SIEMENS

# SIMOTION

# SIMOTION SCOUT
# SIMOTION ST Structured Text

Programming and Operating Manual


Preface

Introduction 1

Getting Started with ST 2

ST Fundamentals 3

Functions, Function Blocks, and Programs 4

Integration of ST in SIMOTION 5

Error Sources and Program Debugging 6

Appendix A


02/2012

## Legal information

### Warning notice system

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 relevant 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 product/system described in this documentation may be operated only by **personnel qualified** for the specific task in accordance with the relevant documentation, in particular its warning notices and safety instructions. Qualified personnel are those who, based on their training and experience, are capable of identifying risks and avoiding potential hazards when working with these products/systems.

### Proper use of Siemens products

Note the following:

| ⚠ WARNING |
| --- |
| Siemens products may only be used for the applications described in the catalog and in the relevant technical documentation. If products and components from other manufacturers are used, these must be recommended or approved by Siemens. Proper transport, storage, installation, assembly, commissioning, operation and maintenance are required to ensure that the products operate safely and without any problems. The permissible ambient conditions must be complied with. The information in the relevant documentation must be observed. |

### Trademarks

All names identified by ® are registered trademarks of 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
90026 NÜRNBERG
GERMANY

Copyright © Siemens AG 2012.
All rights reserved

# Preface

## Scope

This document is part of the SIMOTION Programming documentation package.

This document applies to SIMOTION SCOUT, the engineering system of the SIMOTION V4.3 product family in conjunction with:

- A SIMOTION device with the following versions of SIMOTION Kernel:
  - V4.3
  - V4.2
  - V4.1
  - V4.0
  - V3.2
- The relevant version of the following SIMOTION Technology Packages, depending on the kernel:
  - Cam
  - Path (Kernel V4.1 and higher)
  - Cam_ext
  - TControl

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 outputs warning 16700. 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 in the scope of delivery of SIMOTION SCOUT. It comprises 10 documentation packages.

The following documentation packages are available for SIMOTION V4.3:

- SIMOTION Engineering System
- SIMOTION System and Function Descriptions
- SIMOTION Service and Diagnostics
- SIMOTION IT
- SIMOTION Programming
- SIMOTION Programming - References
- SIMOTION C
- SIMOTION P
- SIMOTION D
- SIMOTION Supplementary Documentation

## Hotline and Internet addresses

## Additional information

Click the following link to find information on the the following topics:

- Ordering documentation/overview of documentation
- Additional links to download documents
- Using documentation online (find and search in manuals/information)

http://www.siemens.com/motioncontrol/docu

Please send any questions about the technical documentation (e.g. suggestions for improvement, corrections) to the following e-mail address:
docu.motioncontrol@siemens.com

## My Documentation Manager

Click the following link for information on how to compile documentation individually on the basis of Siemens content and how to adapt this for the purpose of your own machine documentation:

http://www.siemens.com/mdm

## Training

Click the following link for information on SITRAIN - Siemens training courses for automation products, systems and solutions:

http://www.siemens.com/sitrain

## FAQs

Frequently Asked Questions can be found in SIMOTION Utilities & Applications, which are included in the scope of delivery of SIMOTION SCOUT, and in the Service&Support pages in **Product Support**:

http://support.automation.siemens.com

## Technical support

Country-specific telephone numbers for technical support are provided on the Internet under **Contact**:

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

# Contents


**Preface** ..... **3**

**1 Introduction** ..... **15**

1.1 High-level programming language ..... 15

1.2 Programming language with technology commands ..... 15

1.3 Execution levels ..... 16

1.4 ST editor with tools for writing and testing programs ..... 16

**2 Getting Started with ST** ..... **17**

2.1 Integration of ST in SCOUT ..... 17
2.1.1 Getting to know the elements of the workbench ..... 19

2.2 Requirements for program creation ..... 20

2.3 Working with the ST editor and the compiler ..... 21
2.3.1 Insert ST source file ..... 21
2.3.2 Opening an existing ST source file ..... 24
2.3.3 Changing the properties of an ST source file ..... 25
2.3.4 Working with the ST editor ..... 26
2.3.4.1 Syntax coloring ..... 27
2.3.4.2 Drag&drop ..... 27
2.3.4.3 Settings of the ST editor ..... 29
2.3.4.4 Indentations and tabs ..... 30
2.3.4.5 Folds (show and hide blocks) ..... 33
2.3.4.6 Splitting the editor window ..... 36
2.3.4.7 Display spaces and tabs ..... 37
2.3.4.8 Changing the font size in the ST editor ..... 38
2.3.4.9 Select text ..... 39
2.3.4.10 Generating a simple series of numbers (generating a sequence) ..... 41
2.3.4.11 Use bookmarks ..... 43
2.3.4.12 Automatic completion ..... 45
2.3.4.13 Opening a called block ..... 46
2.3.4.14 Other help for the ST editor ..... 47
2.3.4.15 Using the command library ..... 48
2.3.4.16 ST editor toolbar ..... 48
2.3.4.17 ST editor context menu ..... 49
2.3.4.18 Shortcuts ..... 52
2.3.5 Starting the compiler ..... 55
2.3.5.1 Help for the error correction ..... 55
2.3.6 Making settings for the compiler ..... 56
2.3.6.1 Global compiler settings ..... 56
2.3.6.2 Local compiler settings ..... 59
2.3.6.3 Effectiveness of global or local compiler settings ..... 63
2.3.6.4 Meanings of the warning classes ..... 65
2.3.6.5 Display of the compiler options ..... 65
2.3.7 Know-how protection for ST source files ..... 67

2.3.8 Making preprocessor definitions ..... 67
2.3.9 Exporting, importing and printing an ST source file ..... 69
2.3.9.1 Exporting an ST source file as a text file (ASCII) ..... 69
2.3.9.2 Exporting an ST source file in XML format ..... 70
2.3.9.3 Importing a text file (ASCII) as an ST source file ..... 70
2.3.9.4 Importing XML data into ST source files ..... 70
2.3.9.5 Printing an ST source file ..... 71
2.3.10 Using an external editor ..... 71
2.3.11 ST source file menus ..... 73
2.3.11.1 ST source file menu ..... 73
2.3.11.2 ST source file context menu ..... 74

2.4 Creating a sample program ..... 76
2.4.1 Requirements ..... 76
2.4.2 Opening or creating a project ..... 77
2.4.3 Making the hardware known ..... 78
2.4.4 Entering source text with the ST editor ..... 79
2.4.4.1 Functions of the editor ..... 80
2.4.4.2 Source text of the sample program ..... 81
2.4.5 Compiling a sample program ..... 81
2.4.5.1 Starting the compiler ..... 81
2.4.5.2 Correcting errors ..... 82
2.4.5.3 Example of error messages ..... 82
2.4.6 Running the sample program ..... 83
2.4.6.1 Assigning a sample program to an execution level ..... 83
2.4.6.2 Establishing a connection to the target system ..... 84
2.4.6.3 Downloading the sample program to the target system ..... 85
2.4.6.4 Starting and testing the sample program ..... 86

**3 ST Fundamentals ..... 87**

3.1 Language description resources ..... 87
3.1.1 Syntax diagram ..... 87
3.1.2 Blocks in syntax diagrams ..... 88
3.1.3 Meaning of the rules (semantics) ..... 88

3.2 Basic elements of the language ..... 89
3.2.1 ST character set ..... 89
3.2.2 Identifiers in ST ..... 89
3.2.2.1 Rules for identifiers ..... 89
3.2.2.2 Examples of identifiers ..... 90
3.2.3 Reserved identifiers ..... 91
3.2.3.1 Protected identifiers in the ST programming language ..... 91
3.2.3.2 Reserved identifiers in the ST programming language ..... 96
3.2.3.3 Additional reserved identifiers ..... 97
3.2.4 Numbers and Boolean values ..... 98
3.2.4.1 Integers ..... 98
3.2.4.2 Floating-point numbers ..... 99
3.2.4.3 Exponents ..... 99
3.2.4.4 Boolean values ..... 100
3.2.4.5 Data types of numbers ..... 100
3.2.5 Character strings ..... 101

3.3 Structure of an ST source file ..... 102
3.3.1 Statements ..... 103
3.3.2 Comments ..... 104

3.4 Data types ..... 105
3.4.1 Elementary data types ..... 106
3.4.1.1 Elementary data types ..... 106
3.4.1.2 Value range limits of elementary data types ..... 108
3.4.1.3 General data types ..... 109
3.4.1.4 Elementary system data types ..... 109
3.4.2 User-defined data types ..... 110
3.4.2.1 User-defined data types ..... 110
3.4.2.2 Syntax of user-defined data types (type declaration) ..... 111
3.4.2.3 Derivation of elementary or derived data types ..... 113
3.4.2.4 Derived data type ARRAY ..... 114
3.4.2.5 Derived data type - Enumerator ..... 116
3.4.2.6 Derived data type STRUCT (structure) ..... 118
3.4.3 Technology object data types ..... 120
3.4.3.1 Description of the technology object data types ..... 120
3.4.3.2 Inheritance of the properties for axes ..... 121
3.4.3.3 Examples of the use of technology object data types ..... 122
3.4.4 System data types ..... 123

3.5 Variable declaration ..... 124
3.5.1 Syntax of variable declaration ..... 124
3.5.2 Overview of all variable declarations ..... 126
3.5.3 Initialization of variables or data types ..... 127
3.5.4 Constants ..... 131

3.6 Value assignments and expressions ..... 133
3.6.1 Value assignments ..... 133
3.6.1.1 Syntax of the value assignment ..... 133
3.6.1.2 Value assignments with variables of an elementary data type ..... 135
3.6.1.3 Value assignments with variables of the STRING elementary data type ..... 135
3.6.1.4 Value assignments with variables of a bit data type ..... 137
3.6.1.5 Value assignments with variables of the derived enumerator data type ..... 139
3.6.1.6 Value assignments with variables of the derived ARRAY data type ..... 139
3.6.1.7 Value assignments with variables of the derived STRUCT data type ..... 140
3.6.2 Expressions ..... 141
3.6.2.1 Result of an expression ..... 141
3.6.2.2 Interpretation order of an expression ..... 142
3.6.3 Operands ..... 143
3.6.4 Arithmetic expressions ..... 144
3.6.4.1 Examples of arithmetic expressions ..... 146
3.6.5 Relational expressions ..... 147
3.6.6 Logic expressions and bit-serial expressions ..... 149
3.6.7 Priority of operators ..... 151

3.7 Control statements ..... 152
3.7.1 IF statement ..... 152
3.7.2 CASE statement ..... 154
3.7.3 FOR statement ..... 156
3.7.4 WHILE statement ..... 158
3.7.5 REPEAT statement ..... 159

3.7.6 EXIT statement ... 160
3.7.7 RETURN statement ... 161
3.7.8 WAITFORCONDITION statement ... 162
3.7.9 GOTO statement ... 163

3.8 Data type conversions ... 164
3.8.1 Elementary data type conversion ... 164
3.8.1.1 Implicit data type conversions ... 165
3.8.1.2 Explicit data type conversions ... 167
3.8.2 Supplementary conversions ... 168

**4 Functions, Function Blocks, and Programs ... 169**

4.1 Creating and calling functions and function blocks ... 169
4.1.1 Defining functions ... 170
4.1.2 Defining function blocks ... 171
4.1.3 Declaration section of FB and FC ... 171
4.1.4 Statement section of FB and FC ... 174
4.1.5 ARRAY with a dynamic length (as of Kernel V4.2) ... 175
4.1.6 Call of functions and function block calls ... 177
4.1.6.1 Principle of parameter transfer ... 177
4.1.6.2 Parameter transfer to input parameters ... 177
4.1.6.3 Parameter transfer to in/out parameters ... 178
4.1.6.4 Parameter transfer to output parameters (for FB only) ... 179
4.1.6.5 Parameter access times ... 180
4.1.6.6 Calling a function ... 180
4.1.6.7 Calling function blocks (declaring and calling instances) ... 181
4.1.6.8 Accessing the FB's output parameter outside the FB ... 183
4.1.6.9 Accessing the FB's input parameter outside the FB ... 183
4.1.6.10 Error sources in FB calls ... 184

4.2 Comparison of functions and function blocks ... 185
4.2.1 Description of example ... 185
4.2.2 Source file with comments ... 186

4.3 Programs ... 188
4.3.1 Assignment of a program in the execution system ... 188
4.3.2 Calling a program in the program ("program in program") ... 188

4.4 Expressions ... 190

**5 Integration of ST in SIMOTION ... 193**

5.1 Source file sections ... 193
5.1.1 Use of the source file sections ... 193
5.1.1.1 Interface section ... 194
5.1.1.2 Implementation section ... 197
5.1.1.3 Program organization units (POUs) ... 198
5.1.1.4 Functions (FCs) ... 198
5.1.1.5 Function blocks (FBs) ... 199
5.1.1.6 Programs ... 200
5.1.1.7 Expressions ... 200
5.1.1.8 Declaration section ... 201
5.1.1.9 Statement section ... 202
5.1.1.10 Data type definition ... 203
5.1.1.11 Variable declaration ... 203

5.1.2 Import and export between ST source files ....206
5.1.2.1 Unit identifier ....206
5.1.2.2 Interface section of an exporting unit ....206
5.1.2.3 Example of an exporting unit ....208
5.1.2.4 USES statement in an importing unit ....209
5.1.2.5 Example of an importing unit ....211

5.2 Variables in SIMOTION ....212
5.2.1 Variable model ....212
5.2.1.1 Unit variables ....215
5.2.1.2 Non-retentive unit variables ....215
5.2.1.3 Retentive unit variables ....217
5.2.1.4 Local variables (static and temporary variables) ....218
5.2.1.5 Static variables ....220
5.2.1.6 Temporary variables ....221
5.2.2 Use of global device variables ....222
5.2.3 Memory ranges of the variable types ....223
5.2.3.1 Example of memory areas ....225
5.2.3.2 Memory requirement of the variables on the local data stack ....228
5.2.4 Time of the variable initialization ....228
5.2.4.1 Initialization of retentive global variables ....229
5.2.4.2 Initialization of non-retentive global variables ....231
5.2.4.3 Initialization of local variables ....233
5.2.4.4 Initialization of static program variables ....234
5.2.4.5 Initialization of instances of function blocks (FBs) ....235
5.2.4.6 Initialization of system variables of technology objects ....236
5.2.4.7 Version ID of global variables and their initialization during download ....237
5.2.5 Variables and HMI devices ....238

5.3 Access to inputs and outputs (process image, I/O variables) ....241
5.3.1 Overview of access to inputs and outputs ....241
5.3.2 Important features of direct access and process image access ....242
5.3.3 Direct access and process image of cyclic tasks ....245
5.3.3.1 Address range of the SIMOTION devices ....248
5.3.3.2 Rules for I/O addresses for direct access and the process image of the cyclical tasks ....249
5.3.3.3 Creating I/O variables for direct access or process image of cyclic tasks ....250
5.3.3.4 Syntax for entering I/O addresses ....253
5.3.3.5 Possible data types of I/O variables ....254
5.3.3.6 Detailed status of the I/O variables (as of Kernel V4.2) ....254
5.3.4 Access to fixed process image of the BackgroundTask ....256
5.3.4.1 Common process image (as of Kernel V4.2) ....259
5.3.4.2 Separate process image (up to Kernel V4.1) ....262
5.3.4.3 Absolute access to the fixed process image of the BackgroundTask (absolute PI access) ....264
5.3.4.4 Syntax for the identifier for an absolute process image access ....265
5.3.4.5 Symbolic access to the fixed process image of the BackgroundTask (symbolic PI access) ....266
5.3.4.6 Possible data types for symbolic PI access ....267
5.3.4.7 Example of symbolic PI access ....267
5.3.4.8 Creating an I/O variable for access to the fixed process image of the BackgroundTask ....268
5.3.5 Accessing I/O variables ....269

5.4 Using libraries ....270
5.4.1 Compiling a library ....271
5.4.2 Know-how protection for libraries ....273
5.4.3 Using data types, functions and function blocks from libraries ....274

5.5 Use of the same identifiers and namespaces ... 275
5.5.1 Use of the same identifiers ... 275
5.5.2 Namespaces ... 278

5.6 Reference data ... 282
5.6.1 Cross-reference list ... 282
5.6.1.1 Creating a cross-reference list ... 282
5.6.1.2 Content of the cross-reference list ... 283
5.6.1.3 Working with a cross-reference list ... 285
5.6.1.4 Filtering the cross-reference list ... 286
5.6.2 Program structure ... 286
5.6.2.1 Content of the program structure ... 287
5.6.3 Code attributes ... 288
5.6.3.1 Code attribute contents ... 288

5.7 Controlling the preprocessor and compiler with pragmas ... 289
5.7.1 Controlling the preprocessor ... 290
5.7.1.1 Preprocessor statement ... 291
5.7.1.2 Example of preprocessor statements ... 293
5.7.2 Controlling compiler with attributes ... 294

5.8 SIMOTION devices ... 297
5.8.1 Rules for identifiers of the SIMOTION devices ... 297
5.8.2 Making settings on the device (as of Kernel V4.2) ... 299

5.9 Forward declarations ... 300

5.10 Jump statement and label ... 303

**6 Error Sources and Program Debugging ... 305**

6.1 Notes on avoiding errors and on efficient programming ... 305

6.2 Program debugging ... 305
6.2.1 Operating modes for program testing ... 306
6.2.1.1 Modes of the SIMOTION devices ... 306
6.2.1.2 Important information about the life-sign monitoring ... 308
6.2.1.3 Life-sign monitoring parameters ... 310
6.2.2 Editing program sources in online mode ... 311
6.2.3 Symbol Browser ... 312
6.2.3.1 Properties of the symbol browser ... 312
6.2.3.2 Using the symbol browser ... 312
6.2.4 Monitoring variables in watch table ... 315
6.2.4.1 Variables in the watch table ... 315
6.2.4.2 Using watch tables ... 316
6.2.5 Variable status ... 317
6.2.6 Program run ... 319
6.2.6.1 Program run: Display code location and call path ... 319
6.2.6.2 Program run parameters ... 320
6.2.6.3 Program run toolbar ... 320
6.2.7 Program status ... 321
6.2.7.1 Properties of the program status ... 321
6.2.7.2 Using the status program ... 323
6.2.7.3 Call path for program status ... 325
6.2.7.4 Parameter call path status program ... 326

6.2.8 Breakpoints ..... 327
6.2.8.1 General procedure for setting breakpoints ..... 327
6.2.8.2 Setting the debug mode ..... 327
6.2.8.3 Define the debug task group ..... 329
6.2.8.4 Debug task group parameters ..... 331
6.2.8.5 Debug table parameter ..... 332
6.2.8.6 Setting breakpoints ..... 332
6.2.8.7 Breakpoints toolbar ..... 334
6.2.8.8 Defining the call path for a single breakpoint ..... 335
6.2.8.9 Breakpoint call path / task selection parameters ..... 337
6.2.8.10 Defining the call path for all breakpoints ..... 338
6.2.8.11 Call path / task selection parameters of all breakpoints per POU ..... 340
6.2.8.12 Activating breakpoints ..... 341
6.2.8.13 Display call stack ..... 343
6.2.8.14 Breakpoints call stack parameter ..... 345
6.2.9 Task status function bar ..... 345
6.2.10 Trace ..... 346
6.2.11 Project comparison ..... 347

**A Appendix ..... 349**

A.1 Formal Language Description ..... 349
A.1.1 Language description resources ..... 349
A.1.1.1 Formatted rules (lexical rules) ..... 349
A.1.1.2 Unformatted rules (syntactic rules) ..... 350
A.1.2 Basic elements (terminals) ..... 351
A.1.2.1 Letters, digits and other characters ..... 351
A.1.2.2 Formatting characters and separators in the rules ..... 351
A.1.2.3 Formatting characters and separators for constants ..... 353
A.1.2.4 Predefined identifiers for process image access ..... 354
A.1.2.5 Identifiers of the Taskstartinfo ..... 355
A.1.2.6 Operators ..... 356
A.1.2.7 Reserved words ..... 357
A.1.3 Rules ..... 365
A.1.3.1 Identifiers ..... 365
A.1.3.2 Notation for constants (literals) ..... 366
A.1.3.3 Comments ..... 374
A.1.3.4 Sections of the ST source file ..... 375
A.1.3.5 Structures of ST source files ..... 375
A.1.3.6 Program organization units (POU) ..... 377
A.1.3.7 Declaration sections ..... 379
A.1.3.8 Structure of the declaration blocks ..... 381
A.1.3.9 Data types ..... 388
A.1.3.10 Statement section ..... 392
A.1.3.11 Value assignments and operations ..... 393
A.1.3.12 Call of functions and function block calls ..... 399
A.1.3.13 Control statements ..... 402

A.2 Compiler Error Messages and Remedies ..... 407
A.2.1 File access errors (1000 – 1100) ..... 407
A.2.2 Scanner errors (2001, 2002) ..... 407
A.2.3 Declaration errors in POU (3002 – 3027) ..... 407
A.2.4 Declaration errors in data type declarations (4001 – 4051) ..... 409
A.2.5 Declaration errors in variables declarations (5001 – 5509) ..... 410

A.2.6 Errors in the expression (6001 - 6201) ..... 411
A.2.7 Syntax errors, errors in the expression (7000 - 7014) ..... 414
A.2.8 Error when linking a source file (8001, 8100) ..... 415
A.2.9 Errors while loading the interface of another UNIT or a technology package (10000 - 10101) ..... 416
A.2.10 Implementation restrictions (15001 - 15700) ..... 418
A.2.11 Warnings (16001 - 16700) ..... 419
A.2.12 Information (32010 - 32653) ..... 424
A.3 Template for Example Unit ..... 426
A.3.1 Preliminary information ..... 426
A.3.2 Type definition in the interface ..... 427
A.3.3 Variable declaration in the interface ..... 429
A.3.4 Implementation ..... 430
A.3.5 Function ..... 431
A.3.6 Function block ..... 433
A.3.7 Program ..... 434
**Index ..... 437**

# 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.

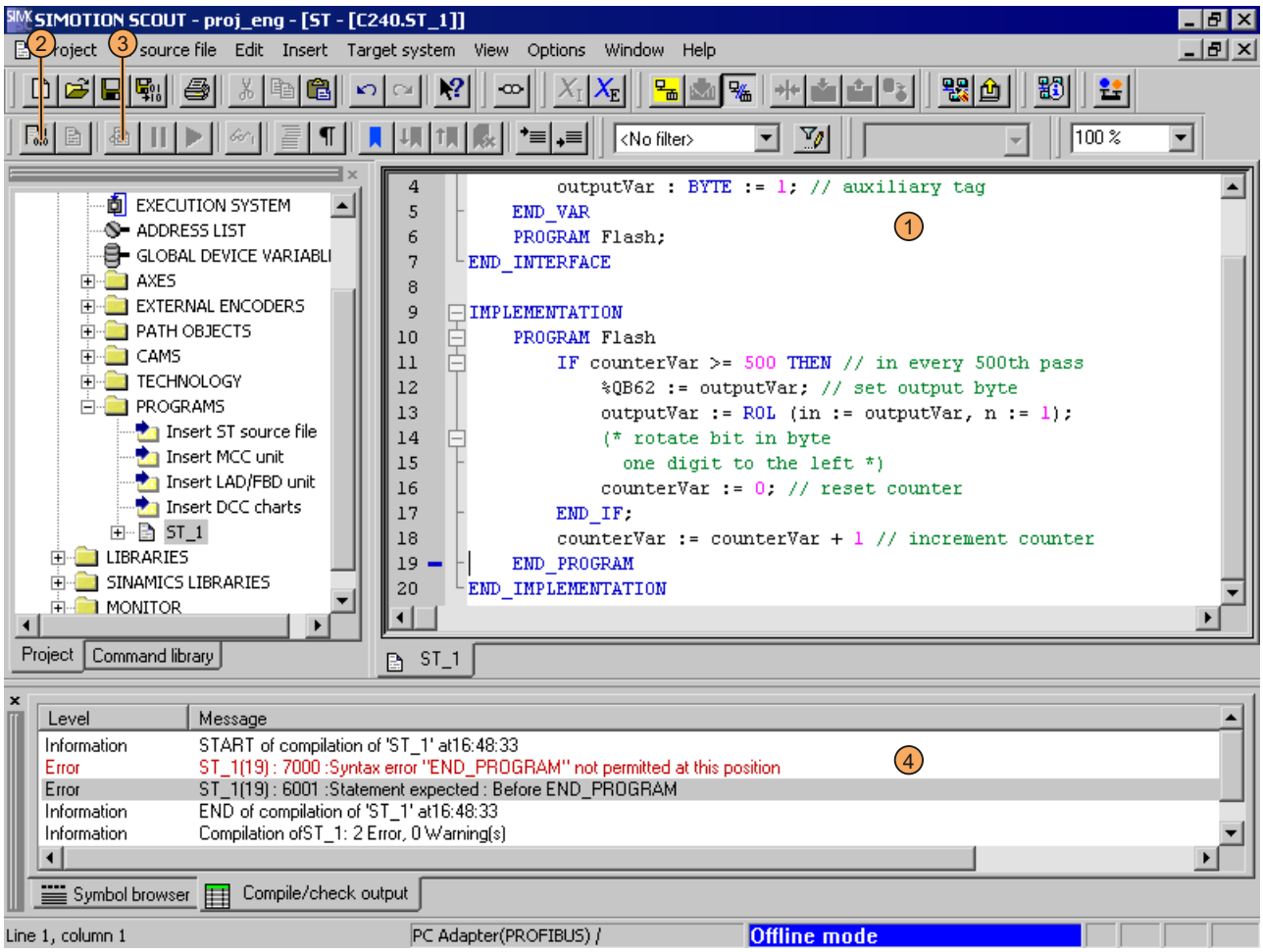
## 2.1 Integration of ST in SCOUT

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
- several diagnostics functions (e.g. **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.

