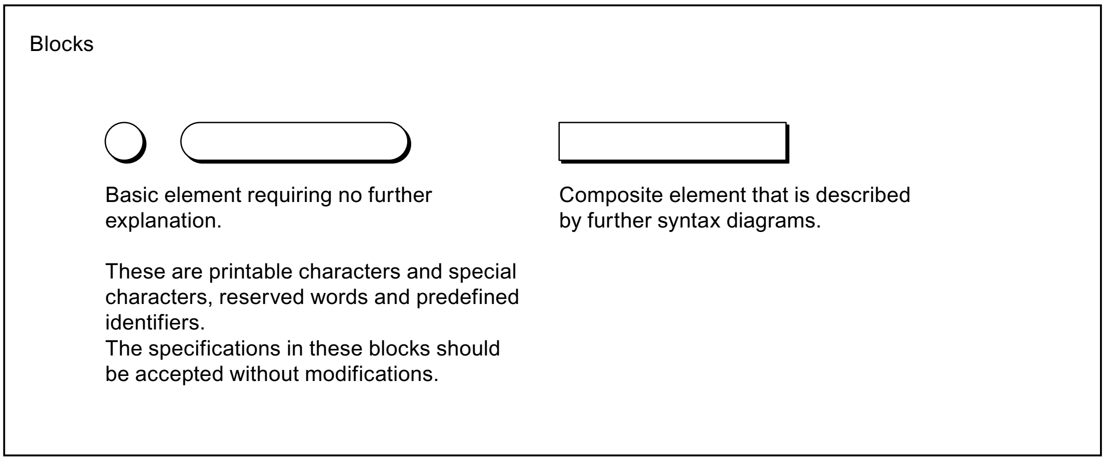
Figure 3-1 Syntax diagram

The syntax diagram in the previous figure is read from left to right. The following rule structures must be observed:

- Sequence: Sequence of blocks
- Option: Statement(s) that can be skipped
- Iteration: Repetition of one or more statements
- Alternative: Branch

3.1.2 Blocks in syntax diagrams

A block is a basic element or an element that is itself composed of blocks. The figure shows the symbol types used to represent the blocks:



Basic element requiring no further explanation.

These are printable characters and special characters, reserved words and predefined identifiers.
The specifications in these blocks should be accepted without modifications.

Composite element that is described by further syntax diagrams.

Figure 3-2 Blocks

Formatted and unformatted rules must be observed when entering source text, i.e. when converting the blocks or elements of a syntax diagram into source text (see Help for the language description (Page 291)).

See also

Formal Language Description (Page 291)

3.1.3 Meaning of the rules (semantics)

The rules can only represent the formal structure of the language. The meaning (i.e. semantics) is not always apparent. For this reason, additional information is written beside the rules if the meaning is critical. Examples are:

- Where elements of the same kind have a different meaning, an additional name is appended. For example, an addition is specified in the *date* rule for every *decimal digit string* element - either *year, month* or *day* (see Literals (Page 308)). The name indicates the usage.
- Important restrictions are noted next to the rules. For example, in the *integer* rule for - (minus), it is noted that the minus can appear only in front of decimal digit strings of data types SINT, INT, and DINT (see Literals (Page 308)).

See also

Formal Language Description (Page 291)

3.2 Basic elements of the language

The basic elements of the ST language include the ST character set, reserved identifiers constructed from the ST character set (e.g. language commands), self-defined identifiers and numbers.

The ST character set and the reserved identifiers are basic elements (terminals) as they are described verbally and not by another rule. Self-defined identifiers and numbers are not terminals as they are described by other rules.

In the syntax diagrams, terminals are represented by circles or oval symbols, while composite elements are represented by rectangles (see Blocks in syntax diagrams (Page 72)). Below is a selection of the main terminals; for a complete overview, refer to Basic elements (terminals) (Page 294).

3.2.1 ST character set

ST uses the following **letters and digits** from the ASCII character set:

- The lower and upper case letters from A to Z
- The Arabic digits from 0 to 9

Letters and digits are the most commonly used characters. For example, identifiers (see Identifiers in ST (Page 73)) consist of a combination of letters, digits and the underscore. The underscore is one of the special characters.

Special characters have a fixed meaning in ST (see Formal Language Description (Page 291), Basic elements (terminals) (Page 294)).

3.2.2 Identifiers in ST

Identifiers are names in ST. These names can be defined by the system, such as language commands. However, the names can also be user-defined, for example, for a constant, variable or function.

3.2.2.1 Rules for identifiers

Identifiers are made up of letters (A … Z, a … z), numbers (0 … 9) or single underscores (_) in any order, whereby the first character must be a letter or underscore. No distinction is made between upper and lower case letters (e.g. Anna and AnNa are considered to be identical by the compiler).

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

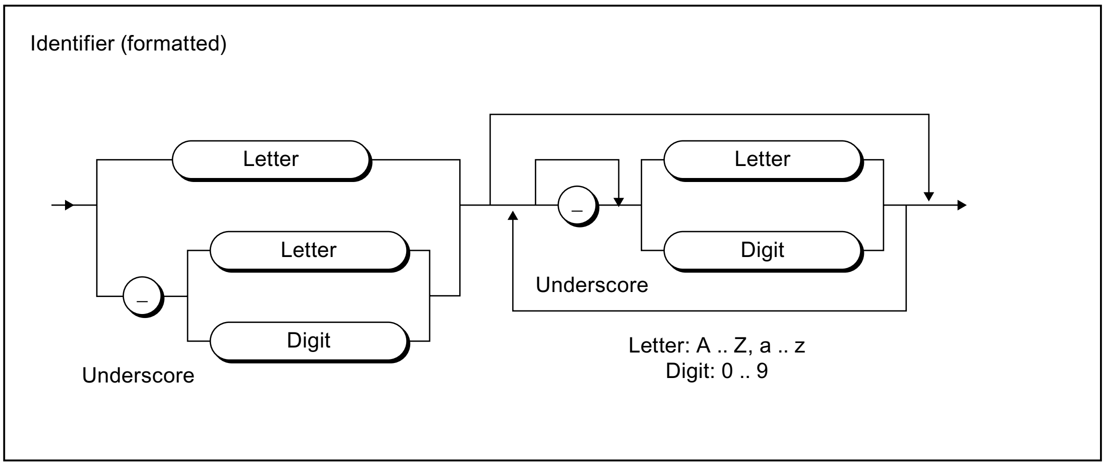


Figure 3-3 Syntax: Identifier

When assigning a name, it is best to choose a unique, meaningful name that contributes to the clarity of the program.

The syntax diagram in the figure says that the first character of an identifier must be a letter or underscore. An underscore must be followed by a letter or number, i.e. more than one underscore in succession is not allowed. This can be followed by any number or sequence of underscores, letters or numbers. The only exception here again is that two underscores may not appear together.

3.2.2.2 Examples of identifiers

Examples of valid identifiers

The following names are valid identifiers:

```
x               y12             _sum            temperature     R_CONTROLLER3
name            area            myFB            table
```

Examples of invalid identifiers

The following names are **not** valid identifiers:

Invalid identifier	Reason
`4ter`	The first character must be a letter or underscore.
`*#AB`	Special characters (except underscores) are not permitted.
`RR__20`	Two underscores in succession are not permitted.
`S value`	Blank spaces are not permitted as they are special characters.
`Array`	While ARRAY is formally a valid identifier, it is a reserved identifier, i.e. it may only be used as predefined. This means you cannot use this name for your own purposes, for example, for a variable.

Identifiers that may not be used

Never define identifiers that:

- Are identical to a **reserved identifier**

  For more information, see Reserved identifiers (Page 75).
- Match a **task name**

  For a more detailed explanation, refer to the *SIMOTION Basic Functions* Function Manual.

Note

If possible, avoid defining identifiers that begin with _ (underscore), **struct**, **enum** or **command**.

While these are valid identifiers, their use can cause errors later when you download (additional) technology packages. Command words, parameters or data types in the basic system and in technology packages begin with these characters.

3.2.3 Reserved identifiers

Reserved identifiers may only be used as predefined. You may not declare a variable or data type with the name of a reserved identifier.

There is no distinction between upper and lower case notation.

A list of all identifiers with a predefined meaning can be found in the SIMOTION Basic Functions Function Manual:

- For more information on protected and reserved identifiers in the ST programming language,
  see also "Protected identifiers (Page 76)" and "Further reserved identifiers (Page 81)"
- For general standard functions and the data types defined in these functions,
  see also "Error Sources and Program Test (Page 251)"
- General system function blocks
- System functions, system variables and data types of SIMOTION devices
  (see also list manuals of the SIMOTION devices)
- System functions, system variables and data types of technology objects
  (see also list manuals of the technology packages)

3.2.3.1 Protected identifiers

The protected identifiers of the ST language are listed in the table.

For a brief explanation of all reserved words, please refer to Appendix Reserved Words (Page 299), and Syntax diagrams (Page 71) in Appendix Rules (Page 307).

Table 3-1 Protected identifiers in ST programming language

A	
ABS	ANYTYPE_TO_LITTLEBYTEARRAY
ACOS	ARRAY
AND	AS
ANYOBJECT	ASIN
ANYOBJECT_TO_OBJECT	AT
ANYTYPE_TO_BIGBYTEARRAY	ATAN
B	
BIGBYTEARRAY_TO_ANYTYPE	BY
BOOL	BYTE
BOOL_TO_BYTE	BYTE_TO_BOOL
BOOL_TO_DWORD	BYTE_TO_DINT
BOOL_TO_WORD	BYTE_TO_DWORD
BOOL_VALUE_TO_DINT	BYTE_TO_INT
BOOL_VALUE_TO_INT	BYTE_TO_SINT
BOOL_VALUE_TO_LREAL	BYTE_TO_UDINT
BOOL_VALUE_TO_REAL	BYTE_TO_UINT
BOOL_VALUE_TO_SINT	BYTE_TO_USINT
BOOL_VALUE_TO_UDINT	BYTE_TO_WORD
BOOL_VALUE_TO_UINT	BYTE_VALUE_TO_LREAL
BOOL_VALUE_TO_USINT	BYTE_VALUE_TO_REAL
C	
CASE	CTD_UDINT
CONCAT	CTU
CONCAT_DATE_TOD	CTU_DINT
CONSTANT	CTU_UDINT
COS	CTUD
CTD	CTUD_DINT
CTD_DINT	CTUD_UDINT

D	
DATE	DO
DATE_AND_TIME	DT
DATE_AND_TIME_TO_DATE	DT_TO_DATE
DATE_AND_TIME_TO_TIME_OF_DAY	DT_TO_TOD
DELETE	DWORD
DINT	DWORD_TO_BOOL
DINT_TO_BYTE	DWORD_TO_BYTE
DINT_TO_DWORD	DWORD_TO_DINT
DINT_TO_INT	DWORD_TO_INT
DINT_TO_LREAL	DWORD_TO_REAL
DINT_TO_REAL	DWORD_TO_SINT
DINT_TO_SINT	DWORD_TO_UDINT
DINT_TO_STRING	DWORD_TO_UINT
DINT_TO_UDINT	DWORD_TO_USINT
DINT_TO_UINT	DWORD_TO_WORD
DINT_TO_USINT	DWORD_VALUE_TO_LREAL
DINT_TO_WORD	DWORD_VALUE_TO_REAL
DINT_VALUE_TO_BOOL	
E	
ELSE	END_REPEAT
ELSIF	END_STRUCT
END_CASE	END_TYPE
END_EXPRESSION	END_VAR
END_FOR	END_WAITFORCONDITION
END_FUNCTION	END_WHILE
END_FUNCTION_BLOCK	ENUM_TO_DINT
END_IF	EXIT
END_IMPLEMENTATION	EXP
END_INTERFACE	EXPD
END_LABEL	EXPRESSION
END_PROGRAM	EXPT
F	
F_TRIG	FOR
FALSE	FUNCTION
FIND	FUNCTION_BLOCK
G	
GOTO	

I	
IF IMPLEMENTATION INSERT INT INT_TO_BYTE INT_TO_DINT INT_TO_DWORD INT_TO_LREAL INT_TO_REAL	INT_TO_SINT INT_TO_TIME INT_TO_UDINT INT_TO_UINT INT_TO_USINT INT_TO_WORD INT_VALUE_TO_BOOL INTERFACE
L	
LABEL LEFT LEN LIMIT LITTLEBYTEARRAY_TO_ANYTYPE LN LOG LREAL LREAL_TO_DINT LREAL_TO_INT	LREAL_TO_REAL LREAL_TO_SINT LREAL_TO_STRING LREAL_TO_UDINT LREAL_TO_UINT LREAL_TO_USINT LREAL_VALUE_TO_BOOL LREAL_VALUE_TO_BYTE LREAL_VALUE_TO_DWORD LREAL_VALUE_TO_WORD
G	
MAX MID MIN	MOD MUX
N	
NOT	
O	
OF	OR
P	
PROGRAM	

R	
R_TRIG	REAL_VALUE_TO_BYTE
REAL	REAL_VALUE_TO_DWORD
REAL_TO_DINT	REAL_VALUE_TO_WORD
REAL_TO_DWORD	REPEAT
REAL_TO_INT	REPLACE
REAL_TO_LREAL	RETAIN
REAL_TO_SINT	RETURN
REAL_TO_STRING	RIGHT
REAL_TO_TIME	ROL
REAL_TO_UDINT	ROR
REAL_TO_UINT	RS
REAL_TO_USINT	RTC
REAL_VALUE_TO_BOOL	
S	
SEL	SINT_TO_WORD
SHL	SINT_VALUE_TO_BOOL
SHR	SQRT
SIN	SR
SINT	STRING
SINT_TO_BYTE	STRING_TO_DINT
SINT_TO_DINT	STRING_TO_LREAL
SINT_TO_DWORD	STRING_TO_REAL
SINT_TO_INT	STRING_TO_UDINT
SINT_TO_LREAL	STRUCT
SINT_TO_REAL	StructAlarmId
SINT_TO_UDINT	STRUCTALARMID_TO_DINT
SINT_TO_UINT	StructTaskId
SINT_TO_USINT	
T	
TAN	TOD
THEN	TOF
TIME	TON
TIME_OF_DAY	TP
TIME_TO_INT	TRUE
TIME_TO_REAL	TRUNC
TO	TYPE

U

UDINT
UDINT_TO_BYTE
UDINT_TO_DINT
UDINT_TO_DWORD
UDINT_TO_INT
UDINT_TO_LREAL
UDINT_TO_REAL
UDINT_TO_SINT
UDINT_TO_STRING
UDINT_TO_UINT
UDINT_TO_USINT
UDINT_TO_WORD
UDINT_VALUE_TO_BOOL
UINT
UINT_TO_BYTE
UINT_TO_DINT
UINT_TO_DWORD
UINT_TO_INT
UINT_TO_LREAL
UINT_TO_REAL
UINT_TO_SINT
UINT_TO_UDINT
UINT_TO_USINT
UINT_TO_WORD
UINT_VALUE_TO_BOOL
UNIT
UNTIL
USELIB
USEPACKAGE
USES
USINT
USINT_TO_BYTE
USINT_TO_DINT
USINT_TO_DWORD
USINT_TO_INT
USINT_TO_LREAL
USINT_TO_REAL
USINT_TO_SINT
USINT_TO_UDINT
USINT_TO_UINT
USINT_TO_WORD
USINT_VALUE_TO_BOOL

V

VAR
VAR_GLOBAL
VAR_IN_OUT
VAR_INPUT
VAR_OUTPUT
VAR_TEMP
VOID

W

WAITFORCONDITION
WHILE
WITH
WORD
WORD_TO_BOOL
WORD_TO_BYTE
WORD_TO_DINT
WORD_TO_DWORD
WORD_TO_INT
WORD_TO_SINT
WORD_TO_UDINT
WORD_TO_UINT
WORD_TO_USINT
WORD_VALUE_TO_LREAL
WORD_VALUE_TO_REAL

X

XOR

3.2.3.2 Additional reserved identifiers

The table contains additional reserved identifiers that are reserved for future expansions.

Table 3-2 Additional reserved identifiers of the ST language

A	
ACTION ADD ADD_DT_TIME	ADD_TIME ADD_TOD_TIME
B	
BCD_TO_BYTE BCD_TO_DINT BCD_TO_DWORD BCD_TO_INT	BCD_TO_LWORD BCD_TO_SINT BCD_TO_WORD BYTE_TO_BCD
C	
CONFIGURATION CTD_LINT CTD_ULINT CTU_LINT	CTU_ULINT CTUD_LINT CTUD_ULINT
D	
DINT_TO_BCD DIV	DIVTIME DWORD_TO_BCD
E	

EN
END_ACTION
END_CONFIGURATION
END_RESOURCE
	END_STEP
	END_TRANSITION
	ENO
	EQ

F	
F_EDGE	FROM
G	
GE	GT
I	
INITIAL_STEP	INT_TO_BCD
L	
LE LINT PM	LWORD LWORD_TO_BCD
G	
MUL	MULTIME
N	
MS	
R	
R_EDGE	RESOURCE

S	
SEMA SINT_TO_BCD STEP SUB SUB_DATE_DATE	SUB_DT_DT SUB_DT_TIME SUB_TIME SUB_TOD_TIME SUB_TOD_TOD
T	
TRANSITION	
U	
ULINT	
V	
VAR_ACCESS VAR_ALIAS	VAR_EXTERNAL VAR_OBJECT
W	
WORD_TO_BCD	

3.2.4 Numbers and Boolean values

Numbers can be written in various ways in ST. A number can contain a sign, a decimal point or an exponent. The following rules apply to all numbers:

- Commas and blanks may not appear within a number.
- An underscore (_) is allowed as a visual separator.
- The number can be preceded by a plus (+) or minus (–). If no sign is used, it is assumed that the number is positive.
- Numbers may not violate certain maximum and minimum values.

3.2.4.1 Integers

An integer contains neither a decimal point nor an exponent. An integer is thus a sequence of numeric digits that can be preceded with a sign.

The following are valid integers:

```
0                    1                    +1                 -1
743                  -5280                60_000             -32_211_321
```

The following integers are **invalid** for the reasons indicated:

`123,456`	Commas are not permitted.
`36.`	An integer may not contain a decimal point.
`10 20 30`	Blanks are not permitted.

In ST, you can represent integers in different number systems. This is achieved by inserting a keyword prefix for the number system.

The following are used:

- 2# for the binary system
- 8# for the octal system
- 16# for the hexadecimal system.

Valid representations of the decimal number 15 are:

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

3.2.4.2 Floating-point numbers

A floating-point number can contain a decimal point or an exponent (or both). A decimal point must appear between two digits. A floating-point number therefore cannot start or end with a decimal point.

The following are valid floating-point numbers:

```
0.0             1.3             -0.2            827.602
0000.0          +0.000743       60_000.15       -315.0066
```

The following floating-point numbers are **invalid**:

`1.`	A numeric digit must be present before the decimal point and after the decimal point.
`1,000.0`	Commas are not permitted.
`1.333.333`	Two points are not permitted.

3.2.4.3 Exponents

An exponent can be included to define the position of the decimal point. If no decimal point appears, it is assumed that it is on the right side of the digit. The exponent itself must be either a positive or negative integer. Base 10 is expressed by the letter E.

The magnitude 3 x 10^8 can be represented in ST by the following correct floating-point numbers:

```
3.0E+8      3.0E8       3e+8        3E8         0.3E+9
0.3e9       30.0E+7     30e7
```

The following floating-point numbers are **invalid**:

`3.E+8` A numeric digit must be present before the decimal point and after the decimal point.

`8e2.3` The exponent must be an integer.

`.333e-3` A numeric digit must be present before the decimal point and after the decimal point.

`30 E8` Blanks are not permitted.

3.2.4.4 Boolean values

Boolean values are bit constants. They must be represented by a value of zero (0) or one (1) or by the keywords FALSE or TRUE.

Example:

```
a := 1;        // is equivalent a := TRUE
b := FALSE; // is equivalent to b := 0
```

3.2.4.5 Data types of numbers

The compiler automatically selects the elementary data type that is suitable for the number depending on its value and use (in an expression or a value assignment).

You can also specify the data type directly: Place the data type (numeric data type or bit data type) and the character "#" in front of the number.

Examples:

```
INT#255              INT#16#FF            INT#8#377
WORD#255             WORD#16#FF           WORD#8#377
REAL#255             REAL#16#FF           REAL#8#377
REAL#255.0           REAL#2.55E2          LREAL#255.0
```

Note

Floating-point numbers can only be assigned to data types REAL and LREAL.

3.2.5 Character strings

What is a character string?

A character string is a sequence of zero or more characters with an apostrophe at the start and at the end. Each character is encoded with 1 byte (8 bits) in the string.

A character can be entered as follows:

- As printable characters (ASCII code $20 to $7E, $80 to $FF), except the dollar signs (ASCII code $24) and apostrophe (ASCII code $27), as these have a special function within the string
- As the 2-digit hexadecimal ASCII code of the relevant character preceded by the dollar sign ($)
- As a combination of two characters according to the following table:

Table 3-3 2-character combinations for special characters in strings

Character combination			Meaning
`$$`			Dollar sign $ ($24)
`$’`			Apostrophe ’ ($27)
`$L`	or	`$l`	Line Feed LF ($0A)
`$N`	or	`$n`	Carriage Return + Line Feed CR + LF ($0D$0A)
`$P`	or	`$p`	Form Feed FF ($0C)
`$R`	or	`$r`	Carriage Return CR ($0D)
`$T`	or	`$t`	Horizontal tab (HT) ($09)

Examples:

`’’`	Empty string (length 0).
`’A’`	String of length 1 containing the letter A.
`’ ’`	String of length 1 containing a blank.
`’$’’`	String of length 1 containing an apostrophe.
`’$R$L’` `’$0D$0A’`	Two equivalent representations for a string of length 2 containing the characters CR and LF.
`’$$1.00’`	String of length 5 containing $1.00.
`’Text$R$L’`	String of length 6 containing the word Text followed by the characters CR and LF.
`’ÄÖü’` `’$C4$D6$FC’`	Two equivalent representations for a string of length 3 containing the German umlauts ÄÖü (A, O, u with diaresis).

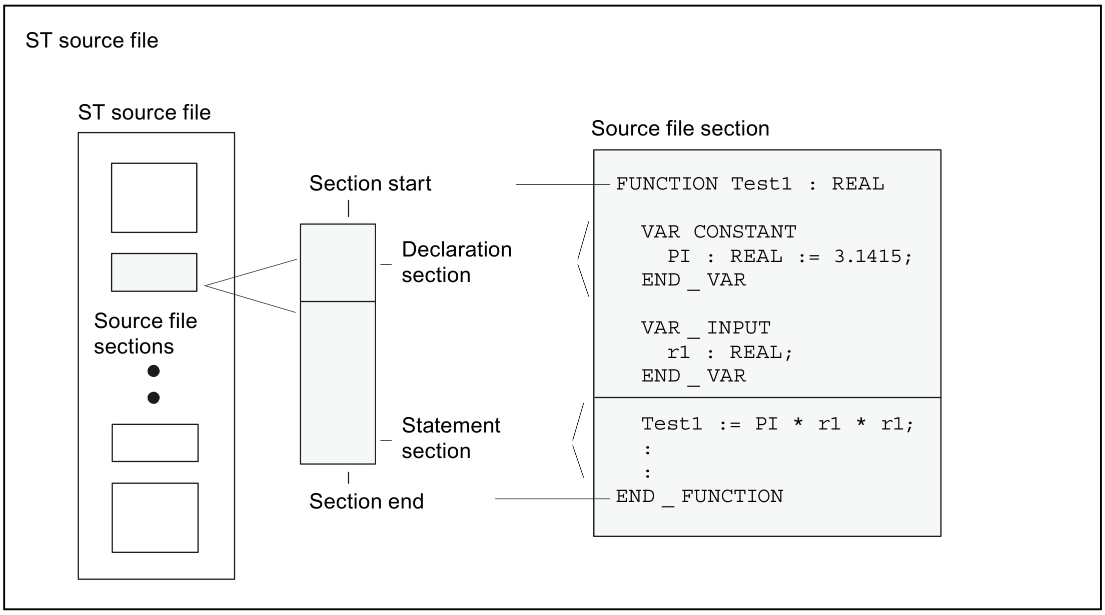
3.3 Structure of an ST source file

An ST source basically consists of continuous text. This text can be structured by dividing it into logical sections. Detailed rules for this can be found in Source file sections (Page 169).

A brief summary is given below:

- An **ST source file** is a logical unit that you create in your project and that can appear several times. It is often referred to as a unit.
- The logic sections of an ST source file are called **Sections** (see table).
- A **user program** is the sum of all program sources (e.g. ST source files, MCC units).

Each logical section of the ST source file has a beginning and end denoted by specific keywords:



```
FUNCTION Test1 : REAL

  VAR CONSTANT
    PI : REAL := 3.1415;
  END _ VAR

  VAR _ INPUT
    r1 : REAL;
  END _ VAR

  Test1 := PI * r1 * r1;
  :
  :
END _ FUNCTION
```

Figure 3-4 Structure of an ST source file

You do not have to program every function yourself. You can also make use of SIMOTION system components. These are preprogrammed sections such as system functions or the functions of the technology objects (TO functions).

Table 3-4 Major sections of an ST source file

Source file section	Description
Unit statement (optional)	Contains the name of the ST
Interface section	Contains statements for importing and exporting variables, types and program organization units (POUs).
Implementation section	Contains executable sections of the ST source file.
POU (program organization unit)	Single executable section of the ST source file (program, function, function block)
Declaration section	Contains declarations (e.g. of variables and types), can be included in the interface section and the implementation section as well as in a POU.
Statement section	Contains executable statements of a POU.

Note

An extensively annotated *template for example unit* is also available in the online Help. You can use it as a template for a new ST source file.

Call the ST editor Help and click the relevant link. Copy the text to the open window of the ST editor and modify the template according to your requirements.

Template for example unit contains a copy of this template.

3.3.1 Statements

The statement section of a program organization unit (POU – program, function, function block) consists of repeated single statements. It follows the declaration section of a POU and ends with the end of the POU. There are no explicit keywords for the start and end.

There are three basic types of statements in ST:

- Value assignments

  Assignment of an expression to a variable, see Variable declaration (Page 105)

- Control statements

  Repetition or branching of statements, see Control statements (Page 130)

- Subroutine execution

  Functions (FC) and function blocks (FB), see Functions, Function Blocks, and Programs (Page 147)

Table 3-5 Examples of statements

```
...
// Value assignment
 Status := 17;

// Control statement
 IF a = b THEN
     FOR c := 1 TO 10 DO
         b := b + c;
     END_FOR;
 END_IF;

// Function call
 retVal := Test1(10.0);
...
```

3.3.2 Comments

Comments are used for documentation purposes and to help the user understand the source file section. After compilation, they have no meaning for the program execution.

There are two types of comments:

- Line comment
- Block comment

The line comment is preceded by //. The compiler will process the text which follows until the end of the line as a comment.

You can enter a block comment over several lines if it is preceded by (* and ends with *).

Please note the following when inserting comments:

- You can use the complete extended ASCII character set in comments.
- The character pairs (* and *) are ignored within the line comment.
- Nesting of block comments is not allowed. However, you can nest line comments in block comments.
- Comments can be inserted at any position, but not in rules that have to be maintained, such as in names of identifiers. For more information about these rules, refer to Language description resources (Page 291).

Table 3-6 Examples of comments

```
 // This is a one-line comment.
 a := 5;

 // This is an example of a one-line comment
 // used several times in succession.
 b := 23;

(* The above example is easier to edit as a
   multi-line comment.
  *)
 c := 87;
```

3.4 Data types

A data type is used to determine how the value of a variable or constant is to be used in a program source.

The following data types are available to the user:

- Elementary data types
- User-defined data types (UDT)
  - Simple derivatives
  - Arrays
  - Enumerators
  - Structures (Struct)
- Technology object data types
- System data types

See also

Elementary data types (Page 90)

Description of the technology object data types (Page 101)

System data types (Page 104)

3.4.1 Elementary data types

3.4.1.1 Elementary data types

Elementary data types define the structure of data that cannot be broken down into smaller units. An elementary data type describes a memory area with a fixed length and stands for bit data, integers, floating-point numbers, duration, time, date and character strings.

All the elementary data types are listed in the table below:

Table 3-7 Bit widths and value ranges of the elementary data types

Type	Reserv. word	Bit width	Range of values
Bit data type Data of this type use either 1 bit, 8 bits, 16 bits or 32 bits. The initialization value of a variable of this data type is 0.			
Bit	BOOL	1	0, 1 or FALSE, TRUE
Byte	BYTE	8	16#0 to 16#FF
Word	WORD	16	16#0 to 16#FFFF
Double word	DWORD	32	16#0 to 16#FFFF_FFFF
Numeric types These data types are available for processing numeric values. The initialization value of a variable of this data type is 0 (all integers) or 0.0 (all floating-point numbers).			
Short integer	SINT	8	-128 to 127 (-2**7 to 2**7-1)
Unsigned short integer	USINT	8	0 to 255 (0 to 2**8-1)
Integer	INT	16	-32_768 to 32_767 (-2**15 to 2**15-1)
Unsigned integer	UINT	16	0 to 65_535 (0 to 2**16-1)
Double integer	DINT	32	-2_147_483_648 to 2_147_483_647 (-2**31 to 2**31-1)
Unsigned double integer	UDINT	32	0 to 4_294_96_7295 (0 to 2**32-1)
Floating-point number (per IEEE -754)	REAL	32	-3.402_823_466E+38 to -1.175_494_351E−38, 0.0, +1.175_494_351E−38 to +3.402_823_466E+38 Accuracy: 23-bit mantissa (corresponds to 6 decimal places), 8-bit exponent, 1-bit sign.
Long floating-point number (in accordance with IEEE-754)	LREAL	64	-1.797_693_134_862_315_8E+308 to -2.225_073_858_507_201_4E−308, 0.0, +2.225_073_858_507_201_4E−308 to +1.797_693_134_862_315_8E+308 Accuracy: 52-bit mantissa (corresponds to 15 decimal places), 11-bit exponent, 1-bit sign.

Type	Reserv. word	Bit width	Range of values
Time types These data types are used to represent various date and time values.			
Duration in increments of 1 ms	TIME	32	T#0d_0h_0m_0s_0ms to T#49d_17h_2m_47s_295ms Maximum of two digits for the values day, hour, minute, second and a maximum of three digits for milliseconds Initialization with T#0d_0h_0m_0s_0ms
Date in increments of 1 day	DATE	32	D#1992-01-01 to D#2200-12-31 Leap years are taken into account, year has four digits, month and day are two digits each Initialization with D#0001-01-01
Time of day in steps of 1 ms	TIME_OF_DAY (TOD)	32	TOD#0:0:0.0 to TOD#23:59:59.999 Maximum of two digits for the values hour, minute, second and maximum of three digits for milliseconds Initialization with TOD#0:0:0.0
Date and time	DATE_AND_TIME (DT)	64	DT#1992-01-01-0:0:0.0 to DT#2200-12-31-23:59:59.999 DATE_AND_TIME consists of the data types DATE and TIME Initialization with DT#0001-01-01-0:0:0.0
String type Data of this type represents character strings, in which each character is encoded with the specified number of bytes. The length of the string can be defined at the declaration. Indicate the length in "[" and "]", e.g. STRING[100]. The default setting consists of 80 characters. The number of assigned (initialized) characters can be less than the declared length.			
String with 1 byte/character	STRING	8	All characters with ASCII code $00 to $FF are permitted. Default ' ' (empty string)

NOTICE

During variable export to other systems, the value ranges of the corresponding data types in the target system must be taken into account.

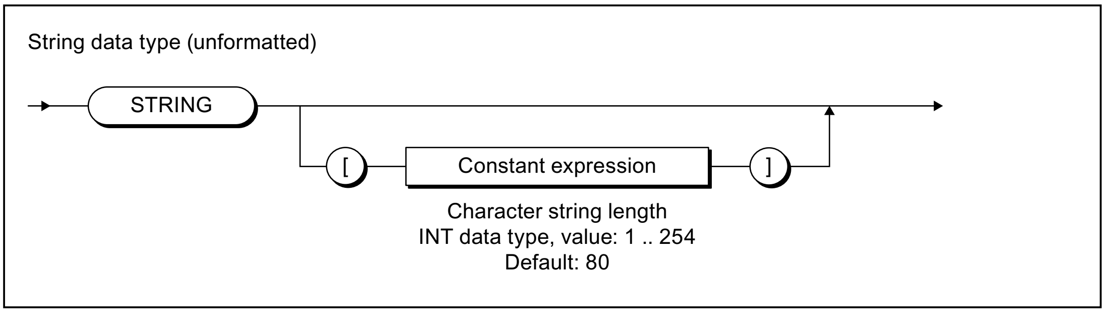


Figure 3-5 Syntax: STRING data type

3.4.1.2 Value range limits of elementary data types

The value range limits of certain elementary data types are available as constants.

Table 3-8 Symbolic constants for the value range limits of elementary data types

Symbolic constant	Data type	Value	Hex notation
SINT#MIN	SINT	-128	16#80
SINT#MAX	SINT	127	16#7F
INT#MIN	INT	-32768	16#8000
INT#MAX	INT	32767	16#7FFF
DINT#MIN	DINT	-2147483648	16#8000_0000
DINT#MAX	DINT	2147483647	16#7FFF_FFFF
USINT#MIN	USINT	0	16#00
USINT#MAX	USINT	255	16#FF
UINT#MIN	UINT	0	16#0000
UINT#MAX	UINT	65535	16#FFFF
UDINT#MIN	UDINT	0	16#0000_0000
UDINT#MAX	UDINT	4294967295	16#FFFF_FFFF
T#MIN TIME#MIN	TIME	T#0ms	16#0000_0000[1]
T#MAX TIME#MAX	TIME	T#49d_17h_2m_47s_295ms	16#FFFF_FFFF[1]
TOD#MIN TIME_OF_DAY#MIN	TOD	TOD#00:00:00.000	16#0000_0000[1]
TOD#MAX TIME_OF_DAY#MAX	TOD	TOD#23:59:59.999	16#0526_5BFF[1]
[1] Internal display only			

3.4.1.3 General data types

General data types are often used for the input and output parameters of system functions and system function blocks. The subroutine can be called with variables of each data type that is contained in the general data type.

The following table lists the available general data types:

Table 3-9 General data types

General data type	Data types contained
ANY_BIT	BOOL, BYTE, WORD, DWORD
ANY_INT	SINT, INT, DINT, USINT, UINT, UDINT
ANY_REAL	REAL, LREAL
ANY_NUM	ANY_INT, ANY_REAL
ANY_DATE	DATE, TIME_OF_DAY (TOD), DATE_AND_TIME (DT)
ANY_ELEMENTARY	ANY_BIT, ANY_NUM, ANY_DATE, TIME, STRING
ANY	ANY_ELEMENTARY, user-defined data types (UDT), system data types, data types of the technology objects

Note

You **cannot** use general data types as type identifiers in variable or type declarations.

The general data type is retained when a user-defined data type (UDT) is derived directly from an elementary data type (only possible with the SIMOTION ST programming language).

3.4.1.4 Elementary system data types

In the SIMOTION system, the data types specified in the table are treated similarly to the elementary data types. They are used with many system functions.

Table 3-10 Elementary system data types and their use

Identifier	Bit width	Use
StructAlarmId	32	Data type of the alarmId for the project-wide unique identification of the messages. The alarmId is used for the message generation. See Function Manual *SIMOTION Basic Functions*. Initialization with STRUCTALARMID#NIL
StructTaskId	32	Data type of the taskId for the project-wide unique identification of the tasks in the execution system. See Function Manual *SIMOTION Basic Functions*. Initialization with STRUCTTASKID#NIL

Table 3-11 Symbolic constants for invalid values of elementary system data types

Symbolic constant	Data type	Significance
STRUCTALARMID#NIL	StructAlarmId	Invalid AlarmId
STRUCTTASKID#NIL	StructTaskId	Invalid TaskId

3.4.2 User-defined data types

3.4.2.1 User-defined data types

User-defined data types (UDT) are created with the construct TYPE/END_TYPE in the declaration sections of subsequent source file sections (see Structure of an ST source file (Page 86) and Source file sections (Page 169)) of the following:

- Interface section
- Implementation section
- Program organization unit (POU)

You can continue to use the data types you created in the declaration section. The source file section determines the range of the type declaration.

See also

Syntax of user-defined data types (type declaration) (Page 95)

Derivation of elementary or derived data types (Page 96)

Derived data type ARRAY (Page 97)

Derived data type - Enumerator (Page 99)

Derived data type STRUCT (structure) (Page 100)

3.4.2.2 Syntax of user-defined data types (type declaration)

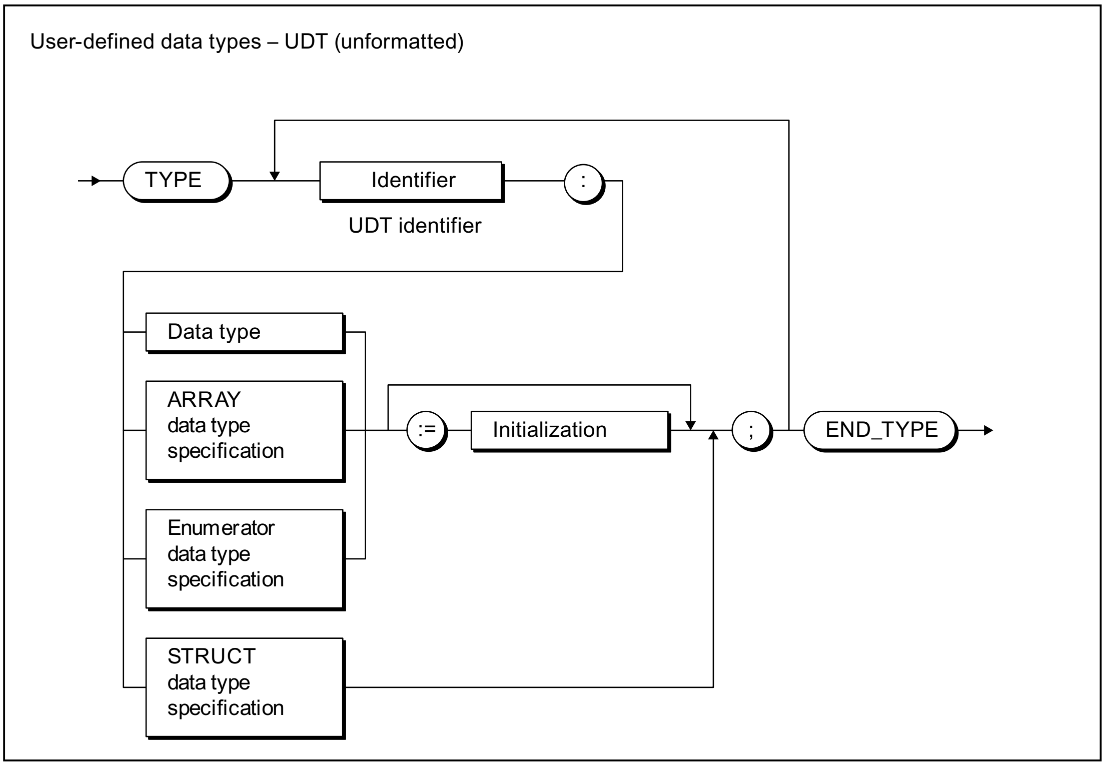

User-defined data types – UDT (unformatted)
TYPE
Identifier
UDT identifier
:
Data type
ARRAY data type specification
Enumerator data type specification
STRUCT data type specification
:=
Initialization
;
END_TYPE

Figure 3-6 Syntax: User-defined data type

The declaration of the UDT is introduced with the keyword TYPE.

For each data type to be declared, this is followed by (see figure):

1. Name:

   The name of the data type must comply with the rules for identifiers.

2. Data type specification

   The term *data type* comprises (see Derivation of elementary or derived data types (Page 96)):

   – Elementary data types

   – Previously declared UDTs

   – TO data types

   – System data types

   The following data type specifications are also possible:

   – ARRAY data type specification (see Derived data type ARRAY - field (Page 97))

   – Enumerator data type specification (see Derived data type enumerator (Page 99))

   – STRUCT data type specification (see Derived data type STRUCT – structure (Page 100))

   The references in brackets refer to the following sections, in which the respective data type specification is described in detail.

3. Optional initialization:

   You can specify an initialization value for the data type. If you subsequently declare a variable of this data type, the initialization value is assigned to the variable.
   Exception: With the STRUCT data type specification, each individual component is initialized within the data type specification.

   See also Initialization of variables or data types (Page 107).

The complete UDT declaration is terminated with the keyword END_TYPE. You can create any number of data types within the TYPE/END_TYPE construct. You can use the defined data types to declare variables or parameters.

UDTs can be nested in any way as long as the syntax in the figure is observed. For example, you can use previously defined UDTs or nested structures as a data type specification. Type declarations can only be used sequentially and not in nested structures.

Note

You can learn how to declare variables and parameters in Overview of all variable declarations (Page 106), and how to assign values with UDT in Syntax for value assignment (Page 113).

Below is a description of individual data type specifications for UDTs and examples demonstrating their use.

3.4.2.3 Derivation of elementary or derived data types

In the derivation of data types, an elementary or user-defined data type (UDT) is assigned to the data type to be defined in the TYPE/END_TYPE construct:

TYPE identifier : Elementary data type { := initialization } ; END_TYPE
TYPE identifier : User-defined data type { := initialization } ; END_TYPE

Once you have declared the data type, you can define variables of derived data type identifier. This is equivalent to declaring variables as data type *elementary data type*.

Table 3-12 Examples of derivation of elementary data types

```
TYPE
    I1: INT;      // Elementary data type
    R1: REAL;     // Elementary data type
    R2: R1;       // Derived data type (UDT)
END_TYPE
VAR
    // These variables can be used wherever
    // variables of type INT can be used.
    myI1 : I1;
    myI2 : INT;    // No derived data type!

    // These variables can be used wherever
    // variables of type REAL can be used.
    myR1 : R1;
    myR2 : R2;
END_VAR
myI1 := 1;
myI2 := 2;
myR1 := 2.22;
myR2 := 3.33;
```

3.4.2.4 Derived data type ARRAY

The ARRAY derived data type combines a defined number of components of the same data type in the TYPE/END_TYPE construct. The syntax diagram in the following figure shows this data type, which is specified more precisely after the reserved identifier OF.

TYPE identifier : ARRAY data type specification { := initialization } ; END_TYPE

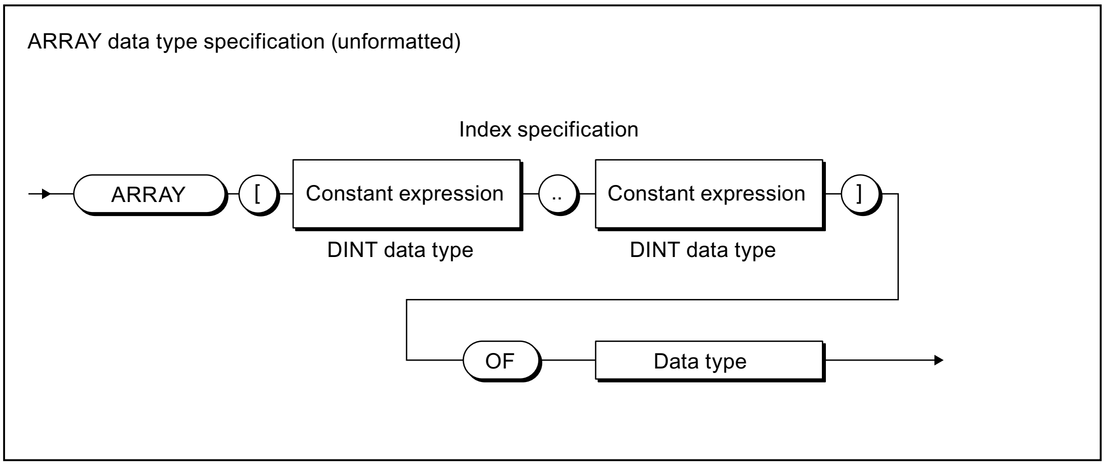


Figure 3-7 Syntax: ARRAY data type specification

The index specification describes the limits of the array:

- The array limits specify the minimum and maximum value for the index. They can be specified using constants or constant expressions; the data type is DINT (or can be implicitly converted to DINT – see Elementary data type conversion (Page 141)).
- The array limits must be separated by two periods.
- The entire index specification is enclosed in square brackets.
- The index itself can be an integer value of data type DINT (or it can be implicitly converted to DINT – see Elementary data type conversion (Page 141)).

Note

If array limits are violated during runtime, a processing error occurs in the program (see *SIMOTION Basic Functions Function Manual*).

You declare the data type of the array components with the data type specification. All of the options described in this chapter can be used as data types, for example, even user-defined data types (UDT).

There are several different ARRAY types:

- The one-dimensional ARRAY type is a list of data elements arranged in ascending order.
- The two-dimensional ARRAY type is a table of data consisting of lines and columns. The first dimension refers to the line number, the second to the column number.
- The higher-dimensional ARRAY type is an expansion of the two-dimensional ARRAY type that includes additional dimensions.

Table 3-13 Examples of one-dimensional arrays

```
TYPE
    x : ARRAY[0..9] OF REAL;
    y : ARRAY[1..10] OF C1;
END_TYPE
```

Two-dimensional arrays are comparable to a table with lines and columns. You can create two- or multi-dimensional arrays by means of a multi-level type declaration, see example:

Table 3-14 Examples of multi-dimensional arrays

```
TYPE
    a : ARRAY[1..3] OF INT;    // one-dimensional array (3 columns):
    matrix1: ARRAY[1..4] OF a;      // two-dimensional Field
                                    // (4 lines with 3 columns)
    b: ARRAY[4..8] OF INT;    // one-dimensional array (5 columns):
    matrix2: ARRAY[10..16] OF b;    // two-dimensional Field
                                    // (7 lines with 5 columns)
END_TYPE

VAR
    m: matrix1;       // Variable m of data type two-dim. Field
    n: matrix2;       // Variable m of data type two-dim. Field
END_VAR

m[4][3] := 9;            // Write to Matrix1 at line 4, column 3
n[16][8] := 10;          // Write to Matrix2 at line 7, column 5
```

In the example, you can define:

1. Table columns a[1] to a[3] as a one-dimensional array that will contain integers.
2. Table lines matrix1[1] to matrix2[4] also as an array but take as the data type specification the array a you just created with the columns of the table.

   When you specify an array in the data type specification, you create a second dimension. You can create further dimensions in this way.

Now declare a variable using the data type created for the table. You address each dimension of the table using square brackets, in this case specifying the line and column.

3.4.2.5 Derived data type - Enumerator

In the case of enumerator data types, a restricted set of identifiers or names is assigned to the data type to be defined in the TYPE/END_TYPE construct:

TYPE identifier : Enumerator data type specification { := initialization } ; END_TYPE

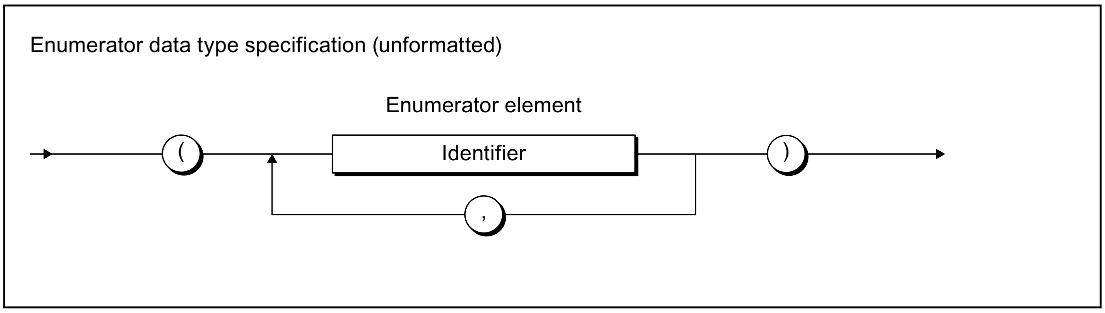


Figure 3-8 Syntax: Enumerator data type specification

Once you have declared the *identifier* data type, you can define variables in the enumerator data type. In the statement section, you can assign only elements from the list of defined identifiers (enumerator elements) to these variables.

You can also specify the data type directly: Place the enumerator data type identifier and the "#" sign in front of the enumerator element (see Table *Examples of enumerator data types*).

You can obtain the first and last value of an enumeration data type with *enum_type*#MIN and *enum_type*#MAX respectively, whereby *enum_type* is the enumeration data type identifier.

You can obtain the numeric value of an enumeration element with the ENUM_TO_DINT conversion function.

Table 3-15 Examples of enumerator data types

```
TYPE
    C1: (RED, GREEN, BLUE);
END_TYPE

VAR
    myC11, myC12, myC13 : C1;
END_VAR

myC11 := GREEN;
myC11 := C1#GREEN;
myC12 := C1#MIN;        // RED
myC13 := C1#MAX;        // BLUE
```

Note

You will also find enumerator data types as system data types.

Enumerator data types can be components of a structure, meaning that they can be found at any lower level in the user-defined data structure.

3.4.2.6 Derived data type STRUCT (structure)

The derived data type STRUCT, or structure, encompasses an area of a fixed number of components in the TYPE/END_TYPE construct; the data types of these components can vary:

TYPE identifier : STRUCT data type specification; END_TYPE

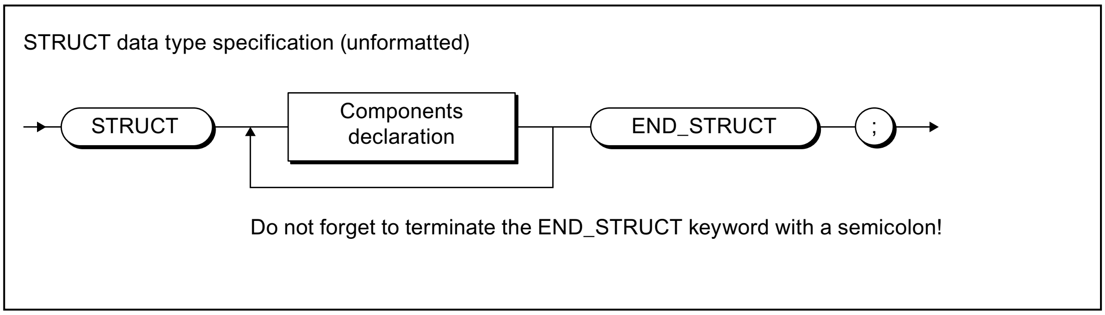

STRUCT data type specification (unformatted)
STRUCT
Components declaration
END_STRUCT
;
Do not forget to terminate the END_STRUCT keyword with a semicolon!

Figure 3-9 Syntax: STRUCT data type specification

The syntax of the component declaration is shown in the following figure.

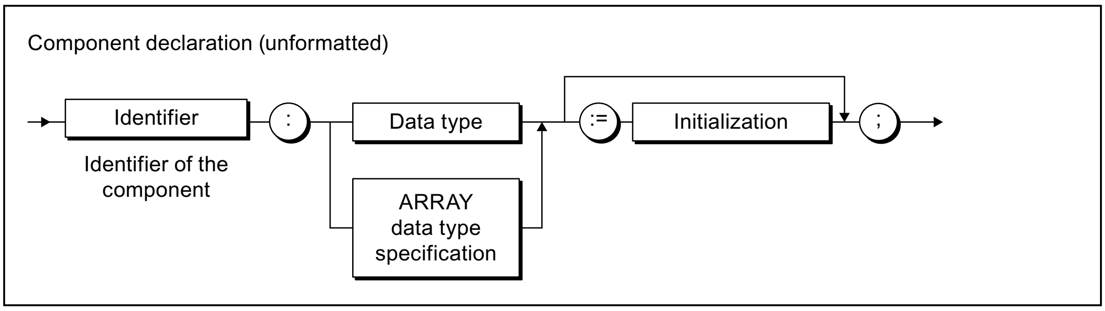

Component declaration (unformatted)
Identifier
Identifier of the component
:
Data type
ARRAY data type specification
:=
Initialization
;

Figure 3-10 Syntax: Component declaration

The following are permitted as data types:

- Elementary data types
- Previously declared UDTs
- System data types
- TO data types
- ARRAY data type specification

You also have the option to assign initialization values to the components. Proceed as for the initialization of variables or data types (see Initialization of variables or data types (Page 107)).

Note

The following data specifications cannot be used directly within a component declaration:

- STRUCT data type specifications
- Enumerator data type specifications

Solution: Declare a UDT (user-defined data type) beforehand with the above-mentioned specifications and use this in the component declaration.

This allows you to nest STRUCT data types.

You will also find STRUCT data types as system data types.

This example shows how a UDT is defined and how this data type is used within a variable declaration.

Table 3-16 Examples of derived data type STRUCT

```
TYPE     // UDT definition
    S1: STRUCT
        var1 : INT;
        var2 : WORD := 16#AFA1;
        var3 : BYTE := 16#FF;
        var4 : TIME := T#1d_1h_10m_22s_2ms;
    END_STRUCT;
END_TYPE

VAR
    myS1 : S1;
END_VAR

myS1.var1 := -4;
myS1.var4 := T#2d_2h_20m_33s_2ms;
```

3.4.3 Technology object data types

3.4.3.1 Description of the technology object data types

You can declare variables with the data type of a technology object (TO). The following table shows the data types for the available technology objects in the individual technology packages.

For example, you can declare a variable with the data type *posaxis* and assign it an appropriate instance of a position axis. Such a variable is often referred to as a reference.

Table 3-17 Data types of technology objects (TO data type)

Technology object	Data type	Contained in the technology package
Drive axis	driveAxis	CAM[1 2], PATH, CAM_EXT
External encoder	externalEncoderType	CAM[1 2], PATH, CAM_EXT
Measuring input	measuringInputType	CAM[1 2], PATH, CAM_EXT
Output cam	outputCamType	CAM[1 2], PATH, CAM_EXT
Cam track (as of V3.2)	_camTrackType	CAM, PATH, CAM_EXT
Position axis	posAxis	CAM[1 3], PATH, CAM_EXT
Following axis	followingAxis	CAM[1 4], PATH, CAM_EXT
Following object	followingObjectType	CAM[1 4], PATH, CAM_EXT
Cam	camType	CAM, PATH, CAM_EXT
Path axis (as of V4.1)	_pathAxis	PATH, CAM_EXT
Path object (as of V4.1)	_pathObjectType	PATH, CAM_EXT
Fixed gear (as of V3.2)	_fixedGearType	CAM_EXT
Addition object (as of V3.2)	_additionObjectType	CAM_EXT
Formula object (as of V3.2)	_formulaObjectType	CAM_EXT
Sensor (as of V3.2)	_sensorType	CAM_EXT
Controller object (as of V3.2)	_controllerObjectType	CAM_EXT
Temperature channel	temperatureControllerType	TControl
General data type, to which every TO can be assigned	ANYOBJECT	
1) As of Version V3.1, the BasicMC, Position and Gear technology packages are no longer contained. *2) For Version V3.0, also contained in the BasicMC, Position and Gear technology packages.* *3) For Version V3.0, also contained in the Position and Gear technology packages.* *4) For Version V3.0, also contained in the Gear technology package.*		

You can access the elements of technology objects (configuration data and system variables) via structures (see SIMOTION Basic Functions Function Manual).

Table 3-18 Symbolic constants for invalid values of technology object data types

Symbolic constant	Data type	Meaning
TO#NIL	ANYOBJECT	Invalid technology object

See also

Inheritance of the properties for axes (Page 103)

Examples of the use of technology object data types (Page 103)

3.4.3.2 Inheritance of the properties for axes

Inheritance for axes means that all of the data types, system variables and functions of the TO driveAxis are fully included in the TO positionAxis. Similarly, the position axis is fully included in the TO followingAxis, the following axis in the TO pathAxis. This has, for example, the following effects:

- If a function or a function block expects an input parameter of the driveAxis data type, you can also use a position axis or a following axis or a path axis when calling.
- If a function or a function block expects an input parameter of the posAxis data type, you can also use a following axis or a path axis when calling.

3.4.3.3 Examples of the use of technology object data types

Below, you will see an example of optional use of a variable with a technology object data type (you will find an example of mandatory use of a variable with a TO data type in the *SIMOTION Basic Functions* Function Manual). A second example shows the alternative without using a variable with TO data type.

A TO function will be used to enable an axis in the main part of a program so that the axis can be positioned. After the positioning operation, the current position of the axis will be recorded using a structure access.

The first example uses a variable with TO data type to demonstrate its use.

Table 3-19 Example of the use of a data type for technology objects

```
VAR
    myAxis : posAxis;   // Declaration variable for axis
    myPos : LREAL;      // Variable for position of axis
    retVal: DINT;      // Variable for return value of the
                // TO function
END_VAR
myAxis := Axis1;       // The name Axis1 was defined when the axis
                       // was configured in the project navigator.

// Call of function with variables of TO data type:
retVal := _enableAxis(axis := myAxis, commandId := _getCommandId());

// Axis is positioned.
retVal := _pos(axis := myAxis,
               position := 100,
               commandId:= _getCommandId() );

// Scan the position using structure access
myPos := myAxis.positioningState.actualPosition;
```

The second example does not use a variable with TO data type.

Table 3-20 Example of using a technology object

```
VAR
    myPos : LREAL;      // Variable for position of axis
    retVal: DINT;  // Variable for return value of TO function
END_VAR

// Call of function without variable of TO data type
// The name Axis1 was defined when the axis
// was configured in the project navigator.
retVal := _enableAxis(axis := Axis1,
                      commandId:= _getCommandId() );

// Axis is positioned.
retVal := _pos(axis := Axis1
                position := 100,
                commandId:= _getCommandId() );

// Scan the position using structure access
myPos := Axis1.positioningState.actualPosition;
```

You will find details for configuration of technology objects in the SIMOTION Motion Control function descriptions.

3.4.4 System data types

There are a number of system data types available that you can use without a previous declaration. And, each imported technology packages provides a library of system data types.

Additional system data types (primarily enumerator and STRUCT data types) can be found

- In parameters for the general standard functions (see *SIMOTION Basic Functions* Function Manual)
- In parameters for the general standard function modules (see *SIMOTION Basic Functions* Function Manual)
- In system variables of the SIMOTION devices (see relevant parameter manuals)
- In parameters for the system functions of the SIMOTION devices (see relevant parameter manuals)
- In system variables and configuration data of the technology objects (see relevant parameter manuals)
- In parameters for the system functions of the technology objects (see relevant parameter manuals)

3.5 Variable declaration

A variable defines a data item with variable contents that can be used in the ST source file. A variable consists of an identifier (e.g. *myVar1*) that can be freely selected and a data type (e.g. BOOL). Reserved identifiers (see Reserved identifiers (Page 75)) must not be used as identifiers.

3.5.1 Syntax of variable declaration

Variables are always created according to the same pattern in the declaration section of a source file section:

1. Start a declaration block with an appropriate keyword (e.g. VAR, VAR_GLOBAL – see Overview of all variable declarations (Page 106)).
2. This is followed by the actual variable declarations (see figure); you can create as many of these as you wish. The order is arbitrary.
3. End the declaration block with END_VAR.
4. You can create further declaration blocks (also with the same keyword).

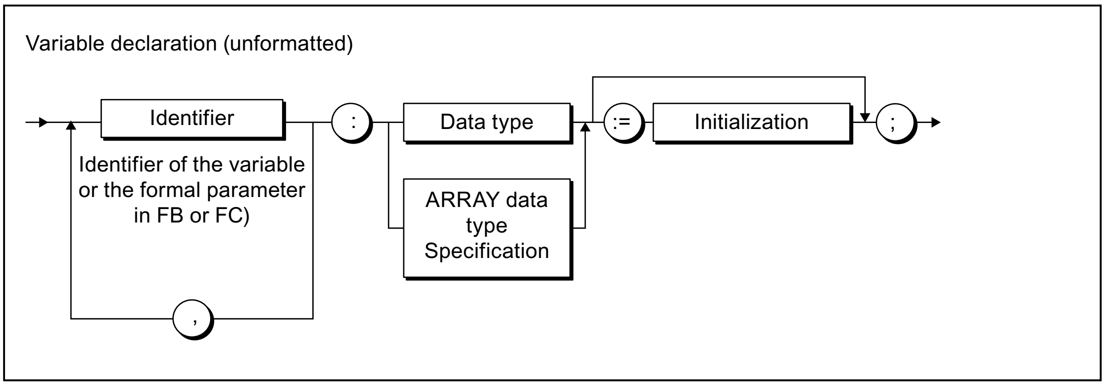

Variable declaration (unformatted)
Identifier
Identifier of the variable or the formal parameter in FB or FC)
,
:
Data type
ARRAY data type Specification
:=
Initialization
;

Figure 3-11 Syntax: Variable declaration

Note the following:

- The variable name must be an identifier, i.e. it can only contain letters, numbers or an underscore, but not special characters.
- The following are permissible as data types:
  - Elementary data types
  - UDT (user-defined data types)
  - System data types
  - TO data types
  - ARRAY data type specifications
  - Designation of a function block (instance declaration – see Calling functions and function modules (Page 153)).
- You can assign initial values to the variables in the declaration statement. This is known as initialization (see Initialization of variables or data types (Page 107)).

Deviations from the pattern presented are as follows:

- For constant declarations (a constant **must** be initialized with a value, see Constants (Page 111)),
- For process image access (see Overview of all variable declarations (Page 106)):
  - A variable declaration is not required for absolute process image access,
  - Initialization is not permitted for symbolic process image access.

Table 3-21 Examples of variable declarations

```
VAR CONSTANT
    PI : REAL := 3.1415;
END_VAR

VAR
    // Declaration of a variable ...
    var1 : REAL;
    // ... or if there are several variables of the same type:
    var2, var3, var4 : INT;
    // Declaration of a one-dimensional array:
    a1 : ARRAY[1..100] OF REAL;
    // Declaration of a character string (string):
    str1 : STRING[40];
END_VAR
```

3.5.2 Overview of all variable declarations

You specify the name, data type, and initial values of variables in the variable and parameter declarations. You always execute these declarations in the declaration sections of the following source file sections:

- Interface section
- Implementation section
- POU (program, function, function block, expression)

The source file section also determines which variables you can declare (see table), as well as their range.

For additional information about the source file sections, refer to Structure of an ST source file (Page 86) and Source file sections (Page 169).

Table 3-22 Keywords for declaration blocks

Keyword	Meaning	Declaration in the following declaration sections
VAR	Declaration of temporary or static variables See Variable model (Page 184)	Any POU
VAR_GLOBAL	Declaration of unit variables See Variable model (Page 184)	Interface section Implementation section
VAR_IN_OUT	Variable declaration of in/out parameter; the POU accesses this variable directly (using a reference) and can change it immediately. See Defining functions (Page 148), Defining function blocks (Page 149)	Function Function block Expression
VAR_INPUT	Variable declaration of input parameter, value is externally supplied and cannot be changed within the POU. See Defining functions (Page 148), Defining function blocks (Page 149)	Function Function block Expression
VAR_OUTPUT	Variables declaration output parameter; value is transmitted from the function block See Defining functions (Page 148), Defining function blocks (Page 149)	Function block
VAR_TEMP	Declaration of temporary variables See Variable model (Page 184)	Program Function block
RETAIN	Declaration of retentive variables See Variable model (Page 184)	Only as a supplement to VAR_GLOBAL in the interface and implementation section
CONSTANT	Declaration of constants See Constants (Page 111)	Only as a supplement: • to VAR in FB, FC, or program • to VAR_GLOBAL in interface or implementation section

3.5.3 Initialization of variables or data types

The assignment of initial values to the variables or data types within a declaration is optional (see Figure *Syntax: Variable declaration* or *Syntax: User-defined data type*):

- If there is no initialization specified in the variable declaration, the compiler automatically assigns the initialization value specified in the data type declaration to the variables.
- If there is no initialization specified in the data type declaration either, the compiler assigns the value of zero to the variables or data types. Exception:
  - For time data types: Initialization values
  - For enumeration data types: 1. value of the enumeration

You preassign a variable or a user-defined data type with initial values by assigning a value (:=) after the data type specification (see Figure *Syntax: Variable initialization*):

- Assign the elementary data types (or data types derived from elementary data types) a constant expression in accordance with Figure *Syntax: constant expression*.
- Assign a field initialization list to a field (ARRAY) according to Figure *Syntax: Field initialization list*.
- Assign a structure initialization list to the individual components of a structure (STRUCT) in accordance with Figure *Syntax: Structure initialization list*.
- Assign an enumerator element to an enumerator data type.

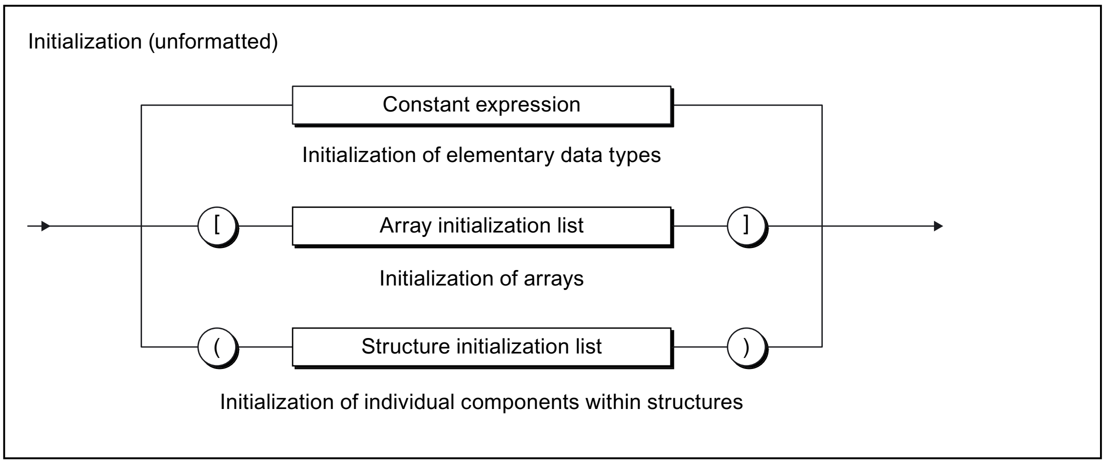

Initialization (unformatted)
Constant expression
Initialization of elementary data types
[
Array initialization list
]
Initialization of arrays
(
Structure initialization list
)
Initialization of individual components within structures

Figure 3-12 Syntax: Variable initialization

The initialization value assigned to a variable is calculated from the constant expression at the time of the compilation. See the figure for the syntax. For information about the syntax of the constant expression, see Figure *Syntax: Constant expression*.

Note that a variable list (a1, a2, a3, .. : INT := ..) can be initialized with a common value. In this case, you do not have to initialize the variables individually (a1 : INT := .. ; a2 : INT := .. ; etc.).

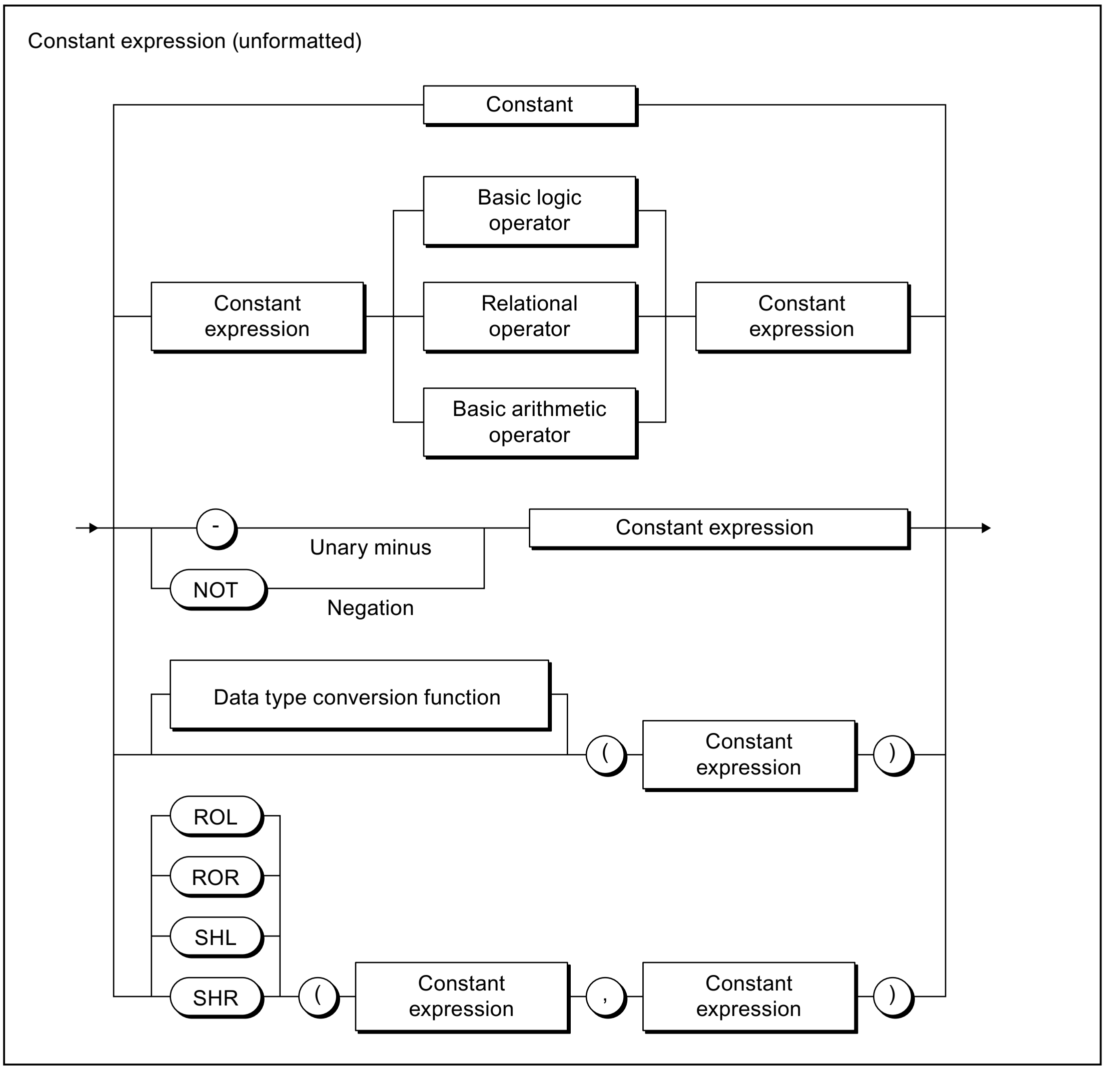


Figure 3-13 Syntax: Constant expression

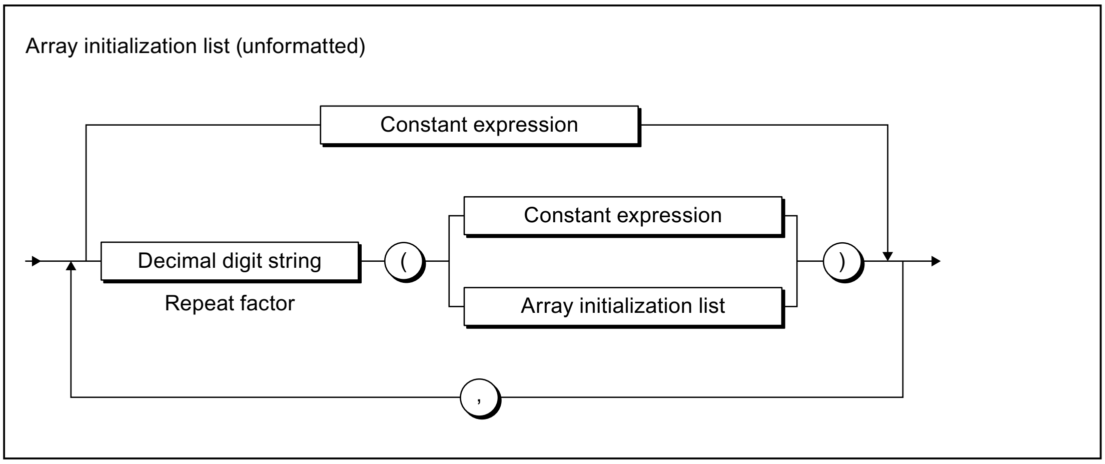


Figure 3-14 Syntax: Array initialization list

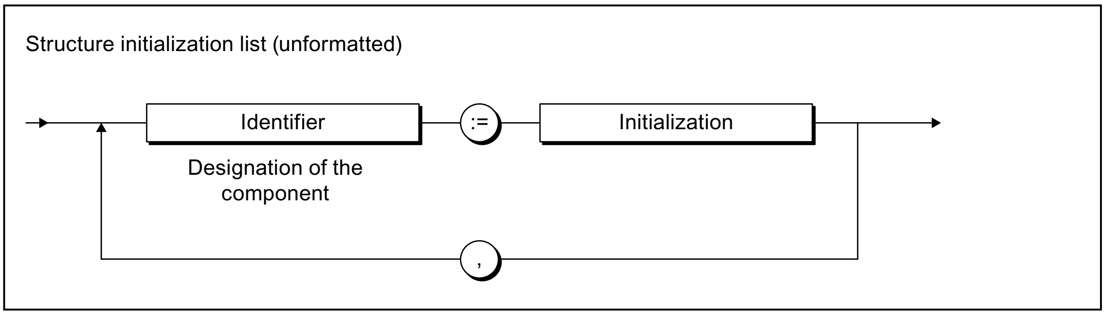

Structure initialization list (unformatted)
Identifier
:=
Initialization
Designation of the component
,

Figure 3-15 Syntax: Structure initialization list

Table 3-23 Examples of variable initialization

```
VAR
    // Declaration of a variable ...
    var1 : REAL := 100.0;
    // ... or if there are several variables of the same type:
    var2, var3, var4 : INT := 1;
    var5 : REAL := 3 / 2;
    var6 : INT  := 5 * SHL(1, 4)
    myC1 : C1 := GREEN;
    array1 : ARRAY [0..4] OF INT  := [1, 3, 8, 4, 0];
    array2 : ARRAY [0..5] OF DINT := [6 (7)];
    array3 : ARRAY [0..10] OF INT := [2 (2(3),3(1)),0];
                // is equivalent to [2(3),3(1),2(3),3(1)),0]
                // Initialization as follows:
                // Array elements 0, 1        with 3;
                // Array elements 2, 3, 4        with 1;
                // Array elements 5, 6        with 3;
                // Array elements 7, 8, 9        with 1;
                // Array element 10            with 0
    myAxis : PosAxis := TO#NIL;
END_VAR
```

Table 3-24 Examples of data type initialization

```
TYPE
    // Initialization of a derived data type
    type1 : REAL := 10.0;
    // Initialization of an enumeration data type
    cmyk_colour : (cyan, magenta, yellow, black) := yellow;
    // Initialization of structures
    var_rgb_colour : STRUCT
            red, green, blue : USINT := 255;// white
        END_STRUCT;
    new_colour : var_rgb_colour := (red := 0, blue := 0);//green
END_TYPE
```

Variables of a technology object (TO) data type are initialized by the compiler with TO#NIL. The effect of tasks on variable initialization is described in the *SIMOTION Basic Functions* Function Manual.

3.5.4 Constants

Constants are data with a fixed value that you cannot change during program runtime. Constants are declared in the same way as variables:

- In the declaration section of a POU for local constants (see Figure *Syntax: Constant block in a POU* and *Syntax: Constant declaration*).
- In the interface or implementation section of the ST source file for unit constants (see Figure*Syntax: Unit constants in the interface or implementation section* and *Syntax: Constant declaration*). You can import unit constants declared in the interface section into other ST source files (see Variable model (Page 184)).

The source file section also determines the range of the constant declaration.

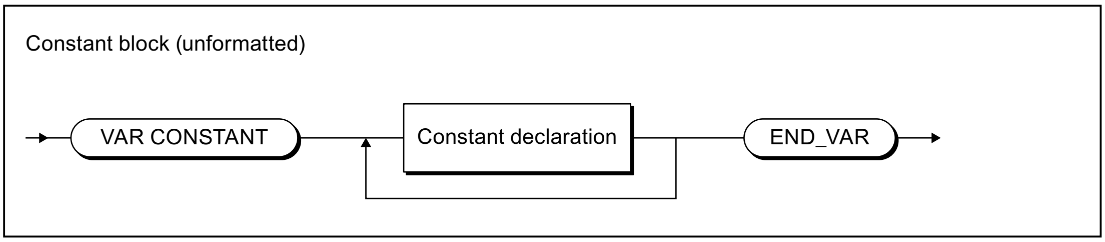


Figure 3-16 Syntax: Constant block in a POU

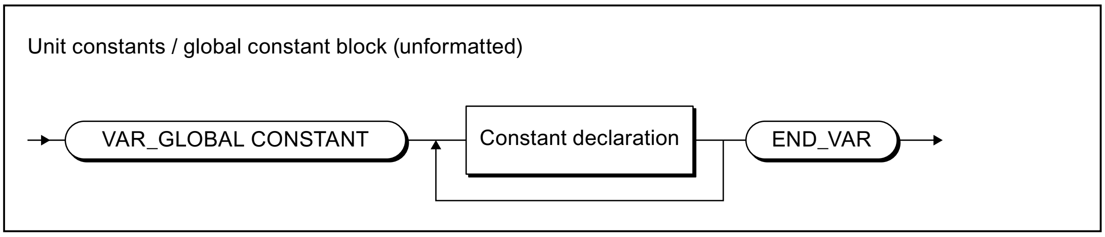


Figure 3-17 Syntax: Unit constants in interface or implementation section

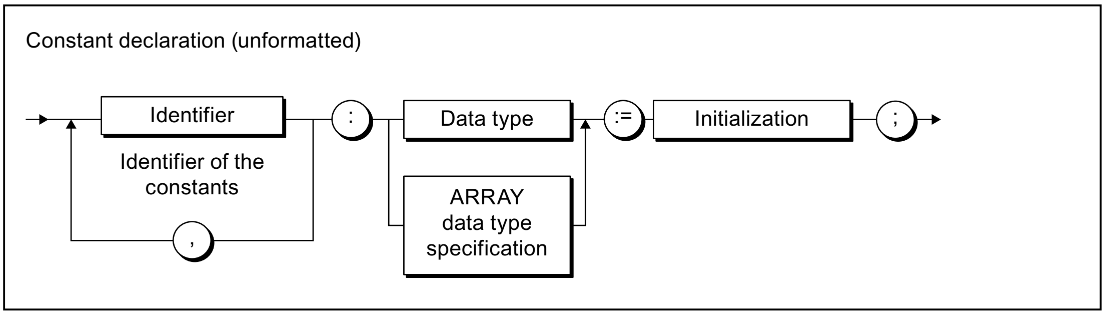

Constant declaration (unformatted)
Identifier
Identifier of the constants
,
:
Data type
ARRAY data type specification
:=
Initialization
;

Figure 3-18 Syntax: Constant declaration

The value assigned to a constant is calculated from the constant expression at the time of compilation. For information about the syntax of the constant expression, see Figure *Syntax: Constant expression*.

Table 3-25 Examples of constants

```
VAR CONSTANT
    PI : REAL := 3.1415;
    intConst       : INT := 10;
    sintConst      : SINT := 0;
    dintConst      : DINT := 10_000;
    timeConst      : TIME := TIME#1h;
    strConst       : STRING[40] := 'Example of a string';
    Two_PI         : REAL := 2 * PI;
END_VAR
```

3.6 Value assignments and expressions

You have already created value assignments with the character string :=, perhaps for a statement as part of the example (see Table *Examples of statements* in Statements (Page 87)) or when initializing variables in the declaration section of a source file section.

However, this is only a small range of the options available for formulating value assignments. This section of the manual now describes this important topic in detail using a large number of examples for illustration purposes.

Note

In arithmetic and logic expressions, the result is always calculated in the largest number format of the expression and converted to the data type of the result. Implicit conversion is not always possible in value assignments. For more information on this error source and its solution, see SIMOTION *Basic Functions* Function Manual.

See also

Notes on avoiding errors and on efficient programming (Page 251)

3.6.1 Value assignments

3.6.1.1 Syntax of the value assignment

A value assignment is used to assign the value of an expression to a variable. The previous value is overwritten. Before a value can be correctly assigned, a variable must be declared in the declaration section (see Syntax of variable declaration (Page 105)).

As shown in the following syntax diagram, the expression is evaluated on the right side of the assignment sign :=. The result is stored in the variable, whose name is on the left side of the assignment sign (target variable). All target variables supported from a formal viewpoint are shown in the figure.