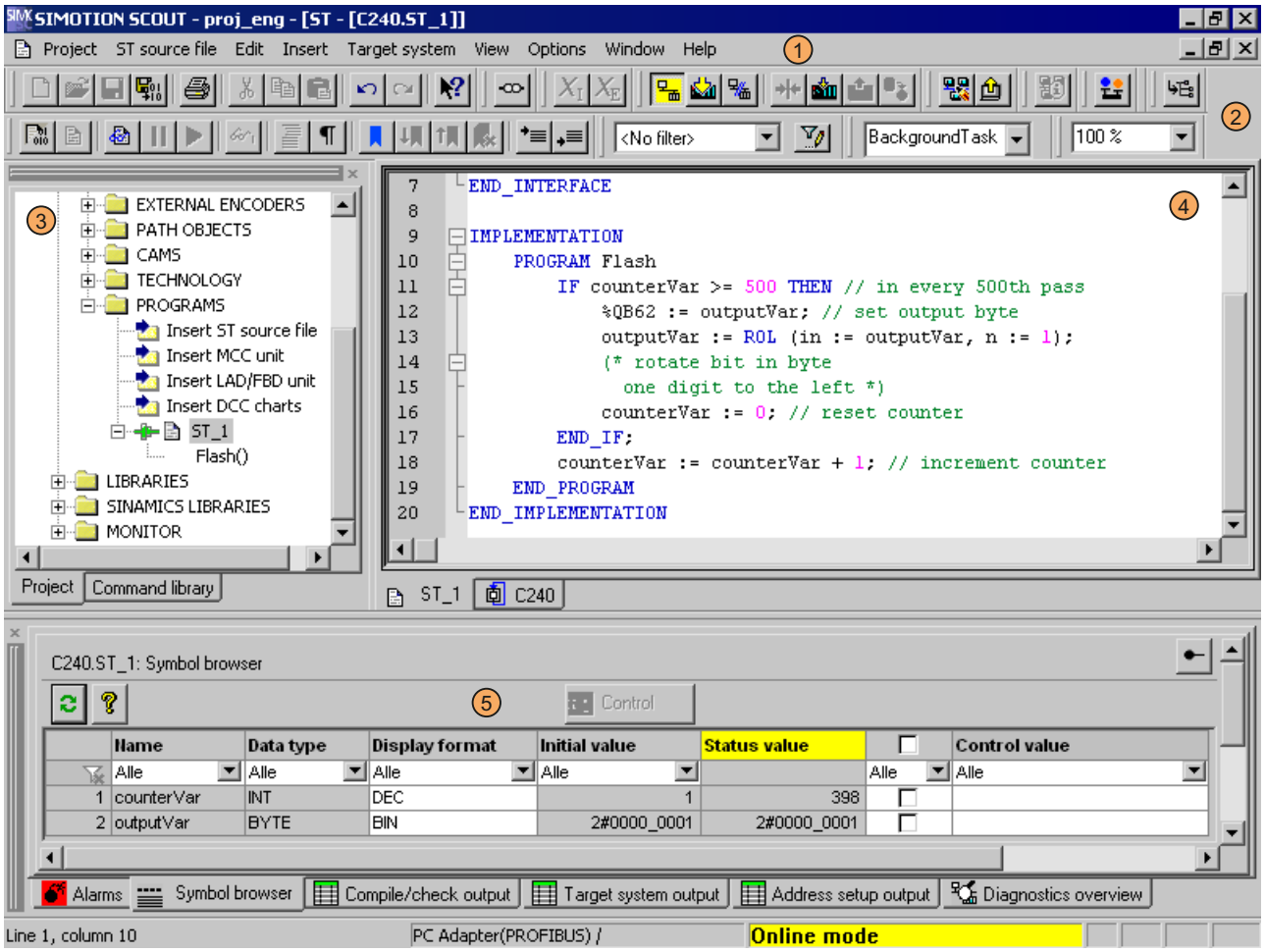



① Editor
② Compiler
③ Program status
④ Detail view (Compile/check output tab)

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.




① Menu bar
② Toolbars
③ Project navigator
④ Working Area
⑤ Detail view (Symbol browser tab)

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.

- Working 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 the programs contained in the source file will subsequently be run (e.g. SIMOTION C240).

They are stored in the project navigator under the SIMOTION device in the PROGRAMS folder.

**Note**

MCC units, LAD/FBD units and DCC charts are also stored in the **PROGRAMS** folder under the SIMOTION device.

For a description of the SIMOTION MCC (Motion Control Chart) programming language, refer to the SIMOTION MCC Programming and Operating Manual.

For a description of the SIMOTION LAD (Ladder Diagram) and SIMOTION FBD (Function Block Diagram) programming languages, refer to the SIMOTION LAD/FBD Programming and Operating Manual.

## Procedure

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.

   The names of program sources must comply with the rules for identifiers (Page 89): 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 91) are not permitted.

   Existing program sources (e.g. ST source files, MCC units) are displayed.
5. You can also enter an author, version, and a comment.
6. Activate the **Open editor automatically** checkbox.
7. If necessary, select further tabs to make local settings (only valid for this ST source file):
   – **Compiler** tab: Local settings of the compiler (Page 59) for code generation and message display.
   – **Additional settings** tab: Definitions for preprocessor (Page 67)
8. Confirm with **OK**.

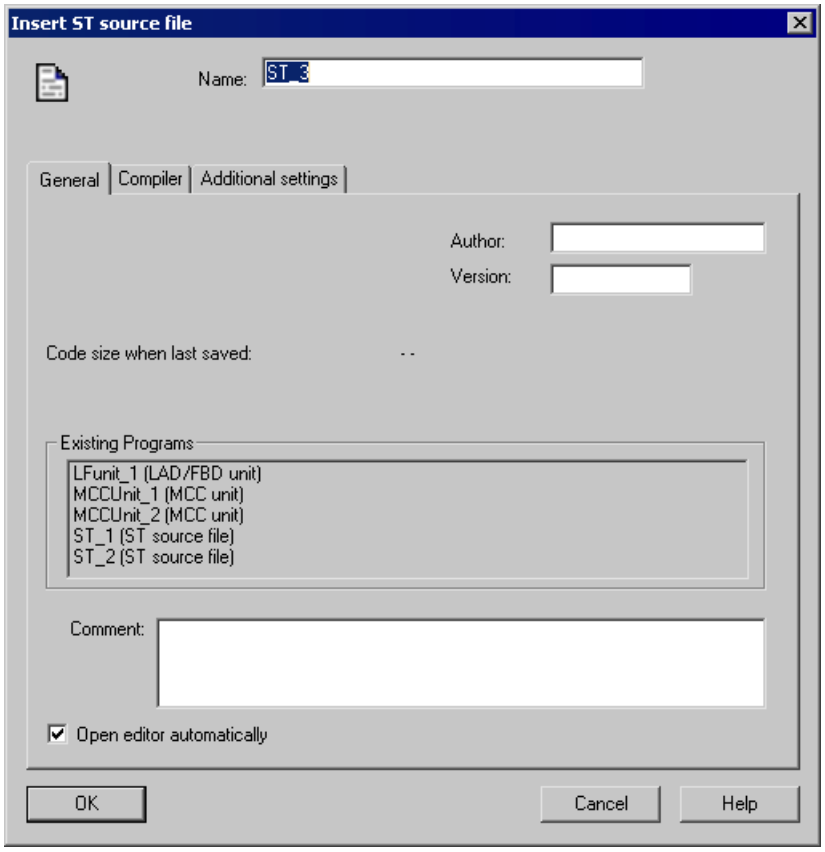



Figure 2-3 Insert ST source file

| NOTICE |
| --- |
| When you click **OK**, the ST source file will only be transferred to the project. The data, together with the project, is only saved to the data carrier if you select, for example, **Project > Save**, **Project > Save and compile changes**, or **Project > Save and recompile all**. |

## 2.3.2 Opening an existing ST source file

### Procedure

How to open an ST source file:

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 ST source file is not already open and the log-in details assigned to it have not yet been used to log in:

   – Enter the corresponding password for the displayed login.

     The know-how protection for this source file is canceled temporarily (until the file is closed).

   – If required, activate the "Use login as a standard login" checkbox.

     You will be logged in with this login and can now open additional units to which the same login is assigned without having to re-enter the password.

This ST source file is opened with the saved folding information (Page 33) and bookmarks (Page 43). Multiple units can be opened.

**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

### Procedure

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 file, e.g. code size for the last compilation, time stamp of the last change, storage location of the project (see figure).
   - **Compiler** tab: Local settings of the compiler (Page 59) for code generation and message display.
   - **Additional settings** tab: Definitions for the preprocessor (Page 67) and display of compiler options (Page 65) as specified in the current settings of the compiler.
   - **Compilation** tab: Display of the compiler options (Page 65) 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.

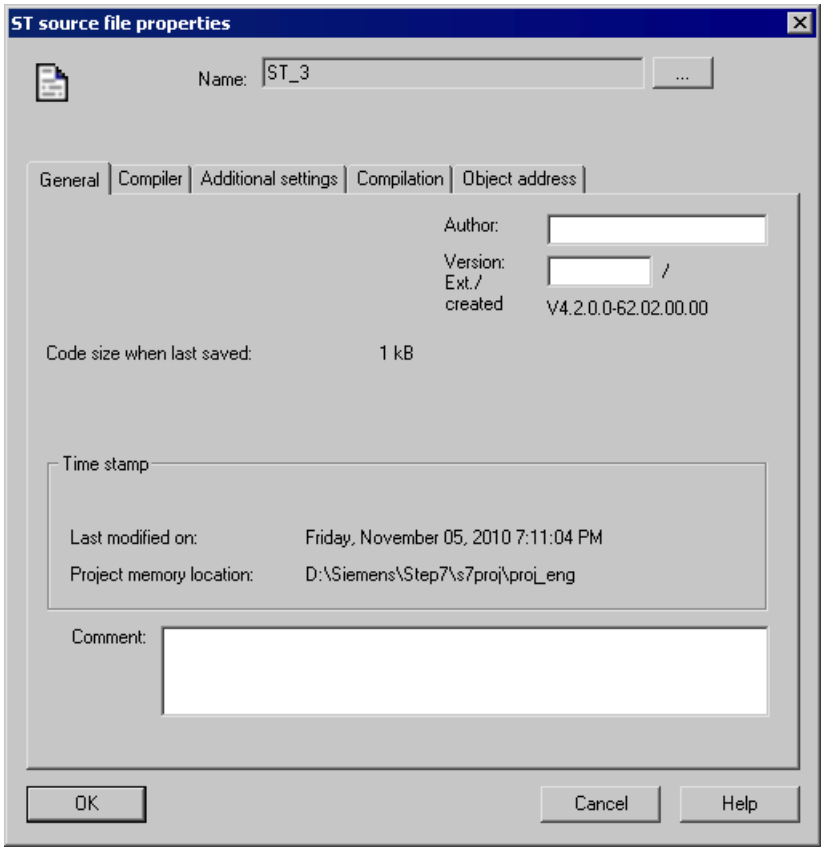



Figure 2-4 Properties of an ST source file

### 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 91) 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

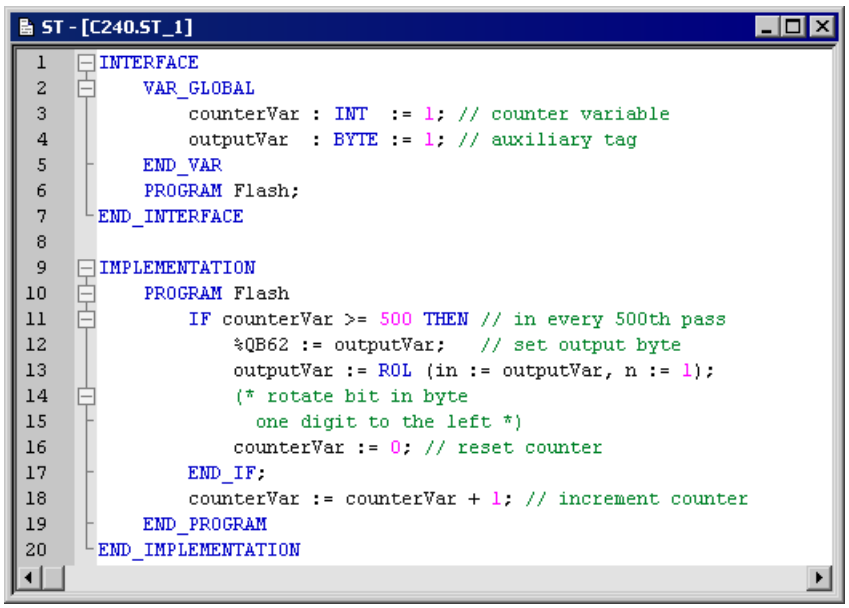



Figure 2-5 Opened ST source file in the ST editor

### 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 Settings of the ST editor

**Procedure:**

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

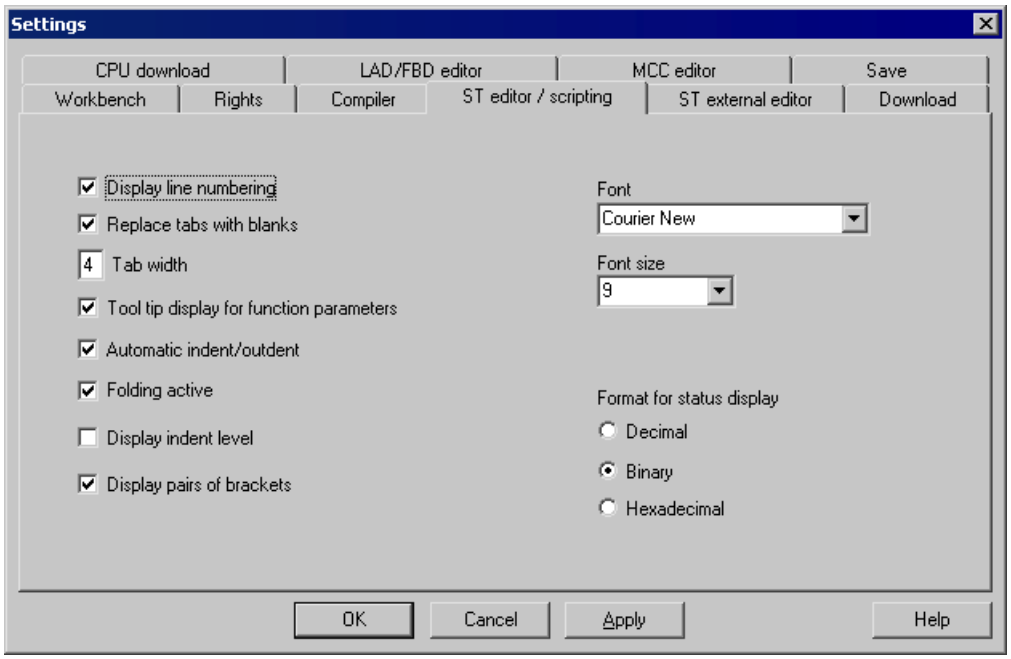



Figure 2-6 ST editor / scripting settings

The settings also apply to the script editor.

The table below contains a description of the individual parameters.

Table 2- 1 ST editor / scripting parameter settings

| Parameter | Description |
|---|---|
| Display line numbering | If active, the line numbers are displayed.<br>See: Other ST editor tools (Page 47). |
| Replace tabs with blanks | You select here how text indentation is performed (for the automatic indentation or by pressing the Tab key):<br>• If active: By adding the appropriate number of space characters ($20).<br>• If inactive: By adding the tab character ($09).<br>See: Indentations and tabs (Page 30). |
| Tab width | Number of characters skipped by a tab.<br>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.<br>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.<br>You can then hide blocks in an ST source file so that only the first line of the block remains visible.<br>See: Folding (showing and hiding blocks) (Page 33). |
| 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).<br>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.<br>See: Other ST editor tools (Page 47). |
| 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.<br>See: Change the font size in the ST editor (Page 38). |
| Format for status display [1] | Format in which the variable values with bit data type are displayed for the program status or the variable status (for ST editor only).<br>See: Properties of the program status (Page 321), variable status (Page 317) |

1 Only for ST editor

### 2.3.4.4 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 current 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 39)).
2. Select the **View > Format current selection** context menu.

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

### Indent or outdent selected area

You can use this function to indent or outdent selected sections of text by one tab width.

Follow these steps:

1. Select the text area that you want to indent or outdent in the ST editor (see Selecting text (Page 39)).

   The selected text area must cover multiple lines.

2. Select the **Indent selected area** or **Undo selected area** context menu.

### Indent help (Display indent level)

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

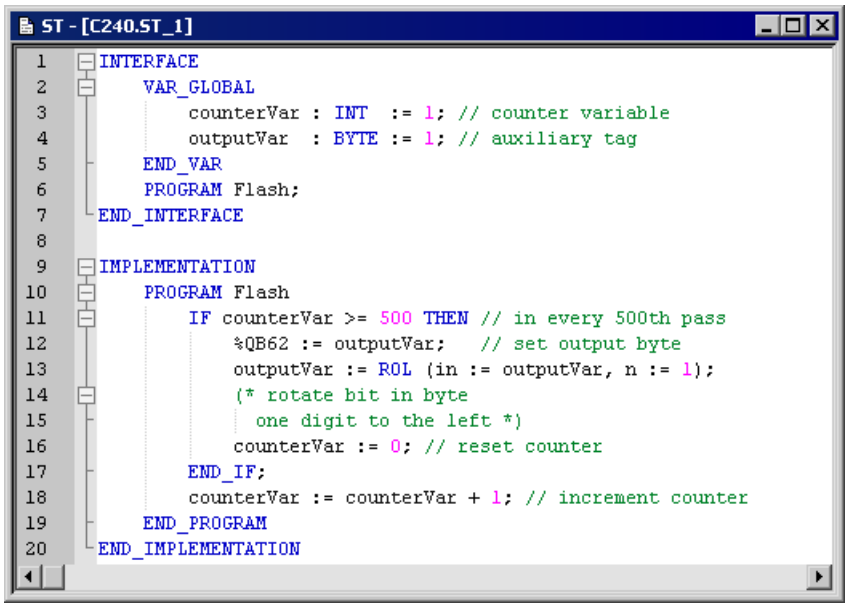



Figure 2-7 ST source with visible indent aid

You can activate or deactivate this function:

- For the active ST source
  - Select the **View > Indent help** context menu.

  This setting is not saved when the ST source file is closed.
- For all open ST sources:
  - Activate or deactivate the **Display indentation level** checkbox in the ST editor settings (Page 29).

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

### 2.3.4.5 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 legibility when editing or reading a longer 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 (* / *)

Folding must be activated (see below) in order to show and hide blocks. Once this function has been activated, the column containing the folding information (to the left of the edit area) is displayed.

How to recognize that a block is displayed with folding activated:

- The ⊟ (minus) character appears next to the first line of the block.

How to recognize that a block is hidden:

- The ⊞ (plus) character appears next to the first visible line of the block.
- A hyphen is displayed below this line. This hyphen is visible even if the column containing the folding information is hidden.

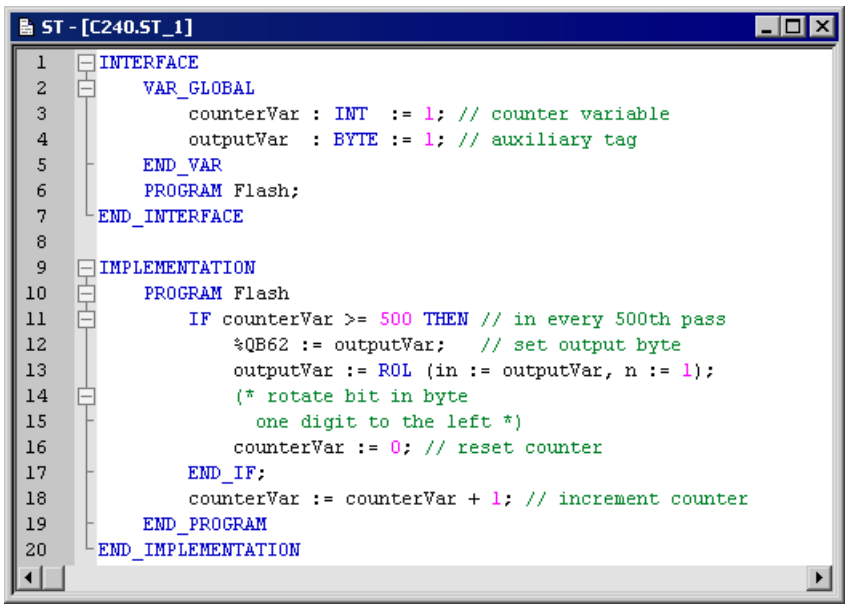



Figure 2-8 ST source file for which all blocks are shown

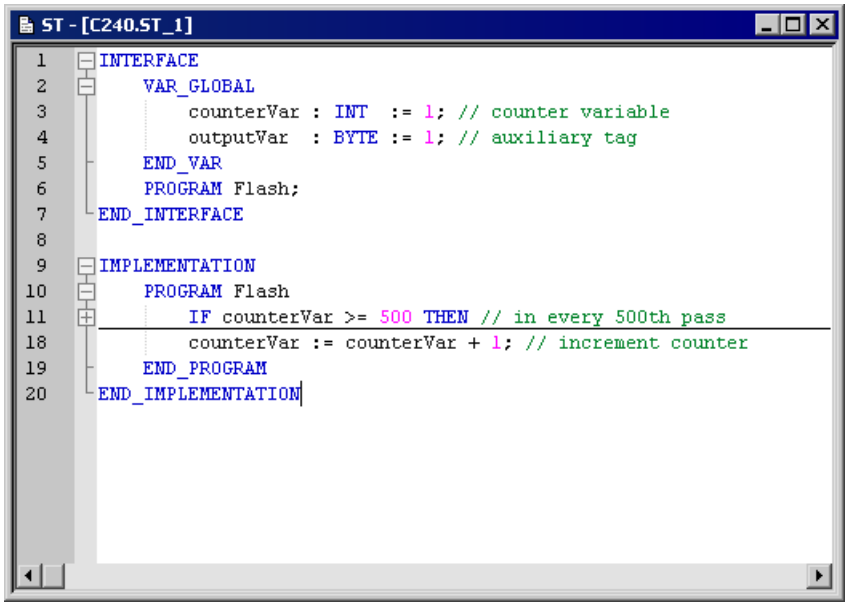



Figure 2-9 ST source file with hidden IF block (including block comment)

## Activating or deactivating folding

Folding must be activated in order to be able to show and hide blocks. Once this function has been activated, the column containing the folding information (to the left of the edit area) is displayed.

How to activate or deactivate folding:

- For the active ST source file:
  - Select the **View > Folding** context menu.

  This setting is not saved when the ST source file is closed.
- For all open ST sources:
  - Activate or deactivate the **Folding active** checkbox in the settings of the ST editor (Page 29).

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

## Showing and hiding blocks

You can only show or hide blocks if folding has been activated for the source file in question (see above). Once this function has been activated, the column containing the folding information is displayed.

How to show or hide blocks:

- Hide an individual block (options):
  - Click on the ⊟ (minus) character in the column containing the folding information.
  - Position the cursor on the corresponding line of the block and press the **CTRL+ALT+T** shortcut.

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

  The show/hide status of the subordinate blocks is saved. This is reinstated when individual blocks are shown.
- Show an individual block (options):
  - Click on the ⊞ (plus) character in the column containing the folding information.
  - Position the cursor on the visible line of the block and press the **CTRL+ALT+T** shortcut.

  All subsequent lines of the block will be shown. Subordinate blocks are shown as follows: If the show/hide status has been saved (when hiding individual blocks, for example), this will be reinstated.
- Hide all blocks:
  - Press the **CTRL+ALT+C** shortcut.

  All the blocks of the ST source file (including all the subordinate blocks) will be hidden. In each case, only the first line of the 1st-level blocks remain visible (usually INTERFACE and IMPLEMENTATION).
- Show all blocks:
  - Press the **CTRL+ALT+D** shortcut.

  All the blocks of the ST source file (including all the subordinate blocks) will be shown. All the lines of the ST source file will be visible.
- Hide subordinate blocks:
  - Position the cursor on the corresponding line of the block and press the **CTRL+ALT+V** shortcut.

  All the subordinate blocks for the current block will be hidden and the current block itself will be shown. Only the lines of the current block and the first lines of the blocks on the next level will be visible.
- Show subordinate blocks:
  - Position the cursor on the corresponding line of the block and press the **CTRL+ALT+R** shortcut.

  The current block and all subordinate blocks will be shown. All the lines of these blocks will be visible.

**Note**

The information relating to displayed and hidden blocks is saved in the project when the ST source file is closed.

This information is not transferred when the ST source file is exported (Page 69).

### 2.3.4.6 Splitting the editor window

You can split the ST editor window into two segments, giving you two views of the same ST source file.

#### Splitting a window or canceling a split

How to split the current editor window into two segments or cancel such a split:

- Select the **ST source file > Split window** menu command.

Following the split, the cursor is located in the upper window segment.

The settings for splitting the editor window are not saved when the ST source file is closed. ST source files are always opened in an entire window (i.e. one that is not split).

**Note**

The editor window cannot be split if the program status (Page 323) test function is being used.

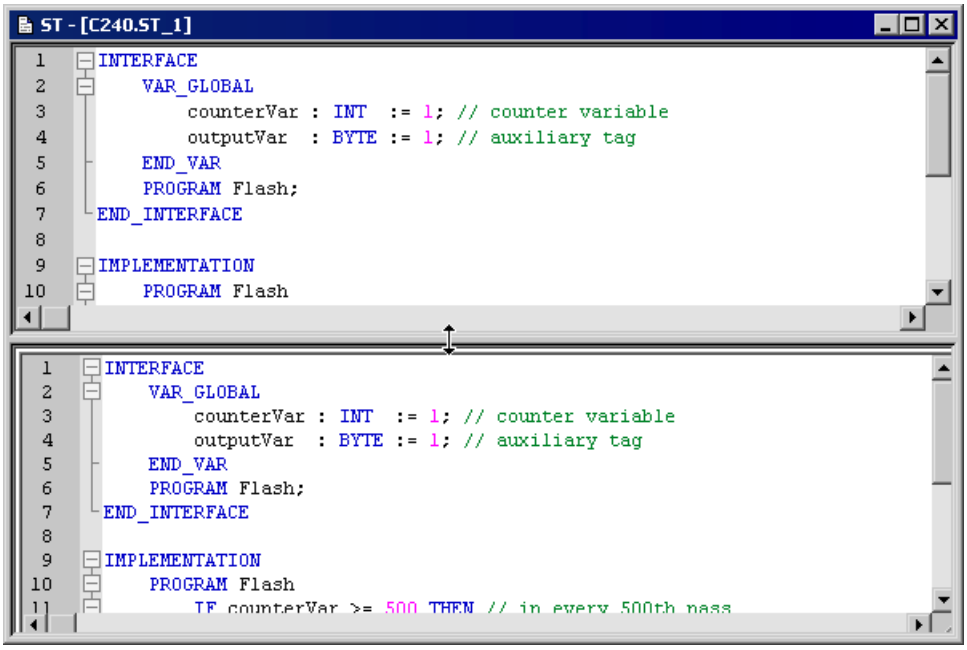



Figure 2-10 ST editor with split window

### Adjusting the segment height

You can adjust the height of the two segments to your individual requirements:

1. Use the mouse to move the cursor to the dividing line between the two segments, until it turns into a double-headed arrow (see screenshot above).
2. Press and hold the left mouse button and drag the dividing line to the required position.

## 2.3.4.7 Display spaces and tabs

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

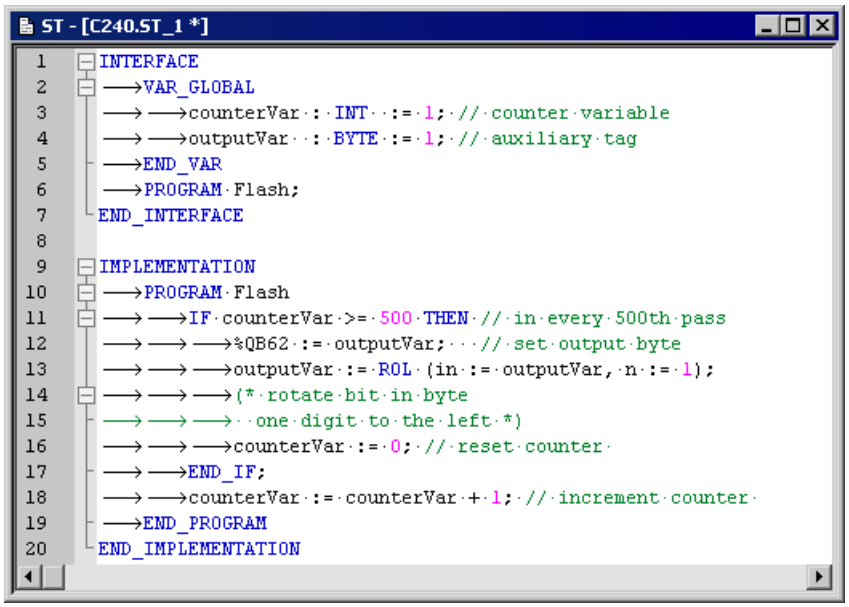



Figure 2-11 ST source file with visible spaces and tabs

### Procedure

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. Select the **View > Formatting symbols** context menu.

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.

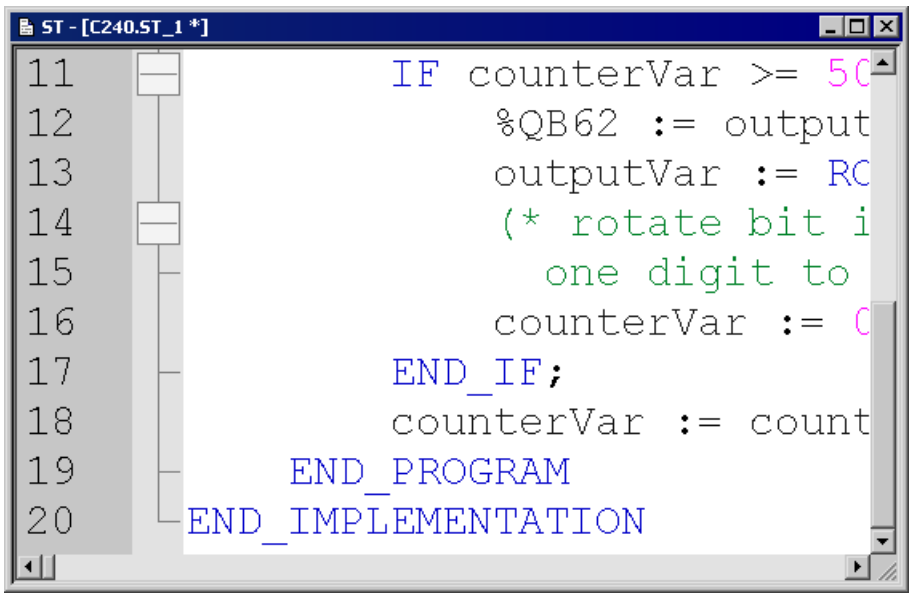



Figure 2-12 Increased size display of the ST source

## 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.
    Please also refer to the keyboard shortcuts (Page 52).

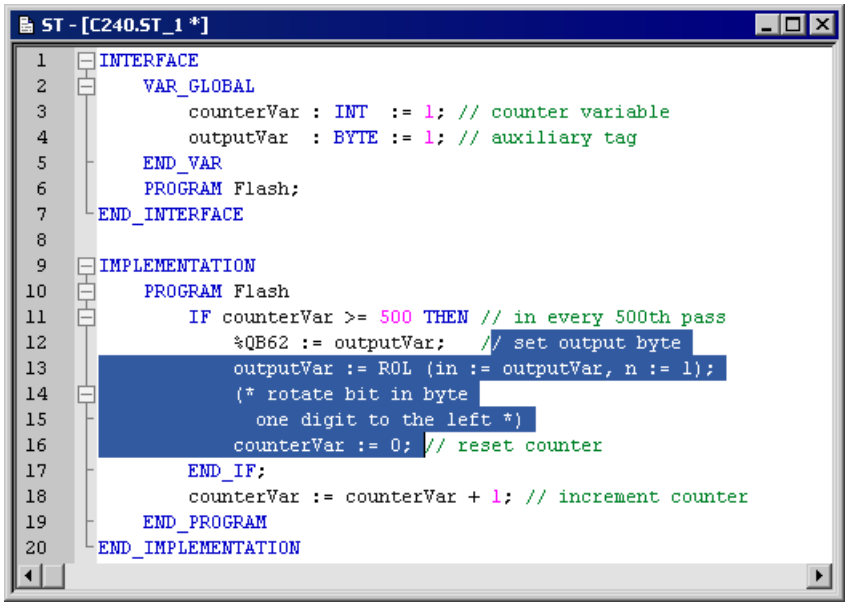



Figure 2-13 ST source with selected lines of text

### 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. Please also refer to the keyboard shortcuts (Page 52).

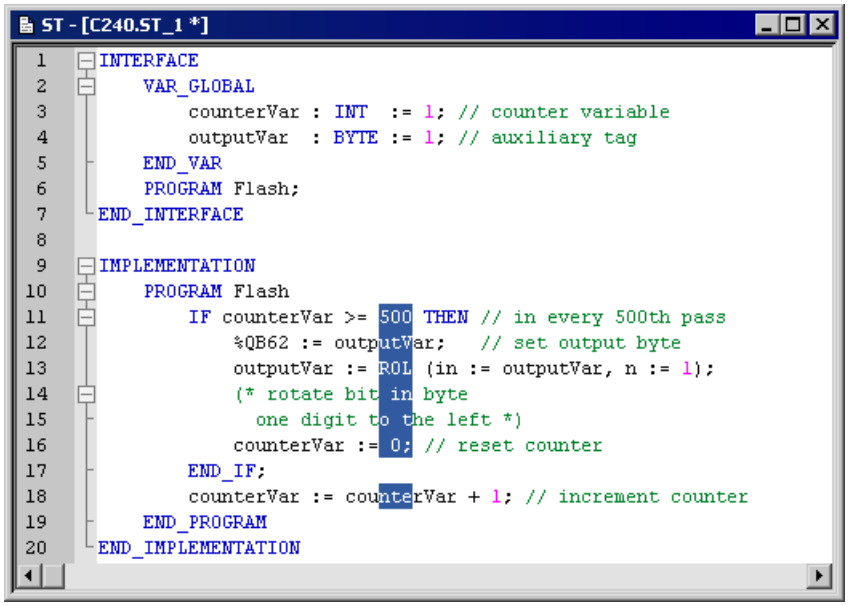



Figure 2-14 ST source with selected columns of text

### 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 Generating a simple series of numbers (generating a sequence)

You can generate simple series of integer numbers (sequences) that follow a specific pattern in the ST editor. This can be useful, for example, for the initialization of a field.

## Procedure

1. In the ST editor, select all the numbers that you want to replace with a series of numbers (column-by-column, for example); see Selecting text (Page 39).
2. Press the **CTRL+SHIFT+F7** shortcut.

The ST editor interprets all selected numbers as being possible elements and attempts to identify the pattern of the series from the initial elements and to calculate the subsequent elements (see below). The initial elements of the series must represent integer values. After the calculation has been performed, the ST editor replaces all the selected numbers which follow the initial elements with the calculated values.

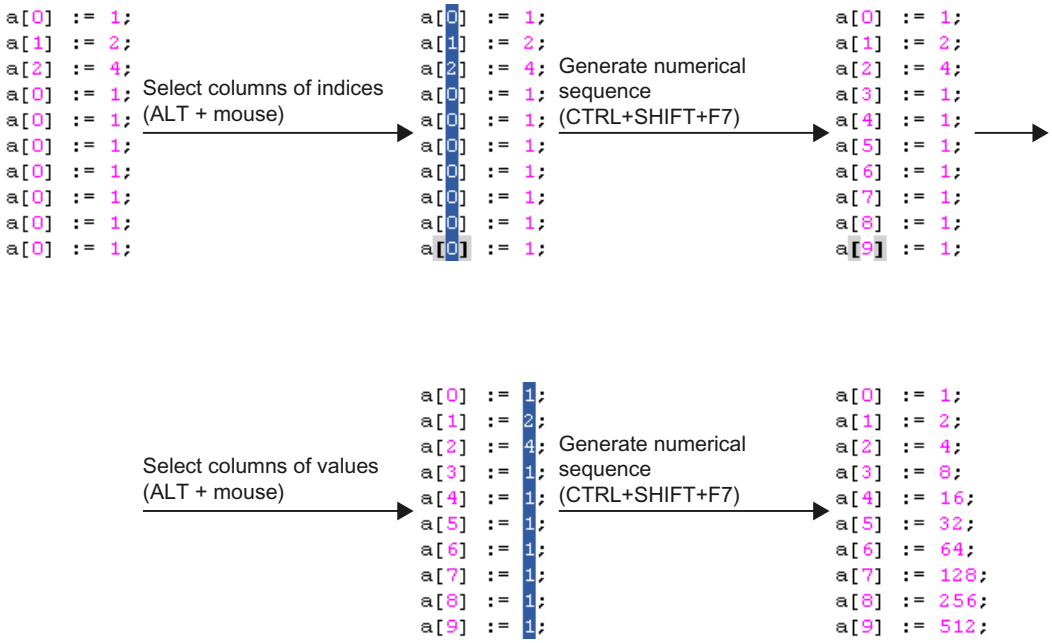



Figure 2-15 Example for generating series of numbers in the ST editor

## Calculation of the elements

All the numbers selected in the ST editor are interpreted as being possible elements. The initial elements from which the series is calculated must represent integer values.

When determining the series of numbers, it is assumed that an element $a_{n+1}$ can be determined from the previous element $a_n$ using a linear function with the integer coefficients j and k. This means the following equation applies:

$$a_{n+1} = f(a_n) = j \cdot a_n + k$$

Coefficients j and k are calculated from the initial elements in the series:

1. First of all, an attempt is made to determine the integer coefficients j and k from the three initial elements of the series of numbers $a_1$, $a_2$, and a3 ($j \geq 0$).
2. If this is not possible, only the first two elements $a_1$ and $a_2$ are used for the calculation:
   - Coefficient j will be set to 1.
   - Coefficient k will be determined from the resulting formula $a_2 = a_1 + k$.

These values are used to calculate the series of numbers. All selected numbers will be replaced by the calculated elements; the initial elements of the series which are used remain unchanged.

The following table shows some examples for series of numbers calculated in this manner.

Table 2- 2 Examples for linear series of numbers with specified initial elements

| Initial elements of the series | | | Calculated result | | | Resulting series of numbers |
|---|---|---|---|---|---|---|
| $a_1$ | $a_2$ | $a_3$ | j | k | Equation | |
| 0 | 1 | 2 | 1 | 1 | $a_{n+1} = a_n + 1$ | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 … |
| 1 | 2 | 3 | 1 | 1 | $a_{n+1} = a_n + 1$ | 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 … |
| 0 | 2 | 4 | 1 | 2 | $a_{n+1} = a_n + 2$ | 0, 2, 4, 6, 8, 10, 12, 14, 16, 18 … |
| 1 | 2 | 4 | 2 | 0 | $a_{n+1} = 2a_n$ | 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 … |
| 2 | 3 | 5 | 2 | -1 | $a_{n+1} = 2a_n - 1$ | 2, 3, 5, 9, 17, 33, 65, 129, 257, 513 … |
| 2 | 1 | 1 | 0 | 1 | $a_{n+1} = 1$ | 2, 1, 1, 1, 1, 1, 1, 1, 1, 1 … |
| 3 | 2 | 1 | 1 | -1 | $a_{n+1} = a_n - 1$ | 3, 2, 1, 0, -1, -2, -3, -4, -5, -6 … |
| 0 | 1 | 0 [1] | 1 | 1 | $a_{n+1} = a_n + 1$ | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 … |
| 0 | 0 | 2 [1] | 1 | 0 | $a_{n+1} = a_n$ | 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 … |

[1] Only the initial elements $a_1$ and $a_2$ of the series are used, $a_3$ is ignored and will be calculated.

**NOTICE**

The initial elements of the series used for the calculation must represent integer values. However, they can also be entered as floating-point numbers.

When the series is calculated, the numbers to be taken into account must all be selected.

The selected area must only contain numbers and certain special characters (e.g. opening and closing brackets, commas).

The first time an impermissible character is encountered in the selected area, the calculation of the series will be aborted.

See the examples in the following figure.

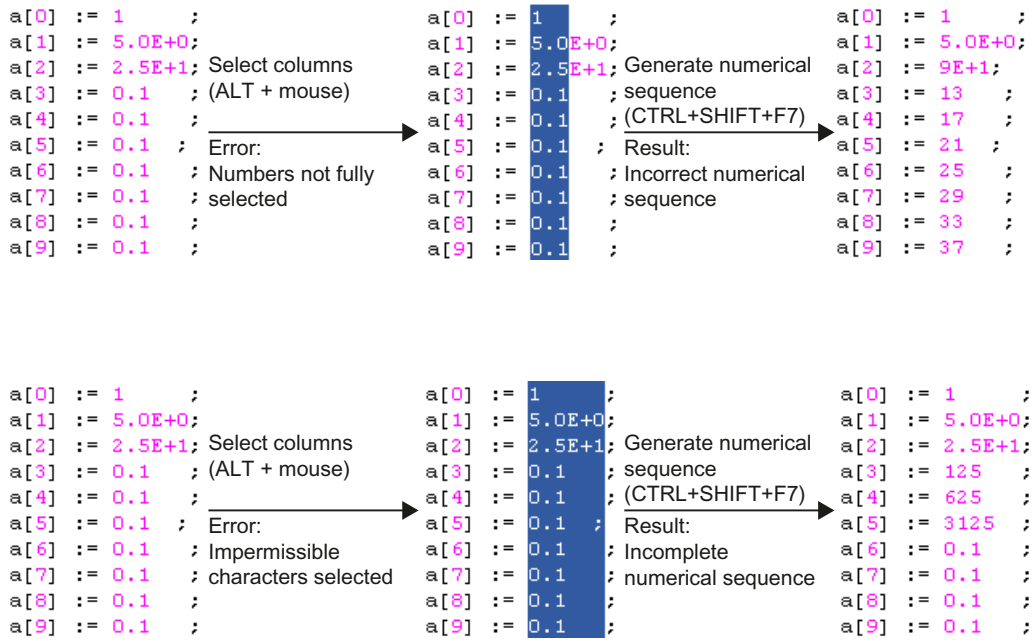



Figure 2-16 Examples of how incorrect selections cause unwanted results when generating a series of numbers in the ST editor

### 2.3.4.11 Use bookmarks

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

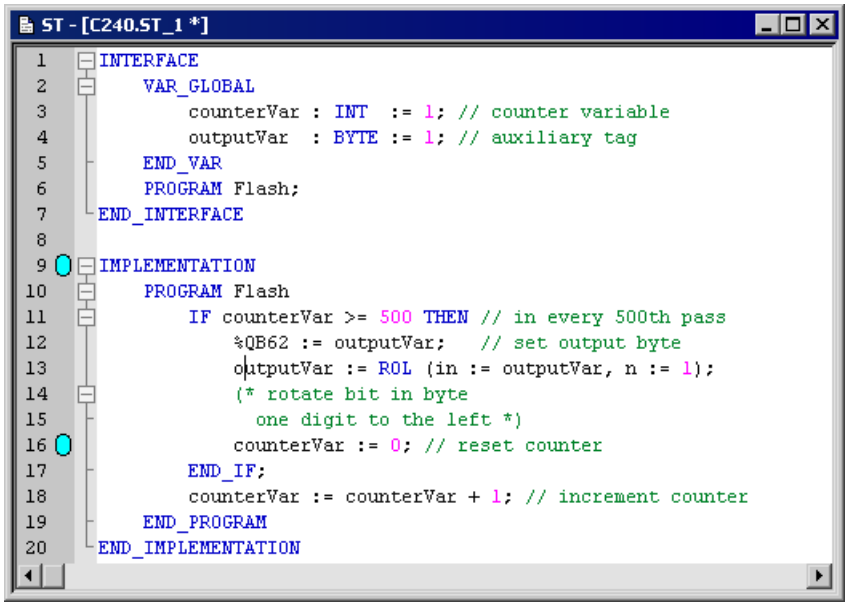



Figure 2-17 ST source with bookmarks

### 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 mouse:
  - Set the cursor in the appropriate line of the ST source.
  - Select the **Bookmarks > Insert/remove bookmark** context menu.

**Note**

Bookmarks are saved when the ST source file is closed.

### Jump to bookmark

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

- Select the **Bookmarks > Next bookmark** context menu.

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

- Select the **Bookmarks > Previous bookmark** context menu.

### Delete all bookmarks

How to delete all bookmarks in an ST source:

- Select the **Bookmarks > Remove all bookmarks** context menu.