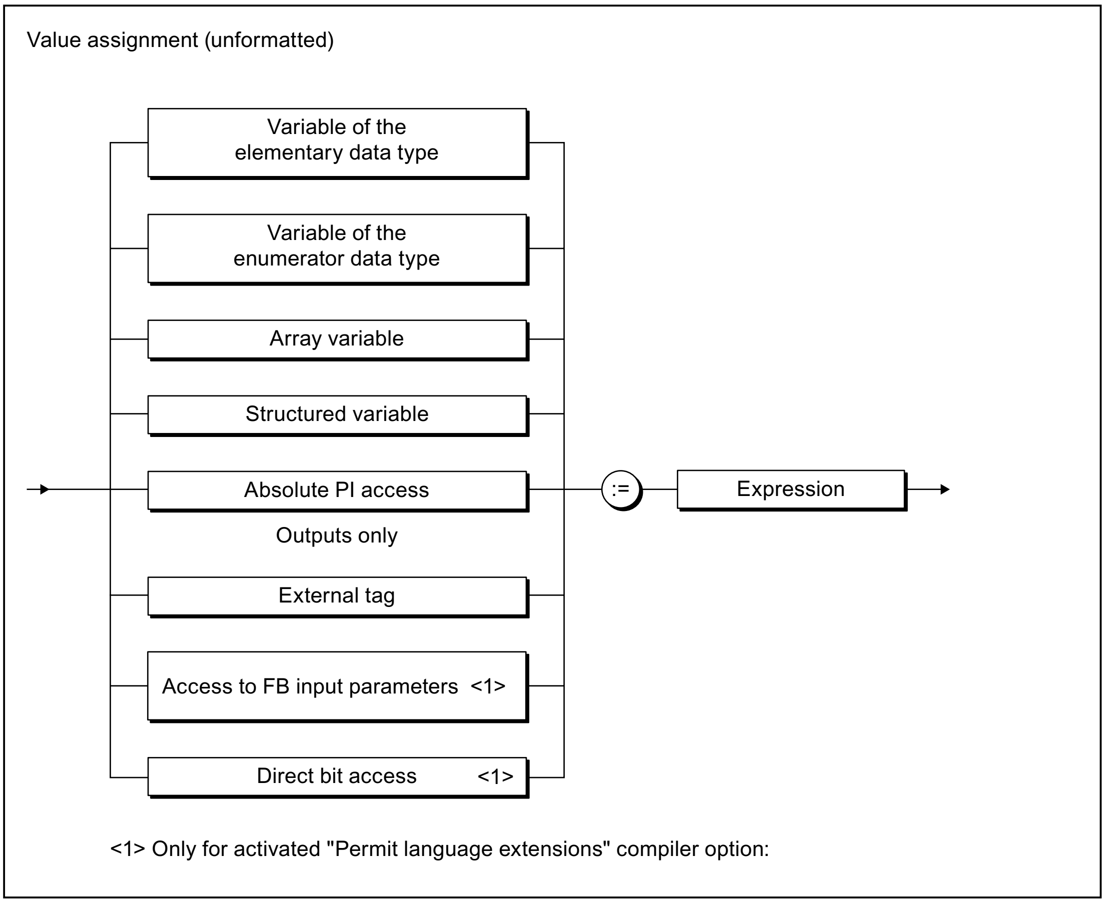

Value assignment (unformatted)
Variable of the elementary data type
Variable of the enumerator data type
Array variable
Structured variable
Absolute PI access
Outputs only
External tag
Access to FB input parameters <1>
Direct bit access <1>
:=
Expression
<1> Only for activated "Permit language extensions" compiler option:

Figure 3-19 Syntax: Value assignments

The following contains explanations and examples for the left side of the value assignment:

- Value assignments with variables of an elementary data type (Page 114) ,
- Value assignments with variables of the derived enumerator data type (Page 117)
- Value assignments with variables of the derived ARRAY data type (Page 118)
- Value assignments with variables of the derived STRUCT data type (Page 118)

- Value assignments with absolute PI access (to addresses of the process image), see: Absolute access to the fixed process image of the BackgroundTask (absolute PI access) (Page 221).

How the right side of a value assignment, i.e. an expression, is formed, is described in Expressions (Page 119).

3.6.1.2 Value assignments with variables of an elementary data type

An expression with an elementary data type (Page 90) can be assigned to a variable when one of the following conditions is fulfilled:

- Expression and target variable have the same data type.

  Note the following information on the STRING data type (Page 114).
- The data type of the expression can be implicitly converted to the data type of the target variable (see Conversion of elementary data types (Page 141) and *Functions for the conversion of numerical data types and bit data types* in the *SIMOTION Basic Functions* Function Manual).

Examples

```
elemVar        := 3*3;
elemVar        := elemVar1;
```

See also

Value assignments with variables of a bit data type (Page 116)

3.6.1.3 Value assignments with variables of the STRING elementary data type

Assignments between variables of the STRING data type

There are no restrictions to assignments between variables of the STRING data type (character strings) that have been declared with different lengths. If the declared length of the target variable is shorter than the current length of the assigned character string, the character string is truncated to the length of the target variable.

Exception: The following applies for an in/out assignment (parameter transfer to an in/out parameter): The declared length of the assigned variable (actual parameter) must be greater than or equal to the declared length of the target variable (formal in/out parameter). See Parameter transfer to in/out parameters (Page 154).

See also Elementary data types (Page 90):

Examples:

```
string20 := 'ABCDEFG';
string20 := string30;
```

Access to elements of a string

The individual elements of a string can be addressed in the same way as the elements of an array [1..n]. These elements are converted implicitly to the elementary data type BYTE. In this way assignments between string elements and variables of the BYTE data type are possible.

Examples:

```
byteVar := string20[5];
string20[10] := byteVar;
```

The following special cases have to be taken into account:

1. When assigning a variable of the BYTE data type to a string element (e.g. `stringVar[n:] := byteVar`):
   - The string element to which the value is to be assigned lies outside of the declared length of the string:

     The string remains unchanged, TSI#ERRNO is set to 1.
   - The string element to which the value is to be assigned lies outside of the assigned length of the string (n > LEN(stringVar)), but within the declared length:

     The length of the string is adjusted, the string elements between LEN(stringvar) and n are set to $00.
2. When assigning a string element to a variable of the BYTE data type (`byteVar := stringVar[n:]`):
   - The string element to which the variable is to be assigned lies outside of the assigned length of the string (n > LEN(stringVar)):

     The variable is set to 16#00, TSI#ERRNO to 2.

Editing strings

Various system functions are available for the editing of strings, such as the joining of strings, replacement and extraction of characters, see *SIMOTION Basic Functions* Function Manual.

Converting between numbers and strings

Various system functions are available for the conversion between variables of numeric data types and strings, see Elementary data type conversion (Page 141) and the *SIMOTION Basic Functions* Function Manual.

3.6.1.4 Value assignments with variables of a bit data type

Access to individual bits of a bit data type variable

You can also access the individual bits of a variable of data type BYTE, WORD or DWORD:

- With standard functions (see *SIMOTION Basic Functions* Function Manual):

  You can read, write or invert any bit of a bit string with the functions _getBit, _setBit and _toggleBit.

  You can specify the number of the bit via a variable.

- With direct bit access:

  You can define the bit of the variable that you want to access as a constant, via a separate point behind the variable.

  You can only specify the number of the bit via a constant.

  To be able to use this option, you must activate the compiler option "Permit language extensions" (see Global compiler settings (Page 45) and Local compiler settings (Page 46)).

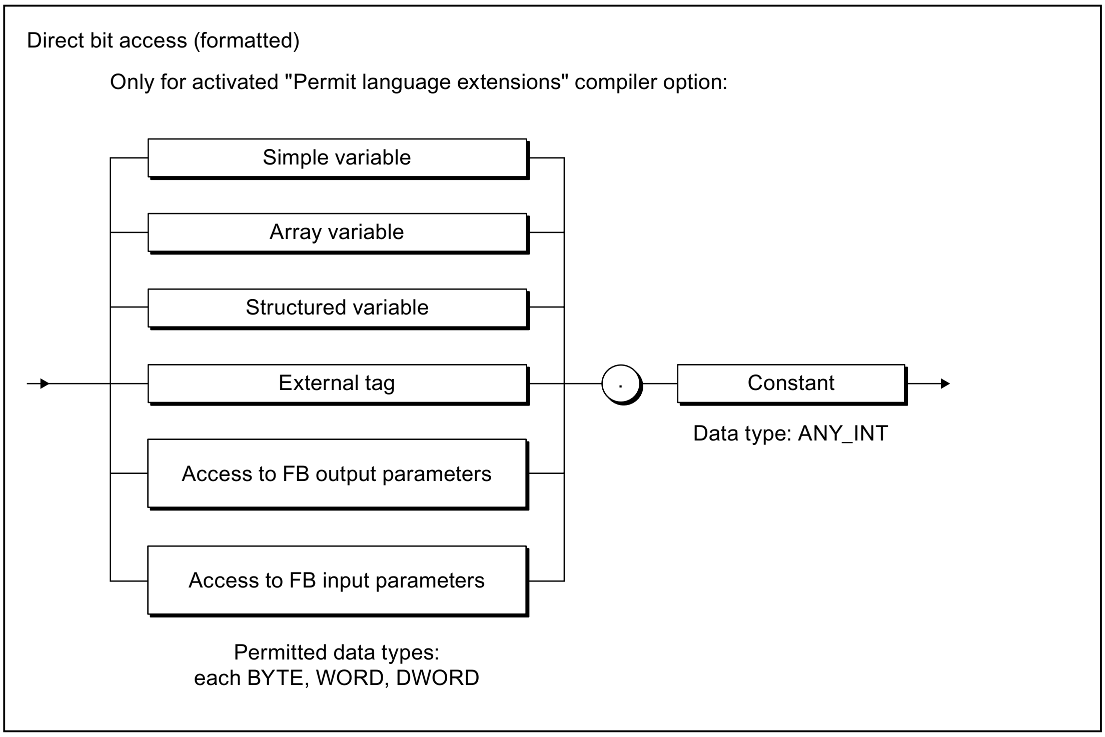

Direct bit access (formatted)
Only for activated "Permit language extensions" compiler option:
Simple variable
Array variable
Structured variable
External tag
Access to FB output parameters
Access to FB input parameters
Permitted data types:
each BYTE, WORD, DWORD
.
Constant
Data type: ANY_INT

Figure 3-20 Syntax: Direct bit access

Table 3-26 Example of direct bit access

```
// Only with compiler option "Permit language extensions"
FUNCTION f : VOID
    VAR CONSTANT
        BIT_7 : INT := 7;
    END_VAR
    VAR
        dw : DWORD;
        b: BOOL;
    END_VAR
    b := dw.BIT_7;  // Access to bit 7
    b := dw.3;      // Access to bit 3
//  b := dw.33;     // Compilation error;
                    // Bit 33 not permitted.
END_FUNCTION
```

NOTICE
The access to bits of an I/O variable or system variable can be interrupted by other tasks. There is therefore no guarantee of consistency.

Editing variables of the bit data types

You can:

1. Combine several variables of the same data type into one variable of a higher-level data type (e.g. two variables of the BYTE data type into one of the WORD data type). Various system functions are available for this, e.g. WORD_FROM_2BYTE.
2. Split one variable into several variables of a lower-level data type (e.g. one variable of the DWORD data type into four of the BYTE data type). Various system functions are available for this, e.g. DWORD_TO_4BYTE.
3. Rotate or shift the bits within a variable. The bit sting standard functions ROL, ROR, SHL and SHR are available for this.

These system functions and system function blocks are described in the *SIMOTION Basic Functions* Function Manual.

Logic operators

Variables of the bit data types can be combined with logic operators, see Logic expressions and bit-serial expressions (Page 127).

3.6.1.5 Value assignments with variables of the derived enumerator data type

Each expression and each variable of the derived enumerator data type (see also: Derived data type - Enumerator (Page 99)) can be assigned another variable of the same type.

```
type1 := BLUE;
```

3.6.1.6 Value assignments with variables of the derived ARRAY data type

An array consists of several dimensions and array elements, all of the same type (see also: Derived data type ARRAY (Page 97)).

There are various ways to assign arrays to variables. You can assign complete arrays, individual elements, or parts of arrays:

- A complete array can be assigned to another array if both the data types of the components and the array limits (the smallest and largest possible array indices) are the same. Valid assignments are:

```
array_1 := array_2;
```

- An individual array element is addressed by the array name followed by the index value in square brackets. An index must be an arithmetic expression of the data type SINT, USINT, INT, UINT or DINT.

```
elem1         := array [i];
array_1 [2]   := array_2 [5];
array [j]     := 14;
```

- A value assignment for a valid subarray can be obtained by omitting a pair of square brackets for each dimension of the array, starting at the right. This addresses a partial area of the array whose number of dimensions is equal to the number of remaining indices (see example below).

  Consequently, you can reference rows and individual components within a matrix but not closed columns (closed in the sense of from...to). Valid assignments are:

```
matrix1[i] := matrix2[k];
array1 :=  matrix2 [k];
```

3.6.1.7 Value assignments with variables of the derived STRUCT data type

Variables of a user-defined data type that contain STRUCT data type specifications are called structured variables (see also Derived data type STRUCT (structure) (Page 100)). They can either represent a complete structure or a component of this structure.

Valid parameters for a structure variable are:

```
struct1                     //Identifier for a structure
struct1.elem1               //Identifier for a structure component
struct1.array1              //Identifier of a simple array
                            //within a structure
struct1.array1[5]           //Identifier of an array component
                            //within a structure
```

There are two ways to assign structures to variables. You can reference complete structures or structure components:

- A complete structure can only be assigned to another structure if the data type and the name of both structure components match.

  A valid assignment is:

```
struct1 := struct2;
```

- You can assign a type-compatible variable, a type-compatible expression or another structure component to each structure component.

  Valid assignments are:

```
struct1.elem1              := Var1;
struct1.elem1              := 20;
struct1.elem1              := struct2.elem1;
struct1.array1             := struct2.array1;
struct1.array1[10]         := 100;
```

Note

You also use structured variables in the *FBInstanceName.OutputParameter* format, e.g. *myCircle.circumference* to access the output variables of a function block, i.e. the result of the function block. For more information about function blocks, see explanations in Defining functions (Page 148) and Defining function blocks (Page 149).

A further application of structured variables is to access TO variables and the variables of the basic system.

3.6.2 Expressions

An expression represents a value that is calculated when the program is compiled or executed. It consists of operands (e.g. constants, variables or function values) and operators (e.g. *, /, +, -).

The data types of the operands and the operators involved determine the expression type.

ST uses the following types of expression:

- Arithmetic expressions
- Relational expressions
- Logic expressions
- Bit-serial expressions

3.6.2.1 Result of an expression

The result of an expression can be:

- Assigned to a variable
- Used as a condition for a control statement
- Used as a parameter for a function or function block call.

Note

Expressions containing only the following elements can be used for variable initialization and index specification in ARRAY declarations (for initialization expressions – see Figure *Syntax: Constant expression* in Initialization of variables or data types (Page 107)):

- Constants
- Basic arithmetic operations
- Logic and relational operations
- Bit string standard functions

3.6.2.2 Interpretation order of an expression

The interpretation order of an expression depends on the following:

- The priority of the operators used,
- The left-to-right rule,
- The use of parentheses (for operators of the same priority).

Expressions are processed according to specific **rules**:

- Operators are executed according to priority
  (see table in Priority of operators (Page 129)).
- Operators of the same priority are executed from left to right.
- A minus symbol in front of an identifier denotes multiplication by -1.
- An arithmetic operator cannot be followed immediately by another.
  The expression *a * -b* is therefore invalid, but *a * (-b)* is allowed.
- Parentheses override the operator priority order, i.e. parentheses have the highest priority.
- Expressions in parentheses are treated as individual operands and are always evaluated first.
- The number of opening parentheses must equal the number of closing parentheses.
- Arithmetic operations cannot be used on characters or logic data. For this reason, expressions such as *(n<=0)* + *(n<0)* are invalid.

Table 3-27 Examples of expressions

```
testVar          // Operand
A AND (B)        // Logic expression
A AND (NOT B)    // Logic expression with negation
 (C) < (D)       // Relational expression
 3+3*4/2         // Arithmetic expression
```

3.6.3 Operands

Definition

Operands are objects which can be used to formulate expressions. Operands can be represented by the syntax diagram:

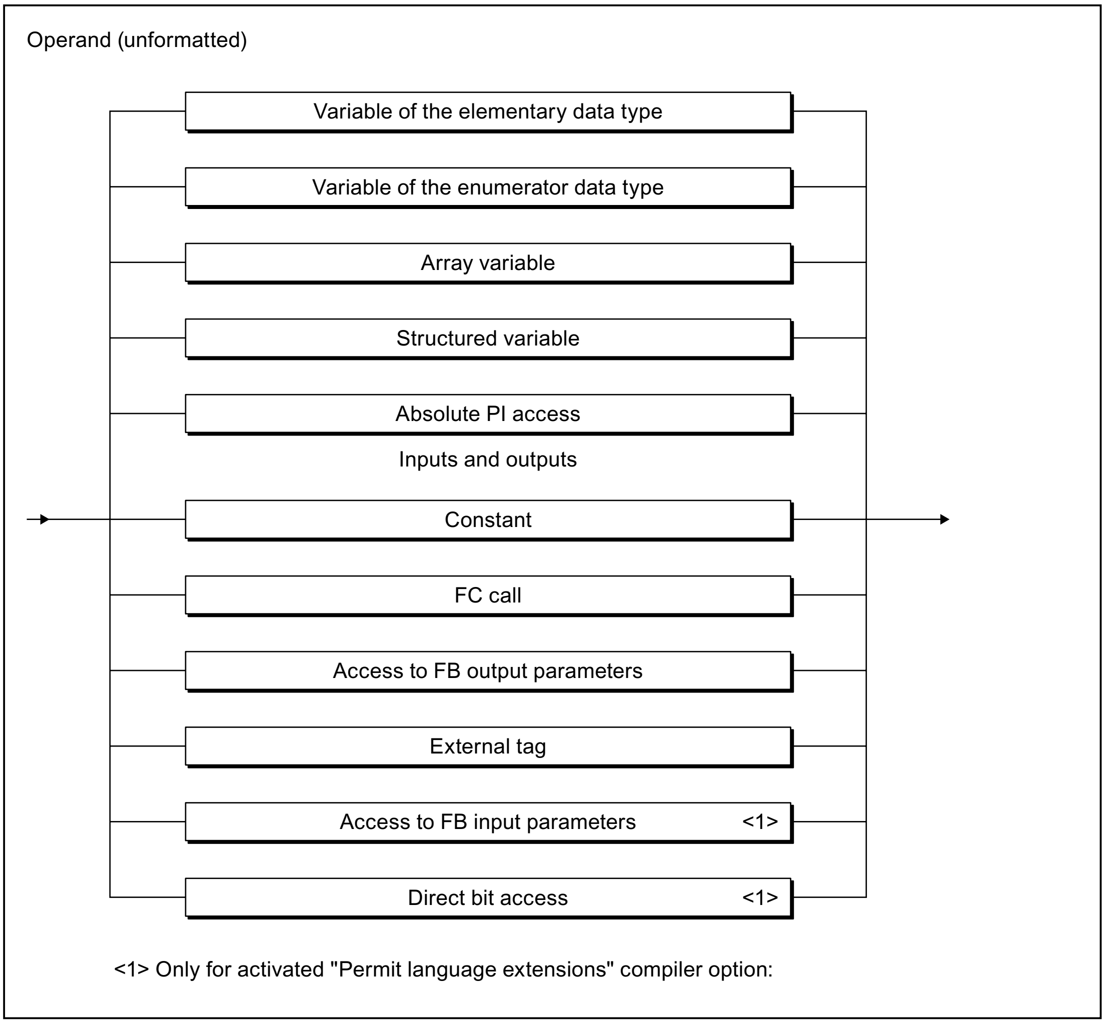

Operand (unformatted)
Variable of the elementary data type
Variable of the enumerator data type
Array variable
Structured variable
Absolute PI access
Inputs and outputs
Constant
FC call
Access to FB output parameters
External tag
Access to FB input parameters <1>
Direct bit access <1>
<1> Only for activated "Permit language extensions" compiler option:

Figure 3-21 Syntax: Operand

Table 3-28 Examples of operands

```
intVar
5
%I4.0
PI
NOT TRUE
axis1.motionStateData.actualVelocity
```

3.6.4 Arithmetic expressions

An arithmetic expression is an expression formed with arithmetical operators. These expressions allow numerical data types to be processed.

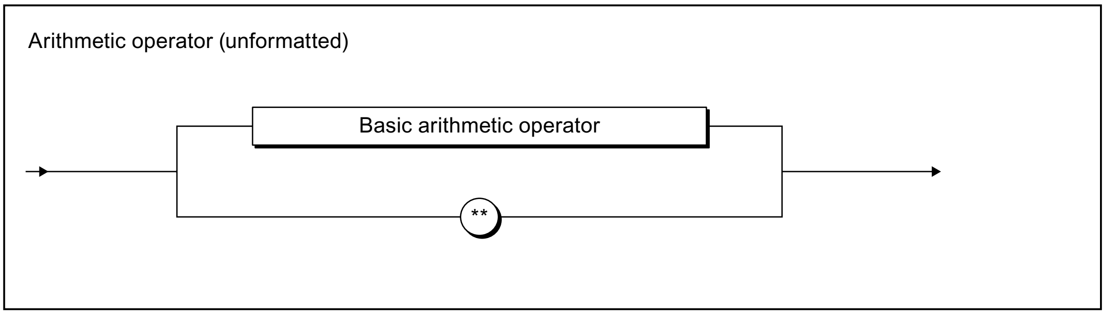


Figure 3-22 Syntax: Arithmetic operator

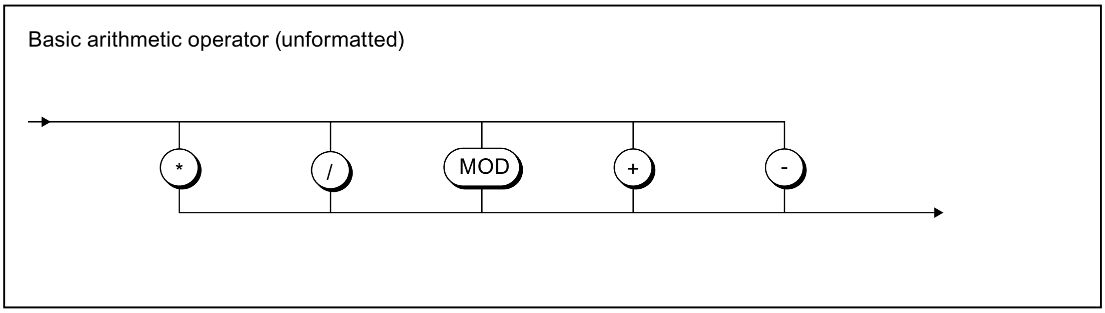


Figure 3-23 Syntax: Basic arithmetic operator

The following table shows for each arithmetic operation:

- The arithmetic operator
- The permitted data types of the operands
- The data type of the result.

Some of the General data types (Page 92) are used here.

Note

Further operations are possible with standard numeric functions, see *Standard numeric functions* in the *SIMOTION Basic Functions* Function Manual.

It is recommended to enclose negative numbers in parentheses, even in cases where it is not absolutely necessary, in order to enhance legibility.

The arithmetic operators are processed in accordance with their rank (Page 129).

Table 3-29 Arithmetic operators

Instruction	Operator	Data type		
		1st operand	2nd operand	Result[1]
Exponential (See also EXPT function)	**	ANY_REAL[2]	ANY_REAL	ANY_REAL[3]
Unary minus	–	ANY_NUM	(None)	ANY_NUM
Multiplication	*	ANY_NUM	ANY_NUM	ANY_NUM
		ANY_BIT[4]	ANY_BIT[4]	ANY_BIT
		TIME	ANY_NUM	TIME
Division	/	ANY_NUM	ANY_NUM[5]	ANY_NUM
		ANY_BIT[4]	ANY_BIT[4] [5]	ANY_BIT
		TIME	ANY_NUM[5]	TIME
		TIME	TIME[5]	UDINT
Modulo division	MOD	ANY_INT	ANY_INT[5]	ANY_INT
		ANY_BIT[4]	ANY_BIT[4] [5]	ANY_BIT
Addition	+	ANY_NUM	ANY_NUM	ANY_NUM
		ANY_BIT[4]	ANY_BIT[4]	ANY_BIT
		TIME	TIME	TIME[6]
		TOD	TIME	TOD[6)]
		DT	TIME	DT[7]
Subtraction	–	ANY_NUM	ANY_NUM	ANY_NUM
		ANY_BIT[4]	ANY_BIT[4]	ANY_BIT
		TIME	TIME	TIME
		TOD	TIME[8]	TOD
		DATE	DATE	TIME[9]
		TOD	TOD	TIME[9]
		DT	TIME	DT
		DT	DT	TIME[9]

[1] The data type of the result is determined by the most powerful data type of the operands.

[2] The first operand must be greater than zero.
Exceptions as of Version V4.1 of the SIMOTION Kernel:
– If the second operand is an integer, the first operand can be less than zero.
– If the second operand is positive, the first operand can be equal to zero.
The following applies up to Version V4.0 of the SIMOTION Kernel: If the first operand is equal to zero, an error message can be caught with ExecutionFaultTask.

[3] Data type of first operand.

[4] Other than BOOL data type. The calculation is made using the unsigned integer of the same bit width.

[5] The second operand must not be equal to zero.

[6] Addition, possibly with overflow.

[7] Addition with date correction.

[8] Restriction of TIME to TOD before calculation.

[9] These operations are based on the modulo of the maximum value of the TIME data type.

Note

If the limits of the value range are exceeded in operations with variables of the general ANY_REAL data type, the result contains the equivalent bit pattern according to IEEE 754.

In order to establish whether the value range was exceeded in the operation, you can verify the result using the function *_finite* (see *SIMOTION Basic Functions* Function Manual).

3.6.4.1 Examples of arithmetic expressions

Examples of arithmetic expressions with numbers

Assuming that *i* and *j* are integer variables (e.g. of data type INT) with the values of 11 and -3 respectively, some example integer expressions and their corresponding values are presented below:

Expression	Value
`i + j`	`8`
`i - j`	`14`
`i * j`	`-33`
`i MOD j`	`-2`
`i / j`	`-3`

Examples of valid arithmetic expressions with time specifications

Assume the following variables:

Variables	Content	Data type
`t1`	`T#1D_1H_1M_1S_1MS`	`TIME`
`t2`	`T#2D_2H_2M_2S_2MS`	`TIME`
`d1`	`D#2004-01-11`	`DATE`
`d2`	`D#2004-02-12`	`DATE`
`tod1`	`TOD#11:11:11.11`	`TIME_OF_DAY`
`tod2`	`TOD#12:12:12.12`	`TIME_OF_DAY`
`dt1`	`DT#2004-01-11-11:11:11.11`	`DATE_AND_TIME`
`dt2`	`DT#2004-02-12-12:12:12.12`	`DATE_AND_TIME`

Some expressions with these variables and their values are shown in the example.

Expression	Value
`t1 + t2`	`T#3D_3H_3M_3S_3MS`
`dt1 + t1`	`DT#2004-01-12-12:12:12.111`
`t1 - t2`	`T#48D_16H_1M_46S_295MS`
`t1 * 2`	`T#2D_2H_2M_2S_2MS`
`t1 / 2`	`T#12H_30M_30S_500MS`
`DATE_AND_TIME_TO_TIME_OF_DAY(dt1)`	`TOD#11:11:11.110`
`DATE_AND_TIME_TO_DATE(dt1)`	`D#2004-01-11`

3.6.5 Relational expressions

Definition

A relational expression is an expression of the BOOL data type formed with relational operators (see figure).

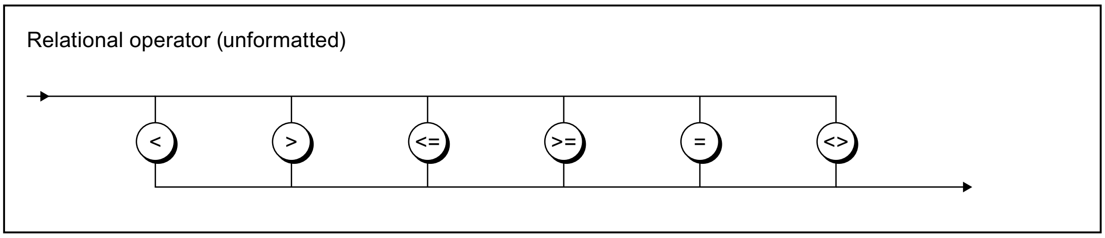


Figure 3-24 Syntax: Relational operators

Relational operators compare the values of two operands (see table) and return a Boolean value as result.

1st Operand Operator 2nd Operand -> Boolean value

Table 3-30 Meaning of relational operators

Operator	Meaning
>	1. operand is **greater than** the 2nd operand
<	1. operand is **less than** the 2nd operand
>=	1. operand is **greater than or equal to** the 2nd operand
<=	1. operand is **less than or equal to** the 2nd operand
=	1. operand is **equal to** the 2nd operand
<>	1. operand is **not equal to** the 2nd operand

The result of the relational expression is:

- 1 (TRUE), when the comparison is satisfied
- 0 (FALSE), when the comparison is not satisfied.

The following table shows permissible combinations of the data types for the two operands and relational operators.

Table 3-31 Relational expressions: Permissible combinations of the data types and relational operators

Data type		Permissible relational operators
1. Operand	2. Operand	
ANY_NUM	ANY_NUM[1]	<, >, <=, >=, =, <>
ANY_BIT	ANY_BIT	<, >, <=, >=, =, <>
DATE	DATE	<, >, <=, >=, =, <>
TIME_OF_DAY (TOD)	TIME_OF_DAY (TOD)	<, >, <=, >=, =, <>
DATE_AND_TIME (DT)	DATE_AND_TIME (DT)	<, >, <=, >=, =, <>
TIME	TIME	<, >, <=, >=, =, <>
STRING	STRING[2]	<, >, <=, >=, =, <>
Enumerator data type	Enumerator data type[3]	=, <>
ARRAY	ARRAY[3]	=, <>
Structure (STRUCT)	Structure (STRUCT)[3]	=, <>

[1] Both operands must be converted to the most powerful data type through implicit conversion (see Elementary data type conversion (Page 141) and *Functions for the conversion of numerical data types and bit data types* in the *SIMOTION Basic Functions* Function Manual).

[2] Variables of the STRING data type can be compared irrespective of the declared length of the string.
To compare two variables of the STRING data type with different lengths, the shorter character string is expanded to the length of the longer character string by inserting $00 on the right-hand side. The comparison starts from left to right and is based on the ASCII code of the respective characters.
Example: 'ABC' < 'AZ' < 'Z' < 'abc' < 'az' < 'z'.

[3] Data type of first operand.

Relational expressions and variables or constants of the BOOL data type can be combined into logic expressions with logic operators (see Logic expressions and bit-serial expressions (Page 127)). This enables the implementation of queries such as *If a < b and b < c, then ...*.

NOTICE

Relational operators have a higher priority than logic operators in an expression (see Priority of operators (Page 129)). Therefore the operands of a relational expression must be placed in brackets if they themselves are logic expressions or bit-serial expressions.

Note that errors can occur when comparing REAL or LREAL variables (also the corresponding system variables, e.g. axis position).

Table 3-32 Examples of relational expressions

```
IF A = 2 THEN
     //...
END_IF;
var_1 := B < C;          // var_1 of BOOL data type
IF D < E OR var_2 THEN  // var_2 of BOOL data type
     // ...
END_IF;
```

3.6.6 Logic expressions and bit-serial expressions

Definition

With the logic operators AND, &, XOR, and OR, it is possible to combine operands and expressions of the general data type ANY_BIT (BOOL, BYTE, WORD, or DWORD).

With the logic operator NOT it is possible to negate operands and expressions of data type ANY_BIT.

The table provides information about the available operators:

Table 3-33 Logic operators

Instruction	Operator	1. Operand	2. Operand	Result[1]
Negation	NOT	ANY_BIT	-	ANY_BIT
Conjunction	AND or &	ANY_BIT	ANY_BIT	ANY_BIT
Exclusive disjunction	XOR	ANY_BIT	ANY_BIT	ANY_BIT
Disjunction	OR	ANY_BIT	ANY_BIT	ANY_BIT

[1] The data type of the result is determined by the most powerful data type of the operands.

The expression is designated

- a **logic expression**, if only operands of data type BOOL are used.

  The operators have the effect on the operands stated in the following truth table.

  The result of a logic expression is 1 (TRUE) or 0 (FALSE).

- a **bit-serial expression**, if operands of data type BYTE, WORD, or DWORD are used.

  The operators have the effect on individual bits of the operands stated in the following truth table.

Table 3-34 Truth table of the logic operators

Operands (data type BOOL)		Result (data type BOOL)				
a	b	NOT a	NOT b	a AND b a & b	a XOR b	a OR b
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	1

Examples

Table 3-35 Logic expressions

Expression (let n = 10)	Value
`(n>0) AND (n<20)`	`TRUE`
`(n>0) AND (n<5)`	`FALSE`
`(n>0) OR (n<5)`	`TRUE`
`(n>0) XOR (n<20)`	`FALSE`
`NOT ((n>0) AND n<20))`	`FALSE`

Table 3-36 Bit-serial expressions

Expression	Value
`2#01010101 AND 2#11110000`	`2#01010000`
`2#01010101 OR 2#11110000`	`2#11110101`
`2#01010101 XOR 2#11110000`	`2#10100101`
`NOT 2#01010101`	`2#10101010`

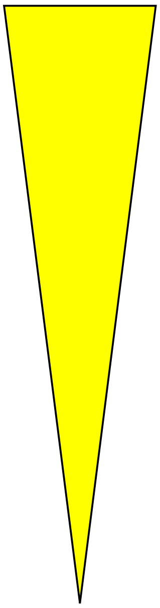
Expression in query (let value1 be 2#01, let value2 be 2#11)

```
IF (value1 AND value2) = 2#01 THEN...
```

Condition returns TRUE, because bit-serial expression returns 2#01.

3.6.7 Priority of operators

Some general rules for the formulation of expressions were described in Expressions (Page 119). The table shows you the priority of the individual operators within an expression.

Instruction	Symbol	Priority
Parentheses	(Expression)	Highest
Function evaluation	Identifier (argument list) e.g. LN(a), EXPT (a,b) etc.	
Negation Complement	– NOT	
Exponentiation	**	
Multiplication Division Modulo	* / MOD	
Addition Subtraction	+ –	
Comparison	<, >, <=, >=	
Equal Not equal	= <>	
Boolean AND	&, AND	
Boolean EXCLUSIVE OR	XOR	
Boolean OR	OR	Lowest

3.7 Control statements

Few source file sections can be programmed such that all statements are executed in sequence from start to end. Usually, some statements will be executed only if a condition is true (alternatives) and some will be executed repeatedly (loops). Program control statements within a source file section are the means for accomplishing this.

3.7.1 IF statement

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

The specified logic expressions are evaluated when the conditional statement is executed. If the value of an expression is TRUE, the condition is fulfilled, if the value is FALSE, it is not fulfilled.

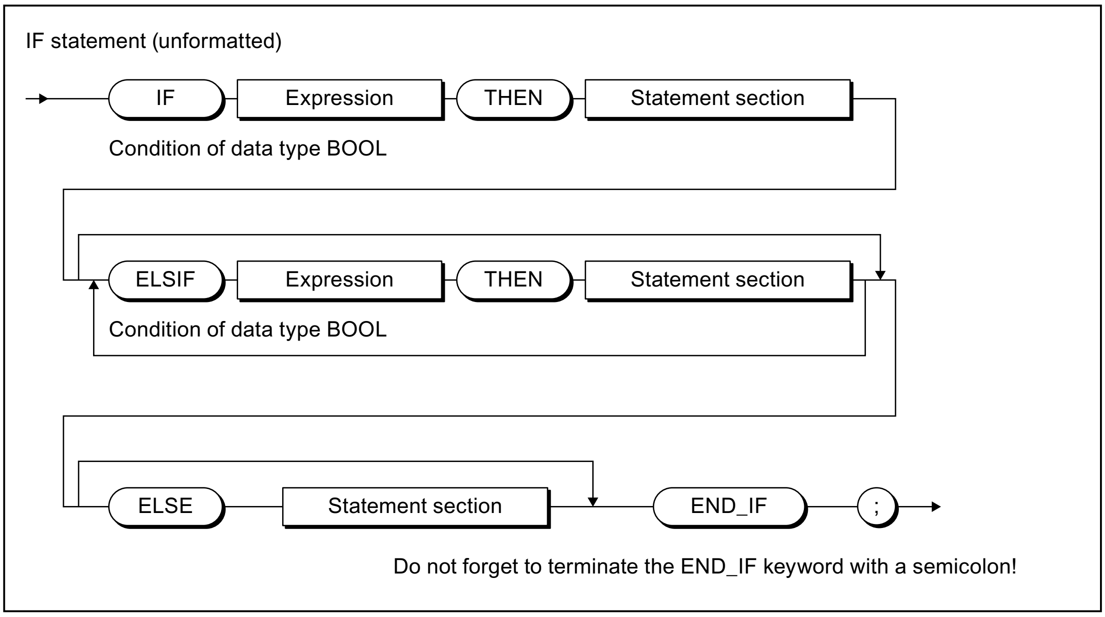

IF statement (unformatted)
IF
Expression
THEN
Statement section
Condition of data type BOOL
ELSIF
Expression
THEN
Statement section
Condition of data type BOOL
ELSE
Statement section
END_IF
;
Do not forget to terminate the END_IF keyword with a semicolon!

Figure 3-25 Syntax: IF statement

The IF statement is processed according to the following rules:

1. If the value of the first expression is TRUE, the statement section after the THEN is executed.

   The program is subsequently resumed after the END_IF.

2. If the value of the first expression is FALSE, the expressions in the ELSIF branches are evaluated. If a Boolean expression in one of the ELSIF branches is TRUE, the statement section following THEN is executed.

   The program is subsequently resumed after the END_IF.

3. If none of the Boolean expressions in the ELSIF branches is TRUE, the sequence of statements after the ELSE is executed (or, if there is no ELSE branch, no further statements are executed).

   The program is subsequently resumed after the END_IF.

Any number of ELSIF statements may be programmed.

Note that there may not be any ELSIF branches and/or ELSE branch. This is interpreted in the same way as if the branches existed with no statements.

Note

An advantage of using one or more ELSIF branches rather than a sequence of IF statements is that the logic expressions following a valid expression are no longer evaluated. This helps to reduce the processing time required for the program and to prevent execution of unwanted program routines.

Table 3-37 Examples of the IF statement

```
IF A=B THEN
    n:= 0;
END_IF;

IF temperature < 5.0 THEN
    %Q0.0 := TRUE;
ELSIF temperature > 10.0 THEN
    %Q0.2 := TRUE;
  ELSE
    %Q0.1 := TRUE;
END_IF;
```

3.7.2 CASE statement

The CASE statement is used to select 1 of n program sections.

This selection determines a selection expression (selector):

- Expression of general data type ANY_INT
- Variable of an enumeration data type (enumerator)

The selection is made from a list of values (value list), whereby a section of the program is assigned to each value or group of values.

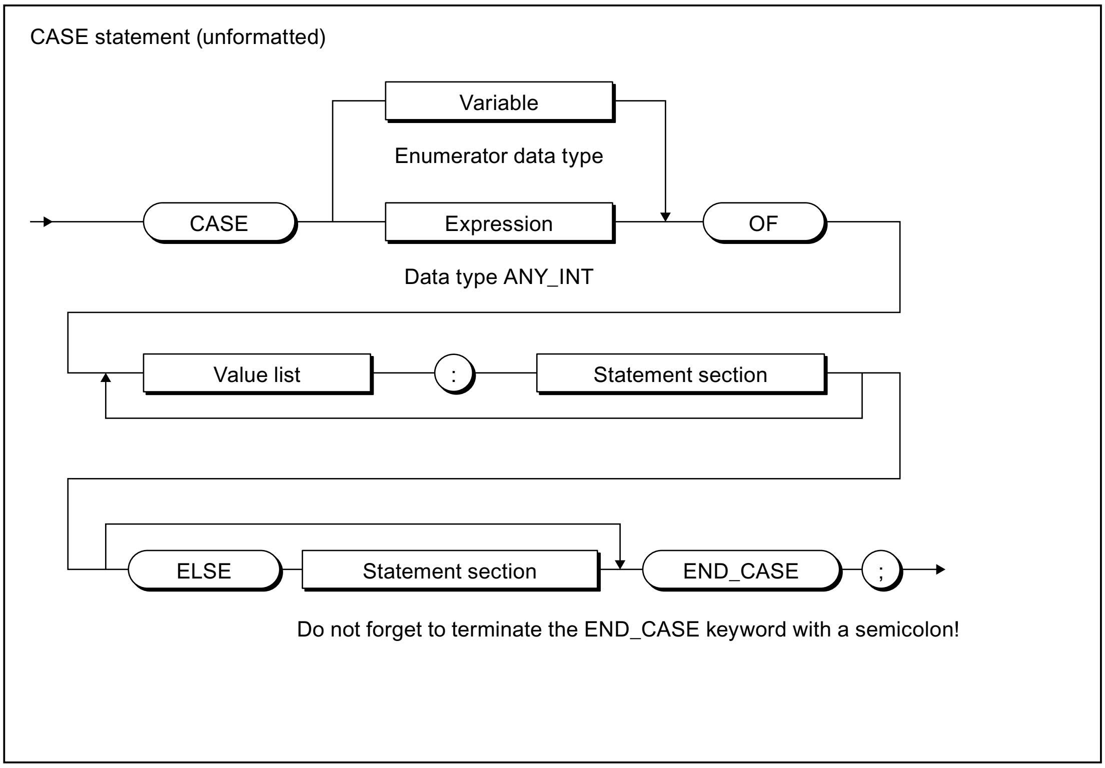


Figure 3-26 Syntax: CASE statement

The CASE statement is processed according to the following rules:

1. The selection expression (selector) is calculated. It must return a value of general data type ANY_INT (integer) or an enumerator data type.
2. Then a check is performed to determine whether the selector value is contained in the value list. Each value in the list represents one of the allowed values for the selection expression.
3. If a match is found, the program section assigned in the list is executed.
4. The ELSE branch is optional. It is executed if no match is found.
5. If the ELSE branch is missing and no match is found, the program is resumed after END_CASE.

The value list contains the allowed values for the selection expression.

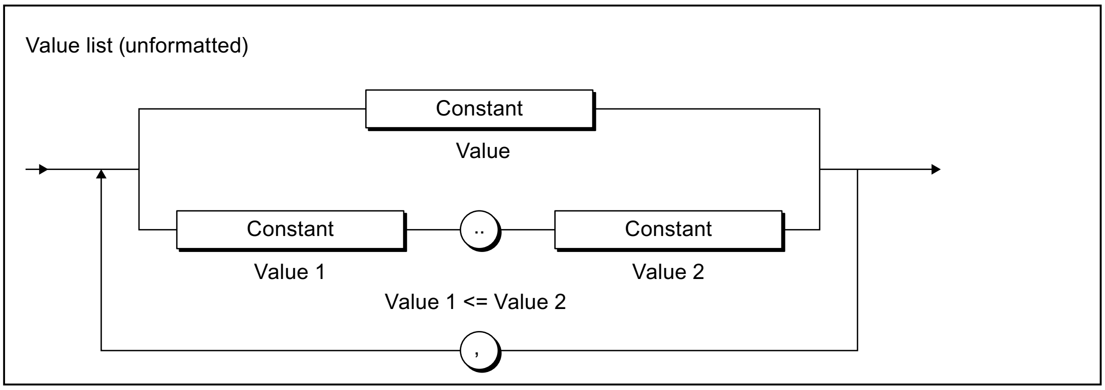


Figure 3-27 Syntax: Value list

Note the following when formulating the value list:

- Each value list can begin with a constant (*value*), a constant list (*value1, value2, value3, etc.*) or a constant range (*value1 to value2*).
- Values in the value list must be integer values or constants/elements of the enumeration data type of the selector.

Note

A value should only occur once in the value lists of a CASE statement.

In the event of multiple occurrence of a value, the compiler will issue an alarm, and only the section of the statement corresponding to the value list in which the value occurred first is executed.

The following example illustrates the use of the CASE statement.

Table 3-38 Examples of the CASE statement

```
CASE intVar OF
    1    : a := 1;
    2,3  : b := 1;
    4..9 : c := 1; d:=2;
  ELSE
    e := 5;
END_CASE;
```

3.7.3 FOR statement

A FOR statement or a repeat statement executes a series of statements in a loop, whereby values are assigned to a variable (a count variable) on each pass. The count variable must be a local variable of type SINT, INT or DINT.

The definition of a loop with FOR includes the specification of a start and end value. Both variables must be the same data type as the count variable.

Note

You use the FOR statement when the number of loop passes is known at the programming stage. If the number of passes is not known, the WHILE or REPEAT statement is more suitable (see WHILE statement (Page 136) and REPEAT statement (Page 137)).

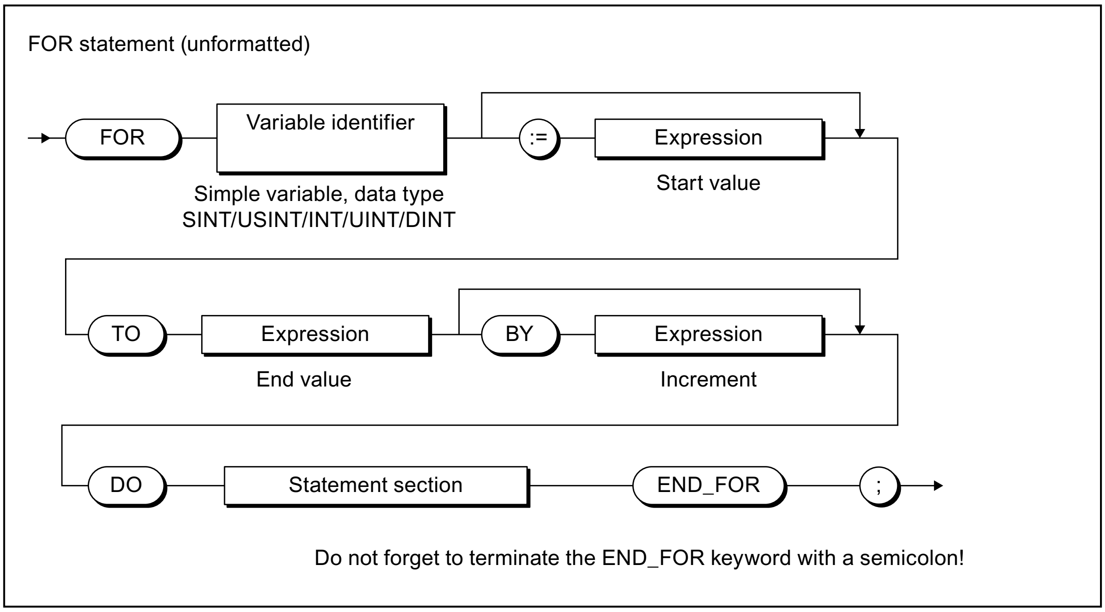

FOR statement (unformatted)
FOR
Variable identifier
Simple variable, data type SINT/USINT/INT/UINT/DINT
:=
Expression
Start value
TO
Expression
End value
BY
Expression
Increment
DO
Statement section
END_FOR
;
Do not forget to terminate the END_FOR keyword with a semicolon!

Figure 3-28 Syntax: FOR statement

3.7.3.1 Processing of the FOR statement

The FOR statement is processed according to the following rules:

1. At the start of the loop, the count variable is set to the **start value** and is increased (positive increment) or decreased (negative increment) by the specified increment after each loop pass until the **end value** is reached. After the first loop pass, the start value is known as the **current value**.
2. On each pass, the system checks whether the following conditions are true:
   - **Start value or current value <= end value** (for **positive increment)** or
   - **Start value or current value >= end value** (for **negative increment)**

   If the condition is fulfilled, the sequence of statements is executed.

   If the condition is not fulfilled, the loop and, thus, the sequence of statements is skipped and the program is resumed after END_FOR.

3. If the FOR loop is not executed due to Step 2, the count variable retains the current value.

3.7.3.2 Rules for the FOR statement

The following rules apply to the FOR statement:

- The *BY [increment]* specification can be omitted. If no increment is specified, the default is +1.
- The start value, end value and increment are expressions (see Expressions (Page 119)). The expression is evaluated once at the beginning of the FOR statement.
- If the start value and end value are of the DINT data type, the value of (end value - start value) must be less than the maximum value range of the double integer, that is, less than 2**31-1.
- Only the first selection statement for which the selector is true is executed.
- The count variable contains the value which triggers the loop exit, i.e. it is incremented before the loop is exited.
- You are not allowed to change the end value and increment value during the execution of the loop.

3.7.3.3 Example of the FOR statement

Table 3-39 Example of the FOR statement

```
FOR k := 1 TO 10 BY 2 DO
    l:=l+1;
    // ...
END_FOR;
```

3.7.4 WHILE statement

The WHILE statement allows a sequence of statements to be executed repeatedly under the control of an iteration condition. The iteration condition is formulated in accordance with the rules for a logic expression.

Note

You use the WHILE statement when the number of loop passes is not known at the programming stage. If the number of passes is known, the FOR statement is more suitable (see FOR statement (Page 134)).

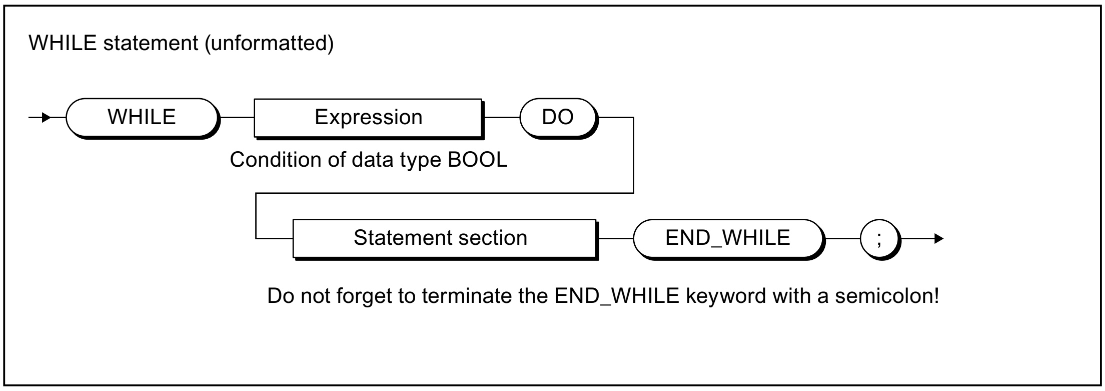

WHILE statement (unformatted)
WHILE
Expression
DO
Condition of data type BOOL
Statement section
END_WHILE
;
Do not forget to terminate the END_WHILE keyword with a semicolon!

Figure 3-29 Syntax: WHILE statement

The statement section after DO is repeated until the iteration condition has the value TRUE.

The WHILE statement is processed according to the following rules:

1. The iteration condition is evaluated each time **before** the statement section is executed.
2. If the value is TRUE, the statement section is executed.
3. If the value is FALSE, the WHILE statement is terminated (this can occur the first time the condition is evaluated) and the program is resumed after END_WHILE.

Table 3-40 Example of the WHILE statement

```
WHILE Index <= 50 DO
      Index:= Index + 2;
END_WHILE;
```

3.7.5 REPEAT statement

A REPEAT statement causes a sequence of statements programmed between REPEAT and UNTIL to be executed repeatedly until a termination condition is true. The termination condition is formulated in accordance with the rules for a logic expression.

Note

You use the REPEAT statement when the number of loop passes is not known at the programming stage. If the number of passes is known, the FOR statement is more suitable (see FOR statement (Page 134)).

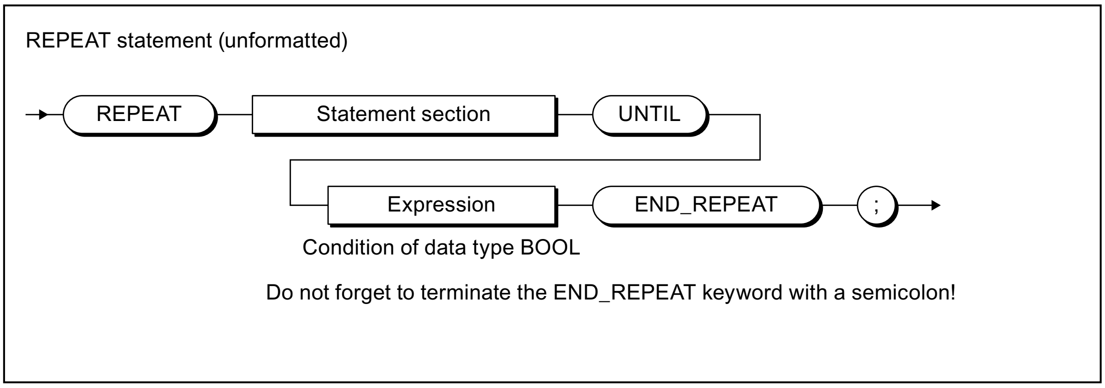

REPEAT statement (unformatted)
REPEAT
Statement section
UNTIL
Expression
END_REPEAT
;
Condition of data type BOOL
Do not forget to terminate the END_REPEAT keyword with a semicolon!

Figure 3-30 Syntax: REPEAT statement

The condition is checked **after** the statement section is executed. That means the statement section is executed at least once, even if the termination condition is true at the start.

The REPEAT statement is processed according to the following rules:

1. The iteration condition is evaluated each time **after** the statement section is executed.
2. If the value is FALSE, the statement section is executed again.
3. If the value is TRUE, execution of the REPEAT statement is terminated and program execution is resumed after END_REPEAT.

Table 3-41 Example of the REPEAT statement

```
Index:= 1;
REPEAT
    Index:= Index + 2;
UNTIL Index > 50
END_REPEAT;
```

3.7.6 EXIT statement

An EXIT statement is used to exit a loop (FOR, WHILE or REPEAT loop) at any point, irrespective of whether the termination condition is true or false.

This statement has the effect of jumping directly out of the loop immediately surrounding the EXIT statement.

The program resumes after the end of the loop (e.g. after END_FOR).

Table 3-42 Example of the EXIT statement

```
Index:= 1;
FOR Index := 1 to 51 BY 2 DO
    IF %I0.0 THEN
        EXIT;
    END_IF;
END_FOR;
// The following value assignment is made after the execution of EXIT
// or after the regular end of the FOR loop
// For the execution:
Index_find := Index_2;
```

3.7.7 RETURN statement

A RETURN statement causes termination of the POU currently being processed (program, function, function block).

When a function or a function block is terminated, program execution continues in the higher-level POU after the position where the function or function block was called.

Table 3-43 Example of the RETURN statement

```
Index:= 1;
FOR Index := 1 to 51 BY 2 DO
    IF %I0.0 THEN
        RETURN;
    END_IF;
END_FOR;
// The following value assignment is made after the regular end
// of the FOR loop for the execution, however, not after the execution
// of RETURN:
Index_find := Index_2;
```

3.7.8 WAITFORCONDITION statement

You can use the WAITFORCONDITION statement to wait for a programmable event or condition in a MotionTask. The statement suspends execution of the calling MotionTask until the condition is true. You program this condition in an Expression (Page 166). More information about the WAITFORCONDITION and expressions in this regard is contained in the *SIMOTION Motion Control Basic Functions* Function Manual.

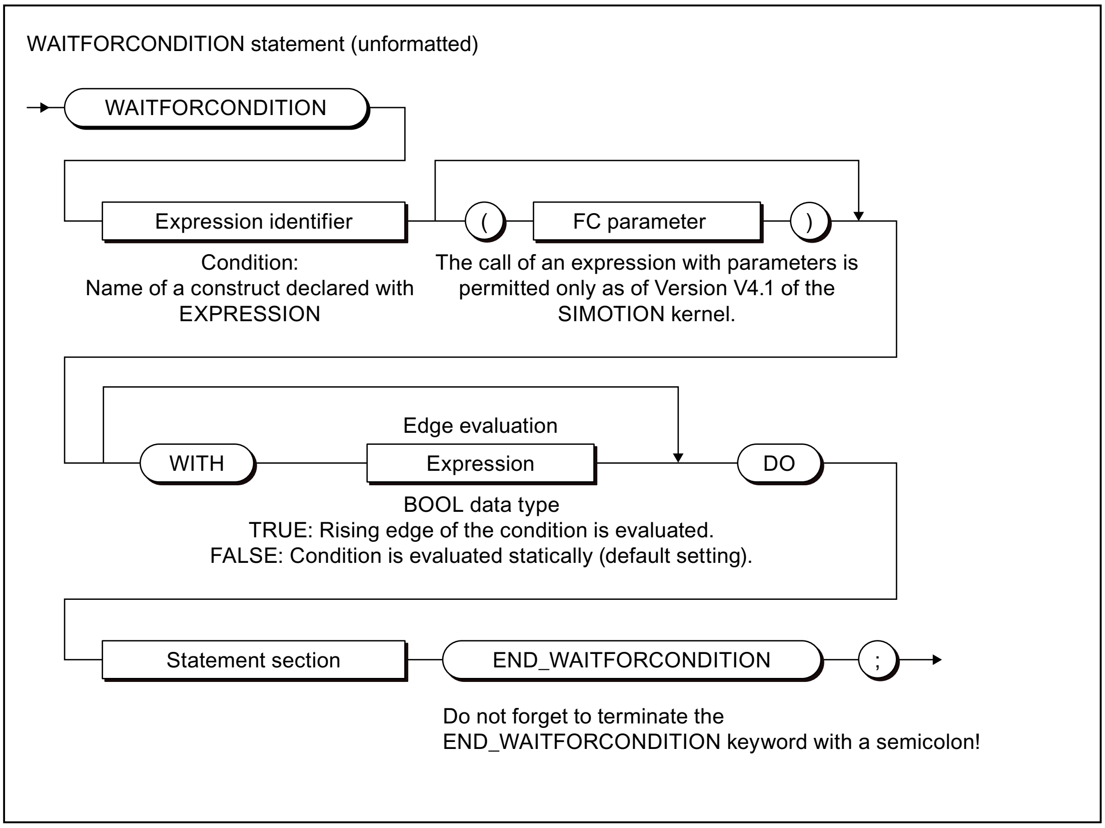

WAITFORCONDITION statement (unformatted)
WAITFORCONDITION
Expression identifier
(
FC parameter
)
Condition:
Name of a construct declared with
EXPRESSION
The call of an expression with parameters is
permitted only as of Version V4.1 of the
SIMOTION kernel.
Edge evaluation
WITH
Expression
DO
BOOL data type
TRUE: Rising edge of the condition is evaluated.
FALSE: Condition is evaluated statically (default setting).
Statement section
END_WAITFORCONDITION
;
Do not forget to terminate the
END_WAITFORCONDITION keyword with a semicolon!

Figure 3-31 Syntax: WAITFORCONDITION statement

Expression identifier is a construct declared with EXPRESSION; its value defines (together with *WITH edge evaluation*, if necessary) whether the condition is considered as been satisfied.

The *WITH edge evaluation* sequence is optional. *Edge evaluation* is an expression of data type BOOL; it determines how the value of *expression identifier* is interpreted:

- *Edge evaluation* = TRUE: The rising edge of *expression identifier* is interpreted; i.e. the condition is satisfied when the value of *expression identifier*changes from FALSE to TRUE.
- *Edge evaluation* = FALSE: The static value of *expression identifier* is interpreted; i.e. the condition is satisfied when the value of *expression identifier* is TRUE.

If *WITH edge evaluation* is not specified, the default setting is FALSE, i.e. the static value of *expression identifier* is evaluated.

The statement section must contain at least one statement (empty statements also possible).

Table 3-44 Example of the WAITFORCONDITION statement

```
// ...
// Call the statement with name of expression
WAITFORCONDITION myExpression WITH TRUE DO
// Here, at least one statement will be executed with higher priority,
e.g.
    %Q0.0 := TRUE;
END_WAITFORCONDITION;
// ...
```

For a complete example, refer to the description for the Expression (Page 166).

3.7.9 GOTO statement

The GOTO statement causes a jump to the jump label specified in the statement (see Jump statement and label (Page 250))

You program jump statements with the GOTO statement and specify the jump label to which you want to jump. Jumps are only permitted within a POU.

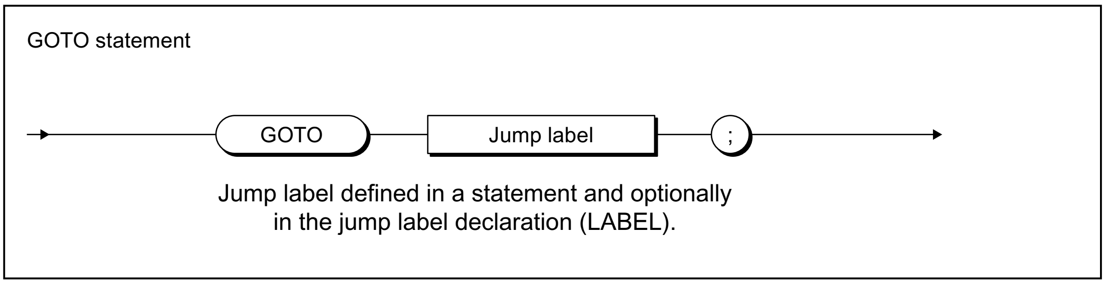

GOTO statement
GOTO
Jump label
;
Jump label defined in a statement and optionally in the jump label declaration (LABEL).

Figure 3-32 Syntax: GOTO statement

Note

You should only use the GOTO statement in special circumstances (for example, for troubleshooting). It should not be used at all according to the rules for structured programming.

Jumps are only permitted within a POU.

The following jumps are illegal:

- Jumps to subordinate control structures (WHILE, FOR, etc.)
- Jumps from a WAITFORCONDITION structure
- Jumps within CASE statements

Jump labels can only be declared in the POU in which they are used. If jump labels are declared, only the declared jump labels may be used.

3.8 Data type conversions

This section describes how you can implicitly and explicitly convert between elementary data types. It also contains an overview of the additional conversion possibilities.

3.8.1 Elementary data type conversion

The table presents an overview of the conversion options between numerical data types and bit data types. The following are distinct conversion options:

- Implicit conversion: Conversion is automatic when different data types are used in an expression or when values are assigned by the compiler.
- Explicit conversion: Conversion is carried out when the user calls a conversion function (see *SIMOTION Basic Functions* Function Manual).

Table 3-45 Type conversion of numeric data types and bit data types

Source data type	Target data type												
	BOOL	BYTE	WORD	DWORD	USINT	UINT	UDINT	SINT	INT	DINT	REAL	LREAL	STRING
BOOL	–	Im/Ex	Im/Ex	Im/Ex	Val	Val	Val	Val	Val	Val	Val	Val	–
BYTE	Ex	–	Im/Ex	Im/Ex	Ex	Ex	Ex	Ex	Ex	Ex	Val	Val	Elem
WORD	Ex	Ex	–	Im/Ex	Ex	Ex	Ex	Ex	Ex	Ex	Val	Val	–
DWORD	Ex	Ex	Ex	–	Ex	Ex	Ex	Ex	Ex	Ex	Ex/Val	Val	–
USINT	Val	Ex	Ex	Ex	–	Im/Ex	Im/Ex	Ex	Im/Ex	Im/Ex	Im/Ex	Im/Ex	–
UINT	Val	Ex	Ex	Ex	Ex	–	Im/Ex	Ex	Ex	Im/Ex	Im/Ex	Im/Ex	–
UDINT	Val	Ex	Ex	Ex	Ex	Ex	–	Ex	Ex	Ex	Ex	Ex	Ex
SINT	Val	Ex	Ex	Ex	Ex	Ex	Ex	–	Im/Ex	Im/Ex	Im/Ex	Im/Ex	–
INT	Val	Ex	Ex	Ex	Ex	Ex	Ex	Ex	–	Im/Ex	Im/Ex	Im/Ex	–
DINT	Val	Ex	Ex	Ex	Ex	Ex	Ex	Ex	Ex	–	Ex	Im/Ex	Ex
REAL	Val	Val	Val	Ex/Val	Ex	Ex	Ex	Ex	Ex	Ex	–	Im/Ex	Ex
LREAL	Val	Val	Val	Val	Ex	Ex	Ex	Ex	Ex	Ex	Ex	–	Ex
STRING	–	Elem	–	–	–	–	Ex	–	–	Ex	Ex	Ex	–

Im: Implicit data type conversion possible

Ex: Explicit data type conversion possible using the Quelldatentyp_TO_Zieldatentyp type conversion function

Val: Explicit data type conversion possible using the Quelldatentyp_VALUE_TO_Zieldatentyp type conversion function

Elem: Implicit data type conversion with an element of the STRING data type

For information on conversion functions for date and time data types: Please refer to the *SIMOTION Basic Functions* Function Manual.

3.8.1.1 Implicit data type conversions

Implicit type conversion is always possible if an enlargement of the value range does not cause any value loss, e.g. from REAL to LREAL or from INT to REAL. The result is always defined.

The following figure provides a graphics-based view of all implicit type conversion chains. Each stage in the type conversion chain - reading from left to right or from top to bottom - always represents an enlargement of the value range.

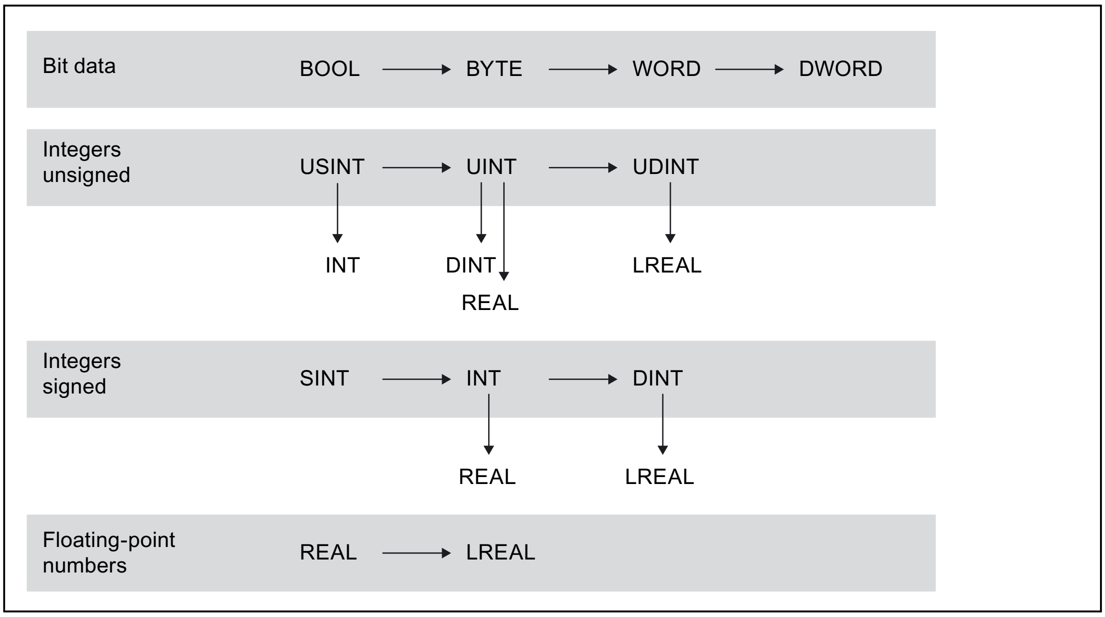


Figure 3-33 Implicit type conversion chains (one or more levels from left to right or one level from top to bottom)

The following implicit type conversions are supported:

1. Horizontally (from left to right) over one or more levels (e.g. USINT to UDINT)
2. Vertically (from top to bottom) over one level (e.g. UINT to REAL)

The implicit type conversions can be combined in the following order (e.g. INT to LREAL).

All other type conversions cannot be performed implicitly (e.g. UDINT to REAL), that is, you must use an explicit function (see *SIMOTION Basic Functions* Function Manual).

Note

In arithmetic expressions, the result is always calculated in the largest number format contained in the expression.

A value can only be assigned to the expression if:

- The calculated expression and the variable to be assigned are of the same data type.
- The data type of the calculated expression can be implicitly converted to the data type of the variable to be assigned.

For more information on this error source and its solution: Please refer to the *SIMOTION Basic Functions* Function Manual.

Table 3-46 Example of data types in expressions and value assignments

```
VAR
    usint_var  : USINT;
    real_var   : REAL;
    byte_var   : BYTE;
    string_var : STRING[80] := 'example for string';
END_VAR

usint_var := 234 / 10;       // Expression data type: USINT
                             // Result = 23

real_var  := 234 / 10;       // Expression data type: USINT
                             // Implicit conversion possible
                             // Result = 23.0

usint_var := 234 / SINT#10;  // Expression data type: INT
                             // Implicit conversion and
                             // value assignment not possible

real_var  := 234 / 10.0;     // Expression data type: REAL
                             // Result = 23.4

usint_var := 234 / 10.0;     // Expression data type: REAL
                             // Implicit conversion and
                             // value assignment not possible

byte_var  := string_var[5];  // Implicit conversion possible
                             // Result = 16#70 ('p')

string_var[10] := byte_var;  // Implicit conversion possible
                             // Result = 'example fpr string'
```

Note

If applicable, specify the data type explicitly for numbers (e.g. UINT#127, if the number 127 is to be of data type UINT instead of USINT).

3.8.1.2 Explicit data type conversions

Explicit conversion is always required if information could be lost, for example, if the value range is decreased or the accuracy is reduced, as is the case for conversion from LREAL to REAL.

The conversion functions for numeric data types and bit data types are listed in the *SIMOTION Basic Functions* Function Manual.

The compiler outputs warnings when it detects conversions associated with loss of precision.

NOTICE

The type conversion may cause errors when the program is running, which will trigger the error response set in the task configuration (see *SIMOTION Basic Functions* Function Manual).

Special attention is required when converting DWORD to REAL. The bit string from DWORD is taken unchecked as the REAL value. You must make sure that the bit string in DWORD corresponds to the bit pattern of a normalized floating-point number in accordance with IEEE. To do this, you can use the *_finite* and *_isNaN* functions.

Otherwise, an error is triggered (see above) as soon as the REAL value is first used for an arithmetic operation (for example, in the program or when monitoring in the symbol browser).

Note

The following applies if the value range limits are exceeded during conversion from LREAL to REAL:

- Underflow (absolute value of LREAL number is smaller than the smallest positive REAL number):
  Result is 0.0.
- Overflow (absolute value of LREAL number is larger than the largest positive REAL number):
  The error response specified during task configuration is triggered.

3.8.2 Supplementary conversions

The ST system functions and ST system functions also permit the following conversions:

- Combining bit-string data types

  These functions combine multiple variables of a bit string data type into one variable of a higher-level data type.

- Splitting bit-string data types

  These function blocks split up a variable of a bit string data type into multiple variables of a higher-level data type.

- Converting between any data types and byte arrays

  They are commonly used to create defined transmission formats for data exchange between various devices.

  For further information (e.g. on the arrangement of the byte arrays, application example): Please refer to the *SIMOTION Basic Functions* Function Manual.

- Conversion of technology object data types

  It converts variables of a hierarchical TO data type (driveAxis, posAxis, or followingAxis) or of the general ANYOBJECT type to a compatible TO data type.

For Application Examples and further information: Please refer to the *SIMOTION Basic Functions* Function Manual.

Functions, Function Blocks, and Programs 4

This chapter describes how to create and call user-defined functions and function blocks. Standard functions are already available in the system for type conversion, trigonometry, and bit string manipulation. The *SIMOTION Basic Functions* Function Manual describes how to use system functions and functions of technology objects (TO functions).

A function (FC) is a logic block with no static data. All local variables lose their value when you exit the function and are reinitialized the next time you call the function.

A function block (FB) is a code block with static data. Since an FB has memory, its output parameters can be accessed at any time and from any point in the user program. Local variables retain their values between calls.

Programs are similar to FBs, but have no parameters. However, they can be assigned execution levels and tasks (see *SIMOTION Basic Functions* Function Manual).

FCs and FBs have the advantage that they can be reused, because they are encapsulated source file sections to which parameters can be assigned.

Functions, function blocks, and programs are program organization units (POUs), i.e. they are executable source file sections. You will find an overview of all source file sections in Use of the source file sections (Page 169).

4.1 Creating and calling functions and function blocks

The following description explains how to create and call functions (FCs) and function blocks (FBs). A complete example showing the differences between FC and FB is contained in Comparison of functions and function blocks (Page 161).

The order in which you must define and call the stipulated source file sections is given in Use of the source file sections (Page 169).

How to export and import FCs and FBs is explained in Section Import and export between ST source files (Page 179).

4.1.1 Defining functions

You define a function in the declaration part of the implementation section before the section of the source file (program, FB, or FC) in which it is called.

Use the following syntax:

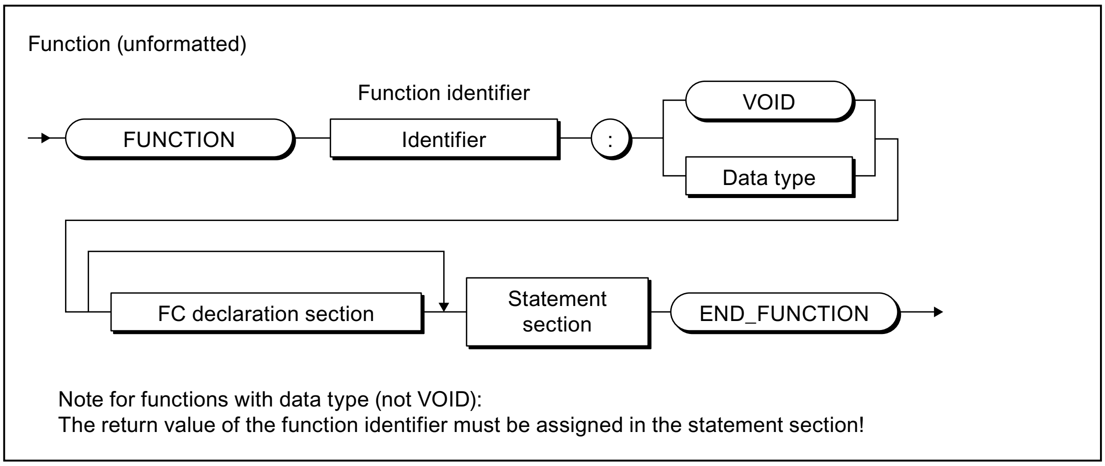

Function (unformatted)
Function identifier
FUNCTION
Identifier
:
VOID
Data type
FC declaration section
Statement section
END_FUNCTION
Note for functions with data type (not VOID):
The return value of the function identifier must be assigned in the statement section!

Figure 4-1 Syntax: Function (FC)

The FUNCTION keyword is followed by an identifier as the FC name and the data type of the return value. Enter VOID as data type if the FC has no return value.

Then enter (see example in Source file with comments (Page 162)):

- The optional declaration section
- The statement section
- The END_FUNCTION keyword

4.1.2 Defining function blocks

You define a function block in the declaration part of the implementation section before the section of the source file (program, FB or FC) in which the FB is called.

Use the following syntax:

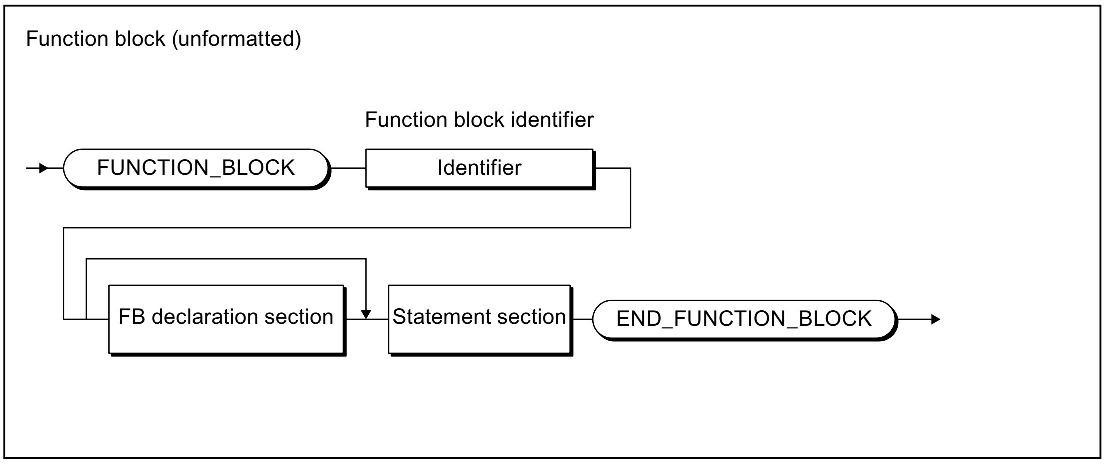


Figure 4-2 Syntax: Function block (FB)

Enter an identifier as the FB name after the FUNCTION_BLOCK keyword.

Then enter (see example in Source file with comments (Page 162)):

- The optional declaration section
- The statement section
- The END_FUNCTION keyword

4.1.3 Declaration section of FB and FC

A declaration section is subdivided into various declaration blocks that are each identified by a separate pair of keywords. Each block contains a declaration list for similar data, such as constants, local variables and parameters. Each type of block may only appear once; the blocks may appear in any order.

The following options are then available for the declaration section of an FC and an FB (see also the example in Source file with comments (Page 162)):

Table 4-1 Declaration blocks for FC and FB: Options

Data	Syntax	FB	FC
Constant	VAR CONSTANT *Declaration list* END_VAR	X	X
Input parameters	VAR_INPUT *Declaration list* END_VAR	X	X

Data	Syntax	FB	FC
In/out parameter	VAR_IN_OUT *Declaration list* END_VAR	X	X
Output parameters	VAR_OUTPUT *Declaration list* END_VAR	X	–
Local variable (for FC and FB)	VAR *Declaration list* END_VAR	X (static)	X (temporary)
Local variable (for FB)	VAR_TEMP *Declaration list* END_VAR	X (temporary)	-
Declaration list: The list of identifiers of the type to be declared			

Parameters are local data and are formal parameters of a function block or function. When the FB or FC is called, the formal parameters are substituted by the actual parameters, thus providing a means of exchanging information between the called and calling source file sections.

- Formal input parameters receive the actual input values (data flow inwards).
- Formal output parameters (only for FB) are used to transfer output values (data flow outwards).
- Formal in/out parameters act as input and output parameters.

The following figures show the syntax for the parameter declaration of an FB or an FC.