### 2.3.4.12 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.

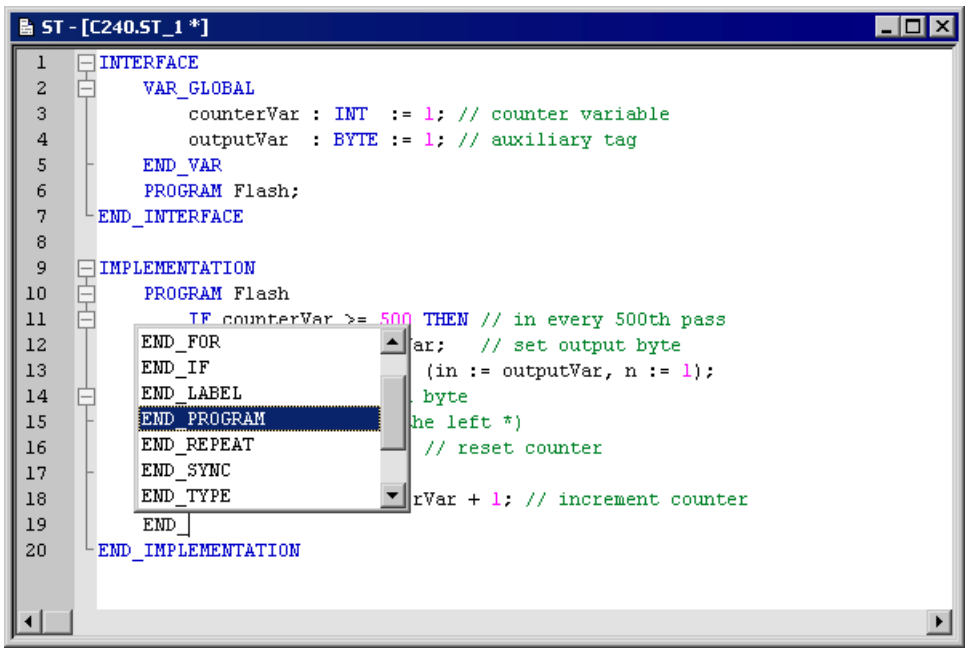



Figure 2-18 ST editor, automatic completion of an identifier (e.g. END_)

## Procedure

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. Expand or refine the selection possibilities which are displayed:
   - Enter additional characters.
   - Delete characters.
   - Use the left/right arrow keys to move the cursor.
4. Use the up/down arrow keys to select the required identifier.
5. Accept the identifier. The current word is overwritten.

**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

## 2.3.4.13 Opening a called block

From the ST editor you can directly open and edit a user-defined program organization unit (POU), such as a function or a function block, which is called in the ST source file.

1. Position the cursor in the identifier of a program organization unit (POU), e.g. when calling a function or declaring an instance of a function block.
2. Select the **Open called block** context menu.

The subsequent response will depend on the programming language in which the source file of the called POU was written:

- SIMOTION ST programming language:

  The corresponding ST source file is opened, if necessary. The cursor is positioned at the start of the relevant POU in the implementation section.

  The cursor jumps to the start of the POU in the implementation section within the same ST source file.

- SIMOTION MCC or SIMOTION LAD/FBD programming languages:

  The corresponding POU (MCC chart, LAD/FBD program) is opened in the associated editor.

### 2.3.4.14 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 highlights both brackets, where applicable. This simplifies the recognition of bracket pairs, in particular for nesting.

How to switch this function on or off:

- For the active ST source file:
  - Select the **View > Display bracket pairs** context menu.

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

#### Showing and hiding line numbering

Line numbers can be displayed in the ST editor:

How to switch this function on or off:

- For the active ST source file:
  - Select the **View > Line numbering** context menu.

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

#### Changing upper/lower case

You can change the case of selected text to upper case or lower case:

1. Select the text whose case you want to change; see Selecting text.
2. Select the **Selection in upper case** or **Selection in lower case** context menu.

#### Tool tip display

The ST editor provides support for some identifiers by displaying tool tips. These are shown if you briefly hover the cursor over the identifier.

For the identifiers of variables, for example, the associated data type is displayed in the tool tip. More tool tips will be available soon.

### 2.3.4.15 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.

### 2.3.4.16 ST editor toolbar

This toolbar contains important operating actions for programming:

Table 2- 3 ST editor toolbar

| Symbol | Meaning |
|---|---|
| | Accept and compile<br>Click this symbol to transfer the active ST source file to the project and compile it into executable code.<br>See: Starting the compiler (Page 55). |
| | Insert ST source file<br>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.<br>See: Insert ST source file (Page 21). |
| | Program status<br>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.<br>The following prerequisites are necessary:<br>1. The program must be compiled with the appropriate compiler option.<br>2. The project and the program must be loaded into the target system.<br>3. An online connection to the target system must have been established.<br>Reclick this icon to end the **program status**.<br>See: Using the program status (Page 323). |
| | Stop monitoring of the program variables<br>Click this icon in the **program status** test mode to stop the monitoring of the program variables.<br>See Using the program status (Page 323). |
| | Continue monitoring of the program variables<br>Click this icon in the **program status** test mode to continue the monitoring of the program variables.<br>See: Using the program status (Page 323). |
| | Refresh<br>Click this symbol in the **program status** test mode to force updating of the values displayed. The monitoring of the program variables must have been activated.<br>See: Using the program status (Page 323). |

| Symbol | Meaning |
|---|---|
| | Format current selection<br>Click this symbol to indent the blocks in the selected text area by one tab width, in accordance with the block hierarchy.<br>See Indentations and tabs (Page 30). |
| | Formatting symbols on/off<br>Click this symbol to show or hide blanks and tabs in the active ST source file.<br>See Displaying blanks and tabs (Page 37). |
| | Insert/remove bookmark<br>Click this symbol to set a bookmark in the current line of the active ST source file or to delete an existing bookmark.<br>See: Using bookmarks (Page 43). |
| | Next bookmark<br>Click this symbol to jump to the next bookmark in the active ST source file.<br>See: Using bookmarks (Page 43). |
| | Previous bookmark<br>Click this symbol to jump to the previous bookmark in the active ST source file.<br>See: Using bookmarks (Page 43). |
| | Remove all bookmarks<br>Click this symbol to remove all bookmarks from the active ST source file.<br>See: Using bookmarks (Page 43). |
| | Go to start of block<br>Click this symbol to move the cursor to the start of the current or higher-level block. |
| | Go to end of block<br>Click this symbol to move the cursor to the end of the current block. |

### 2.3.4.17 ST editor context menu

The ST editor context menu is shown if you right-click in an open ST source file.

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

As far as editor functions are concerned, this context menu also applies to the script editor.

You can select the following functions:

Table 2- 4 ST editor and script editor context menu

| Function | Meaning/Note |
|---|---|
| Open called block [1] | If the cursor is positioned in the identifier of a user-defined program organization unit (POU), e.g. when calling a function or declaring an instance of a function block:<br>Select this command to open and edit the source of this POU (ST source file, MCC chart, LAD/FBD program).<br>See: Open called block (Page 46). |
| Cut | Select this command to cut the selected area from the ST source file and save it to the clipboard. |

| Function | | Meaning/Note |
|---|---|---|
| Copy | | Select this command to copy the selected area to the clipboard. |
| Inserting | | Select this command to insert the contents of the clipboard into the ST source file at the current cursor position. |
| Delete | | Select this command to delete the selected area or the character to the right of the cursor. |
| Select all | | Select this command to select all of the text in the ST source file.<br>See: Select text (Page 39). |
| Assign parameters to a call | | Available soon. |
| **View** | | |
| | Line numbering on/off | Select this command to show or hide the line numbers in the active ST source file.<br>See: Other ST editor tools (Page 47). |
| | Indent help | Select this command to highlight the indents and outdents for blocks in the active ST source file by means of vertical auxiliary lines (in accordance with the set tab width).<br>See: Indentations and tabs (Page 30). |
| | Display bracket pairs | Select this command to highlight both brackets of a pair in the active ST source file, if the cursor is positioned at one of the two brackets.<br>See: Other ST editor tools (Page 47). |
| | Formatting symbols | Select this command to show or hide blanks and tabs in the active ST source file.<br>See: Displaying blanks and tabs (Page 37). |
| | Folding | Select this command to activate or deactivate folding in the active ST source file. The folding information in the active ST source file is then displayed or hidden.<br>See: Folding (showing and hiding blocks) (Page 33). |
| | Format current selection | Select this command to indent the blocks in the selected text area by one tab width, in accordance with the block hierarchy.<br>See: Indentations and tabs (Page 30). |
| | Split window | Select this command to split the active window of the ST editor into two segments horizontally, giving you two views of the same ST source file.<br>See: Splitting the editor window (Page 36). |
| **Bookmarks** | | |
| | Insert/remove bookmark | Select this command to set a bookmark in the current line of the active ST source file or to delete a bookmark which has been set there.<br>See: Using bookmarks (Page 43). |
| | Next bookmark | Select this command to jump to the next bookmark in the active ST source file.<br>See: Using bookmarks (Page 43). |
| | Previous bookmark | Select this command to jump to the previous bookmark in the active ST source file.<br>See: Using bookmarks (Page 43). |
| | Remove all bookmarks | Select this command to remove all bookmarks from the active ST source file.<br>See: Using bookmarks (Page 43). |

| Function | Meaning/Note |
|---|---|
| Selection in upper case | Select this command to change the selected text to upper case.<br>See: Other ST editor tools (Page 47). |
| Selection in lower case | Select this command to change the selected text to lower case.<br>See: Other ST editor tools (Page 47). |
| Indent selected area | • If no text is selected:<br>Select this command to move the text on the right of the cursor to the next tab position.<br>• If text is selected in one single line:<br>Select this command to delete the selected text and move the subsequent text to the next tab position.<br>• If text is selected in multiple lines:<br>Select this command to indent the selected area (Page 30).<br>A tab character ($09) or the equivalent number of spaces ($20) will be inserted, depending on the settings for the ST editor (Page 29). |
| Undo selected area | • If no text is selected:<br>Select this command to move the cursor to the previous tab position.<br>• If text is selected in one single line:<br>Select this command to cancel the selection and move the cursor to the previous tab position.<br>• If text is selected in multiple lines:<br>Select this command to outdent the selected area (Page 30). |
| Go to start of block | Select this command to move the cursor to the start of the current or higher-level block. |
| Go to end of block | Select this command to move the cursor to the end of the current block. |
| Go to start of block, level 0 | Select this command to move the cursor to the start of the higher-level block, 1st level. |
| Go to start of block, level 1 | Select this command to move the cursor to the start of the higher-level block, 2nd level. |
| Set/remove breakpoint [1] | Select this command to set a breakpoint at the selected code position or to remove an existing breakpoint.<br>See: Setting breakpoints (Page 332). |
| Activate/deactivate breakpoint [1] | Select this command to activate or deactivate the breakpoint at the selected code position.<br>See: Activating breakpoints (Page 341). |

[1] Not available in the script editor context menu.

### 2.3.4.18 Shortcuts

The ST editor also provides keyboard shortcuts. Some commands can also be called via the **Edit** or **ST editor** menus:

The keyboard shortcuts related to editor functions also apply to the script editor.

Table 2- 5 Keyboard shortcuts for ST editor and script editor

| Shortcuts | Description |
|---|---|
| F2 | Jump to the next bookmark. |
| F3 | Find next (menu **Edit > Find next**). |
| F9 [1] | Set or remove a breakpoint (menu **Debug > Set/remove breakpoint**). |
| F12 [1] | Activate or deactivate a set breakpoint (menu **Debug > Activate/deactivate breakpoint**). |
| BACK | Delete the character to the left of the cursor. |
| INS | Switch between insert mode and overwrite mode. |
| DEL | Delete the selected area or the character to the right of the cursor (menu **Edit > Delete**). |
| Arrow key | Move the cursor. |
| POS1 | Move cursor to the beginning of the line. |
| END | Move cursor to the end of the line. |
| PG UP | Move up one page. The cursor follows. |
| PG DN | Move down one page. The cursor follows. |
| TAB | • If no text is selected:<br>Move the text on the right of the cursor to the next tab position.<br>• If text is selected in one single line:<br>Delete the selected text and move the subsequent text to the next tab position.<br>• If text is selected in multiple lines:<br>Indent selection.<br>A tab character ($09) or the equivalent number of spaces ($20) will be inserted, depending on the settings for the ST editor. |
| SHIFT+F2 | Jump to the previous bookmark. |
| SHIFT+BACK | Delete the character to the left of the cursor. |
| SHIFT+INS | Paste clipboard contents (menu **Edit > Paste**). |
| SHIFT+DEL | Cut the selected area (menu **Edit > Cut**). |
| SHIFT+Arrow key | Select line of text. |
| SHIFT+POS1 | Select text back to the beginning of the line. |
| SHIFT+END | Select text to the end of the line. |
| SHIFT+PG UP | Move up one page. Select lines of text up to the new cursor position. |
| SHIFT+PG DN | Move down one page. Select lines of text up to the new cursor position. |

| Shortcuts | Description |
|---|---|
| SHIFT+TAB | • If no text is selected:<br>Jump to the preceding tab position.<br>• If text is selected in one single line:<br>Jump to the preceding tab position.<br>• If text is selected in multiple lines:<br>Outdent selection. |
| CTRL+A | Select all text (menu **Edit > Select All**). |
| CTRL+B [1] | Accept and compile ST source file (menu **ST source file > Accept and compile**). |
| CTRL+C | Copy the selected area to the clipboard (menu **Edit > Copy**). |
| CTRL+D | Duplicate the current line or the area selected. |
| CTRL+F | Find text in ST source file (menu **Edit > Find**) |
| CTRL+H | Replace text in ST source file (menu **Edit > Replace**). |
| CTRL+J | Display the next search result in the project-wide search (menu **Edit > Display next position**). |
| CTRL+L | Copy the current line or the selected area to the clipboard. |
| CTRL+U | Change selected text to lower case. |
| 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 remove bookmarks. |
| CTRL+F4 | Close the active window (e.g. menu **ST source file > Close**). |
| CTRL+F5 [1] | Remove all the breakpoints (in all the program source) in the SIMOTION device (menu **Debug > Remove all breakpoints**). |
| CTRL+F7 [1] | Activate or deactivate the program status function (menu **ST source file > Program status on/off**). |
| CTRL+F8 [1] | Continue to execute the program at the activated breakpoint (menu **Debug > Continue**). |
| CTRL+BACK | Delete the word to the left of the cursor. |
| CONTROL+INS | Copy the selected area to the clipboard (menu **Edit > Copy**). |
| CTRL+DEL | Delete the word to the right of the cursor. |
| CTRL+arrow key (left/right) | Move cursor left or right by one word. |
| CTRL+arrow key (up/down) | Move up or down one page. The cursor remains in the same position for as long as it is visible in the window. |
| CTRL+POS1 | Move cursor to the beginning of the ST source file. |
| CTRL+END | Move cursor to the end of the ST source file. |
| CTRL+SHIFT+B | Highlight bracket pairs in the current ST source file. |
| CTRL+SHIFT+F | Search for texts within the project (menu **Edit > Search in the project**) |
| CTRL+SHIFT+G | Replace texts within the project (menu **Edit > Replace in the project**) |
| CTRL+SHIFT+H | Assign parameters to a call. Available soon. |
| CTRL+SHIFT+U | Change selected text to upper case. |
| CTRL+SHIFT+F2 | Remove all bookmarks from the ST source file. |

| Shortcuts | Description |
|---|---|
| CTRL+SHIFT+F3 | Arrange windows, tile horizontally. |
| CTRL+SHIFT+F5 | Arrange windows, tile vertically. |
| CTRL+SHIFT+F7 | Generate a simple series of numbers (sequence) in the selected area. |
| 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+SHIFT+BACK | Delete text to the left of the cursor up to the beginning of the line. |
| CTRL+SHIFT+DEL | Delete text to the right of the cursor up to the end of the line. |
| CTRL+SHIFT+arrow key (left/right) | Select word to the left or right of the cursor. |
| CTRL+SHIFT+POS1 | Select lines of text back to the beginning of the ST source file. |
| CTRL+SHIFT+END | Select lines of text up to the end of the ST source file. |
| CTRL+ALT+C | Folding: Hide all blocks of the current ST source file. |
| CTRL+ALT+D | Folding: Show all blocks of the current ST source file. |
| CTRL+ALT+F | Folding: Show 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 | Show or hide line numbers in the current ST source file. |
| CTRL+ALT+O [1] | If the cursor is in the identifier of a program organization unit (POU): Open called block, i.e. open program source of the POU and position the cursor. |
| CTRL+ALT+R | Folding: Show all subordinate blocks. |
| CTRL+ALT+S | Split window or cancel split (menu **ST source file > Split window**). |
| CTRL+ALT+T | Folding: Show/hide block. |
| CTRL+ALT+V | Folding: Hide all subordinate blocks. |
| CTRL+ALT+W | Show or hide blanks 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+L | Change selected text to upper case. |
| ALT+SHIFT+U | Change selected text to lower case. |
| ALT+SHIFT+Arrow key | Select text by column. |
| ALT+SHIFT+POS1 | Select columns of text back to the beginning of the line. |
| ALT+SHIFT+END | Select columns of text to the end of the line. |
| ALT+SHIFT+Pg Up | Move down one page. Select columns of text up to the new cursor position. |
| ALT+SHIFT+PG DN | Move down one page. Select columns of text up to the new cursor position. |

[1] Keyboard shortcut does not apply to the script editor.

Table 2- 6 Combined keyboard and mouse actions for ST editor and script editor

| 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. |
| | Turn mouse wheel | Scroll vertically. Cursor remains unchanged. |
| SHIFT | Single left-click in text | Select line of text. |
| SHIFT | Turn mouse wheel | Scroll horizontally. |
| CTRL | Single left-click on line number | Select all text (menu **Edit > Select All**). |
| CTRL | Single left-click in bookmark column | Set or delete 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. |

## 2.3.5 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 56)
- Locally for an individual ST source within the SIMOTION project, see Local settings of the compiler (Page 59)

### 2.3.6.1 Global compiler settings

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

#### Procedure

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

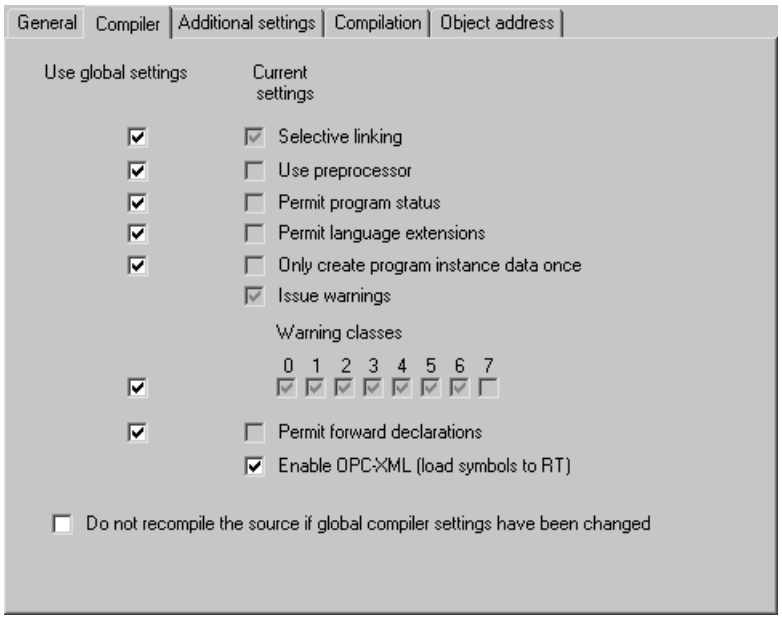



Figure 2-19 Global compiler settings

## Parameter

Table 2- 7 Parameters for global compiler settings

| Parameter | Description |
|---|---|
| Display warning classes[1] | In addition to error messages, the compiler can issue warnings and information. You can set the scope of the warning messages to be output:<br>**Active**: The compiler issues warnings and information for the selected class.<br>**Inactive**: The compiler suppresses warnings and information for the respective class.<br>See also Meanings of the warning classes (Page 65). |
| Selective linking[1] | **Active** (standard): Unused code is removed from the executable program.<br>**Inactive**: Unused code is retained in the executable program. |
| Use preprocessor[1] | **Active**: Preprocessor is used (see Controlling the preprocessor (Page 290)).<br>**Inactive** (standard): Preprocessor is not used. |
| Enable program status[1] | **Active**: Additional program code is generated to enable monitoring of program variables (including local variables).<br>**Inactive** (standard): Program status not possible.<br>See Properties of the program status (Page 321). |
| Permit language extensions[1] | **Active**: Language elements are permitted that do not comply with IEC 61131-3.<br>• Direct bit access to variables of a bit data type (Page 137)<br>• Accessing the input parameter of a function block while outside the function block (Page 183)<br>• Calling a program while in a different program (Page 188)<br>**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. This setting is required for calling a program while inside a different program (Page 188).<br>**Inactive** (standard): The local variables of a program are stored according to the task assignment in the user memory of the respective task.<br>See Memory areas of the variable types (Page 223). |
| Permit forward declarations (ST) | Only for the ST programming language.<br>Forward declarations enable you to use program organization units (POUs) before they are fully defined. Prototype declarations of POUs are possible prior to their use, but only required for an instance declaration of a function block.<br>**Active**: Forward declarations are permitted.<br>**Inactive** (standard): Forward declarations are not permitted.<br>See Forward declarations (Page 300).<br>Forward declarations are always permitted with the MCC and LAD/FBD programming languages.<br>It is also possible to make a local setting on the ST source file (Page 59). Please also refer to the description of the effectiveness of global or local compiler settings (Page 63). |

| Parameter | Description |
|---|---|
| Permit single step (MCC) | Only for the MCC programming language.<br>**Active**: An additional program code is created which enables individual program steps to be monitored.<br>**Inactive**: Single step is not possible.<br>This function facilitates debugging of your program.<br>See "Tracking single steps in the program" in the MCC Programming Manual.<br>It is also possible to make a local setting on the MCC unit. Please also refer to the description of the effectiveness of global or local compiler settings (Page 63). |
| Permit trace (MCC - only for RT versions >= 4.2) | Only for the MCC programming language and for SIMOTION Kernel as of version V4.2.<br>**Active**: An additional program code is created which enables monitoring of the program execution in program branches which are executed cyclically.<br>**Inactive**: Trace is not possible.<br>This function facilitates debugging of your program.<br>See "Tracking program execution per trace" in the MCC Programming Manual.<br>It is also possible to make a local setting on the MCC unit. Please also refer to the description of the effectiveness of global or local compiler settings (Page 63). |
| Display all messages with "Save and compile changes"[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 changes** command in SIMOTION SCOUT.<br>**Active**: A detailed log is created that is similar to that for single compilation of an ST source file.<br>**Inactive**: A compressed error log is created. |
| Optimize execution order (LAD/FBD) | Only for LAD/FBD programming languages.<br>**Active**: LAD/FBD networks are calculated in the optimized execution order.<br>**Inactive**: LAD/FBD networks are calculated in the non-optimized execution order. |
| [1] Local settings also possible, see Local settings of the compiler (Page 59). Please also refer to the description of the effectiveness of global or local compiler settings (Page 63).<br>[2]User-specific settings. Valid for all SIMOTION projects that the user processes. | |

| NOTICE |
|---|
| You may have to save and compile 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.

## Procedure

1. Open the Properties window for the ST source file, see Changing the properties of an ST source file (Page 25):

   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**.