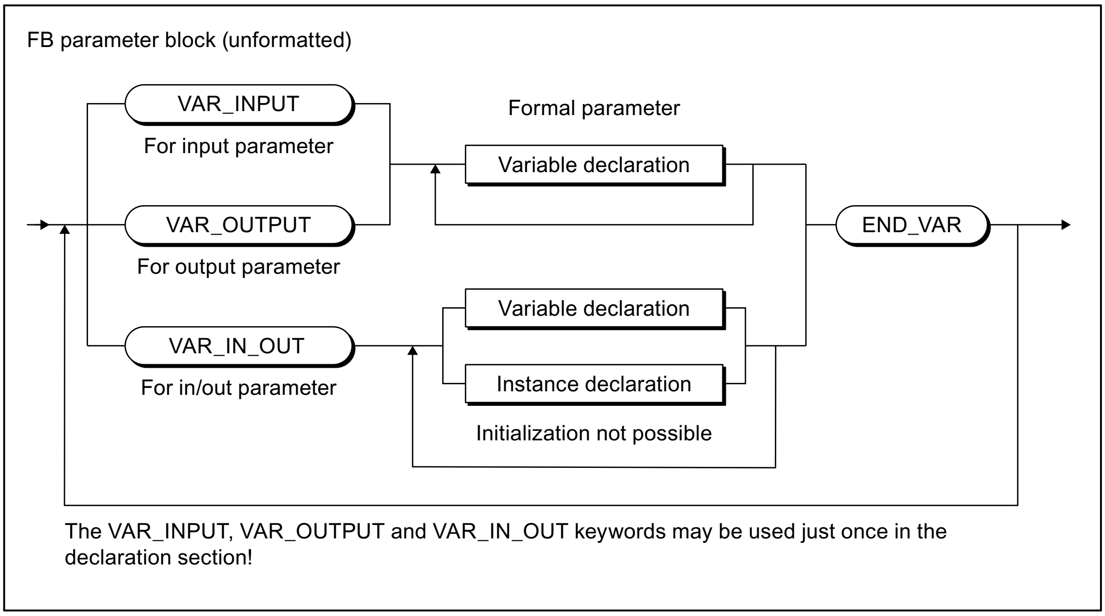


Figure 4-3 Syntax: FB parameter block

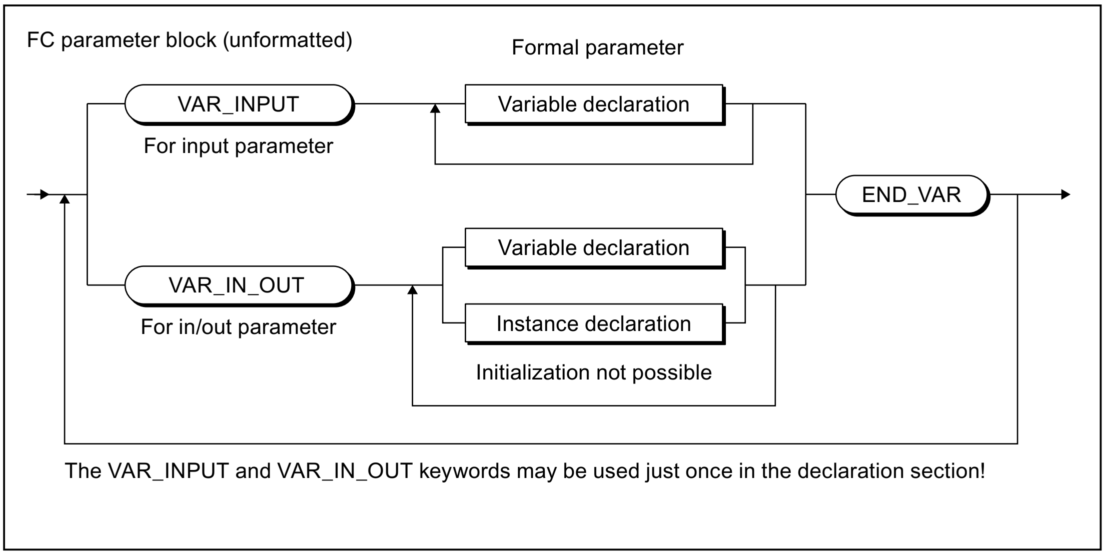

FC parameter block (unformatted)
Formal parameter
VAR_INPUT
For input parameter
Variable declaration
END_VAR
Variable declaration
VAR_IN_OUT
For in/out parameter
Instance declaration
Initialization not possible
The VAR_INPUT and VAR_IN_OUT keywords may be used just once in the declaration section!

Figure 4-4 Syntax: FC parameter block

You can use the declared parameters the same as other variables within the FB or FC, with the following exception: You cannot assign values to input parameters.

From outside of an FB or an FC, you can access:

- The input and output parameters of an FB by means of structured variables (see User-defined data types (Page 94)).

  The access to the input parameter is possible only when the "Permit language extensions" compiler option has been activated (see Global compiler settings (Page 45) or Local compiler settings (Page 46)).

  Data access to the output parameter is possible as standard.

- The return value of an FC by using the function in an expression and assigning this, for example, to a variable (the specification of the function name calls the function and simultaneously returns a result).

4.1.4 Statement section of FB and FC

The statement section of the FC or FB contains statements that are executed when the FC or FB is called. There is no difference compared to the formal rules for creating a statement section; however, you should note the information in the following table.

Note

For tips on the efficient use of parameters, please refer to *Runtime-optimized Programming* in the *SIMOTION Basic Functions* Function Manual.

Table 4-2 Use of parameters and variables in FCs and FBs

Parameter/variable	Use
Input parameters	With the call of an FC or an FB, assign the current values to the input parameters. These values are used for data processing within the FC or the FB, for example, for calculations, but cannot be modified themselves. Only for activated "Permit language extensions" compiler option (see Global compiler settings (Page 45) or Local compiler settings (Page 46)): The input parameters of an FB can be read and written using structured variables, also outside the FB (e.g. in the calling source file section).
In/out parameter	You assign a variable to an in/out parameter for the call of the FB or FC. The FC or the FB accesses this variable directly and can change it immediately. Type conversions are not supported. The variable assigned to an in/out parameter must be able to be directly read and written. Therefore, system variables (of the SIMOTION device or a technology object), I/O variables or process image accesses cannot be assigned to an in/out parameter.
Output parameters (for FB only)	You assign a variable to an in/out parameter for the call of an FB using the **=>** operator. The value of the output parameter (result) is transferred to the variables when the FB is closed. The output parameters of an FB can also be read using structured variables, also outside the FB (e.g. in the calling source file section). An FC has no formal output parameters, because the function name receives the return value. The function name itself is, in a sense, the output parameter.
Local variables	Local variables are variables that are declared and used only within the block. All local variables (VAR ... END_VAR) are temporary in an FC, i.e. they lose their value when the FC is terminated. The next time the FC is called, they are reinitialized. A differentiation between static and temporary local variables is made in the FB: • Static variables (VAR ... END_VAR) retain their value when the FB is closed. • Temporary variables (VAR_TEMP ... END_VAR) lose their value when the FB is closed. The next time the FB is called, they are reinitialized. The value of the local variable cannot be queried directly by the calling block. This is only possible using an output parameter.

4.1.5 Call of functions and function block calls

This provides an overview of the call of the functions and function blocks.

4.1.5.1 Principle of parameter transfer

When you call an FC or FB, data exchange takes place between the calling and the called block. The parameters to be transferred must be specified as a parameter list in the call. The parameters are written in parentheses. Several parameters are separated by commas.

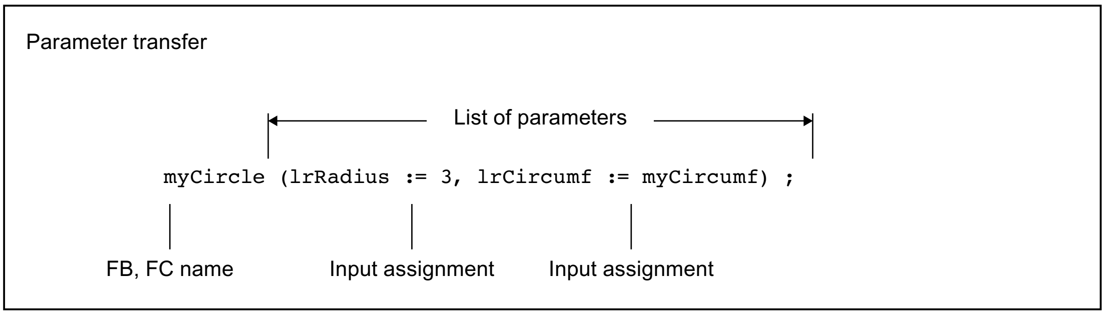


Figure 4-5 Principle of parameter transfer for the call

Input and in/out parameters are normally specified as a value assignment. In this way, you assign values (actual parameters) to the parameters you have defined in the declaration section of the called block (formal parameters).

The assignment of output parameters is made using the **=>** operator. In this way, you assign a variable (actual parameter) to the output parameters you have defined in the declaration section of the called block (formal parameters).

4.1.5.2 Parameter transfer to input parameters

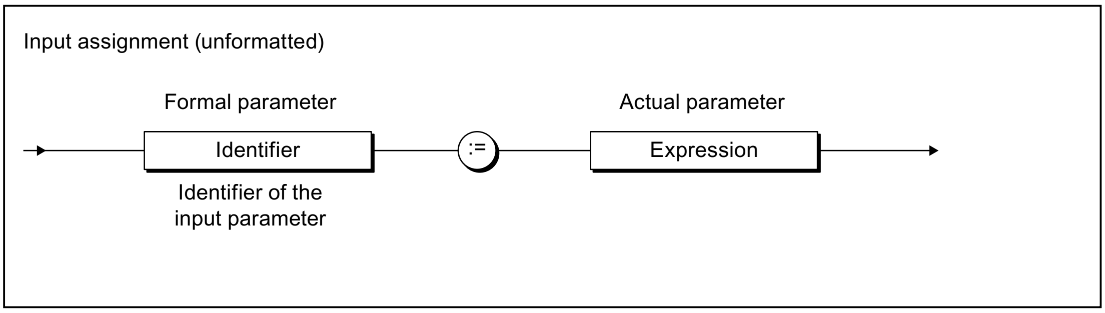

Input assignment (unformatted)
Formal parameter
Identifier
Identifier of the input parameter
:=
Actual parameter
Expression

Figure 4-6 Syntax: Input assignment

You transfer the data (actual parameters) to the formal input parameters of an FB or FC by means of input assignments. You can specify the actual parameters in the form of expressions. You can use the formal input parameters in statements within the FB or FC, but you cannot modify their values.

A short form of parameter transfer is supported, but should not be applied in conjunction with user-defined FBs. This short form is required only for some FCs, see *SIMOTION Basic Functions* Function Manual.

The assignment of actual parameters is optional for an FB. If no input assignment is specified, the values of the last call are retained because an FB is a source file section with memory.

The assignment of an actual parameter is optional for an FC when an initialization expression was specified for the declaration of the formal parameter.

Also refer to the examples in Calling functions (Page 156) and Calling function blocks (instance calls) (Page 157).

You can also read and write an FB's input parameter at any time outside the FB. For further details, see: Accessing the FB's input parameter outside the FB (Page 159).

4.1.5.3 Parameter transfer to in/out parameters

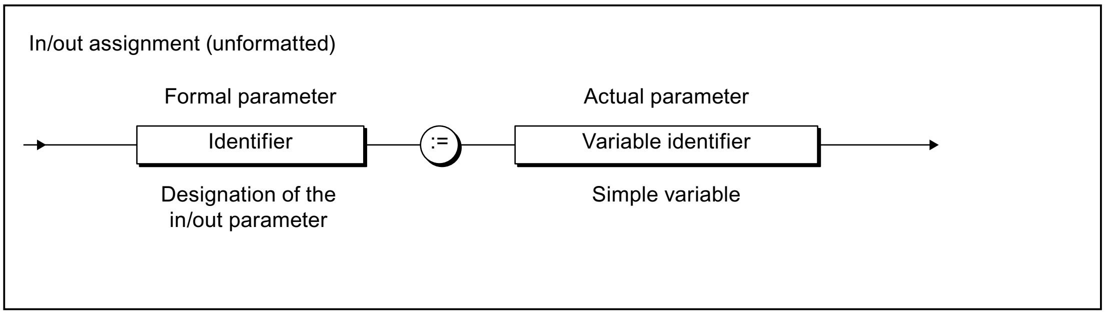


Figure 4-7 Syntax: In/out assignment

You transfer the data (actual parameters) to the formal in/out parameters of an FB or an FC using in/out assignments. You can only assign a variable of the same type to the formal in/out parameter, data type conversions are not possible.

You can use and change the formal in/out parameters in statements within the FC or the FB. The FC or the FB accesses the variable of the actual parameter directly and can change it immediately.

Also refer to the examples in Calling functions (Page 156) and Calling function blocks (instance calls) (Page 157).

When using the STRING data type in in/out assignments, the declared length of the actual parameter must be greater than or equal to the length of the formal in/out parameter (see following example).

Table 4-3 Example of the use of the STRING data type in in/out assignments

```
FUNCTION_BLOCK REF_STRING
    VAR_IN_OUT
        io : STRING[80];
    END_VAR
;    // Statements
END_FUNCTION_BLOCK

FUNCTION_BLOCK test
    VAR
        my_fb : REF_STRING;
        str1 : STRING[100];
        str2 : STRING[50];
    END_VAR
    my_fb(io := str1);     // Permitted call
    my_fb(io := str2);     // Not permitted call,
                           // compiler error message

END_FUNCTION_BLOCK
```

The variable assigned to an in/out parameter must be able to be directly read and written. Therefore, system variables (of the SIMOTION device or a technology object), I/O variables or process image accesses cannot be assigned to an in/out parameter.

Please note the different parameter access times!

4.1.5.4 Parameter transfer to output parameters (for FB only)

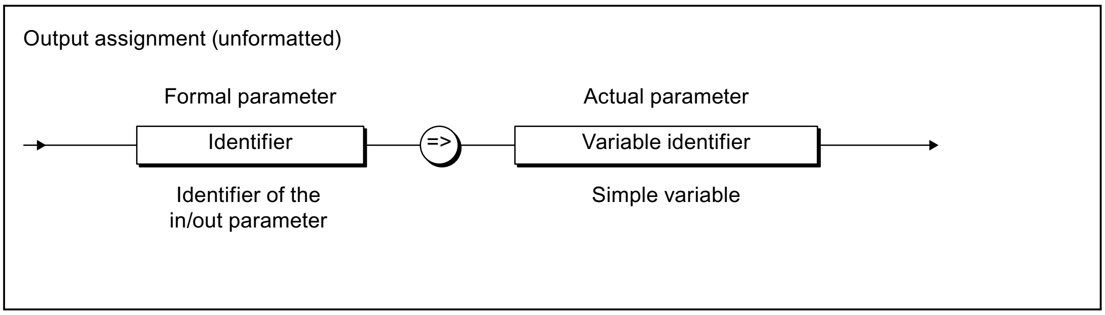


Figure 4-8 Syntax: Output assignment

You use an output assignment to assign the formal output parameters of an FB to the variables (actual parameter) that accept the value of the formal output parameter when the FB is closed.

You can use and change the formal output parameters in statements within the FB.

Also refer to the examples in Calling function blocks (instance calls) (Page 157).

Output assignments are optional for the parameter transfer. You can read and write an FB's output parameter at any time, even outside the FB. For further details, see: Accessing the FB's output parameter outside the FB (Page 159).

4.1.5.5 Parameter access times

The types of access and thus the parameter access times are different:

- In the case of input assignments, the values of the actual parameters are copied into the formal parameters. If large structures, such as arrays, are copied and the FC or FB is called frequently, this can limit performance.
- Values are not copied in in/out assignments. Rather, in this case a link is established between the memory addresses of the formal parameters and those of the actual parameters. Transferring the variables is therefore faster than input assignments (especially where large volumes of data are involved). However, accessing variables from the FB can be slower.
- If you are using unit variables, nothing is copied to the function or function block because these variables are valid in the entire ST source file (see Variable model (Page 184)).

Note

Using in/out parameters instead of input parameters is only faster if a large volume of data is to be passed to the function block.

If unit variables are used predominantly instead of parameters, the resulting program structure will be complex and confusing: object orientation, data encapsulation, multiple use of variable names (encapsulation of validity ranges), etc., are no longer possible.

4.1.5.6 Calling a function

A function is called as follows:

- Function with return value (data type other than **VOID**):

  The function is placed on the right-hand side of a value assignment. It can also appear as operand within an expression. After calling the function, its return value is used at the appropriate point to calculate the expression.

  Examples:

```
y:=sin(x);
y := sin(in := x);
y := sqrt (1 - cos(x) * cos(x));
```

- Function without return value (**VOID** data type)

  The assignment consists only of the function call.

  The following example is valid provided a funct1 function with the in1 and in2 input parameters and the inout in/out parameter has already been defined.

  Example:

```
funct1 (in1 := var11, in2 := var12, inout1 := var13);
```

Note

In the function itself, the result (return value) is assigned to the function name (except for data type VOID).

4.1.5.7 Calling function blocks (instance calls)

Before you call a function block (FB), you must declare an instance. You declare a variable and enter the name of the function block as the data type. You declare this instance:

- Locally (within VAR/END_VAR in the declaration section of a program or function block)
- Globally (within VAR_GLOBAL/END_VAR in the interface of implementation section)
- As an in/out parameter (within VAR_IN_OUT / END_VAR in the declaration section of a function block or a function).

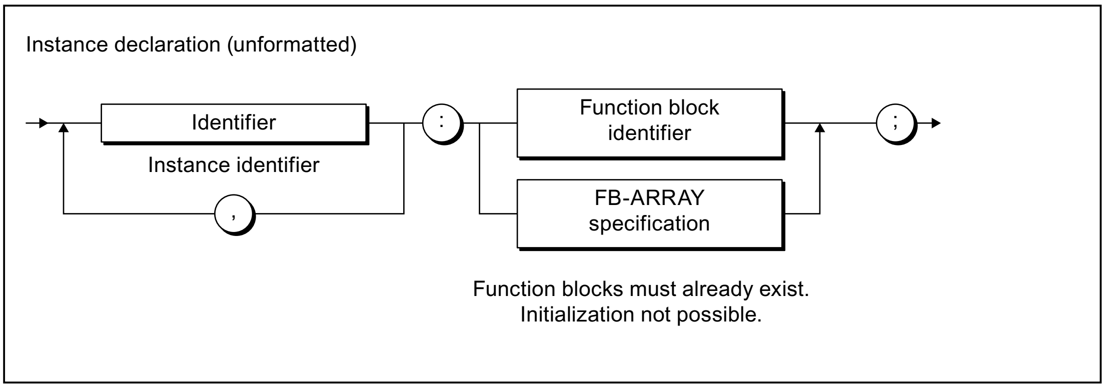

Instance declaration (unformatted)
Identifier
Instance identifier
,
:
Function block identifier
FB-ARRAY specification
;
Function blocks must already exist.
Initialization not possible.

Figure 4-9 Syntax: Instance declaration

The instance declaration can also be an array, e.g.:

```
FB_inst : ARRAY [1..2] OF FB_name.
```

Note

Pay attention to the different initialization times for different variable types.

You call a function block instance in the statement section of a POU (for information about syntax, see Figure). FB parameters are input and in-out assignments separated by commas.

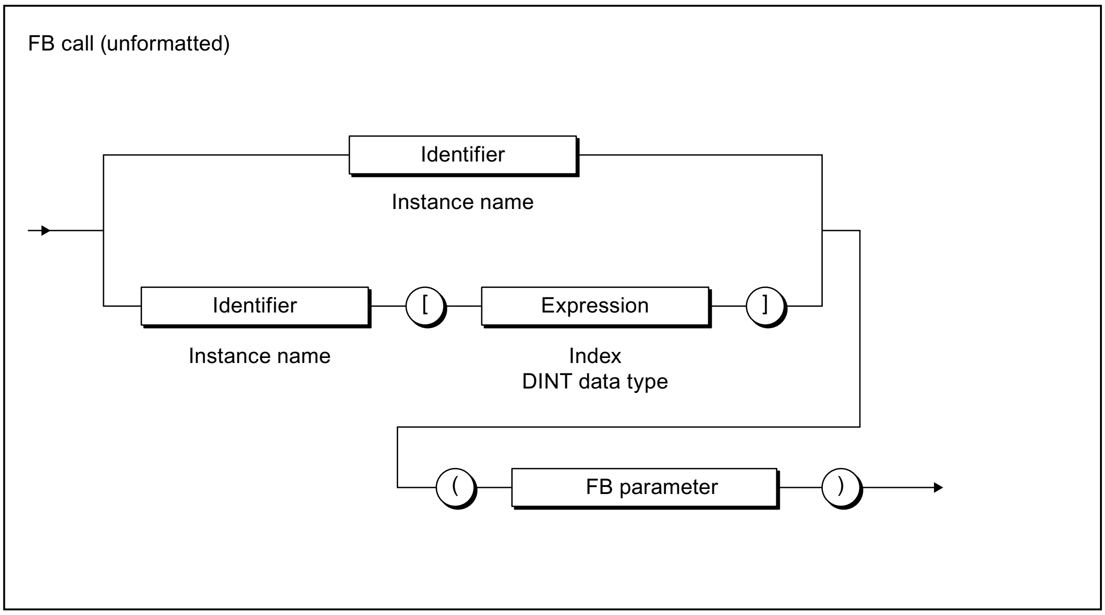


Figure 4-10 FB call syntax

The example in the following table is applicable, assuming that the *supply and motor* function blocks have already been defined:

- FB Supply:
  Input parameters in1, in2; in/out parameter inout; output parameter out
- FB motor:
  In/out parameters inout1, inout2; output parameters out1, out2

Table 4-4 Example of instance declaration, FB call, and access to output parameters

```
VAR
    Supply1, Supply2: Supply;
    Motor1 : Motor;
END_VAR

Supply1 (in1 := var11, in2 := expr12, inout := var13, out => var14) ;
Supply2 (in1 := var21, in2 := expr22, inout := var23, out => var24) ;
Motor1 (inout1 := var31, inout2 := var32, out1 => var33, out2 => var34);
// ...
var15 := PowerSupply1.out;
var25 := PowerSupply2.out;
var35 := Motor1.out1;
var36 := Motor1.out2;
var41 := Motor1.out1 * Motor1.out2 * (Supply1.out + Supply2.out);
```

4.1.5.8 Accessing the FB's output parameter outside the FB

In addition to the output assignment (Page 155) for the call of an FB, it is always possible to access an FB's output parameter outside the FB.

To do so, use structured variables (Page 100) in the *FB instance name.output parameter* format , e.g. *Supply1.out*.

The instance name of the FB itself must not be used in a value assignment!

See also

User-defined data types (Page 94)

4.1.5.9 Accessing the FB's input parameter outside the FB

In addition to the input assignment (Page 153) for the call of an FB, it is always possible to read and write an FB's input parameter outside the FB.

To do so, use structured variables (Page 100) in the *FB instance name.input parameter* format , e.g. *Supply1.in1*.

NOTICE
To be able to use this option, the "Permit language extensions" compiler option (see Global compiler settings (Page 45) and Local compiler settings (Page 46)) must have been activated.

The instance name of the FB itself must not be used in a value assignment!

Table 4-5 Example of assignment to input parameter

```
// Only with compiler option "Permit language extensions" activated
VAR
    var_fb   : _WORD_TO_2BYTE;
    var_word : WORD;
END_VAR
var_fb.wordin := var_word;
// ..
var_fb();
```

4.1.5.10 Error sources in FB calls

Note the following when calling a function block instance:

- **Only assign in/out parameters with variables that are stored directly in the memory.**

  Only the following variables are permissible actual parameters:

  – Global variables (unit variables and global device user variables)

  – Local variables

  – Variables of the data type of the TO (TO instances)

The following, in particular, are permitted:

- System variables (TO variables)
- Names of technological objects from the Engineering System
- I/O variables
- Absolute and symbolic process image access

- **Do not use functions (FCs) as in/out parameters.**

  The FC return value, i.e. the FC call, cannot be an actual parameter in an in/out assignment. You must first store the result of the FC in a local variable and then use this variable as an actual parameter in the in/out assignment.

- **Do not use constants as in/out parameters.**

  Only variables can be used as actual parameters of an in/out assignment because the value is written back.

- **In/out parameters cannot be initialized.**

4.2 Comparison of functions and function blocks

The differences between user-defined function blocks (FBs) and functions (FCs) are succinctly illustrated below using a thorough example.

4.2.1 Description of example

The following example illustrates the differences between FBs and FCs. For simplicity, each type of parameter is used only once, although, in reality, you can define any number of parameters. The terms used are defined both in the detailed description Defining functions (Page 148) and Defining function blocks (Page 149).

A block will be created as an FB and an FC in the declaration part of the implementation section for use in calculating the circumference and the area of a circle for a radius input variable.

- An input parameter is defined for the radius.
- An in/out parameter is defined for the circumference of the circle, i.e. the value of the transferred variable is assigned directly during the call of the FB or the FC.
- There are several ways of defining the area of the circle for the FB and the FC:
  - For the FB, an output parameter is defined.
  - For the FC, its return value is used; the data type of the return value is defined appropriately.
- Each FB and FC call will be recorded in a counter (local variable). The explanations for the example state: We will see that this value will continue to be counted only in the FB.
- In the program section, the FB or the FC is called and the actual parameters assigned to the following formal parameters:
  - For the FB: Input, in/out and output parameters
  - For the FC: Input and in/out parameters.

  The values for the circumference and the area are available after calling the FB or the FC:

  - For the FB: in the actual parameters of the in/out and output parameter.

    The output parameter can be read even outside the FB.
  - For the FC: in the return value of the function and in the actual parameter of the in/out parameter.

4.2.2 Source file with comments

Table 4-6 Example of differences between FB and FC

Function block (FB)

```
INTERFACE
  PROGRAM CircleCalc1;
END_INTERFACE
IMPLEMENTATION
  FUNCTION_BLOCK Circle1
    //Constant declaration
    VAR CONSTANT
      PI : LREAL := 3.1415 ;
    END_VAR
    //Input parameter
    VAR_INPUT
      Radius : LREAL;
    END_VAR
    //In/out parameter
    VAR_IN_OUT
      circumference : LREAL;
    END_VAR
    //Output parameter
    VAR_OUTPUT
      Area : LREAL;
    END_VAR
    // Local variables, static
    VAR
      Counter : DINT;
      (* Variable retains its value
      between calls *)
    END_VAR
    //Call counter
    Counter := counter + 1 ;
    Circumference := 2 * PI * Radius ;
    Area := PI * Radius**2 ;
  END_FUNCTION_BLOCK
  PROGRAM CircleCalc1
    VAR
      myCircle1        : Circle1 ;
      myArea1, myArea2 : LREAL;
      myCircf          : LREAL;
    END_VAR;
    myCircle1(Radius := 3
      , Circumference := myCircf
      , Area => myArea1) ;
    myArea2 := myCircle1.Area ;
    // myCircf has the value 18,849
    // myArea1 has the value 28,274
    // myArea2 has the value 28,274
  END_PROGRAM
END_IMPLEMENTATION
```

Function (FC)

```
INTERFACE
  PROGRAM CircleCalc2;
END_INTERFACE
IMPLEMENTATION
  FUNCTION Circle2 : LREAL
    //Constant declaration
    VAR CONSTANT
      PI : LREAL := 3.1415 ;
    END_VAR
    //Input parameter
    VAR_INPUT
      Radius : LREAL;
    END_VAR
    //In/out parameter
    VAR_IN_OUT
      circumference : LREAL;
    END_VAR
    //Output parameter
    // Not possible


    // Local variables, temporary
    VAR
      Counter : DINT;
      (* Variable will be initialized
      with 0 for each call *)
    END_VAR
    //Call counter
    Counter := Counter + 1 ;
    Circumference := 2 * PI * Radius ;
    Circle2 := PI * Radius**2 ;
  END_FUNCTION
  PROGRAM CircleCalc2
    VAR

      myArea  : LREAL;
      myCircf          : LREAL;
    END_VAR;
    myArea := Circle2(Radius := 3
      , Circumference := myCircf);


    // myCircf has the value 18,849
    // myArea has the value 28,274

  END_PROGRAM
END_IMPLEMENTATION
```

Table 4-7 Example of the differences between FB and FC for the previous example

Function block (FB)	Function (FC)
Comments	
Reserved words for the definition: FUNCTION_BLOCK and END_FUNCTION_BLOCK	Reserved words for the definition: FUNCTION and END_FUNCTION
No return value permitted.	The data type of the return value must be specified after the name (VOID data type, if no return value).
Input parameters can be used to transfer values to the FB.	Input parameters can be used to transfer values to the FC.
In/out parameters can be used to read and write the transferred variables in the FB.	In/out parameters can be used to read and write the transferred variables in the FC.
Output parameters can be used to return values from an FB.	No output parameters permitted.
The local variables are static, i.e. they retain their value between FB calls. The *Counter* local variable is incremented; its value is retained when the FB is terminated. The variable is therefore incremented each time the FB is called. To see this behavior: Assign the value of the local variables to a global variable in the FB. Monitor the value of the global variable after repeated FB calls.	The local variables are temporary, i.e. they lose their value when the function is terminated. Although the *Counter* local variable is incremented, its value is lost when the FC is exited. The variable is reinitialized (to 0 in the example) at the next FC call. To see this behavior: Assign the value of the local variables to a global variable in the FC. The value of the global variable remains unchanged after repeated FC calls.
In the statement section, the results (return values) are assigned to the output or in/out parameters.	In the statement section, the result (return value) is assigned to the function name (except when VOID data type is specified).
In the declaration section of the block that executes the call, an instance of the FB is declared: you declare a variable and specify the name of the FB as its data type. You use the declared instance name to call the FB and to access its output parameters. The name of the FB itself must not be used in the statement section.	
• You assign a variable to the in/out parameters when the FB instance is called. • With the call, you can assign the output parameters to a variable. • You can read an FB's output parameters, even outside the FB. For this purpose, use structured variables in the following format: *FB-instancename.outputparameter*.	• You assign a variable to the in/out parameters when the FB instance is called. • To obtain the return value of the FC: – Assign the function to a variable. – Use the function in an expression on the right side of a value assignment.
The program that executes the call cannot access variables other than the in/out variables and output parameters of the FB. Exception: For activated "Permit language extensions" compiler option (see Global compiler settings (Page 45) or Local compiler settings (Page 46)), the called program can also access the input parameters of an FB. For this purpose, use structured variables in the following format: *FB-instancename.inputparameter*.	The program that executes the call cannot access any variables other than the return value.

4.3 Programs

Programs are a series of statements placed between the PROGRAM and END_PROGRAM keywords.

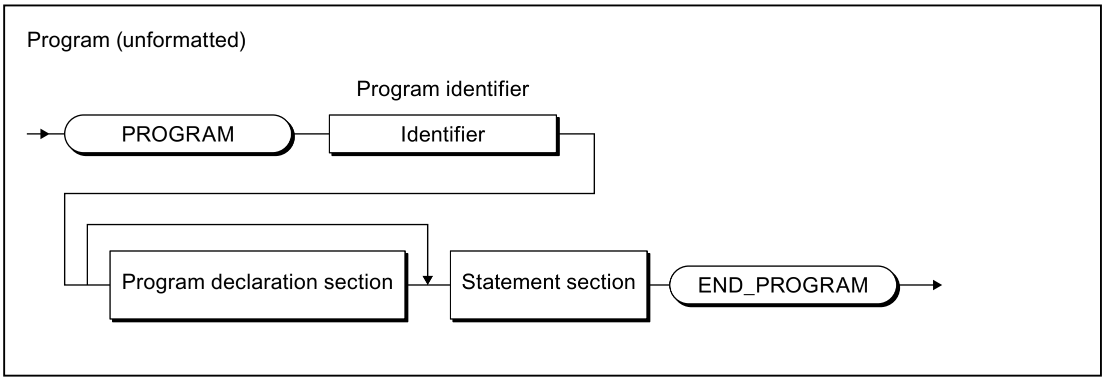


Figure 4-11 Syntax: Program

Programs are declared in the Implementation section (Page 171) of an ST source file and are comparable with the FB. Static local variables (VAR...END_VAR) or temporary local variables (VAR_TEMP...END_VAR) can be created, for example. However, they do not have any formal parameters and so cannot be called with arguments. Examples for programs are contained in the Source file with comments (Page 162) and Source text of the sample program (Page 64) sections.

Assignment of a program in the execution system

By default, programs in the execution system are assigned to a task. The execution behavior of the programs, e.g. the associated task determines the initialization of the variables. For more information about the execution system and the tasks, refer to the *SIMOTION Basic Functions* Function Manual. This requires that the program in the Interface section (Page 170) of the ST source file must be specified as the program organization unit to be exported.

Calling a program in the program ("program in program")

Optionally, a program can also be called within a different program or a function block. This requires that the following compiler options be activated (see Global settings on the compiler (Page 45) and Local settings on the compiler (Page 46)):

1. "Permit language extensions" for the program source of the calling program or function block and
2. "Create program instance data only once" for the program source of the calling program.

The call is performed as for a function with parameters and return value, see following example.

NOTICE

The activated "Create program instance data only once" compiler option causes:

- The static variables of the programs (program instance data) are stored in a different Memory area (Page 194). This also changes the Initialization behavior (Page 204).
- All called programs with the same name use the same program instance data.

Table 4-8 Example for calling a program in a program

```
PROGRAM my_prog
    ; // ...
END_PROGRAM

PROGRAM main_prog
    ; // ...
    my_prog();
    ; // ...
END_PROGRAM
```

This can be used to perform most of the programming of the assignment of the programs to the tasks. In the execution system, only each associated calling program needs to be assigned to the tasks.

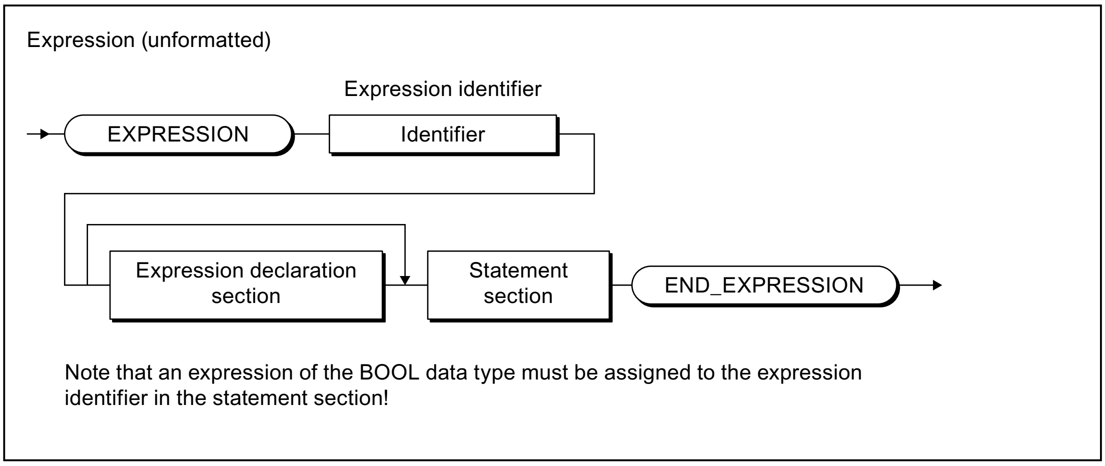
4.4 Expressions

The expression is a special case of a function declaration:

- The data type of the return value is defined as BOOL and is not specified explicitly.

It is used in conjunction with the WAITFORCONDITION statement (Page 139).

An expression can only be declared in the implementation section of the ST source file.



Note that an expression of the BOOL data type must be assigned to the expression identifier in the statement section!

Figure 4-12 Syntax: Expression

Optionally, the following can be declared in the declaration section:

- Local (temporary) variables
- Local constants
- User-defined data types (UDT)
- Input and in/out parameters (as of Version V4.1 of the SIMOTION kernel)

The following can be accessed in the statement section:

- To the local variables of the expression
- To the input and in/out parameters (provided their declaration is permitted)
- Unit variables
- Global device variables, I/O variables, and the process image

An expression of data type BOOL must be assigned to the expression name in the statement section of the expression (see figure).

Note

The statement section of the expression cannot contain any function calls or loops.

Example

The following example assumes that the feeder program is running in a MotionTask. The option Activation after StartupTask is selected for this MotionTask. The assignment of programs to tasks is performed in SIMOTION SCOUT (see SIMOTION Motion Control Basic Functions function description).

Table 4-9 Example of the use of an EXPRESSION and the WAITFORCONDITION statement

```
INTERFACE
    USEPACKAGE cam;
    PROGRAM feeder; // in MotionTask_1
END_INTERFACE

IMPLEMENTATION
    // Condition for WAITFORCONDITION statement
    EXPRESSION automaticExpr
        automaticExpr := IOfeedCam; // Digital input
    END_EXPRESSION

    PROGRAM feeder
        VAR
            retVal : DINT ;
        END_VAR ;
        retVal := _enableAxis (axis := realAxis,
            enableMode := ALL,
            servoCommandToActualMode := INACTIVE,
            nextCommand := WHEN_COMMAND_DONE,
            commandId := _getCommandId() );

        // Wait until the start condition is satisfied
        WAITFORCONDITION automaticExpr WITH TRUE DO
            // High-priority execution of all statements
            // to the END_WAITFORCONDITION command
            retVal := _pos (axis := realAxis,
                positioningMode := RELATIVE,
                position := 500,
                velocityType := DIRECT,
                velocity := 300,
                velocityProfile := TRAPEZOIDAL,
                mergeMode := IMMEDIATELY,
                nextCommand := WHEN_MOTION_DONE,
                commandId:= _getCommandId() );
        END_WAITFORCONDITION;

        retVal := _disableAxis (axis := realAxis,
            disableMode := ALL,
            servoCommandToActualMode := INACTIVE,
            nextCommand := WHEN_COMMAND_DONE,
            commandId := _getCommandId() );
    END_PROGRAM
END_IMPLEMENTATION
```

Further examples are contained in the SIMOTION Motion Control Basic Functions Function Manual. In particular, the manual describes how, as of Version V4.1 of the SIMOTION kernel, you use an EXPRESSION with parameters and, for example, program a time monitoring in a WAITFORCONDITION statement.

Integration of ST in SIMOTION 5

This section describes the interoperability of ST programs and SIMOTION SCOUT.

5.1 Source file sections

An overview of the meaning of the source file sections was provided in Structure of an ST source file (Page 86). This section describes details, such as the syntax of the sections and how to use them to import and export data between several ST source files.

5.1.1 Use of the source file sections

You must follow certain structure and syntax rules in your source file sections (modules), so that the ST source file can be compiled. A few general guidelines are presented here; details on source file sections are presented later in this section:

- When creating the source file, you should always pay attention to the order of the source file sections. A section that is to be called must always precede the calling section; otherwise the section that is to be called will not recognize the calling section.

  For example, variables must always be declared before they are used and functions must be defined before they are called.
- The source text for the most common source file sections – program, function or function block – consists of the following:
  - Start of section with reserved word and identifier
  - Declaration section (optional)
  - Statement section
  - End of section with reserved word
- Identifiers for source file sections – hereinafter referred to as *name* or *name_list* - follow the general syntax rules for identifiers (Identifiers in ST (Page 73)).

Note

A template with all possible source file sections is available in the online help.

5.1.1.1 Interface section

The interface section contains statements for importing and exporting data (data types, variables, function blocks, functions, and programs). Technology packages and libraries can also be downloaded.

The interface section has the following syntax:

Table 5-1 Syntax of interface section

Syntax	`INTERFACE`
	`// Interface statements (optional)`
	`END_INTERFACE`
	An individual identifier of the section cannot be specified.
	Optionally, interface statements exist in the following order between reserved words INTERFACE and END_INTERFACE.
	1. Specification of utilized technology package. Syntax:
	`USEPACKAGE tp-name [AS namespace];`
	For more details, refer to the *SIMOTION Basic Functions* Function Manual.
	2. Specification of utilized libraries.
	Syntax:
	`USELIB library-name-list [AS namespace];`
	For more information, see "Using data types, functions and function blocks from libraries (Page 230)".
	3. Reference to other units in order to use their exported components.
	Syntax:
	`USES unit_name-list;`
	For more information, see "USES statement in an importing unit (Page 181)".
	4. Declarations and specifications for the export
	– Data type definitions (Page 176):
	User-defined data types (UDT) that are valid in the entire ST source file and that are to be exported
	– Variable declarations (Page 177):
	Unit variables and unit constants valid in the entire ST source file and exported.
	Permissible keywords: See table in "Variable declaration (Page 177)".
	– Information regarding program organization units (POU) to be exported.
	Syntax:
	`FUNCTION fc_name;`
	`FUNCTION_BLOCK fb_name;`
	`PROGRAM program_name;`
	All technology packages, libraries, imported units, data type declarations, variable declarations and program organization units listed in the interface section will be exported. For more information on export, see "Interface section of an exporting unit (Page 179)".

Sequence	The interface section is the first section of an ST source file[1]. The order of the interface statements 1 to 4 is fixed. Within number 4, any order is permitted. The individual declaration blocks for data type definitions and variable definitions can appear more than once. Attention: Identifiers must be declared before they are used.
Frequency	Once per ST source file
Mandatory section	yes
[1] Optionally, the unit statement can precede the interface section (see "Identifier of the unit (Page 179)".	

5.1.1.2 Implementation section

The implementation section contains the executable sections, comprising the main part of the ST source file.

The implementation section has the following syntax:

Table 5-2 Syntax of the implementation section

Syntax	`IMPLEMENTATION`
	`// Implementation statements (optional)`
	`END_IMPLEMENTATION`
	An individual identifier of the section cannot be specified.
	Optionally, implementation statements (main part of the ST source file) exist in the following order between the reserved words IMPLEMENTATION and END_IMPLEMENTATION:
	1. Reference to other units in order to use their exported components. Syntax:
	`USES unit_name-list;`
	For more information, see "USES statement in an importing unit (Page 181)".
	2. Declarations
	– Data type definitions (Page 176):
	User-defined data types (UDT) that are valid in the entire ST source file
	– Variable declarations (Page 177):
	Unit variables and constants that are valid in the entire ST source file
	Permissible keywords: See table in "Variable declaration (Page 177)".
	3. Program organization units (POUs) (Page 171)

Sequence	Always follows the interface section. The order of the implementation statements indicated above is mandatory; within number 2 and 3, any order is permitted: Attention: Identifiers must be declared before they are used.
Frequency	Once per ST source file
Mandatory section	yes

5.1.1.3 Program organization units (POUs)

POUs are the executable source file sections:

- Functions (FC) (Page 172)
- Expressions (Page 174)
- Function blocks (FB) (Page 173)
- Programs (Page 174)

Note

Called POUs always precede the calling POUs so that they are recognized by the latter.

5.1.1.4 Functions (FCs)

Functions (FC) are classified as program organization units (POUs). Functions are paramterized source file sections with temporary data that can be called from programs and function blocks. All internal variables lose their values when the function is exited and are reinitialized the next time the function is called.

FCs have the following syntax:

Table 5-3 Syntax of functions (FCs)

Syntax	`FUNCTION name : function_data_type`
	`// Declaration section`
	`// Statement section`
	`END_FUNCTION`
	name stands for the identifier of the function, while *function_data_type* stands for the data type of the return value.
	Permissible keywords for the variable declaration in the declaration section: See table in "Variable declaration (Page 177)".
	Note the following for functions with *function_data_type* <> VOID: In the statement section, an expression of data type *function_data_type* must be assigned to the function identifier!

Sequence	FCs can only be defined in the implementation section. Pay attention to the order: FCs must come before the POUs from which they are called! The declaration section (Page 175) must precede the statement section (Page 176).
Frequency	Any number of times per ST source file
Mandatory section	no

For information on functions (FC), see Creating and calling functions and function blocks (Page 147).

5.1.1.5 Function blocks (FBs)

Function blocks (FB) are classified as program organization units (POUs). They are source file sections with static data that can be called from programs and assigned parameters (internal variables retain their value between calls). Since an FB has memory, its output parameters can be accessed at any time and from any point in the user program.

FBs have the following syntax:

Table 5-4 Syntax of the function blocks

Syntax	`FUNCTION_BLOCK name`
	`// Declaration section`
	`// Statement section`
	`END_FUNCTION_BLOCK`
	name stands for the identifier of the function block.
	Permissible keywords for the variable declaration in the declaration section: See table in "Variable declaration (Page 177)".

Special features	Before you call a function block (FB), you must declare an instance: You declare a variable and enter the identifier of the function block as the data type. You can declare the instance locally (within VAR / END_VAR in the declaration sections of a program or a function block). You can declare the instance globally (within VAR_GLOBAL / END_VAR in the interface or Implementation section), however, not using function blocks defined in the same ST source file. This is possible only with function blocks made available by imported program source files and libraries. You cannot declare an instance of an FB in FCs.
Sequence	FBs can only be defined in the implementation section. Pay attention to the order: FBs must precede the POE in which an instance is declared as local variable. The declaration section (Page 175) must precede the statement section (Page 176).
Frequency	Any number of times per ST source file
Mandatory section	no

For information on the FB, see Creating and calling functions and function blocks (Page 147).

5.1.1.6 Programs

Programs are classified as program organization units (POUs). They are called on the target system according to their task assignment (see *Configuring the execution system* in the *SIMOTION Basic Functions* Function Manual) and can call FCs and FBs.

Programs have the following syntax:

Table 5-5 Syntax of the programs

Syntax	`PROGRAM name`
	`// Declaration section`
	`// Statement section`
	`END_PROGRAM`
	name stands for the name of the program.
	Permissible keywords for the variable declaration in the declaration section: See table in "Variable declaration (Page 177)".

Sequence	Programs can only be defined in the implementation section. It is advantageous to place programs after expressions, FCs, and FBs. This enables the program to recognize and use the source file sections. The declaration section (Page 175) must precede the statement section (Page 176).
Frequency	Any number of times per ST source file
Mandatory section	no

For more information about programs, see Programs (Page 164).

5.1.1.7 Expressions

Expressions are a special case of a function declaration with the specified data type BOOL of the return value. The expression within the EXPRESSION <expression identifier> ... END_EXPRESSION reserved words assigned to the function name is evaluated.

You can use the WAITFORCONDITION construct to wait directly for a programmable event or condition in a MotionTask. The statement suspends the task that called it until the condition (expression) is true.

Expressions have the following syntax:

Table 5-6 Syntax of the expressions

Syntax	`EXPRESSION name`
	`// Declaration section`
	`// Statement section`
	`END_EXPRESSION`
	name stands for the identifier of the expression.
	Permissible keywords for the variable declaration in the declaration section: See table in "Variable declaration (Page 177)".
	Attention: In the statement section, an expression of data type BOOL must be assigned to the expression identifier!

Sequence	An expression can only be declared in the implementation section of an ST source file. Therefore, expressions precede the program in which they are called from a WAITFORCONDITION control structure. The declaration section (Page 175) must precede the statement section (Page 176).
Frequency	Any number of times per ST source file
Mandatory section	no

For more information on expressions, see Expressions (Page 166). In conjunction with the WAITFORCONDITION statement, see *SIMOTION Basic Functions* Function Manual.

5.1.1.8 Declaration section

The declaration section of a program organization unit (POU) contains the data type definition and the variable declaration of the POU.

The declaration section has the following structure:

Table 5-7 Structure of the declaration section

Structure	`// Data type definition`
	`// Variable declaration`

Sequence	The declaration section has no explicit keywords at the start or end. It begins after the keyword of the respective program organization unit (POU) and ends with the first executable statement of the statement section. It contains the following in any order: • Data type definitions (Page 176): User-defined data types (UDT) that are valid locally in the POU • Variable declarations (Page 177): Variables and constants that are valid locally in the POU Permissible keywords according to the respective POU: See table in "Variable declaration (Page 177)". Attention: Identifiers must be declared before they are used.
Frequency	Once per POU
Mandatory section	no

5.1.1.9 Statement section

The statement section of a POU consists of the individual (executable) statements.

The statement section has the following structure:

Table 5-8 Structure of the statement section

Structure	`// Statements`
Sequence	The statement section has no explicit keywords at the start or end. It begins after the declaration section and ends with the keyword of the respective POU.
Frequency	Once per POU
Mandatory section	no

For more information on statements, see

- Value assignments and expressions (Page 112)
- Control statements (Page 130)
- Calling functions and function blocks (Page 153)

5.1.1.10 Data type definition

For the data type definition, you specify user-defined data types (UDT). You can use them for variable declarations. UDTs can be defined in the interface section, the implementation section, and the declaration section of FCs, FBs, and programs.

The data type definition has the following syntax:

Table 5-9 Syntax of the data type definition

Syntax	`TYPE`
	`name : data_type_specification;`
	` // ...`
	`END_TYPE`
	name represents the name of the individual data type that you use for the Variable declarations.
	data_type_specification stands for any data type or a structure. Any number of individual data types can appear between TYPE and END_TYPE.

Sequence	You can define UDTs as follows: • In the Interface section: The UDTs are recognized within the ST source file and will be exported They can be used in the interface and implementation section for declaration of unit variables and in all POUs for declaration of local variables. In addition, they can be used in all units which import this ST source file (in SIMOTION ST with the USES statement). • In the Implementation section: The UDTs are recognized within the ST source file They can be used in the implementation section for declaration of unit variables and in all POUs for declaration of local variables. • In the Declaration section of a POU (FC, FB, program, expression) The UDTs are only recognized locally within the POU They can only be used within the POU for declaration of local variables. UDTs must be defined before they are used in a variable declaration.

Frequency	The TYPE / END_VAR declaration block may appear more than once in a source file section; any number of UDTs are possible within a declaration block.
Mandatory section	no

For more information about the UDT, see User-defined data types (Page 94).

5.1.1.11 Variable declaration

A declaration section contains variable declarations and can itself be contained in FCs, FBs, and programs (POUs) as well as in the interface section and the implementation section.

The variable declaration has the following syntax:

Table 5-10 Syntax of variable declaration

Syntax	`variable_type`
	` name_list : data_type;`
	` // ...`
	`END_VAR`
	variable_type represents the keyword of the variable type being declared. The permitted keywords depend on the source file section.
	• In the Interface section or Implementation section of an ST source file:
	`VAR_GLOBAL`: Non-retentive unit variable
	`VAR_GLOBAL CONSTANT`: Unit constant
	`VAR_GLOBAL RETAIN`: Retentive unit variable
	• In the Declaration section of a function:
	`VAR`: Local variable
	`VAR CONSTANT`: Local constant
	`VAR_INPUT`: Input parameter
	`VAR_IN_OUT`: In/out parameter
	• In the Declaration section of a function block:
	`VAR`: Local variable
	`VAR CONSTANT`: Local constant
	`VAR_TEMP`: Temporary variable
	`VAR_INPUT`: Input parameter
	`VAR_OUTPUT`: Output parameter
	`VAR_IN_OUT`: In/out parameter
	• In the Declaration section of a program:
	`VAR`: Local variable
	`VAR CONSTANT`: Local constant
	`VAR_TEMP`: Temporary variable
	• In the Declaration section of an expression:
	`VAR`: Local variable
	`VAR CONSTANT`: Local constant
	`VAR_INPUT`: Input parameter (as of Version 4.1 of the SIMOTION kernel)
	`VAR_IN_OUT`: In/out parameter (as of Version 4.1 of the SIMOTION kernel)
	name_list is the list of identifiers of the *data_type* data type to be declared.

Sequence	The variable is declared: • In the Interface section of the ST source file: Permissible keywords: see table field syntax. The unit variables are recognized within the ST source file and will be exported. They can be used in all POUs of the ST source file. In addition, they can be used in all units which import this ST source file (in SIMOTION ST with the USES statement). • In the Implementation section of the ST source file: Permissible keywords: see table field syntax. The unit variables are recognized within the ST source file. They can be used in all POUs of the ST source file. • In the Declaration section of a POU (FC, FB, program, expression) Permissible keywords according to the type of POU: See table cell *Syntax*. The variables are only recognized locally within the POU. They can only be used within the POU for declaration of local variables. Exceptions: – You can also access the output parameters of a function block outside the FB. – You can access the input parameters of a function block outside the FB provided the "Permit language extensions" compiler option has been activated. See Global settings of the compiler (Page 45) and Local settings of the compiler (Page 46). Variables must be declared before they are used.
Frequency	The number of times the variable_type / END_VAR declaration block of a specific variable type can appear depends on the associated source file section: • In the interface and implementation section of the ST source: The declaration blocks may appear more than once. • In the declaration section of a POU (FC, FB, program, expression): Each declaration block (other than VAR CONSTANT / END_VAR) may appear just once in the declaration section. Permitted declaration blocks and keywords depending on the associated source file section: See table cell *Syntax*. Any number of variable declarations are possible within a declaration block.
Mandatory section	no

For more information about variable declarations, see Variable declaration (Page 105).

5.1.2 Import and export between ST source files

ST applies the unit concept, where you can access the global variables, data types, functions (FCs), function blocks (FBs), and programs of other source files. Thus, for example, you can compile reusable subroutines and make them available.

5.1.2.1 Unit identifier

Below, unit refers to a program source file (e.g. ST source file, MCC source file). The name of the program source file defined in SIMOTION SCOUT is applied as the identifier.

Optionally, you can set the unit statement as first statement for an ST source file (preceding the interface section). Syntax:

```
UNIT name;
```

name corresponds to the name of the ST source file defined in SIMOTION SCOUT, see Add ST source (Page 21) or Change the properties of an ST source file (Page 23).

The unit statement is ignored if the name specified there differs from the name of the ST source file.

5.1.2.2 Interface section of an exporting unit

You can enter the following constructs in the interface section of an exporting unit. The syntax of the constructs is only implied here, for details, see "Interface section (Page 170)".

- The type declarations to be exported

  TYPE

  User-defined data types with their complete declaration.

- The variable declarations to be exported

  VAR_GLOBAL, VAR_GLOBAL RETAIN, or VAR_GLOBAL CONSTANT

  Non-retentive and retentive unit variables and unit constants with their complete declaration.

- POUs (functions, function blocks, and programs) to be exported

  Specify each POU (function, function block, or program) to be exported with the relevant keyword. Close each entry with a semicolon.

  – FUNCTION_BLOCK *fb_name* ;

  – FUNCTION *fc_name* ;

  – PROGRAM *program_name* ;

Specifications can be made in any order; the POU itself is programmed in the implementation section of the ST source file.

Note

The following further specifications are possible in the interface section, they are listed **before** the exported data types, variables and POU:

1. Specification of utilized technology packages (USEPACKAGE …).
2. Specification of utilized libraries (USELIB …).
3. Reference to other units in order to use their exported units (USES …).

These imported technology packages, libraries and units are also exported. For inheritance, see "USES statement in an importing unit (Page 181)".

You must adhere to the order presented for the specifications in the interface section of a unit (ST source file), see "Interface section (Page 170)". Otherwise, error-free compilation of the ST source file will not be possible.

The programs of an ST source file must be listed in the interface section so that they can be assigned to a task in the execution system (see *Configuring the execution system* in the *SIMOTION Basic Functions* Function Manual). The compiler outputs a warning message if programs cannot be exported in the interface section of an ST source file.

Functions and function blocks that are only used in the ST source file should not be listed in the interface section.

5.1.2.3 Example of an exporting unit

Below is an example of an exporting unit (*myUnit_A*). It is imported by *myUnit_B* (see Example of an importing unit (Page 183)).

Table 5-11 Example of an exporting unit

```
UNIT myUnit_A;    // Optional, name of the ST source file

INTERFACE
    // ... USES statement also possible here
    TYPE        // Declaration of data types to be exported
        color : (RED, GREEN, BLUE);
    END_TYPE
    VAR_GLOBAL
        cycle : INT := 1;  // Declaration of the
                           // unit variables to be exported
    END_VAR
    FUNCTION myFC;         // Export statement of an FC
    FUNCTION_BLOCK myFB;   // Export statement of an FB
    PROGRAM myProgram_A;   // Export statement of a program
                           // (to interface with the execution system)
END_INTERFACE

IMPLEMENTATION
    Function myFC : LREAL   // Function written out
        ;     // ... (Statements)
    END_FUNCTION

    Function_BLOCK myFB      // Function block written out
        ;     // ... (Statements)
    END_FUNCTION_BLOCK

    PROGRAM myProgram_A     // Program written out
        ;     // ... (Statements)
    END_PROGRAM
END_IMPLEMENTATION
```

5.1.2.4 USES statement in an importing unit

Enter the following statement in the interface or implementation section of an importing unit:

```
USES unit_name-list
```

unit_name-list is a list of units separated by commas from which the modules are to be imported.

Example:

```
USES unit_1, unit_2, unit_3;
```

This enables you to access the following elements specified or declared in the interface section of the imported unit (e.g. ST source file, MCC source):

- User-defined data types (UDT)
- Unit variables and unit constants
- Programs, functions and function blocks
- Imported technology packages, libraries and units

You can use the imported elements as if they existed in the current unit.

Note

The keyword USES can only occur once in the interface section or in the implementation section of a unit. When multiple units are to be imported, enter them as a list separated by commas after the keyword USES.

The USES statement can appear in either the interface section or the implementation section of a unit. This has far-reaching implications:

Table 5-12 Implications regarding placement of USES statement in interface section or in implementation section

Effect	USES statement in the interface section	USES statement in the implementation section
Inheritance	The current unit continues exporting the imported unit; the imported unit is inherited by all other units that access the current unit. Example: 1. Unit B imports Unit A in the **interface** section. 2. Unit C in turn imports Unit B. 3. Then Unit C also imports Unit A automatically. A → B → C ⇒ A → C Because of inheritance, Unit A must not be imported explicitly into Unit C.	Inheritance is interrupted. Example: 1. Unit B imports Unit A in the **implementation** section. 2. Unit C in turn imports Unit B. 3. Then Unit C has no automatic access to Unit A. Unit C must explicitly import Unit A if it wants to access Unit A.
Variable declaration	The declaration of a unit variable of an imported data type is possible in: • Interface section • Implementation section	The declaration of a unit variable of an imported data type is only possible in the implementation section.

Note

You will find tips for use of unit variables in the SIMOTION Basic Functions Function Manual.

5.1.2.5 Example of an importing unit

Below is an example of an importing unit (*myUnit_B*). It imports the unit *myUnit_A* from Example of an exporting unit (Page 181).

Table 5-13 Example of an importing unit

```
UNIT myUnit_B;        // Optional, name of the ST source file
INTERFACE
    // ... if required, USES statement
    PROGRAM myProgram_B;
    // Specification of programs to be exported, FB, FC
    // Data types and unit variables
END_INTERFACE

IMPLEMENTATION
    USES myUnit_A;    // Specification of unit to be imported

    VAR_GLOBAL
        myInstance : myFB;      // Declaration of an instance
                                // of the imported FB
        mycolor : color;        // Declaration of a variable
                                // of the imported data type
    END_VAR

    PROGRAM myProgram_B

        mycolor := GREEN;       // Value assignment to a variable of the
                                // data type to be imported
        cycle := cycle + 1;     // Value assignment to
                                // imported variable
    END_PROGRAM
END_IMPLEMENTATION
```

5.2 Variables in SIMOTION

This summarizes the variables available in ST.

5.2.1 Variable model

The following table shows all the variable types available for programming with ST.

- System variables of the SIMOTION device and the technology objects
- Global user variables (I/O variables, device-global variables, unit variables)
- Local user variables (variables within a program, a function or a function block)

System variables

Variable type	Meaning
System variables of the SIMOTION device	Each SIMOTION device and technology object has specific system variables. These can be accessed as follows: • Within the SIMOTION device from all programs • From HMI devices You can monitor system variables in the symbol browser.
System variables of technology objects	

Global user variables

Variable type	Meaning
I/O variables	You can assign symbolic names to the I/O addresses of the SIMOTION device or the peripherals. This allows you to have the following direct accesses and process image accesses to the I/O: • Within the SIMOTION device from all programs • From HMI devices You create these variables in the symbol browser after you have selected the I/O element in the project navigator. You can monitor I/O variables in the symbol browser.
Global device variables	User-defined variables which can be accessed by all SIMOTION device programs and HMI devices. You create these variables in the symbol browser after you have selected the GLOBAL DEVICE VARIABLES element in the project navigator. Global device variables can be defined as retentive. This means that they will remain stored even when the SIMOTION device power supply is disconnected. You can monitor global device variables in the symbol browser.

Variable type	Meaning
Unit variables	User-defined variables that all programs, function blocks, and functions (e.g. ST source, MCC source, LAD/FBD source) can access within a unit. Declare these variables in the unit: • In the interface section: You can import these variables into other units (ST source files, MCC source files, LAD/FBD source files) and they are also available on HMI devices as standard. • In the implementation section: You can only access these variables within the associated unit. You can declare unit variables as retentive. This means that they will remain stored even when the SIMOTION device power supply is disconnected. You can monitor unit variables in the symbol browser.

Local user variables

Variable type	Meaning
	User-defined variables which can be accessed from within the program (or function, function block) in which they were defined.
Variable of a program (program variable)	Variable is declared in a program. The variable can only be accessed within this program. A differentiation is made between static and temporary variables: • Static variables are initialized according to the memory area in which they are stored. Specify this memory area by means of a compiler option. By default, the static variables are initialized depending on the task to which the program is assigned (see *SIMOTION Basic Functions* Function Manual). You can monitor static variables in the symbol browser. • Temporary variables are initialized every time the program in a task is called. Temporary variables cannot be monitored in the symbol browser.
Variable of a function (FC variable)	Variable is declared in a function (FC). The variable can only be accessed within this function. FC variables are temporary; they are initialized each time the FC is called. They cannot be monitored in the symbol browser.
Variable of a function block (FB variable)	Variable is declared in a function block (FB) source. The variable can only be accessed within this function block. A differentiation is made between static and temporary variables: • Static variables retain their value when the FB terminates. They are initialized only when the instance of the FB is initialized; this depends on the variable type with which the instance of the FB was declared. You can monitor static variables in the symbol browser. • Temporary variables lose their value when the FB terminates. The next time the FB is called, they are reinitialized. Temporary variables cannot be monitored in the symbol browser.

Further information is available from the following sources:

- In the corresponding list manuals, you can find the compressed information on all system variables of the SIMOTION technology packages and SIMOTION devices.
- For more details on the use of system variables of technology objects, please refer to the *SIMOTION Motion Control Technology Objects* Function Manuals.

- In the SIMOTION Basic Functions Function Manual you can find information on how to access system variables and configuration data.
- This documentation contains information on:
  - Access to I/O addresses with I/O variables (see Direct access and process image of cyclic tasks (Page 214))
  - Process image access (see),
  - Creation and use of global device variables (see Use of global device variables (Page 193)),
  - Use of unit variables and local variables (static and temporary variables).

Note

Please note that downloading the ST source file to the target system and running tasks affect variable initialization and thus the contents of the variables, see Time of the variable initialization (Page 200).

See also

Access to fixed process image of the BackgroundTask (Page 220)

5.2.1.1 Unit variables

Unit variables are valid throughout the entire ST source file, i.e. they can be accessed in any source file section.

Unit variables are declared in the interface and/or implementation section of an ST source file; the location of the declaration determines the validity of the unit variable:

- If you declare the unit variables in the interface section, you create variables that can be used in other program sources (e.g. ST source files, MMC units). For more on importing and exporting between program source files, see Import and export between ST source files (Page 179).

  By default, these unit variables are also available on HMI devices. The total size of the unit variables that can be exported to HMI devices is limited to 64 KB per unit.
- If you declare the unit variables in the implementation section, you create variables that can be used by all program organization units (POUs) of the current source file.

You can change the default setting for the HMI export of the unit variables using a pragma within a declaration block, see Variables and HMI devices (Page 208) and Controlling compiler with attributes (Page 247).

You can define unit variables with different behavior, e.g. in case of power failure:

- Non-retentive unit variables (keyword VAR_GLOBAL): its value is lost in the event of a power failure.
- Retentive unit variables (keyword VAR_GLOBAL RETAIN): its value remains in the event of a power failure.
- Unit constants (keyword VAR_GLOBAL CONSTANT): its value is retained unchanged (see Constants (Page 111)).

You will find tips for the efficient use of unit variables in the *SIMOTION Basic Functions* Function Manual.

5.2.1.2 Non-retentive unit variables

Non-retentive unit variables lose their value in the event of a power failure.

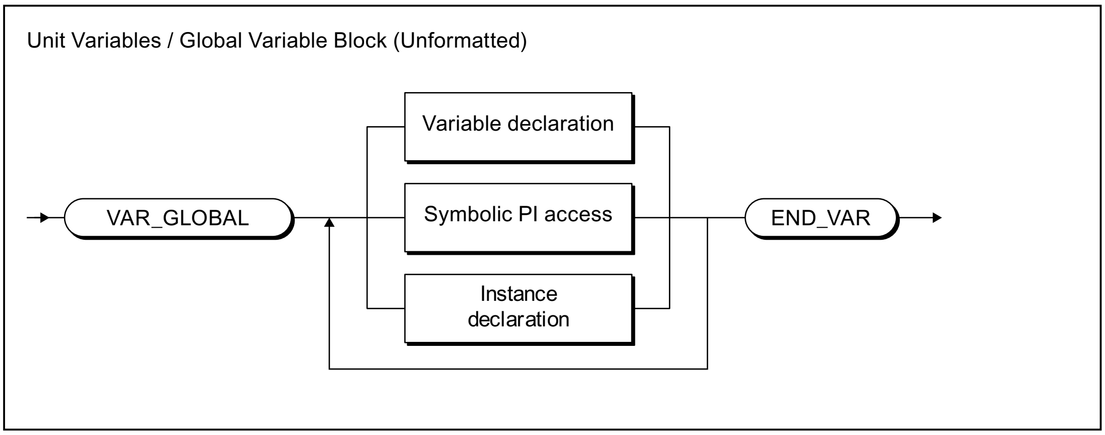


Figure 5-1 Syntax: Unit variables

This declaration block may appear more than once within an interface or implementation section. You specify the variable name and data type for the variable declaration (see Overview of all variable declarations (Page 106) and Initialization of variables or data types (Page 107)).

For the scope of the declaration and the HMI export, see Unit variables (Page 186).

Note

For initialization of the non-retentive unit variables:

- See Initialization of non-retentive global variables (Page 202).
- The behavior during downloading can be set (**Options > Settings** menu command, **Project Download** tab, **Initialize all non-retentive device-global variables and program data** checkbox)
- The type of version ID and therefore the initialization behavior on downloading depends on the SIMOTION Kernel version. For details, see Version ID of global variables and their initialization during download (Page 207).

Table 5-14 Examples of non-retentive unit variables

```
INTERFACE
    VAR_GLOBAL    //These variables can be exported.
        rotation1        : INT;
        field1           : ARRAY [1..10] OF REAL;
        flag1            : BOOL;
        motor1           : motor;    // Instance declaration
    END_VAR
END_INTERFACE
IMPLEMENTATION
    VAR_GLOBAL    //These variables cannot be exported
                  //  MotionTask.
        rotation2        : INT;
        field2           : ARRAY [1..10] OF REAL;
        flag2            : BOOL;
        motor2           : motor;    // Instance declaration
    END_VAR
END_IMPLEMENTATION
```

5.2.1.3 Retentive unit variables

Retentive unit variables permit permanent storage of variable values even throughout a power failure.

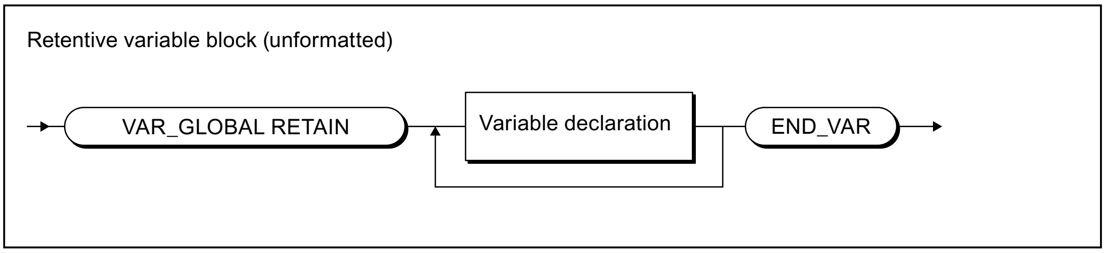


Figure 5-2 Syntax: Retentive variable block

This declaration block may appear more than once within an interface or implementation section. You specify the variable name and data type for the variable declaration (see Overview of all variable declarations (Page 106) and Initialization of variables or data types (Page 107)).

For the scope of the declaration and the HMI export, see Unit variables (Page 186).

Note

- For initialization of the retentive unit variables:
  - See Initialization of retentive global variables (Page 201).
  - The behavior during downloading can be set (**Options > Settings** menu command, **Project Download** tab, **Initialize all retentive device-global variables and program data** checkbox).
  - The type of version ID and therefore the initialization behavior on downloading depends on the SIMOTION Kernel version. For details, see Version ID of global variables and their initialization during download (Page 207).
- The amount of memory available for retentive variables depends on the device (see quantity framework in the SIMOTION SCOUT Configuration Manual).

  To make efficient use of limited memory space, use the memory in a single ST source file and sort the variables in descending order!
- Check the capacity utilization of the retentive memory in SIMOTION SCOUT.

  In online mode, call the **device diagnostics** of the SIMOTION device to be checked (see online help). In the **System utilization** tab under **Retentive data**, you can see how much memory is available.

Table 5-15 Examples of retentive variables

```
VAR_GLOBAL RETAIN
    Measuring field : ARRAY[1.0.10] OF REAL;
    Pass : INT;
    Switch: BOOL;
END_VAR
```

5.2.1.4 Local variables (static and temporary variables)

Local variables are valid only in the source file section (e.g. program, FC or FB) in which they were declared. We distinguish between the following:

- Static variables (Page 191):

  Static variables retain their value over all passes of the source file section (block memory).
- Temporary variables (Page 192):

  Temporary variables are initialized each time the source file section is called again.

See also: Initialization of local variables (Page 204).

Note

Local variables cannot be accessed outside the source file section in which they were declared.

The following table provides an overview of the declaration of static and temporary variables. It shows the source file sections in which these variables can be declared and the keywords that can be used to declare them.

Table 5-16 Keywords for declaring static and temporary variables depending on source file section.

Source file section	Keywords for the declaration	
	Static variables	Temporary variables
Function	–	VAR / END_VAR or VAR_INPUT / END_VAR or VAR_IN_OUT / END_VAR[2]
Expression	–	VAR / END_VAR or VAR_INPUT / END_VAR or VAR_IN_OUT / END_VAR[2]
Function block	VAR / END_VAR[1] or VAR_INPUT / END_VAR[1] or VAR_OUTPUT / END_VAR[1]	VAR_TEMP / END_VAR or VAR_IN_OUT / END_VAR[2]
Program	VAR / END_VAR[3]	VAR_TEMP / END_VAR

[1] The initialization of the variable depends on initialization of the declared instance. See Initialization of instances of function blocks (FBs) (Page 205).

[2] The reference (pointer) for the transferred variable is temporary.

[3] The initialization of the variables depends on the memory area in which they are stored. See Initialization of static program variables (Page 204).

Note

Please note that downloading the ST source file to the target system and running tasks affect variable initialization and thus the contents of the variables, see Time of the variable initialization (Page 200).

Table 5-17 Examples of static and temporary variables

```
IMPLEMENTATION
    FUNCTION testFkt
        VAR            // Declaration of temporary variables
            flag : BOOL;
        END_VAR
    END_FUNCTION
    FUNCTION_BLOCK testFbst;
        VAR            // Declaration of static variables
            rotation1        : INT;
        END_VAR

        VAR_TEMP       // Declaration of temporary variables
            help1, help2 : REAL;
        END_VAR
    END_FUNCTION_BLOCK
    PROGRAM testPrg;
        VAR            // Declaration of static variables
            rotation2        : INT;
        END_VAR

        VAR_TEMP       // Declaration of temporary variables
            help1, help2 : REAL;
        END_VAR
    END_PROGRAM
END_IMPLEMENTATION
```

5.2.1.5 Static variables

Static variables retain their most recent value when the source file section is exited. This value is used again at the next call.

The following source file sections contain static variables:

- Programs
- Function blocks

Static variables are declared in a static variable block.

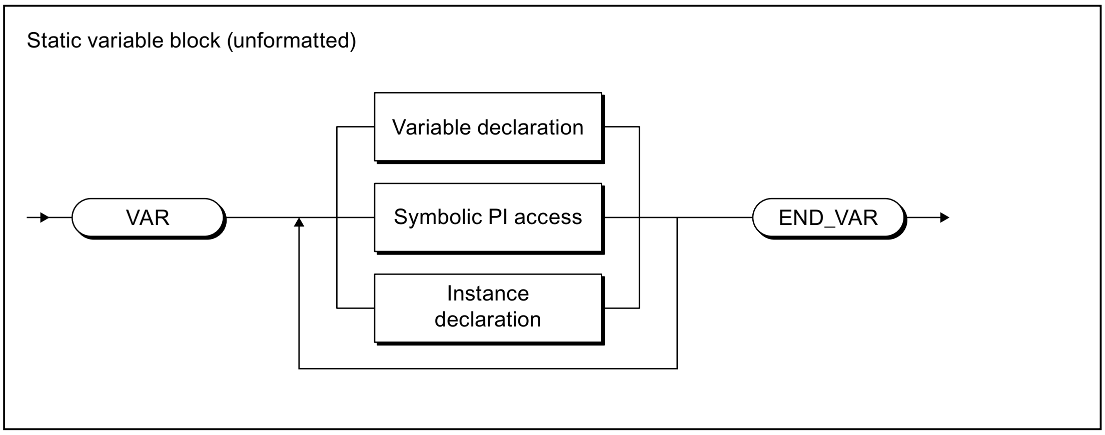


Figure 5-3 Syntax: Static variable block

You can do the following in the static variable block, according to the syntax in the figure:

- Declare variables (name and data type), optionally with initialization.
- Declare symbolic accesses to the process image of the BackgroundTask.
- Declare instances of the function blocks.

For initialization of the static variables:

- In programs: Depending on the execution behavior to which the program is assigned (see *SIMOTION Basic Functions* Function Manual).

  See also Initialization of static program variables (Page 204).
- In function blocks: Depending on the initialization of the declared instance.

  See also Initialization of instances of function blocks (FBs) (Page 205).

5.2.1.6 Temporary variables

Temporary variables are initialized each time the source file section is called. Their value is retained only during execution of the source file section.

The following source file sections contain temporary variables:

- Programs
- Function blocks
- Functions
- Expression

In functions and expressions, you declare temporary variables in the FB temporary variable block (see following figure):

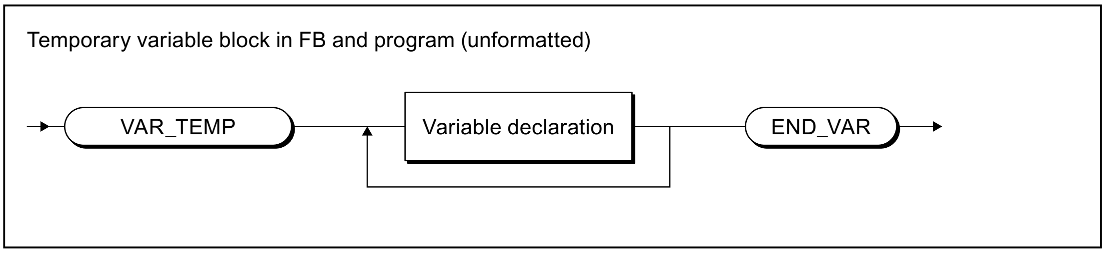


Figure 5-4 Syntax: Temporary variable block in the FB or program

In functions and expressions, you declare temporary variables in the FC temporary variable block (see following figure):

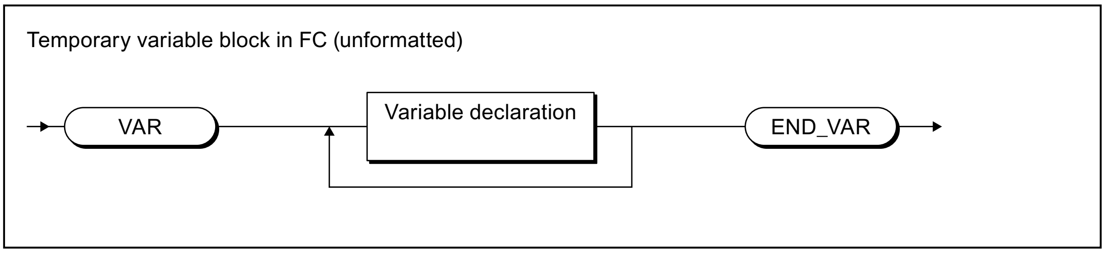


Figure 5-5 Syntax: Temporary variable block in an FC

5.2.2 Use of global device variables

Global device variables are user-defined variables that you can access from all program sources (e.g. ST source files, MCC units) of a SIMOTION device.

Global device variables are created in the symbol browser tab of the detail view; to do this, you must be working in offline mode.

Here is a brief overview of the procedure:

1. In the project navigator of SIMOTION SCOUT, select the **GLOBAL DEVICE VARIABLES** element in the SIMOTION device subtree.
2. In the detail view, select the **Symbol browser** tab and scroll down to the end of the variable table (empty row).
3. In the last (empty) row of the table, enter or select the following:
   - **Name** of variable
   - **Data type** of variable (only elementary data types are permitted)
4. Optionally, you can make the following entries:
   - Selection of **Retain** checkbox (This declares the variable as retentive, so that its value will be retained after a power failure.)
   - **Array length** (array size)
   - **Initial value** (if array, for each element)
   - **Display format** (if array, for each element)

You can now access this variable using the symbol browser or any program of the SIMOTION device.

In ST source files, you can use a global device variable, just like any other variable.

Note

If you have declared unit variables or local variables of the same name (e.g. *var-name*), specify the global device variable with *_device.var-name*.

An alternative to global device variables is the declaration of unit variables in a separate unit, which is imported into other units. This has the following advantages:

1. Variable structures can be used.
2. The initialization of the variables during the STOP-RUN transition is possible (via Program in StartupTask).
3. For newly created global unit variables, a download in RUN is also possible.

Please refer to the SIMOTION Basic Functions Function Manual.

5.2.3 Memory ranges of the variable types

The different variable types are stored in different memory areas, which are initialized at different times. The table shows:

- The available memory areas for variable types that are declared in ST source files (possibly dependent on the version of the SIMOTION Kernel).
- The initialization time for each memory area.

An explanation using an example is contained in the Example for memory areas, valid as of Kernel V3.1 (Page 196) section.

Table 5-18 Memory ranges assigned to different variable types and their initialization

Memory area	Assigned variable types	Initialization[4]
Retentive memory	Retentive unit variables	During download using the download settings
User memory of unit	• Non-retentive unit variables • Function block instances declared with VAR_GLOBAL, including the associated static variables (VAR, VAR_INPUT, VAR_OUTPUT) Also for the activated "Create program instance data only once" compiler option (Page 44): • Local variables of the unit programs declared with VAR • Function block instances declared with VAR_GLOBAL, including the associated static variables (VAR, VAR_INPUT, VAR_OUTPUT)	• When the device is switched on • During download using the download settings • As of Version V4.1 of the SIMOTION Kernel: For transition to the RUN mode when the associated declaration block specifies the following pragma: { BlockInit_OnDeviceRun := ALWAYS; } See also Controlling compiler with attributes (Page 247)

Memory area	Assigned variable types	Initialization[4]
User memory of task	For the deactivated "Create program instance data only once" compiler option (Page 44) (default): • Local variables declared with VAR of the assigned programs • Function block instances declared with VAR within the assigned programs, including the associated static variables (VAR, VAR_INPUT, VAR_OUTPUT)	According to execution behavior of task: • Sequential tasks: Each time task is started • Cyclic tasks: For CPU transition to the RUN mode
Local data stack of the task (as of Version V3.1 of the SIMOTION kernel)[2]	• Reference (pointer) to the program called in the task • Local variables declared with VAR_TEMP of the program called in the task	On each call of the program in the task
	• Reference (pointer) to called function block instances • Local variables of function blocks declared with VAR_TEMP • In/out parameters of function blocks declared with VAR_IN_OUT[1]	Each time the function block instance is called
	• Variables of called functions declared with VAR, VAR_INPUT or VAR_IN_OUT[1] • Return value of called functions	Each time the function is called
Local data stack of the task (up to Version V3.0 of the SIMOTION kernel)[3]	• Copied data of the program called in the task, including all associated variables (VAR, VAR_TEMP)	On each call of the program in the task
	• Copied data from instances of called function blocks, including all associated variables (VAR, VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT[1], VAR_TEMP)	Each time the function block instance is called
	• Variables of called functions declared with VAR, VAR_INPUT or VAR_IN_OUT[1] • Return value of called functions	Each time the function is called

[1] References (pointers) to the transferred variables.

[2] Also for the use of libraries that have been compiled with reference to the SIMOTION device and the associated version of the SIMOTION kernel (as of Version V3.1). See also Memory requirement of the variables on the local data stack (as of Kernel V3.1).

[3] Also for the use of libraries that have been compiled device-dependent (i.e. without reference to a SIMOTION device and a SIMOTION Kernel version). See also Memory requirement of the variables on the local data stack (up to Kernel V3.0).

[4] For a detailed description of the initialization behavior of the individual variable types, see Time of the variable initialization (Page 200).