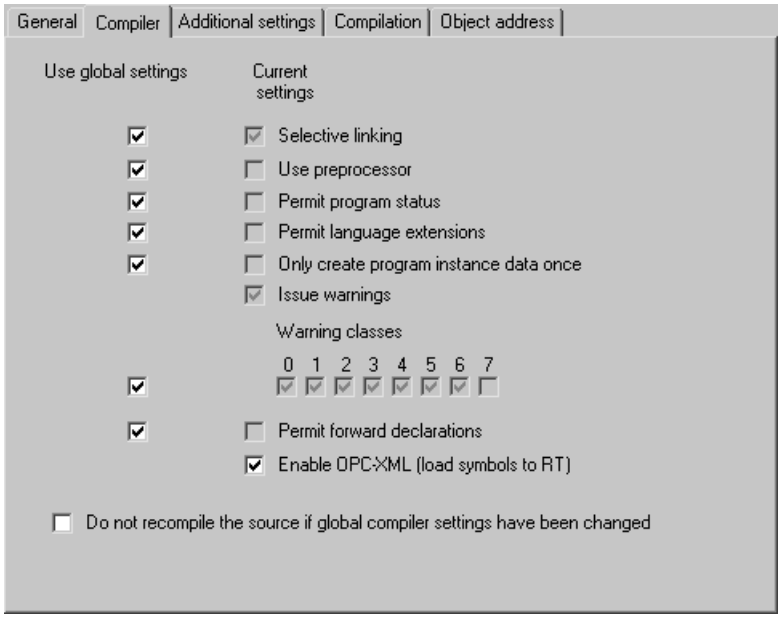



Figure 2-20 Local compiler settings for the ST source file

## Parameter

Table 2- 8 Parameters for the local compiler settings for the ST source file

| Parameter | Description |
|---|---|
| Use global settings | This checkbox is available for every parameter which also has a global setting. It can only be selected when the "Do not recompile the source if global compiler settings have been changed" checkbox is inactive. This is where you define whether the global settings are adopted or whether the local settings will apply.<br>See the description under "Effectiveness of global or local compiler settings (Page 63)".<br>Use the second checkbox or the other checkboxes for the relevant parameters (described below) to define the local settings. |
| Selective linking[1] | **Active**: Unused code is removed from the executable program.<br>**Inactive**: Unused code is retained in the executable program.<br>**Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active). |
| Use preprocessor[1] | **Active**: Preprocessor is used.<br>**Inactive**: Preprocessor is not used.<br>**Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active).<br>See Controlling the preprocessor (Page 290). |
| Enable program status[1] | **Active**: Additional program code is generated to enable monitoring of program variables (including local variables).<br>**Inactive**: Program status not possible.<br>**Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active).<br>See Properties of the program status (Page 321). |
| Permit language extensions[1] | **Active**: Language elements are permitted that do not comply with IEC 61131-3.<br>• Direct bit access to variables of a bit data type (Page 137)<br>• Accessing the input parameter of a function block while outside the function block (Page 183)<br>• Calling a program while in a different program (Page 188)<br>**Inactive**: Only language elements are permitted that comply with IEC 61131-3.<br>**Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active). |
| 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. This setting is required for calling a program while inside a different program (Page 188).<br>**Inactive**: The local variables of a program are stored according to the task assignment in the user memory of the respective task.<br>**Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active).<br>See Memory areas of the variable types (Page 223).<br>For further information, refer to the SIMOTION Basic Functions Function Manual. |

| Parameter | Description |
|---|---|
| Issue warnings<br>Warning classes[1] | In addition to error messages, the compiler can issue warnings and information. You can set the scope of the warning messages to be issued.<br>"Issue warnings" checkbox:<br>**Active**: The compiler issues the warnings and information according to the warning class selection that follows.<br>**Inactive**: The compiler suppresses all warnings and information concerning this unit. The checkboxes for the warning classes are hidden.<br>**Gray background** (display only): Operating on a global setting basis, the compiler always issues warnings and information in accordance with the global warning class selection shown below (if "Use global settings" = active).<br>"Warning classes" checkboxes (only if "Issue warnings" = active):<br>**Active**: The compiler issues warnings and information for the selected class.<br>**Inactive**: The compiler suppresses warnings and information for the respective class.<br>**Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active).<br>See also Meanings of the warning classes (Page 65). |
| Permit forward declarations[1] | Forward declarations enable you to use program organization units (POUs) before they are fully defined. Prototype declarations of POUs are possible prior to their use, but only required for an instance declaration of a function block.<br>**Active**: Forward declarations are permitted.<br>**Inactive** (standard): Forward declarations are not permitted.<br>**Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active).<br>See Forward declarations (Page 300). |
| Enable OPC-XML | **Active:** Symbol information for the unit variables of the ST source file is available in the SIMOTION device.<br>This is required for:<br>• The *_exportUnitDataSet* and *_importUnitDataSet* functions; see the SIMOTION Basic Functions Function Manual<br>• The watch function of IT DIAG.<br>**Inactive:** Symbol information is not created. |

| Parameter | Description |
|---|---|
| Do not recompile unit if global compiler settings have been changed | **Active:** The global settings of the compiler have no effect for all parameters. The "Use global settings" checkboxes cannot be selected and are grayed out. When changing the global compiler settings, the ST source file is not recompiled.<br>This setting is required for libraries that have a know-how protection program (Page 67) with a "High" level:<br>• For all program sources in this library.<br>• For the library itself.<br>**Inactive:** The "Use global settings" checkboxes can be selected for all parameters and are displayed with a white background. These checkboxes specify whether the global properties are taken over for the corresponding parameters.<br>This setting means that in the following case the ST source file is compiled even though all "Accept global settings" checkboxes are inactive:<br>• The global settings of the compiler have been changed and the menu **Project > Save and compile changes** is selected.<br>See the description under "Effectiveness of global or local compiler settings (Page 63)". |
| [1] Global setting also possible, see Global settings of the compiler (Page 56). Please also refer to the description of the effectiveness of global or local compiler settings (Page 63). | |

### 2.3.6.3 Effectiveness of global or local compiler settings

You control the effectiveness of the global and local settings of a parameter using the local compiler settings (Page 59).

#### "Do not recompile the source if global compiler settings have been changed" checkbox

With the "Do not recompile the source if global compiler settings have been changed" checkbox, you specify whether the global properties of the compiler influence the program source.

- Active: The global settings of the compiler have no effect for all parameters. The "Use global settings" checkboxes cannot be selected and are grayed out. When changing the global compiler settings, the ST source file is not recompiled.

  This setting is required for libraries that have a know-how protection program (Page 67) with a "High" level:

  – For all program sources in this library.

  – For the library itself.

- Inactive: The "Use global settings" checkboxes can be selected for all parameters and are displayed with a white background. These checkboxes specify whether the global properties are taken over for the corresponding parameters.

  This setting means that in the following case the program source is compiled even though all "Accept global settings" checkboxes are inactive:

  – The global settings of the compiler have been changed and the menu **Project > Save and compile changes** is selected.

  This causes problems for libraries that have a know-how protection program (Page 67) with a "High" level: Online inconsistencies can also occur for projects that have been converted from an earlier version of SIMOTION SCOUT.

### Checkboxes for the other parameters

Every parameter that has a global setting also has at least two checkboxes:

- "Accept global settings" checkbox:

  You use this checkbox to define whether global settings are to be accepted. It can only be selected when the "Do not recompile the source if global compiler settings have been changed" checkbox is inactive.

  The following settings are possible:

  – **Active**:

    The global settings are accepted for the corresponding parameter.

    No other checkboxes may be selected for this parameter; they are grayed out and show the global setting.

  – **Inactive**:

    The global settings are ignored for the corresponding parameter. Only the local settings, which you define with the other checkboxes, are effective.

- One or more checkboxes for local settings:

  The function of these checkboxes depends on the "Accept global settings" checkbox:

  – If the "Accept global settings" checkbox is active:

    The global settings are accepted for the corresponding parameter. The checkboxes for the local settings can no longer be selected; they are grayed out and show the global setting.

  – If the "Accept global settings" checkbox is inactive:

    The checkboxes for the local settings can be selected and are displayed on a white background. You can use them to define the local settings for the corresponding parameter. The global settings are ignored.

This behavior applies to the following compiler settings:

- Selective linking
- Use preprocessor
- Enable program status
- Permit language extensions
- Only create program instance data once
- Display warnings with warning classes
- Permit forward declarations (only for the ST programming language)
- Permit single step (only in MCC programming language)
- Permit trace (only for the MCC programming language and for SIMOTION Kernel as of version V4.2)

**Note**

You can check the current compiler options which will be effective the next time the program source is compiled.

- To do this, select the "Additional settings" tab (Page 65) in the Properties window of the program source.

### 2.3.6.4 Meanings of the warning classes

The table lists the warning classes and their meanings.

Table 2- 9 Meanings of the 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 |

The detailed description of the compiler error messages specifies which warning classes are assigned to the individual warnings (16001 - 16700) (Page 419) and information (32010 - 32653) (Page 424).

### 2.3.6.5 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 25) is open.

#### Procedure

To display the current compiler options using the global or local settings of the compiler (Page 56):

- 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<br>-C lang_iec | "Permit language extensions"[1] active.<br>"Permit language extensions" inactive. |
| -C opcsym<br>-C no_opcsym | "Permit OPC-XML"[1] active.<br>"Permit OPC-XML" inactive. |
| -C opcsym<br>-C no_preproc | "Use preprocessor"[1] active.<br>"Use preprocessor" inactive. |
| -C prog_once<br>-C prog_multi | "Only create program instance data once"[1] active.<br>"Only create program instance data once" inactive. |
| -C scan_twice<br>-C scan_once | "Permit forward declarations"[1] active.<br>"Permit forward declarations" inactive. |
| -D *text* | Preprocessor definition (Page 67). |
| -e user[2]<br>-e local | Only global settings act.<br>"Do not recompile the source if global compiler settings have been changed"[1] active. Only local settings act.<br>No details (default): "Do not recompile the source if global compiler settings have been changed" inactive. Global settings will be augmented with local settings. |
| -I[2] | Accept the package settings from device or library. |
| -l sel<br>-l no_sel | "Selective linking"[1] active.<br>"Selective linking" inactive. |
| -s<br>-s_off | "Enable program status"[1] active.<br>"Enable program status" inactive. |
| -w no_warn<br>-w all_warn<br>-w *n*_off<br>-w *n*_on | "Suppress warnings"[1] active.<br>Display all warnings.<br>Warning class *n* active[1].<br>Warning class *n* inactive[1]. |
| Further options | Internal options of the SIMOTION compiler. |
| [1] Meaning of the compiler option: see "Local compiler settings (Page 59)".<br>[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.

The SIMOTION online help provides additional information on know-how protection.

**Note**

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

### See also

Know-how protection for libraries (Page 273)

## 2.3.8 Making preprocessor definitions

You can make definitions for the preprocessor (see Controlling the preprocessor (Page 290)) in the Properties dialog box of the ST source file. This enables you to control the preprocessor for ST source files with know-how protection too (see Know-how protection for ST source files (Page 67)).

### 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 file (Page 25)):

   Select the ST source file in the project navigator, followed by 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**.

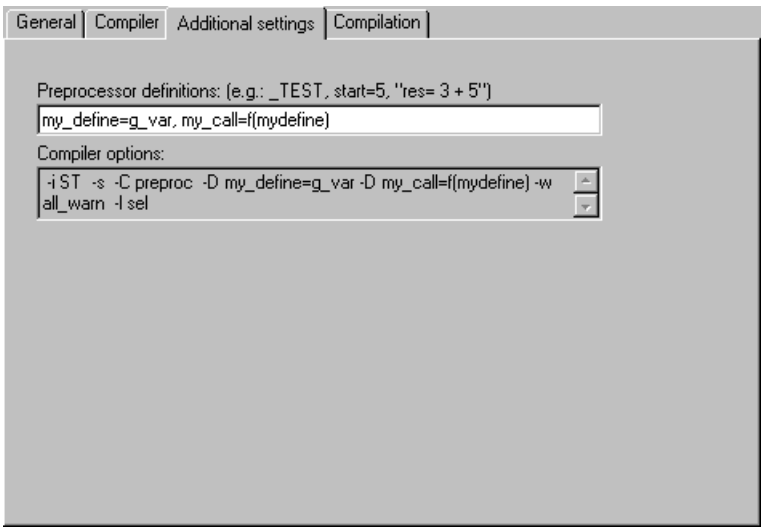



Figure 2-21 Preprocessor definitions

Table 2- 10 Syntax of the preprocessor definitions

| Syntax | Meaning |
|---|---|
| *Identifier=text*<br>*'Identifier=text'*<br>*"Identifier=text"* | The specified *identifier* is defined and replaced in the ST source file by the specified *text*.<br>Permissible characters: See table footnote.<br>If the expression contains blanks (e.g. in the text), the syntax *"Identifier=text"* must be used. |
| *Identifier* | The specified *identifier* is defined and replaced in the ST source file by blank text.<br>Permissible characters: See table footnote. |
| Multiple preprocessor definitions are separated by commas: *Definition_1, Definition_2, …*<br>Permissible characters:<br>• For *identifiers*: 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.<br>• 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, which are made within an ST source file with pragmas, overwrite the definitions in the Properties dialog box.

Note the information relating to preprocessor statements (Page 291).

## 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)

You can export an ST source file as a text file in ASCII format and then either reimport this file as an ST source file or edit it with any ASCII editor.

#### Procedure

To export an ST source file as an ASCII file:

1. Open the ST source file (Page 24), entering the password if necessary (for ST source files with know-how protection (Page 67)).
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 67) and which are not already open:

   If the user with the log-in details assigned to the ST source file has not yet logged in:

   – Enter the corresponding password for the displayed login.

     The know-how protection for this unit is temporarily canceled (for this export).

   – If required, activate the "Use login as a standard login" checkbox.

     You will be logged in with this login and can now export or open additional units 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.

**Note**

Folding information (Page 33) and bookmarks (Page 43) are not exported.

### 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 encrypted XML files, the know-how protection, including login and password, is retained.

Folding information (Page 33) and bookmarks (Page 43) are not exported.

### 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.

Alternative:

1. In the project navigator, select the **PROGRAMS** folder.
2. In the context menu, select **Import object**.
3. Select the XML data to be imported.

   A new ST source file is created, and the XML data is imported. This ST source file is assigned the name which is saved in the XML data; if a naming conflict occurs, it is automatically renamed. The entire project is saved and recompiled.

**Note**

Note that if the XML data to be imported was exported from an ST source file that was know-how protected: When importing the encrypted XML data, the know-how protection, including login and password, is retained.

#### 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.

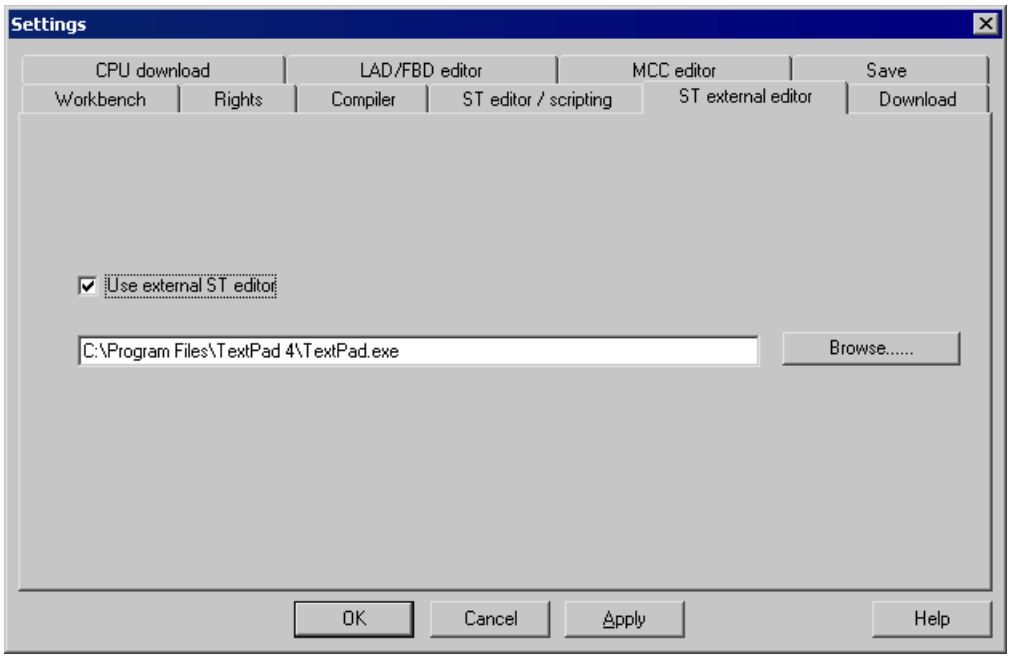



Figure 2-22 Settings for the use of an external editor

## 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- 11 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. |
| Properties | Select this command to display the properties of the active ST source file. Several tabs are provided to make local settings for this source.<br>See: Changing the properties of an ST source file (Page 25). |
| Accept and compile | Choose this command to transfer the current ST source file to the project and compile into executable code.<br>See: Starting the compiler (Page 55). |
| Execute 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).<br>See: Exporting an ST source file as a text file (ASCII) (Page 69). |
| Split window | Select this command to split the active window of the ST editor into two segments horizontally, giving you two views of the same ST source file.<br>See: Splitting the editor window (Page 36). |
| Program status on/off | Select this command to start the **program status** test mode. During the program execution, you can monitor the values of the variables marked in the ST source.<br>The following prerequisites are necessary:<br>1. The program must be compiled with the appropriate compiler option.<br>2. The project and the program must be loaded into the target system.<br>3. An online connection to the target system must have been established.<br>Select the command again to close the **program status**.<br>See: Using the program status (Page 323). |

| Function | Meaning/Note |
|---|---|
| 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. |

### 2.3.11.2 ST source file context menu

The ST source file context menu is shown if you select an ST source file in the project navigator and then right-click it.

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- 12 ST source file context menu

| Function | | Meaning/Note |
|---|---|---|
| Open | | Select this command to open the selected ST source file.<br>See: Opening an existing ST source file (Page 24). |
| Cut | | The selected ST source files are deleted and saved on the clipboard. |
| Copy | | The selected ST source files are copied to the clipboard. |
| Paste | | The contents of the clipboard are inserted in the selected folder. |
| Delete | | The selected ST source file is deleted, including all the data. |
| Rename | | Select this command in order to change the name of the selected 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 Rules for identifiers (Page 89). |
| 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 | Select this command to import XML data to the selected ST source file from an ST source file which you have previously exported to another project. The existing data in the ST source file being imported is overwritten.<br>See: Importing XML data into an ST source file (Page 70). |
| | Save project and export object | Select this command to export the selected ST source file in XML format. You can import the exported data into other projects.<br>See: Exporting an ST source file in XML format (Page 70). |

| Function | Meaning/Note |
|---|---|
| Accept and compile | Choose this command to transfer the current ST source file to the project and compile into executable code.<br>See: Starting the compiler (Page 55). |
| Execute 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 | Select this command to start the **program status** test mode. During the program execution, you can monitor the values of the variables marked in the ST source file.<br>The following requirements are necessary:<br>1. The program must be compiled with the appropriate compiler option.<br>2. The project and the program must be loaded into the target system.<br>3. An online connection to the target system must have been established.<br>Select the command again to close the **program status**.<br>See: Using the program status (Page 323). |
| Export | Select this command to export the selected ST source file as a text file (ASCII).<br>See: Exporting an ST source file as a text file (ASCII) (Page 69). |
| **Know-how protection** | |
| Set | Select this command to protect the selected ST source file from unauthorized access by third parties. Protected ST source files can only be opened or exported as plain text files by entering a password.<br>See: Know-how protection for ST source files (Page 67). |
| Delete | Select this command to cancel the know-how protection for the selected source file permanently. The password needs to be entered in order to do this.<br>See: Know-how protection for ST source files (Page 67). |
| **Reference data** | |
| Create | Select this command in order to create the reference data (cross-reference list, program structure, code attributes) for the selected ST source file.<br>The cross-reference list for the selected ST source file appears after this command is executed.<br>See: Reference data (Page 282), creating a cross-reference list (Page 282). |
| **Display** | |
| Cross references | The cross-reference list for the selected ST source file is formed from the reference data created previously and is displayed. The cross-reference list contains the declaration and uses all the identifiers for the selected ST source file.<br>See: Content of the cross-reference list (Page 283). |
| Program structure | The program structure for the selected ST source file is formed from the reference data created previously and is displayed. The program structure contains all the subprogram calls and their nesting within the selected ST source file.<br>See: Content of the program structure (Page 287). |

| Function | | | Meaning/Note |
|---|---|---|---|
| | | Code attributes | The code attributes for the selected ST source file are formed from the reference data created previously and are displayed. The code attributes contain information about the storage requirements of various data areas of the selected ST source file.<br>See: Code attribute contents (Page 288). |
| Print | | | Select this command to print the selected ST source file. You can choose whether you wish to print the text of the ST source file and/or their properties. |
| Print preview | | | Choose this command to generate a preview of the expected print output. |
| Properties | | | Select this command to display the properties of the selected ST source file. Several tabs are provided to make local settings for this source file.<br>See: Changing the properties of an ST source file (Page 25). |