5.2.3.1 Example of memory areas, valid as of Kernel V3.1

Table 5-19 Example of memory ranges of the variable types, as of Kernel V3.1 (Part 1)

```
INTERFACE
// The statements in the interface section specify,
// what source content is exported.
    FUNCTION FC1;
    FUNCTION_BLOCK FB1;
    PROGRAM p1;

    // Unit variables of the interface section are also visible
    // on HMI devices.
    VAR_GLOBAL                // Non-retentive unit variables
                              // are present in the UNIT user memory
        u1_if : INT;
    END_VAR
    VAR_GLOBAL CONSTANT       // Unit constants are located
                              // in the unit user memory
    END_VAR
    VAR_GLOBAL RETAIN         // Retentive unit variables are located
                              // in the retentive (power-fail-safe) memory
    END_VAR
END_INTERFACE

IMPLEMENTATION
// The implementation section contains the executable code sections
// in different program organization units (POU)
// A POU can be a program, FC, or FB.
    // Unit variables of the implementation section can only be used
    // within the source file.
    VAR_GLOBAL                // Non-retentive unit variables are located
                              // in the unit user memory
        u1_glob : INT;
    END_VAR
    VAR_GLOBAL CONSTANT       // Unit constants are located
                              // in the unit user memory
    END_VAR
    VAR_GLOBAL RETAIN         // Retentive unit variables are located
                              // in the retentive (power-fail-safe) memory
    END_VAR
//----------------------------------------------------------------------
```

Table 5-20 Example of memory ranges of the variable types, as of Kernel V3.1 (Part 2)

```
// Continuation
//----------------------------------------------------------------------
    FUNCTION_BLOCK FB1 // Declaration of an instance
    // instance determines where its data are located:
    // - as VAR_GLOBAL in a unit:
    // in the unit user memory
    // - as VAR in a program:
    // in the user memory of the task (default)
    // - As VAR in a function block:
    //     in the user memory of the unit or task,
    //     depending on the instance declaration of the higher-level FB
    // When the instance is called, a pointer to the instance data
    // is placed on the stack of the calling task

        VAR_INPUT        // Input parameters
                         // are in the user memory
                         // are written when the instance is called
            fb_in     : INT;
        END_VAR
        VAR_OUTPUT       // Output parameters
                         // are in the user memory
            fb_out    : INT;
        END_VAR
        VAR_IN_OUT       // In/out parameter
                         // references are in the user memory
                         // are written when the instance is called
            fb_in_out : INT;
        END_VAR

        VAR              // Static variables
                         // are in the user memory
                         // can be used locally in the FB
             fb_var1   : INT;
        END_VAR

        VAR_TEMP         // Temporary variables
                         // are on the stack of the calling task
                         // are initialized on each call
            fb_temp1  : INT;
        END_VAR

        // Code is in the user memory of the unit
        fb_var1   := fb_var1 + 1;
        fb_out    := fb_var1;
        fb_temp1  := fb_in_out;
        fb_in_out := fb_temp1 + fb_in;
    END_FUNCTION_BLOCK
//----------------------------------------------------------------------
```

Table 5-21 Example of memory ranges of the variable types, as of Kernel V3.1 (Part 3)

```
// Continuation
//------------------------------------------------------------------------
    FUNCTION FC1 : INT     // The function data is on the
        // stack of the calling task; they are initialized each time
        // the function is called.
        // The return value is on the stack of the calling task

        VAR_INPUT        // Input parameters
                         // are on the stack of the calling task
                         // are written when the function is called
            fc_in   : INT;
        END_VAR

        VAR              // Temporary variables
                         // are on the stack of the calling task
            fc_var  : INT;
        END_VAR
        // Code is in the user memory of the unit
        fc_var := 567;
        fc1 := fc_in + fc_var;
    END_FUNCTION

    PROGRAM p1
        VAR              // By default, variables are located in the
                         // in the user memory of the task
            p_var   : INT;
            p_varFB : FB1;
        END_VAR

        VAR_TEMP         // Temporary variables
                         // are on the stack of the task,
                         // are initialized on each task pass
            p_temp  : INT;
        END_VAR

        // Code is in the user memory of the unit
        p_temp  := p_var;
        p_varFB (fb_in_out := p_temp);
        u1_glob := 4711;
    END_PROGRAM
END_IMPLEMENTATION
```

5.2.3.2 Memory requirement of the variables on the local data stack (Kernel V3.1 and higher)

The variables stored on the local data stack of a task are listed in Memory ranges of the variable types (Page 194). You set the stack size for each task in the task configuration.

Note the following for memory requirements in the local stack:

- Temporary local variables require their own size on the stack.
- Global variables and static local variables do not require any resources on the stack.

  If you are using them as input parameters for a function, however, they require their own data size on the stack.
- Even if a function is called more than once in a task, it only uses the stack's resources once.
- Variables of type BOOL require one byte on the stack.

Note

The above details are also true for the use of libraries that have been compiled with reference to the SIMOTION device and the associated version of the SIMOTION kernel (as of Version V3.1).

NOTICE

If the library is not device-dependent (i.e. compiled without reference to a SIMOTION device or SIMOTION Kernel version): These libraries are compiled compatible to the permitted versions of the SIMOTION kernel.

Consequently, the variables of program organization units (POU) called from these libraries occupy the local data stack as for versions of the SIMOTION kernel up to V3.0. See Memory requirement of the variables on the local data stack (up to Kernel V3.0) (Page 199).

You can obtain information about the memory requirements of a POU in the local data stack using the Program Structure (Page 239) function.

5.2.3.3 Memory requirement of variables on local data stack (Kernel V3.0 and below)

The variables stored on the local data stack of a task are listed in Memory ranges of the variable types (Page 194). You set the stack size for each task in the task configuration.

Note the following for memory requirements in the local stack:

- Static local variables in programs require double their size on the stack.
- Static local variables in FBs require several times their size on the stack, depending on the calling depth.
- Temporary local variables (in programs, FBs, and FCs) require their own size on the stack.
- Global variables do not occupy any stack memory space.

  If you are using them as input parameters for a function or function block, however, they will occupy their usual space on the stack.

- Even if a function is called more than once in a task, it only uses the stack's resources once.
- Variables of type BOOL require one byte on the stack.

NOTICE
When a function block instance is called, all instance data is copied to the local data stack, even if the instance is declared as a VAR_GLOBAL instance. If the library is not device-dependent (i.e. compiled without reference to a SIMOTION device or SIMOTION Kernel version): These libraries are compiled compatible to the permitted versions of the SIMOTION kernel. Consequently, the variables of program organization units (POU) called from these libraries occupy the local data stack as described in this section. The memory requirement on the local data stack is significantly larger than for versions of the SIMOTION kernel as of V3.1, see Memory requirement of the variables on the local data stack (as of Kernel V3.1) (Page 199). Take this into consideration for setting the stack size for the task configuration!

You can obtain information about the memory requirements of a POU in the local data stack using the Program Structure (Page 239) function.

5.2.4 Time of the variable initialization

The timing of the variable initialization is determined by:

- Memory area to which the variable is assigned
- Operator actions (e.g. source file download to the target system)
- Execution behavior of the task (sequential, cyclic) to which the program was assigned.

All variable types and the timing of their variable initialization are shown in the following tables. You will find basic information about tasks in the *SIMOTION Basic Functions* Function Manual.

The behavior for variable initialization during download can be set: To do this, as a default setting select the **Options > Settings** menu and the **Download** tab or define the setting during the current download.

Note

You can upload values of unit variables or global device variables from the SIMOTION device into SIMOTION SCOUT and save them in XML format.

1. Save the required data segments of the unit variables or global device variables as a data set with the function *_saveUnitDataSet*.
2. Use the **Save variables** function in SIMOTION SCOUT.

You can use the **Restore variables** function to download these data sets and variables back to the SIMOTION device.

For more information, refer to the SIMOTION SCOUT Configuration Manual.

This makes it possible, for example, to obtain this data, even if it is initialized by a project download or if it becomes unusable (e.g. due to a version change of SIMOTION SCOUT).

5.2.4.1 Initialization of retentive global variables

Retentive variables retain their last value after a loss of power. All other data is reinitialized when the device is switched on again.

Retentive global variables are initialized:

- When the backup or buffer for retentive data fails.
- When the firmware is updated.
- When a memory reset (MRES) is performed.
- With the restart function (Del. SRAM) in SIMOTION P350.
- By applying the *_resetUnitData* function (as of kernel V3.2), possible selectively for different data segments of the retentive data.
- When a download is performed according to the following description.

Table 5-22 Initializing retentive global variables during download

Variable type	Time of the variable initialization
Retentive global device variables	The behavior when downloading depends on the *Initialization of all retentive global device variables and program data* setting[1]: • **Yes**[2]: All retentive global device variables are initialized. • **No**[3]: – As of version V3.2 of the SIMOTION Kernel: **Separate version ID** for retentive global device variables. If the version ID is changed, the retentive global device variables are initialized. – Up to Version V3.1 of the SIMOTION kernel: **Joint version ID** for all global device variables (retentive and non-retentive). If the version ID is changed, all global device variables are initialized. See: Version ID of global variables and their initialization during download (Page 207).

Variable type	Time of the variable initialization
Retentive unit variables	The behavior when downloading depends on the *Initialization of all retentive global device variables and program data* setting[1]: • **Yes**[2]: All retentive unit variables (all units) are initialized. • **No**[3]: – As of version V3.2 of the SIMOTION Kernel: **Separate version ID** for each individual data block (= declaration block)[4] of the retentive unit variables in the interface or implementation section. If the version identification is changed, only the associated data block will be initialized[5]. – Up to Version V3.1 of the SIMOTION kernel: **Common version ID** for all unit variables (retentive and non-retentive, in the interface and implementation section) of a unit. If the version ID is changed, all unit variables of this unit are initialized. See: Version ID of global variables and their initialization during download (Page 207).

[1] Default setting in the **Options > Settings** menu, **Download** tab,
or the current setting for the download.

[2] The corresponding checkbox is active.

[3] The corresponding checkbox is inactive.

[4] Several data blocks for retentive unit variables in the interface or implementation section can be declared only in the SIMOTION ST programming language. For the SIMOTION MCC and SIMOTION LAD/FBD programming languages, only one data block for retentive unit variables will be created in the interface or implementation section.

[5] Also for the download in RUN, provided the associated prerequisites have been satisfied and the following attribute has been specified in the associated declaration block within a pragma (only for the SIMOTION ST programming language):
{ BlockInit_OnChange := TRUE; }.
For the download in RUN, see the SIMOTION Basic Functions Function Manual.

5.2.4.2 Initialization of non-retentive global variables

Non-retentive global variables lose their value during power outages. They are initialized:

- For the Initialization of retentive global variables (Page 201), e.g. during a firmware update or general reset (MRES).
- During power up.
- By applying the *_resetUnitData* function (as of kernel V3.2), possible selectively for different data segments of the non-retentive data.
- During the download in accordance with the description on the following table.
- Only as of Version V4.1 of the SIMOTION Kernel and for non-retentive unit variables:

  For transition to the RUN mode when the associated declaration block within a pragma specifies the following attribute (only for SIMOTION ST programming language): { BlockInit_OnDeviceRun := ALWAYS; }

Table 5-23 Initializing non-retentive global variables during download

Variable type	Time of the variable initialization
Non-retentive global device variables	The behavior when downloading depends on the *Initialization of all non-retentive global device variables and program data* setting[1]: • **Yes**[2]: All non-retentive global device variables are initialized. • **No**[3]: – As of version V3.2 of the SIMOTION Kernel: **Separate version ID** for non-retentive global device variables. If the version ID is changed, the non-retentive global device variables are initialized. – Up to Version V3.1 of the SIMOTION kernel: **Joint version ID** for all global device variables (retentive and non-retentive). If the version ID is changed, all global device variables are initialized. See: Version ID of global variables and their initialization during download (Page 207).
Non-retentive unit variables	The behavior when downloading depends on the *Initialization of all non-retentive global device variables and program data* setting[1]: • **Yes**[2]: All non-retentive unit variables (all units) are initialized. • **No**[3]: – As of version V3.2 of the SIMOTION Kernel: **Separate version ID** for each individual data block (= declaration block)[4] of the non-retentive unit variables in the interface or implementation section. If the version identification is changed, only the associated data block will be initialized[5]. – Up to Version V3.1 of the SIMOTION kernel: **Common version ID** for all unit variables (retentive and non-retentive, in the interface and implementation section) of a unit. If the version ID is changed, all unit variables of this unit are initialized. See: Version ID of global variables and their initialization during download (Page 207).

[1] Default setting in the **Options > Settings** menu, **Download** tab,
or the current setting for the download.

[2] The corresponding checkbox is active.

[3] The corresponding checkbox is inactive.

[4] Several data blocks for non-retentive unit variables in the interface or implementation section can be declared only in the SIMOTION ST programming language. For the SIMOTION MCC and SIMOTION LAD/FBD programming languages, only one data block for non-retentive unit variables will be created in the interface or implementation section.

[5] Also for the download in RUN, provided the associated prerequisites have been satisfied and the following attribute has been specified in the associated declaration block within a pragma (only for the SIMOTION ST programming language):
{ BlockInit_OnChange := TRUE; }.
For the download in RUN, see the SIMOTION Basic Functions Function Manual.

5.2.4.3 Initialization of local variables

Local variables are initialized:

- For the initialization of retentive unit variables (Page 201).
- For the initialization of non-retentive unit variables (Page 202).
- Also, according to the following description:

Table 5-24 Initialization of local variables

Variable type	Time of the variable initialization
Local program variables	Local variables of programs are initialized differently: • Static variables (VAR) are initialized according to the memory area in which they are stored. See: Initialization of static program variables (Page 204). • Temporary variables (VAR_TEMP) are initialized every time the program of the task is called.
Local variables of function blocks (FB)	Local variables of function blocks are initialized differently: • Static variables (VAR, VAR_IN, VAR_OUT) are only initialized when the FB instance is initialized. See: Initialization of instances of function blocks (FBs) (Page 205). • Temporary variables (VAR_TEMP) are initialized every time the FB instance is called.
Local variables of functions (FC)	Local variables of functions are temporary and are initialized every time the function is called.

Note

You can obtain information about the memory requirements of a POU in the local data stack using the Program Structure (Page 239) function.

5.2.4.4 Initialization of static program variables

The following versions affect the following static variables:

- Local variables of a unit program declared with VAR
- Function block instances declared with VAR within a unit program, including the associated static variables (VAR, VAR_INPUT, VAR_OUTPUT).

The initialization behavior is determined by the memory area in which the static variables are stored. This is determined by the "Create program instance data only once" (Page 44) compiler option.

- For the deactivated "Create program instance data only once" compiler option (default):

  The static variables are stored in the user memory of each task, which is assigned to the program.

The initialization of the variables thus depends on the execution behavior of the task to which the program is assigned (see SIMOTION Basic Functions Function Manual):

  – Sequential tasks (MotionTasks, UserInterruptTasks, SystemInterruptTasks, StartupTask, ShutdownTask): The static variables are initialized every time the task is started.
  – Cyclic tasks (BackgroundTask, SynchronousTasks, TimerInterruptTasks): The static variables are initialized only during transition to RUN mode.

- For the activated "Create program instance data only once" compiler option:

  This setting is necessary, for example, if a program is to be called within a program.

  The static variables are stored only once in the user memory of the task.

  – They are thus initialized together with the non-retentive unit variables, see Initialization of non-retentive global variables (Page 202).
  – Only as of Version V4.1 of the SIMOTION Kernel:

    In addition, they can be initialized during transition to RUN mode. To do this, the following attribute must be specified in the associated declaration block within a pragma (only SIMOTION ST programming language):
    { BlockInit_OnDeviceRun := ALWAYS; }.

5.2.4.5 Initialization of instances of function blocks (FBs)

The initialization of a function block instance (Page 157) is determined by the location of its declaration:

- Global declaration (within VAR_GLOBAL/END_VAR in the interface of implementation section):

  Initialization as for a non-retentive unit variable, see Initialization of non-retentive global variables (Page 202).

- Local declaration in a program (within VAR / END_VAR):

  Initialization as for static variables of programs, see Initialization of static variables of programs (Page 204).

- Local declaration in a function block (within VAR / END_VAR):

  Initialization as for an instance of this function block.

- Declaration as in/out parameter in a function block or a function (within VAR_IN_OUT / END_VAR):

  For the initialization of the POU, only the reference (pointer) will be initialized with the instance of the function block remaining unchanged.

Note

You can obtain information about the memory requirements of a POU in the local data stack using the Program Structure (Page 239) function.

5.2.4.6 Initialization of system variables of technology objects

The system variables of a technology object are usually not retentive. Depending on the technology object, a few system variables are stored in the retentive memory area (e.g. absolute encoder calibration).

The initialization behavior (except in the case of download) is the same as for retentive and non-retentive global variables. See Initialization of retentive global variables (Page 201) and Initialization of non-retentive global variables (Page 202).

The behavior during the download is shown below for:

- Non-retentive system variables
- Retentive system variables

Table 5-25 Initializing technology object system variables during download

Variable type	Time of the variable initialization
Non-retentive system variables	Behavior during download, depending on the *Initialization of all non-retentive data for technology objects* setting[1]: • Yes[2]: All technology objects are initialized. – All technology objects are restructured and all non-retentive system variables are initialized. – All technological alarms are cleared. • No[3]: Only technology objects changed in SIMOTION SCOUT are initialized. – The technology objects in question are restructured and all non-retentive system variables are initialized. – All alarms that are pending on the relevant technology objects are cleared. – If an alarm that can only be acknowledged with *Power On* is pending on a technology object that will not be initialized, the download is aborted.
Retentive system variables	Only if a technology object was changed in SIMOTION SCOUT, will its retentive system variables be initialized. The retentive system variables of all other technology objects are retained (e.g. absolute encoder calibration).
[1] Default setting in the Options > Settings menu, Download tab, or the current setting for the download. [2] The corresponding checkbox is active. [3] The corresponding checkbox is inactive.	

5.2.4.7 Version ID of global variables and their initialization during download

Table 5-26 Version ID of global variables and their initialization during download

Data segment		As of Version V3.2 of the SIMOTION kernel	Up to Version V3.1 of the SIMOTION kernel
Global device variables			
	Retentive global device variables Non-retentive global device variables	• **Separate version ID** for each data segment of the global device variables. • The version identification of the data segment changes for: – Add or remove a variable within the data segment – Change of the identifier or the data type of a variable within the data segment • This version ID does not change on: – Changes in the other data segment – Changes to initialization values[1] • During downloading[2], the rule is: **Initialization of a data segment** only if its version ID has changed. • Use of the functions for data backup and initialization possible.	• **Common version ID** for all data segments of the global device variables. • This version ID changes when the variable declaration is changed in a data segment. • During downloading[2], the rule is: **Initialization of all data segments** if the version ID changes. • Use of the functions for data backup not possible.
Unit variables of a unit			
	Retentive unit variables in the interface section Retentive unit variables in the implementation section Non-retentive unit variables in the interface section Non-retentive unit variables in the implementation section	• Several data blocks (= declaration blocks)[3] in each data segment possible. • **Own version ID** for each data block. • The version identification of the data block changes for: – Add or remove a variable in the associated declaration block – Change of the identifier or the data type of a variable in the associated declaration block – Change of a data type definition (from a separate or imported[4] unit) used in the associated declaration block – Add or remove declaration blocks within the same data segment **before** the associated declaration block • This version ID does not change on: – Add or remove declaration blocks in other data segments – Add or remove declaration blocks within the same data segment **after** the associated declaration block – Changes in other data blocks – Changes to initialization values[1] – Changes to data type definitions that are not used in the associated data block – Changes to functions • During downloading[2], the rule is: **Initialization of a data block** only if its version ID has changed.[5] • Functions for data backup and initialization take into account the version ID of the data blocks.	• One data block in each data segment (also for several declaration blocks)[3] • **Common version ID** for all global declarations in a unit. • This version ID changes in response to the following changes: – Variable declaration in a data segment – Declaration of global data types in the unit – Declaration in the interface section of an imported[4] unit. • During downloading[2], the rule is: **Initialization of all data segments** if the version ID changes. • Use of the functions for data backup only possible for: Non-retentive unit variables in the interface section

Data segment	As of Version V3.2 of the SIMOTION kernel	Up to Version V3.1 of the SIMOTION kernel

[1] Changed initialization values are not effective until the data block or data segment in question is initialized.

[2] If *Initialization of all retentive global device variables and program data* = No and *Initialization of all non-retentive global device variables and program data* = No.
In the case of other settings: See the sections "Initialization of retentive global variables (Page 201)" and "Initialization of non-retentive global variables (Page 202)".

[3] Several declaration blocks per data segment are possible only in the SIMOTION ST programming language. For the SIMOTION MCC and SIMOTION LAD/FBD programming languages, only one declaration block per data segment will be created.

[4] The import of units depends on the programming language, refer to the associated section (Page 181).

[5] Also for the download in RUN, provided the associated prerequisites have been satisfied and the following attribute (Page 247) has been specified in the associated declaration block within a pragma (Page 242) (only for the SIMOTION ST programming language): { BlockInit_OnChange := TRUE; }.
For the download in RUN, see the SIMOTION Basic Functions Function Manual.

5.2.5 Variables and HMI devices

The following variables are exported to HMI devices where they are available:

- System variables of the SIMOTION device
- System variables of technology objects
- I/O variables
- Global device variables
- Retentive and non-retentive unit variables of the interface section (default setting).

  This default setting can be changed for each declaration block using the following pragma:

  ```
  { HMI_Export := FALSE; }
  ```

  The unit variables of such an identified declaration block are not exported to HMI devices. The HMI consistency check is also omitted for them during the download.

  See also Controlling compiler with attributes (Page 247).

The following variables are **not** exported to HMI devices and are **not** available there:

- Retentive and non-retentive unit variables of the implementation section (default setting).

  This default setting can be changed for each declaration block using the following pragma:

  ```
  { HMI_Export := TRUE; }
  ```

  The unit variables of such an identified declaration block are exported to HMI devices. Consequently, they are subject to the HMI consistency check during the download.

  See also Controlling compiler with attributes (Page 247).

- Local variables of a POU

NOTICE

The total size of the unit variables that can be exported to HMI devices is limited to 64 KB per unit.

The effect of the pragma `{ HMI_Export := FALSE; }` and `{ HMI_Export := TRUE; }` depends on the version of the SIMOTION Kernel:

- As of Version V4.1 of the SIMOTION Kernel:

  The pragma affects the export of the corresponding declaration block to HMI devices **and** the structure of the HMI address space:

  – Only those variables in declaration blocks exported to HMI devices occupy the HMI address space.

  – Within the HMI address space, the variables are arranged according to order of their declaration.
- Up to version V3.2 or V4.0 of the SIMOTION kernel:

  The pragma affects **only** the export of the corresponding declaration block to HMI devices.

  The HMI address space is also occupied by unit variables of the interface section whose declaration blocks are not assigned to HMI devices.

  Within the HMI address space, the variables are sorted in the following order:

  – Retentive unit variables of the interface section (exported and not exported).

  – Retentive unit variables of the implementation section (only exported).

  – Non-retentive unit variables of the interface section (exported and not exported).

  – Non-retentive unit variables of the implementation section (only exported).

  Within these segments, the variables are arranged according to order of their declaration.
- Up to Version V3.1 of the SIMOTION kernel:

  The pragma has no effect.

Table 5-27 Example for the control of the HMI export with the corresponding pragma

```
INTERFACE
    VAR_GLOBAL
        // HMI export
        x1 : DINT;
    END_VAR
    VAR_GLOBAL
        { HMI_Export := FALSE; }
        // No HMI export
        x2 : DINT;
    END_VAR
    // ...
END_INTERFACE

IMPLEMENTATION
    VAR_GLOBAL
        // No HMI export
        y1 : DINT;
    END_VAR
    VAR_GLOBAL
        { HMI_Export := TRUE; }
        // HMI export
        y2 : DINT;
    END_VAR
    // ...
END_IMPLEMENTATION
```

5.3 Access to inputs and outputs (process image, I/O variables)

5.3.1 Overview of access to inputs and outputs

SIMOTION provides several possibilities to access the device inputs and outputs of the SIMOTION device as well as the central and distributed I/O:

- Via direct access with I/O variables

  Direct access is used to directly access the corresponding I/O address.

  Define an I/O variable (name and I/O address) without assigning a task to it. The entire address space of the SIMOTION device can be used.

  It is preferable to use direct access with sequential programming (in MotionTasks); access to current input and output values at a particular point in time is especially important in this case.

  Further information: Direct access and process image of the cyclic tasks (Page 214).

- Via the process image of cyclic tasks using I/O variables

  The process image of the cyclic tasks is a memory area in the RAM of the SIMOTION device, on which the whole I/O address space of the SIMOTION device is mirrored. The mirror image of each I/O address is assigned to a cyclic task and is updated using this task. The task remains consistent throughout the whole cycle. This process image is used preferentially when programming the assigned task (cyclic programming).

  Define an I/O variable (name and I/O address) and assign a task to it. The entire address area of the SIMOTION device can be used.

  Direct access to this I/O variable is still possible: Specify direct access with *_direct.var-name*.

  Further information: Direct access and process image of the cyclic tasks (Page 214).

- Using the fixed process image of the BackgroundTask

  The process image of the BackgroundTask is a memory area in the RAM of the SIMOTION device, on which a subset of the I/O address space of the SIMOTION device is mirrored. The mirror image is refreshed with the BackgroundTask and is consistent throughout the entire cycle. This process image is used preferentially when programming the BackgroundTask (cyclic programming).

  The address space 0 .. 63 can be used. I/O addresses that are accessed using the process image of the cyclic task are excluded.

  Further information: Access to the fixed process image of the BackgroundTask (Page 220).

A comparison of the most important properties is contained in "Important properties of direct access and process image (Page 212)".

You can use I/O variables like any other variable, see "Access I/O variables (Page 226)".

Note

An access via the process image is more efficient than direct access.

5.3.2 Important features of direct access and process image access

Table 5-28 Important features of direct access and process image access

<table>
<tr><th></th><th>Direct access</th><th>Access to process image of cyclic tasks</th><th>Access to fixed process image of the BackgroundTask</th></tr>
<tr><td>Permissible address range</td><td colspan="2">Entire address range of the SIMOTION device
Exception: I/O variables comprising more than one byte must not contain addresses 63 and 64 contiguously (example: PIW63 or PQD62 are not permitted).
The addresses used must be present in the I/O and appropriately configured.</td><td>0 .. 63,
except for the addresses used in the process image of the cyclic tasks
Addresses that are not present in the I/O or have not been configured can also be used.</td></tr>
<tr><td>Assigned task</td><td>None.</td><td>Cyclic task for selection:
• SynchronousTasks,
• TimerInterruptTasks,
• BackgroundTask.</td><td>BackgroundTask.</td></tr>
<tr><td>Updating</td><td>• Onboard I/O of SIMOTION devices C230-2, C240, and P350:
Update occurs in a cycle clock of 125 µs.
• I/O via PROFIBUS DP, PROFINET, P-Bus, and DRIVE-CLiQ as well as Onboard I/O of the D4xx SIMOTION devices:
Update occurs in the position control cycle clock.
Inputs are read at the start of the cycle clock.
Outputs are written at the end of the cycle clock.</td><td>Update occurs with the assigned task:
• Inputs are read before the assigned task is started and transferred to the process input image.
• Process output image is written to the outputs after the assigned task has been completed.</td><td>An update is made with the *BackgroundTask*:
• Inputs are read before the *BackgroundTask* is started and is transferred to the process input image.
• Process output image is written to the outputs when the *BackgroundTask* is complete.</td></tr>
<tr><td rowspan="2">Consistency</td><td>–</td><td>During the entire cycle of the assigned task.
Exception: Direct access to output occurs.</td><td rowspan="2">During the entire cycle of the *BackgroundTask*.
Exception: Direct access to output occurs.</td></tr>
<tr><td colspan="2">Consistency is only ensured for elementary data types.
When using arrays, the user is responsible for ensuring data consistency.</td></tr>
<tr><td>Use</td><td>Preferred in MotionTasks</td><td>Preferred in the assigned task</td><td>Preferred in the BackgroundTask</td></tr>
<tr><td>Use the absolute address</td><td colspan="2">Not supported.</td><td>Possible, with the following syntax: e.g. %IW62, %Q63.3.</td></tr>
<tr><td>Declaration as variable</td><td colspan="2">Necessary, for the entire device as an I/O variable in the symbol browser.
Syntax of I/O address: e.g. PIW1022, PQ63.3.</td><td>Possible, but not necessary:
• for the entire device as an I/O variable in the symbol browser,
• As unit variable,
• As local variable in a program.</td></tr>
</table>

	Direct access	**Access to process image of cyclic tasks**	**Access to fixed process image of the BackgroundTask**
Write protection for outputs	Possible; **Read only** status can be selected.	Not supported.	Not supported.
Declaration of arrays	Possible.		Not supported.
Further information	Direct access and process image of the cyclic tasks (Page 214).		Access to the fixed process image of the BackgroundTask (Page 220).
Responses in the event of an error	Error during access from user program, alternative reactions available: • CPU stop[1] • Substitution value • Last value	Error during generation of process image, alternative reactions available: • CPU stop[2] • Substitution value • Last value	Error during generation of process image, reaction: CPU stop[2]. **Exception**: If a direct access has been created at the same address, the behavior set there applies.
	Please refer to the SIMOTION Basic Functions function description.		
Access			
• In RUN mode	Without any restrictions.	Without any restrictions.	Without any restrictions.
• During StartupTask	Possible with restrictions: • Inputs can be read. • Outputs are not written until StartupTask is complete.	Possible with restrictions: • Inputs are read at the start of the StartupTask. • Outputs are not written until StartupTask is complete.	Possible with restrictions: • Inputs are read at the start of the StartupTask. • Outputs are not written until StartupTask is complete.
• During ShutdownTask	Without any restrictions.	Possible with restrictions: • Inputs retain status of last update • Outputs are no longer written.	Possible with restrictions: • Inputs retain status of last update • Outputs are no longer written.
[1] Call the ExecutionFaultTask. [2] Call the PeripheralFaultTask.			

5.3.3 Direct access and process image of cyclic tasks

Properties

Direct access to inputs and outputs and access to the process image of the cyclic task always take place via I/O variables. The entire address range of the SIMOTION device (see table below) can be used.

A comparison of the most important properties, also in comparison to the fixed process image of the BackgroundTask (Page 220) is contained in "Important properties of direct access and process image (Page 212)".

Direct access

The direct access is used to directly access the corresponding I/O address. Direct access is used primarily for sequential programming (in MotionTasks). The access to the current value of the inputs and outputs at a specific time is particularly important.

For direct access, you define an I/O variable (Page 217) without assigning it a task.

Note

An access via the process image is more efficient than direct access.

Process image of the cyclic task

The process image of the cyclic tasks is a memory area in the RAM of the SIMOTION device, on which the whole I/O address space of the SIMOTION device is mirrored. The mirror image of each I/O address is assigned to a cyclic task and is updated using this task. The task remains consistent throughout the whole cycle. This process image is used preferentially when programming the assigned task (cyclic programming). The consistency during the complete cycle of the task is particularly important.

For the process image of the cyclical task you define an I/O variable (Page 217) and assign it a task.

Direct access to this I/O variable is still possible: Specify direct access with *_direct.var-name*.

Address range of the SIMOTION devices

The address range of the SIMOTION devices depending on the version of the SIMOTION kernel is contained in the following table. The complete address range can be used for direct access and process image of the cyclical tasks.

Table 5-29 Address range of the SIMOTION devices depending on the version of the SIMOTION kernel

SIMOTION device	Address range for SIMOTION Kernel version		
	V3.0	V3.1, V3.2	As of V4.0
C230-2	0 .. 1023	0 .. 2047 [3]	0 .. 2047 [3]
C240	–	–	0 .. 4096 [3]
D410 [1]	–	–	0 .. 16383 [3] [4]
D425[2]	–	0 .. 4095 [3]	0 .. 16383 [3] [4]
D435	0 .. 1023	0 .. 4095 [3]	0 .. 16383 [3] [4]
D445[2]	–	0 .. 4095 [3]	0 .. 16383 [3] [4]
P350	0 .. 1023	0 .. 2047 [3]	0 .. 4095 [3]

[1] Available as of V4.1.

[2] Available as of V3.2.

[34] For distributed I/O (over PROFIBUS DP), the transmission volume is restricted to 1024 bytes per PROFIBUS DP line.

[5] For distributed I/O (over PROFINET), the transmission volume is restricted to 4096 bytes per PROFINET segment.

Note

Observe the rules for I/O addresses for direct access and the process image of the cyclical tasks (Page 216).

5.3.3.1 Rules for I/O addresses for direct access and the process image of the cyclical tasks

NOTICE

You must observe the following rules for the I/O variable addresses for direct access and the process image of the cyclic task (Page 214). Compliance with the rules is checked during the consistency check of the SIMOTION project (e.g. during the download).

1. Addresses used for I/O variables must be present in the I/O and configured appropriately in the HW Config.
2. I/O variables comprising more than one byte must not contain addresses 63 and 64 contiguously.

   The following I/O addresses are not permitted:

   – Inputs: PIW63, PID61, PID62, PID63
   – Outputs: PQW63, PQD61, PQD62, PQD63
3. All addresses of an I/O variable comprising more than one byte must be within an address area configured in HW-config.
4. An I/O address (input or output) can only be used by a single I/O variable of data type BYTE, WORD or DWORD or an array of these data types. Access to individual bits with I/O variables of data type BOOL is possible.
5. If several processes (e.g. I/O variable, technology object, PROFIdrive telegram) access an I/O address, the following applies:

   – Only a single process can have write access to an I/O address of an output (BYTE, WORD or DWORD data type).

     Read access to an output with an I/O variable that is used by another process for write access, is possible.

   – All processes must use the same data type (BYTE, WORD, DWORD or ARRAY of these data types) to access this I/O address. Access to individual bits is possible irrespective of this.

     Take care, for example, if you want to use an I/O variable to read the transferred PROFIBUS telegram of a drive: The length of the I/O variables must match the length of the telegram.

   – Write access to different bits of an address is possible from several processes; however, write access with the data types BYTE, WORD or DWORD is then not possible.

Note

These rules do not apply to accesses to the fixed process image of the BackgroundTask (Page 220). These accesses are not taken into account during the consistency check of the project (e.g. during download).

5.3.3.2 Creating I/O variables for direct access or process image of cyclic tasks

You create I/O variables in the symbol browser of the detail view; to do this, you must be working in offline mode.

Here is a brief overview of the procedure:

1. In the project navigator of SIMOTION SCOUT, select the I/O element in the subtree of the SIMOTION device.
2. In the detail view, select the "Symbol browser" tab and scroll down to the end of the variable table (empty row).
3. In the last (empty) row of the table, enter or select the following:
   - Name of variable.
   - I/O address according to the "syntax for entering I/O addresses (Page 219)".
   - Optional for outputs:

     Activate the "Read only" checkbox if you only want to have read access to the output.

     You can then read an output that is already being written by another process (e.g. output of an output cam, PROFIdrive telegram).

     A read-only output variable cannot be assigned to the process image of a cyclic task.
   - Data type of the variables in accordance with "Possible data types of the I/O variables (Page 220)".
4. Optionally, you can also enter or select the following (not for data type BOOL):
   - Array length (array size).
   - Process image or direct access:

     Can only be assigned if the "Read only" checkbox is cleared.

     For process image, select the cyclic task to which you want to assign the I/O variable. To select a task, it must have been activated in the execution system.

     For direct access, select the blank entry.
   - Strategy for the behavior in an error situation (see *SIMOTION Basic Functions* Function Manual).
   - Substitute value (if array, for each element).
   - Display format (if array, for each element), when you monitor the variable in the symbol browser.

You can now access this variable using the symbol browser or any program of the SIMOTION device.

NOTICE

Note the following for the process image for cyclic tasks:

- A variable can only be assigned to one task.
- Each byte of an input or output can only be assigned to one I/O variable.

In the case of data type BOOL, please note:

- The process image for cyclic tasks and a strategy for errors cannot be defined. The behavior defined via an I/O variable for the entire byte is applicable (default: direct access or CPU stop).
- The individual bits of an I/O variable can also be accessed using the bit access functions.

Take care when making changes within the I/O variables (e.g. inserting and deleting I/O variables, changing names and addresses):

- In some cases the internal addressing of other I/O variables may change, making all I/O variables inconsistent.
- If this happens, all program sources that contain accesses to I/O variables must be recompiled.

Note

I/O variables can only be created in offline mode. You create the I/O variables in SIMOTION SCOUT and then use them in your program sources (e.g. ST sources, MCC sources, LAD/FBD sources).

Outputs can be read and written to, but inputs can only be read.

Before you can monitor and modify new or updated I/O variables, you must download the project to the target system.

You can use I/O variables like any other variable, see "Access I/O variables (Page 226)".

5.3.3.3 Syntax for entering I/O addresses

For the input of the I/O address for the definition of an I/O variable for direct access or process image of cyclical tasks (Page 214), use the following syntax. This specifies not only the address, but also the data type of the access and the mode of access (input/output).

Table 5-30 Syntax for the input of the I/O addresses for direct access or process image of the cyclic tasks

Data type	Syntax for		Permissible address range					
	Input	Output	Direct access		Process image		e.g. direct access D435 V4.1	
BOOL	PIn.x	PQn.x	n: x:	0 .. *MaxAddr* 0 .. 7		-[1]	n: x:	0 .. 16383 0 .. 7
BYTE	PIBn	PQBn	n:	0 .. *MaxAddr*	n:	0 .. *MaxAddr*	n:	0 .. 16383
WORD	PIWn	PQWn	n:	0 .. 62 64 .. *MaxAddr* - 1	n:	0 .. 62 64 .. *MaxAddr* - 1	n:	0 .. 62 64 .. 16382
DWORD	PIDn	PQDn	n:	0 .. 60 64 .. *MaxAddr* - 3	n:	0 .. 60 64 .. *MaxAddr* - 3	n:	0 .. 60 64 .. 16380
n = logical address x = bit number								
MaxAddr =	Maximum I/O address of the SIMOTION device depending on the version of the SIMOTION kernel, see address range of the SIMOTION devices in "direct access and process image of the cyclical tasks (Page 214)".							
[1] For data type BOOL, it is not possible to define the process image for cyclic tasks. The behavior defined via an I/O variable for the entire byte is applicable (default: direct access).								

Examples:

Input at logic address 1022, WORD data type: PIW1022.

Output at logical address 63, bit 3, BOOL data type: PQ63.3.

Note

Observe the rules for I/O addresses for direct access and the process image of the cyclical tasks (Page 216).

5.3.3.4 Possible data types of I/O variables

The following data types can be assigned to the I/O variables for direct access and process image of the cyclical tasks (Page 214). The width of the data type must correspond to the data type width of the I/O address.

If you assign a numeric data type to the I/O variables, you can access these variables as integer.

Table 5-31 Possible data types of the I/O variables for direct access and the process image of the cyclical tasks

Data type of I/O address	Possible data types for I/O variables
BOOL (PIn.x, PQn.x)	BOOL
BYTE (PIBn, PQBn)	BYTE, SINT, USINT
WORD (PIWn, PQWn)	WORD, INT, UINT
DWORD (PIDn, PQDn)	DWORD, DINT, UDINT

For details of the data type of the I/O address, see also "Syntax for entering I/O addresses (Page 219)".

5.3.4 Access to fixed process image of the BackgroundTask

The process image of the BackgroundTask is a memory area in the RAM of the SIMOTION device, on which a subset of the I/O address space of the SIMOTION device is mirrored. It is preferably used for programming the BackgroundTask (cyclic programming) as it is consistent throughout the entire cycle.

The size of the fixed process image of the BackgroundTask for all SIMOTION devices is 64 bytes (address range 0 ... 63).

A comparison of the most important properties in comparison to the direct access and process image of the cyclical tasks (Page 214) is contained in "Important properties of direct access and process image (Page 212)".

NOTICE

I/O addresses that are accessed with the process image of the cyclic tasks must not be used. These addresses cannot be read or written to with the fixed process image of the BackgroundTask.

Note

The rules for I/O addresses for direct access and the process image of the cyclical tasks (Page 216) do **not** apply. The accesses to the fixed process image of the BackgroundTask are not taken into account during the consistency check of the project (e.g. during download).

Addresses not present in the I/O or not configured in HW Config are treated like normal memory addresses.

You can access the fixed process image of the BackgroundTask by means of:

- Using an absolute PI access (Page 221): The absolute PI access identifier contains the address of the input/output and the data type.
- Using a symbolic PI access (Page 223): You declare a variable that references the relevant absolute PI access.
  - A unit variable
  - A static local variable in a program.
- Using an I/O variable (Page 225): In the symbol browser, you define a valid I/O variable for the entire device that references the corresponding absolute PI access.

NOTICE

Please observe that if the inputs and outputs work with the Little Endian byte order (e.g. the integrated digital inputs of the SIMOTION devices C230-2 or C240) and the following conditions are fulfilled:

1. The inputs and outputs are configured to an address 0 .. 62.
2. An I/O variable for direct access (data type WORD, INT or UINT) has been created for these inputs and outputs.
3. You also access these inputs and outputs via the fixed process image of the BackgroundTask.

then the following is valid:

- Access with the data type WORD supplies the same result via the I/O variable and the fixed process image of the BackgroundTask.
- The access to the individual bytes with the *_getInOutByte* function (see *SIMOTION Basic Functions* Function Manual) supplies these in the Little Endian order.
- Access to the individual bytes or bits with the fixed process image of the BackgroundTask supplies these in the Big Endian order.

For information on the order of the bytes Little Endian and Big Endian: Please refer to the *SIMOTION Basic Functions* Function Manual.

5.3.4.1 Absolute access to the fixed process image of the BackgroundTask (absolute PI access)

You make absolute access to the fixed process image of the BackgroundTask (Page 220) by directly using the identifier for the address (with implicit data type). The syntax of the identifier (Page 222) is described in the following section.

You can use the identifier for the absolute PI access in the same manner as a normal variable (Page 222).

Note

Outputs can be read and written to, but inputs can only be read.

5.3.4.2 Syntax for the identifier for an absolute process image access

For the absolute access to the fixed process image of the BackgroundTask (Page 221), use the following syntax. This specifies not only the address, but also the data type of the access and the mode of access (input/output).

You also use these identifiers:

- For the declaration of a symbolic access to the fixed process image of the BackgroundTask (Page 223).
- For the creation of an I/O variables for accessing the fixed process image of the BackgroundTask (Page 225).

Table 5-32 Syntax for the identifier for an absolute process image access

Data type	Syntax for		Permissible address range	
	Input	Output		
BOOL	%In.x or %IXn.x[1]	%Qn.x or %QXn.x[1]	n: x :	0 .. 63 [2] 0 .. 7
BYTE	%IBn	%QBn	n:	0 .. 63 [2]
WORD	%IWn	%QWn	n:	0 .. 63 [2]
DWORD	%IDn	%QDn	n:	0 .. 63 [2]

n = logical address
x = bit number				

[1] The syntax %IXn.x or %QXn.x is not permitted when defining I/O variables. [2] Except for the addresses used in the process image of the cyclic tasks.				

Examples

Input at logic address 62, WORD data type: %IW62.

Output at logical address 63, bit 3, BOOL data type: %Q63.3.

NOTICE

Addresses that are accessed with the process image of the cyclic tasks must not be used. These addresses cannot be read or written to with the fixed process image of the BackgroundTask.

Note

The rules for I/O addresses for direct access and the process image of the cyclical tasks (Page 216) do not apply. The accesses to the fixed process image of the BackgroundTask are not taken into account during the consistency check of the project (e.g. during download).

Addresses not present in the I/O or not configured in HW Config are treated like normal memory addresses.

Several examples for the assignment of variables of the same type follow:

Table 5-33 Examples of absolute CPU memory access

```
status1 := %I1.1; // BOOL data type
status2 := %IB10; // BYTE data type
status3 := %IW20; // WORD data type
status4 := %ID20; // DWORD data type

%Q1.1 := status1; // BOOL data type
%QB20 := status2; // BYTE data type
%QW20 := status3; // WORD data type
%QD20 := status4; // DWORD data type
```

5.3.4.3 Symbolic access to the fixed process image of the BackgroundTask (symbolic PI access)

You can access the fixed process image of the BackgroundTask (Page 220) symbolically without needing to always specify the absolute process image access.

You can declare symbolic access:

- As a static variable of a program (within the VAR/END_VAR structure in the declaration section)
- As a unit variable (within the VAR_GLOBAL / END_VAR structure in the interface or implementation section of the ST source file)

The syntax for declaring a symbolic name for the PI access is shown in the figure:

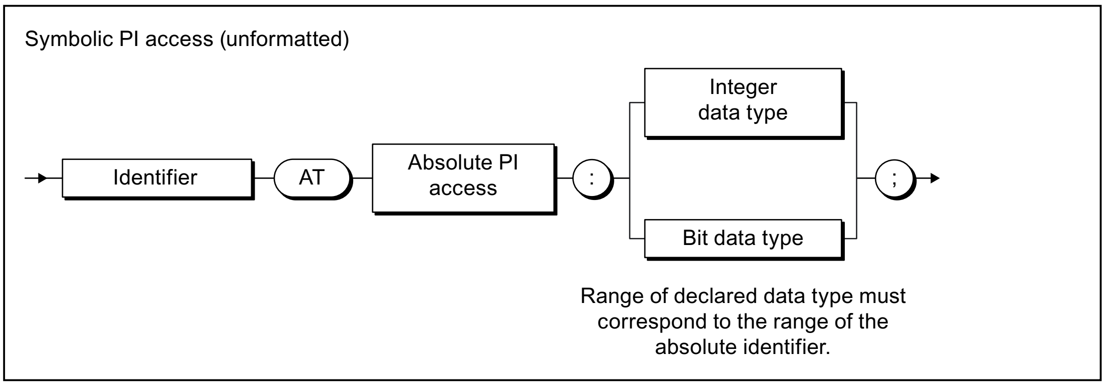

Symbolic PI access (unformatted)
Identifier
AT
Absolute PI access
:
Integer data type
Bit data type
;
Range of declared data type must correspond to the range of the absolute identifier.

Figure 5-6 Declaration of a symbolic access to the process image

For the absolute PI access, see "Syntax for the identifier for an absolute PI access (Page 222)".

The range of the declared integer or bit data type must correspond to the range of the absolute PI access, see "Possible data types of the symbolic PI access (Page 224)". After declaring a numerical data type, you can address the contents of the process image as an integer.

See also Example for the declaration (Page 224).

5.3.4.4 Possible data types for symbolic PI access

In the following cases, a data type that differs from that of the absolute PI access can be assigned to the fixed process image of the BackgroundTask (Page 220). The data type width must correspond to the data type width of the absolute PI access.

- For the declaration of a symbolic PI access (Page 223).
- For the creation of an I/O variable (Page 225).

If you assign a numeric data type to the symbolic PI access or to the I/O variables, you can access these variables as integer.

Table 5-34 Possible data types for symbolic PI access

Data type of the absolute PI access	Possible data types of the symbolic PI access
BOOL (%In.x, %IXn.x, %Qn.x. %QXn.x)	BOOL
BYTE (%IBn, %QBn)	BYTE, SINT, USINT
WORD (%IWn, %QWn)	WORD, INT, UINT
DWORD (%IDn, %PQDn)	DWORD, DINT, UDINT

For the data type of the absolute PI access, see also "Syntax for the identifier for an absolute PI access (Page 222)".

5.3.4.5 Example of symbolic PI access

If, for example, you want to access the CPU memory area (absolute PI access (Page 222)) %IB10, but can respond flexibly to changes in your program, then declare a *myInput* variable with this CPU memory area as follows:

```
VAR
    myInput AT %IB10 : BYTE;
END_VAR
```

If you want to use the integer value of the memory area, declare the *myInput* variable as follows:

```
VAR
    myInput AT %IB10 : SINT;
END_VAR
```

If you want to use a CPU memory area other than %IB10 in your program at a later time, you only need to change the absolute PI access in the variable declaration.

5.3.4.6 Creating an I/O variable for access to the fixed process image of the BackgroundTask

You create I/O variables in the symbol browser of the detail view; to do this, you must be working in offline mode.

Here is a brief overview of the procedure:

1. In the project navigator of SIMOTION SCOUT, select the I/O element in the subtree of the SIMOTION device.
2. In the detail view, select the Symbol browser tab and scroll down to the end of the variable table (empty row).
3. In the last (empty) row of the table, enter or select the following:
   - **Name** of variable.
   - Under **I/O address**, the absolute PI access according to the "syntax for the identifier for an absolute PI access (Page 222)"
     (exception: The syntax %IXn.x or %QXn.x is not permitted for data type BOOL).
   - **Data type** of the I/O variables according to the "possible data types of the symbolic PI access (Page 224)".
4. Select optionally the display format used to monitor the variable in the symbol browser.

You can now access this variable using the symbol browser or any program of the SIMOTION device.

Note

I/O variables can only be created in offline mode. You create the I/O variables in SIMOTION SCOUT and use them in your program sources.

Note that you can read and write outputs but you can only read inputs.

Before you can monitor and modify new or updated I/O variables, you must download the project to the target system.

You can use I/O variables like any other variable, see "Access I/O variables (Page 226)".

5.3.5 Accessing I/O variables

You have created an I/O variable for:

- Direct access or process image of the cyclic tasks (Page 214).
- Access to the fixed process image of the BackgroundTask (Page 220).

You can use this I/O variable just like any other variable.

NOTICE
Consistency is only ensured for elementary data types. When using arrays, the user is responsible for ensuring data consistency.

Note

If you have declared unit variables or local variables of the same name (e.g. *var-name*), specify the I/O variable using *_device.var-name* (predefined namespace, see the "Predefined namespaces" table in "Namespaces").

It is possible to directly access an I/O variable that you created as a process image of a cyclic task. Specify direct access with *_direct.var-name* or *_device._direct.var-name*.

If you want to deviate from the default behavior when errors occur during variable access, you can use the *_getSafeValue* and *_setSafeValue* functions (see *SIMOTION Basic Functions* Function Manual).

For errors associated with access to I/O variables, see *SIMOTION Basic Functions* Function Manual.

5.4 Using libraries

Libraries provide you with user-defined data types , functions and function blocks that can be used from all SIMOTION devices.

Libraries can be written in all programming languages; they can be used in all program sources (e.g. ST source files, MCC units).

You can obtain more details on inserting and managing libraries in the online help.

NOTICE

The same rules as for the names of program source files apply to the library names, see Insert ST source file (Page 21). In particular, the permissible length of the name depends on the SIMOTION Kernel version:

- As of Version V4.1 of the SIMOTION Kernel: maximum 128 characters.
- Up to Version V4.0 of the SIMOTION Kernel: maximum 8 characters.

With versions of the SIMOTION Kernel up to V4.0, a violation of the permissible length of the library name may not be detected until a consistency check or a download of the project is performed!

5.4.1 Compiling a library

In libraries, you can use all ST commands except for the ones listed in the table. In addition, you are not allowed to access some variables; these variables are also listed in this table .

Table 5-35 Illegal ST commands and variable access in libraries

Prohibited commands:
• *_getTaskId* function (see *SIMOTION Basic Functions* Function Manual).
• *_getAlarmId* function (see *SIMOTION Basic Functions* Function Manual).
• *_checkEqualTask* function (see *SIMOTION Basic Functions* Function Manual).
• Following functions that are intended for SIMOTION kernel versions up to V3.0: – Task control commands – Commands for runtime measurement of tasks – Commands for message programming With these functions, the name of the task of the configured message is transferred.
• If the library is not device-dependent (i.e. compiled without reference to a SIMOTION device or SIMOTION Kernel version): – System functions of SIMOTION devices (see the Parameter Manual for SIMOTION devices) – Version-dependent system functions

Prohibited variable accesses:

- Unit variables (retentive and non-retentive)
- Global device variables (retentive and non-retentive)
- I/O variables
- Instances of the technology objects and their system variables
- Variables of task names and configured messages (*_task* and *_alarm* namespaces, see Namespaces (Page 233), Predefined namespaces (Page 233) table)
- If the library is not device-dependent (i.e. compiled without reference to a SIMOTION device or SIMOTION Kernel version):
  - System variables of SIMOTION devices (see the Parameter Manual for SIMOTION devices)
  - Configuration data of technology objects (see Parameter Manual of configuration data for the relevant SIMOTION technology package)

Note

The **Program status** debug function is not available in libraries.

Compiling an individual library

To compile an individual library, proceed as follows:

1. Select the library in the project navigator.
2. Select the **Edit > Object Properties** menu command.
3. Select the **TPs/TOs** tab.
4. Select the SIMOTION devices (with SIMOTION kernel version) and the technology packet that you want to use as a basis for compiling the library; see the *SIMOTION Basic Functions* Function Manual.
5. Select **Accept and compile** from the context menu.

The library is compiled with reference to **all** selected SIMOTION devices, SIMOTION kernel versions and technology packages (and independently of devices).

NOTICE

If the library to be compiled imports another library, note the following:

1. For the imported library, at least the same devices and SIMOTION kernel versions must be selected as for the importing library.

   Alternatively, the imported library can be compiled independently of devices if the prerequisites for this are fulfilled (refer to the *SIMOTION Basic Functions* Function Manual).
2. The imported library must already be compiled individually with reference to all configured devices, kernel versions and technology packages.

   Compilation of the library as part of a project-wide compilation is generally not sufficient.

Compiling a library as part of a project-wide compilation

When you compile the whole SIMOTION project (e.g. by choosing **Project > Save and recompile all** or by performing an XML import), the libraries used are also compiled.

NOTICE
When performing project-wide compilation, note the following: 1. The system automatically identifies dependencies between libraries and selects the appropriate compilation sequence. 2. A library is **only** compiled with reference to the SIMOTION devices (including versions of the SIMOTION kernel) that are configured in the project and which use the library. 3. Other SIMOTION devices and kernel versions set for the library are ignored.

5.4.2 Know-how protection for libraries

You can protect libraries and their source files against unauthorized access by third parties. Protected libraries or sources can only be opened or exported as plain text files by entering a password.

You can:

- Provide individual sources of a library with know-how protection:

  Only the sources are protected against unauthorized access.

  The setting of the SIMOTION devices including the versions of the SIMOTION Kernel and the technology packages, for which the library is to be compiled, can still be changed and adapted by the user. Please refer to the *SIMOTION Basic Functions* Function Manual.

  The user can thus use the library for other SIMOTION devices and kernel versions.

- Provide the library with know-how protection:

  The following is then protected against unauthorized access:

  - All sources of the library
  - The setting of the SIMOTION device including the versions of the SIMOTION Kernel and the technology packages for which the library is to be compiled.

  You thus prevent that the user can use the library for other SIMOTION devices and kernel versions.

  Only use this setting if this is intended.

For information about how to apply know-how protection, refer to the online help.

Note

If you export in XML format, the libraries or sources are exported in an encrypted form. When importing the encoded XML files, the know-how protection, including login and password, remains in place.

5.4.3 Using data types, functions and function blocks from libraries

Before using data types, functions or function blocks from libraries, you must make them known to the ST source file. To do so, use the following construct in the interface section of the ST source file:

```
USELIB library-name [AS namespace];
```

In this case, *library-name* is the name of the library as it appears in the project navigator.

When multiple libraries are to be specified, enter them as a list separated by commas, e.g.:

```
USELIB library-name_1 [AS namespace_1],
    library-name_2 [AS namespace_2],
    library-name_3 [AS namespace_1], ...
```

You can use the optional *AS namespace* add-on to define a namespace (see Namespaces (Page 233)).

- You can then access data types, functions, and function blocks in the library that have the same name as such an ST source file of a SIMOTION device (in the PROGRAMS folder).
- You can also use namespaces to change the names of data types, functions and function blocks in the library so that they have different names.

You can also assign the same namespace to different libraries.

Table 5-36 Example of use of namespaces with libraries

```
INTERFACE
    USELIB Bib_1 AS NS_1, Bib_2 AS NS_2;
    PROGRAM Main_Program;
END_INTERFACE

IMPLEMENTATION
    FUNCTION Function1 : VOID
        VAR
            ComID : CommandIdType;
        END_VAR
        ComId := _getCommandId();
    END_FUNCTION

    PROGRAM Main_program
        function1();         // Function from this source
        NS_1.Var1:=1;
        NS_2.Var1:=2;
        NS_1.function1();    // Function from the Bib1 library
        NS_2.function1();    // Function from the Bib2 library
    END_PROGRAM
END_IMPLEMENTATION
```

5.5 Use of the same identifiers and namespaces

5.5.1 Use of the same identifiers

It is possible to use unit variables and local variables (program variables, FB variables, FC variables) with the same name. When compiling a program source, the compiler searches for identifiers beginning with the current POU. The smaller validity range always takes priority over the larger validity range.

You can therefore use the same identifiers in different source file sections, as long as the rules below are adhered to. If a higher-level identifier is hidden by an identifier in a unit or POE, the compiler issues a warning.

NOTICE
Under certain circumstances, the compiler may not issue a warning if, for example, the associated technology package is not imported.

Identifiers in a program organization unit (POU)

All following identifiers in a POU must be unique:

- Local variables of the POU.
- Local data types of the POU.

They may not also be identical with the following identifiers:

- Reserved identifiers.
- Identifiers of the POU itself.

The compiler issues a warning when the following identifiers are hidden:

- Unit variables, data types and POU or the same or imported units
- Standard system functions, standard system function blocks and associated data types
- System functions and system data types of the SIMOTION device
- Program organization units (POU) and data types from imported libraries
  - This can be resolved by entering a user-defined namespace.
- System functions and system data types from imported technology packages.
  - This can be resolved by entering a user-defined namespace.
- SIMOTION device variables (system variables, I/O variables, global device variables)
  - This can be resolved by entering the predefined namespace *_device*.
- Technology objects configured on the SIMOTION device
  - This can be resolved by entering the predefined namespace *_to*.

Identifiers in a unit

All following identifiers in a unit must be unique:

- Unit variables (declared in the interface or implementation section)
- Data types (declared in the interface or implementation section)
- Program organization units (POUs)

These must not be identical to the following identifiers either:

- Reserved identifiers.
- Unit variables, data types and POU imported units.
- Standard system functions, standard system function blocks and associated data types.
- System functions and system data types of the SIMOTION device.
- Program organization units (POU) and data types from imported libraries.
  - This can be resolved by entering a user-defined namespace.
- System functions and system data types from imported technology packages.
  - This can be resolved by entering a user-defined namespace.

The compiler issues a warning when the following identifiers are hidden:

- SIMOTION device variables (system variables, I/O variables, global device variables).
  - This can be resolved by entering the predefined namespace *_device*.
- Technology objects configured on the SIMOTION device.
  - This can be resolved by entering the predefined namespace *_to*.

Identifiers on the SIMOTION device (e.g., I/O variables, global device variables)

All the following identifiers on the SIMOTION device must be unique:

- I/O variables
- Global device variables
- System variables of the SIMOTION device
- System functions and system data types of the SIMOTION device.

These must not also be identical to the following identifiers:

- Reserved identifiers.
- Standard system functions, standard system function blocks and associated data types.

Example

The following example illustrates this situation. It shows that for use of identical names for unit variables (large validity range) and FC variables (small variable scope), only the variables declared in the function are valid within this source file section. The unit variables are only valid in POUs in which no local variables of the same name were declared. See the example.

Table 5-37 Example of identifier validity

```
TYPE
    type_a : (enum1, enum2, enum3);
END_TYPE

VAR_GLOBAL
    var_a, var_b : DINT;     // Unit variables
END_VAR

FUNCTION fc_1 : VOID
    VAR
        var_a : type_a;      // Declaration hides UNIT variable
        var_c : DINT;        // Local variable
    END_VAR
    // Permitted statements
    var_a := enum2;          // Access to local variable
    var_b := 100;            // Access to unit variable
    var_c := -1;             // Access to local variable
    // Invalid statement
    // var_a := 200;
END_FUNCTION

FUNCTION fc_2 : VOID
    VAR
        var_b : type_a;     // Declaration hides UNIT variable
        var_c : type_a;     // Local variable
    END_VAR
    // Permitted statements
    var_a := -100;          // Access to unit variable
    var_b := enum3;         // Access to local variable
    var_c := enum1;         // Access to local variable
    // Invalid statement
    // var_b := 200;
END_FUNCTION
```

5.5.2 Namespaces

You can also access data types, unit variables, functions, and function blocks defined outside of a program source (e.g. in libraries, technology packages, and on the SIMOTION device) using their names.

When compiling a program source, the compiler searches for identifiers beginning with the current POU. The data types, variables, functions, or function blocks declared in a program source therefore hide identifiers with the same name which have been defined outside the source, see Use of the same identifiers (Page 231). In order to still access these hidden identifiers, you can use namespaces in certain cases.

User-defined namespace

In the import instruction for libraries and technology packages, you can define namespaces in order to reach the data types, functions, or function blocks of these libraries and technology packages.

```
USELIB library-name_1 [AS lib_namespace_1],
    library-name_2 [AS lib_namespace_2],
    library-name_3 [AS lib_namespace_1], ...

USEPACKAGE tp-name_1 [AS tp_namespace_1],
    tp-name_2 [AS tp_namespace_2],
    tp-name_3 [AS tp_namespace_1], ...
```

You can also use namespaces to make names consistent within different libraries.

If you wish to use a data type, a function or a function block from a library or a technology package, place the namespace identifier in front of the name, separated by a period, for example, *namespace.fc-name*, *namespace.fb-name*, *namespace.type-name.*

Example

The following example shows how to select the Cam technology package, assign it the namespace Cam1 and use the namespace:

Table 5-38 Example of selecting a technology package and using a namespace

```
INTERFACE
    USEPACKAGE Cam AS Cam1;
    USES ST_2;
    FUNCTION function1;
END_INTERFACE

IMPLEMENTATION
    FUNCTION function1 : VOID
    VAR_INPUT
        p_Axis : posAxis;
    END_VAR
    VAR
        retVal : DINT;
    END_VAR

    retVal:= Cam1._enableAxis (
            axis := p_Axis,
            nextCommand := Cam1.WHEN_COMMAND_DONE,
            commandId := _getCommandId() );
    END_FUNCTION
END_IMPLEMENTATION
```

NOTICE
If a namespace is defined for an imported library or technology package, this must **always** be specified if a function, function block, or data type from this library or technology package is being used. See above example: Cam1._enableAxis, Cam1.WHEN_COMMAND_DONE.

Predefined namespace

Namespaces are predefined for device- and project-specific variables as well as TaskID and AlarmID variables. If necessary, write their designation before the variable names, separated by a period, for example, *_device.var-name* or *_task.task-name*

Table 5-39 Predefined namespaces

Name space	Description
_alarm	For AlarmId: The _alarm.*name* variable contains the AlarmId of the message with the *name* identifier (see *SIMOTION Basic Functions* Function Manual).
_device	For device-specific variables (global device variables, I/O variables, and system variables of the SIMOTION device).
_direct	For direct access to I/O variables – see Direct access and process image of the cyclical tasks (Page 214). Local namespace for _device. Nesting as in _device._direct.*name* is permitted.
_project	For names of SIMOTION devices in the project; only used with technology objects on other devices. With unique project-wide names of technology objects, used also for these names and their system variables.
_task	For TaskID: The _task.*name* variable contains the TaskId of the task with the *name* identifier (see *SIMOTION Basic Functions* Function Manual).
_to	For technology objects configured on the SIMOTION device, and their system variables and configuration data. Not for system functions and data types of the technology objects. In this case, if necessary, use the user-defined namespace for the imported technology package

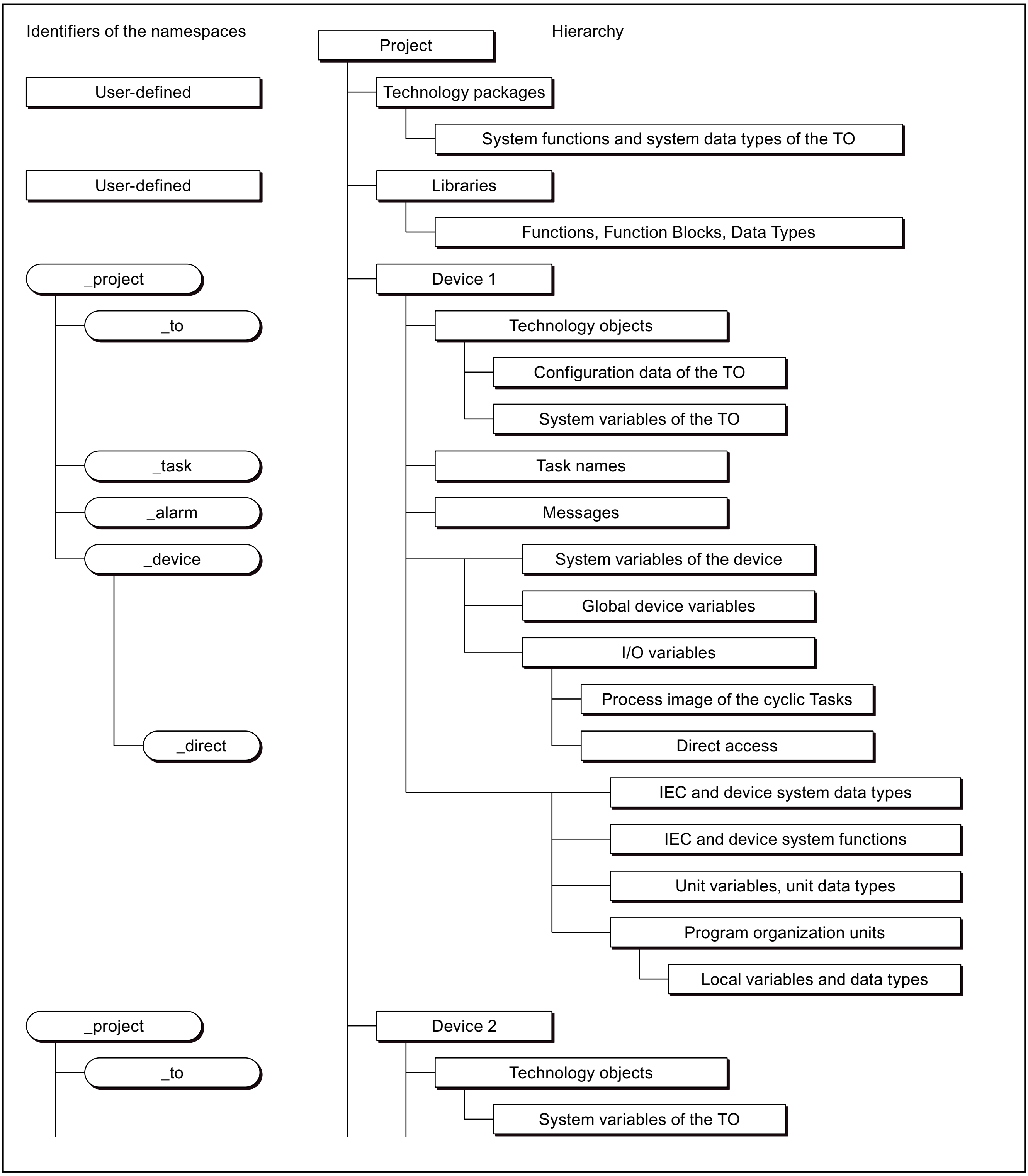

Identifiers of the namespaces
Hierarchy
Project
User-defined
Technology packages
System functions and system data types of the TO
User-defined
Libraries
Functions, Function Blocks, Data Types
_project
Device 1
_to
Technology objects
Configuration data of the TO
System variables of the TO
_task
Task names
_alarm
Messages
_device
System variables of the device
Global device variables
I/O variables
Process image of the cyclic Tasks
_direct
Direct access
IEC and device system data types
IEC and device system functions
Unit variables, unit data types
Program organization units
Local variables and data types
_project
Device 2
_to
Technology objects
System variables of the TO

Figure 5-7 Namespaces and identifier hierarchy

5.6 Reference data

The reference data provide you with an overview of:

- Utilized identifiers with information on their declaration and use (Cross reference list)
- Function calls and their nesting (Program structure)
- For the memory requirement of various data areas of the program sources (code attribute)

See also

Cross-reference list (Page 237)

Program structure (Page 239)

Code attributes (Page 241)

5.6.1 Cross-reference list

The cross-reference list shows all identifiers in program sources (e.g. ST source files, MCC source files):

- Declared as variables, data types, or program organization units (program, function, function block)
- Used as previously defined type identifiers in declarations
- Used as variables in the statement section of a program organization unit

You can generate the cross-reference list as required for:

- An individual program source (e.g. ST source file, MCC source file, LAD/FBD source)
- All program sources of a SIMOTION device
- All program sources and libraries of the project
- Libraries (all libraries, single library)

5.6.1.1 Creating a cross-reference list

To create the cross-reference list:

1. In the project navigator, select the element for which you want to create a cross-reference list.
2. Select the menu **Edit > Reference data > Create**.

The cross-reference list is displayed in its own tab in the detail view.

5.6.1.2 Content of the cross-reference list

The created cross-reference list shows the following for each identifier:

- The identifier name (for structures and enumerators, also the individual components and elements).
- The type (e.g. data type, POU type).
- The declaration location (e.g. name of the program source, name of the technology package).
- Information about the current use of the identifier:
  - Type of the use (e.g. R = read access, W = write access, variable type = declaration),
  - Path details of the program source (SIMOTION device, name of the program source),
  - Area in the program source (e.g. implementation section, POU name),
  - Program language of the program source,
  - Line number in the ST source file (or block number in the MCC chart or reference number in the LAD/FBD source).

Note

The generated cross-reference list is saved automatically and can be displayed selectively after selecting the appropriate element in the project navigator. To display the cross-reference list, select the **Edit > Reference data > Display > Cross-Reference List** menu command.

When a cross-reference list is recreated, it is updated selectively (corresponding to the selected element in the project navigator). Other existing cross-reference data are retained and displayed, if applicable.

Note

Activated single-step monitoring in MCC programming

Each task is assigned two variables TSI#dwuser_1 and TSI#dwuser_2, which can be written and read.

When single step monitoring is activated, the compiler uses these variables to control single step monitoring if at least one MCC chart is assigned to the relevant task. The user then cannot use these variables, because their contents are overwritten by single step monitoring and may cause undesirable side effects.

5.6.1.3 Working with a cross-reference list

In the cross-reference list you are able to:

- Sort the column contents alphabetically
- Set filter functions (via context menu, which you can call with the right mouse button)
- Copy contents to the clipboard in order, for example, to paste them into a spread-sheet program
- Print contents
- Open the referenced program source and position the cursor on the relevant line of the ST command (or MCC or LAD/FBD element):
  - Double-click on the corresponding line in the cross-reference list.

    or
  - Place the cursor in the corresponding line of the cross-reference list and click the **Go to application** button.

Further details about working with cross-reference lists can be found in the online help.

5.6.2 Program structure

In the program structure are all function calls and their nesting within a selected element.

When the cross-reference list has been successfully created, you can display the program structure selectively for:

- An individual program source (e.g. ST source file, MCC source file, LAD/FBD source)
- All program sources of a SIMOTION device
- All program sources and libraries of the project
- Libraries (all libraries, single library, individual program source within a library)

Follow these steps:

1. In the project navigator, select the element for which you want to display the program structure.
2. Select the menu **Edit > Reference data > Display > Program structure**.

   The cross-reference tab is replaced by the program structure tab in the detail view.

5.6.2.1 Content of the program structure

A tree structure appears, showing:

- as base respectively
  - the program organization units (programs, functions, function blocks) declared in the program source, or
  - the execution system tasks used
- below these, the subroutines referenced in this program organization unit or task.

For structure of the entries, see table:

Table 5-40 Elements of the display for the program structure

Element	Description
Base (declared POU or task used))	List separated by a comma • Identifier of the program organization unit (POU) or task • Identifier of the program source in which the POU or task was declared, with add-on [UNIT] • Minimum and maximum stack requirement (memory requirement of the POU or task on the local data stack), in bytes [Min, Max] • Minimum and maximum overall stack requirement (memory requirement of the POU or task on the local data stack including all called POUs), in bytes [Min, Max]
Referenced POU	List separated by a comma: • Identifier of called POU • Optionally: Identifier of the program source / technology package in which the POU was declared: Add-on (UNIT): User-defined program source Add-on (LIB): Library Add-on (TP): System function from technology package • Only for function blocks: Identifier of instance • Only for function blocks: Identifier of program source in which the instance was declared: Add-on (UNIT): User-defined program source Add-on (LIB): Library • Line of (compiled) source in which the POU is called; several lines are separated by "|".

5.6.3 Code attributes

You can find information on or the memory requirement of various data areas of the program sources under code attribute.

When the cross-reference list has been successfully created, you can display the code attributes selectively for:

- An individual program source (e.g. ST source file, MCC source file, LAD/FBD source)
- All program sources of a SIMOTION device
- All program sources and libraries of the project
- Libraries (all libraries, single library, individual program source within a library)

Follow these steps:

1. In the project navigator, select the element for which you want to display the code attributes.
2. Select the **Edit > Reference data > Display > Code attributes** menu.

   The **Cross-references** tab is now replaced by the **Code attributes** tab in the detail view.

5.6.3.1 Code attribute contents

The following are displayed in a table for all selected program source files:

- Identifier of program source file,
- Memory requirement, in bytes, for the following data areas of the program source file:
  - **Dynamic data**: All unit variables (retentive and non-retentive, in the interface and implementation sections),
  - **Retain data**: Retentive unit variables in the interface and implementation section,
  - **Interface data**: Unit variables (retentive and non-retentive) in the interface section,
- Number of referenced sources.

5.7 Controlling the preprocessor and compiler with pragmas

A pragma is used to insert an ST source file text (e.g. statements), which influences the compilation of the ST source file.

Pragmas are enclosed in { and } braces and can contain
(see figure):

- Preprocessor statements for controlling the preprocessor, see Controlling the preprocessor (Page 243).

  The pragmas with preprocessor statements contained in an ST source file are evaluated by the preprocessor and interpreted as control statements.
- Attributes for compiler options to control the compiler, see Controlling compiler with attributes (Page 247).

  The pragmas with attributes for compiler options contained in an ST source file are evaluated by the compiler and interpreted as control statements.

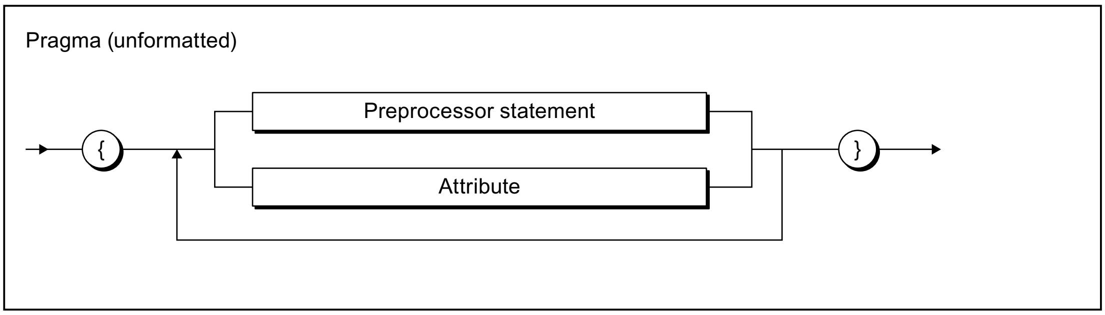


Figure 5-8 Pragma syntax

NOTICE
Be sure to use the correct pragma syntax (e.g. upper- and lower-case notation of attributes). Unrecognized pragmas are ignored with no warning message.

5.7.1 Controlling a preprocessor

The preprocessor prepares an ST source file for compilation. For example, character strings can be defined as replacement texts for identifiers, or sections of the source program can be hidden/shown for compilation.

The preprocessor is disabled by default. You can activate it as follows:

- Globally for all program source files and programming languages within the project, see "Global settings of the compiler (Page 45)".
- Local for a program source file, see "Local compiler settings (Page 46)".

During the compilation of a program source file, you will be informed about the preprocessor actions. This requires, however, that the display of warnings class 7 is activated, see Meaning of the warning classes (Page 49). You specify the details for issued warnings and information:

- In the global or local settings of the compiler.
- With the _U7_PoeBld_CompilerOption := warning:*n*:off or warning:*n*:on attribute within an ST source file, see "Controlling compiler with attributes (Page 247)".

Like compiler messages, information about the preprocessor is shown in the "Compile/check output" tab of the detail view.

Note

You can also view the text of the ST source file modified by the preprocessor:

1. Open the ST source file.
2. Select the **ST source file > Execute preprocessor** menu command.

The modified source text is shown in the "Compile/check output" tab of the detail view.

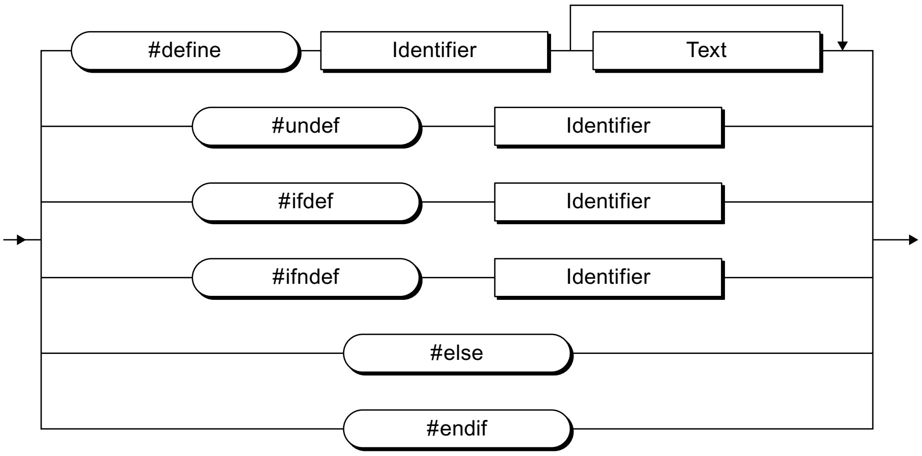
5.7.1.1 Preprocessor statement

You can control the preprocessor by means of statements in pragmas. The statements specified in the following syntax diagram can be used. These statements act on all subsequent lines of the ST source file.

They can be used in ST source files of a SIMOTION device or a library.

You can make definitions for the preprocessor in the property dialog box of the ST source file (see Making preprocessor definitions (Page 51)). This enables you also to control the preprocessor with ST source files with know-how protection (see Know-how protection for ST sources (Page 51)).

Preprocessor statement (unformatted)



Each statement must begin with a new line and end with a line break.

The following order must be maintained for the statements below:
#ifdef – #else (optional) – #endif or #ifndef – #else (optional) – #endif.

Text: String of any characters except:
\ (backslash), ’ (single quote) and ” (double quote).
The keywords USES, USELIB and USEPACKAGE are not permitted.

Figure 5-9 Syntax of a preprocessor statement

Table 5-41 Preprocessor statements

Statement	Meaning
#define	The specified identifier will be replaced below by the specified text. Permissible characters: See table footnote.
#undef	The replacement rule for the identifier is cancelled.
#ifdef	For variant formation (conditional compilation) If the specified identifier is defined, the following program lines (until the next pragma that contains #else or #endif) are compiled by the compiler.
#ifndef	For variant formation (conditional compilation) If the specified identifier is not defined, the following program lines (until the next pragma that contains #else or #endif) are compiled by the compiler.
#else	For variant formation (conditional compilation) Alternative branch to #ifdef or #ifndef. The following program lines (until the next pragma containing #endif) are compiled by the compiler, if the preceding query with #ifdef or #ifndef was not fulfilled.
#endif	Concludes variant formation with #ifdef or #ifndef.

Permissible characters:

- For identifiers: In accordance with the rules for identifiers (Page 73).
- For text: Sequence of any characters other than \ (backslash), ’ (single quote) and ” (double quote). The keywords USES, USELIB and USEPACKAGE are not permitted.

Note

Each preprocessor statement must begin with a new line and end with a line break. Consequently, the curly brackets ({ and }) enclosing the pragma must be placed in separate lines of the ST source file!

In the case of pragmas with #define statements, please note:

- Pragmas with #define statements in the interface section of an ST source file are exported. The defined identifiers can be imported with the USES statement into other ST source files of the same SIMOTION device or of the same library.
- Identifiers defined in pragmas of libraries cannot be imported into ST source files of a SIMOTION device.
- Redefinition of reserved identifiers is not possible.

You can also make preprocessor definitions in the property dialog box of the ST source file. In the case of different definitions of the same identifiers, #define statements within the ST source file have priority.

5.7.1.2 Example of preprocessor statements

Table 5-42 Example of preprocessor statements

```
               ST source file                               Preprocessor output
        With preprocessor statements
INTERFACE                                     INTERFACE
    FUNCTION_BLOCK fb1;                           FUNCTION_BLOCK fb1;
    VAR_GLOBAL                                    VAR_GLOBAL
        g_var     : INT;                              g_var     : INT;
    END_VAR                                       END_VAR
// Preprocessor definitions
{                                             {
#define my_define g_var
#define my_call f(my_define)
}                                             }
//     my_define -> g_var
//     my_call   -> f(g_var)
END_INTERFACE                                 END_INTERFACE

IMPLEMENTATION                                IMPLEMENTATION
    FUNCTION f : INT                              FUNCTION f : INT
        VAR_INPUT                                     VAR_INPUT
            i : INT;                                      i : INT;
        END_VAR                                       END_VAR
        f := i;                                       f := i;
    END_FUNCTION                                  END_FUNCTION

    FUNCTION_BLOCK fb1                            FUNCTION_BLOCK fb1
        VAR_INPUT                                     VAR_INPUT
            i_var : INT;                                  i_var : INT;
        END_VAR                                       END_VAR
        VAR_OUTPUT                                    VAR_OUTPUT
            o_var : INT;                                  o_var : INT;
        END_VAR                                       END_VAR
        my_define := i_var;                           g_var := i_var;
// Delete the preprocessor definition
// For my_define
{                                             {
#undef my_define
}                                             }
        o_var := my_call + 1;                         o_var := f(g_var) + 1;
{                                             {
#ifdef my_define
}
        my_define := i_var;
{
#endif
}                                             }
    END_FUNCTION_BLOCK                            END_FUNCTION_BLOCK
END_IMPLEMENTATION                            END_IMPLEMENTATION
```

5.7.2 Controlling compiler with attributes

You can use attributes within a pragma to control the compiler.

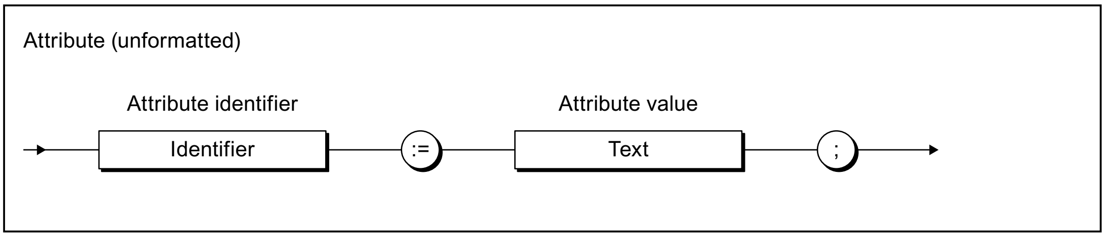


Figure 5-10 Syntax of an attribute for compiler options

Table 5-43 Permissible attributes for compiler options

Attribute identifier	Attribute value	Meaning
_U7_PoeBld_CompilerOption	The attribute affects the output of compiler warnings within an ST source file. It affects all subsequent lines of the ST source file.	
	warning:*n*:off	Warnings specified by the number *n* are not displayed
	warning:*n*:on	Warnings specified by the number *n* are displayed
	Permissible value for *n*: *n* = 0 to 7: Warning class, see also meaning of the warning classes (Page 49). *n* = 16000 and higher: Number of a warning.	
HMI_Export	The attribute changes the unit variables available on HMI devices by default. It must be placed directly after the associated keyword of the following declaration blocks: • VAR_GLOBAL • VAR_GLOBAL RETAIN It affects only the unit variables declared in the associated declaration block. Detailed description of the HMI export, in particular the effect of the attribute depending on the version of the SIMOTION kernel: see Variables and HMI devices (Page 208).	
	FALSE	In the interface section of an ST source file. The unit variables declared in the associated declaration block are not available on HMI devices.
	TRUE	In the implementation section of an ST source file. The unit variables declared in the associated declaration block are available on HMI devices.

<table>
<tr><th>Attribute identifier</th><th>Attribute value</th><th>Meaning</th></tr>
<tr><td rowspan="3">BlockInit_OnChange</td><td colspan="2">Only as of Version V3.2 of the SIMOTION kernel.
The attribute changes the standard definition whether a download in RUN mode is possible when a change is made to the version identification of the associated declaration block. It must be placed directly after the associated keyword of the following declaration blocks:
• VAR_GLOBAL (in the interface and implementation section)
• VAR_GLOBAL RETAIN (in the interface and implementation section)
• VAR (only for programs in a unit when the "Create program instance data only once" compiler option is active).
It affects only the variables declared in the associated declaration block.
See also Version ID of global variables and their initialization during download (Page 207).</td></tr>
<tr><td>FALSE</td><td>Download in RUN mode is <b>not</b> possible when the version identification of the declaration block is changed (default).</td></tr>
<tr><td>TRUE</td><td>Download in RUN mode is possible despite the change of the version identification of the declaration block. The variables of the declaration block are also initialized.</td></tr>
<tr><td rowspan="3">BlockInit_OnDeviceRun</td><td colspan="2">Only as of Version V4.1 of the SIMOTION kernel.
The attribute changes the standard definition whether the variables of the associated declaration block will be initialized for the transition to the RUN mode. It must be placed directly after the associated keyword of the following declaration blocks:
• VAR_GLOBAL (in the interface and implementation section)
• VAR (only for programs in a unit when the "Create program instance data only once" compiler option is active).
It affects only the variables declared in the associated declaration block.
See also Memory ranges of the variable types (Page 194).</td></tr>
<tr><td>DISABLE</td><td>The variables declared in the associated declaration block are <b>not</b> initialized in the transition of the mode from STOP to RUN (default).</td></tr>
<tr><td>ALWAYS</td><td>The variables declared in the associated declaration block are initialized in the transition of the mode from STOP to RUN.</td></tr>
</table>

NOTICE

Be sure to use the correct upper- and lower-case notation for attributes!

Note

The insert, delete or change of the HMI_Export, BlockInit_OnChange or BlockInit_OnDeviceRun attributes in a declaration block does not change its version identification!

Table 5-44 Example of attributes for compiler options

```
INTERFACE
    VAR_GLOBAL
        { HMI_Export := FALSE;
          BlockInit_OnChange := TRUE; }
        // No HMI export, download in RUN possible
        x : DINT;
    END_VAR
    FUNCTION_BLOCK fb1;
END_INTERFACE

IMPLEMENTATION
    VAR_GLOBAL
        { HMI_Export := TRUE;
          BlockInit_OnDeviceRun := ALWAYS; }
        // HMI export, initialization for the STOP -> RUN transition
        y : DINT;
    END_VAR
    FUNCTION_BLOCK fb1
        VAR_INPUT
            i_var    : INT;
        END_VAR
        VAR_OUTPUT
            o_var    : INT;
        END_VAR

        { _U7_PoeBld_CompilerOption := warning:2:on; }
        o_var := REAL_TO_INT(1.0);    // Warning 16004
        { _U7_PoeBld_CompilerOption := warning:2:off; }
        o_var := REAL_TO_INT(1.0);    // No warning 16004
        { _U7_PoeBld_CompilerOption := warning:16004:on; }
        o_var := REAL_TO_INT(1.0);    // Warning 16004
        { _U7_PoeBld_CompilerOption := warning:16004:off; }
        o_var := REAL_TO_INT(1.0);    // No warning 16004
        { _U7_PoeBld_CompilerOption := warning:2:off;
          _U7_PoeBld_CompilerOption := warning:16004:on; }
        o_var := REAL_TO_INT(1.0);    // Warning 16004
    END_FUNCTION_BLOCK
END_IMPLEMENTATION
```

5.8 Jump statement and label

In addition to control statements (see Control statements (Page 130)), a jump statement is also available.

You program jump statements with the GOTO statement and specify the jump label to which you want to jump. Jumps are only permitted within a POU.

Enter the jump label (separated by a colon) in front of the statement at which you want the program to resume.

Alternatively, you can declare the jump labels in the POU (with the structure LABEL/END_LABEL in the POU). Only the declared jump labels can then be used in the statement section.

Syntax of jump statements and labels:

Table 5-45 Example of syntax for jump statements

```
FUNCTION func : VOID
    VAR
        x, y, z BOOL;
    END_VAR
    LABEL
        lab_1, lab_2;     // Declaration of the jump labels
    END_LABEL
    x := y;
    lab_1 : y := z;        // Jump label with statement
    IF x = y THEN
        GOTO lab_2;        // Jump statement
    END_IF;
    GOTO lab_1;            // Jump statement
    lab_2 : ;              // Jump label with blank statement
END_FUNCTION
```

Note

You should only use the GOTO statement in special circumstances (for example, for troubleshooting). It should not be used at all according to the rules for structured programming.

Jumps are only permitted within a POU.

The following jumps are illegal:

- Jumps to subordinate control structures (WHILE, FOR, etc.)
- Jumps from a WAITFORCONDITION structure
- Jumps within CASE statements

Jump labels can only be declared in the POU in which they are used. If jump labels are declared, only the declared jump labels may be used.

Error Sources and Program Debugging 6

This chapter describes various sources of programming errors and shows you how to program efficiently. You also learn what options are available for program testing. All possible compiler error messages, namely, compiler errors, see Compiler Error Messages and Remedies (Page 350). Possible reactions and remedies are described for each error.

6.1 Notes on avoiding errors and on efficient programming

The SIMOTION *Basic Functions* Function Manual lists some common error sources, which hinder the compilers or prevent the proper execution of a program. There are notes on, e.g.:

- Data types for assigning arithmetic expressions
- Starting functions in cyclic tasks
- Wait times in cyclic tasks
- Errors on download
- CPU does not switch to RUN
- CPU goes to STOP
- Size of the local data stack
- etc.

In addition, you will also find notes on efficient programming there, particularly for

- runtime-oriented programming
- change-optimized programming

6.2 Program debugging

Syntax errors are detected and displayed by the ST compiler during the compilation procedure. Runtime errors in the execution of the program are displayed by system alarms or lead to the operating mode STOP. You can find logical programming errors with the test functions of ST, e.g. with the symbol browser, status program, trace.

To achieve the same results as shown below using the test functions, use of the sample program in Creating a sample program (Page 59) is recommended.

6.2.1 Modes for program testing

6.2.1.1 Modes of the SIMOTION devices

Various SIMOTION device modes are available for program testing.

How to select the mode of a SIMOTION device:

1. Highlight the SIMOTION device in the project navigator.
2. Select the "Test mode" context menu.
3. Select the required mode (see following table).

   If you have selected "Debug mode":

   – Accept the safety information.

   – Parameterize the sign-of-life monitoring.

   Observe the following section: Important information about the life-sign monitoring (Page 254).
4. Confirm with "OK".

   The SIMOTION device switches to the selected mode.

   When the SIMOTION device switches to "Debug mode":

   – A connection to the target system will be established automatically (online mode) if SIMOTION SCOUT is currently in offline mode.

   – The activated debug mode is indicated in the status bar.

   – The breakpoints toolbar is displayed.

Table 6-1 Modes of a SIMOTION device

Setting	Meaning
Process mode	Program execution on the SIMOTION device is optimized for maximum system performance. The following diagnostic functions are available, although they may have only restricted functionality because of the optimization for maximum system performance: • Monitor variables in the symbol browser or a watch table. • Program status (only restricted): – Restricted monitoring of variables (e.g. variables in loops, return values for system functions). – As of version V4.0 of the SIMOTION kernel: No more than one program source (e.g. ST source, MCC source, LAD/FBD source) can be monitored **per task**. – Up to version V3.2 of the SIMOTION kernel: No more than one program source (e.g. ST source, MCC source, LAD/FBD source) can be monitored. • Trace tool (only restricted) with measuring functions for drives and function generator, see online help: – No more than **one** trace on each SIMOTION device.
Test mode	The diagnostic functions of the process mode are available to the full extent: • Monitor variables in the symbol browser or a watch table. • Program status: – Monitoring of all variables possible. – As of version V4.0 of the SIMOTION kernel: Several program sources (e.g. ST sources, MCC sources, LAD/FBD sources) can be monitored per task. – Up to version V3.2 of the SIMOTION kernel: No more than one program source (e.g. ST source, MCC source, LAD/FBD source) can be monitored **per task**. • Trace tool with measuring functions for drives and function generator, see online help: – No more than **four** traces on each SIMOTION device. **Note** Runtime and memory utilization increase as the use of diagnostic functions increases.
Debug mode	This mode is available in SIMOTION kernel as of V3.2. In addition to the diagnostic functions of the test mode, you can use the following functions: • Breakpoints Within a program source file, you can set breakpoints (Page 271). When an activated breakpoint is reached, selected tasks will be stopped. • Controlling MotionTasks In the "Task Manager" tab of the device diagnostics, you can use task control commands for MotionTasks, see the SIMOTION Basic Functions Function Manual. No more than one SIMOTION device of the project can be switched to debug mode. SIMOTION SCOUT is in online mode, i.e. connected with the target system. Observe the following section: Important information about the life-sign monitoring (Page 254).

6.2.1.2 Important information about the life-sign monitoring

⚠ WARNING

You must observe the appropriate safety regulations.

Use the debug mode or a control panel only with the life-sign monitoring function activated with a suitably short monitoring time! Otherwise, if problems occur in the communication link between the PC and the SIMOTION device, the axis may start moving in an uncontrollable manner.

The function is released exclusively for commissioning, diagnostic and service purposes. The function should generally only be used by authorized technicians. The safety shutdowns of the higher-level control have no effect.

Therefore, there must be an EMERGENCY STOP circuit in the hardware. The appropriate measures must be taken by the user.

Accept safety notes

After selecting the debug mode or a control panel, you must accept the safety notes. You can set the parameters for the life-sign monitoring.

Proceed as follows:

1. Click the **Settings** button.

   The "Debug settings" window opens.

2. Read there, as described in the following section, the safety notes and parameterize the life-sign monitoring.

Parameterizing the life-sign monitoring

In the Life-sign monitoring parameterization window, proceed as described below:

1. Read the warning!
2. Click the **Safety notes** button to open the window with the detailed safety notes.
3. Do not make any changes to the defaults for life-sign monitoring.

   Changes should only be made in special circumstances and in observance of all danger warnings.
4. Click **Accept** to confirm you have read the safety notes and have correctly parameterized the life-sign monitoring.

NOTICE

Pressing the spacebar or switching to a different Windows application causes:

- In debug mode for activated breakpoints:
  - The SIMOTION device switches to STOP mode.
  - The outputs are deactivated (ODIS).
- For controlling an axis or a drive using the control panel (control priority for the PC):
  - The axis or the drive is brought to a standstill.
  - The enables are reset.

⚠ WARNING

This function is not guaranteed in all operating modes. Therefore, there must be an EMERGENCY STOP circuit in the hardware. The appropriate measures must be taken by the user.

6.2.1.3 Life-sign monitoring parameters

Table 6-2 Life-sign monitoring parameter description

Field	Description
Life-sign monitoring	The SIMOTION device and SIMOTION SCOUT regularly exchange life-sign signals to ensure a correctly functioning connection. If the exchange of the life-sign is interrupted longer than the set monitoring time, the following response occurs: • In debug mode for activated breakpoints: – The SIMOTION device switches to STOP mode. – The outputs are deactivated (ODIS). • For controlling an axis or a drive using the control panel (control priority for the PC): – The axis is brought to a standstill. – The enables are reset. The following parameterizations are possible: • **Active** check box: If the check box is selected, life-sign monitoring is active. The deactivation of the life-sign monitoring is not always possible. • Monitoring time: Enter the timeout. **Prudence** Do not make any changes to the defaults for life-sign monitoring, if possible. Changes should only be made in special circumstances and in observance of all danger warnings.
Safety information	**Please observe the warning!** Click the button to obtain further safety information. See: Important information about the life-sign monitoring (Page 254)

6.2.2 Symbol Browser

6.2.2.1 Properties of the symbol browser

In the symbol browser, you can view and, if necessary, change the name, data type, and variable values. In particular, you can: see the following variables:

- Unit variables and static variables of a program or function block
- System variables of a SIMOTION device or a technology object
- I/O variables or global device variables.

For these variables, you can:

- View a snapshot of the variable values
- Monitor variable values as they change
- Change (modify) variable values

However, the symbol browser can only display/modify the variable values if the project has been loaded in the target system and a connection to the target system has been established.

6.2.2.2 Using the symbol browser

Requirements

- Make sure that a connection to the target system has been established and a project has been downloaded to the target system. To load the project with the sample program, see "Executing the sample program (Page 66)".
- You can run the user program, but you do not have to. If the program is not run, you only see the initial values of the variables.

The procedure depends on the memory area in which the variables to be monitored are stored.

Variables in the user memory of the unit or in the retentive memory

You can use the symbol browser to monitor the variables contained in the user memory of the unit or in the retentive memory, see Memory ranges of the variable types (Page 194):

- Retentive and non-retentive unit variables of the interface section of a program source file (unit)
- Retentive and non-retentive unit variables of the implementation section of a program source file (unit)
- Static variables of the function blocks whose instances are declared as unit variables.
- In addition, if the program source file (unit) has been compiled with the "Create program instance data only once" compiler option (Page 44):
  - Static variables of the programs.
  - Static variables of the function blocks whose instances are declared as static variables of programs.

Follow these steps:

1. Select the program source file in the project navigator (e.g. ST_1).
2. In the detail view, click the Symbol browser tab.

You see in the symbol browser all variables of the program source file contained in the user memory of the unit or in the retentive memory.

- All unit variables of the program source file.
- Only if the program source file has been compiled with the "Create program instance data only once" compiler option: The programs of the program source file and below their static variables (including instances of function blocks).

Variables in the user memory of the task

You can use the symbol browser to monitor the variables contained in the user memory of the associated task, see Memory ranges of the variable types (Page 194):

If the program source (unit) was compiled **without** the compiler option (Page 44) "Create program instance data only once" (default), the user memory of the task to which the program was assigned contains the following variables:

- Static variables of the programs.
- Static variables of the function blocks whose instances are declared as static variables of programs.

Follow these steps:

1. In the project navigator of SIMOTION SCOUT, select the EXECUTION SYSTEM element in the subtree of the SIMOTION device.
2. In the detail view, click the Symbol browser tab.

The symbol browser shows all tasks used in the execution system together with the assigned programs. The associated variables contained in the user memory of the task are listed below.

Note

You can monitor temporary variables (together with unit variables and static variables) with Program status (see Properties of the program status (Page 265)).

System variables and global device variables

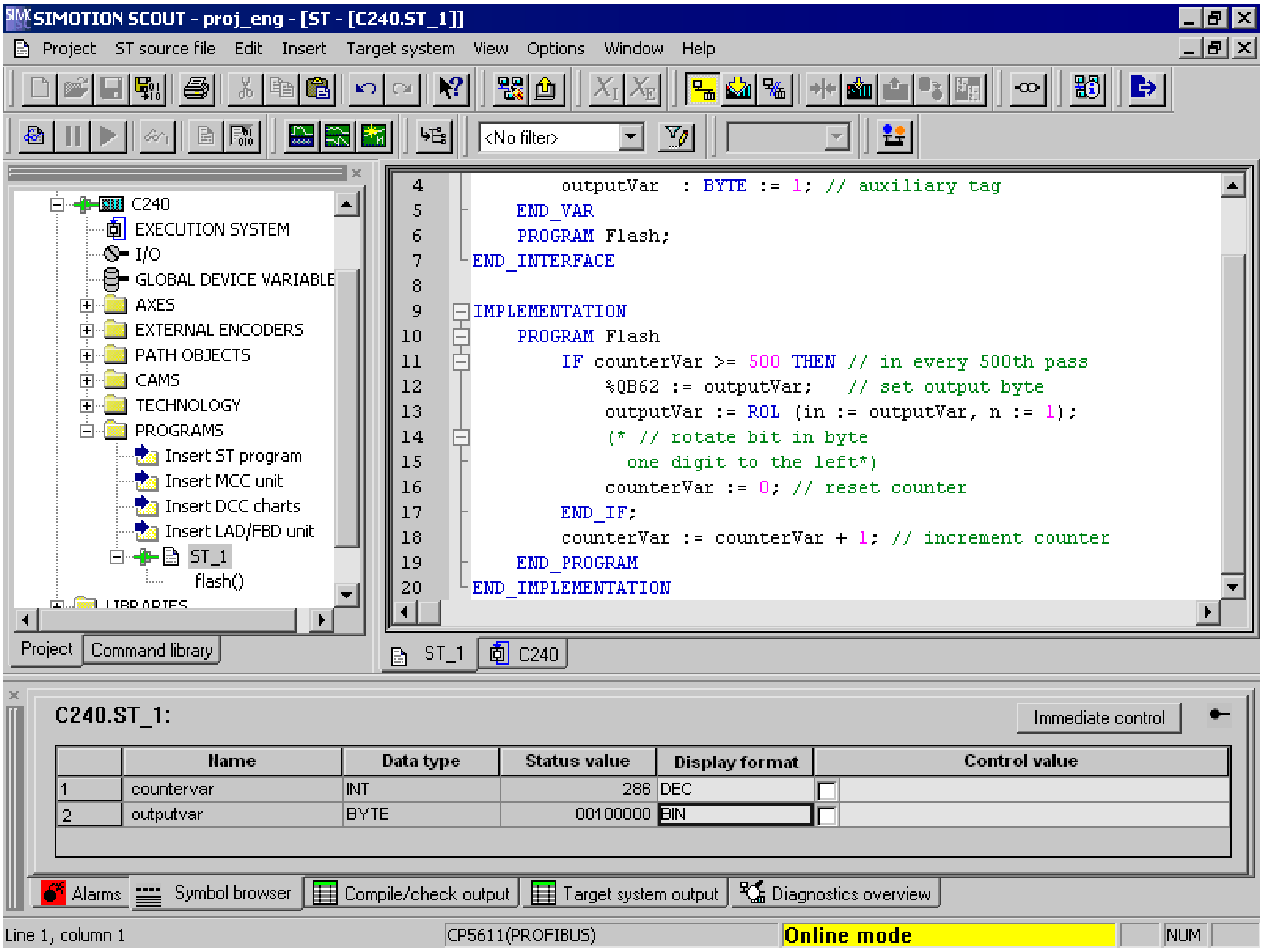
You can also monitor the following variables in the symbol browser:

- System variables of SIMOTION devices
- System variables of technology objects
- I/O variables
- Global device variables

Follow these steps:

1. Select the appropriate element in the SIMOTION SCOUT project navigator.
2. In the detail view, click the Symbol browser tab.

The corresponding variables are displayed in the symbol browser.



```
        outputVar  : BYTE := 1; // auxiliary tag
    END_VAR
    PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
    PROGRAM Flash
        IF counterVar >= 500 THEN // in every 500th pass
            %QB62 := outputVar;   // set output byte
            outputVar := ROL (in := outputVar, n := 1);
            (* // rotate bit in byte
              one digit to the left*)
            counterVar := 0; // reset counter
        END_IF;
        counterVar := counterVar + 1; // increment counter
    END_PROGRAM
END_IMPLEMENTATION
```

C240.ST_1:

	Name	Data type	Status value	Display format	Control value
1	countervar	INT	286	DEC	□
2	outputvar	BYTE	00100000	BIN	□

Figure 6-1 Displaying the contents of variables using the symbol browser

Status and controlling variables

In the **Status value** column, the current variable values are displayed and periodically updated.

You can change the value of one or several variables. Proceed as follows for the variables to be changed:

1. Enter a value in the **Control value** column.
2. Activate the checkbox in this column
3. Click the **Immediate control** button.

The values you entered are written to the selected variables.

NOTICE

Note when you change the values of several variables:

The values are written sequentially to the variables. It can take several milliseconds until the next value is written. The variables are changed from top to bottom in the symbol browser. There is therefore no guarantee of consistency.

Fix the display of the symbol browser

You can fix the display of the symbol browser for the active object:

- To do so, click the **Retain display**⊸ icon in the right upper corner of the symbol browser. The displayed symbol changes to 📌 .

  The variables of this object are still displayed and updated in the symbol browser even if another object is selected in the project navigator.
- To remove the display, click the 📌 icon again. The displayed symbol changes back to ⊸ .

Display invalid floating-point numbers

Invalid floating-point numbers are displayed as follows in the symbol browser (independently of the SIMOTION device):

Table 6-3 Display invalid floating-point numbers

LED	Meaning
1.#QNAN -1.#QNAN	Invalid bit pattern in accordance with IEEE 754 (NaN Not a Number) There is no distinction between signaling NaN (NaNs) and quiet NaN (NaNq).
1.#INF -1.#INF	Bit pattern for + infinity in accordance with IEEE 754 Bit pattern for – infinity in accordance with IEEE 754
-1.#IND	Bit pattern for indeterminate

6.2.3 Monitoring variables in watch table

6.2.3.1 Variables in the watch table

With the symbol browser you see only the variables of an object within the project. With program status you see only the variables of an ST source file within a freely selectable monitoring area.

With watch tables, in contrast, you can monitor selected variables from different sources as a group (e.g. program sources, technology objects, SINAMICS drives - even on different devices).

You can see the data type of the variables in offline mode. You can view and modify the value of the variables in online mode.

6.2.3.2 Using watch tables

You can group variables from various program sources, technology objects, SIMOTION devices, etc. (even on different devices), in a watch table where you can monitor them together and, if necessary, change them.

Creating a watch table

Procedure for creating a watch table and assigning variables:

1. In the project navigator, select the **Monitor** folder.
2. Select **Insert > Watch table** to create a watch table, and enter the name of the watch table. A watch table with this name appears in the **Monitor** folder.
3. In the project navigator, click the object from which you want to move variables to the watch table.
4. In the symbol browser, select the corresponding variable line by clicking its number in the left column.
5. From the context menu, select the item **Move variable to watch table** and the appropriate watch table, e.g. **Watch table_1**.
6. If you click the watch table, you will see in the detail view of the **Watch table** tab that the selected variable is now in the watch table.
7. Repeat steps 3 to 6 to monitor the variables of various objects.

If you are connected with the target system, you can monitor the variable contents.

Status and controlling variables

In the **Status value** column, the current variable values are displayed and periodically updated.

You can change the value of one or several variables. Proceed as follows for the variables to be changed:

1. Enter a value in the **Control value** column.
2. Activate the checkbox in this column
3. Click the **Immediate control** button.

The values you entered are written to the selected variables.

NOTICE
Note when you change the values of several variables: The values are written sequentially to the variables. It can take several milliseconds until the next value is written. The variables are changed from top to bottom in the watch table. There is therefore no guarantee of consistency.

Fix the display of the watch table

You can fix the display of the active watch table:

- To do so, click the **Retain display**●– icon in the right upper corner of the Watch table tab in the detail view. The displayed symbol changes to 📍 .

  This watch table is still displayed even if another one is selected in the project navigator.
- To remove the display, click the 📍 icon again. The displayed symbol changes back to ●– .

Display invalid floating-point numbers

Invalid floating-point numbers are displayed as follows in the watch table (independently of the SIMOTION device):

Table 6-4 Display invalid floating-point numbers

LED	Meaning
1.#QNAN -1.#QNAN	Invalid bit pattern in accordance with IEEE 754 (NaN Not a Number) There is no distinction between signaling NaN (NaNs) and quiet NaN (NaNq).
1.#INF -1.#INF	Bit pattern for + infinity in accordance with IEEE 754 Bit pattern for – infinity in accordance with IEEE 754
-1.#IND	Bit pattern for indeterminate

6.2.4 Program run

6.2.4.1 Program run: Display code location and call path

You can display the position in the code (e.g. line of an ST source file) that a MotionTask is currently executing along with its call path.

Follow these steps:

1. Click the "Show program run" button on the Program run toolbar.

   The "Program run call stack (Page 264)" window opens.

2. Select the desired MotionTask.
3. Click the "Update" button.

The window shows:

- The position in the code being executed (e.g. line of the ST source file) stating the program source and the POU.
- Recursively positions in the code of other POUs that call the code position being executed.

The following names are displayed for the SIMOTION RT program source files:

Table 6-5 SIMOTION RT program source files

Name	Meaning
taskbind.hid	Execution system
stdfunc.pck	IEC library
device.pck	Device-specific library
tp-name.pck	Library of the *tp-name* technology package, e.g. cam.pck for the library of the CAM technology package.

6.2.4.2 Parameter call stack program run

You can display the following for all configured tasks:

- the current code position in the program code (e.g. line of an ST source file)
- the call path of this code position

Table 6-6 Parameter description call stack program run

Field	Description
Selected CPU	The selected SIMOTION device is displayed.
Refresh	Clicking the button reads the current code positions from the SIMOTION device and shows them in the open window.
Calling task	Select the task for which you want to determine the code position being executed. All configured tasks of the execution system.
Current code position	The position being executed in the program code (e.g. line of an ST source file) is displayed (with the name of the program source file, line number, name of the POU).
is called by	The code positions that call the code position being executed within the selected task are shown recursively (with the name of the program source file, line number, name of the POU, and name of the function block instance, if applicable).

For names of the SIMOTION RT program sources, refer to the table in "Program run (Page 263)".

6.2.4.3 Program run toolbar

You can display the position in the code (e.g. line of an ST source file) that a MotionTask is currently executing along with its call path with this toolbar.

Table 6-7 Program run toolbar

Symbol	Meaning
	Display program run Click this button to open the Program run call stack window. In this window, you can display the currently active code position with its call path. See: Program run: Display code position and call path (Page 263)

6.2.5 Program status

6.2.5.1 Properties of the program status

Status program enables monitoring the variable values accurately to the cycle during program execution.

You can select a monitoring area in the ST source file and monitor, in addition to global and static local variables, also temporary local variables (e.g. within a function) there.

The values of the following variables are displayed:

- Simple data type variables (INT, REAL, etc.)
- Individual elements of a structure, provided an assignment is made
- Individual elements of an array, provided an assignment is made
- Enumeration data type variables

While the selected monitoring range is running in the ST source file, the corresponding buffer for the variables to be monitored is filled with the corresponding values on the SIMOTION device. Once the selected monitoring range has been run, the buffer is formatted for display in the SIMOTION SCOUT. SIMOTION SCOUT calls the formatted values at regular intervals and displays them.

As of SIMOTION Kernel V3.2, you can select a location in an ST source file at which a function or instance of a function block is called (call path). This enables you to observe the variable values specifically for this call.

Note

Due to the restricted buffer capacity and the requirement for minimum runtime tampering, the following variables cannot be displayed:

- Complete arrays
- Complete structures

Individual array elements or individual structure elements are displayed, however, provided an assignment is made in the ST source file.

Table 6-8 Differences between process mode and test mode in Program Status

	Process mode	Test mode
Optimization of program execution	For maximum system performance, only restricted diagnosis is possible	For full diagnosis options
Maximum number of monitored program sources (e.g. ST source files, MCC source files, LAD/FBD sources)	• As of version V4.0 of the SIMOTION kernel: Maximum 1 program source **per task** • Up to version V3.2 of the SIMOTION kernel: Maximum 1 program source	• As of version V4.0 of the SIMOTION kernel: Multiple program sources per task • Up to version V3.2 of the SIMOTION kernel: Maximum 1 program source **per task**
Loops (e.g. WHILE, REPEAT, FOR)	On repeat loops, the recording is interrupted. If the whole loop is selected, the values are displayed on the first run of the loop.	If there are repeats, the recording continues correctly. If the whole loop is selected, the values are displayed on the last run of the loop.
System functions that contain internal loops (e.g. functions for processing strings)	Values are not displayed in some cases	Values are displayed correctly.

NOTICE

Program status requires additional CPU resources.

Please note if you want to monitor several programs at the same time with the status program:

- Test mode must be activated (see Operating modes of the SIMOTION devices (Page 252))
- Up to version V4.0 of the SIMOTION Kernel, the programs must be assigned to various tasks.

6.2.5.2 Using the status program

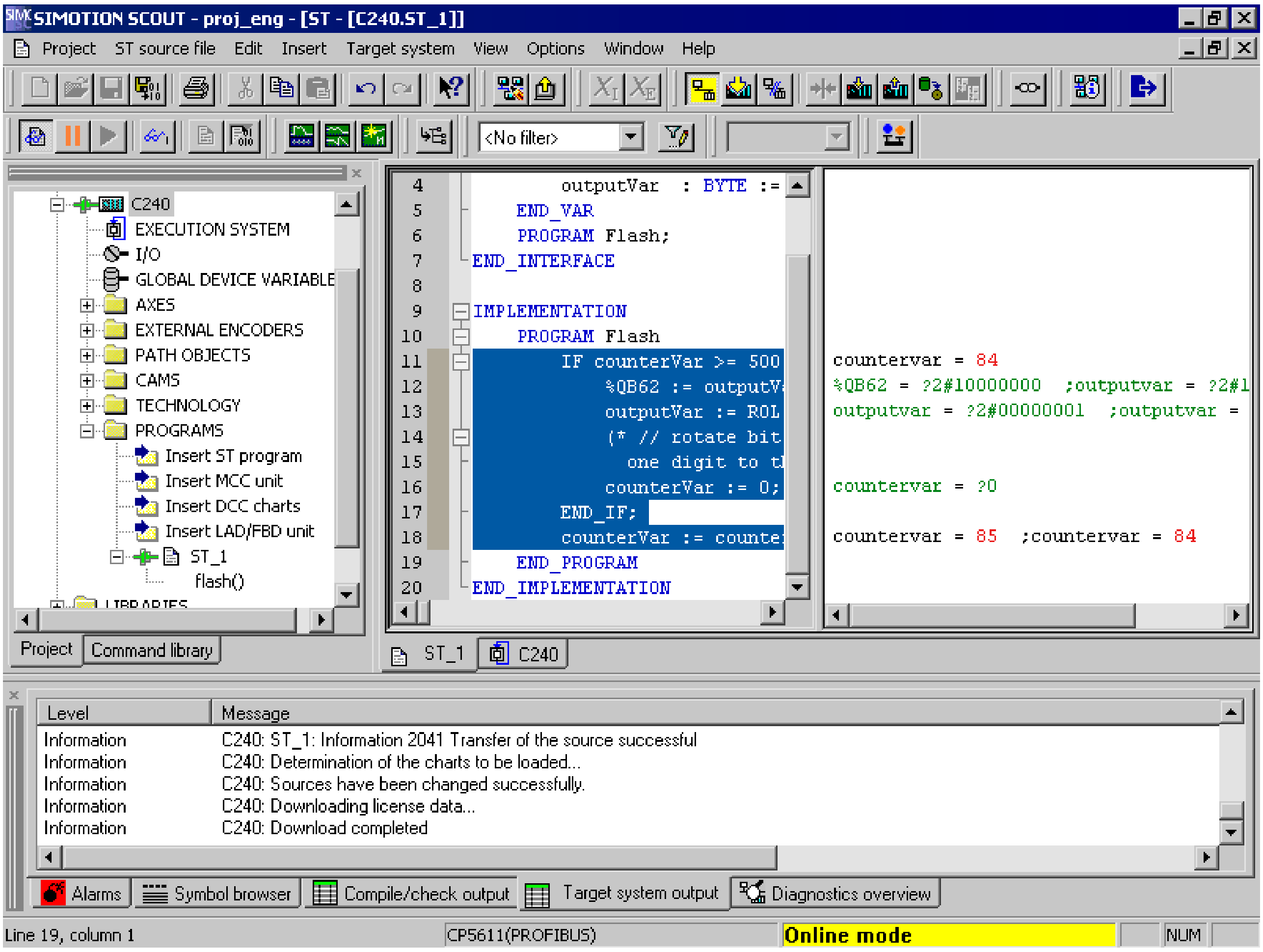
Before you can work with the Status program, you must instruct the system to run in a special mode:

1. Make sure that the ST source file generates the additional debug code during compilation:
   - Select the ST source file in the project navigator and select the **Edit > Object properties** menu command.
   - Select the **Compiler** tab to change the local settings of the compiler (Page 46).
   - Make sure that the **Enable Status program** checkbox is activated and confirm with **OK.**

     You can also change this compiler option at global settings of the compiler (Page 45).
2. Open the ST source file and recompile it with **ST source file > Accept and compile.**
3. Download and start the program in the usual way.

4. Click the button for **program status** in the ST editor toolbar (Page 43) to start this test mode.

The ST editor window is now divided vertically: You can see the ST source file in the left pane; the right pane displays the selected variables and their values.



```
            outputVar   : BYTE :=
        END_VAR
        PROGRAM Flash;
END_INTERFACE

IMPLEMENTATION
        PROGRAM Flash
            IF counterVar >= 500
                %QB62 := outputV
                outputVar := ROL
                (* // rotate bit
                  one digit to t
                counterVar := 0;
            END_IF;
            counterVar := counte
        END_PROGRAM
END_IMPLEMENTATION
```

```
countervar = 84
%QB62 = ?2#10000000  ;outputvar = ?2#1
outputvar = ?2#00000001  ;outputvar =


countervar = ?0

countervar = 85  ;countervar = 84
```

Level	Message
Information	C240: ST_1: Information 2041 Transfer of the source successful
Information	C240: Determination of the charts to be loaded...
Information	C240: Sources have been changed successfully.
Information	C240: Downloading license data...
Information	C240: Download completed

Figure 6-2 Part of an ST program in program status test mode

Follow the procedure below to test with program status:

1. In the editor, select the section of the ST source file you want to test.
2. As of version V3.2 of the SIMOTION Kernel:

   If you have selected a section of a POU that is called by several positions in a program source file or several tasks:

   Enter the call path for program status (Page 268).

For the selected section, you can see variables and their values in the right pane of your screen; they are updated cyclically:

- Values that have changed in the current pass are displayed in **red**.
- Values that have not changed are displayed in **black**.

- Variables without values, e.g. variables in an unused IF branch are shown in **green** and marked with a question mark.

If the display of the variable values changes too fast:

- Click the ⏸ button for **Stop monitoring of program variables** in the ST editor toolbar (Page 43) to stop the display.
- Click the ▶ button for **Continue monitoring of program variables** in the ST editor toolbar (Page 43) to continue the display.

You can force the update of the displayed values:

- Click the button for **Update** on the ST editor toolbar (Page 43).

  The buffer of the SIMOTION device is read, even if the selected monitoring range has not yet been completely processed and the values are incomplete. This can be useful, for example, if the program is waiting for a WAITFORCONDITION statement.

  The monitoring of the program variables must have been activated.

6.2.5.3 Call path for program status

With SIMOTION Kernel V3.2 and higher, you can specify the call path when monitoring variable values of functions and function blocks. This enables you to observe the variable values specifically for this call.

For this purpose, the **Call path** window automatically opens in the following cases:

- You have selected a section of a function:

  The function is called at various points in the program source files (e.g. ST source files) of the SIMOTION device.
- You have selected a section of a function block:

  There are several instances of the function block or the instance is called at various points in the program source files (e.g. ST source files) of the SIMOTION device.
- You have selected a section of a program:

  The program is assigned to more than one task.

How to select the call path:

In the **Call path status program** window, the marked section of the POU (code position) is displayed (with the name of the ST source file, line number, name of the POU).

1. If the code position is called in several tasks:
   - Select the task.
2. Select the code position to be called (in the calling POU).

   You can select from the following:
   - The code positions to be called within the selected task (with the name of the program source, line number, name of the POU).

     If the selected calling code position is in turn called by several code positions, further lines are displayed in which you proceed similarly.
   - **All**:

     All displayed code positions are selected. Moreover, all code positions (up to the top level of the hierarchy) are selected from which the displayed code positions are called.

Program Status for devices with SIMOTION kernel versions up to V3.1

NOTICE

Note the following if you use Program Status in devices with SIMOTION kernels up to V3.1:

- If the project was compiled using SIMOTION SCOUT up to version V3.1, the call path is not available in the described format. You can only use the diagnosis functions available at the time of compilation.
- You can only specify the call path if the project was compiled using SIMOTION SCOUT version V3.2 or higher.

When performing a recompilation with the current version of the compiler, note the following:

- Among other effects, this generates new version identifiers in the data storage areas of the programs.
- All retentive and non-retentive data on the SIMOTION device is deleted and initialized.
- In some cases, minor changes to the program sources may be required.
- When converting back to the old project status, the project must be recompiled.

6.2.5.4 Parameter call path status program

Table 6-9 Program status call path parameter description

Field	Description
Calling task	Select the task. All tasks in which the selected code position is called are available for selection.
Current code position	The selected section of the POU (code position) is shown (with the name of the ST source file, line number, name of the POU)
is called by	Select the calling code position. The following are available: • The code positions to be called within the selected task (with the name of the program source, line number, name of the POU). If the selected calling code position is in turn called by several code positions, further lines are displayed in which you proceed similarly. • **All:** All displayed code positions are selected. Moreover, all code positions (up to the top level of the hierarchy) are selected from which the displayed code positions are called.

6.2.6 Breakpoints

6.2.6.1 General procedure for setting breakpoints

You can set breakpoints within a POU of a program source (e.g. ST source, MCC chart, LAD/FBD source). On reaching an activated breakpoint, the task in which the POU with the breakpoint is called is stopped. If the breakpoint that initiated the stopping of the tasks is located in a program or function block, the values of the static variables for this POU are displayed in the "Variables status" tab of the detail display. Temporary variables (also in/out parameters for function blocks) are not displayed. You can monitor static variables of other POUs or unit variables in the symbol browser.

Requirement:

- The program source with the POU (e.g. ST source file, MCC chart, LAD/FBD program) is open.

Proceed as follows

Follow these steps:

1. Select "Debug mode" for the associated SIMOTION device,
   see Set debug mode (Page 271).
2. Specify the debug task group, see Specifying the debug task group (Page 273).
3. Set breakpoints, see Setting breakpoints (Page 276).
4. Define the call path, see Defining a call path for a single breakpoint (Page 279).
5. Activate the breakpoints, see Activating breakpoints (Page 285).

6.2.6.2 Setting the debug mode

⚠WARNING

You must observe the appropriate safety regulations.

Use the debug mode only with activated life-sign monitoring (Page 254) with a suitably short monitoring time! Otherwise, if problems occur in the communication link between the PC and the SIMOTION device, the axis may start moving in an uncontrollable manner.

The function is released exclusively for commissioning, diagnostic and service purposes. The function should generally only be used by authorized technicians. The safety shutdowns of the higher-level control have no effect!

Therefore, there must be an EMERGENCY STOP circuit in the hardware. The appropriate measures must be taken by the user.

To set the debug mode, proceed as follows:

1. Highlight the SIMOTION device in the project navigator.
2. Select **Test mode** from the context menu.
3. Select **Debug**mode (Page 252).
4. Accept the safety information
5. Parameterize the sign-of-life monitoring.

   See also section: Important information about the life-sign monitoring (Page 254).
6. Confirm with **OK**.

   If no connection has been established with the target system (offline mode), the online mode will be established automatically.

   The activated debug mode is indicated in the status bar.

   The breakpoints toolbar (Page 278) is displayed.

Note

You cannot change the program sources in debug mode!

NOTICE

Pressing the spacebar or switching to a different Windows application causes in debug mode for activated breakpoints:

- The SIMOTION device switches to STOP mode.
- The outputs are deactivated (ODIS).

⚠WARNING

This function is not guaranteed in all operating modes. Therefore, there must be an EMERGENCY STOP circuit in the hardware. The appropriate measures must be taken by the user.

6.2.6.3 Define the debug task group

On reaching an activated breakpoint, all tasks that are assigned to the debug task group are stopped.

Requirement

- The relevant SIMOTION device is in debug mode.

Proceed as follows

How to assign a task to the debug task group:

1. Highlight the relevant SIMOTION device in the project navigator.
2. Select **Debug task group** from the context menu.

   The Debug Task group window opens.
3. Select the tasks to be stopped on reaching the breakpoint:
   - If you only want to stop individual tasks (in RUN mode): Activate the **Debug task group** selection option.

     Assign all tasks to be stopped on reaching a breakpoint to the **Tasks to be stopped** list.
   - If you only want to stop individual tasks (in HALT mode): Activate the **All tasks** selection option.

     In this case, also select whether the outputs and technology objects are to be released again after resumption of program execution.

NOTICE
Note the different behavior when an activated breakpoint is reached, see the following table.

Table 6-10 Behavior at the breakpoint depending on the tasks to be stopped in the debug task group.

Properties		Tasks to be stopped	
		Single selected tasks (debug task group)	All tasks
Behavior on reaching the breakpoint			
	Operating mode	RUN	STOP
	Stopped tasks	Only tasks in the debug task group	All tasks
	Outputs	Active	Deactivated (ODIS activated)
	Technology	Closed-loop control active	No closed-loop control (ODIS activated)
	Runtime measurement of the tasks	Active for all tasks	Deactivated for all tasks
	Time monitoring of the tasks	Deactivated for tasks in the debug task group	Deactivated for all tasks
	Real-time clock	Continues to run	Continues to run
Behavior on resumption of program execution			
	Operating mode	RUN	RUN
	Started tasks	All tasks in the debug task group	All tasks
	Outputs	Active	The behavior of the outputs and the technology objects depends on the **'Continue' activates the outputs (ODIS deactivated)** checkbox. • Active: ODIS will be deactivated. All outputs and technology objects are released. • Inactive: ODIS remains activated. All outputs and technology objects are only released after another download of the project.
	Technology	Closed-loop control active	

Note

You can only make changes to the debug task group if no breakpoints are active.

Proceed as follows:

1. Set breakpoints (see Setting breakpoints (Page 276)).
2. Define the call path (see Defining a call path for a single breakpoint (Page 279)).
3. Activate the breakpoints (see Activating breakpoints (Page 285)).

6.2.6.4 Debug task group parameters

Use this window to define the debug task group. On reaching an activated breakpoint, all tasks that are assigned to the debug task group are stopped.

This requires that the relevant SIMOTION device is in debug mode, see Modes of the SIMOTION devices (Page 252).

Table 6-11 Debug settings parameter description

Field	Description
Debug task group	Select this selection option if you only want to stop individual tasks. The SIMOTION device remains in RUN mode after an activated breakpoint is reached. Outputs and technology objects remain activated. Assign all tasks to be stopped on reaching a breakpoint to the **Tasks to be stopped** list.
All tasks	Select this selection option if you only want to stop all user tasks. The SIMOTION device remains in STOP mode after an activated breakpoint is reached, all outputs and technology objects will be deactivated (ODIS activated). In this case, also select whether the outputs and technology objects are to be released again after resumption of program execution.
'Resume' activates the outputs (ODIS deactivated).	Only if **All tasks** is selected. Activate the checkbox, to release again the outputs and technology objects after program execution has been resumed. All outputs and technology objects can only be released after a download of the project with deactivated checkbox.

NOTICE

Note the different behavior at the activated breakpoint depending on the tasks to be stopped, see table in Define the debug task group (Page 273).

You can only make changes to the debug task group if no breakpoints are active.

6.2.6.5 Debug table parameter

The debug table shows all debug points (e.g. breakpoints, trace points) in the program sources of a SIMOTION device.

Table 6-12 Debug table parameter description

Field		Description
Debug points (table)		
	Active	The activation state of the breakpoint is displayed. Click the checkbox to change the activation state. See: Activating breakpoints (Page 285).
	Source, line (POU)	The code position is shown with the debug point set (with the name of the program source file, line number, name of the POU).
	Debug type	The type of the debug point is shown (e.g. breakpoint, trace point).
	Call path	Click the button to define the call path for the breakpoint. See: Defining the call path for a single breakpoint (Page 279).
All breakpoints ...		
	Activate	Click the button to activate all breakpoints (in all program sources) of the SIMOTION device. See: Activating breakpoints (Page 285).
	Deactivate	Click the button to deactivate all breakpoints (in all program sources) of the SIMOTION device. See: Activating breakpoints (Page 285).
	Delete	Click the button to clear all breakpoints (in all program sources) of the SIMOTION device. See: Setting breakpoints (Page 276).

6.2.6.6 Setting breakpoints

Requirements:

1. The program source with the POU (e.g. ST source file, MCC chart, LAD/FBD program) is open.
2. The relevant SIMOTION device is in debug mode,
   see Setting debug mode (Page 271).
3. The debug task group is defined, see Defining the debug task group (Page 273).

Proceed as follows

How to set a breakpoint:

1. Select the code location where no breakpoint has been set:
   - SIMOTION ST: Place the cursor on a line in the ST source file that contains a statement.
   - SIMOTION MCC: Select an MCC command in the MCC chart (except module or comment block).
   - SIMOTION LAD/FBD: Set the cursor in a network of the LAD/FBD program.
2. Alternative:
   - Select the **Edit > Set breakpoint** menu command.
   - Click the ● button in the Breakpoints toolbar.

To remove a breakpoint, proceed as follows:

1. Select the code position with the breakpoint.
2. Alternative:
   - Select the **Edit > Set breakpoint** menu command.
   - Click the ● button in the Breakpoints toolbar.

To remove all breakpoints (in all program sources) of the SIMOTION device, proceed as follows:

- Alternative:
  - Select the **Debug > Remove all breakpoints** menu command.
  - Click the ✖ button in the Breakpoints toolbar.

Note

You cannot set breakpoints:

- For SIMOTION ST: In lines that contain only comment.
- For SIMOTION MCC: On the module or comment block commands.
- For SIMOTION LAD/FBD: Within a network.
- At code locations in which other debug points (e.g. trigger points) have been set.

You can list the debug points in all program sources of the SIMOTION device in the debug table:

- Click the 📋 button for "debug table" in the Breakpoints toolbar.

In the debug table, you can also remove all breakpoints (in all program sources) of the SIMOTION device:

- Click the button for "Clear all breakpoints".

Set breakpoints remain saved also after leaving the "debug mode", they are displayed only in debug mode.

You can use the program status (Page 266) diagnosis functions and breakpoints together in a program source file or POU. However, the following restrictions apply depending on the program languages:

- SIMOTION ST: For Version V3.2 of the SIMOTION Kernel, the (marked) ST source file lines to be tested with program status must not contain a breakpoint.
- SIMOTION MCC and LAD/FBD: The commands of the MCC chart (or networks of the LAD/FBD program) to be tested with program status must not contain a breakpoint.

Proceed as follows

1. Define the call path, see Defining a call path for a single breakpoint (Page 279).
2. Activate the breakpoints, see Activating breakpoints (Page 285).

6.2.6.7 Breakpoints toolbar

This toolbar contains important operator actions for setting and activating breakpoints:

Table 6-13 Breakpoints toolbar

Symbol	Meaning
●	Set/remove breakpoint Click this icon to set at breakpoint for the selected code position or to remove an existing breakpoint. See: Setting breakpoints (Page 276).
◐	Activate/deactivate breakpoint Click this icon to activate or deactivate the breakpoint at the selected code position. See: Activating breakpoints (Page 285).
⊞	Edit the call path Click this icon to define the call path for the breakpoints: • If a code position **with** breakpoint is selected: The call path for this breakpoint. • If a code position **without** breakpoint is selected: The call path for all breakpoints of the POU. See: Defining the call path for a single breakpoint (Page 279), Defining the call path for all breakpoints (Page 282).
⁝	Activate all breakpoints Click this icon to activate all breakpoints in the current program source or POU (e.g. ST source file, MCC chart, LAD/FBD program). See: Activating breakpoints (Page 285).
⁝	Deactivate all breakpoints Click this icon to deactivate all breakpoints in the current program source or POU (e.g. ST source file, MCC chart, LAD/FBD program). See: Activating breakpoints (Page 285).
✖	Remove all breakpoints Click this icon to remove all breakpoints in the current program source or POU (e.g. ST source file, MCC chart, LAD/FBD program). See: Setting breakpoints (Page 276).

Symbol	Meaning
	Debug table Click this icon to display the debug table. See: Debug table parameters (Page 276).
	Display call stack Click this icon after reaching an activated breakpoint to: • View the call path at the current breakpoint. • View the code positions at which the other tasks of the debug task group have been stopped together with their call path. See: Displaying the call stack (Page 287).
	Resume Click this icon to continue the program execution after reaching an activated breakpoint. See: Activating breakpoints (Page 285), Displaying the call stack (Page 287).

6.2.6.8 Defining the call path for a single breakpoint

Requirements:

1. The program source with the POU (e.g. ST source file, MCC chart, LAD/FBD program) is open.
2. The relevant SIMOTION device is in debug mode,
   see Setting debug mode (Page 271).
3. The debug task group is defined, see Defining the debug task group (Page 273).
4. Breakpoint is set, see Setting breakpoints (Page 276).

Proceed as follows

To define the call path for a single breakpoint, proceed as follows:

1. Select the code location where a breakpoint has already been set:
   - SIMOTION ST: Set the cursor in an appropriate line of the ST source.
   - SIMOTION MCC: Select an appropriate command in the MCC chart.
   - SIMOTION LAD/FBD: Set the cursor in an appropriate network of the LAD/FBD program.
2. Click the button for "edit call path" in the Breakpoints toolbar.

   In the Call path / task selection breakpoint window, the marked code position is displayed (with the name of the program source file, line number, name of the POU).
3. Select the task in which the user program (i.e. all tasks in the debug task group) will be stopped when the selected breakpoint is reached.

   The following are available:

   - **All calling locations starting at this call level**

     The user program will always be started when the activated breakpoint in any task of the debug task group is reached.

   - The individual tasks from which the selected breakpoint can be reached.

     The user program will be stopped only when the breakpoint in the selected task is reached. The task must be in the debug task group.

     The specification of a call path is possible.

4. Only for functions and function blocks: Select the call path, i.e. the code position to be called (in the calling POU).

   The following are available:

   - **All calling locations starting at this call level**

     No call path is specified. The user program is always stopped at the activated breakpoint if the POU in the selected tasks is called.

   - Only when a single task is selected: The code positions to be called within the selected task (with the name of the program source, line number, name of the POU).

     The call path is specified. The user program will be stopped at the activated breakpoint only when the POU is called from the selected code position.

     If the POU of the selected calling code position is also called from other code positions, further lines are displayed successively in which you proceed similarly.

5. If the breakpoint is only to be activated after the code position has been reached several times, select the number of times.

Note

You can also define the call path to the individual breakpoints in the debug table:

1. Click the button for "debug table" in the Breakpoints toolbar.

   The "Debug table" window opens.
2. Click the appropriate button in the "Call path" column.
3. Proceed in the same way as described above:
   - Specify the task.
   - Define the call path (only for functions and function blocks).
   - Specify the number of passes after which the breakpoint is to be activated.

Proceed as follows:

- Activate the breakpoints, see Activating breakpoints (Page 285).

Note

You can use the "Display call stack (Page 287)" function to view the call path at a current breakpoint and the code positions at which the other tasks of the debug task group were stopped.

See also

Defining the call path for all breakpoints (Page 282)

6.2.6.9 Breakpoint call path / task selection parameters

Table 6-14 Breakpoint call path / task selection parameter description

Field	Description
Selected CPU	The selected SIMOTION device is displayed.
Calling task	Select the task in which the user program (i.e. all tasks in the debug task group) will be stopped when the selected breakpoint is reached. The following are available: • **All calling locations starting at this call level** The user program will always be started when the activated breakpoint in any task of the debug task group is reached. • The individual tasks from which the POU with the selected breakpoint can be reached. The user program will be stopped only when the breakpoint in the selected task is reached. The task must be in the debug task group. The specification of a call path is possible.
Current code position	The code position is shown with the set breakpoint (with the name of the program source file, line number, name of the POU).

Field	Description
is called by	Only for functions and function blocks: Select the call path, i.e. the code position to be called (in the calling POU). The following are available: • **All calling locations starting at this call level** **No** call path is specified. The user program will always be stopped at the activated breakpoint when the POU in the tasks is reached. • Only when a single task is selected: The code positions to be called within the selected task (with the name of the program source, line number, name of the POU). The call path is specified. The user program will be stopped at the activated breakpoint only when the POU is called from the selected code position. If the POU of the selected calling code position is also called from other code positions, further lines are displayed successively in which you proceed similarly.
The breakpoint will be activated at each nth pass.	If you do not want the breakpoint to be activated until the code position has been reached a certain number of times, set this number.

NOTICE
You can only make changes to the debug task group if no breakpoints are active.

6.2.6.10 Defining the call path for all breakpoints

With this procedure, you can:

- Select a default setting for all future breakpoints in a POU (e.g. MCC chart, LAD/FBD program or POU in an ST source file).
- Accept and compare the call path for all previously set breakpoints in this POU.

Requirements

- The program source with the POU (e.g. ST source file, MCC chart, LAD/FBD program) is open.
- The relevant SIMOTION device is in debug mode,
  see Setting debug mode (Page 271).
- The debug task group is defined, see Defining the debug task group (Page 273).

Proceed as follows

To define the call path for all future breakpoints of a POU, proceed as follows:

1. Select the code location where **no** breakpoint has been set:
   - SIMOTION ST: Set the cursor in an appropriate line of the ST source.
   - SIMOTION MCC: Select an appropriate command in the MCC chart.
   - SIMOTION LAD/FBD: Set the cursor in an appropriate network of the LAD/FBD program.
2. Click the button for "edit call path" in the Breakpoints toolbar.

   In the "Call path / task selection all breakpoints for each POU" window, the marked code position is displayed (with the name of the program source file, line number, name of the POU).
3. Select the task in which the user program (i.e. all tasks in the debug task group) will be stopped when a breakpoint in this POU is reached.

   The following are available:
   - **All calling locations starting at this call level**

     The user program will always be started when an activated breakpoint of the POU in any task of the debug task group is reached.
   - The individual tasks from which the selected breakpoint can be reached.

     The user program will be stopped only when a breakpoint in the selected task is reached. The task must be in the debug task group.

     The specification of a call path is possible.
4. Only for functions and function blocks: Select the call path, i.e. the code position to be called (in the calling POU).

   The following are available:
   - **All calling locations starting at this call level**

     No call path is specified. The user program is always stopped at an activated breakpoint when the POU in the selected tasks is called.
   - Only when a single task is selected: The code positions to be called within the selected task (with the name of the program source, line number, name of the POU).

     The call path is specified. The user program will be stopped at an activated breakpoint only when the POU is called from the selected code position.

     If the selected calling code position is in turn called by other code positions, further lines are displayed successively in which you proceed similarly.
5. If a breakpoint is only to be activated after the code position has been reached several times, select the number of times.
6. If you want to accept and compare this call path for all previously set breakpoints in this POU:
   - Click **Accept**.

Proceed as follows:

- Activate the breakpoints, see Activating breakpoints (Page 285).

Note

You can use the "Display call stack (Page 287)" function to view the call path at a current breakpoint and the code positions at which the other tasks of the debug task group were stopped.

See also

Defining the call path for a single breakpoint (Page 279)

6.2.6.11 Call path / task selection parameters of all breakpoints per POU

Here you can define a presetting for the call path of all future breakpoints to be set in a POU. Moreover, you can also accept this setting for all previously set breakpoints of this POU.

Table 6-15 Call path / task selection parameter description of all breakpoints per POU

Field	Description
Selected CPU	The selected SIMOTION device is displayed.
Calling task	Select the task in which the user program (i.e. all tasks in the debug task group) will be stopped when a breakpoint in this POU is reached. The following are available: • **All calling locations starting at this call level** The user program will always be started when an activated breakpoint of the POU in any task of the debug task group is reached. • The individual tasks from which the selected breakpoint can be reached. The user program will be stopped only when an activated breakpoint in the selected task is reached. The task must be in the debug task group. The specification of a call path is possible.
Current POU	The POU in which the cursor is located is displayed (with the name of the program source file, name of the POU).

Field	Description
is called by	Only for functions and function blocks: Select the call path, i.e. the code position to be called (in the calling POU). The following are available: • **All calling locations starting at this call level** **No** call path is specified. The user program will always be stopped at an activated breakpoint when the POU in the selected tasks is called. • Only when a single task is selected: The code positions to be called within the selected task (with the name of the program source, line number, name of the POU). The call path is specified. The user program will be stopped at an activated breakpoint only when the POU is called from the selected code position. If the POU of the selected calling code position is also called from other code positions, further lines are displayed successively in which you proceed similarly.
The breakpoint will be activated at each nth pass.	If you do not want the breakpoint to be activated until the code position has been reached a certain number of times, set this number.
Apply this call path to all previous breakpoints of this POU	Click the **Apply** button, if you want to apply the call path to all previously set breakpoints of the current POU. Any existing settings will be overwritten.

6.2.6.12 Activating breakpoints

Breakpoints must be activated if they are to have an effect on program execution.

Requirements

1. The program source with the POU (e.g. ST source file, MCC chart, LAD/FBD program) is open.
2. The relevant SIMOTION device is in debug mode,
   see Setting debug mode (Page 271).
3. The debug task group is defined, see Defining the debug task group (Page 273).
4. Breakpoints are set, see Setting breakpoints (Page 276).
5. Call paths are defined, see Defining a call path for a single breakpoint (Page 279).

Activating breakpoints

How to activate a single breakpoint:

1. Select the code location where a breakpoint has already been set:
   - SIMOTION ST: Set the cursor in an appropriate line of the ST source.
   - SIMOTION MCC: Select an appropriate command in the MCC chart.
   - SIMOTION LAD/FBD: Set the cursor in an appropriate network of the LAD/FBD program.
2. Alternative:
   - Select the **Debug > Activate/deactivate breakpoint** menu command.
   - Click the button in the Breakpoints toolbar.

To activate all breakpoints (in all program sources) of the SIMOTION device, proceed as follows:

- Alternative:
  - Select the **Debug > Activate all breakpoints** menu command.
  - Click the button in the Breakpoints toolbar.

Note

Breakpoints of all program sources of the SIMOTION device can also be activated and deactivated in the debug table:

1. Click the button for "debug table" in the Breakpoints toolbar.
   The "Debug table" window opens.
2. Perform the action below, depending on which breakpoints you want to activate or deactivate:
   - Single breakpoints: Check or clear the corresponding checkboxes.
   - All breakpoints (in all program sources): Click the corresponding button.

Behavior at the activated breakpoint

On reaching an activated breakpoint (possibly using the selected call path (Page 279)), all tasks assigned to the debug task group will be stopped. The behavior depends on the tasks in the debug task group and is described in "Defining a debug task group (Page 273)". The breakpoint is highlighted.

If the breakpoint that initiated the stopping of the tasks is located in a program or function block, the values of the static variables for this POU are displayed in the "Variables status" tab of the detail display. Temporary variables (also in/out parameters for function blocks) are not displayed. You can monitor static variables of other POUs or unit variables in the symbol browser (Page 257).

You can use the "Display call stack (Page 287)" function to:

- View the call path at the current breakpoint.
- View the code positions with the call path at which the other tasks of the debug task group have been stopped.

Resuming program execution

How to resume program execution:

- Click the button for "resume" (Ctrl+F8 shortcut) in the Breakpoint toolbar.

Deactivate breakpoints

To deactivate a single breakpoint, proceed as follows:

1. Select the code position with the activated breakpoint.
2. Alternative:
   - Select the **Debug > Activate/deactivate breakpoint** menu command.
   - Click the button in the Breakpoints toolbar.

To deactivate all breakpoints (in all program sources) of the SIMOTION device, proceed as follows:

- Alternative:
  - Select the **Debug > Deactivate all breakpoints** menu command.
  - Click the button in the Breakpoints toolbar.

6.2.6.13 Display call stack

You can use the "Display call stack" function to:

- View the call path at the current breakpoint.
- View the code positions with the call path at which the other tasks of the debug task group have been stopped.

Requirement

The user program is stopped at an activated breakpoint, i.e. the tasks of the debug task group (Page 273) have been stopped.

Proceed as follows

To call the "Display call stack" function, proceed as follows:

- Click the button for "display call stack" in the Breakpoints toolbar.

  The "Breakpoint call stack" dialog opens. The current call path (including the calling task and the number of the set passes) is displayed.

  The call path cannot be changed.

To use the "Display call stack" function, proceed as follows:

1. Keep the "Breakpoint call stack" dialog open.
2. To display the code position at which the other task was stopped, proceed as follows:
   – Select the appropriate task. All tasks of the debug task group can be selected.

   The code position, including the call path, is displayed. If the code position is contained in a user program, the program source with the POU (e.g. ST source file, MCC chart, LAD/FBD program) will be opened and the code position marked.
3. How to resume program execution:
   – Click the button for "resume" (Ctrl+F8 shortcut) in the Breakpoint toolbar.

   When the next activated breakpoint is reached, the tasks of the debug task group will be stopped again. The current call path, including the calling task, is displayed.
4. Click "OK" to close the "Breakpoint call stack" dialog.

For names of the SIMOTION RT program sources, refer to the table in "Program run (Page 263)".

6.2.6.14 Breakpoints call stack parameter

When an activated breakpoint (Page 285) is reached, you can display the following for each task in the debug task group (Page 273):

- The position in the program code (e.g. line of an ST source file) at which the task stopped.
- The call path of this code position.

Table 6-16 Breakpoint call path parameter description

Field	Description
Selected CPU	The selected SIMOTION device is displayed.
Calling task	Select the task for which you want to display the code position at which the task was stopped. All tasks of the debug task group can be selected.
Current code position	The position in the program code (e.g. line of an ST source file) at which the selected task was stopped is displayed (with the name of the program source file, line number, name of the POU).
is called by	The code positions that call the current code position within the selected task are shown recursively (with the name of the program source file, line number, name of the POU, and name of the function block instance, if applicable).

For names of the SIMOTION RT program sources, refer to the table in "Program run (Page 263)".

6.2.7 Trace

Using the **trace tool**, you can record and store the course of variable values over time (z. B. unit variables, local variables, system variables, I/O variables). This allows you to document the optimization, for example, of axes.

You can set the recording time, display up to four channels, select trigger conditions, parameterize timing adjustments, select between different curve displays and scalings, etc.

Aside from isochronous recording, you can also select **Recording at code position**. This lets you record the values of variables whenever the program runs through a specific point in the ST source file.

The trace tool is described in detail in the online help.

Appendix A

A.1 Formal Language Description

In this chapter, you will find overviews of the basic elements of ST and a complete compilation of all syntax diagrams with the language elements. This appendix summarizes the basic features of the ST language.

A.1.1 Language description resources

Syntax diagrams are used as a basis for the language description in the individual sections. They provide you with an invaluable insight into the syntactic (i.e. grammatical) structure of ST.

Instructions for using syntax diagrams were presented in *Language description resources*. Information about the difference between formatted and unformatted rules, of interest to the advanced user, is presented below.

A.1.1.1 Formatted rules (lexical rules)

The lexical rules describe the structure of the elements processed by the compiler during lexical analysis. This means that the notation is formatted and the rules must be followed. In particular, that means:

- Insertion of formatting characters is not allowed.
- Block and line comments cannot be inserted.
- Attributes for identifiers cannot be inserted.

The following figure shows a lexical rule for legal identifiers.

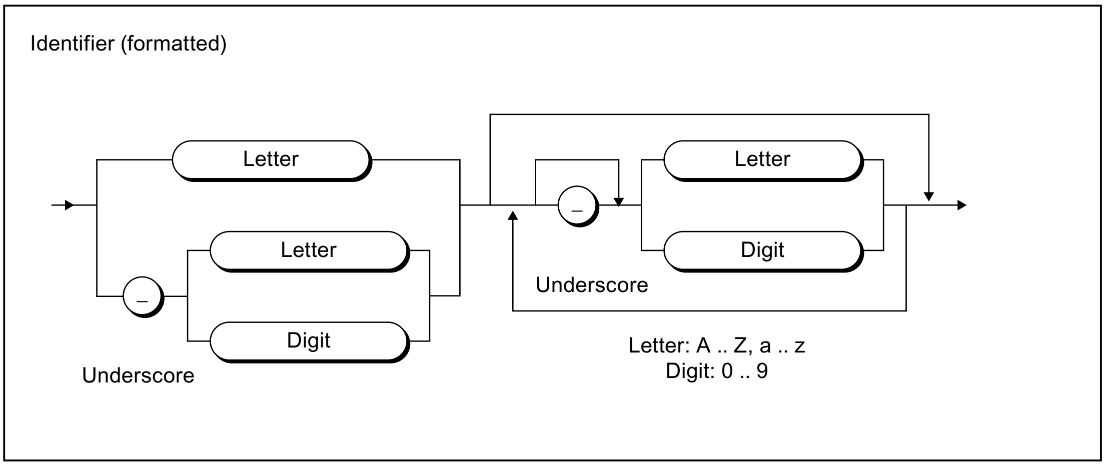


Figure A-1 Example of a lexical rule

Valid examples according to this rule include:

```
R_CONTROLLER3
_A_ARRAY
_100_3_3_10
```

A.1.1.2 Unformatted rules (syntactic rules)

The syntactic rules build on the lexical rules and describe the structure of ST. You can write your ST program unformatted within the framework of these rules.

The unformatted property means:

- Formatting characters can be inserted anywhere.
- Block and line comments can be inserted.

The following example shows the syntactic rule for assigning a value in a statement.

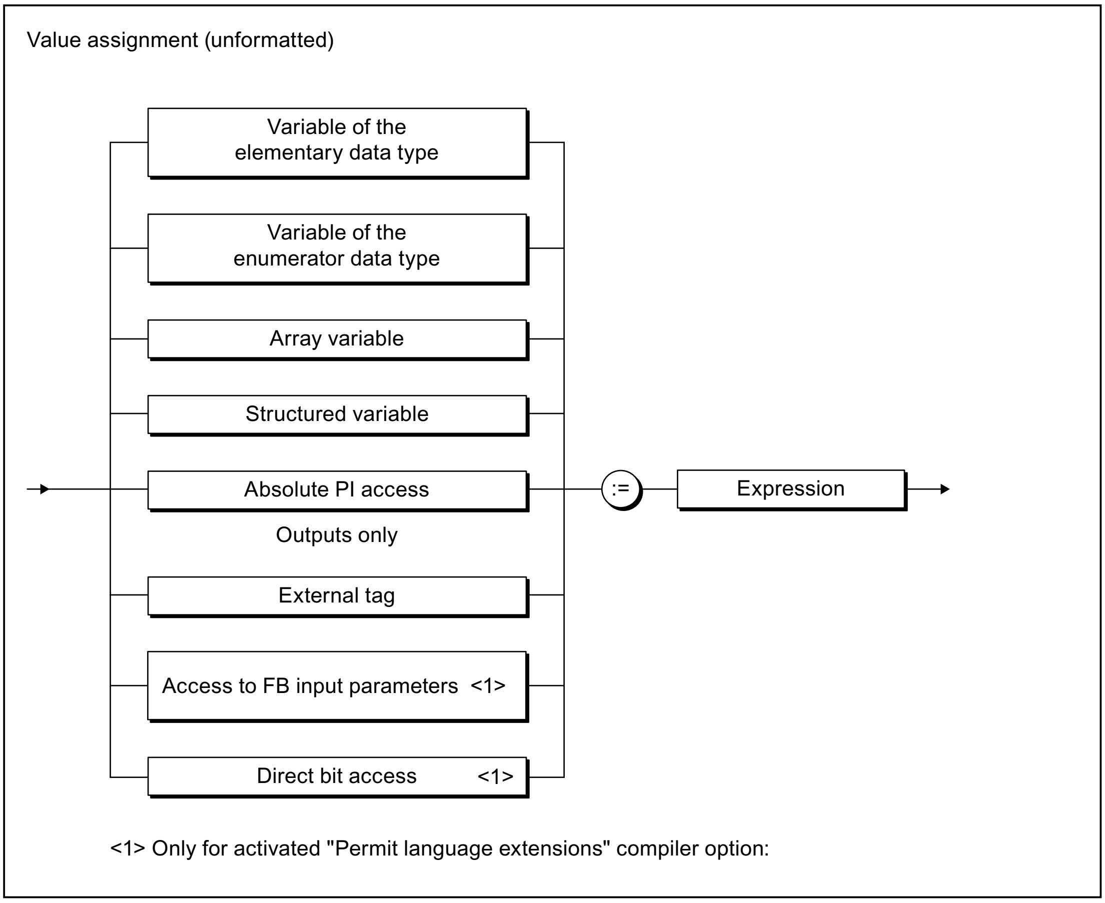

Value assignment (unformatted)
Variable of the elementary data type
Variable of the enumerator data type
Array variable
Structured variable
Absolute PI access
Outputs only
External tag
Access to FB input parameters <1>
Direct bit access <1>
:=
Expression
<1> Only for activated "Permit language extensions" compiler option:

Figure A-2 Example of a syntactic rule

Valid examples according to this rule include:

```
VARIABLE_1 := 100; SWITCH := FALSE;
//'This is a comment
VARIABLE_2:=3.2 +VARIABLE_1;
```

A.1.2 Basic elements (terminals)

A terminal is a basic element that is declared verbally and not by a further rule. It is represented in the syntax diagrams by an oval or circle.

A.1.2.1 Letters, digits and other characters

Letters and digits are the most commonly used characters. The *identifier*, for example, consists of a combination of letters, digits, and the underscore. The underscore is one of the special characters.

Table A-1 Letters and digits

Characters	Subgroup	Character set elements
Letter	Upper case	A .. Z
	Lower case	a .. z
Digit	Decimal digit	0 .. 9
Octal digit	Octal digit	0 .. 7
Hexadecimal digit	Hexadecimal digit	0 .. 9, A .. F, a .. f
Bit	Binary digit	0, 1

You can use the complete extended ASCII character set in comments. You can use all printable ASCII code characters starting from decimal equivalent 32 (blank).

For language commands, identifiers, constants, expressions and operators, you can use special characters, i.e. characters other than letters and digits, only according to certain rules.

A.1.2.2 Formatting characters and separators in the rules

Formatting characters and separators are used differently in formatted (lexical) and unformatted (syntactic) rules. Language description resources (Page 291) describes the differences between syntactic and lexical rules.

In the tables below, you will find the formatting characters and separators of the lexical and syntactic rules. You are also provided with a description and a list of all rules in which the formatting characters and separators are used as terminals (see Rules (Page 307)).

Table A-2 Formatting characters and separators in lexical rules

Characters	Description	Lexical rule
:	Separator between hours, minutes, and seconds	Time of day information
.	Separator for floating-point representation, time interval representation, absolute addressing	Floating-point representation, time-of-day information, decimal representation, access to local or global instance
_ Underscore	Separator for identifiers, separator for numerical values in constants	Identifiers, decimal digit string, binary digit string, octal digit string, hexadecimal digit string, sequence representation
%	Prefix for direct identifier on CPU memory access	Simple memory access
//	Comment	Line comment
(**)	Comment	Block comment

Table A-3 Formatting characters and separators in syntactic rules

Characters	Description	Syntactic rule
:	Separator for type information	Function, variable declaration, component declaration, CASE statement, instance declaration
;	Ends a declaration or statement	Constant block, statement, variable declaration, instance declaration, component declaration, statement section
,	Separator for lists	Variable declaration, array initialization list, instance declaration, ARRAY data type specification, FB parameter, FC parameter, value list
..	Range information	Array data type specification, value list
.	Structure access	Structured variable
( )	Initialization list for arrays, parentheses in expressions, function and function block calls	Array initialization list, expression, simple multiplication, operand, exponent, FB call, function call
[ ]	Array declaration, structured variable section of array	Array data type specification

See also

Language description resources (Page 71)

A.1.2.3 Formatting characters and separators for constants

Below, you will find all formatting characters and separators for constants with information on the lexical rule in which they are used.

Table A-4 Formatting characters and separators for constants

Characters	Code for	Lexical rule
2#	Integer constant	Binary digit string
8#	Integer constant	Octal digit string
16#	Integer constant	Hexadecimal digit string
E	Separator for floating-point constants	Exponent
E	Separator for floating-point constants	Exponent
D#	Time information	Date
DATE#	Time information	Date
DATE_AND_TIME#	Time information	Date and time
DT#	Time information	Date and time
T#	Time information	Duration
TIME#	Time information	Duration
TIME_OF_DAY#	Time information	Time of day
TOD#	Time information	Time of day
d	Separator for time interval (day)	Days (rule: Sequence representation)
h	Separator for time interval (hours)	Hours (rule: Sequence representation)
m	Separator for time interval (minutes)	Minutes (rule: Sequence representation)
ms	Separator for time interval (milliseconds)	Milliseconds (rule: Sequence representation)
s	Separator for time interval (seconds)	Seconds (rule: Sequence representation)

A.1.2.4 Predefined identifiers for process image access

Below is a list of all predefined variables in ST that you can use to access CPU memory areas (absolute identifiers). Note that you can read and write outputs but you can only read inputs.

Table A-5 Absolute identifier

Identifier	Description	Lexical rule
%In.x or %IXn.x	CPU input range with byte and bit address	Absolute PI access
%IBn	CPU input range with byte address	Absolute PI access
%IWn	CPU input range with word address	Absolute PI access
%IDn	CPU input range with double word address	Absolute PI access
%Qn.x or %QXn.x	CPU output range with byte and bit address	Absolute PI access
%QBn	CPU output range with byte address	Absolute PI access
%QWn	CPU output range with word address	Absolute PI access
%QDn	CPU output range with double word address	Absolute PI access

A.1.2.5 Identifiers of the Taskstartinfo

The following identifiers are defined for the Taskstartinfo:

Table A-6 Identifiers of the Taskstartinfo

Identifier	Data type	Description
TSI#alarmNumber	DINT	Scan for alarm number
TSI#commandId.high	UDINT	Scan for commandId (most significant word)
TSI#commandId.low	UDINT	Scan for commandId (least significant word)
TSI#currentTaskId	StructTaskId	Scan for TaskId of current task
TSI#cycleTime	TIME	Scan for configured cycle time of current task
TSI#details	DWORD	Scan for detailed information
TSI#executionFaultType	UDINT	Scan for type of execution error
TSI#interruptId	UDINT	Scan for triggering event
TSI#logBaseAdrIn	DINT	Scan for logical base address
TSI#logBaseAdrOut	DINT	Scan for logical base address
TSI#logDiagAddr	DINT	Scan for logical diagnostic address
TSI#shutDownInitiator	UDINT	Scan for cause of transition to STOP
TSI#startTime	DT	Scan for start time
TSI#taskId	StructTaskId	Scan for TaskId of triggering task
TSI#toInst	ANYOBJECT	Scan for TO instance

A.1.2.6 Operators

Below is a list of all ST operators and the syntactic rules in which they are used.

Table A-7 ST operators

Operator	Description	Rule
:=	Assignment operator (also for initialization values)	Value assignment, input assignment, in/out assignment, variable declaration, constant declaration, user-defined data types, component declaration
+, –	Arithmetic operators: Unary operators, sign	Expression, exponent
+, –, *, / MOD	Basic arithmetic operators	Expression, basic arithmetic operator
**	Arithmetic operators: Exponent operator	Expression
NOT	Logic operators: Negation	Expression, operand
AND, &, OR, XOR	Basic logic operator	Basic logic operator
<, >, <=, >=, =, <>	Relational operator	Relational operator
=>	Assignment operator	Output assignment

A.1.2.7 Reserved words

Below is an alphabetical list of keywords, predefined identifiers, and standard functions of the basic ST system. You are also provided with a description and the syntactic rule from *rules* in which they are used as terminals. An exception is standard functions, which are included only implicitly in the syntactic rule for *function calls* as the standard function name.

Note

Variables must not be assigned the names of keywords or predefined identifiers. For more information about identifiers, see *Identifiers in ST*. You will find an overview of the identifiers reserved for technology objects and other reserved identifiers in *Reserved identifiers*.