## 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 305).

### 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.

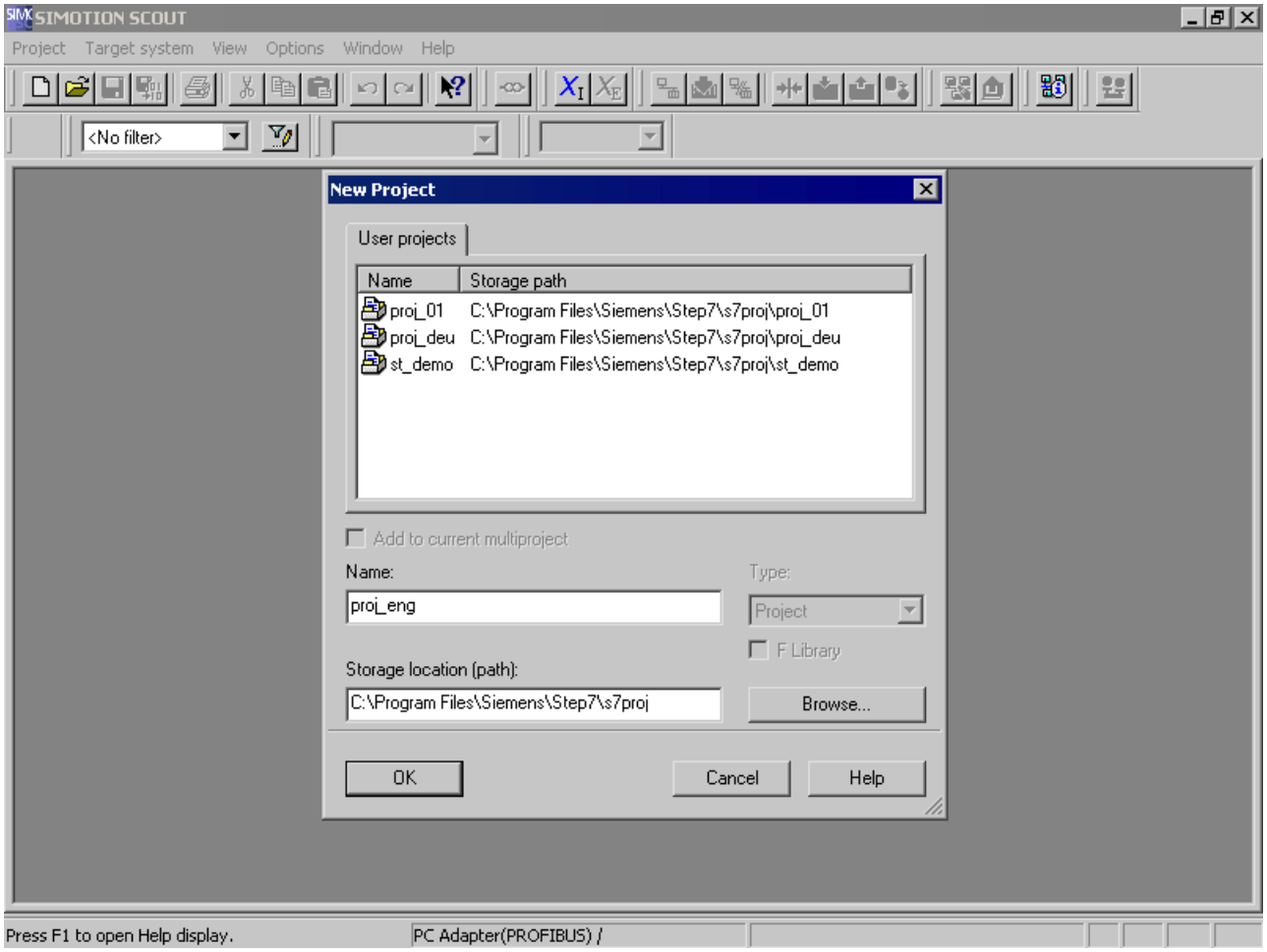



Figure 2-23 Creating a new project

## 2.4.3 Making the hardware known

### The steps are as follows:

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

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

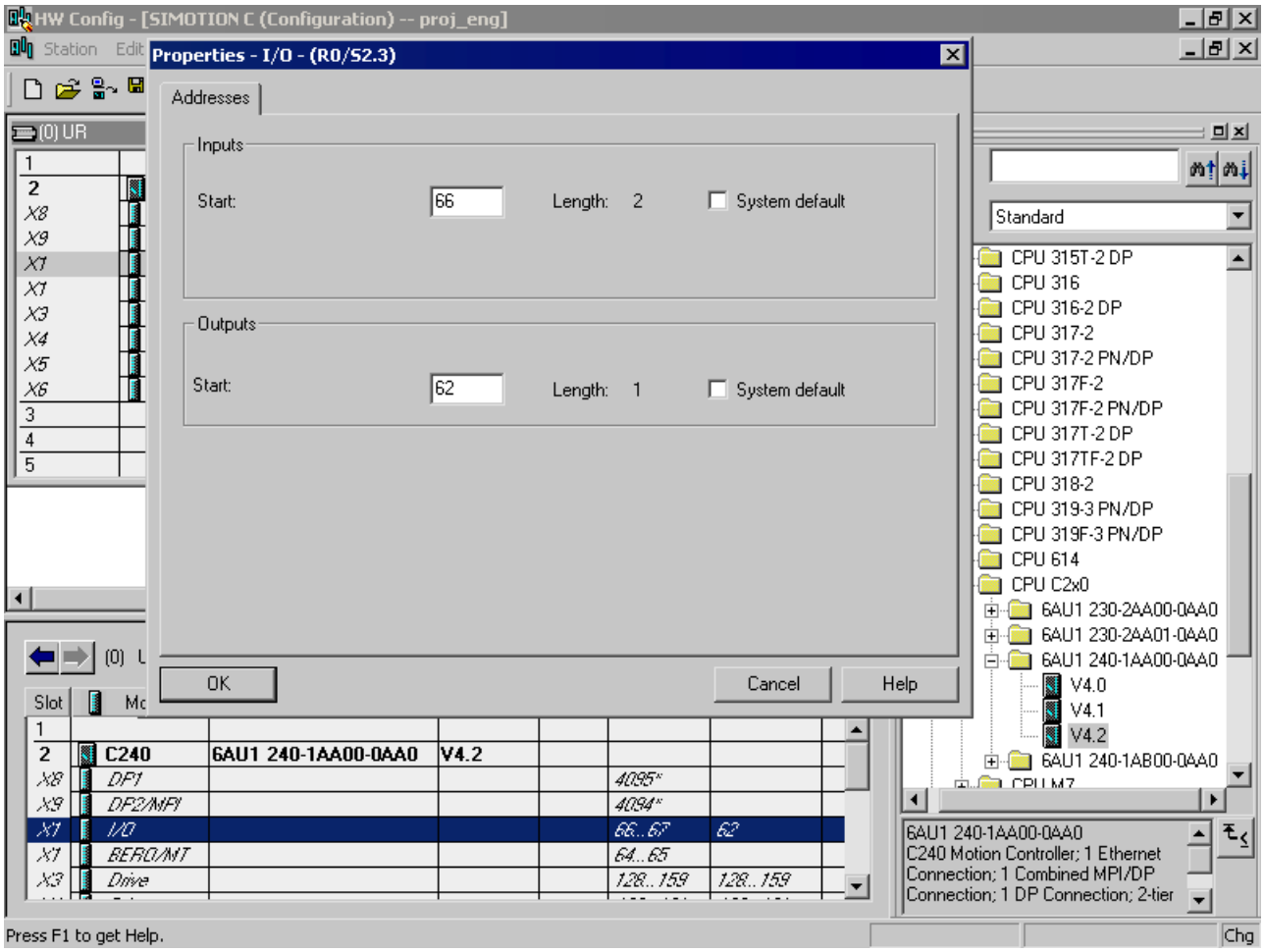



Figure 2-24 Change in HW Config

### 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 81), 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 26); the structure of an ST source file is described in detail in Structure of the ST source file (Page 102) and in Source file sections (Page 193).
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**.

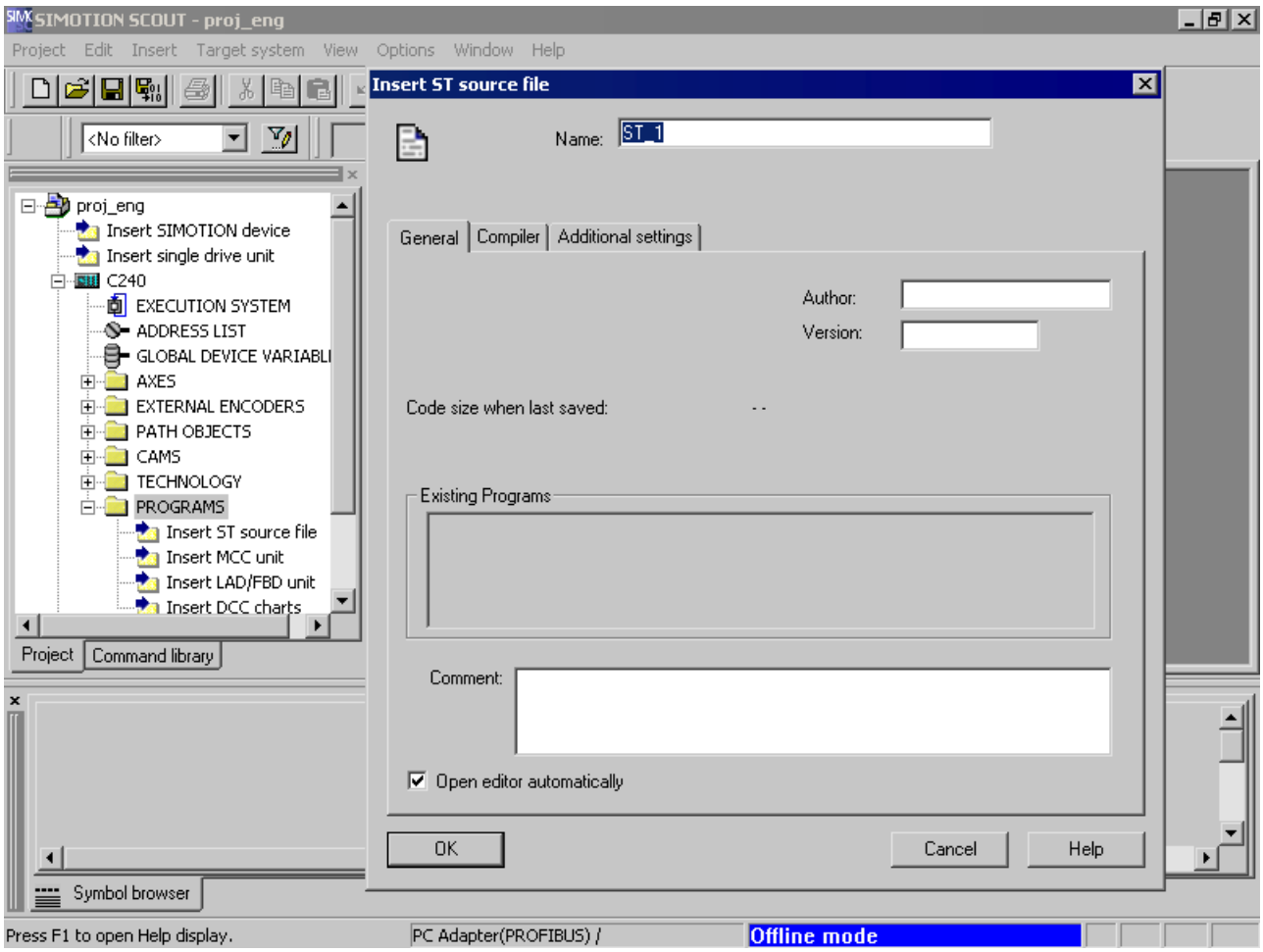



Figure 2-25 Naming the ST source file

### 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 26) and in Making settings for the compiler (Page 56).

#### 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- 13 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 82).
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

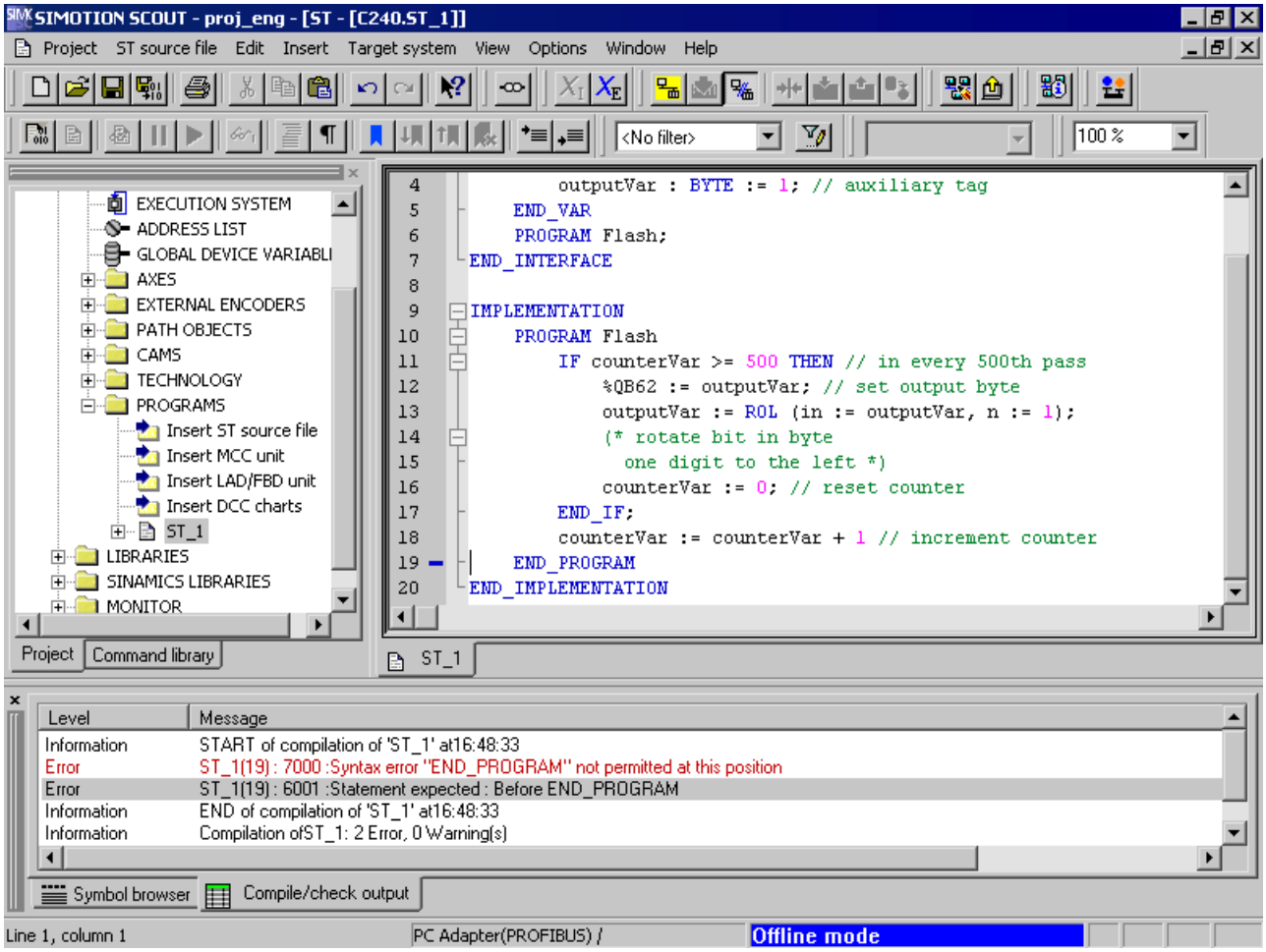



Figure 2-26 Error messages during ST source file compilation

The figure shows an example of compiling the ST source file ST_1 (see Source text of the sample program (Page 81), 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 407).

## 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.

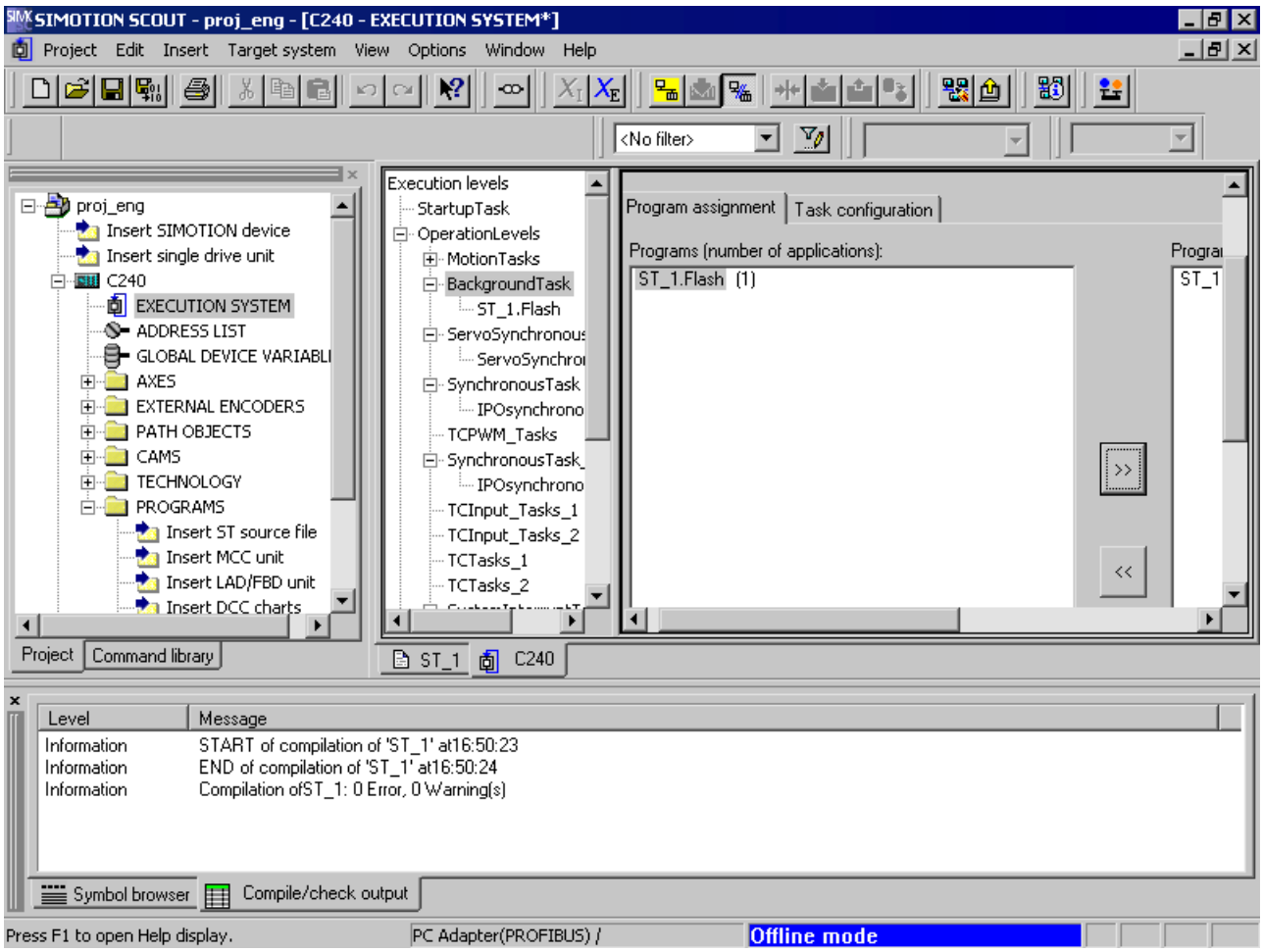



Figure 2-27 Assigning the sample program to the BackgroundTask

### 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 selected target devices** 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 Configuration Manual and SIMOTION SCOUT online help.

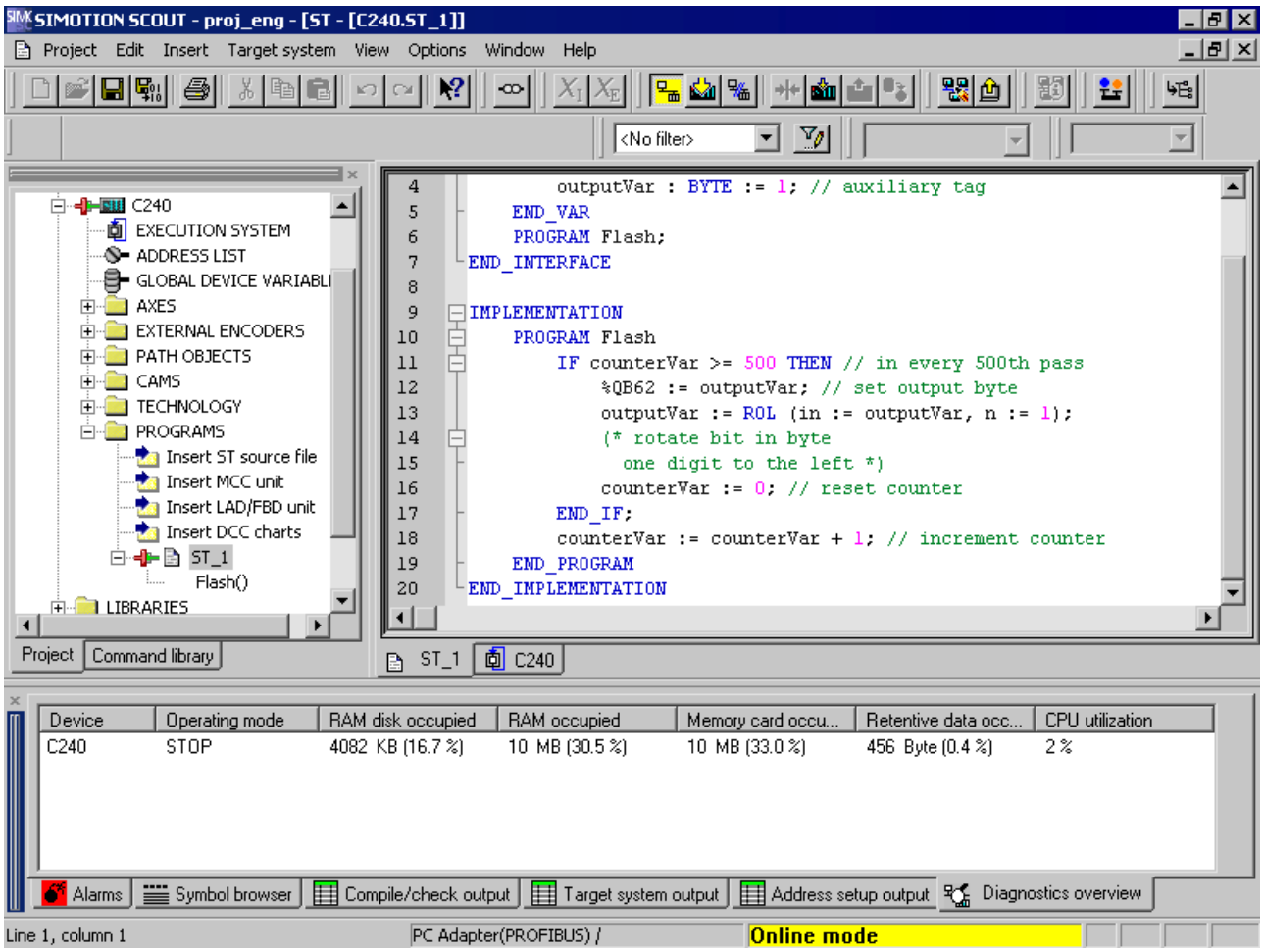



Figure 2-28 Establishing a connection to the target system

### 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 > 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.

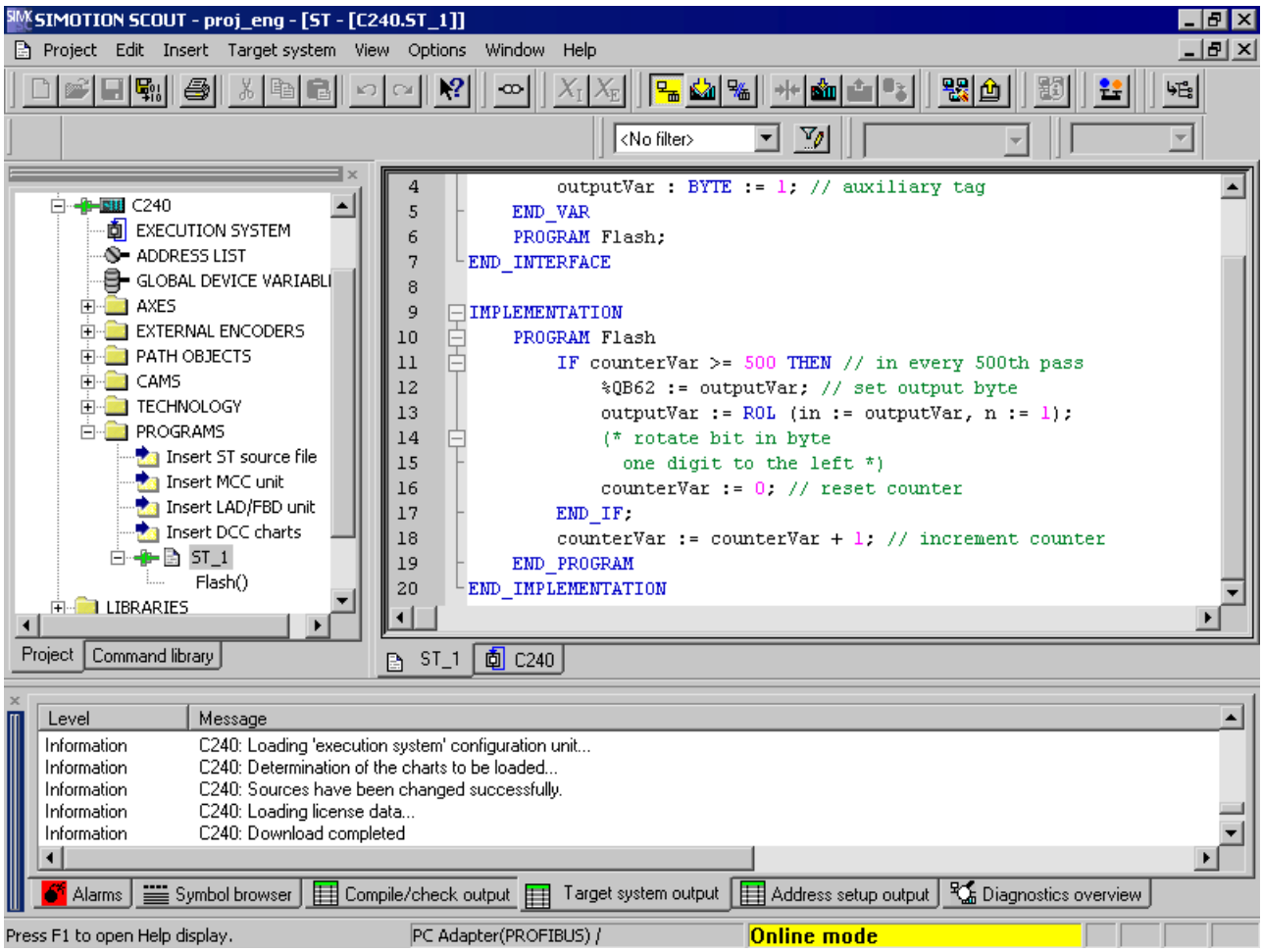



Figure 2-29 Downloading the sample program to the target system

### 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 305).

# 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 365).

## 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.

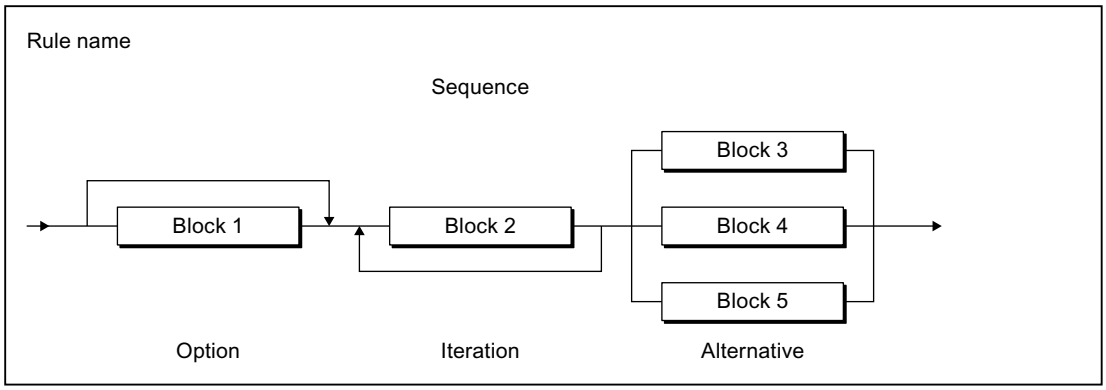



Figure 3-1 Syntax diagram

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:

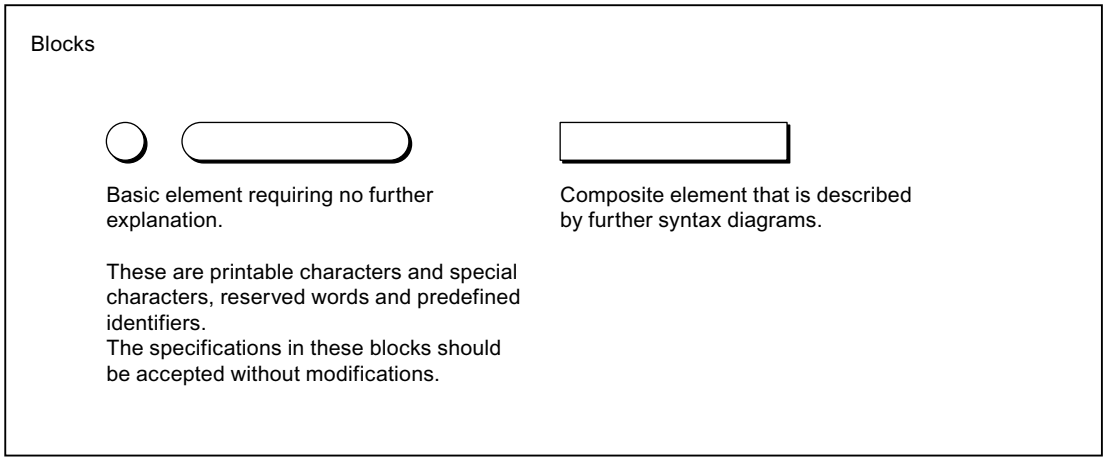



Figure 3-2 Blocks

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 349)).

#### See also

Formal Language Description (Page 349)

### 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 366)). 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 366)).

#### See also

Formal Language Description (Page 349)

# 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 88)). Below is a selection of the main terminals; for a complete overview, refer to Basic elements (terminals) (Page 351).

## 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 89)) 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 349), Basic elements (terminals) (Page 351)).

## 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

#### Valid 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:

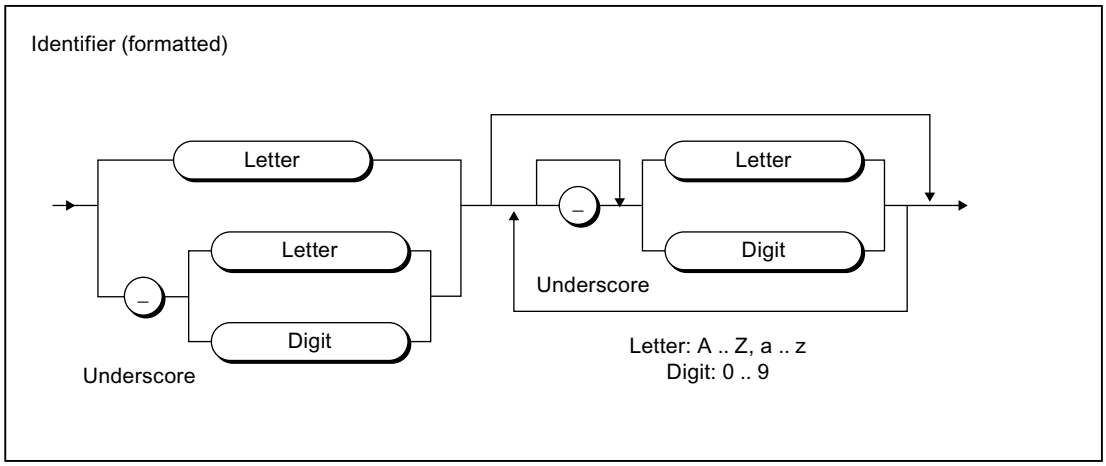



Figure 3-3 Syntax: Identifier

According to this syntax diagram, the first character of an identifier must be a letter or an 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.

### Permissible identifiers, which you can define (user-defined identifiers)

Identifiers which you define yourself, e.g. in variables declarations, data type declarations, and function names, must not be the same as reserved identifiers (Page 91). When assigning a name, it is best to choose a unique, meaningful name that contributes to the clarity of the program.

## 3.2.2.2 Examples of identifiers

### Examples of valid identifiers

The following names are valid identifiers:

```
x              y12            sum            temperature      P_CONTROLLER
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` | ARRAY, REAL, and _sizeOf are valid identifiers from a technical point of view, but they are also reserved identifiers (Page 91), i.e. they may only be used as predefined. This means you cannot use this name for your own purposes, for example, for a variable. |
| `REAL` | |
| `_sizeOf` | |

## 3.2.3 Reserved identifiers

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

There is no distinction between upper and lower case notation.

- Protected identifiers in the ST programming language (Page 91)
- Reserved identifiers in the ST programming language (Page 96)
- Other reserved identifiers (Page 97), such as standard functions, system functions, system variables

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

### 3.2.3.1 Protected identifiers in the ST programming language

The table shows the protected identifiers of the ST programming language; they can only be used as predefined. A brief explanation can be found in the Appendix, under Reserved words (Page 357). The associated syntax diagrams (Page 87) can be found in the Appendix, under Rules (Page 365).

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 | INT_TO_SINT |
| IMPLEMENTATION | INT_TO_TIME |
| INSERT | INT_TO_UDINT |
| INT | INT_TO_UINT |
| INT_TO_BYTE | INT_TO_USINT |
| INT_TO_DINT | INT_TO_WORD |
| INT_TO_DWORD | INT_VALUE_TO_BOOL |
| INT_TO_LREAL | INTERFACE |
| INT_TO_REAL | |
| **L** | |
| LABEL | LREAL_TO_REAL |
| LEFT | LREAL_TO_SINT |
| LEN | LREAL_TO_STRING |
| LIMIT | LREAL_TO_UDINT |
| LITTLEBYTEARRAY_TO_ANYTYPE | LREAL_TO_UINT |
| LN | LREAL_TO_USINT |
| LOG | LREAL_VALUE_TO_BOOL |
| LREAL | LREAL_VALUE_TO_BYTE |
| LREAL_TO_DINT | LREAL_VALUE_TO_DWORD |
| LREAL_TO_INT | LREAL_VALUE_TO_WORD |
| **M** | |
| MAX | MOD |
| MID | MUX |
| MIN | |

| | |
|---|---|
| **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 | UINT_TO_UDINT |
| UDINT_TO_BYTE | UINT_TO_USINT |
| UDINT_TO_DINT | UINT_TO_WORD |
| UDINT_TO_DWORD | UINT_VALUE_TO_BOOL |
| UDINT_TO_INT | UNIT |
| UDINT_TO_LREAL | UNTIL |
| UDINT_TO_REAL | USELIB |
| UDINT_TO_SINT | USEPACKAGE |
| UDINT_TO_STRING | USES |
| UDINT_TO_UINT | USINT |
| UDINT_TO_USINT | USINT_TO_BYTE |
| UDINT_TO_WORD | USINT_TO_DINT |
| UDINT_VALUE_TO_BOOL | USINT_TO_DWORD |
| UINT | USINT_TO_INT |
| UINT_TO_BYTE | USINT_TO_LREAL |
| UINT_TO_DINT | USINT_TO_REAL |
| UINT_TO_DWORD | USINT_TO_SINT |
| UINT_TO_INT | USINT_TO_UDINT |
| UINT_TO_LREAL | USINT_TO_UINT |
| UINT_TO_REAL | USINT_TO_WORD |
| UINT_TO_SINT | USINT_VALUE_TO_BOOL |
| **V** | |
| VAR | VAR_OUTPUT |
| VAR_GLOBAL | VAR_TEMP |
| VAR_IN_OUT | VOID |
| VAR_INPUT | |
| **W** | |
| WAITFORCONDITION | WORD_TO_INT |
| WHILE | WORD_TO_SINT |
| WITH | WORD_TO_UDINT |
| WORD | WORD_TO_UINT |
| WORD_TO_BOOL | WORD_TO_USINT |
| WORD_TO_BYTE | WORD_VALUE_TO_LREAL |
| WORD_TO_DINT | WORD_VALUE_TO_REAL |
| WORD_TO_DWORD | |
| **X** | |
| XOR | |

### 3.2.3.2 Reserved identifiers in the ST programming language

The table contains additional reserved identifiers that are reserved for future expansions to the ST programming language.

Table 3- 2 Reserved identifiers in the ST programming language

| | |
|---|---|
| **A** | |
| ACTION<br>ADD<br>ADD_DT_TIME | ADD_TIME<br>ADD_TOD_TIME |
| **B** | |
| BCD_TO_BYTE<br>BCD_TO_DINT<br>BCD_TO_DWORD<br>BCD_TO_INT | BCD_TO_LWORD<br>BCD_TO_SINT<br>BCD_TO_WORD<br>BYTE_TO_BCD |
| **C** | |
| CONFIGURATION<br>CTD_LINT<br>CTD_ULINT<br>CTU_LINT | CTU_ULINT<br>CTUD_LINT<br>CTUD_ULINT |
| **D** | |
| DINT_TO_BCD<br>DIV | DIVTIME<br>DWORD_TO_BCD |
| **E** | |
| EN<br>END_ACTION<br>END_CONFIGURATION<br>END_RESOURCE | END_STEP<br>END_TRANSITION<br>ENO<br>EQ |
| **F** | |
| F_EDGE | FROM |
| **G** | |
| GE | GT |
| **I** | |
| INITIAL_STEP | INT_TO_BCD |
| **L** | |
| LE<br>LINT<br>LT | LWORD<br>LWORD_TO_BCD |
| **M** | |
| MUL | MULTIME |
| **N** | |
| NE | |
| **R** | |
| R_EDGE | RESOURCE |

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

### 3.2.3.3 Additional reserved identifiers

User definitions must avoid not only protected identifiers (Page 91) and reserved identifiers (Page 96) of the ST programming language, but also identifiers which have a defined meaning in SIMOTION or which are intended for future expansions.

#### Identifiers with a defined meaning in SIMOTION

Do not declare any identifiers (in variables declarations or data type declarations, for example) which are already defined in SIMOTION. If you do, your declarations can hide the identifiers defined in SIMOTION, which may make it impossible to access them. Under certain circumstances, the compiler may not be able to issue a corresponding warning if, for example, the associated technology package is not imported.

The following are defined as identifiers in SIMOTION:

- General standard functions including the associated data types
  (see also the SIMOTION Basic Functions Function Manual)
- General standard function blocks including the associated data types
  (see also the SIMOTION Basic Functions Function Manual)
- Functions for task control, runtime measurement, and message programming including the associated data types
  (see also the SIMOTION Basic Functions Function Manual)
- System functions and system variables of SIMOTION devices including the associated data types
  (see also the list manuals of the SIMOTION devices)
- System functions, system variables, and configuration data of technology objects including the associated data types
  (see also the list manuals of the technology packages)
- Protected and reserved identifiers of other programming languages

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

### Identifiers which are intended for future SIMOTION expansions

Avoid user-defined identifiers which are intended for future SIMOTION expansions. This is a precaution to ensure that your user-defined identifiers will not hide any identifiers defined in future versions of SIMOTION.

In particular, all identifiers starting with the following character strings are intended for future SIMOTION expansions:

- _ (underscore)
- **Command**
- **Enum**
- **Struct**

If you declare an identifier that starts with one of these character strings, a corresponding warning (e.g. warning number 16026) is issued.

## 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'<br>'$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$L' | String of length 6 containing the word Text followed by the characters CR and LF. |
| 'ÄÖÜ'<br>'$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 modules (Page 193).

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:

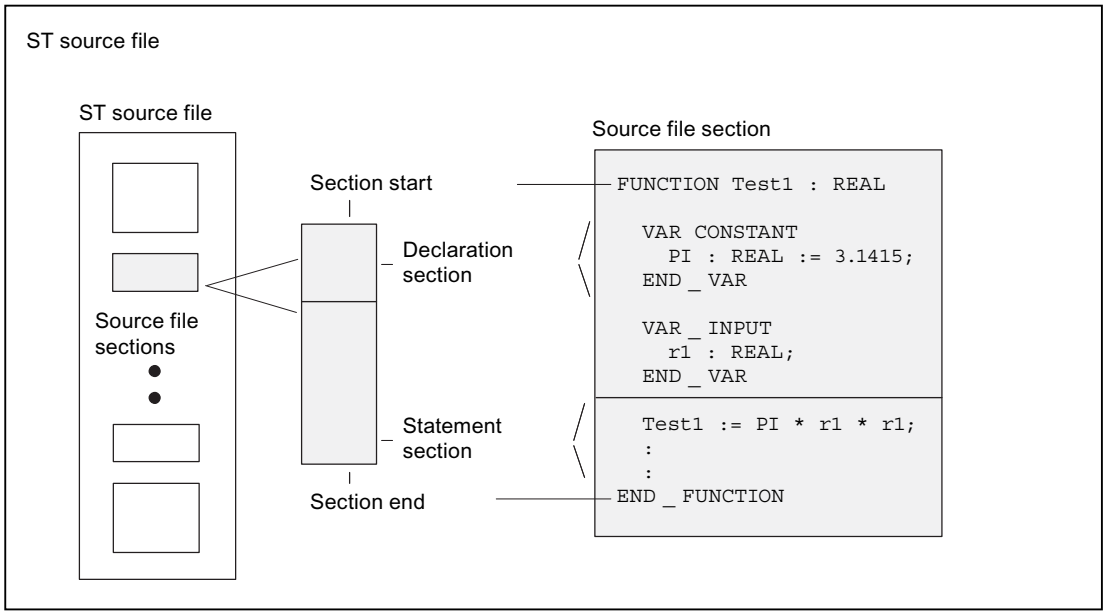



Figure 3-4 Structure of an ST source file

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 Variables declaration (Page 124)

- Control statements

  Repetition or branching of statements; see Control statements (Page 152)

- Subroutine execution

  Functions (FC) and function blocks (FB); see Functions, function blocks, programs (Page 169)

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 Help for the language description (Page 349).

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 106)

Description of the technology object data types (Page 120)

System data types (Page 123)

## 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**<br>Data of this type uses 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**<br>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,<br>0.0,<br>+1.175_494_351E-38 to +3.402_823_466E+38<br>Accuracy:<br>23-bit mantissa (corresponds to 6 decimal places), 8-bit exponent, 1-bit sign. |
| | Long floating-point number<br>(in accordance with IEEE-754) | LREAL | 64 | -1.797_693_134_862_315_8E+308 to<br>-2.225_073_858_507_201_4E-308,<br>0.0,<br>+2.225_073_858_507_201_4E-308 to<br>+1.797_693_134_862_315_8E+308<br>Accuracy:<br>52-bit mantissa (corresponds to 15 decimal places), 11-bit exponent, 1-bit sign. |

| Type | Reserv. word | Bit width | Range of values |
|---|---|---|---|
| **Time types**<br>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<br>Maximum of two digits for the values day, hour, minute, second and a maximum of three digits for milliseconds<br>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<br>Leap years are taken into account, year has four digits, month and day are two digits each<br>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<br>Maximum of two digits for the values hour, minute, second and maximum of three digits for milliseconds<br>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<br>DATE_AND_TIME consists of the data types DATE and TIME<br>Initialization with DT#0001-01-01-0:0:0.0 |
| **String type**<br>Data of this type represents character strings, in which each character is encoded with the specified number of bytes.<br>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.<br>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.<br>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.

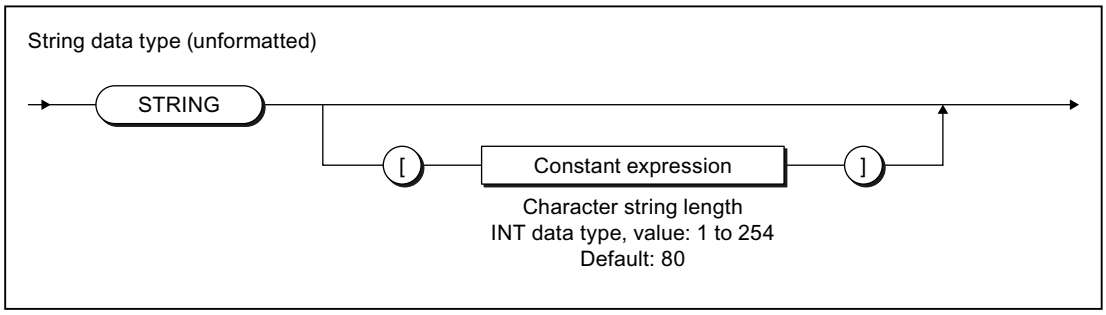



Figure 3-5 Syntax: STRING data type

### 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<br>TIME#MIN | TIME | T#0ms | 16#0000_0000[1] |
| T#MAX<br>TIME#MAX | TIME | T#49d_17h_2m_47s_295ms | 16#FFFF_FFFF[1] |
| TOD#MIN<br>TIME_OF_DAY#MIN | TOD | TOD#00:00:00.000 | 16#0000_0000[1] |
| TOD#MAX<br>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 in a similar way 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.<br>Please refer to the *SIMOTION Basic Functions* Function Manual.<br>Initialization with STRUCTALARMID#NIL |
| StructTaskId | 32 | Data type of the taskId for the project-wide unique identification of the tasks in the execution system.<br>Please refer to the *SIMOTION Basic Functions* Function Manual.<br>Initialization with STRUCTTASKID#NIL |

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

| Symbolic constant | Data type | Meaning |
|---|---|---|
| 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 subsections of subsequent source file modules (see Breakdown of ST source file (Page 102) and Source file modules (Page 193)):

- 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 111)

Derivation of elementary or derived data types (Page 113)

Derived data type ARRAY (Page 114)

Derived data type - Enumerator (Page 116)

Derived data type STRUCT (structure) (Page 118)

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

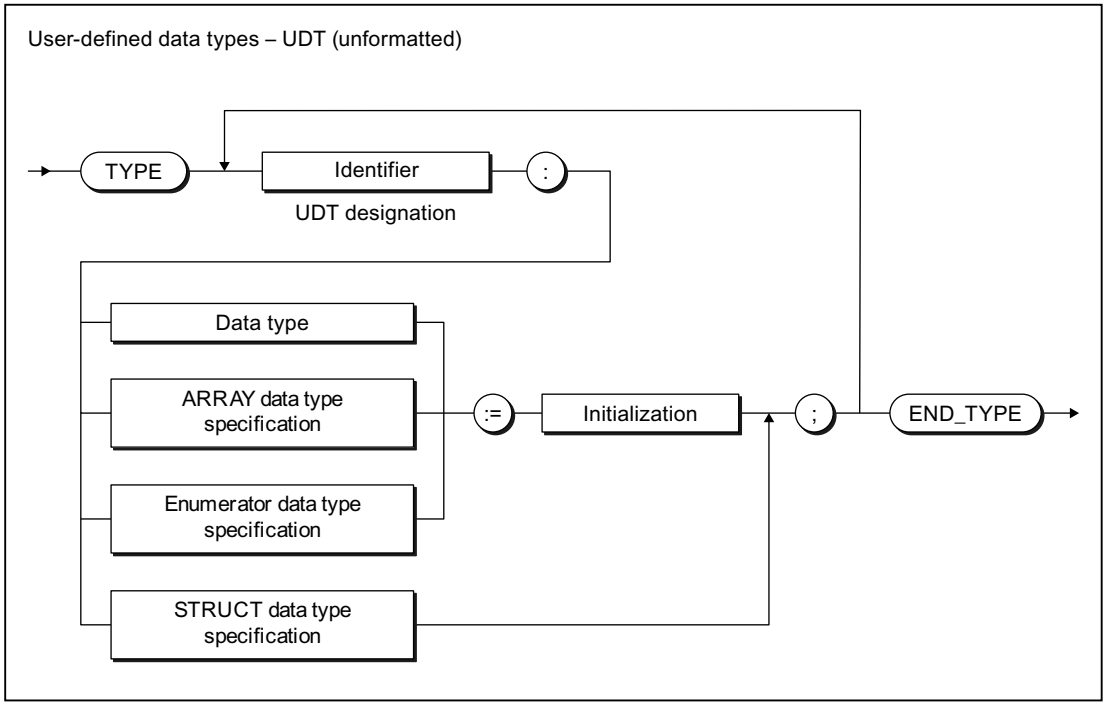



Figure 3-6 Syntax: User-defined data type

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 113)):

   – 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 114))
   – Enumerator data type specification (see Derived data type enumerator (Page 116))
   – STRUCT data type specification (see Derived data type STRUCT – structure (Page 118))

   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 127).

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 126), and how to assign values with UDT in Syntax for value assignment (Page 133).

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 elements 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**

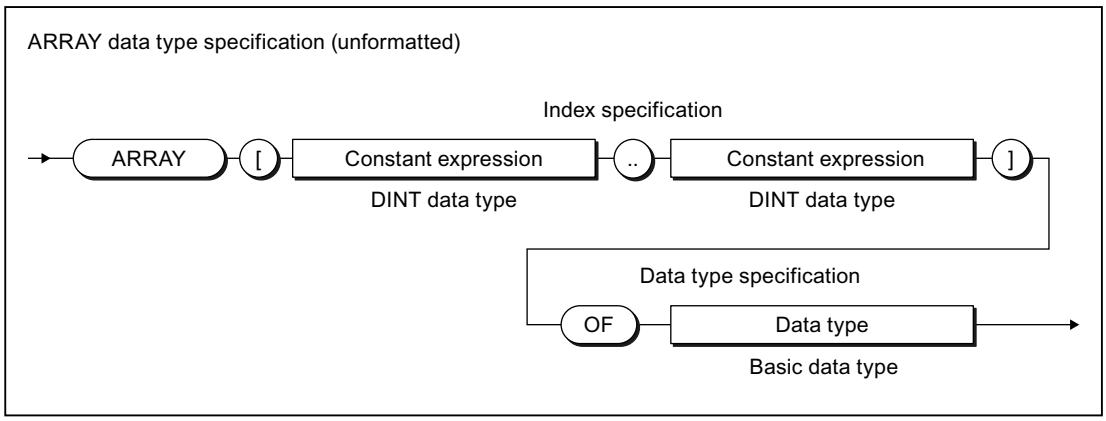



Figure 3-7 Syntax: ARRAY data type specification

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 Converting elementary data types (Page 164)).
- 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 Converting elementary data types (Page 164)).

**Note**

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

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

You can define the size of the memory space occupied by the ARRAY with _sizeOf (in := *array-name*). The following applies: _sizeOf (in := *array_of_type*) := _lengthIndexOf (in := *array_of_type*) * _sizeOf (in := *type*).

The syntax of these functions is described in the "SIMOTION Basic Functions" Function Manual.

**Note**

As of SIMOTION Kernel version V4.2, you can declare arrays with a dynamic length as in-out parameters in functions and function blocks. For more information, refer to the corresponding section (Page 175) for the FB and FC declaration subsection (Page 171).

## One- and multidimensional arrays

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

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

### Redeclaration

The data type definitions of arrays are recognized as identical (even in different sources) if the following conditions are met:

1. The identifier of the data type is identical.
2. Both array limits are identical.
3. The data type of the array elements is identical.
4. The optional array initialization list is identical (including repeat factors and constant expressions).

### Initialization

Per default, the array elements are assigned the initialization value of the basic data type. Optionally the initialization value of the array element can be changed by assigning an array initialization list enclosed in square brackets [ ], see"Initialization of variables or data types" (Page 127).

## 3.4.2.5 Derived data type - Enumerator

In the case of enumeration 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 : Enumeration data type specification { := initialization } ; END_TYPE**

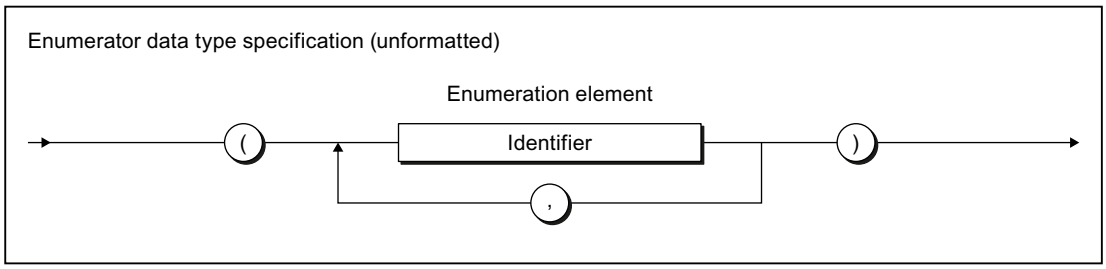



Figure 3-8 Syntax: Enumeration data type specification

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

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

## Example

Table 3- 15 Examples of enumeration 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 enumeration data types as system data types.

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

## Redeclaration

The data type definitions of enumeration data types are recognized as identical (even in different sources) if the following conditions are met:

1. The identifier of the data type is identical.
2. All enumeration elements are identical.
3. The sequence of the enumeration elements is identical, so their numerical values are identical.
4. The optional initialization value is identical.

### Initialization

Per default, an enumeration data type is assigned the 1st value of the enumeration as initialization value. Optionally the initialization value can be changed by assigning another enumeration element, see"Initialization of variables or data types" (Page 127).

## 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**

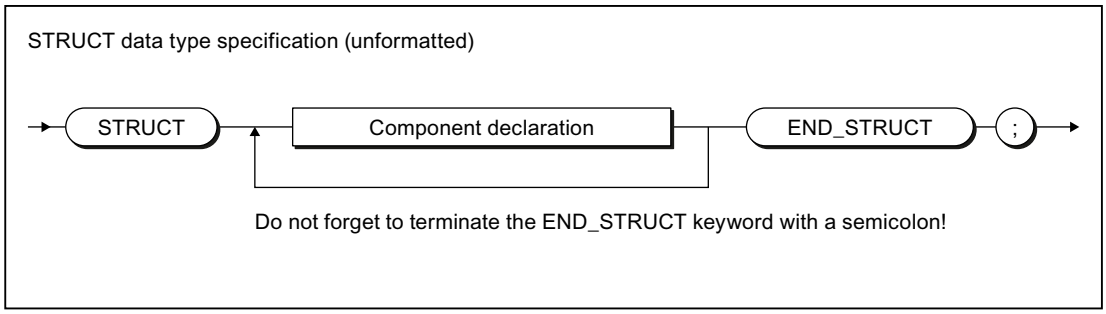



Figure 3-9 Syntax: STRUCT data type specification

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

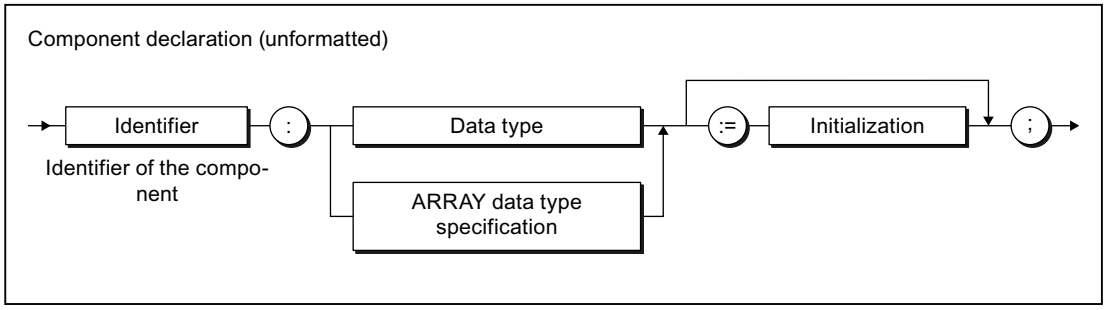



Figure 3-10 Syntax: Component declaration

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 see "Initialization" in this section.

**Note**

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

- STRUCT data type specifications
- Enumeration data type specifications

**Remedy**: 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.

## Example

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;
```