Table A-8 ST keywords and predefined identifiers in the basic ST system

Keyword/identifier	Description	Rule
ABS	Standard numeric function	Function call
ACOS	Standard numeric function	Function call
AND	Logic operator	Basic logic operator
ANYOBJECT	General data type for technology objects	TO data type
ANYOBJECT_TO_OBJECT	Standard function for type conversion (technology objects)	Function call
ANYTYPE_TO_BIGBYTEARRAY	Standard function (marshalling)	Function call
ANYTYPE_TO_LITTLEBYTEARRAY	Standard function (marshalling)	Function call
ARRAY	Introduces the specification of an array and is followed by the index list between **[** and **]**	Array data type specification
AS	Introduces a namespace	–
ASIN	Standard numeric function	Function call
AT	Reserved identifier	–
ATAN	Standard numeric function	Function call
BIGBYTEARRAY_TOANYTYPE	Standard function (marshalling)	Function call
BOOL	Elementary data type for binary data	Bit data type
BOOL_TO_BYTE	Standard function for type conversion	Function call
BOOL_TO_DWORD	Standard function for type conversion	Function call
BOOL_TO_WORD	Standard function for type conversion	Function call
BOOL_VALUE_TO_DINT	Standard function for type conversion	Function call
BOOL_VALUE_TO_INT	Standard function for type conversion	Function call
BOOL_VALUE_TO_LREAL	Standard function for type conversion	Function call
BOOL_VALUE_TO_REAL	Standard function for type conversion	Function call
BOOL_VALUE_TO_SINT	Standard function for type conversion	Function call
BOOL_VALUE_TO_UDINT	Standard function for type conversion	Function call
BOOL_VALUE_TO_UINT	Standard function for type conversion	Function call
BOOL_VALUE_TO_USINT	Standard function for type conversion	Function call
BY	Introduces the increment	FOR statement

Keyword/identifier	Description	Rule
BYTE	Elementary data type	Bit data type
BYTE_TO_BOOL	Standard function for type conversion	Function call
BYTE_TO_DINT	Standard function for type conversion	Function call
BYTE_TO_DWORD	Standard function for type conversion	Function call
BYTE_TO_INT	Standard function for type conversion	Function call
BYTE_TO_SINT	Standard function for type conversion	Function call
BYTE_TO_UDINT	Standard function for type conversion	Function call
BYTE_TO_UINT	Standard function for type conversion	Function call
BYTE_TO_USINT	Standard function for type conversion	Function call
BYTE_TO_WORD	Standard function for type conversion	Function call
BYTE_VALUE_TO_LREAL	Standard function for type conversion	Function call
BYTE_VALUE_TO_REAL	Standard function for type conversion	Function call
CASE	Introduces a control statement for selection	CASE statement
CONCAT	Standard function for string editing	Function call
CONCAT_DATE_TOD	Standard function for type conversion	Function call
CONSTANT	Introduces a constant definition	Constant block
COS	Standard numeric function	Function call
CTD	Down counter	Function block call
CTD_DINT	Down counter	Function block call
CTD_UDINT	Down counter	Function block call
CTU	Up counter	Function block call
CTU_DINT	Up counter	Function block call
CTU_UDINT	Up counter	Function block call
CTUD	Up/down counter	Function block call
CTUD_DINT	Up/down counter	Function block call
CTUD_UDINT	Up/down counter	Function block call
DATE	Elementary data type for date	Time type
DATE_AND_TIME	Elementary data type for date and time	DATE_AND_TIME
DATE_AND_TIME_TO_DATE	Standard function for type conversion	Function call
DATE_AND_TIME_TO_TIME_OF_DAY	Standard function for type conversion	Function call
DELETE	Standard function for string editing	Function call
DINT	Elementary data type for double precision integer with value range -2**31 to 2**31-1	Numeric data type
DINT_TO_BYTE	Standard function for type conversion	Function call
DINT_TO_DWORD	Standard function for type conversion	Function call
DINT_TO_INT	Standard function for type conversion	Function call
DINT_TO_LREAL	Standard function for type conversion	Function call
DINT_TO_REAL	Standard function for type conversion	Function call
DINT_TO_SINT	Standard function for type conversion	Function call
DINT_TO_STRING	Standard function for type conversion	Function call
DINT_TO_UDINT	Standard function for type conversion	Function call

Keyword/identifier	Description	Rule
DINT_TO_UINT	Standard function for type conversion	Function call
DINT_TO_USINT	Standard function for type conversion	Function call
DINT_TO_WORD	Standard function for type conversion	Function call
DINT_VALUE_TO_BOOL	Standard function for type conversion	Function call
DO	Introduces the statement section for FOR statement or WHILE statement	FOR statement, WHILE statement
DT	Shorthand notation for DATE_AND_TIME	DATE_AND_TIME
DT_TO_DATE	Standard function for type conversion	Function call
DT_TO_TOD	Standard function for type conversion	Function call
DWORD	Elementary data type for double word	Bit data type
DWORD_TO_BOOL	Standard function for type conversion	Function call
DWORD_TO_BYTE	Standard function for type conversion	Function call
DWORD_TO_DINT	Standard function for type conversion	Function call
DWORD_TO_INT	Standard function for type conversion	Function call
DWORD_TO_REAL	Standard function for type conversion	Function call
DWORD_TO_SINT	Standard function for type conversion	Function call
DWORD_TO_UDINT	Standard function for type conversion	Function call
DWORD_TO_UINT	Standard function for type conversion	Function call
DWORD_TO_USINT	Standard function for type conversion	Function call
DWORD_TO_WORD	Standard function for type conversion	Function call
DWORD_VALUE_TO_LREAL	Standard function for type conversion	Function call
DWORD_VALUE_TO_REAL	Standard function for type conversion	Function call
ELSE	Introduces the clause to be executed if no condition true	IF statement, CASE statement
ELSIF	Introduces alternative condition	IF statement
END_CASE	Ends the CASE statement	CASE statement
END_EXPRESSION	Ends the EXPRESSION statement	Function
END_FOR	Ends the FOR statement	FOR statement
END_FUNCTION	Ends the function	Function
END_FUNCTION_BLOCK	Ends the function block	Function block
END_IF	Ends the IF statement	IF statement
END_IMPLEMENTATION	Ends the implementation section	Implementation section
END_INTERFACE	Ends the interface section	Interface section
END_LABEL	Ends the LABEL statement	–
END_PROGRAM	Ends the program section	Program section
END_REPEAT	Ends the REPEAT statement	REPEAT statement
END_STRUCT	Ends the specification of a structure	STRUCT data type specification
END_TYPE	Ends the UDT	User-defined data type
END_VAR	Ends a declaration block	Temporary variable block, static variable block, parameter block, constant block
END_WAITFORCONDITION	Ends the control statement for a task waiting for a programmable event	WAITFORCONDITION statement

Keyword/identifier	Description	Rule
END_WHILE	Ends the WHILE statement	WHILE statement
ENUM_TO_DINT	Standard function for type conversion	Function call
EXIT	Direct exit from loop execution	EXIT
EXP	Standard numeric function	Function call
EXPD	Standard numeric function	Function call
EXPRESSION	Programmable event for waiting task	Function
EXPT	Standard numeric function	Function call
F_TRIG	Detects falling edge	Function block call
FALSE	Predefined Boolean constant: Logical condition false, value equal to 0	–
FIND	Standard function for string editing	Function call
FOR	Introduces control statement for loop execution	FOR statement
FUNCTION	Introduces the function	Function
FUNCTION_BLOCK	Introduces the function block	Function block
GOTO	Jump	–
IF	Introduces a control statement for selection	IF statement
IMPLEMENTATION	Introduces the IMPLEMENTATION section	IMPLEMENTATION section
INSERT	Standard function for string editing	Function call
INT	Elementary data type for single precision integer with value range -2**15 to 2**15-1	Numeric data type
INT_TO_BYTE	Standard function for type conversion	Function call
INT_TO_DINT	Standard function for type conversion	Function call
INT_TO_DWORD	Standard function for type conversion	Function call
INT_TO_LREAL	Standard function for type conversion	Function call
INT_TO_REAL	Standard function for type conversion	Function call
INT_TO_SINT	Standard function for type conversion	Function call
INT_TO_TIME	Standard function for type conversion	Function call
INT_TO_UDINT	Standard function for type conversion	Function call
INT_TO_UINT	Standard function for type conversion	Function call
INT_TO_USINT	Standard function for type conversion	Function call
INT_TO_WORD	Standard function for type conversion	Function call
INT_VALUE_TO_BOOL	Standard function for type conversion	Function call
INTERFACE	Introduces the interface section	Interface section
LABEL	Definition of jump labels	–
LEFT	Standard function for string editing	Function call
LEN	Standard function for string editing	Function call
LIMIT	Standard function for selection	Function call
LITTLEBYTEARRAY_TOANYTYPE	Standard function (marshalling)	Function call
LN	Standard numeric function	Function call
LOG	Standard numeric function	Function call

Keyword/identifier	Description	Rule
LREAL	Elementary data type for 64-bit double-precision floating-point number (long real)	Numeric data type
LREAL_TO_DINT	Standard function for type conversion	Function call
LREAL_TO_INT	Standard function for type conversion	Function call
LREAL_TO_REAL	Standard function for type conversion	Function call
LREAL_TO_SINT	Standard function for type conversion	Function call
LREAL_TO_STRING	Standard function for type conversion	Function call
LREAL_TO_UDINT	Standard function for type conversion	Function call
LREAL_TO_UINT	Standard function for type conversion	Function call
LREAL_TO_USINT	Standard function for type conversion	Function call
LREAL_VALUE_TO_BOOL	Standard function for type conversion	Function call
LREAL_VALUE_TO_BYTE	Standard function for type conversion	Function call
LREAL_VALUE_TO_DWORD	Standard function for type conversion	Function call
LREAL_VALUE_TO_WORD	Standard function for type conversion	Function call
MAX	Standard function for selection	Function call
MID	Standard function for string editing	Function call
MIN	Standard function for selection	Function call
MOD	Arithmetic operator for division remainder	Basic arithmetic operator, simple multiplication
MUX	Standard function for selection	Function call
NOT	Logic operator, belongs to the unary operators	Expression, operand
OF	Introduces data type specification	Array data type specification, CASE statement
OR	Logic operator	Basic logic operator
PROGRAM	Introduces the PROGRAM section	Program
R_TRIG	Detects rising edge	Function block call
REAL	Elementary data type for 32-bit single precision floating-point number (real)	Numeric data type
REAL_TO_DINT	Standard function for type conversion	Function call
REAL_TO_DWORD	Standard function for type conversion	Function call
REAL_TO_INT	Standard function for type conversion	Function call
REAL_TO_LREAL	Standard function for type conversion	Function call
REAL_TO_SINT	Standard function for type conversion	Function call
REAL_TO_STRING	Standard function for type conversion	Function call
REAL_TO_TIME	Standard function for type conversion	Function call
REAL_TO_UDINT	Standard function for type conversion	Function call
REAL_TO_UINT	Standard function for type conversion	Function call
REAL_TO_USINT	Standard function for type conversion	Function call
REAL_VALUE_TO_BOOL	Standard function for type conversion	Function call
REAL_VALUE_TO_BYTE	Standard function for type conversion	Function call
REAL_VALUE_TO_DWORD	Standard function for type conversion	Function call
REAL_VALUE_TO_WORD	Standard function for type conversion	Function call

Keyword/identifier	Description	Rule
REPEAT	Introduces control statement for loop execution	REPEAT statement
REPLACE	Standard function for string editing	Function call
RETAIN	Declaration of buffered variables	Retentive variable block
RETURN	Control statement for returning from subroutine	RETURN statement
RIGHT	Standard function for string editing	Function call
ROL	Bit string standard functions	Function call
ROR	Bit string standard functions	Function call
RS	Bistable function block (priority reset)	Function block call
RTC	Real-time clock	Function block call
SEL	Standard function for selection	Function call
SHL	Bit string standard functions	Function call
SHR	Bit string standard functions	Function call
SIN	Standard numeric function	Function call
SINT	Elementary data type for short integer with value range -128 to 127	Numeric data type
SINT_TO_BYTE	Standard function for type conversion	Function call
SINT_TO_DINT	Standard function for type conversion	Function call
SINT_TO_DWORD	Standard function for type conversion	Function call
SINT_TO_INT	Standard function for type conversion	Function call
SINT_TO_LREAL	Standard function for type conversion	Function call
SINT_TO_REAL	Standard function for type conversion	Function call
SINT_TO_UDINT	Standard function for type conversion	Function call
SINT_TO_UINT	Standard function for type conversion	Function call
SINT_TO_USINT	Standard function for type conversion	Function call
SINT_TO_WORD	Standard function for type conversion	Function call
SINT_VALUE_TO_BOOL	Standard function for type conversion	Function call
SQRT	Standard numeric function	Function call
SR	Bistable function block (priority set)	Function block call
STRING	Elementary data type for character strings	String data type
STRING_TO_DINT	Standard function for type conversion	Function call
STRING_TO_LREAL	Standard function for type conversion	Function call
STRING_TO_REAL	Standard function for type conversion	Function call
STRING_TO_UDINT	Standard function for type conversion	Function call
STRUCT	Introduces the specification of a structure and is followed by a list of components	STRUCT data type specification
StructAlarmId	Data type for AlarmId	–
StructAlarmId_TO_DINT	Standard function for type conversion	Function call
StructTaskId	Data type for TaskId	–

Keyword/identifier	Description	Rule
TAN	Standard numeric function	Function call
THEN	Introduces subsequent actions if condition true	IF statement
TIME	Elementary data type for time information	Time type
TIME_OF_DAY	Elementary data type for time of day	Time type
TIME_TO_INT	Standard function for type conversion	Function call
TIME_TO_REAL	Standard function for type conversion	Function call
TO	Introduces end value	FOR statement
TOD	Shorthand notation for TIME_OF_DAY	Time type
TOF	OFF delay	Function block call
TON	ON delay	Function block call
TP	Pulse	Function block call
TRUE	Predefined Boolean constant: Logical condition true, value not equal to 0	–
TRUNC	Standard numeric function	Function call
TYPE	Introduces UDT	User-defined data type
UDINT	Elementary data type for unsigned double precision integer with value range 0 to 2**32-1	Numeric data type
UDINT_TO_BYTE	Standard function for type conversion	Function call
UDINT_TO_DINT	Standard function for type conversion	Function call
UDINT_TO_DWORD	Standard function for type conversion	Function call
UDINT_TO_INT	Standard function for type conversion	Function call
UDINT_TO_LREAL	Standard function for type conversion	Function call
UDINT_TO_REAL	Standard function for type conversion	Function call
UDINT_TO_SINT	Standard function for type conversion	Function call
UDINT_TO_STRING	Standard function for type conversion	Function call
UDINT_TO_UINT	Standard function for type conversion	Function call
UDINT_TO_USINT	Standard function for type conversion	Function call
UDINT_TO_WORD	Standard function for type conversion	Function call
UDINT_VALUE_TO_BOOL	Standard function for type conversion	Function call
UINT	Elementary data type for unsigned single precision integer with value range 0 to 2**16-1	Numeric data type
UINT_TO_BYTE	Standard function for type conversion	Function call
UINT_TO_DINT	Standard function for type conversion	Function call
UINT_TO_DWORD	Standard function for type conversion	Function call
UINT_TO_INT	Standard function for type conversion	Function call
UINT_TO_LREAL	Standard function for type conversion	Function call
UINT_TO_REAL	Standard function for type conversion	Function call
UINT_TO_SINT	Standard function for type conversion	Function call
UINT_TO_UDINT	Standard function for type conversion	Function call
UINT_TO_USINT	Standard function for type conversion	Function call

Keyword/identifier	Description	Rule
UINT_TO_WORD	Standard function for type conversion	Function call
UINT_VALUE_TO_BOOL	Standard function for type conversion	Function call
UNIT	Introduces the UNIT section	Unit section
UNTIL	Introduces exit condition for REPEAT statement	REPEAT statement
USELIB	Introduces the library name	–
USEPACKAGE	Introduces the package name	–
USES	Introduces a reference to other units	–
USINT	Elementary data type for unsigned short integer with value range 0 to 255	Numeric data type
USINT_TO_BYTE	Standard function for type conversion	Function call
USINT_TO_DINT	Standard function for type conversion	Function call
USINT_TO_DWORD	Standard function for type conversion	Function call
USINT_TO_INT	Standard function for type conversion	Function call
USINT_TO_LREAL	Standard function for type conversion	Function call
USINT_TO_REAL	Standard function for type conversion	Function call
USINT_TO_SINT	Standard function for type conversion	Function call
USINT_TO_UDINT	Standard function for type conversion	Function call
USINT_TO_UINT	Standard function for type conversion	Function call
USINT_TO_WORD	Standard function for type conversion	Function call
USINT_VALUE_TO_BOOL	Standard function for type conversion	Function call
VAR	Introduces a declaration block for local variables	Static variable block
VAR_GLOBAL	Introduces a declaration block for unit variables (global variables)	Unit variables
VAR_IN_OUT	Introduces a declaration block	Parameter block
VAR_INPUT	Introduces a declaration block	Parameter block
VAR_OUTPUT	Introduces a declaration block	Parameter block
VAR_TEMP	Introduces a declaration block	Parameter block
VOID	No return value on function call	Function
WAITFORCONDITION	Introduces the control statement for a task waiting for a programmable event	WAITFORCONDITION statement
WHILE	Introduces control statement for loop execution	WHILE statement
WITH	Use in conjunction with WAITFORCONDITION	WAITFORCONDITION statement
WORD	Elementary data type for word	Bit data type
WORD_TO_BOOL	Standard function for type conversion	Function call
WORD_TO_BYTE	Standard function for type conversion	Function call
WORD_TO_DINT	Standard function for type conversion	Function call
WORD_TO_DWORD	Standard function for type conversion	Function call
WORD_TO_INT	Standard function for type conversion	Function call
WORD_TO_SINT	Standard function for type conversion	Function call
WORD_TO_UDINT	Standard function for type conversion	Function call

Keyword/identifier	Description	Rule
WORD_TO_UINT	Standard function for type conversion	Function call
WORD_TO_USINT	Standard function for type conversion	Function call
WORD_VALUE_TO_LREAL	Standard function for type conversion	Function call
WORD_VALUE_TO_REAL	Standard function for type conversion	Function call
XOR	Logic operator	Basic logic operator

A.1.3 Rules

The following syntax rules of the ST language are subdivided into rules with formatted notation (lexical rules) and unformatted notation (syntactic rules). *Language description resources* describes the differences between syntactic and lexical rules.

A.1.3.1 Identifiers

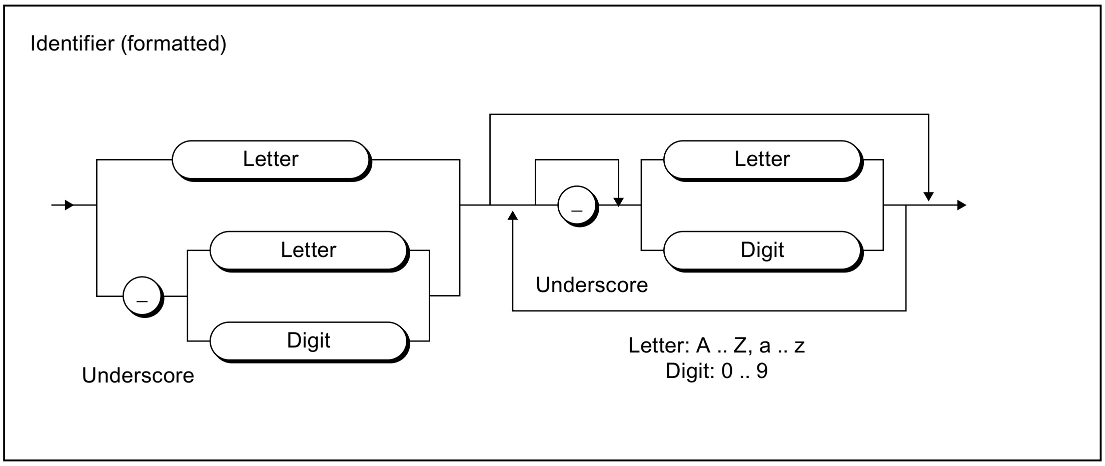


Figure A-3 Identifier

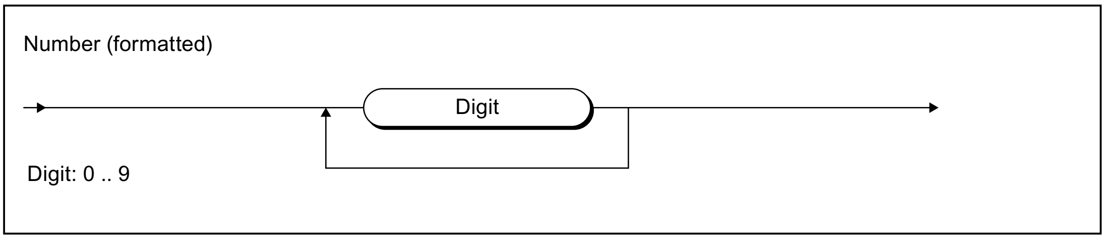


Figure A-4 Number

A.1.3.2 Notation for constants (literals)

Literals

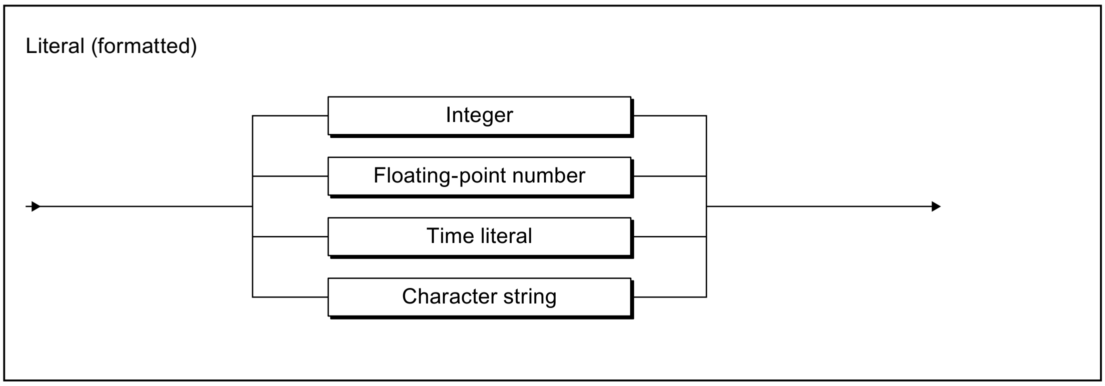


Figure A-5 Literal

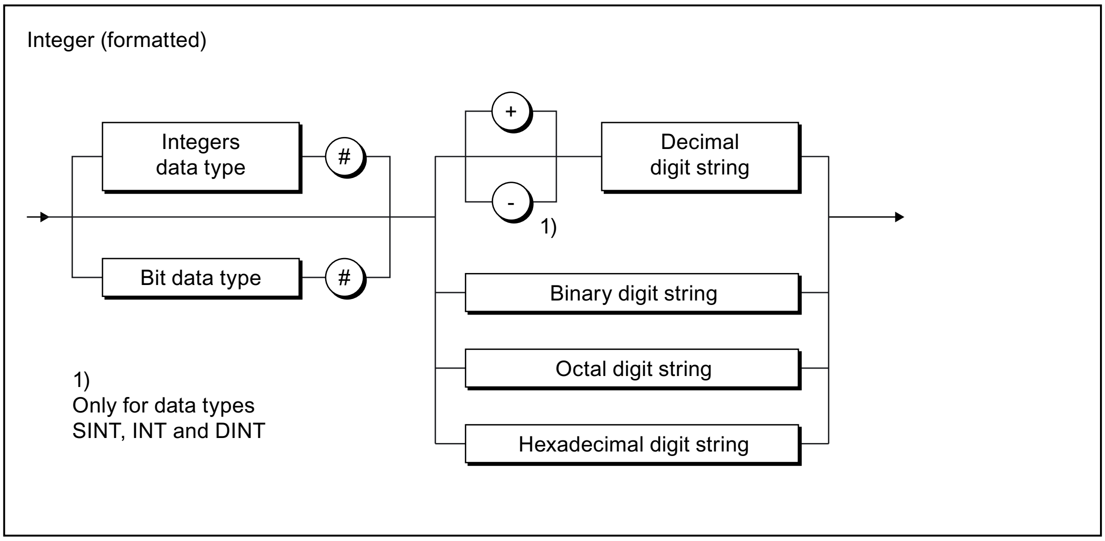


Figure A-6 Integer

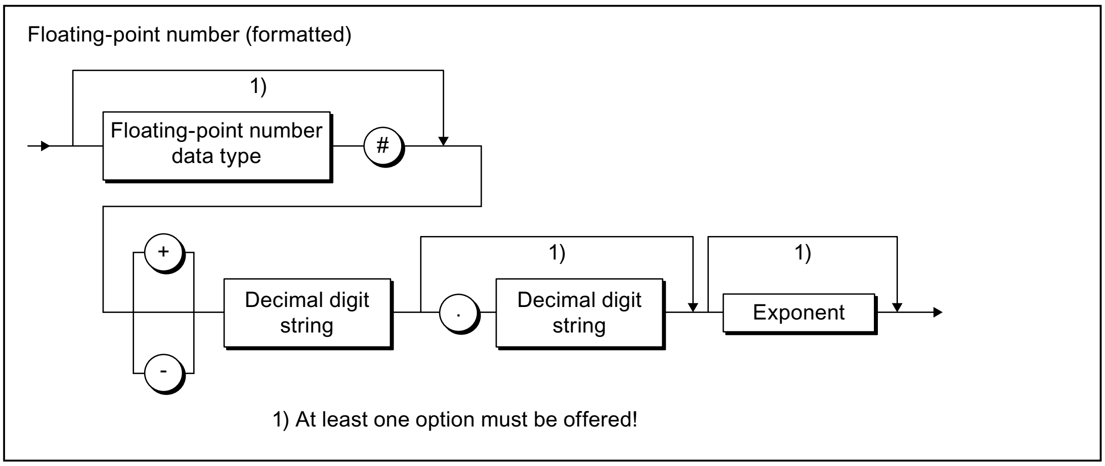

Floating-point number (formatted)
1)
Floating-point number data type
#
+
-
Decimal digit string
1)
.
Decimal digit string
1)
Exponent
1) At least one option must be offered!

Figure A-7 Floating-point number

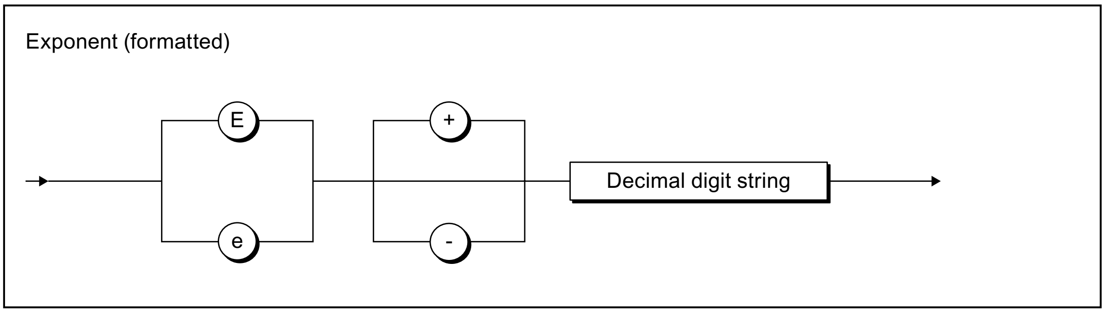


Figure A-8 Exponent

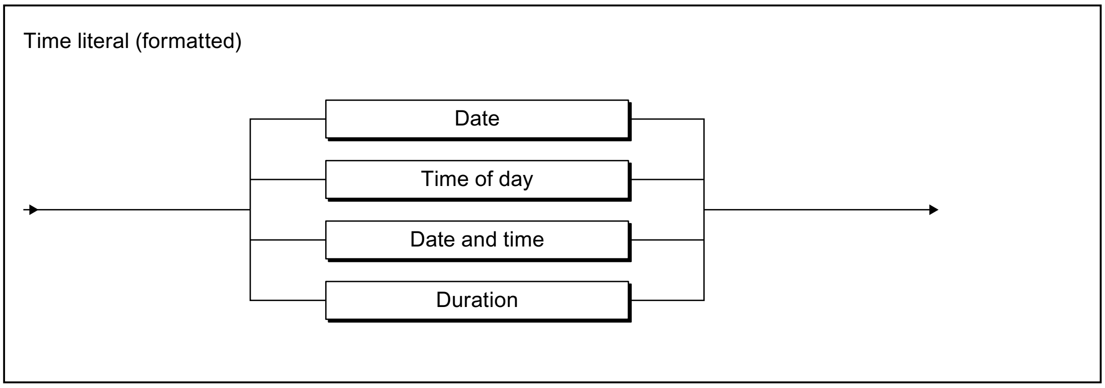


Figure A-9 Time literal

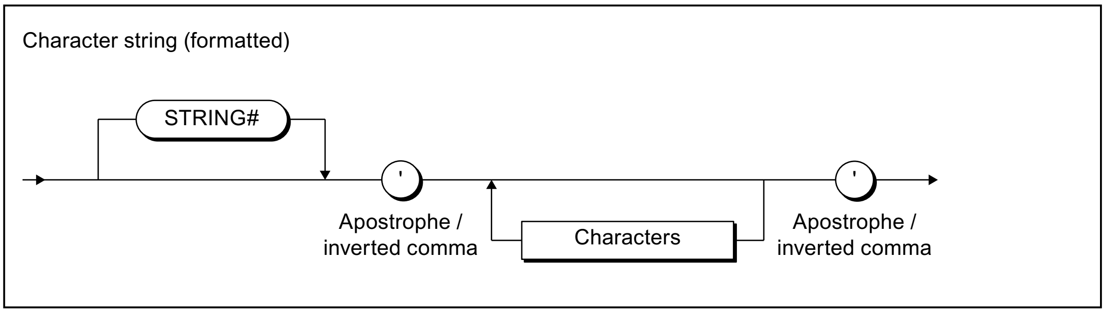
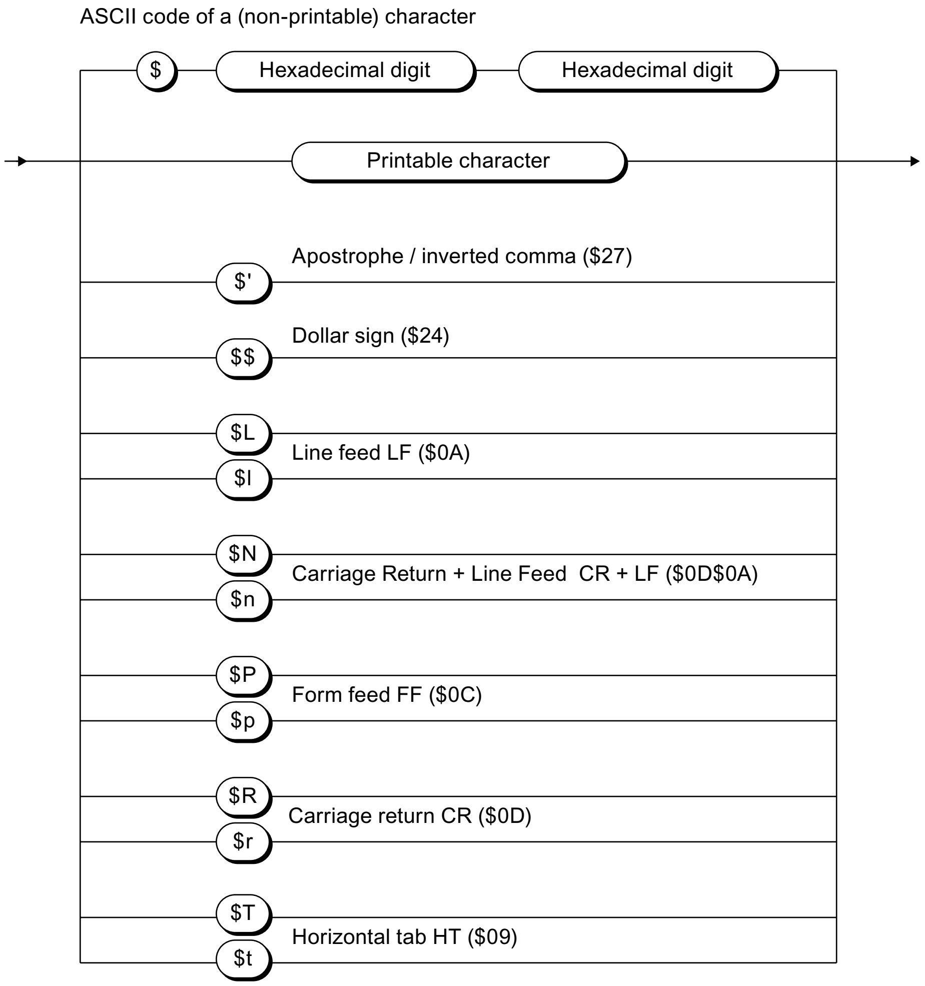

Character string (formatted)
STRING#
'
Apostrophe / inverted comma
Characters
'
Apostrophe / inverted comma

Figure A-10 Character string

Characters (formatted)



Printable character:

Any character from the extended ASCII character set (ASCII code $20 to $7E and $80 to $FF)

except: Dollar sign ($24) and
apostrophe / inverted comma ($27).

Hexadecimal digits: 0 .. 9, A .. F.

Figure A-11 Character

Digit string

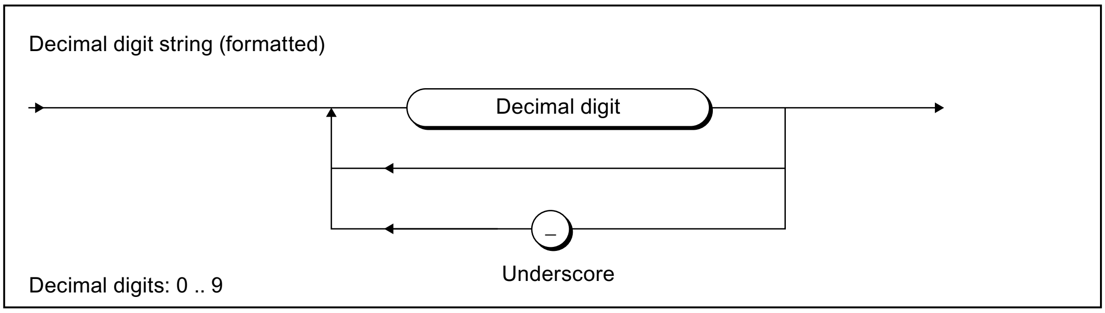


Figure A-12 Decimal digit string

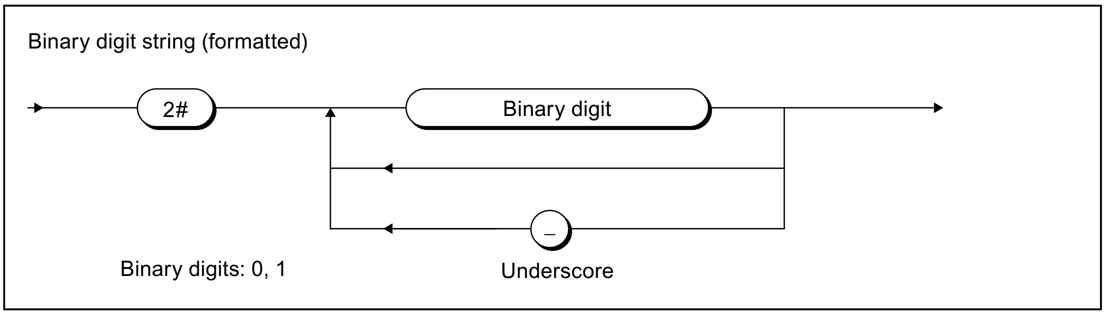


Figure A-13 Binary digit string

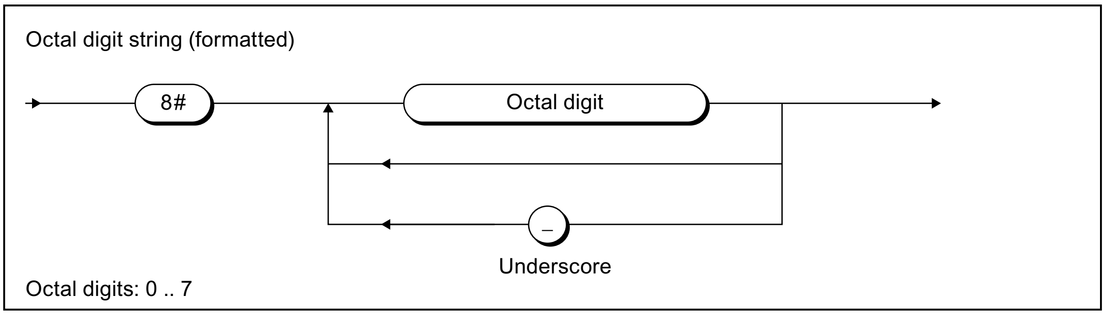


Figure A-14 Octal digit string

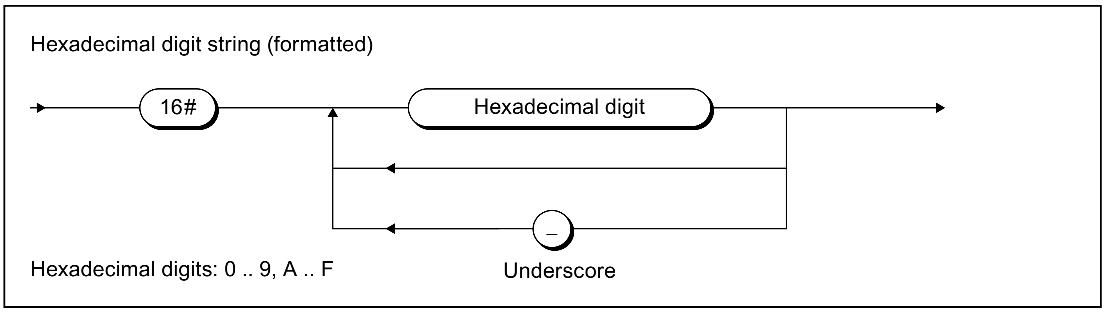

Hexadecimal digit string (formatted)
16#
Hexadecimal digit
_
Hexadecimal digits: 0 .. 9, A .. F
Underscore

Figure A-15 Hexadecimal digit string

Date and time

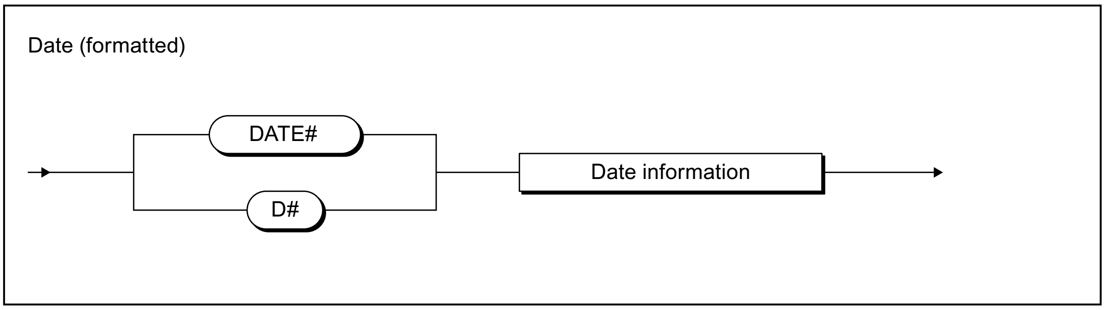
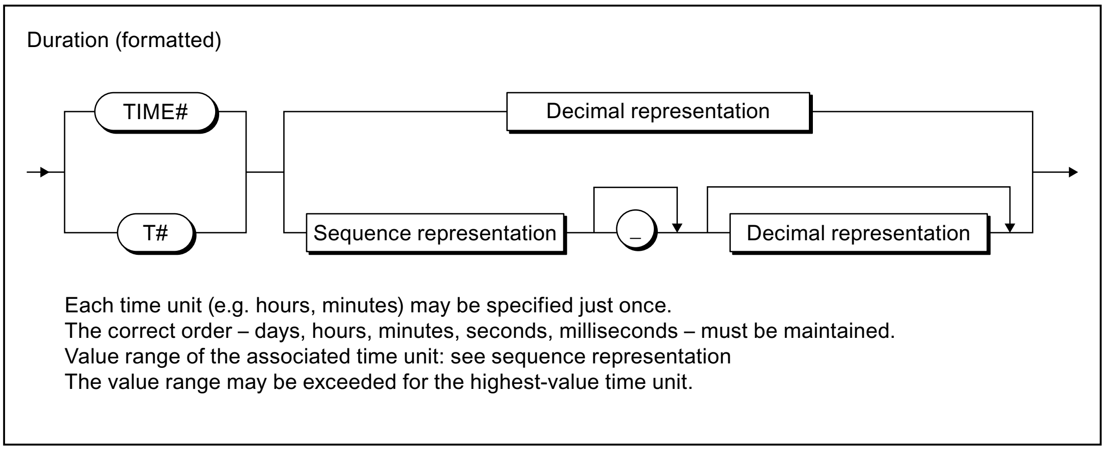


Figure A-16 Date



Each time unit (e.g. hours, minutes) may be specified just once.
The correct order – days, hours, minutes, seconds, milliseconds – must be maintained.
Value range of the associated time unit: see sequence representation
The value range may be exceeded for the highest-value time unit.

Figure A-17 Time

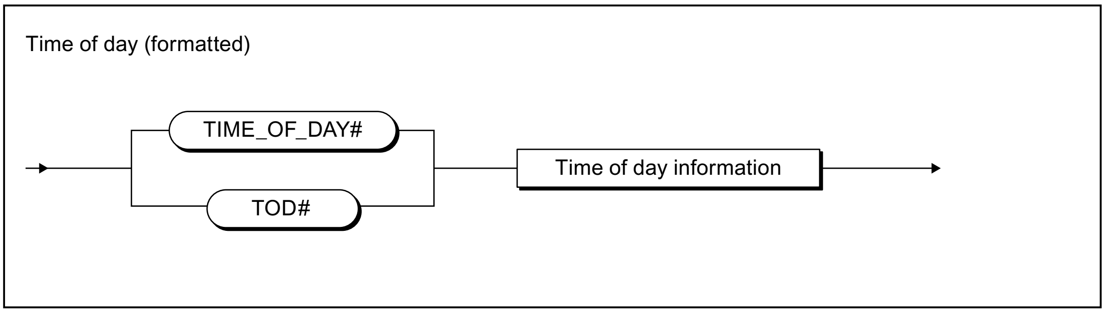


Figure A-18 Time

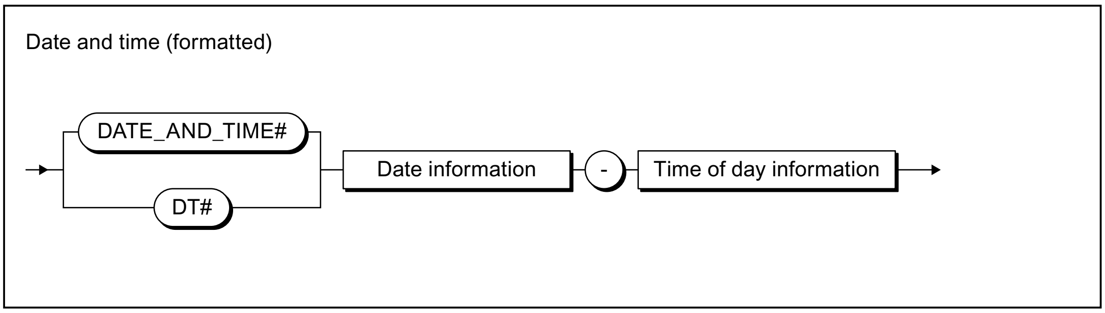


Figure A-19 Date and time

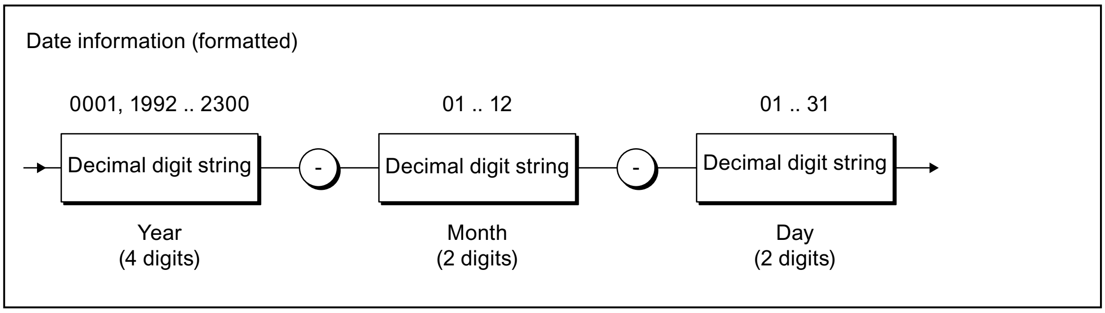


Figure A-20 Date information

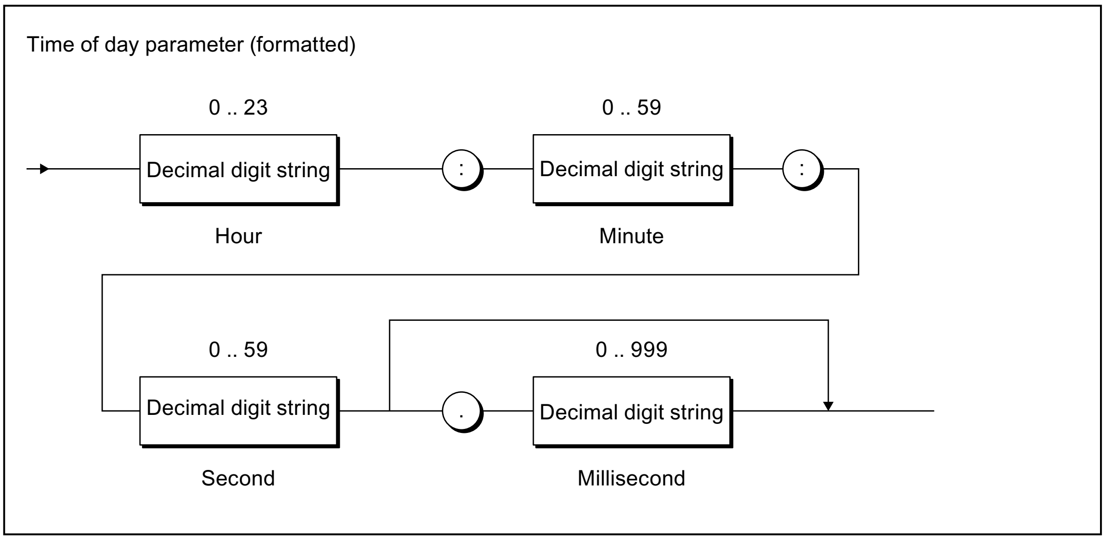


Figure A-21 Time of day information

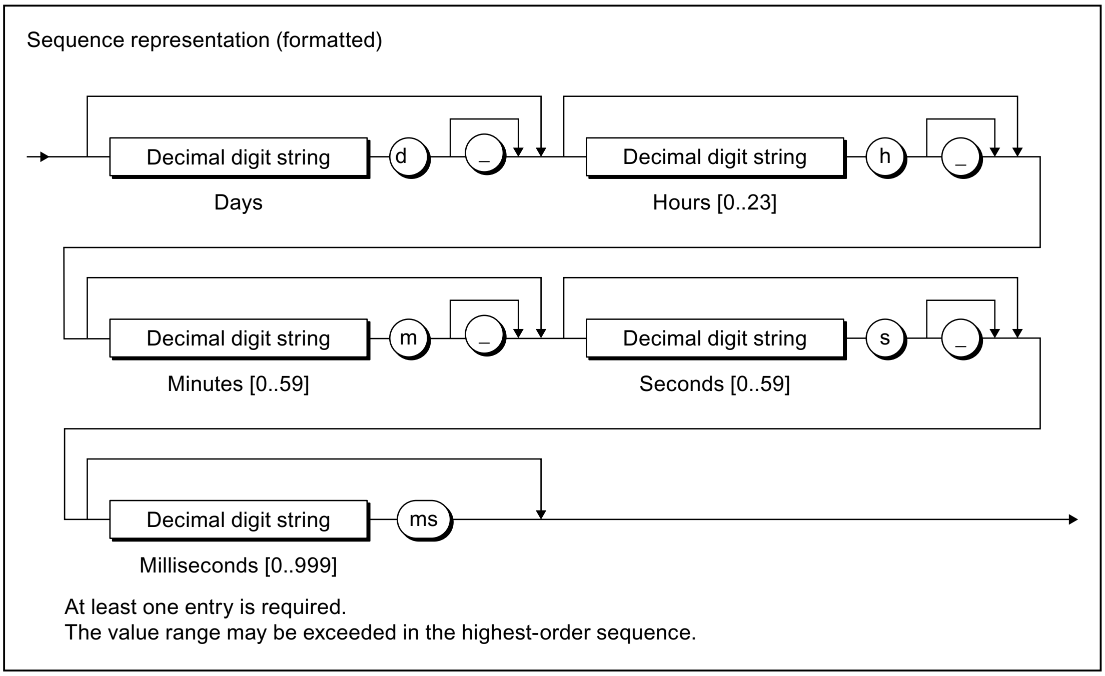


Figure A-22 Sequence representation

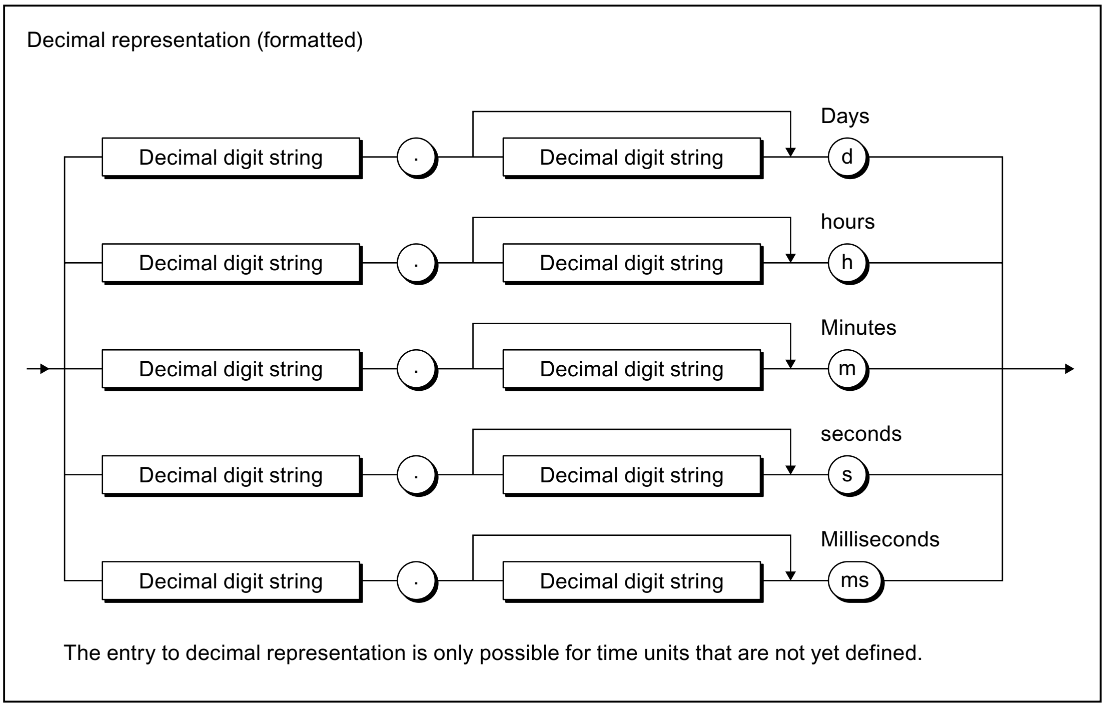

Decimal representation (formatted)
Decimal digit string
.
Decimal digit string
Days
d
hours
h
Minutes
m
seconds
s
Milliseconds
ms
The entry to decimal representation is only possible for time units that are not yet defined.

Figure A-23 Decimal representation

A.1.3.3 Comments

Note the following when inserting comments:

- Nesting of line comments is not allowed.
- Nesting of block comments is not allowed, but you can nest line comments in block comments.
- Comments are allowed at any position in the unformatted (syntactic) rules.
- Comments are not allowed in formatted (lexical) rules.

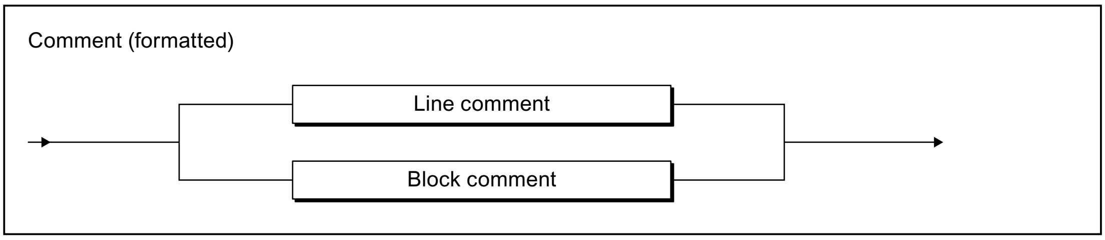


Figure A-24 Comments

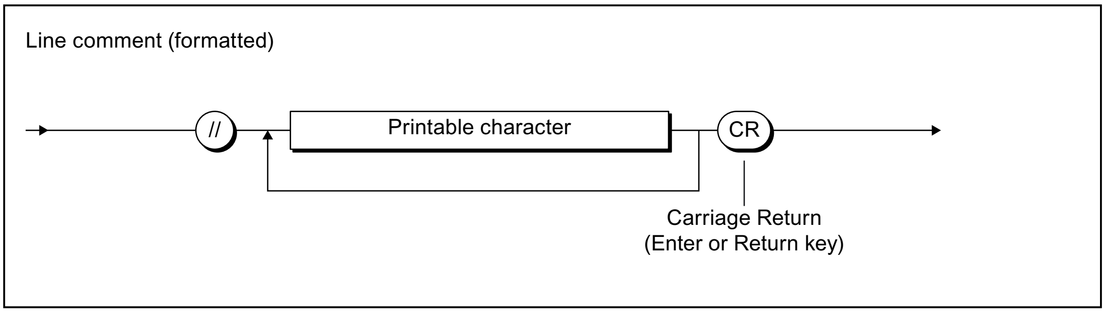


Figure A-25 Line comment

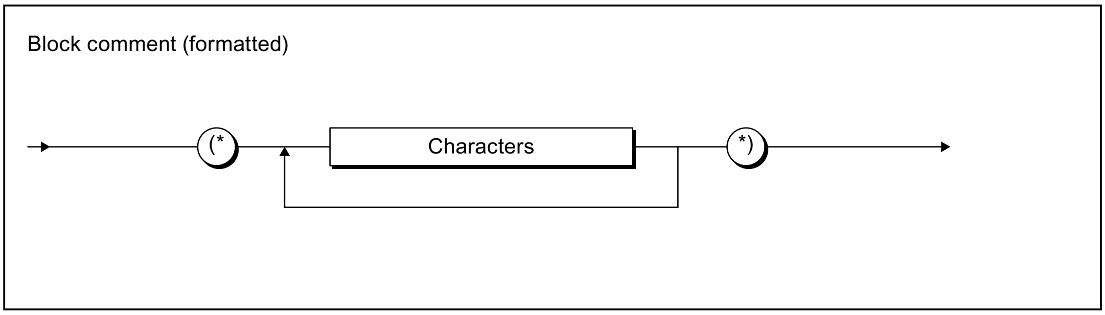
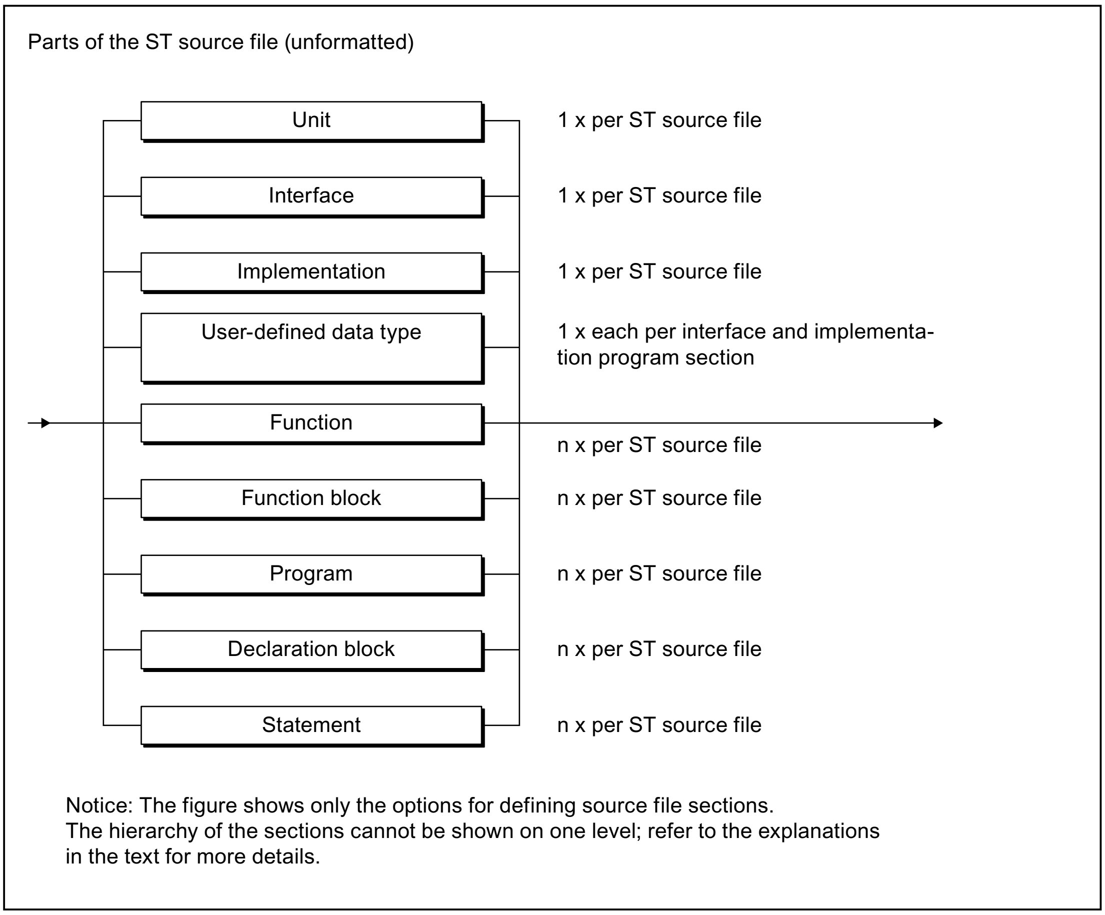


Figure A-26 Block comment

A.1.3.4 Sections of the ST source file



Notice: The figure shows only the options for defining source file sections.
The hierarchy of the sections cannot be shown on one level; refer to the explanations in the text for more details.

Sections of the ST source file

A.1.3.5 Structures of ST source files

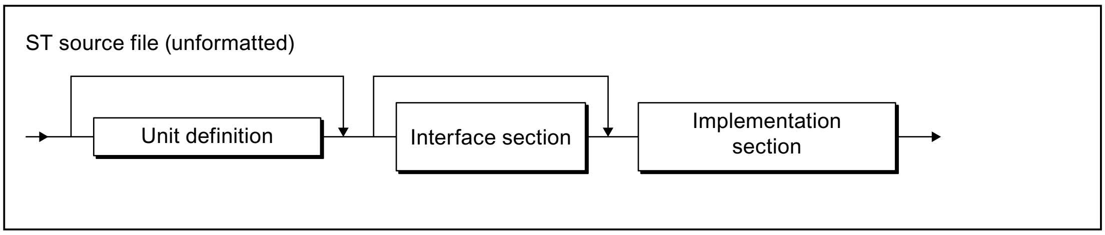


Figure A-27 ST source file

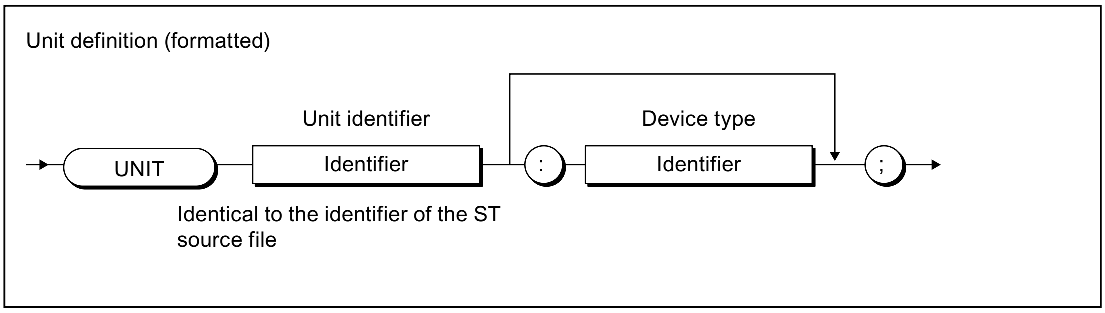
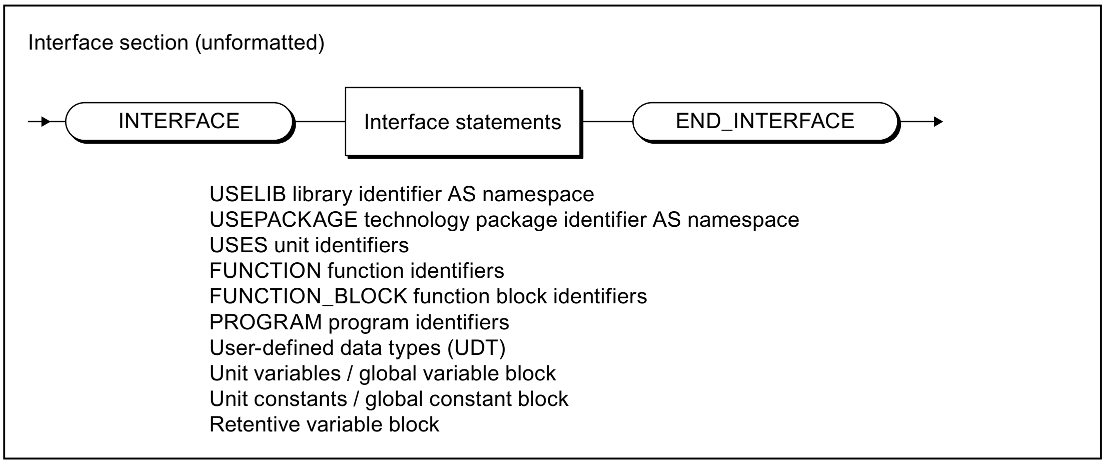

Unit definition (formatted)
Unit identifier
Device type
UNIT
Identifier
:
Identifier
;
Identical to the identifier of the ST source file

Figure A-28 Unit definition



USELIB library identifier AS namespace
USEPACKAGE technology package identifier AS namespace
USES unit identifiers
FUNCTION function identifiers
FUNCTION_BLOCK function block identifiers
PROGRAM program identifiers
User-defined data types (UDT)
Unit variables / global variable block
Unit constants / global constant block
Retentive variable block

Figure A-29 Interface section

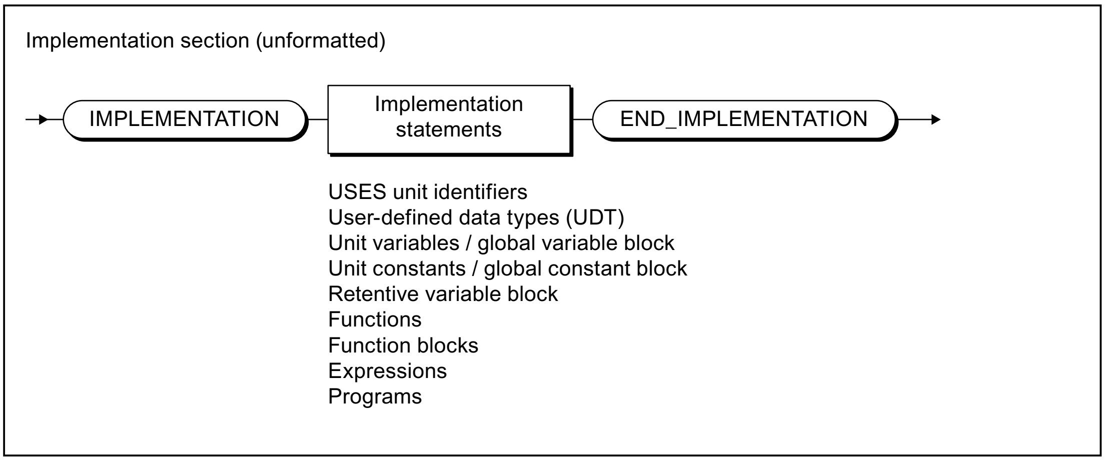

Implementation section (unformatted)
IMPLEMENTATION
Implementation statements
END_IMPLEMENTATION
USES unit identifiers
User-defined data types (UDT)
Unit variables / global variable block
Unit constants / global constant block
Retentive variable block
Functions
Function blocks
Expressions
Programs

Figure A-30 Implementation section

A.1.3.6 Program organization units (POU)

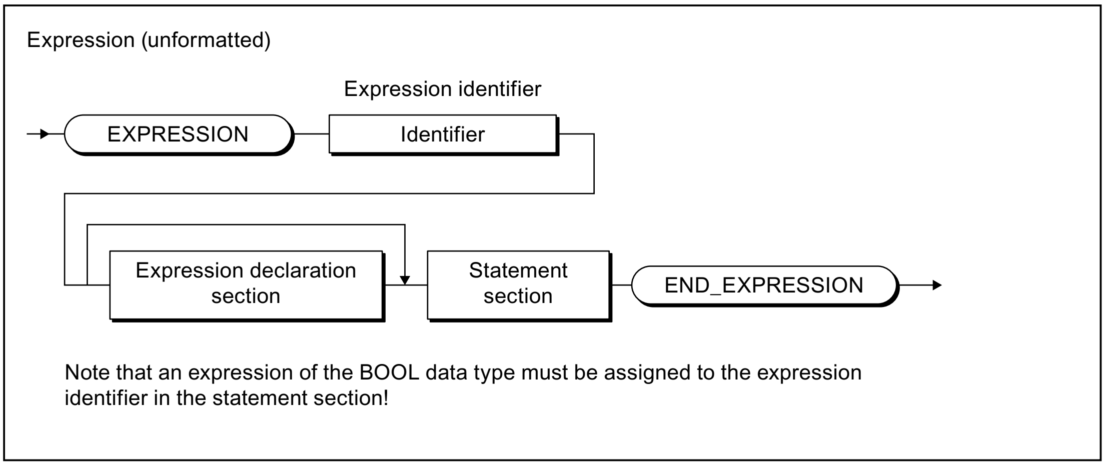

Expression (unformatted)
Expression identifier
EXPRESSION
Identifier
Expression declaration section
Statement section
END_EXPRESSION
Note that an expression of the BOOL data type must be assigned to the expression identifier in the statement section!

Figure A-31 Expression

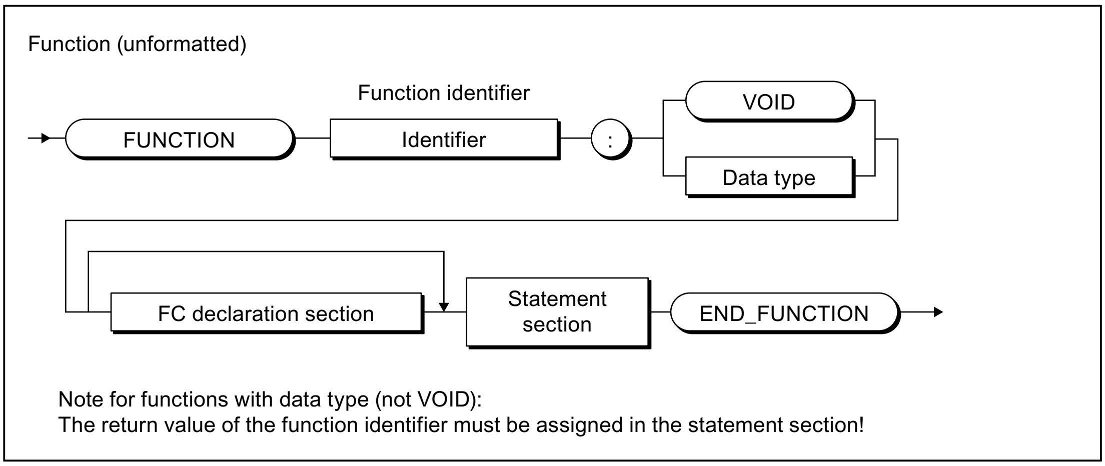

Function (unformatted)
Function identifier
FUNCTION
Identifier
:
VOID
Data type
FC declaration section
Statement section
END_FUNCTION
Note for functions with data type (not VOID):
The return value of the function identifier must be assigned in the statement section!

Figure A-32 Function (FC)

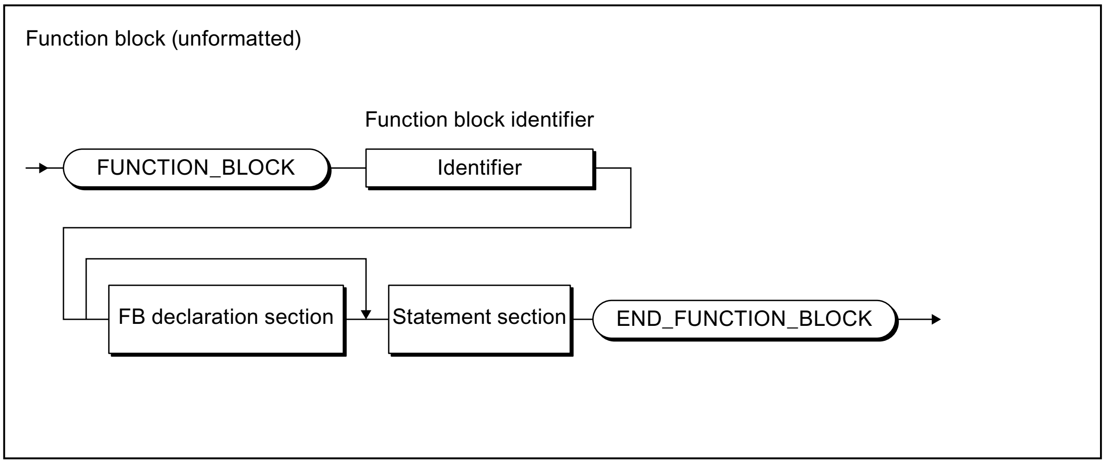


Figure A-33 Function block (FB)

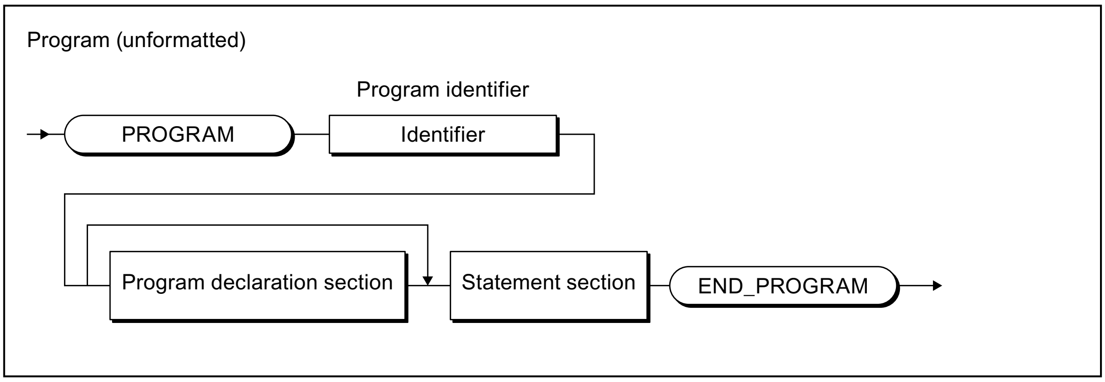
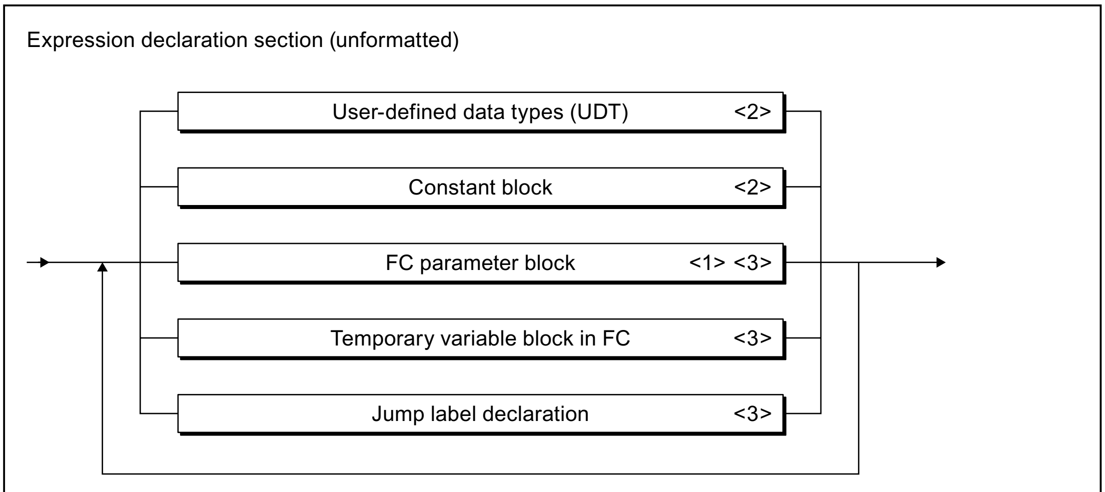


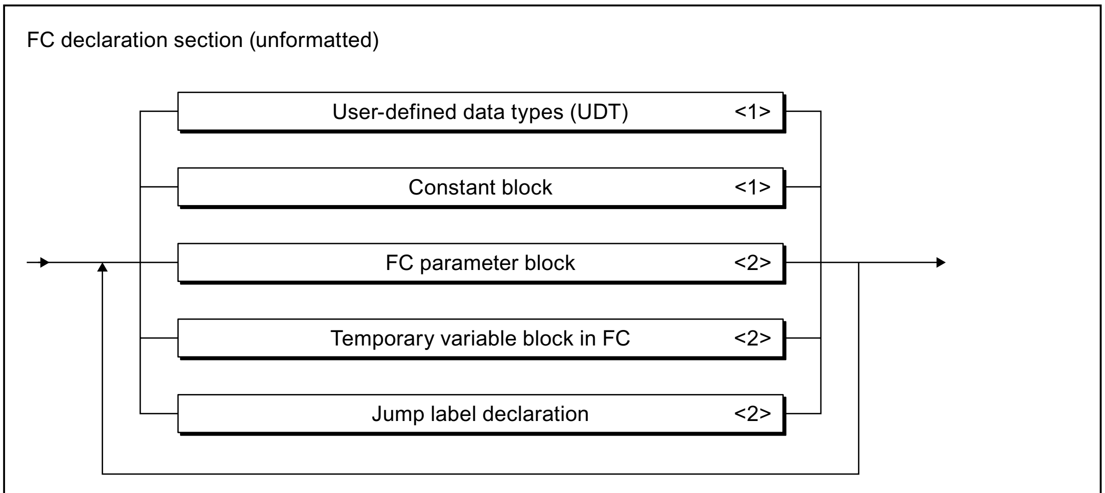
Figure A-34 Program

A.1.3.7 Declaration sections


Expression declaration section (unformatted)
User-defined data types (UDT) <2>
Constant block <2>
FC parameter block <1> <3>
Temporary variable block in FC <3>
Jump label declaration <3>

<1> The block is permitted only as of Version V4.1 of the SIMOTION kernel.
<2> This block may appear more than once in the declaration section.
<3> This block may appear just once in the declaration section.

Figure A-35 Expression declaration section


FC declaration section (unformatted)
User-defined data types (UDT) <1>
Constant block <1>
FC parameter block <2>
Temporary variable block in FC <2>
Jump label declaration <2>

<1> This block may appear more than once in the declaration section.
<2> This block may appear just once in the declaration section.

Figure A-36 FC declaration section

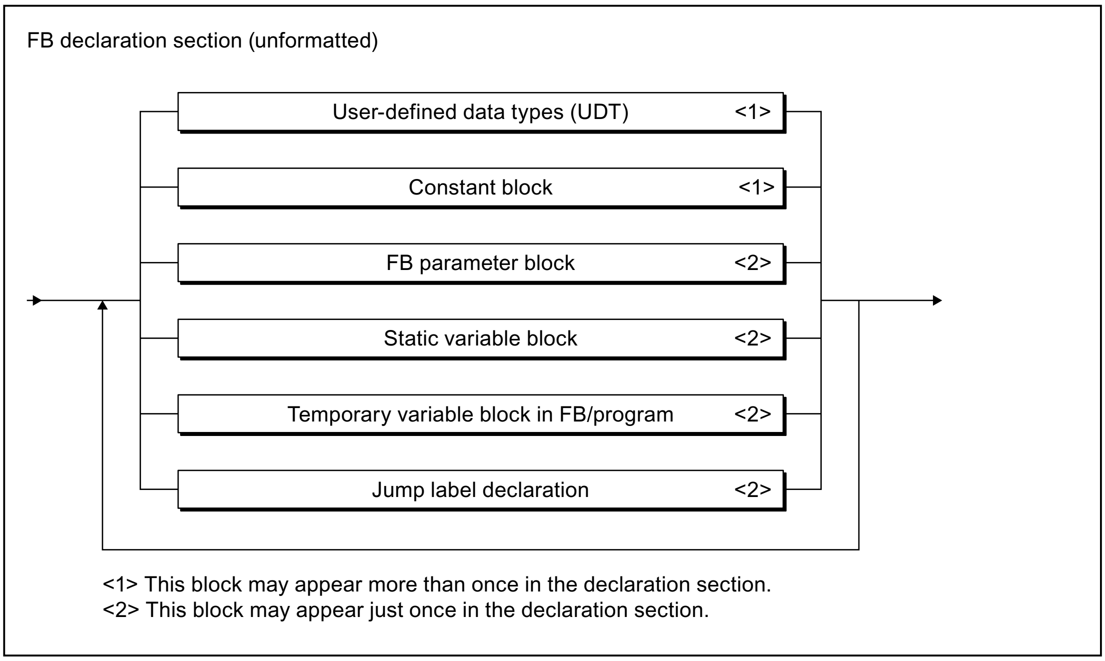

FB declaration section (unformatted)
User-defined data types (UDT) <1>
Constant block <1>
FB parameter block <2>
Static variable block <2>
Temporary variable block in FB/program <2>
Jump label declaration <2>
<1> This block may appear more than once in the declaration section.
<2> This block may appear just once in the declaration section.

Figure A-37 FB declaration section

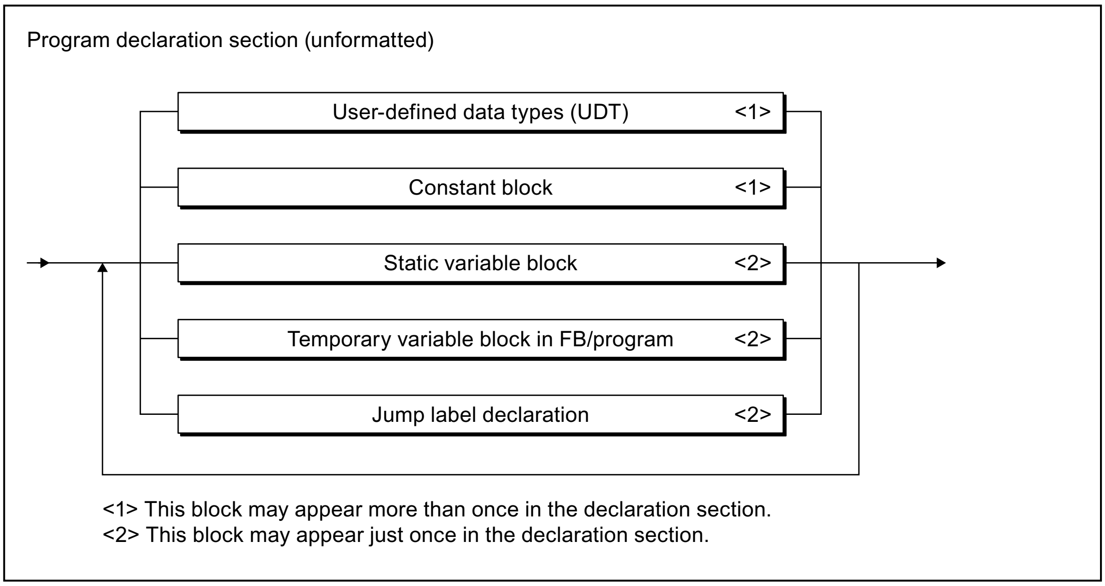

Program declaration section (unformatted)
User-defined data types (UDT) <1>
Constant block <1>
Static variable block <2>
Temporary variable block in FB/program <2>
Jump label declaration <2>
<1> This block may appear more than once in the declaration section.
<2> This block may appear just once in the declaration section.

Figure A-38 Program declaration section

A.1.3.8 Structure of the declaration blocks

Constant blocks

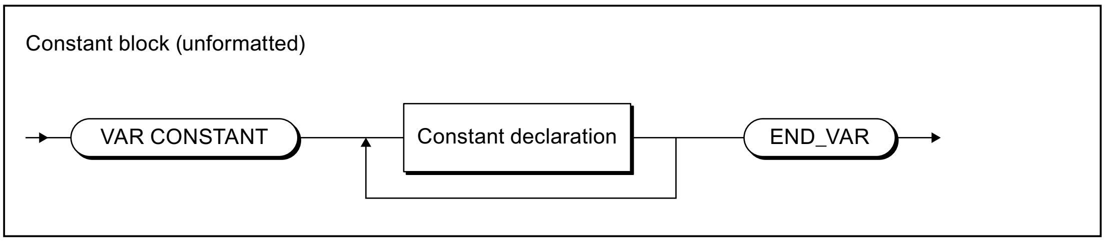


Figure A-39 Constant block

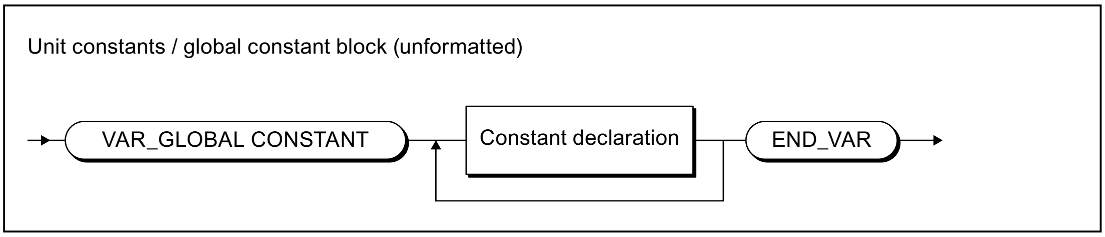


Figure A-40 Unit constants / global constant block

Variable blocks

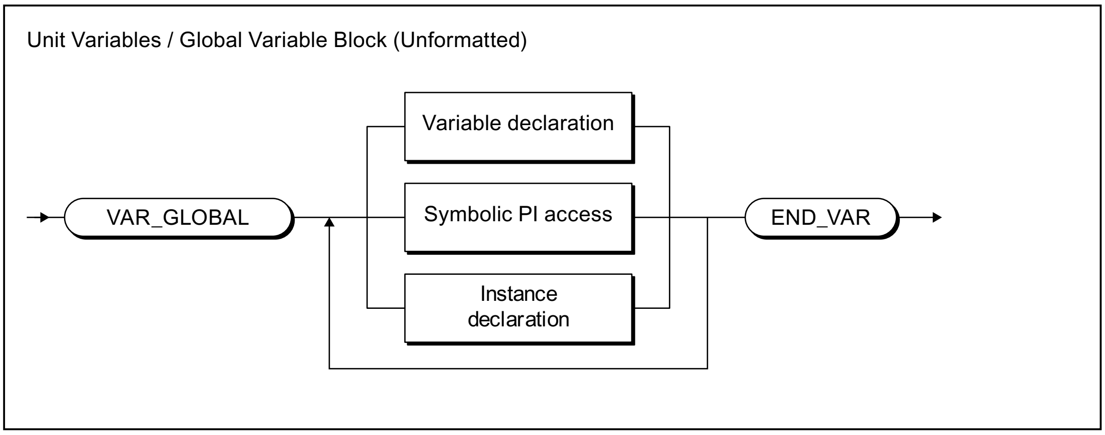


Figure A-41 Unit variables / global variable block

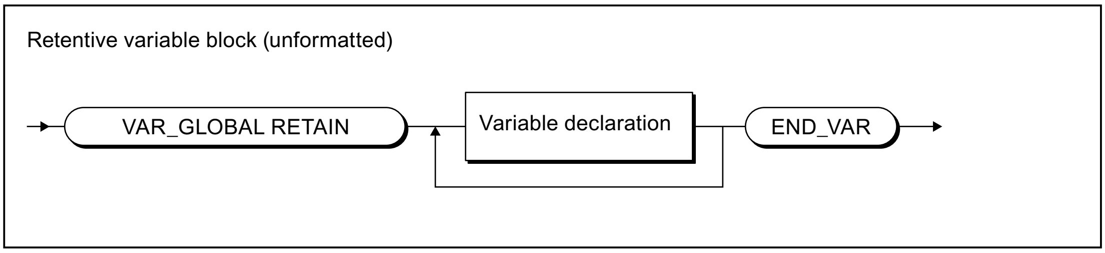


Figure A-42 Retentive variable block

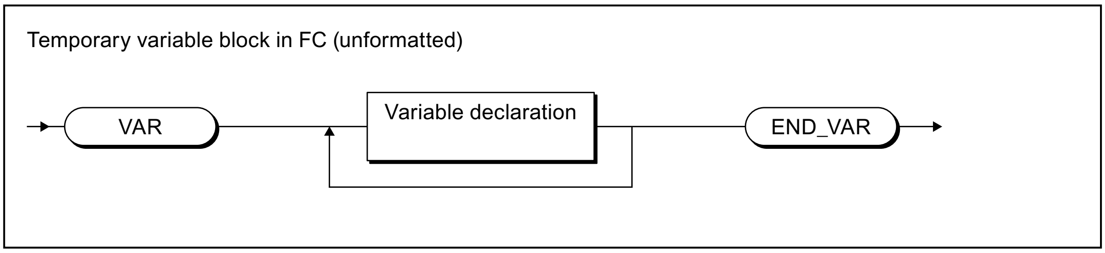


Figure A-43 Temporary variable block in FC

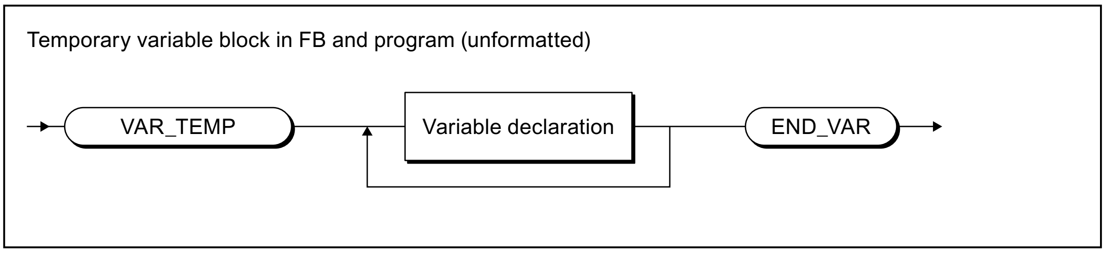


Figure A-44 Temporary variable block in the FB/program

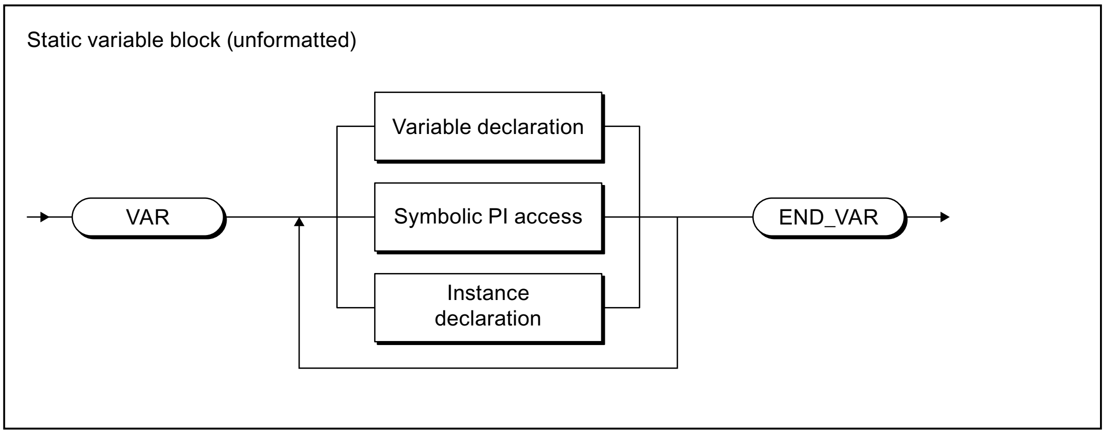


Figure A-45 Static variable block

Parameter fields

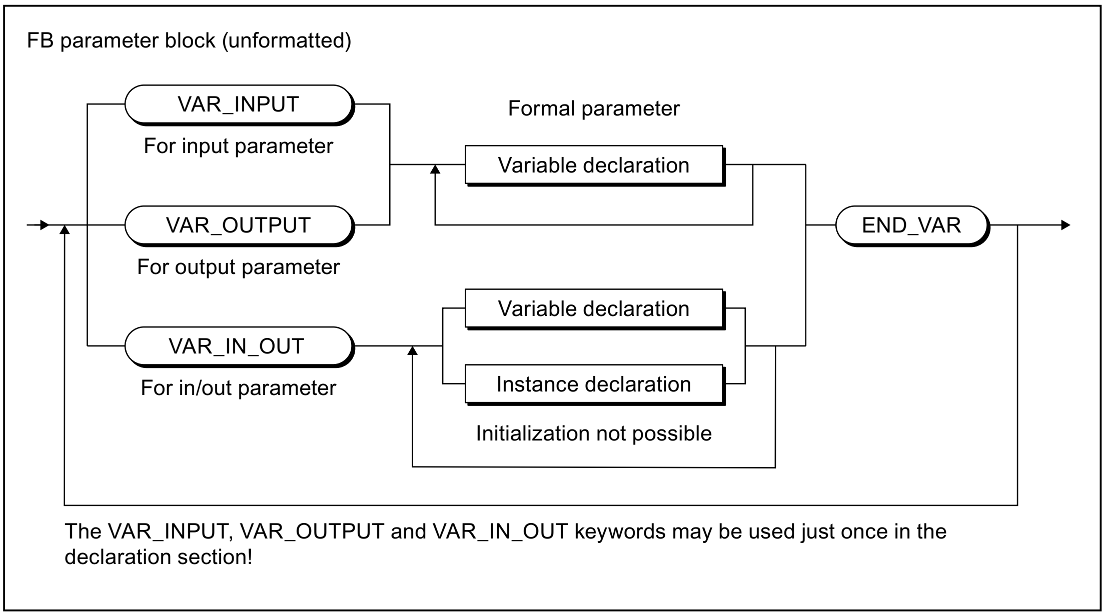


Figure A-46 FB parameter block

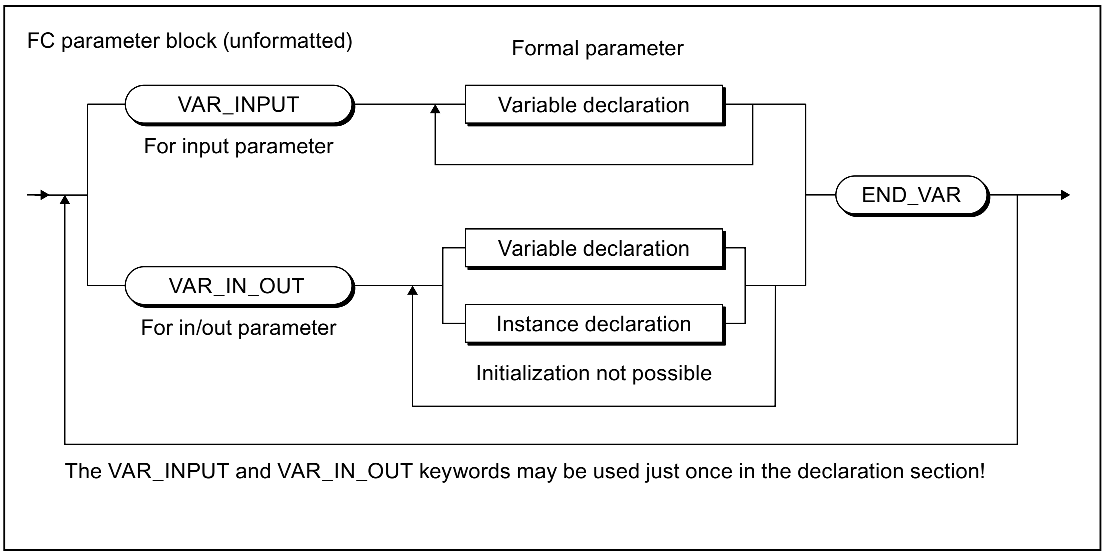

FC parameter block (unformatted)
Formal parameter
VAR_INPUT
For input parameter
Variable declaration
END_VAR
Variable declaration
VAR_IN_OUT
For in/out parameter
Instance declaration
Initialization not possible
The VAR_INPUT and VAR_IN_OUT keywords may be used just once in the declaration section!

Figure A-47 FC parameter block

Jump labels

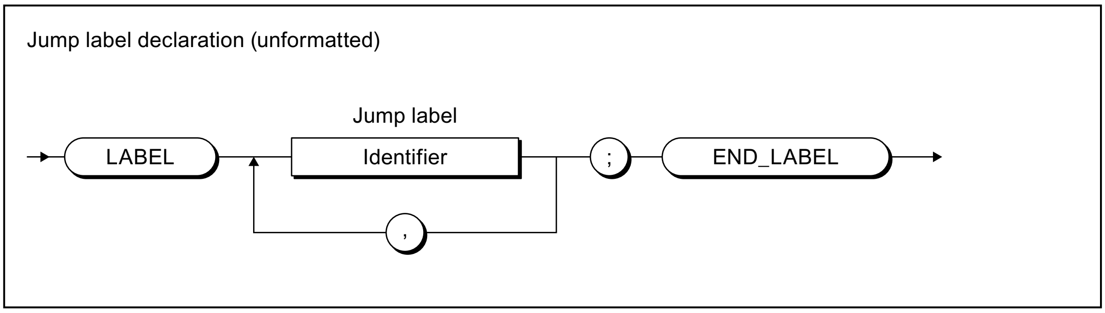


Figure A-48 Jump label declaration

Declarations

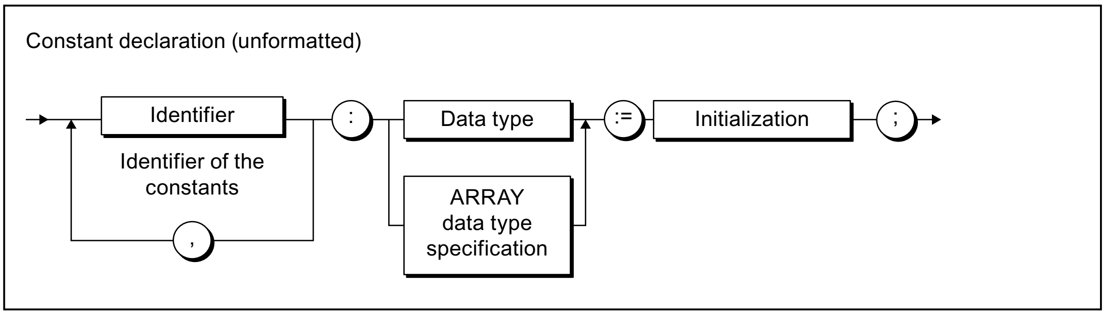

Constant declaration (unformatted)
Identifier
Identifier of the constants
,
:
Data type
ARRAY data type specification
:=
Initialization
;

Figure A-49 Constant declaration

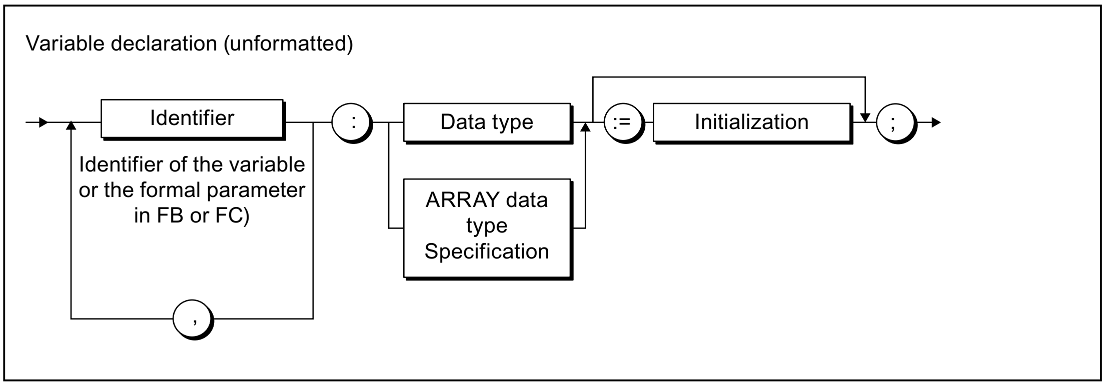

Variable declaration (unformatted)
Identifier
Identifier of the variable or the formal parameter in FB or FC)
,
:
Data type
ARRAY data type Specification
:=
Initialization
;

Figure A-50 Variable declaration

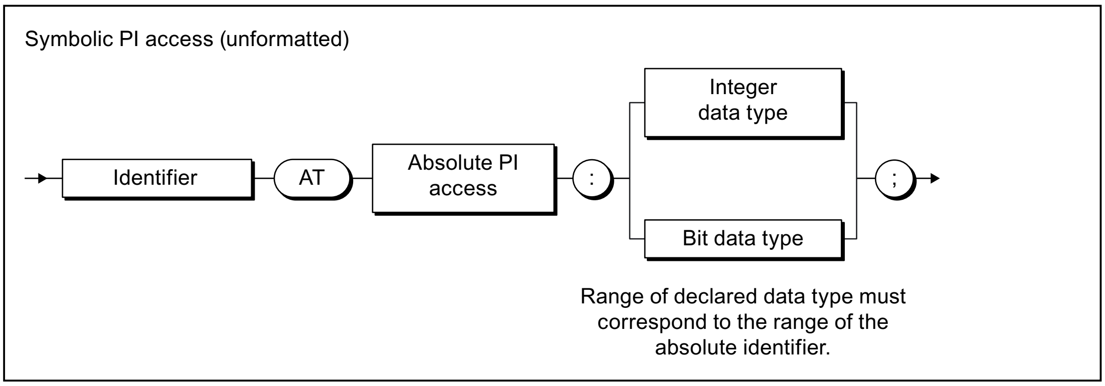

Symbolic PI access (unformatted)
Identifier
AT
Absolute PI access
:
Integer data type
Bit data type
;
Range of declared data type must correspond to the range of the absolute identifier.

Figure A-51 Symbolic PI access

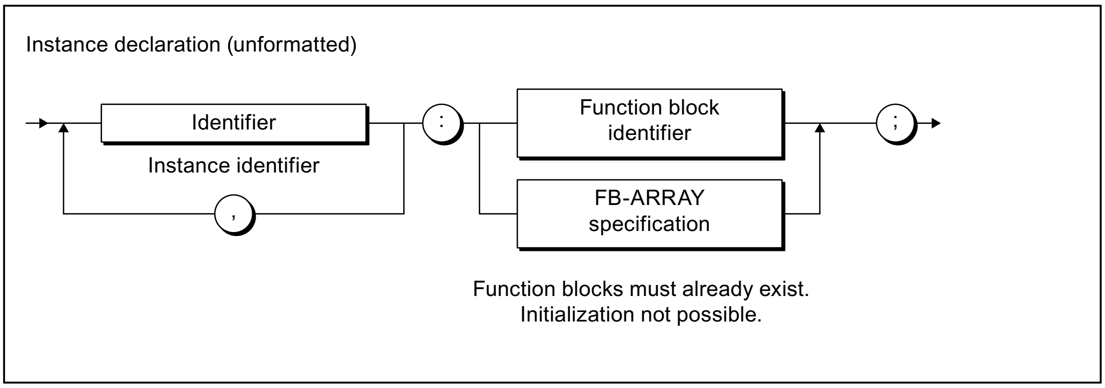

Instance declaration (unformatted)
Identifier
Instance identifier
,
:
Function block identifier
FB-ARRAY specification
;
Function blocks must already exist.
Initialization not possible.

Figure A-52 Instance declaration

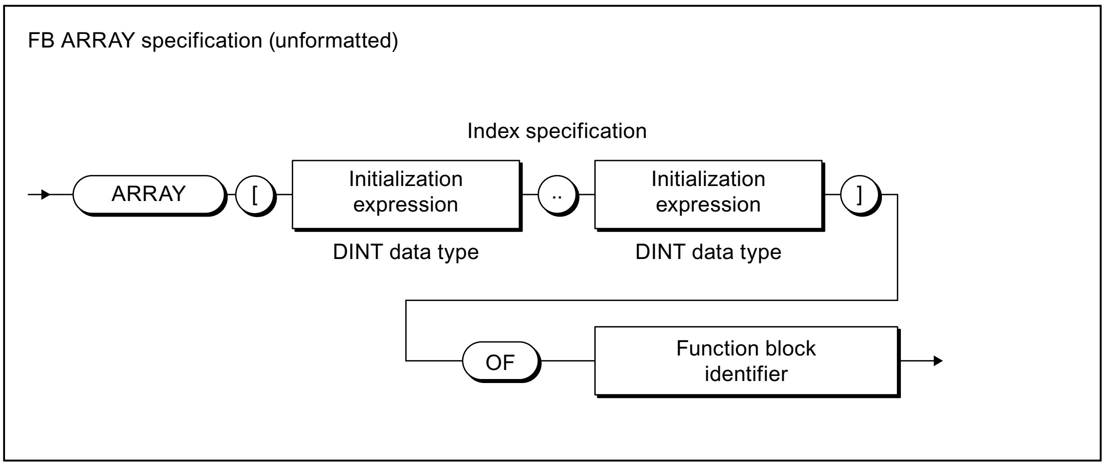

FB ARRAY specification (unformatted)
Index specification
ARRAY
[
Initialization expression
DINT data type
..
Initialization expression
DINT data type
]
OF
Function block identifier

Figure A-53 FB ARRAY specification

Initialization

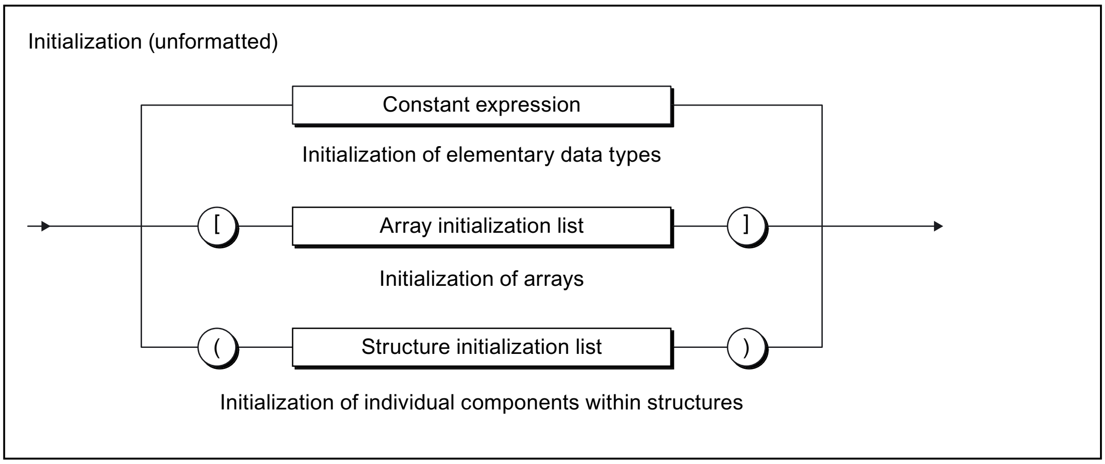

Initialization (unformatted)
Constant expression
Initialization of elementary data types
[
Array initialization list
]
Initialization of arrays
(
Structure initialization list
)
Initialization of individual components within structures

Figure A-54 Initialization

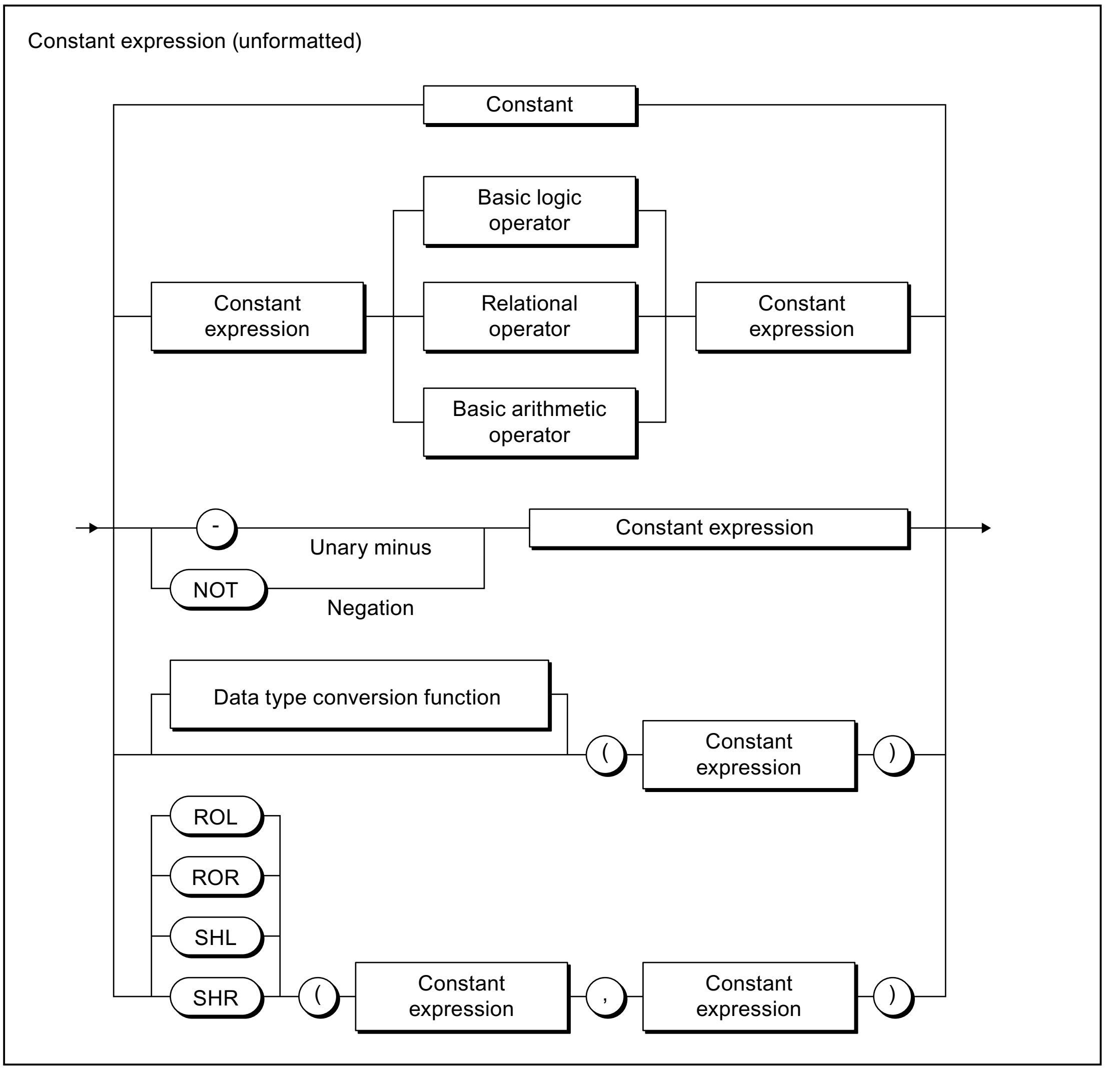


Figure A-55 Constant expression

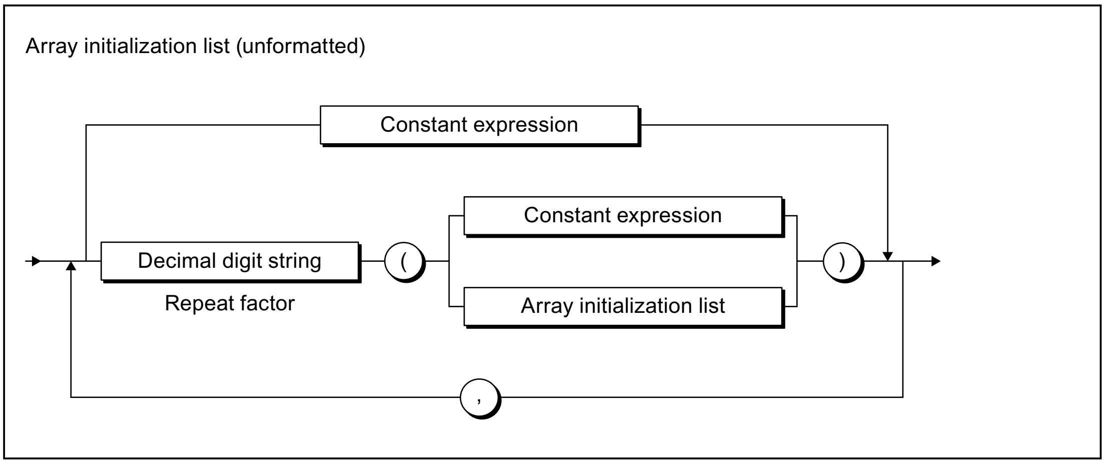


Figure A-56 Array initialization list

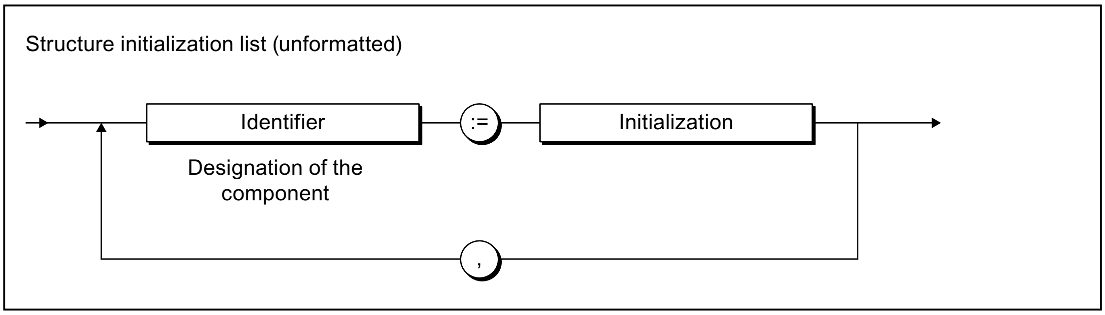

Structure initialization list (unformatted)
Identifier
Designation of the component
:=
Initialization
,

Figure A-57 Structure initialization list

A.1.3.9 Data types

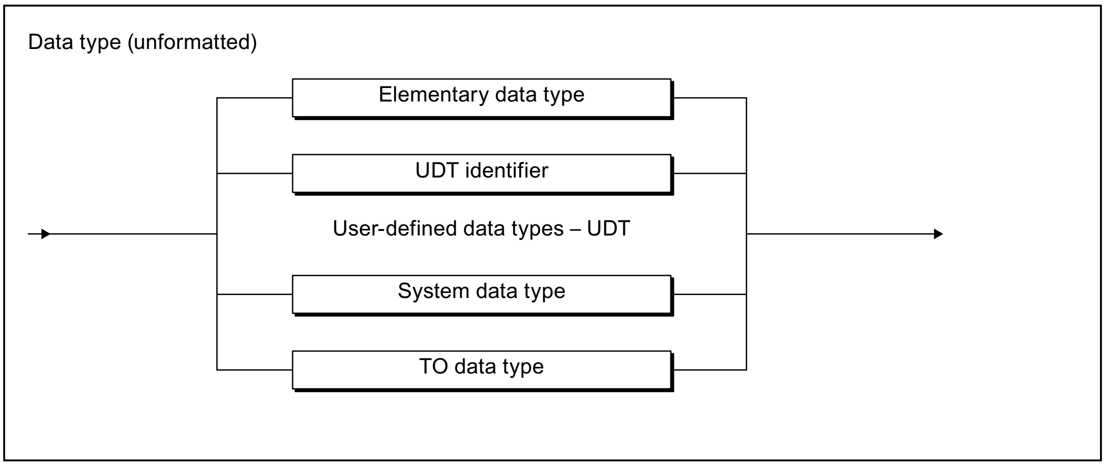


Figure A-58 Data type

Elementary data types

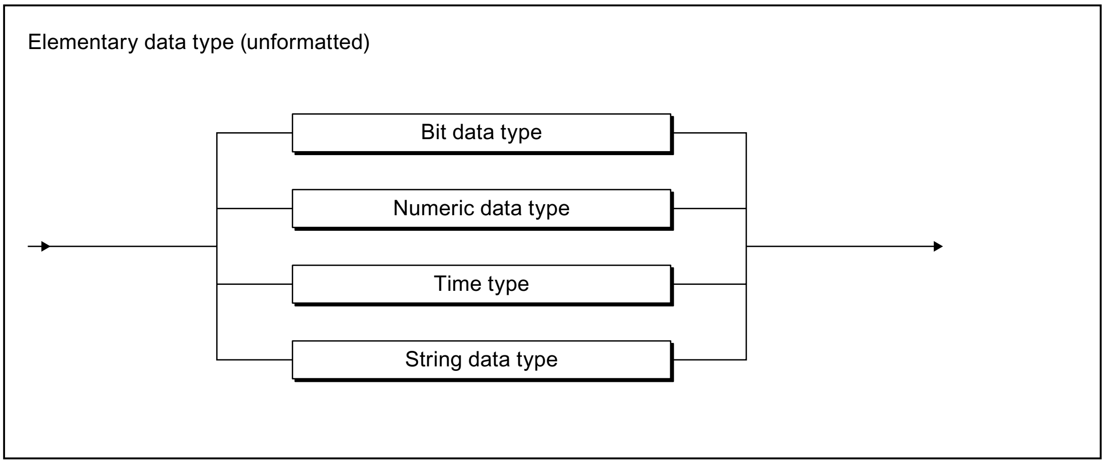


Figure A-59 Elementary data type

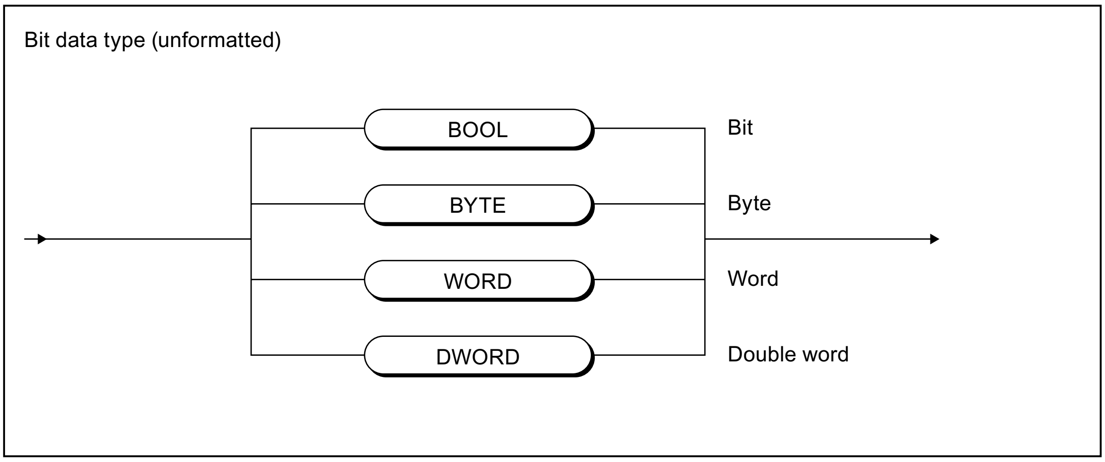


Figure A-60 Bit data type

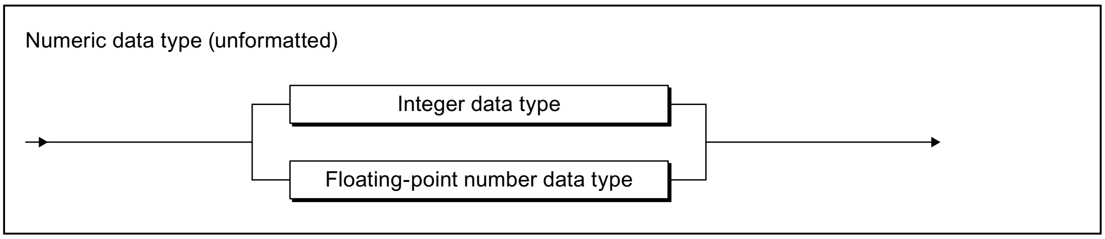

Numeric data type (unformatted)
Integer data type
Floating-point number data type

Figure A-61 Numeric data type

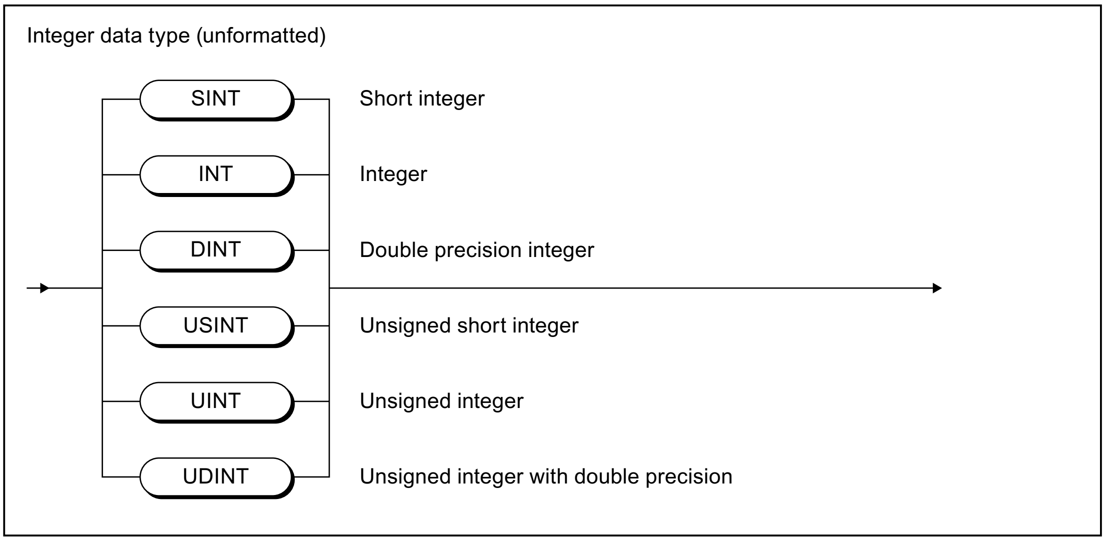


Figure A-62 Integer data type

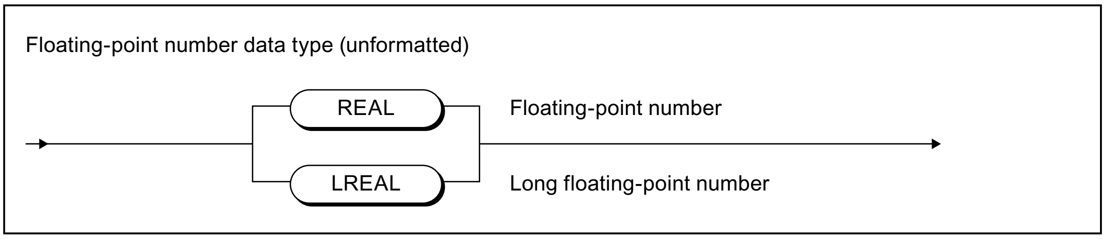

Floating-point number data type (unformatted)
REAL
Floating-point number
LREAL
Long floating-point number

Figure A-63 Floating-point number data type

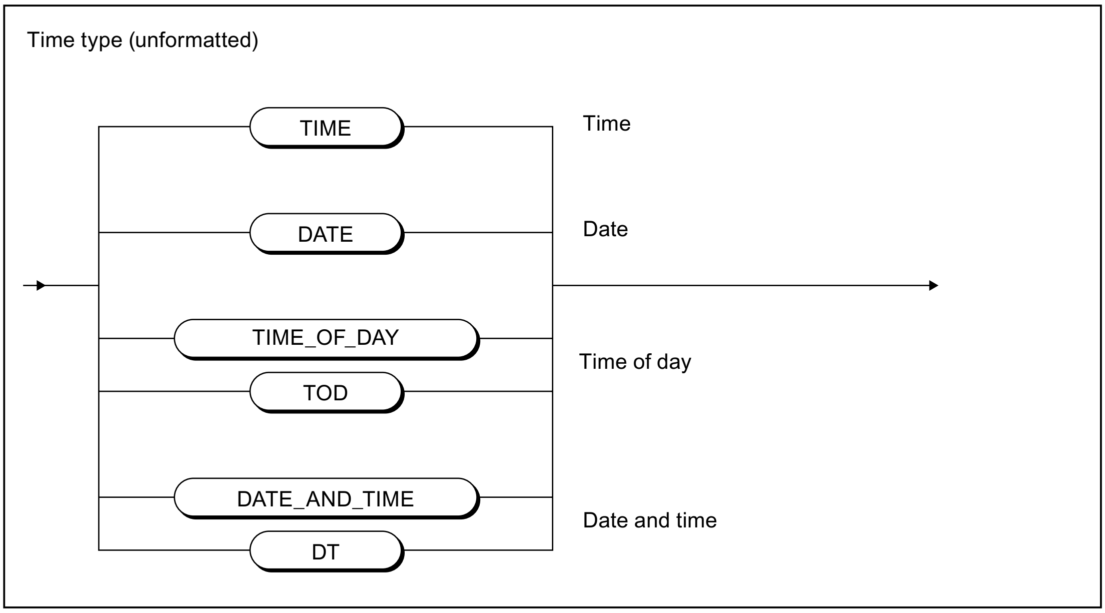


Figure A-64 Time data type

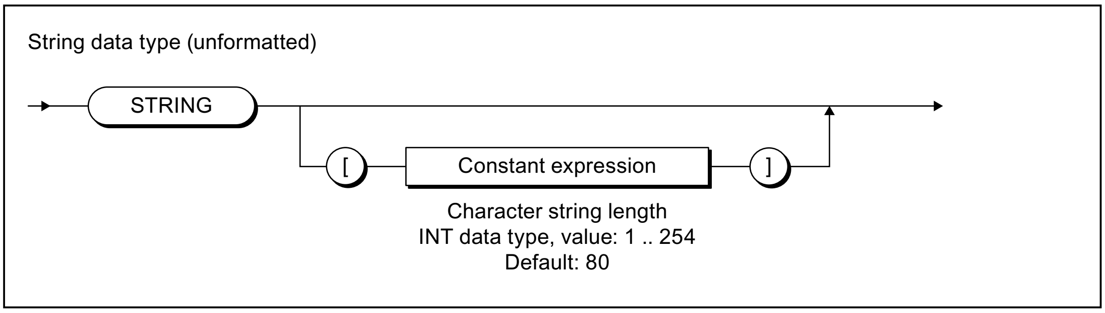


Figure A-65 String data type

User-defined data types

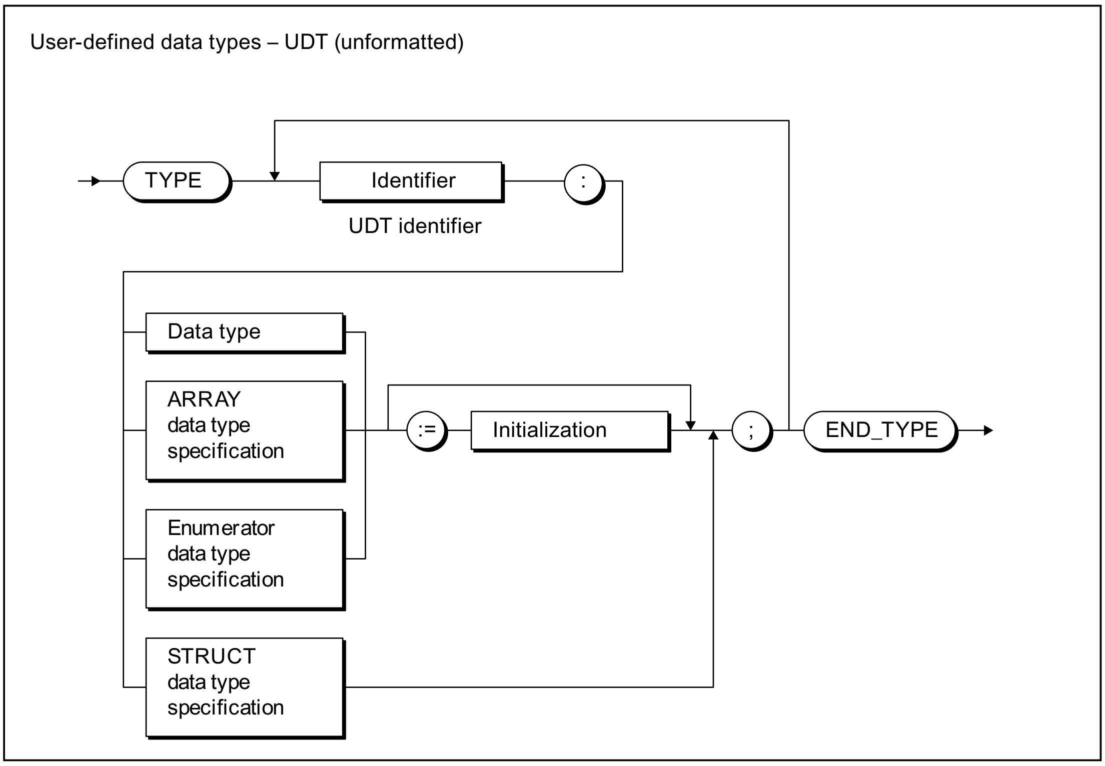

User-defined data types – UDT (unformatted)
TYPE
Identifier
:
UDT identifier
Data type
ARRAY data type specification
Enumerator data type specification
STRUCT data type specification
:=
Initialization
;
END_TYPE

Figure A-66 User-defined data type

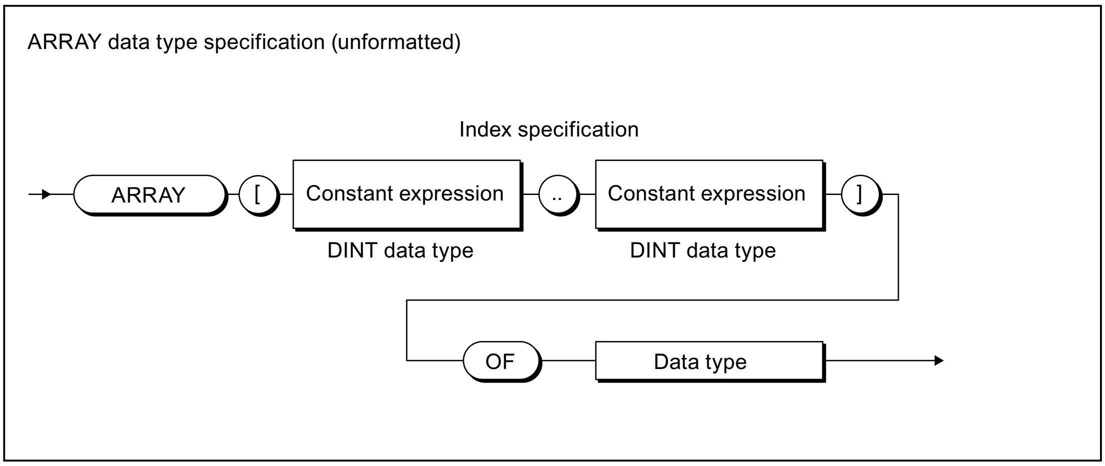


Figure A-67 ARRAY data type specification

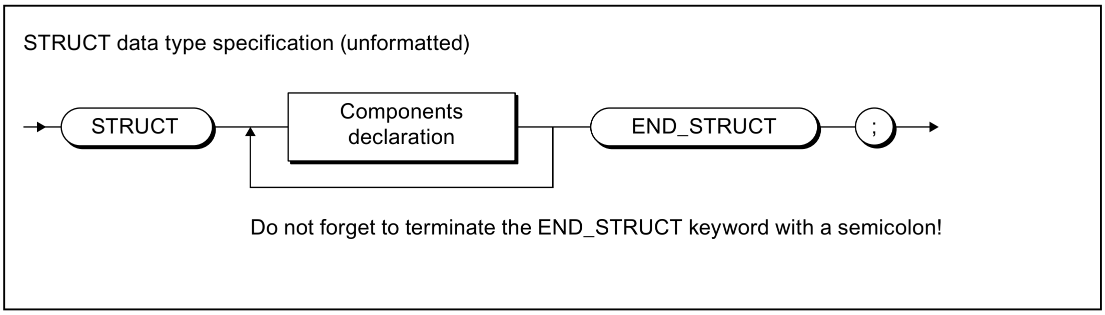

STRUCT data type specification (unformatted)
STRUCT
Components declaration
END_STRUCT
;
Do not forget to terminate the END_STRUCT keyword with a semicolon!

Figure A-68 STRUCT data type specification

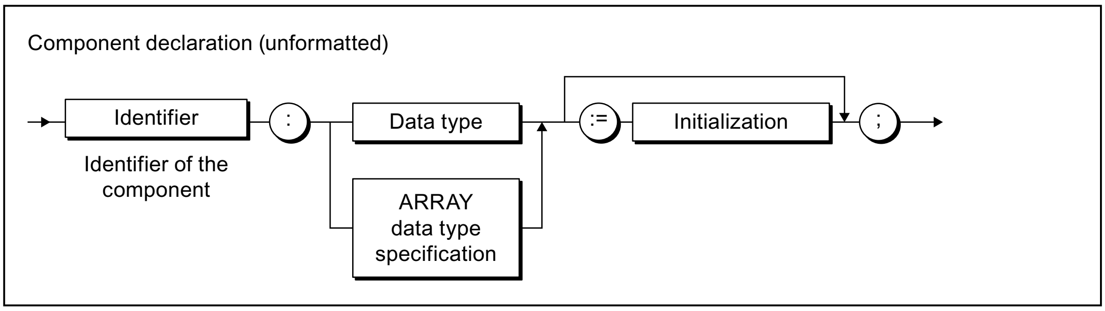

Component declaration (unformatted)
Identifier
Identifier of the component
:
Data type
ARRAY data type specification
:=
Initialization
;

Figure A-69 Component declaration

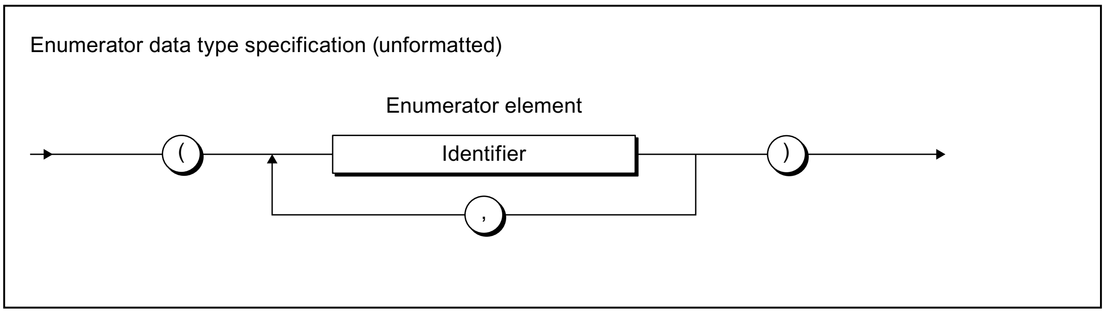


Figure A-70 Enumerator data type specification

A.1.3.10 Statement section

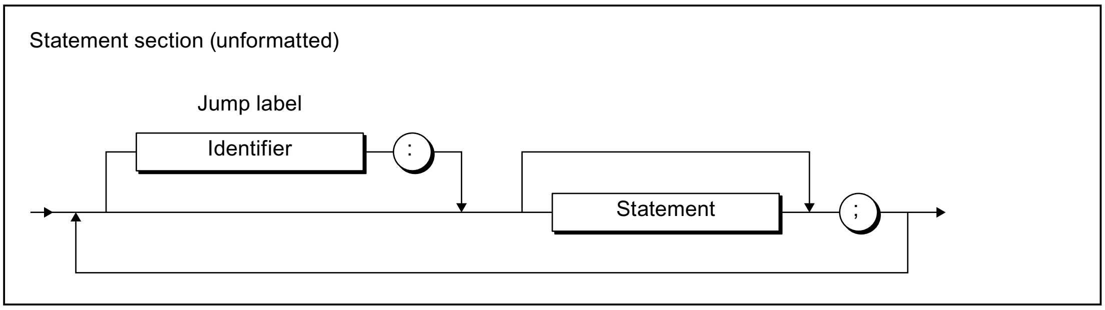


Figure A-71 Statement section

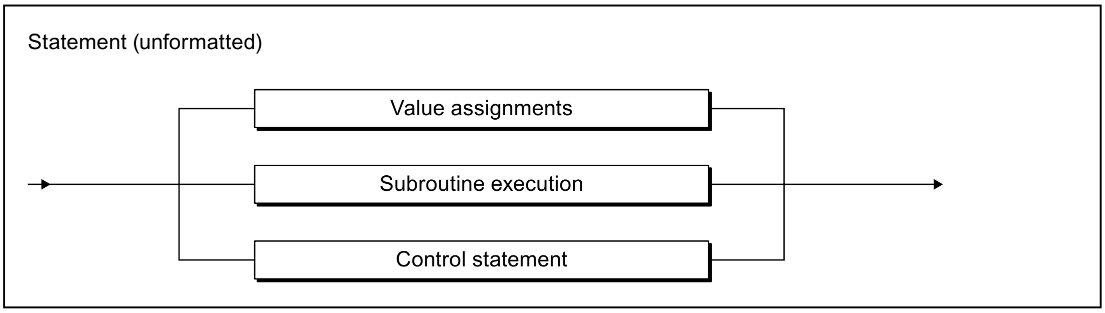


Figure A-72 Statement

A.1.3.11 Value assignments and operations

Value assignment and expression

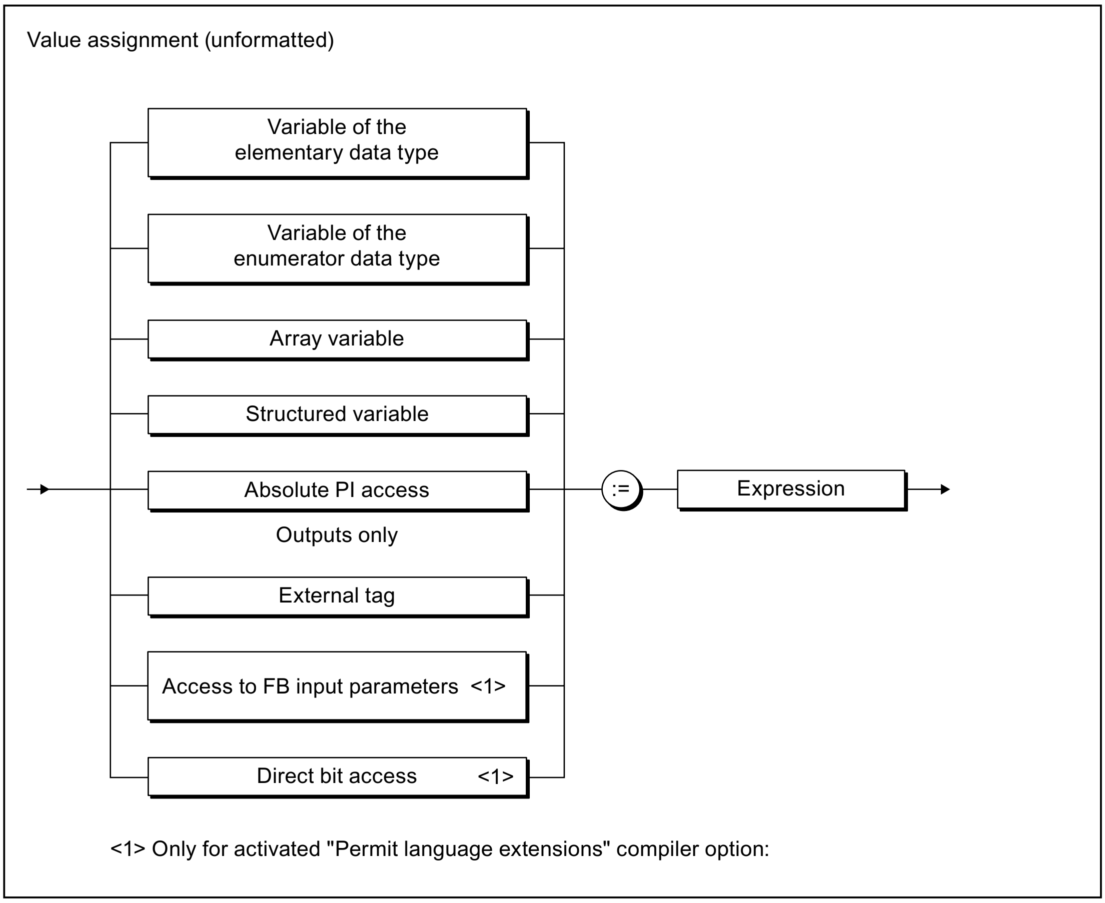

Value assignment (unformatted)
Variable of the elementary data type
Variable of the enumerator data type
Array variable
Structured variable
Absolute PI access
Outputs only
External tag
Access to FB input parameters <1>
Direct bit access <1>
:=
Expression
<1> Only for activated "Permit language extensions" compiler option:

Figure A-73 Value assignments

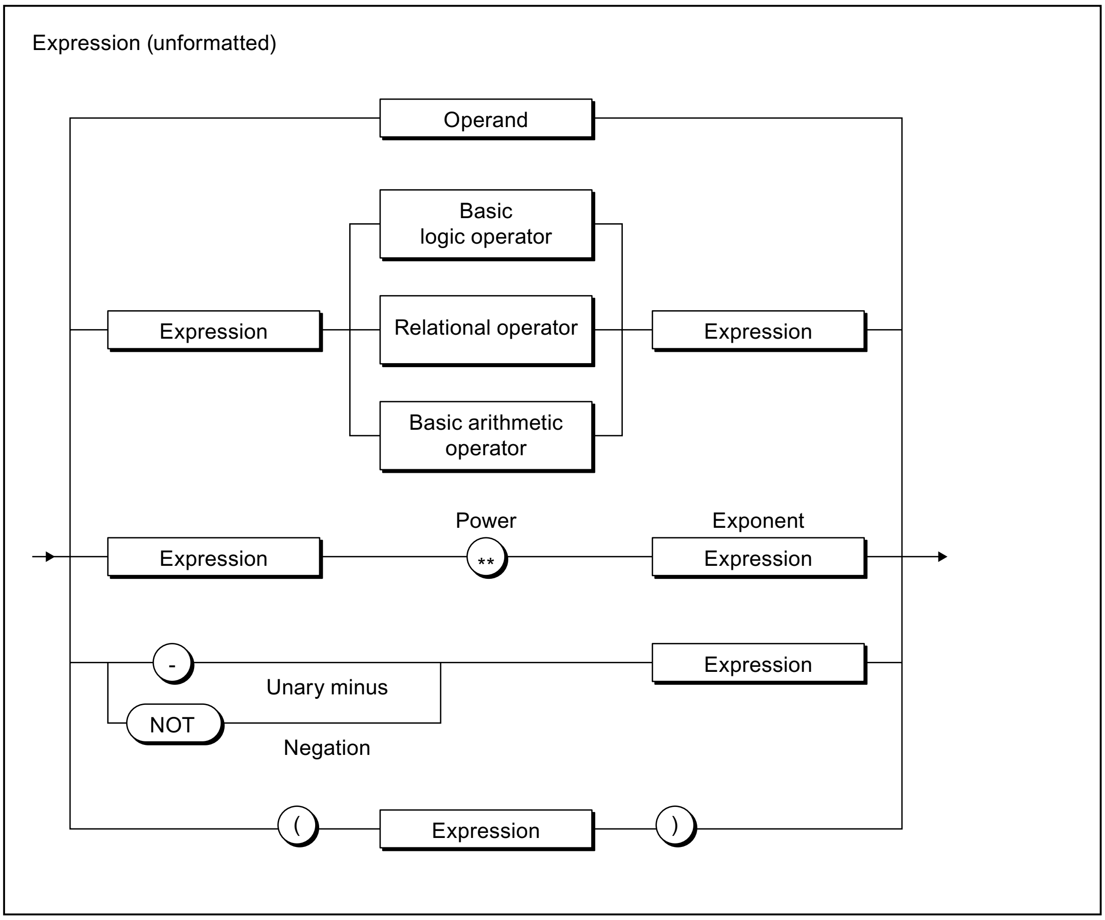


Figure A-74 Expression

Operands

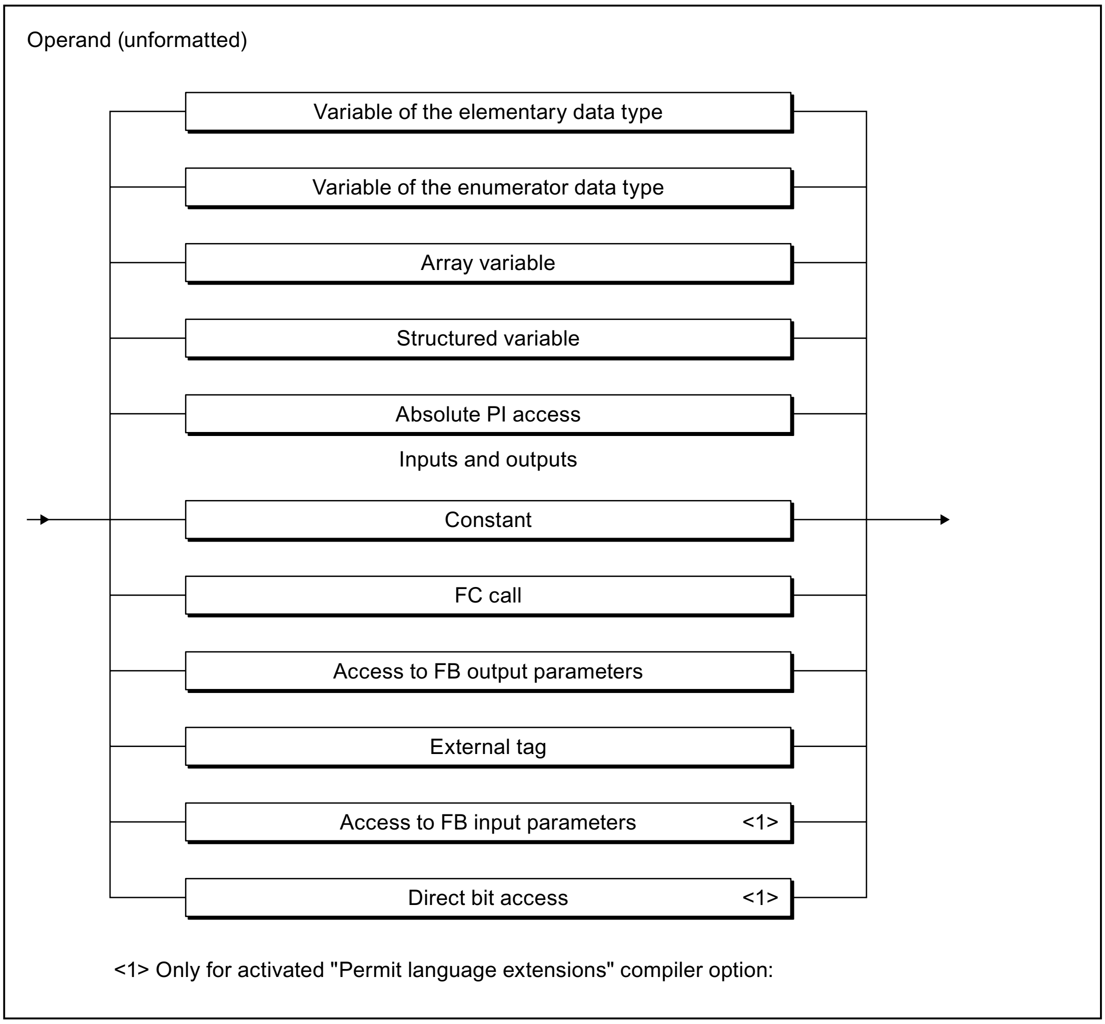

Operand (unformatted)
Variable of the elementary data type
Variable of the enumerator data type
Array variable
Structured variable
Absolute PI access
Inputs and outputs
Constant
FC call
Access to FB output parameters
External tag
Access to FB input parameters <1>
Direct bit access <1>
<1> Only for activated "Permit language extensions" compiler option:

Figure A-75 Operand

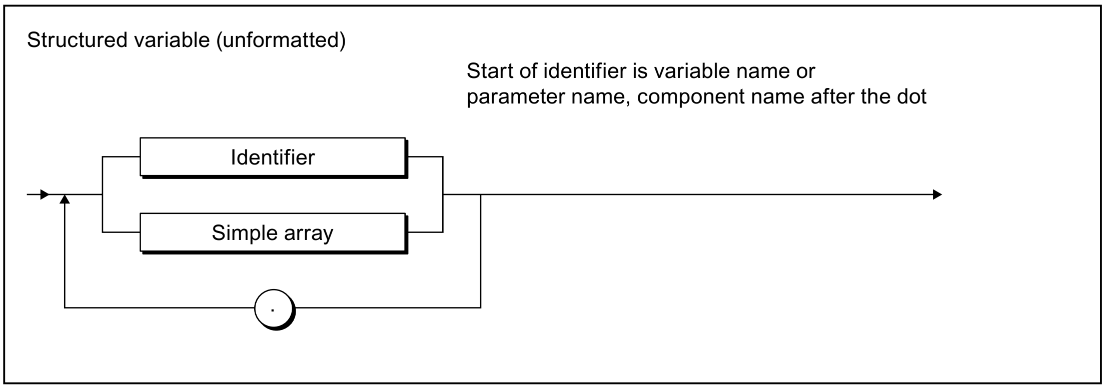

Structured variable (unformatted)
Start of identifier is variable name or parameter name, component name after the dot
Identifier
Simple array
.

Figure A-76 Structured variable

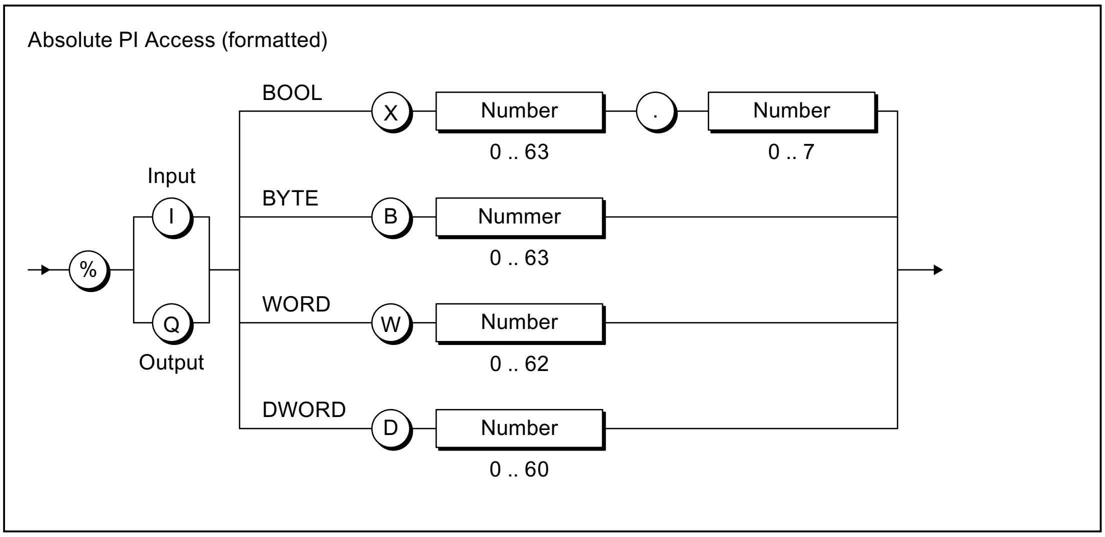


Figure A-77 Absolute PI access

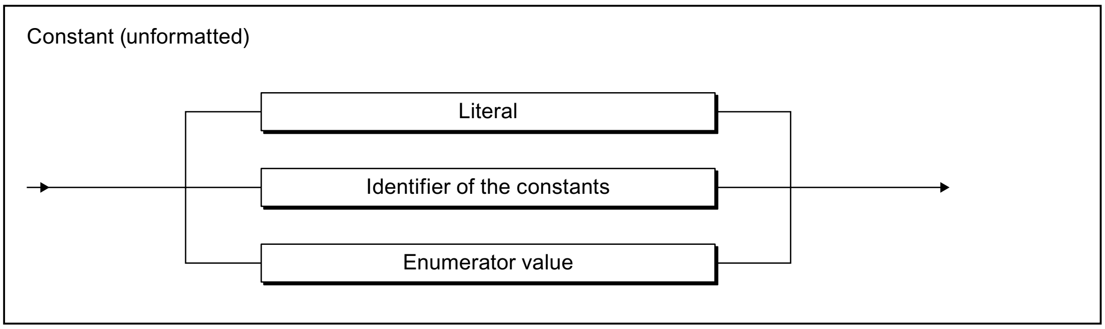


Figure A-78 Constant

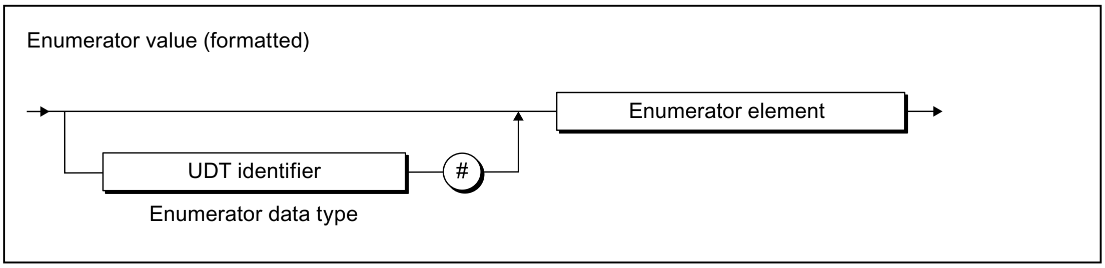


Figure A-79 Enumerator value

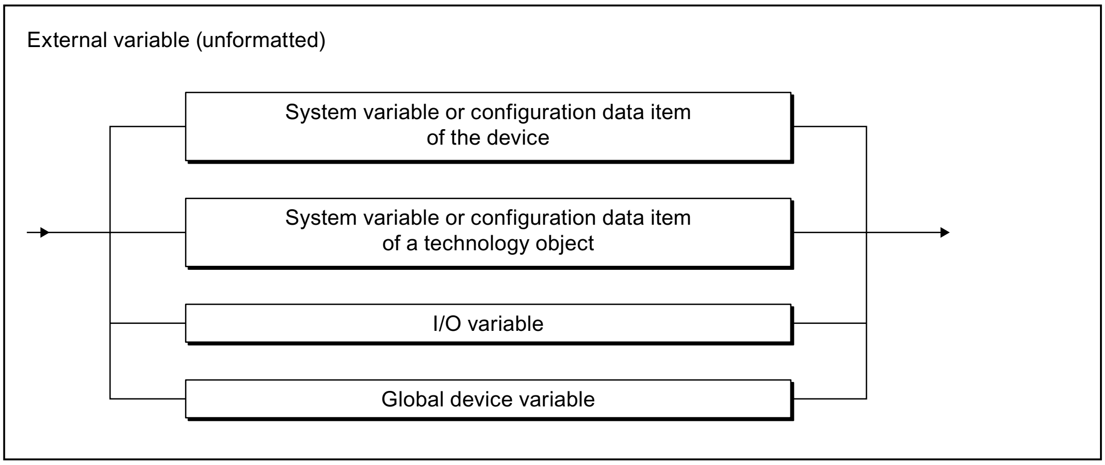

External variable (unformatted)
System variable or configuration data item of the device
System variable or configuration data item of a technology object
I/O variable
Global device variable

Figure A-80 External tag

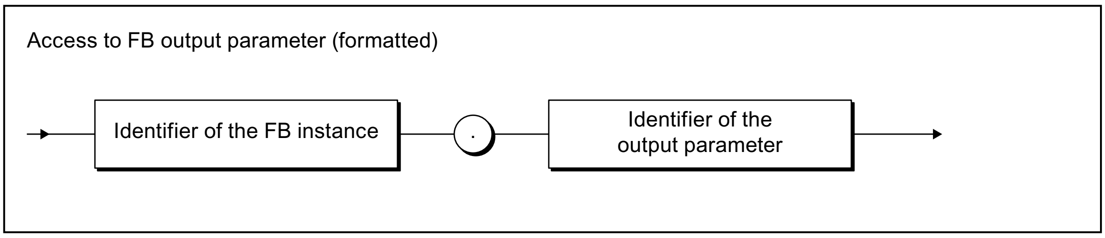

Access to FB output parameter (formatted)
Identifier of the FB instance
.
Identifier of the output parameter

Figure A-81 Access to FB output parameters

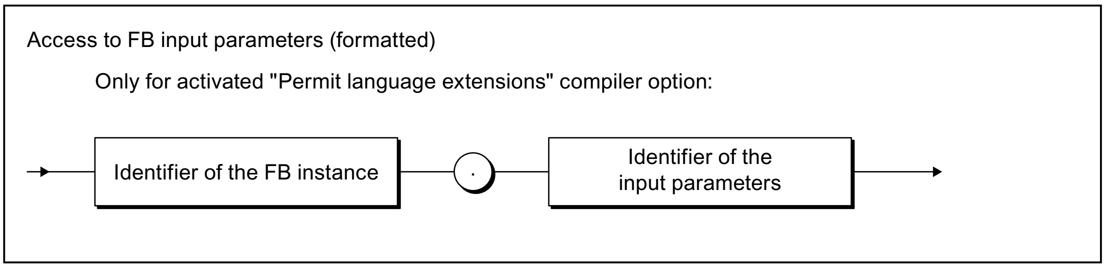

Access to FB input parameters (formatted)
Only for activated "Permit language extensions" compiler option:
Identifier of the FB instance
.
Identifier of the input parameters

Figure A-82 Access to FB input parameters

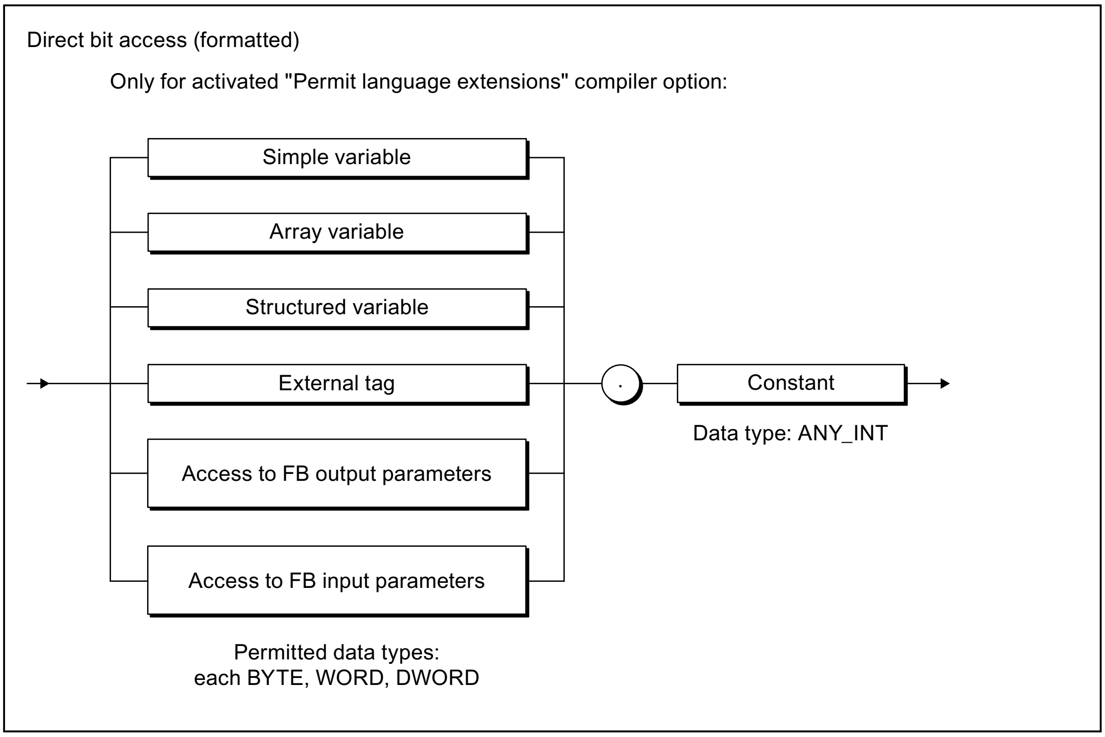

Direct bit access (formatted)
Only for activated "Permit language extensions" compiler option:
Simple variable
Array variable
Structured variable
External tag
Access to FB output parameters
Access to FB input parameters
Permitted data types:
each BYTE, WORD, DWORD
.
Constant
Data type: ANY_INT

Figure A-83 Bit access

Operators

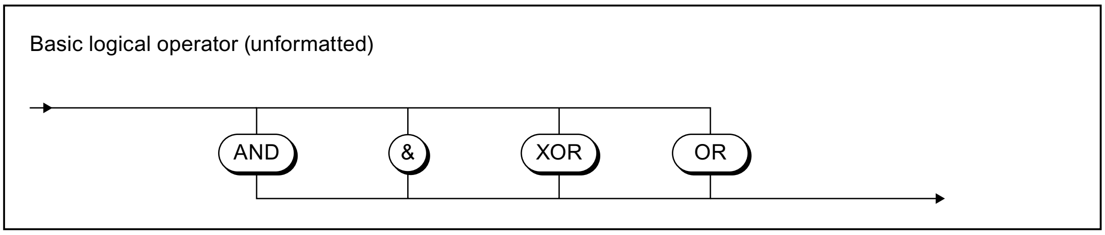


Figure A-84 Basic logic operator

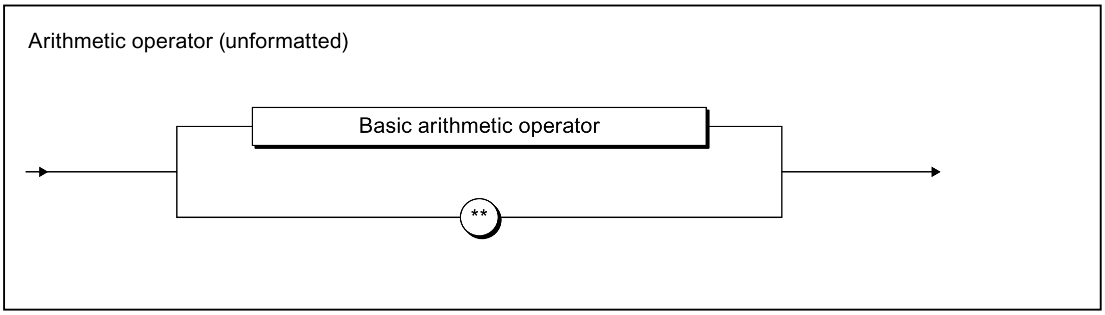


Figure A-85 Arithmetic operator

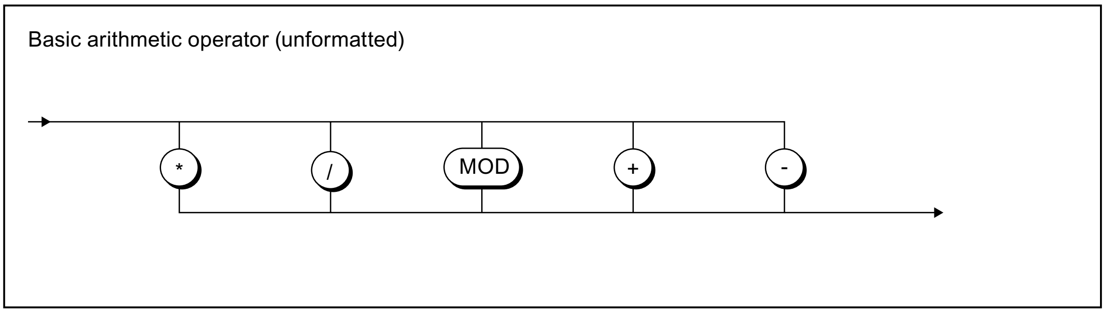


Figure A-86 Basic arithmetic operator

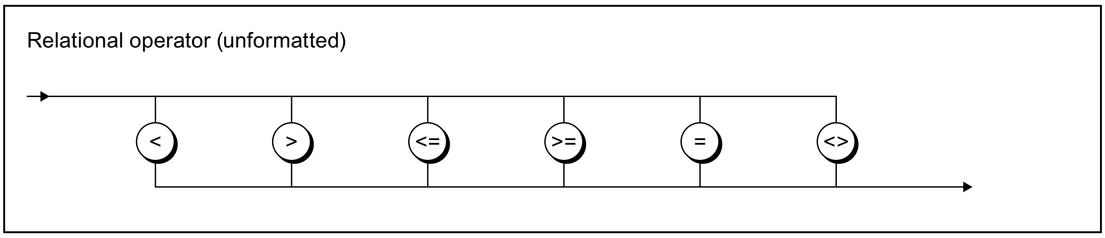


Figure A-87 Relational operators

A.1.3.12 Call of functions and function block calls

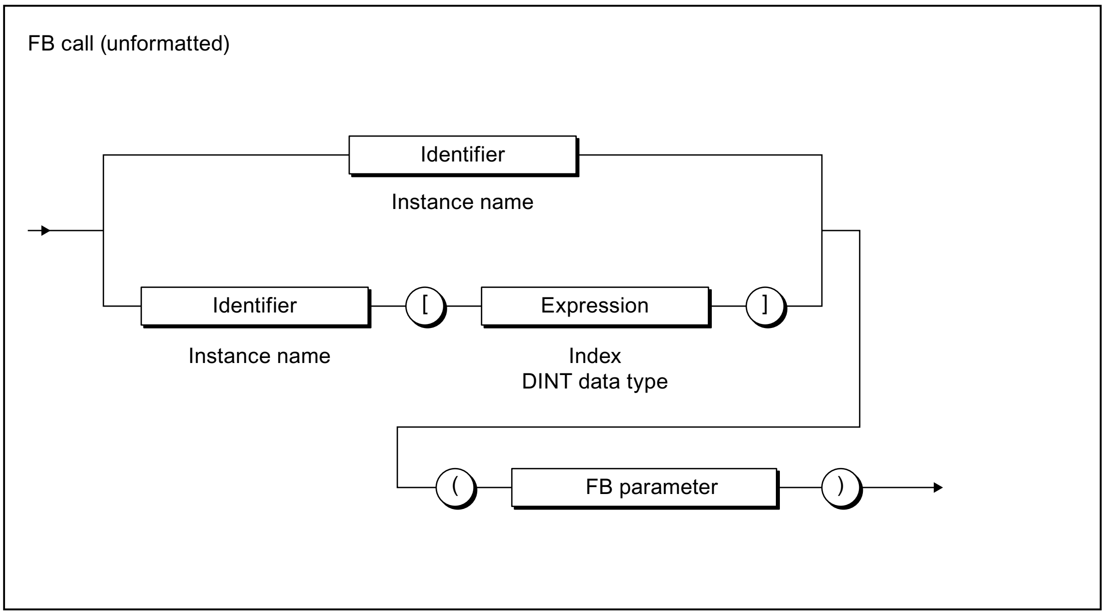


Figure A-88 FB call

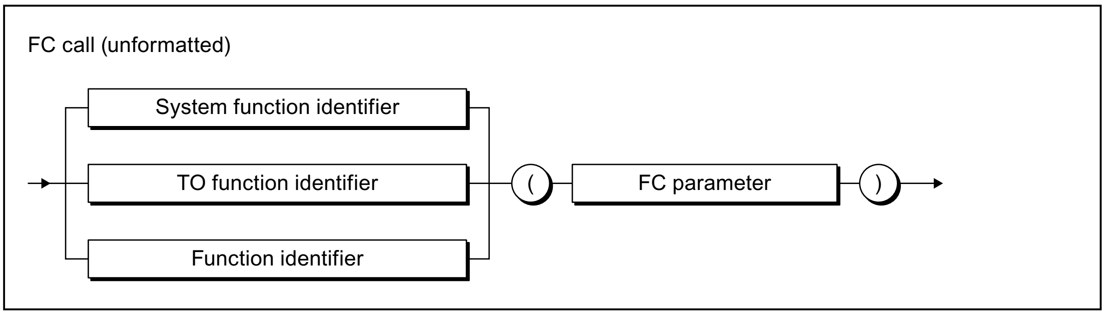


Figure A-89 FC call

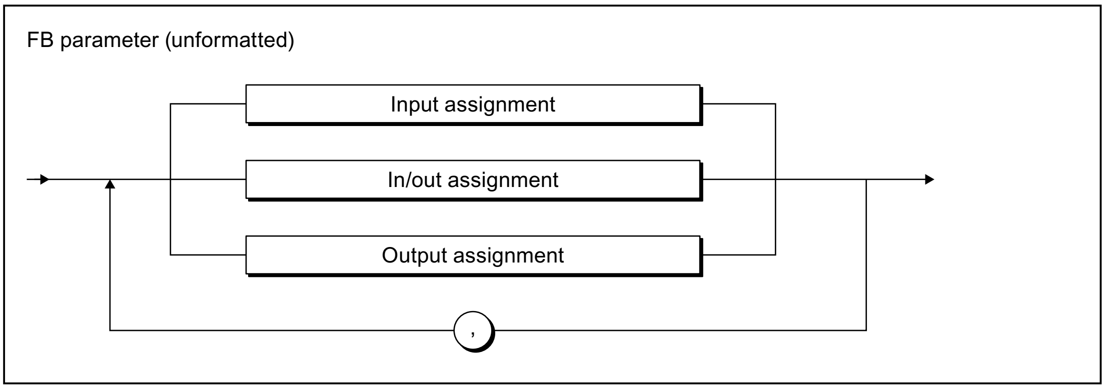


Figure A-90 FB parameter

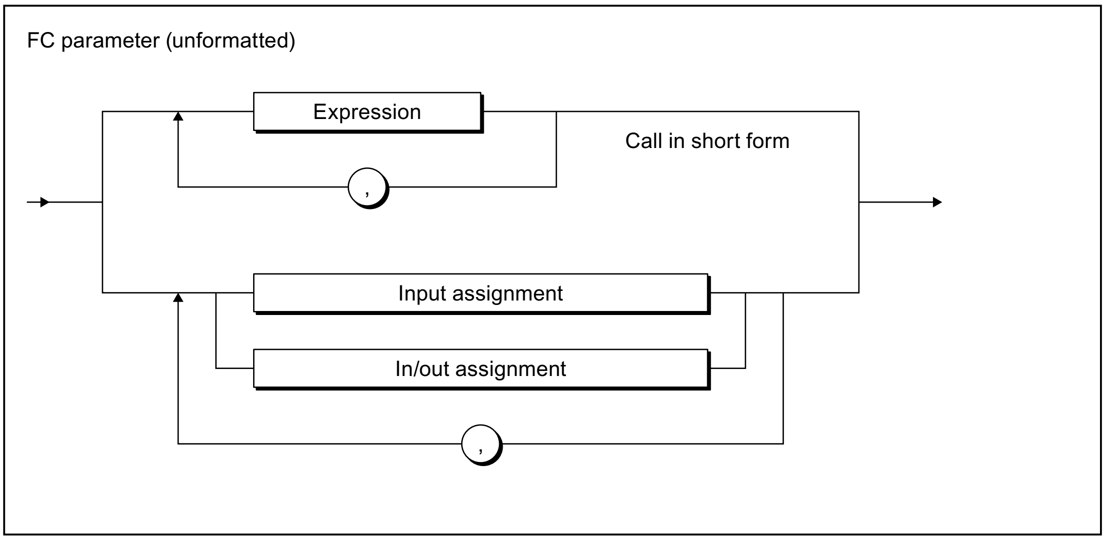


Figure A-91 FC parameter

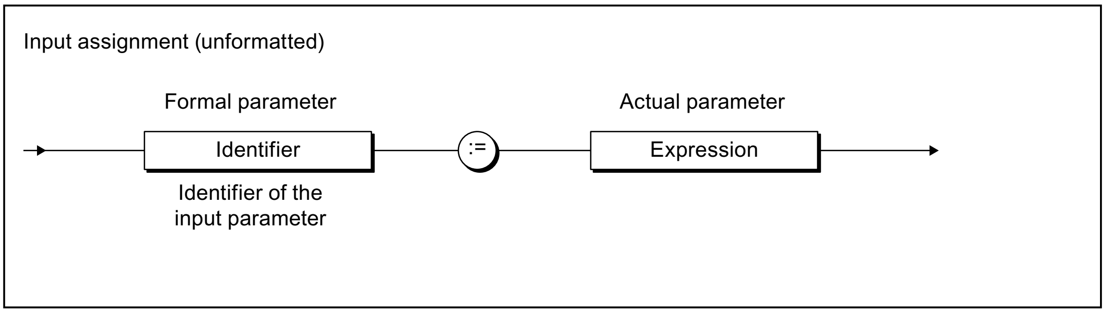

Input assignment (unformatted)
Formal parameter
Actual parameter
Identifier
:=
Expression
Identifier of the input parameter

Figure A-92 Input assignment

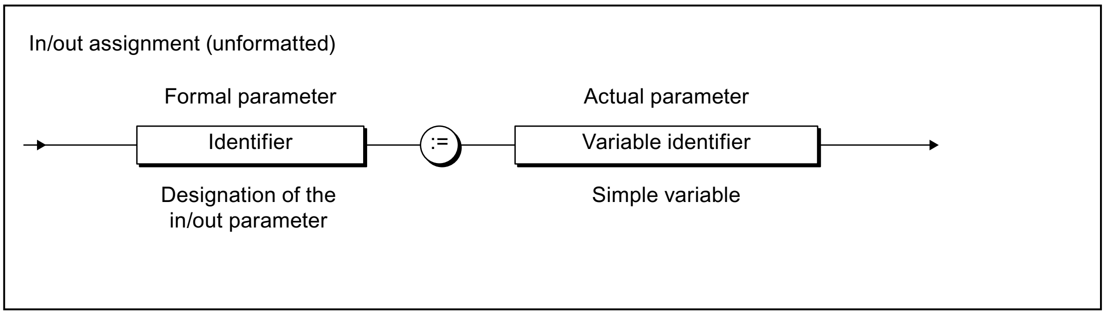


Figure A-93 In/out assignment

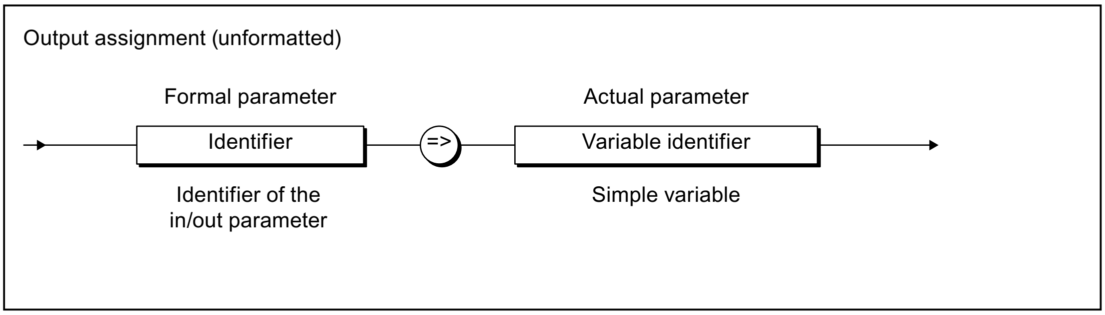


Figure A-94 Output assignment

A.1.3.13 Control statements

Branches

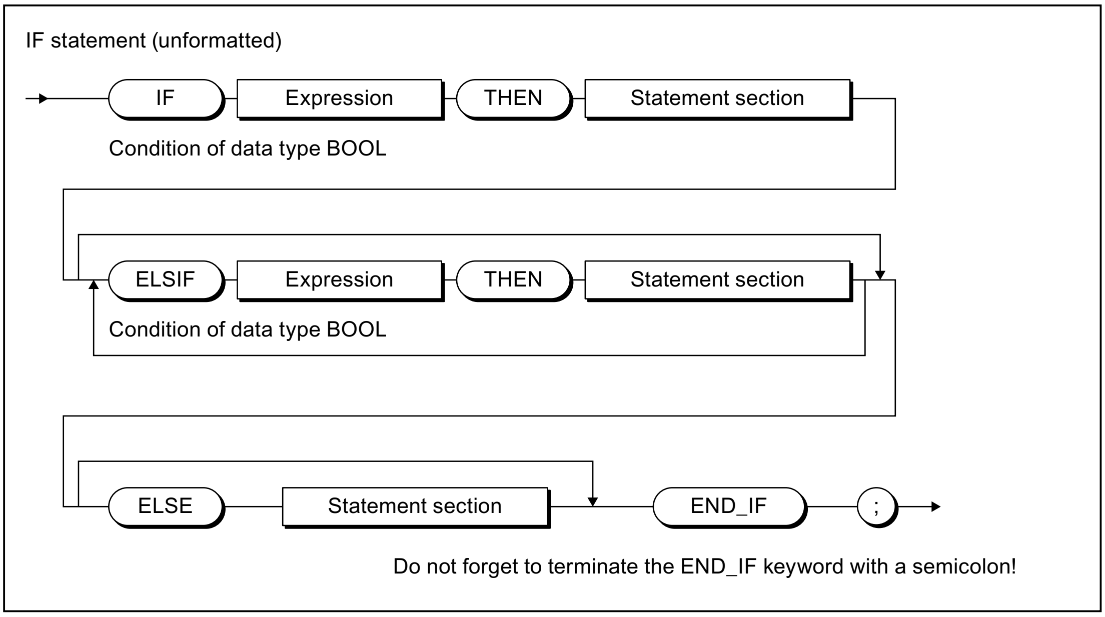

IF statement (unformatted)
IF
Expression
THEN
Statement section
Condition of data type BOOL
ELSIF
Expression
THEN
Statement section
Condition of data type BOOL
ELSE
Statement section
END_IF
;
Do not forget to terminate the END_IF keyword with a semicolon!

Figure A-95 IF statement

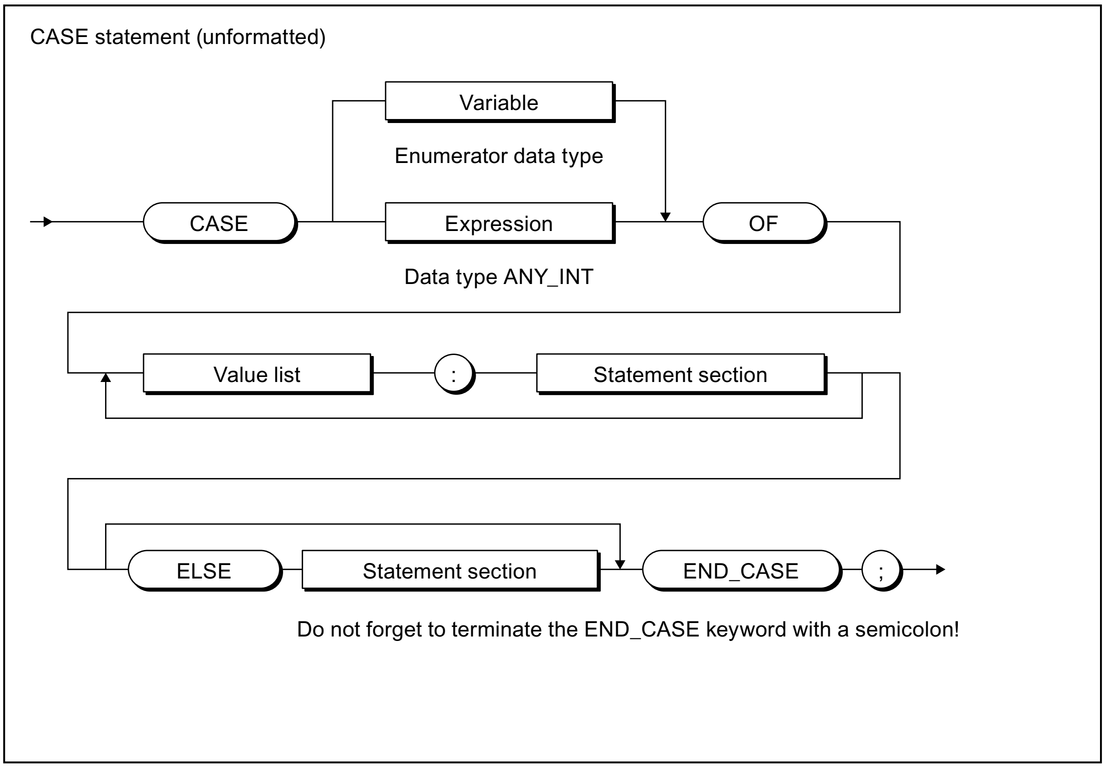


Figure A-96 CASE statement

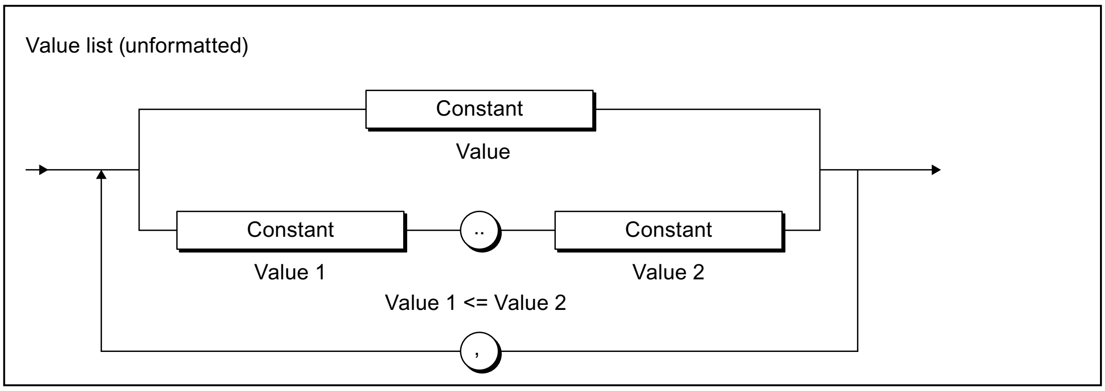


Figure A-97 Value list

Repetition statements and jump statements

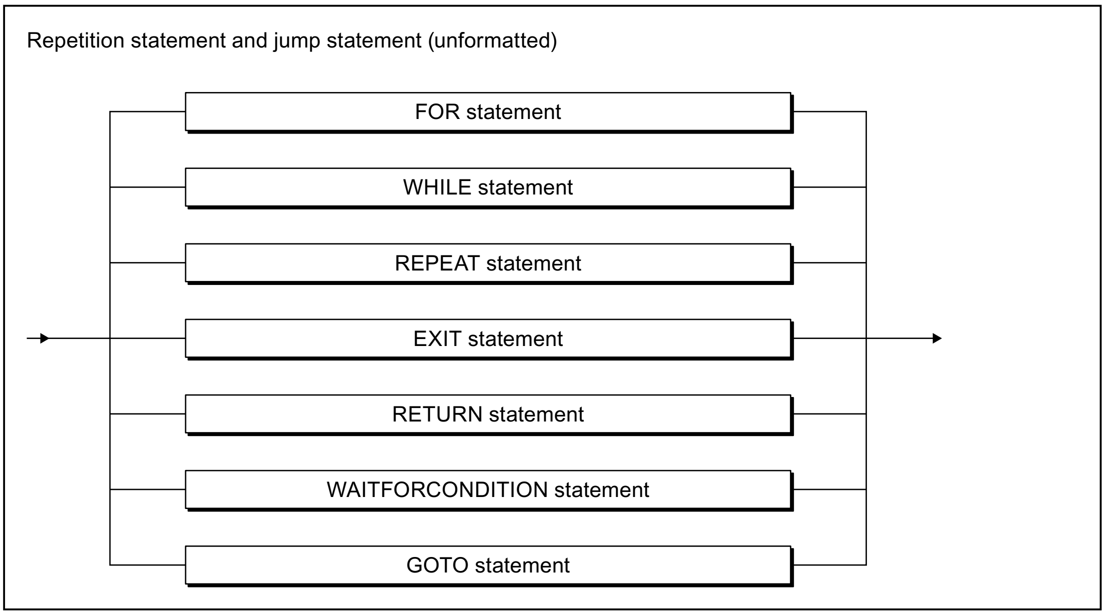


Figure A-98 Repetition statement and jump statements

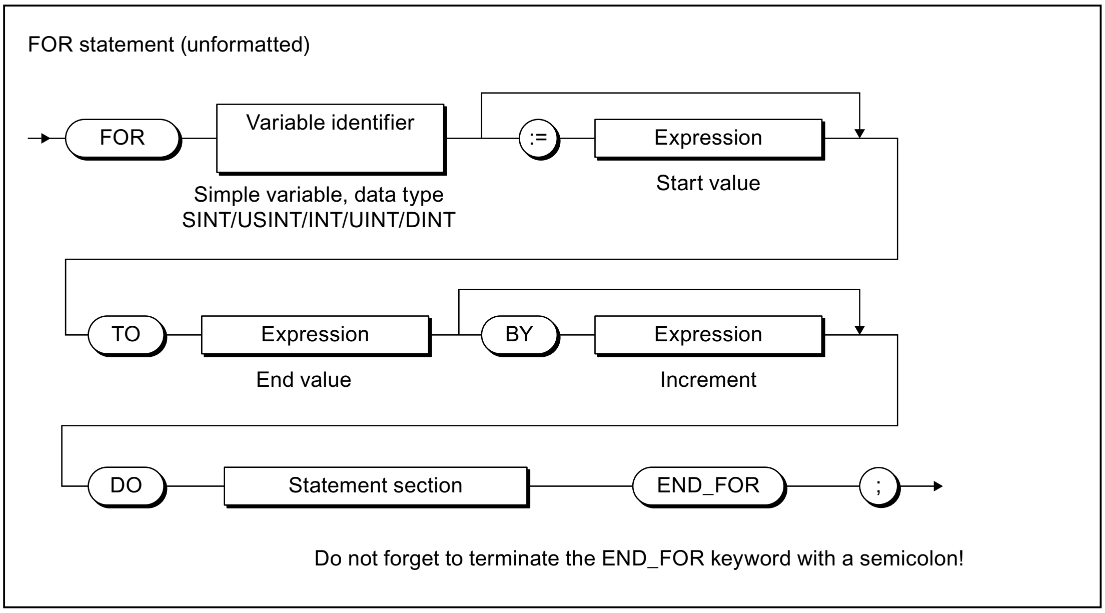

FOR statement (unformatted)
FOR
Variable identifier
Simple variable, data type SINT/USINT/INT/UINT/DINT
:=
Expression
Start value
TO
Expression
End value
BY
Expression
Increment
DO
Statement section
END_FOR
;
Do not forget to terminate the END_FOR keyword with a semicolon!

Figure A-99 FOR statement

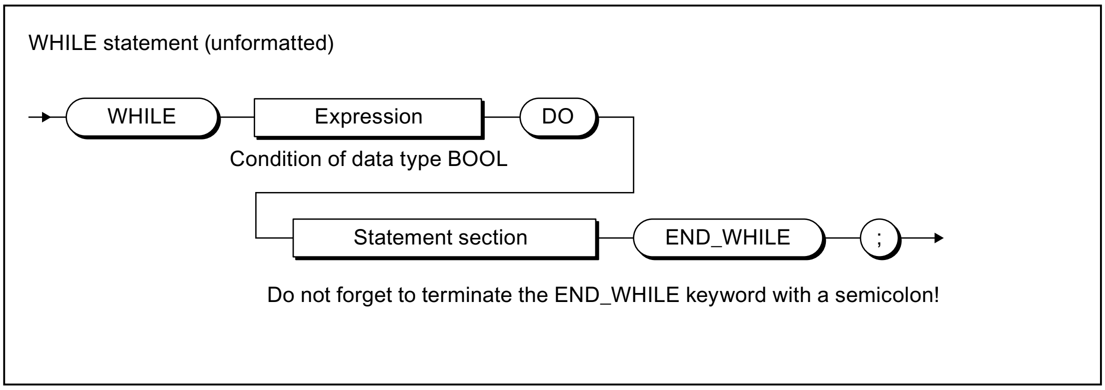

WHILE statement (unformatted)
WHILE
Expression
DO
Condition of data type BOOL
Statement section
END_WHILE
;
Do not forget to terminate the END_WHILE keyword with a semicolon!

Figure A-100 WHILE statement

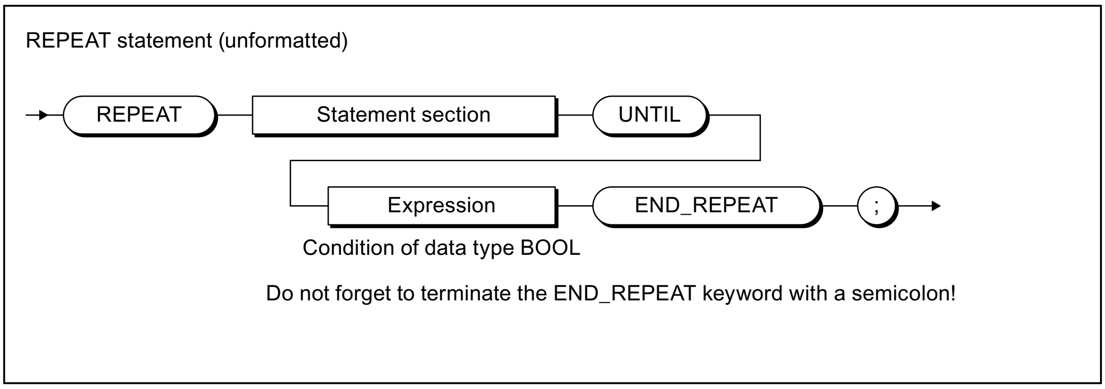

REPEAT statement (unformatted)
REPEAT
Statement section
UNTIL
Expression
END_REPEAT
;
Condition of data type BOOL
Do not forget to terminate the END_REPEAT keyword with a semicolon!

Figure A-101 REPEAT statement

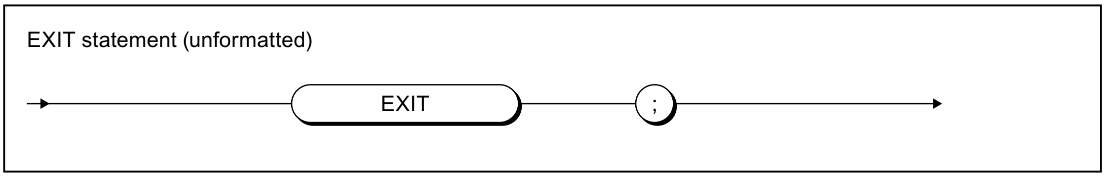


Figure A-102 EXIT statement

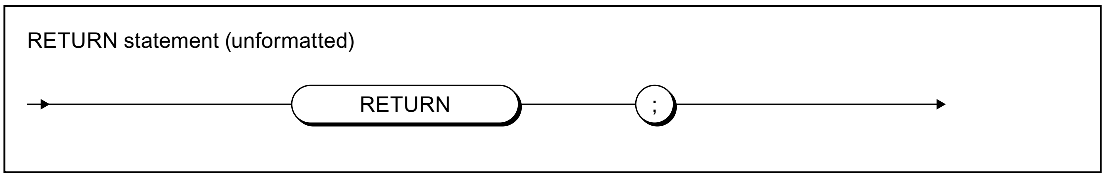


Figure A-103 RETURN statement

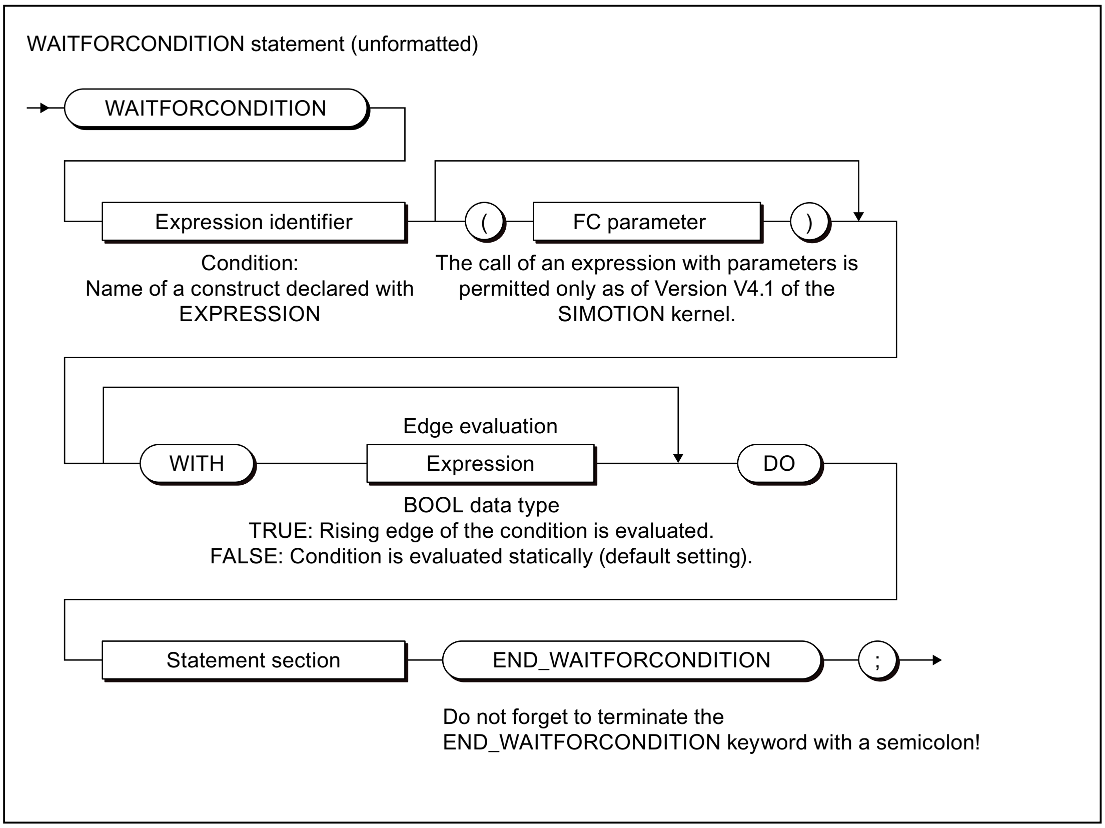

WAITFORCONDITION statement (unformatted)
WAITFORCONDITION
Expression identifier
(
FC parameter
)
Condition:
Name of a construct declared with EXPRESSION
The call of an expression with parameters is permitted only as of Version V4.1 of the SIMOTION kernel.
Edge evaluation
WITH
Expression
DO
BOOL data type
TRUE: Rising edge of the condition is evaluated.
FALSE: Condition is evaluated statically (default setting).
Statement section
END_WAITFORCONDITION
;
Do not forget to terminate the END_WAITFORCONDITION keyword with a semicolon!

Figure A-104 WAITFORCONDITION statement

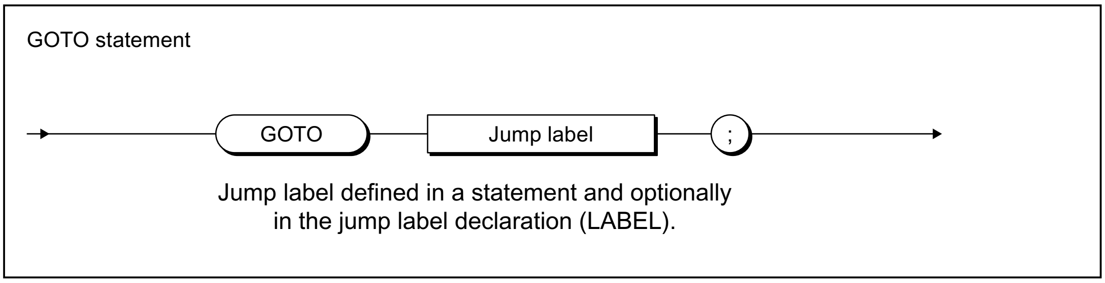

GOTO statement
GOTO
Jump label
;
Jump label defined in a statement and optionally in the jump label declaration (LABEL).

Figure A-105 GOTO statement

A.2 Compiler Error Messages and Remedies

This section provides an overview of the compiler error messages and their correction.

A.2.1 File access errors

Table A-9 File access errors

Error	Description
1000	A read/write error has occurred on file access.
1001	Unable to load the file with the plain text error messages; cannot output error message texts. Please refer to the online help using the error number!
1002	The created code could not be stored. Please close some windows and recompile!
1003	A read/write error has occurred on opening the file. Please close the application and try again!
1100	The option for stating a preprocessor definition contains an invalid identifier as the defined token. The correct syntax of the call option is: -D identifier[=[text]] Examples: • -D myident // Definition of myident; this can be queried using #ifdef. • -D myident= // myident is defined as empty character string • -D "myident=This is a text" // myident is defined as character string 'This is a text'. The quotation marks only have to be used if the replacement text contains a blank.

A.2.2 Scanner errors

Table A-10 Scanner errors (2001 – 2002)

Error	Description
2001	The specified character is illegal.
2002	The specified identifier contains illegal characters or combinations of characters. According to IEC 61131, an identifier must start with a letter or an underscore. Any number of letters, digits, or underscores may follow, but no more than one underscore in a row.

A.2.3 Declaration errors in POU

Table A-11 Declaration errors in POU (3002 – 3027)

Error	Description
3002	Keyword "IMPLEMENTATION" to identify the code section of the load unit is expected.
3003	The specified declaration block is not permitted in this context.
3004	The VAR, VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT, VAR CONSTANT variable declaration blocks are permitted just once for each POU. Up to Version V3.1 of the SIMOTION kernel, the VAR_GLOBAL, VAR_GLOBAL CONSTANT, VAR_GLOBAL RETAIN declaration blocks are permitted just once in the interface or implementation section.
3005	TASK statement: The task link has already been made in the source file for the specified task. Further task linking not possible.
3006	Incorrect stack size for task specified. Only positive integers are permitted.
3007	The specified identifier must be a task identifier; see task configuration.
3008	The specified identifier must be a program identifier. The declaration is made in the statement PROGRAM xx ... END_PROGRAM.
3009	The EXPRESSION keyword must be followed by an identifier. The declaration is made in the statement EXPRESSION xx ... END_EXPRESSION.
3010	The specified identifier is not an EXPRESSION identifier. Check whether the declaration was made using the statement EXPRESSION xx ... END_EXPRESSION.
3011	The TASK statement is not permitted in the unit. Use the task configuration in the Workbench.
3012	The specified identifier has already been declared at another position. It cannot be used again as a function identifier.
3013	The specified identifier has already been declared at another position. It cannot be used again as a function block identifier.
3014	The UNIT statement is expected. The following forms are permissible: • UNIT myunit; • UNIT myunit : dvtype; The UNIT statement is only required when compiling at the ASCII file level. It is optional when the compiler is called from the Workbench.
3015	The source file is not ended with END_IMPLEMENTATION. Observe the structure for a source file!
3016	No further statements may be specified after keyword END_IMPLEMENTATION.
3017	The task declaration is not ended with END_TASK. Observe the structure for a source file!
3018	The POU declaration is not ended with END_FUNCTION, END_FUNCTION_BLOCK, or END_PROGRAM. Observe the structure for a source file!
3019	A POU starting with keywords FUNCTION, FUNCTION_BLOCK, or PROGRAM is expected.
3020	The task linking statement is expected. Configuration: TASK tname ... END_TASK;
3022	The keyword INTERFACE is expected. See the structure for a source file.
3023	Keyword INTERFACE or IMPLEMENTATION is expected. See the structure for a source file.
3024	Syntax error in TASK statement. Correct structure: TASK tname ... END_TASK;
3025	The specified identifier has already been declared at another position. It cannot be used again as a program identifier.

Error	Description
3026	The WAITFORCONDITION statement cannot be used recursively. An attempt was made to use a WAITFORCONDITION statement a second time within a WAITFORCONDITION statement. This is not possible.
3027	An attempt was made to insert a WAITFORCONDITION statement within an EXPRESSION ... END_EXPRESSION block. This is not possible. The WAITFORCONDITION statement cannot be used within an expression.

A.2.4 Declaration errors in type declaration

Table A-12 Declaration errors in type declarations (4001 - 4051)

Error	Description
4001	The specified identifier is a standard function identifier that cannot be overwritten. Choose a different identifier.
4002	The specified identifier has already been used. Use as a type identifier is not possible. Choose a different identifier.
4003	The specified identifier has already been used. Use as a constant identifier is not possible. Choose a different identifier.
4004	The specified initialization value has an incorrect format. Choose the initialization value that corresponds to the data type declaration.
4005	Syntax error in type declaration.
4006	Syntax error in the structure element specification in the structure declaration.
4007	Syntax error in declaration of an ARRAY data type.
4008	Syntax error in the identifier list specification. The identifiers must be separated by commas.
4009	The specified constant identifier has been assigned different values. This occurs when enumeration data types are declared. Identical enumeration elements in different enumeration data types must be located in the same position in the type declaration.
4010	The specified type identifier is not exported from the source file, although the POU in which it is used, is exported. Use a different data type or declare the data type in the implementation section.
4011	A constant declaration requires the specification of an initialization value. Example: x : DINT := 5;
4012	The specified data type must be declared outside the POU. For VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT, the type identifiers must not be declared locally in the POU, as they must also be known outside the POU for parameter transfer purposes.
4013	The specified value is used several times in the enumeration data type. The values in the enumeration data type must differ, however.
4050	The data type or variable declaration creates a data type that is larger than the specified maximum permissible data size.
4051	The variable declaration requires a memory area that is larger than the specified maximum permissible memory size.

A.2.5 Declaration errors in variable declarations

Table A-13 Declaration errors in variable declarations (5001 – 5016, 5100 – 5112, 5500 – 5509)

Error	Description
5001	The specified constant value causes the value range to be exceeded and cannot be converted to the requested type.
5002	The specified identifier has already been used. Use as a variable identifier is not possible. Choose a different identifier.
5003	Syntax error in variable declaration.
5004	The specification of a data type is expected (simple or derived data type).
5005	The specified constant value has the wrong data type or causes the value range to be exceeded.
5006	Check the number of initialization values for array initialization.
5007	Syntax error in the specification of the time and date literals.
5008	A function block instance cannot be created at the specified position. For example, FB instances cannot be created in functions. In addition, output parameters (VAR_OUTPUT) of function blocks cannot be FB instances.
5009	The data type specified in the declaration cannot be applied to the variable with absolute address. An integer or bit data type with matching bit width must be used.
5010	An attempt was made to assign a memory address to a variable. This is not possible at the specified position. Use this assignment only within the VAR_GLOBAL declaration of a unit or within the VAR declaration of a PROGRAM.
5012	The specified variables cannot be preassigned an initialization value.
5014	Incorrect initialization of a data structure. The initialization value for a component was specified more than once.
5016	The initialization of variables and data types with technology objects defined in the project is not possible. Technology objects are themselves variables and so cannot be used for the initialization.
5100	The specified variables cannot be preassigned an initialization value.
5110	Special characters can be specified via $... in the following way: $$, $', $L, $N, $P, $R, $T. Moreover, the numeric value of a character can be specified via $xx, whereby xx stands for the two-digit hexadecimal specification of the character code.
5111	The special character can only be specified via $... . This affects $L, $N, $P, $R, $T
5112	Multi-line character string constants are not permitted. To produce a new line in the output, use the appropriate special character in the character string, e.g. $N, RL.
5500	The specified jump label identifier was already defined. Choose a different name.
5501	The specified jump label identifier has not been defined. Include this identifier in the LABEL declaration.
5502	The jump label identifier has been assigned more than once. However, each jump label can only be used once as a label.
5503	The jump label is specified as a jump destination, but the associated label is missing.
5504	No jumps are possible in subordinate control structures (e.g. WHILE loops). The specified jump label cannot be used at this position.
5505	No jumps are possible in subordinate control structures (e.g. WHILE loops). The specified jump destination cannot be reached.
5506	No jumps are possible in WAITFORCONDITION blocks. The specified jump label cannot be used at this position.
5507	No jumps are possible in WAITFORCONDITION blocks. The specified jump destination cannot be reached.
5509	Jump labels cannot be used within a CASE statement. The syntax does not allow any differentiation between a jump label and the value list of the CASE statement.

A.2.6 Errors in expression

Table A-14 Errors in the expression (6001 - 6140)

Error	Description
6001	Syntax error: A statement terminated with a semicolon is expected, e.g. a := b*c;
6002	Syntax error: An expression is expected, e.g. x < y .
6003	The specified identifier is no variable identifier. You must specify a variable identifier. Check whether the indicated identifier is covered. Up to and including V4.0, access to global device identifiers was possible within a program or function block of the same name despite warning 16021.
6004	The index for array access must be the DINT data type. Perform a suitable type conversion or use another expression.
6005	Type conflict in expression. One of the operands cannot be converted to the data type of the calculation, or the result assignment produces a type conflict.
6006	The specified variable cannot be accessed. Therefore it cannot be used in the expression. Possible causes: • Variable cannot be read. • Attempt to access a local variable of a function or function block from outside.
6007	Cannot write specified variable. A value assignment is not possible.
6008	The specified function does not supply a return value. An application in the expression is therefore not possible (function declared with a return value of VOID).
6009	The specified identifier does not refer to a function or a function block instance. Therefore it cannot be used as function identifier.
6010	The specified identifier is not included as an input parameter (VAR_INPUT) or in/out parameter (VAR_IN_OUT) in the declaration of the POU (function or function block). It cannot be used in the POU call.
6011	The number of function arguments in the call differs from the declaration, or the call parameters required are missing in the call.
6012	RETURN is not permitted syntactically at this position. RETURN may only be used in functions.
6013	EXIT is not permitted syntactically at this position. EXIT can only be used within FOR, WHILE, and REPEAT.
6014	The specified index value is outside the array limits. Only index values that match the array declaration are permissible.
6015	The specified task control command cannot be applied to the task. It is not allowed for this type of task.
6016	The specified task is deactivated in the execution system. It must be enabled before it can be used.
6017	Syntax error on specifying programs within a task. The programs must be listed by name and separated by commas.
6018	The specified identifier does not refer to a PROGRAM. Therefore it cannot be used as a program identifier.
6019	Multiple assignment of program to task. Only one assignment is possible.
6020	Syntax error on specifying directly displayed variables. Inputs must have the syntax %Ix.y and outputs the syntax %Qx.y.
6021	The specified byte offset of the directly displayed variables lies outside the permissible address space.
6022	The specified byte offset of the directly displayed variables lies outside the permissible address space. Values 0 to 7 are permissible.
6023	The return value of the function was not assigned. An assignment is however imperative.
6024	A variable with the specified identifier is not included in the task start information.

Error	Description
6025	The condition variable and condition values of a CASE statement must be of the data type SINT, INT, DINT, USINT, UINT or UDINT. It must be possible to implicitly convert the condition values to the data type of the condition variables.
6026	The specified message identifier is not contained in the message configuration. Switch to the message configuration and add the identifier.
6027	System variable access is only possible directly by means of a technology object reference. Access by means of a structure or array is not possible. Create a local variable of type TO and assign the TO reference to this variable. You can then access the required system variable by means of this local TO variable.
6028	Type conflict in expression at specified operation. One of the operands cannot be converted to the data type of the calculation, or the result assignment produces a type conflict. The specified data type in the expression is expected.
6029	The specified function parameter does not have a default value, so it is imperative to specify a value when the function is called.
6030	An attempt was made to transfer an expression to an in/out parameter (VAR_IN_OUT). This is not possible. User variables must be specified as in/out parameters.
6031	An attempt was made to transfer a system variable (TO, I/O direct access) to an in/out parameter (VAR_IN_OUT). This is not possible. User variables must be specified as in/out parameters.
6032	An attempt was made to transfer a variable in the process image to an in/out parameter (VAR_IN_OUT). This is not possible. User variables must be specified as in/out parameters.
6033	An attempt was made to transfer a variable with a non-matching data type to an in/out parameter (VAR_IN_OUT). However, an Implicit type conversion is not possible. User variables with the correct data type must be specified as in/out parameters.
6034	An attempt was made to transfer a read only variable to an in/out parameter (VAR_IN_OUT). This is not possible. In/out parameters must be read/write.
6035	An attempt was made to transfer a constant to an in/out parameter (VAR_IN_OUT). This is not possible. In/out parameters must be user variables.
6036	An operation is applied to a constant. The value of the constant is outside the definition range for the function. Examples are: • Application of SQRT to a negative number. • Use of logarithmic functions on a number <= 0. • Use of ASIN or ACOS on a number outside the interval [0..1]
6037	An attempt was made to divide a constant by zero. This operation is not permitted.
6038	The specified function parameter occurs more than once in the argument list.
6039	The specified POU (function or function block) cannot be used. Possible causes: • The definition of the POU in the implementation section is missing. Only the prototype was specified in the interface section. • The POU is fully defined only after its use (e.g. call, instance declaration). If necessary, move this POU in the program source before the POU in which it is used. • An instance of the function block cannot be declared as unit variable in the same program source in which this function block is defined.
6040	Only simple variables may be used as semaphores; indexing is not possible.
6041	The message function requires an auxiliary value of the specified data type. Type conversion is not possible.
6042	The message function requires that you specify a message number. The specified message number is invalid.
6050	Type conflict in expression at specified operation/variable. One of the operands cannot be converted to the type of the calculation, or the result assignment produces a type conflict. A conversion between source file type and target type is not possible.

Error	Description
6051	The expression contains a type conflict for the specified operation. One of the operands cannot be converted to the data type of the other operand to perform the calculation, or the operand data types are not permitted for this operation.
6052	Type conflict in expression. The specified data type cannot be used for the operation (see marshalling functions).
6053	The expression contains a type conflict for the specified operation. This operation is not permissible on the specified data type.
6054	Type conflict in expression. The specified variable cannot be used as indexed array variable.
6060	At the function call, there is a mixture of assignments of function arguments and setting parameters. Use one form of the function call. Example: • f (x, y); or • f (in1 := x, in2 := y);
6061	The specified parameter of the function or the function block is an in/out parameter. Consequently, a variable must be assigned when the POU is called.
6062	The specified identifier cannot be used as a function argument. Only variables from the declaration blocks VAR_INPUT and VAR_IN_OUT are permitted.
6070	Access to configuration data is only possible for variables that have been specified completely. Append the name according to the configuration data for the selected technology object.
6080	The specified variable is no input or output variable that can be directly accessed. Such a variable must be declared in the I/O container of the respective device; it must have the syntax PI* or PQ*.
6100	The specified construct can only be compiled if the device type is set. Add the device type to the unit statement or set the device type in the program container.
6110	The specified construct cannot be used in libraries.
6111	The specified construct cannot be used in libraries.
6112	The specified construct cannot be used in libraries.
6113	Access to technology objects and devices is not allowed in libraries.
6130	The specification of an interval is not permissible for the data type indicated in the CASE statement.
6140	The specification of a constant in ENUM_TO_DINT requires specifying the data type in the form of enum_type#value.
6150	The specified bit offset lies outside the valid range for the specified data type.
6200	Only for "Permit language extensions" compiler option (-C lang_ext): The called PROGRAM contains instance data (VAR … END_VAR declaration block) stored in the user memory of the assigned task. This means a call of the PROGRAM from another POU is not possible. Compile the source file with the "Create program instance data only once" compiler option (-C prog_once) or remove the instance data.
6201	Only for "Permit language extensions" compiler option (-C lang_ext): The call of a PROGRAM is not supported in functions. Such calls can be made only in function blocks or another PROGRAM.

A.2.7 Syntax errors, errors in expression

Table A-15 Syntax errors, errors in the expression (7000 - 7014)

Error	Description
7000	A syntax error has occurred. Possible causes: • Incorrectly ended control structures (e.g. END_IF missing) • Statements not terminated with ; • Missing parentheses
7001	The specified identifier does not refer to a constant. Please enter one constant per value or identifier.
7002	A signed integer is expected. The integer can be of data type SINT, INT, or DINT.
7003	When specifying the interval, the initial value must be less than or equal to the end value. This applies to the declaration of arrays and the specification of the interval in CASE selection conditions.
7004	An initialization value is expected. The value must be a constant. Constants can be assigned as follows: • Directly per value • Symbolically via a preceding constant declaration • As an expression containing constants only
7009	An expression that supplies data type BOOL is expected as condition for WHILE, REPEAT, and IF. This can be specified as a variable of data type BOOL or via a comparison expression. You can also specify a function with a return value of data type BOOL.
7010	A syntax error has occurred. Possible causes: • Incorrectly terminated control structures (e.g. END_IF missing) • Statements not terminated with ; • Missing parentheses
7011	A syntax error has occurred. Possible causes: • Incorrectly terminated control structures (e.g. END_IF missing) • Statements not terminated with ; • Missing parentheses
7012	A syntax error in the statement, that starts at the specified line, has occurred. Possible causes: • Incorrectly terminated control structures (e.g. END_IF missing) • Statements not terminated with ; • Missing parentheses
7013	A syntax error has occurred. An illegal construct is being used.
7014	A syntax error has occurred. Possible causes: • Incorrectly terminated control structures (e.g. END_IF missing) • Statements not terminated with ; • Missing parentheses

A.2.8 Error when linking a source file

Table A-16 Error when linking a source file (8001, 8100)

Error	Description
8001	The specified POU has been exported to the INTERFACE section, but an IMPLEMENTATION section is missing. Either delete the export statement or specify a valid implementation.
8100	The maximum size of the data area that can be reached using HMI is 65536 bytes. This limit has been exceeded with the specified variable. All subsequent variables cannot be reached either.

A.2.9 Errors while loading the interface of another UNIT or technology package

Table A-17 Errors while loading the interface of another UNIT or a technology package (10000 - 10037, 10100 - 10101)

Error	Description
10000	The specified unit has an invalid file format. Probably, the unit was created using an older version of the compiler or compiled using incompatible options. If a unit is involved, it should compiled first. Then repeat the current compilation. If a package is involved, a newer version should be installed.
10001	The unit name has an invalid format. The rules for identifiers in ST are also true for unit names; the following restrictions apply to their length: • Up to Version V4.0 of the SIMOTION Kernel: 8 characters. • As of Version V4.1 of the SIMOTION Kernel: 128 characters.
10002	Error while loading the interface of another UNIT, a library or technology package. The specified identifier is contained in two different imported units, libraries or technology packages. • Remove a unit, library or technology package from the import list or • Establish uniqueness between the identifiers in imported units, libraries or technology packages. Change the exporting units in the interface section or specify a namespace for a library or a technology package (USELIB … AS namespace; USEPACKAGE … AS namespace;).
10003	The specified data type has an invalid memory layout. Probably, the unit was created using an older version of the compiler or compiled using incompatible options. If a unit is involved, it should compiled first. Then repeat the current compilation. You can also perform "Save and recompile everything". If a package is involved, a newer version should be installed. If the error persists, inform the support department.
10004	The exported identifiers of the specified unit could not be loaded. Close some applications and try again.
10005	A recursion was detected on loading packages. The specified package has already been loaded with USEPACKAGE and cannot be specified a second time.
10006	A recursion was detected on loading the unit. The specified unit has already been loaded with USES and cannot be specified a second time.
10007	The maximum number of imported units which can be referenced in a unit was exceeded. A maximum of 223 imported units per load unit are permissible. Both units imported directly with USES and indirectly imported units are counted.
10008	The number of imported packages that can be referenced in a unit has been exceeded. A maximum of 127 imported packages per load unit are permissible.

Error	Description
10009	The specified package is used in the unit, but it is not available on the device. This error message occurs when you compile with the "implicit package utilization" option and have programmed a USEPACKAGE statement that has a different content than the packages specified on the device.
10010	The specified package is used in Unit a but not in Unit b. This error message occurs when different packages have been specified with USEPACKAGE in units that reference each other with USES. Correct the USEPACKAGE statements.
10011	The specified unit is used directly or indirectly by itself via one or more units. Correct the USES statements.
10012	The specified unit is imported directly or indirectly into several units in different compilation versions. Recompile all units that reference the specified unit in the USES statement.
10013	The specified unit has not yet been compiled, or an error occurred during the last compilation. Compile this unit first to ensure successful compilation.
10014	The type of specified technology object (TO) is not supported by the package specified previously during compilation with USEPACKAGE. Use a package that contains the TO type.
10015	The maximum number of technology objects (TO) which can be referenced in a unit was exceeded. A maximum of 65535 TOs can be referenced.
10016	The device type parameter is not available. If the unit to be compiled is not to be assigned to a device, use the statement UNIT xx : dvtype;
10017	The device type has not been specified uniquely. In the unit, the statement UNIT xx : dvtype; specifies a different device type than the one determined via the assignment of the unit to the device.
10018	The specified unit could not be found. Check whether the unit name is available in the PROGRAM container of Workbench or whether the specified file is contained in the current working directory (only u7bt00ax - command line).
10019	The specified technology package could not be found. Observe the preceding error outputs.
10020	Error occurred while loading the technology package. Observe further error outputs.
10021	The technology package is used in the specified source file, however, it is not selected on the device. Correct the USEPACKAGE statement, or select the technology package on the device.
10022	The specified technology package is being used with different versions. Correct the settings for the technology package selection on the device and, if required, in the library. Only one version of a technology package can be used on a device.
10030	The device type has not been specified uniquely. In the unit, the statement UNIT xx : dvtype; specifies a different device type than the one determined via the assignment of the unit to the library container.
10031	The specified library is used directly or indirectly by itself via one or more libraries. Correct the USELIB statements.
10032	The specified library could not be found. Check your project.
10033	A recursion was detected on loading the library. The specified library has already been loaded with USELIB and cannot be specified a second time.
10034	The specified library is not completely compiled. Possible causes: • The library has not yet been compiled. • The library has not been compiled for all device types specified for the library container (e.g. in project-wide compilation). • An error occurred in the last compilation. First compile this library individually (accept and compile).
10035	The specified library could not be found. Check whether the library name is available in the Workbench project or whether the specified file is contained in the current working directory (only u7bt00ax command line).

Error	Description
10036	The specified package is used in the source file, but it is not available in the library. Libraries are generally compiled against the package versions specified in the library container. You have programmed a USEPACKAGE statement that has a different content than the packages specified in the library. Either select the correct package version or remove the USEPACKAGE statement from the source file.
10037	The code variant for the current device type is not selected for the specified library. This means this library cannot be used. Activate the code variant for this library.
10100	The specified type of a technology object is contained in several packages that were referenced by the source file. Please choose the technology package that meets your requirements.
10101	The specified technology object is not compatible with the types of technology objects supported by the loaded packages Update the package or change the type of technology object.

A.2.10 Implementation restrictions

Table A-18 Implementation restrictions (15001 – 15200)

Error	Description
15001	The specified construct is not supported by the current version of the compiler.
15002	The currently selected device does not support the specified function. Select a different device version if you want to use this function. To do so, replace the CPU in the hardware catalog and, if necessary, update the firmware.
15003	The specified identifier is a keyword that is not supported and therefore cannot be used as user-specific in order to ensure compatibility with later compiler versions.
15004	The specified identifier denotes a standard function that is not supported and cannot be used as user-specific identifier in order to ensure compatibility with later compiler versions.
15005	The specified identifier denotes a non-supported standard function and cannot be used as user-specified identifier in order to ensure compatibility with later compiler versions.
15006	The specified construct can only be used in source files generated with MCC. Usage in ST is not possible.
15007	A source/library/package is used in the implementation section either directly or indirectly without specifying a namespace. In the interface section, it is used with a namespace. Solve this conflict by specifying a namespace in the interface section for the specified source/library/package.
15070	The specified construct does not conform to the language standard, however, for compatibility reasons, is not supported for old platforms. Convert the usage to the specified alternative.
15152	A USES, USELIB, or USEPACKAGE statement was found in a source file section hidden by conditional compilation. This is illegal. Source file sections that contain these statements cannot be complied conditionally.
15153	The specified definition is not considered during code generation. It is not possible to define keywords differently.
15200	The specification of a bit offset for a bitstring variable requires the "Permit language extensions" compiler option (-C lang_ext).

A.2.11 Warnings

Table A-19 Warnings (16001 - 16602)

Error	Description
16001	(Warning class: 0) Only in conjunction with the "Selective Linking" compiler option. The specified function, the function block, or the program are neither exported nor called in the current unit. No code is generated.
16002	(Warning class: 0) Only in conjunction with the "Selective Linking" compiler option. The specified unit does not contain any exported PROGRAM nor any task link. No code is generated for the unit.
16003	(Warning class: 2) The operands of the comparison operation do not contain any explicit type definition. The data type listed in the comparison can be seen in the warning message issued. Specify the data type of the used constants explicitly with <type>#<value>.
16004	(Warning class: 2) The specified type conversion may cause the variable value to change due to the reduced display width or inadequate accuracy of the target data type.
16005	(Warning class: 2) During type conversion, the dependency of the variable value can cause the sign to change.
16006	(Warning class: 2) The specified value will be rounded to the next displayable value due to insufficient display width.
16007	(Warning class: 2) A loss of accuracy occurred during type conversion. Not all decimal places are considered.
16008	(Warning class: 2) A loss of accuracy occurred during initialization of the specified variables. The constant will be converted to the specified data type. Not all decimal places are considered.
16009	(Warning class: 0) Only in connection with compiler option Selective Linking. The specified unit does not contain any exported PROGRAMs or any task linking. Unable to access unit code. Unable to call relevant POU.
16010	(Warning class: 0) Specified program not exported to unit; therefore unable to use it in configuration of the execution level.
16011	(Warning class: 0) The source file does not contain any exported global variables. No data are loaded to the target system.
16012	(Warning class: 0) The specified source file name was taken over from the PROGRAMS container of the selected device. The identifier of the source file in the UNIT statement was ignored.
16013	(Warning class: 2) Because of the marshalling function, the specified data type is not portably convertible. Only use SIMOTION devices in connection with this data type, or perform an explicit conversion of the data type.
16014	(Warning class: 2) With the specified operation, a data type conversion is performed between signed and unsigned. Because the bit string is adopted in this case, the resulting numerical value can differ from the specified value.

Error	Description
16015	(Warning class: 2) For the assignment of the character string constants to the variables, only part of the character string constants is transferred, because the length of the variable is insufficient to accept all characters.
16016	(Warning class: 2) The operands in the expression do not contain any explicit type definition. The data type of the operation is determined by specifying the values. The resulting data type in which the expression is calculated can be seen in the issued warning message. To define the data type: • Specify the data type of the used constants explicitly with <type>#<value>. • Use an explicit data type conversion.
16017	(Warning class: 2) The operands in the expression contain only constants. The data type of the operation can be determined by specifying the data type (in the form <type>#<value>) or explicit data type conversion. This output is used for finding problems, in particular, for the use of symbolic constants, because the data type of the operation cannot normally be determined easily.
16018	(Warning class: 2) The data type of the comparison operation is defined using the value of a constant that has a larger value range than the contained variable. The comparison is performed with the data type of the constant.
16020	(Warning class: 1) The declaration hides the specified identifier, which has been globally defined in its own source file or an imported source file. Access to the global identifier is no longer possible from the POU where this identifier is declared locally.
16021	(Warning class: 1) The declaration hides the specified identifier, which is defined on the device. You can access the global device identifier with _device.<name>.
16022	(Warning class: 1) The declaration hides the specified identifier, which is defined in the project (e.g. technology object or device). You can access the global project identifier with _project.<name>.
16023	(Warning class: 1) The declaration hides the specified identifier for the data type of a technology object. Access to the data type identifier is no longer possible.
16024	(Warning class: 1) The declaration hides the access to the technology object on the device. You can access this TO with _to.<name>.
16025	(Warning class: 1) The declaration hides the IEC standard function with the identical name. Access to this function is no longer possible in the current context.
16026	(Warning class: 1) The specified identifier is reserved by SIEMENS for potential extensions. The use of this identifier can cause compiler errors in later versions. If you want to avoid this, change this identifier.
16030	(Warning class: 1) A label has been specified several times in a CASE statement. Only the first label is ever evaluated. Other specifications have no effect.
16102	(Warning class: 3) The option for output of code for the program status diagnosis function is ignored because no debug information was generated. Output of debug information was deactivated via compiler options.

Error	Description
16103	(Warning class: 3) The option for outputting code at the library for the program status diagnosis function is ignored. The code for program status is generated as defined in the option in the individual source files.
16150	(Warning class: 7) A new definition has been made for the specified identifier. Consequently, the previous definition is invalid. This warning enables the work of the preprocessor to be tracked.
16151	(Warning class: 7) An attempt has been made to delete the definition of the specified identifier with #undef. However, the identifier is not defined or the definition is already deleted. This warning enables the work of the preprocessor to be tracked.
16152	(Warning class: 7) The specified definition is not considered during code generation. The cause for this can be that the preprocessor is deactivated for the compiled source.
16153	(Warning class: 7) The preprocessor is not active in the current source, even though preprocesssor statements are used. Activate the preprocessor or remove the statements.
16170	(Warning class: -) The definition from sources imported using USES are not considered during the code generation.
16171	(Warning class: -) The definition from the specified sources imported using USES could not be loaded. Compile the specified source file beforehand.
16200	(Warning class: 4) The use of a semaphore requires a global variable to enable access to it from a different task. Local task operations do not have to be blocked via semaphores.
16210	(Warning class: 4) The basis of the exponential function (EXPT standard function or ** operator) is negative. The operation can be executed at run time only under the following conditions: 1. It can be used on a device with a version of the SIMOTION kernel as of V4.1. 2. The exponent is an integer. The ExecutionFaultTask will be initiated for non-integer exponents or for use on a device with a version of the SIMOTION kernel up to V4.0. The program will be aborted here.
16220	(Warning class: 4) The condition of an IF statement, WHILE statement or REPEAT statement is a constant expression.
16230	(Warning class: 4) The expression with the specified values does not cause any change to the result; optimized code will be created.
16240	(Warning class: 4) The expression with the specified values exceeds the definition range of the operation. The result may be incorrect.
16300	(Warning class: 5) The auxiliary value has a data type that cannot be converted to the data type configured for the message.
16301	(Warning class: 5) The specified auxiliary value is not evaluated during output of the message.

Error	Description
16302	(Warning class: 5) The data type of the auxiliary value cannot be determined from the message configuration. The specified data type is used.
16303	(Warning class: 5) No auxiliary value has been specified for the function although the message configuration requires such a value. A default value of the corresponding data type was added.
16304	(Warning class: 5) An alarm accompanying value is specified using a constant or a constant expression. The resulting data type of the alarm accompanying value can be seen in the issued warning message. To define the data type: • Specify the data type of the used constants explicitly with <type>#<value>. • Use an explicit data type conversion.
16400	(Warning class: 6) A global variable has been declared in a library. This may mean that the library cannot be used more than once.
16420	(Warning class: 6) The return value has not been assigned within the function. If such a function is called, it returns a random value.
16421	(Warning class: 6) A variable that has neither been assigned nor read in the code has been declared.
16450	(Warning class: –) A global variable has been created in the retentive memory range. This declaration is not permissible at the specified position.
16451	(Warning class: –) The initialization of large arrays with values other than 0 causes a high data volume in the controller. This results in long load times as well as high memory utilization.
16452	(Warning class: –) The specified program has a large quantity of instance data to be initialized. This can lead to a runtime violation when the task is started because both the initialization code and the user code are being executed. In particular, caution is advised in the case of SynchronousTasks.
16470	(Warning class: -) The specified construct does not conform to the language standard, however, for compatibility reasons, is not supported for old platforms. Convert the usage to the specified alternative.
16600	(Warning class: 6) The specified variable is not contained in the initialization list. The default initialization value is used.
16601	(Warning class: 6) The specified variable is not contained in the initialization list. The default initialization value is used.
16602	(Warning class: 6) The specified variable is not contained in the initialization list. The default initialization value is used.

A.2.12 Information

Table A-20 Information

Error	Description
32010	(Warning class: 6) The specified jump label identifier has been declared but not used.
32020	(Warning class: –) The specified variable was declared globally in this source file or in another source file with the indicated data type. This information helps when searching for the cause of compilation errors. It is issued together with error messages.
32021	(Warning class: –) The specified variable was declared on the device as an I/O variable, a global device variable, or a system variable. This information helps when searching for the cause of compilation errors. It is issued together with error messages.
32022	(Warning class: –) The specified variable was declared in the project as a global identifier. This information helps when searching for the cause of compilation errors. It is issued together with error messages.
32023	(Warning class: –) Until now, no valid declaration has been found for the specified identifier. This information is issued together with error messages.
32024	(Warning class: 0) The specified variable has been declared as a global identifier in the current unit or in an importing unit. This information helps when searching for the cause of compilation errors. It is issued together with error messages.
32030	(Warning class: 0) The specified array initialization does not conform to IEC 61131-3. For portable programs, the array initialization values should be placed into square brackets. Example of field initialization in compliance with the standard: x : ARRAY [0 to 1] OF INT := [1, 2];
32050	(Warning class: 0) The maximum size that can be reached via an HMI is 65536 bytes. This limit has been exceeded with the specified variable. All subsequent variables cannot be reached either.
32300	(Warning class: 1) A label has been specified several times in a CASE statement. Only the first label is ever evaluated. Other specifications have no effect.
32650	(Warning class: 7) The specified identifier will be replaced thereafter by the output text. This information enables the work of the preprocessor to be tracked.
32651	(Warning class: 7) The definition of the specified identifier has been deleted with #undef. This information enables the work of the preprocessor to be tracked.

Error	Description
32652	(Warning class: 7) The identifier will be used with the specified replacement text in the source file. Compilation takes place with the replacement text. This information enables the work of the preprocessor to be tracked.
32653	(Warning class: 7) The specified identifier will be replaced thereafter by the output text. This information appears if additional replacements are loaded with a USES statement. This information enables the work of the preprocessor to be tracked.

A.3 Template for Example Unit

A.3.1 Preliminary information

This appendix presents a comprehensive annotated template that you can call in the online Help. You can use it as a template for a new ST source file.

```
//----------------------------------------------------------------------------
//    Notes for the INITIALIZATION of the user data are available
//    at the end of the template
//----------------------------------------------------------------------------
INTERFACE
//    All statements added between INTERFACE and END_INTERFACE/
//    Keywords are used to define which source contents
//    (variables, functions, function blocks, etc.) also in other
//    sources (units) are available or exported.

    USEPACKAGE cam;
    // The technology packages to be used are known here and thus
    // made usable in the source. Technology object (TO)-specific
    // Commands can be used in this UNIT only when the
    // appropriate package has been included.
    // If a source file that uses USEPACKAGE cam is integrated via USES,
    // it will be "inherited". USEPACKAGE can then be omitted.
    // The package used in this example is "cam". However, other
    // technology packages can also be used (see documentation).

    // USELIB testlib;

    // If library functions are to be used in the source file, they must be made
    // known in the source, too. If the library
    // with the name "testlib" does not exist in the project,
    // the error message
    // "Error 10035, "testlib.lib" library could not be loaded"
    // "Error 10032, "testlib" library could not be loaded"
    // will be output.
    // If libraries are not being used, this line can be
    // deleted..

    // USES header;

    // USES is used to import contents exported from a different source
    // (NAME here "header") and made usable in "Template".
    // If the source with the name "header" does not exist in the project,
    // the error message
    // "Error 10018, "header" source could not be loaded"
    // will be output. In this case, the NAME of an existing source file must be
    // used in place of "header".
```

A.3.2 Type definition in the interface

```
// ********************************************************
// * Type definition in the INTERFACE                   *
// ********************************************************

VAR_GLOBAL CONSTANT
    PI : REAL := 3.1415;
    ARRAY_MAX : INT := 3;
END_VAR
// Declaration of a global constant. In the source file
// no other value can be assigned to the identifier.

// User defined variable types (UDT) are
// defined between TYPE and END_TYPE.
TYPE
    array1dim : ARRAY [0..ARRAY_MAX] OF INT;
    // Definition of a one-dimensional array with four array elements from
    // type INT under the name "array1dim". With "array1dim" as the data type
    // in all source file segments, one-dimensional arrays can now
    // be declared by type INT.

    array2dim : ARRAY [0..3] OF array1dim;
    // A two-dimensional array is an array of one-dimensional arrays.
    // Here a two-dimensional field with 16 elements occurs
    // of the type INT under the name "array2dim"

    enumTrafficLight : (RED, YELLOW, GREEN);
    // Definition of enumerator "enumTrafficLight" as a
    // user-defined variable type. Variables of this type can
    // only accept the values "RED", "YELLOW", and "GREEN".

    structCollection : STRUCT
        toAxisX : posaxis;
        aInStruct1dim : array1dim;
        eTrafficInStruct : enumTrafficLight;
        iCounter : INT;
        bStatus : WORD;
    END_STRUCT;
    // A user-defined structure is created here. It is possible to
    // combine elementary data types (here INT and WORD) or already defined
    // user data types (here "array1dim" and "enumTrafficLight") into
    // one structure. In addition, types
    // of technology objects can also be used.
    // In the example, the structure contains an element of
    // a positioning axis (posAxis).
    // In the definition, make certain to sort the variables
    // by size in increasing sequence
    // (ARRAY, STRUCT, LREAL, DWORD, INT, BOOL ...)

    arrayOfStruct : ARRAY [0..5] OF structCollection;
    // Nesting is also possible. The type "arrayOfStruct"
    // contains a field comprising six elements of type "structCollection"
END_TYPE
```

A.3.3 Variable declaration in the interface

```
// ****************************************************
// * Variable declaration in the INTERFACE            *
// ****************************************************

VAR_GLOBAL        // In the user memory of the UNIT.
                  // Also visible using HMI services.

    g_aMyArray : ARRAY [0..11] OF REAL := [3 (2(4), 2(18))];
    // Example of a declaration of a one-dimensional array without
    // previous type declaration. The initialization performed here is
    // read as follows:
    // Two elements each are initialized with the value 4,
    // two elements with the value 18. This pattern is used in the field

    // "g_aMyArray" three times in succession.
    // The field elements are thus assigned as follows:
    // 4, 4, 18, 18, 4, 4, 18, 18, 4, 4, 18, 18.

    g_aMy2dim : array2dim;
    // Example of a declaration of a two-dimensional array

    g_aMy1dim : array1dim;
    // Example of a declaration of a one-dimensional array with
    // use of a type declaration.

    g_sMyStruct : structCollection;
    // Variable of the type or with the structure of
    // user_struct.

    g_aMyArrayOfStruct : arrayOfStruct;
    // The variable generated here contains a field from
    // structural elements as declared in section TYPE/END_TYPE


    g_tMyTime : TIME := T#0d_1h_5m_17s_4ms;
    // ...as elementary time types and derived data types (see below).

    g_eMyTraffic : enumTrafficLight := RED;
    // An enumerator of type "enumTrafficLight" is created here and
    // assigned the value "RED".

    g_iMyInt : INT := -17;
    // Variables of an elementary numerical data type can
    // also be declared in variable declarations...

END_VAR

VAR_GLOBAL RETAIN
END_VAR
// The variables declared with the add-on RETAIN are
// stored in the RETAIN data area of the hardware platform used and
// are therefore safe from network failure.
// The declaration of VAR, VAR CONSTANT, VAR_TEMP, VAR_INPUT, VAR_OUTPUT
// and VAR_IN_OUT is not permissible here.
// Variables that are defined in this section and thus exported
// can be reimported by means of the USES "template" into another source file (UNIT)
```

```
    FUNCTION FC_myFirst;
    FUNCTION_BLOCK FB_myFirst;
    PROGRAM myPRG;
    // The function blocks (FBs),
    // functions (FCs) and programs defined in the implementation part are exported here
in the interface part,
    // so that they can be used in other units.
    // Non-exported FBs and FCs can only be used in this source file
    // ("information hiding", placing in the interface only
    // what other units absolutely need).
    // A program that has not been exported cannot be assigned to any TASK
    // deleted..
END_INTERFACE
```

A.3.4 Implementation

```
    // ****************************************************
    // * IMPLEMENTATION section                           *
    // ****************************************************

IMPLEMENTATION
    // In the IMPLEMENTATION section of a unit, the executable code sections
    // are stored in various program organization units (POUs).
    // A POU can be a program, FC, or FB.

    VAR_GLOBAL CONSTANT
    END_VAR

    TYPE
    END_TYPE
    // The type definition can also be made in the IMPLEMENTATION section.
    // However, this definition cannot be imported in another source file.
      The type definition can, however, be used for variables
    // in all POUs of the source file "Template". The definition of types must
    // be performed before the declaration of a variable.
    VAR_GLOBAL        // In the user memory of the UNIT
        g_boDigInput1 : BOOL;
        // Boolean variable for "EXPRESSION" example (see below).
    END_VAR


    VAR_GLOBAL RETAIN
    END_VAR
    // The variables declared with the add-on RETAIN are
    // stored in the RETAIN data area of the hardware platform used and
    // are therefore safe from network failure.
    // Variable declaration in the IMPLEMENTATION section.
    // The declaration of VAR, VAR CONSTANT, VAR_TEMP, VAR_INPUT, VAR_OUTPUT
    // and VAR_IN_OUT is not permissible here.

    EXPRESSION xCond
        xCond := g_boDigInput1;
    END_EXPRESSION
    // Definition of an EXPRESSION.
    // An EXPRESSION is a special function case, which recognizes only the
    // return values TRUE and FALSE. It is used in conjunction with the
    // statement WAITFORCONDITON (see myPRG) and should only be used
    // if the program is executed as part of
    // a MotionTask. If "dig_input_1" (usual in a digital input or a
    // condition in the program) takes on the value 1, the return value of the
    // EXPRESSION is TRUE.
```

A.3.5 Function

```
// ********************************************************
// * FUNCTION                                *
// ********************************************************

// The declaration of an FB or FC must be placed in the source file
// before the actual use (the call), so that the code of the
// block is already known to the calling point.

FUNCTION FC_myFirst : INT
    // The statement section of the POU FUNCTION begins here. The return value
    // of the function has the type integer in this case.
    // The stack of the calling TASK is initialized on each call.
      The return value is located on the stack and is
    // written by the FUNCTION.

    VAR CONSTANT
    END_VAR

    TYPE
    END_TYPE
    // The type declaration can also be made in POUs. The
    // basic difference is the validity of the
    // type declaration. A type declared in a POU can only
    // be used for variables within associated POU.

    VAR_INPUT   // In the stack of the calling TASK, will be placed on
                 // stack on call, assignment optional.
    END_VAR

    VAR         // In the stack of the calling task
                // is used in FUNCTION.
    END_VAR
    // Variable declaration in an FC.
    // The declaration of VAR_TEMP, VAR_GLOBAL, VAR_GLOBAL CONSTANT,
    // VAR_GLOBAL RETAIN, VAR_OUTPUT and VAR_IN_OUT is not
    // permissible here.

    // The use of unit-global variables for data acceptance in FCs
    // and FBs is the fastest option for the runtime. The use
    // of the input parameters VAR_INPUT and the return via the
    // return value is slower, since the values are copied respectively.

    // Comment: Variables declared with VAR and VAR CONSTANT are
    // temporary. On the next call, the contents from the latest
    // call are no longer available, in contrast to the FB.

    // ********************************************
    // * Area for FC code or statements *
    // ********************************************)
    // Code is in the user memory.
    g_eMyTraffic := YELLOW; // e.g. change the traffic light.

    FC_myFirst := 17;
    // In this example, the function returns the value "17" to the
    // calling program.

END_FUNCTION
```

A.3.6 Function block

```
// ******************************************************
// * FUNCTION_BLOCK                           *
// ******************************************************

// The declaration of an FB or FC must be placed in the source file
// before the actual use (the call), so that the code of the
// block is already known to the calling point.

FUNCTION_BLOCK FB_myFirst
    // The statement section of the FUNCTION_BLOCK POU begins here.
    // Instance data are dependent where the instance is formed
    // (see comments at the template end) in the user memory of UNIT
    // or TASK and are initialized with STOP->RUN or starting the TASK

    // The pointer to the instance data is transferred during the call.

    VAR CONSTANT
    END_VAR
    // Variables declared with VAR and VAR CONSTANT are
    // static, i.e., on the next block call, their contents remain
    // available and valid.

    TYPE
    END_TYPE
    // The type definition can also be made in POUs. The
    // basic difference is the validity of the
    // Type definition. A type defined in a POU can only
    // be used for variables within associated POU.

    VAR_INPUT    // In the user memory of the UNIT or TASK,
                 // assignment optional on call.
    END_VAR

    VAR_IN_OUT   // In the user memory of the UNIT or TASK,
                 // reference must be assigned on call.
    END_VAR

    VAR_OUTPUT   // In the user memory of the UNIT or TASK.
    END_VAR

    VAR          // In the user memory of the UNIT or TASK,
                 // can be used in the FB.
    END_VAR

    VAR_TEMP     // In the stack of the calling task,
                 // is initialized on each call.
    END_VAR

    // Variable declaration in an FB.
    // The declaration of VAR_GLOBAL, VAR_GLOBAL CONSTANT and
    // VAR_GLOBAL RETAIN is not permissible here.

    // ********************************************
    // * Area for FB code or statements *
    // ********************************************

    g_eMyTraffic := GREEN;        // e.g. change the traffic light.
END_FUNCTION_BLOCK
```

A.3.7 Program

```
// ****************************************************
// * PROGRAM                             *
// ****************************************************

PROGRAM myPRG
    // The statement section of the POU PROGRAM begins here.

    VAR CONSTANT
    END_VAR

    TYPE
    END_TYPE
    // The type definition can also be made in POUs. The
    // basic difference is the validity of the
    // Type definition. A type defined in a POU can only
    // be used for variables within associated POU.
    VAR     // In the user memory of the TASK.
        instFBMyFirst : FB_myFirst;
        // In order to be able to call an FB, an area for static
        // variables (forming an instance) must be generated. This has to do with
        // the "memory" of the FB.

        retFCMyFirst : INT;
        // Variable for the return value of the function.
    END_VAR

    VAR_TEMP    // In the stack of the task, initialized in each pass.
    END_VAR
    // Variable declaration in a PROGRAM.
    // The declaration of VAR_GLOBAL, VAR_GLOBAL CONSTANT,
    // VAR_GLOBAL RETAIN, VAR_INPUT, VAR_OUTPUT and VAR_IN_OUT
    // is not permissible here.

    // Comment: Whether the local variables declared via VAR
    // are temporary variables depends on the task context in which the
    // PROGRAM is used.
    //
    // In non-cyclic tasks (StartupTask, ShutdownTask, MotionTasks,
    // SystemInterruptTasks and UserInterruptTasks) the previous
    // contents of VAR and VAR_TEMP are no longer available.
    // The variables are thus temporary.
    //
    // With other cyclic tasks (BackgroundTask, IPOsynchronousTask,
    // IPOsynchronousTask_2 and TimerInterruptTasks), the contents
    // of variables declared in the VAR section remain the same
    // for the following run. The variables are thus static.
    // Variables from VAR_TEMP are always temporary.

    instFBMyFirst ();
    // FB call with a valid instance.

    retFCMyFirst := FC_myFirst ();
    // FC call and assignment of return value.
```

```
WAITFORCONDITION xCond WITH TRUE DO
            // The statements programmed here come immediately for
            // execution if the condition "xcond" defined in the associated
            // EXPRESSION is logically true.
            ;
        END_WAITFORCONDITION;
        // WAITFORCONDITION is generally used only in MotionTasks.
          These remain in the location and the
        // condition defined in the EXPRESSION is checked with high priority.

    END_PROGRAM

END_IMPLEMENTATION
//-----------------------------------------------------------------------------
```

A.3.8 Notes on initialization

```
//    INSTRUCTION FOR INITIALIZATION OF USER DATA
//    * User data (variables from elementary data types, structures, and arrays)
//    * are initialized as different times. The time
//    * depends on the location (i.e., memory area) of the data.
//    * A distinction is always made between the main memory of a task (stack) and
//    * in the user memory of the TASK. There is a user memory
//    * for a TASK and for a UNIT.

//    Data in the main memory of a task (stack):
//    ==============================================
//    Each task has a reserved memory for stack data (parameters for
//    function calls, temporary variables). The stack size of a TASK is
//    calculated by the compiler and can be influenced by the user in the
//    execution system under task configuration (Reserve for Download in the RUN).
//    * The main memory of a TASK (stack) contains the following data:
//     -    VAR of FUNCTIONs
//     -    VAR_TEMP of FUNCTION_BLOCKs and PROGRAMs
//     -    VAR_INPUT and return value of FUNCTIONs
//    * These are initialized at each call (delete / set to zero and
//     from the program, if necessary).

//    The user memory (heap) is managed separately for each UNIT and for each
//    TASK:
//    ============================================================================
//    * The user memory of a UNIT contains the following data:
//     -    VAR_GLOBAL from INTERFACE and IMPLEMENTATION
//    * These are initialized (delete / set to zero and write initial values
 //     from the program, if necessary):
//     -    During startup
//     -    During loading (if initialization of all non-retentive data is
 //          selected)
//
//    * The user memory of a TASK contains the following data:
//        VAR of PROGRAMs
//    * These are initialized (delete / set to zero and write initial values
//     from the program, if necessary):
//     -    For cyclic tasks, once when STOP->RUN
//     -    For non-cyclic tasks, at start of task
```

```
//
//    * The instance data of FUNCTION_BLOCKs (VAR_INPUT, VAR_OUTPUT,
 //     VAR_IN_OUT (reference), VAR) are dependent on where the instance of the FB
 //     is formed, in the user memory of a UNIT or TASK.
//     Instantiation of the FB in
//     -    VAR_GLOBAL:           Instance is located in the user memory of the UNIT
//     -    VAR in the PROGRAM:       Instance is located in the user memory of the TASK
//     -    VAR in the FB:            Instance is located in the user memory according to
 //                               higher-level FB
//    * The instance data are initialized as described above.
//    Which variable type is located in which data area can be obtained in
 //    comments in the template.
//----------------------------------------------------------------------------
```

Index