### Redeclaration

The data type definitions of structures are recognized as identical (even in different sources) if the following conditions are met:

1. The identifier of the data type is identical.
2. All components of the structure are identical in terms of their:
   - Sequence
   - Identifier
   - Data types or ARRAY data type specifications
   - Initialization values (including any array or structure initialization lists)

### Initialization

Per default, each component is assigned the initialization value of its data type. Optionally, the initialization value of the component can be changed by assigning an appropriate initialization.

When using the structure in another declaration (e.g. variable or data type declaration), the initialization values of individual components can be changed by assigning a structure initialization list, enclosed in round brackets ( ).

See "Initialization of variables or data types" (Page 127).

## 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, PATH[1], CAM_EXT |
| External encoder | ExternalEncoderType | CAM, PATH[1], CAM_EXT |
| Measuring input | MeasuringInputType | CAM, PATH[1], CAM_EXT |
| Output cam | OutputCamType | CAM, PATH[1], CAM_EXT |
| Cam track | _CamTrackType | CAM, PATH[1], CAM_EXT |
| Position axis | PosAxis | CAM, PATH[1], CAM_EXT |
| Following axis | FollowingAxis | CAM, PATH[1], CAM_EXT |
| Following object | FollowingObjectType | CAM, PATH[1], CAM_EXT |

| Technology object | Data type | Contained in the technology package |
|---|---|---|
| Cam | CamType | CAM, PATH[1], CAM_EXT |
| Path axis[1] | _PathAxis | PATH[1], CAM_EXT |
| Path object[1] | _PathObjectType | PATH[1], CAM_EXT |
| Fixed gear | _FixedGearType | CAM_EXT |
| Addition object | _AdditionObjectType | CAM_EXT |
| Formula object | _FormulaObjectType | CAM_EXT |
| Sensor | _SensorType | CAM_EXT |
| Controller object | _ControllerObjectType | CAM_EXT |
| Temperature channel | TemperatureControllerType | TControl |
| General data type, to which every TO can be assigned | ANYOBJECT | |

[1] Available as of version V4.1.

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 121)

Examples of the use of technology object data types (Page 122)

### 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 synchronizedAxis, 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 synchronized 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 synchronized 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 91)) 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 126)).
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).

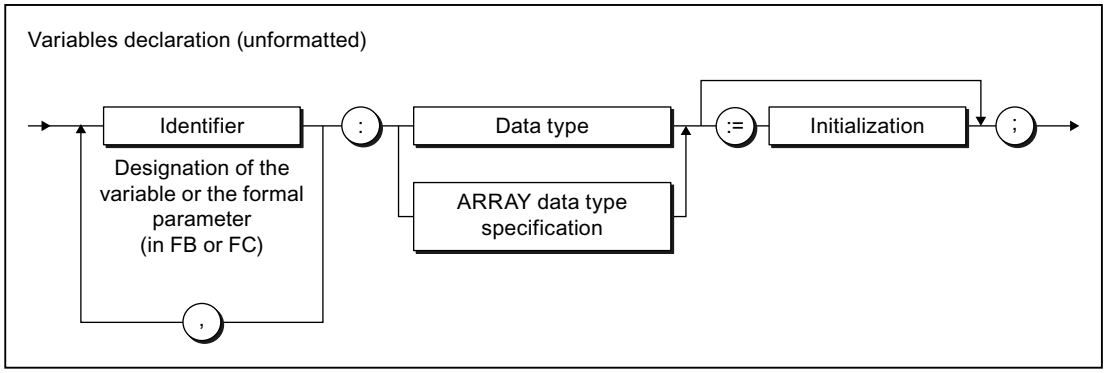



Figure 3-11 Syntax: Variable declaration

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 177)).
- 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 127)).

Deviations from the pattern presented are as follows:

- For constant declarations (a constant **must** be initialized with a value, see Constants (Page 131)),
- For process image access (see Overview of all variable declarations (Page 126)):
  - 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:
    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 source file modules, refer to Breakdown of ST source file (Page 102) and Source file modules (Page 193).

Table 3- 22 Keywords for declaration blocks

| Keyword | Meaning | Declaration in the following declaration sections |
|---|---|---|
| VAR | Declaration of temporary or static variables<br>See Variable model (Page 212). | Any POU |
| VAR_GLOBAL | Declaration of unit variables<br>See Variable model (Page 212). | Interface section<br>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.<br>See Defining functions (Page 170), Defining function blocks (Page 171). | Function<br>Function block<br>Expression |
| VAR_INPUT | Variable declaration of input parameter, value is externally supplied and cannot be changed within the POU.<br>See Defining functions (Page 170), Defining function blocks (Page 171). | Function<br>Function block<br>Expression |
| VAR_OUTPUT | Variables declaration output parameter; value is transmitted from the function block<br>See Defining functions (Page 170), Defining function blocks (Page 171). | Function block |
| VAR_TEMP | Declaration of temporary variables<br>See Variable model (Page 212). | Program<br>Function block |

| Keyword | Meaning | Declaration in the following declaration sections |
|---|---|---|
| RETAIN | Declaration of retentive variables<br>See Variable model (Page 212). | Only as a supplement to VAR_GLOBAL in the interface and implementation section |
| CONSTANT | Declaration of constants<br>See Constants (Page 131). | Only as a supplement:<br>• to VAR in FB, FC, or program<br>• 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 Syntax of variables declaration (Page 124) and Syntax of user-defined data types (Page 111).

- 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.

  The following applies for arrays (ARRAY): The array elements are assigned the initialization value of the basic data type.

- 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: The initialization value for each data type.
  – For enumeration data types: 1st value of the enumeration.
  – For arrays (ARRAY): The array elements are assigned the initialization value of the basic data type.
  – For structures (STRUCT): The structure components are assigned the initialization value of the respective data type.

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

- 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 a structure (STRUCT) in accordance with Figure *Syntax: Structure initialization list* (when using the structure in other declarations).
- Assign an enumeration element to an enumeration data type.

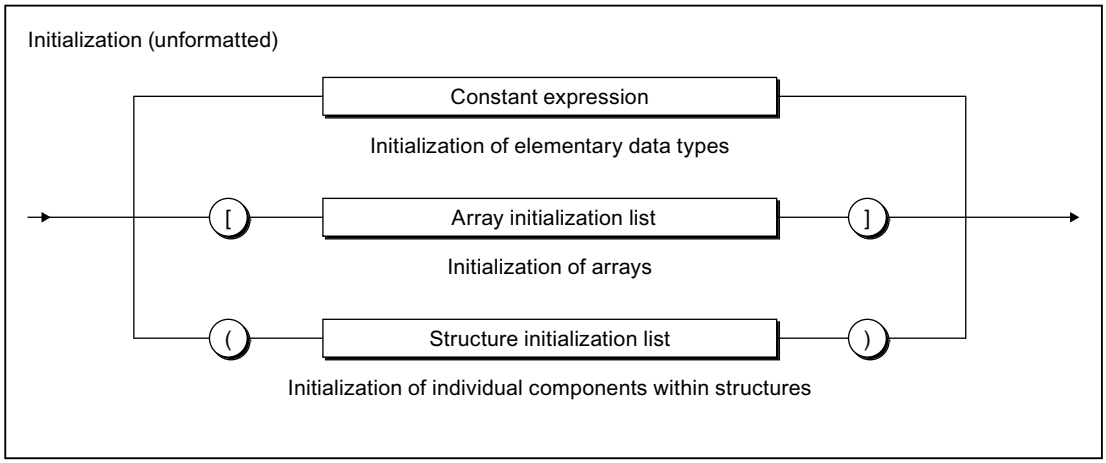



Figure 3-12 Syntax: Variable initialization

The initialization value assigned to a variable is calculated from the constant expression at the time of the compilation. For information about the syntax of the constant expression, see the figure titled *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.).

**Note**

The constant expressions used for initialization are calculated in the data type of the declared variables or in the declared data type.

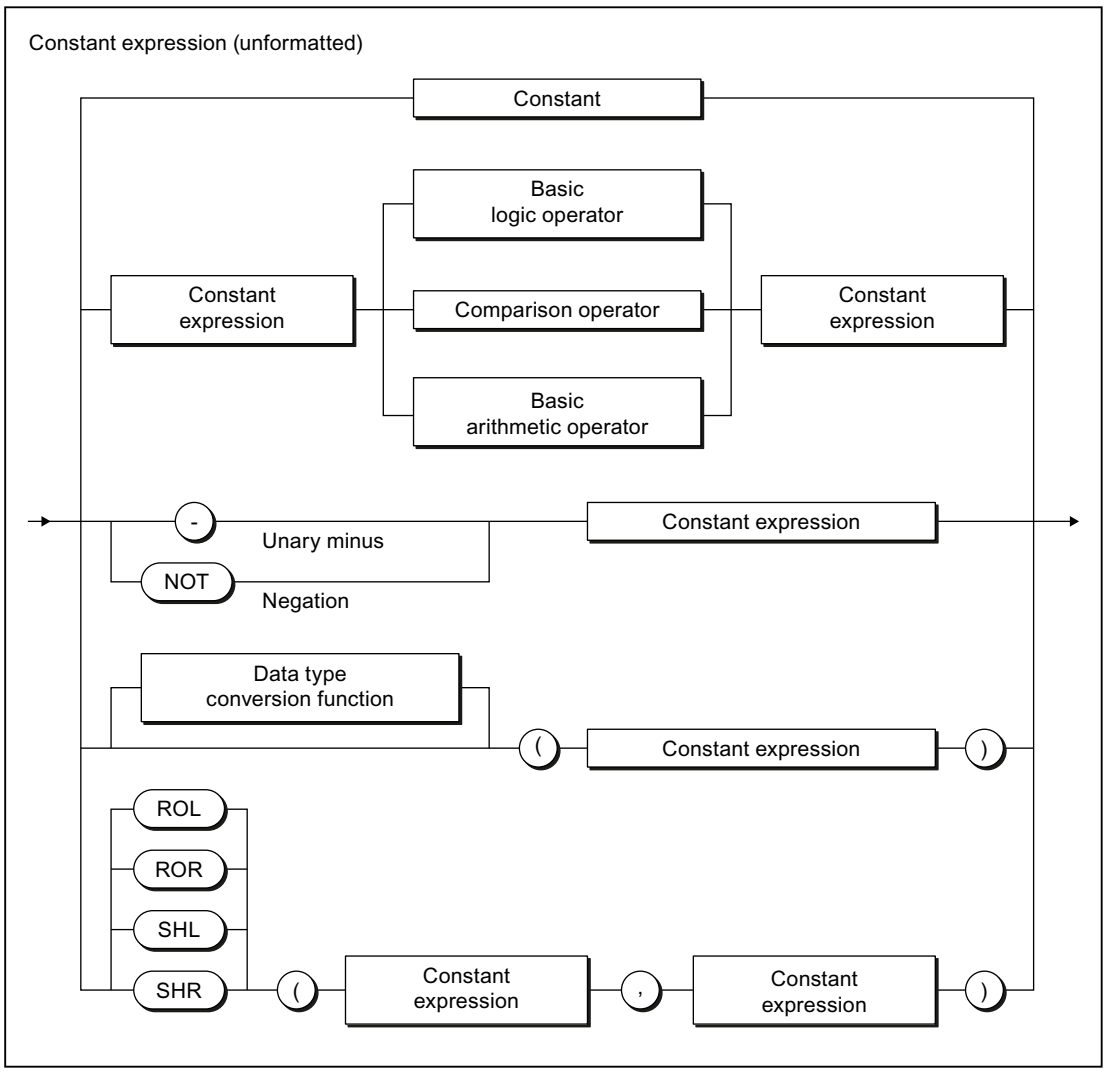



Figure 3-13 Syntax: Constant expression

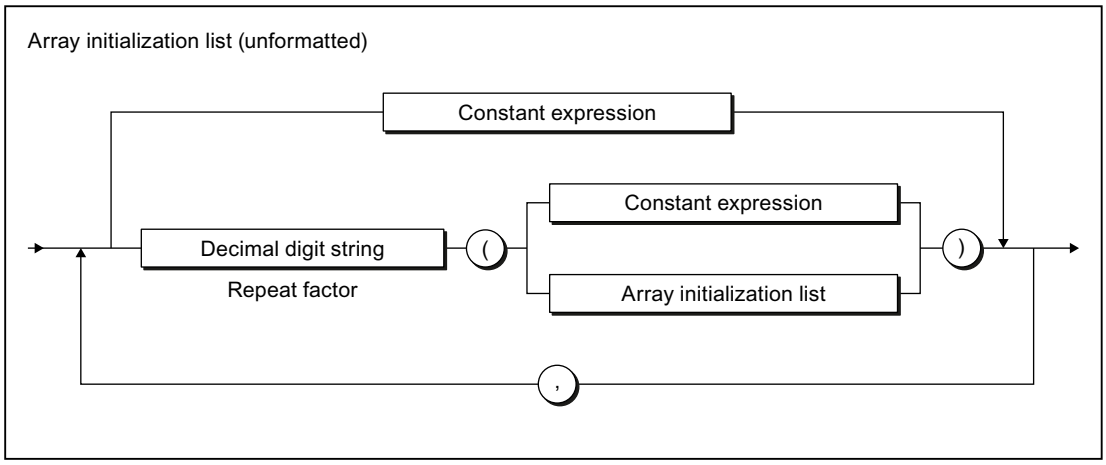



Figure 3-14 Syntax: Array initialization list

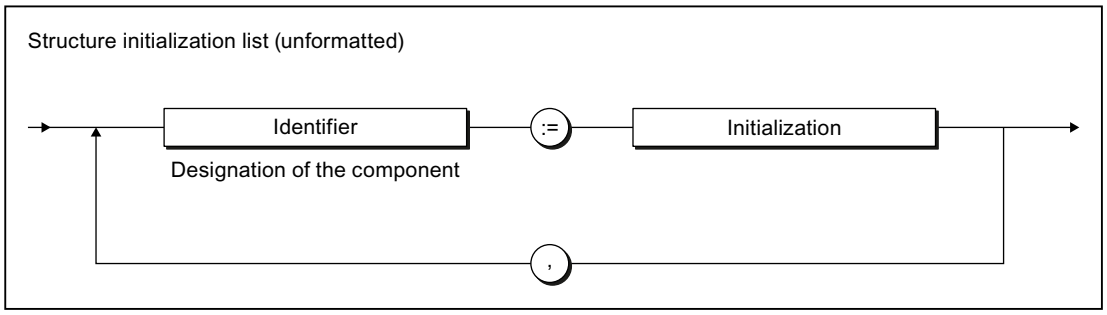



Figure 3-15 Syntax: Structure initialization list

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
    myInt : INT := 9;
    myArray : ARRAY [0..5] OF myInt := [1, 2, 3];
                // Initialization as follows:
                // Array element 0            with 1;
                // Array element 1            with 2;
                // Array element 2            with 3;
                // Array elements 3, 4, 5     with 9
END_TYPE
```

Variables of a technology object data type cannot be assigned an initialization value. They are always initialized by the compiler with TO#NIL.

Information about the circumstances when variables are initialized can be found in Section "Time of the variable initialization" (Page 228).

## 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 212)).

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

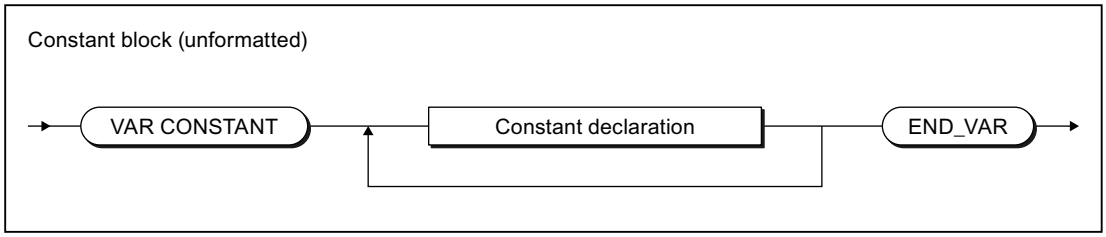



Figure 3-16 Syntax: Constant block in a POU

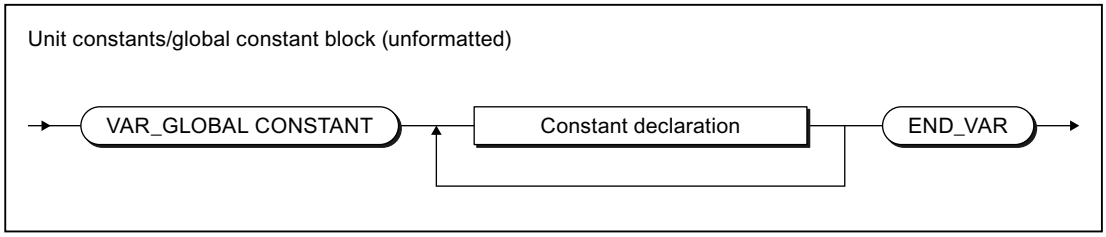



Figure 3-17 Syntax: Unit constants in interface or implementation section

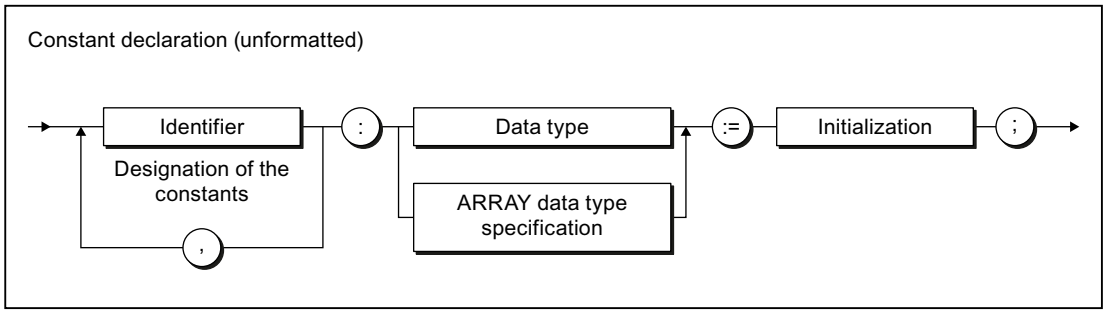



Figure 3-18 Syntax: Constant declaration

To initialize a constant, see "Initialization of variables or data types (Page 127)". The value assigned to a constant is calculated from the constant expression at the time of compilation.

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 no doubt already created value assignments with the character string :=. This may have been for a statement as part of an example (see table titled *Examples of statements* in Statements (Page 103)) or when initializing variables in the declaration subsection of a source file module.

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.

### See also

Notes on avoiding errors and on efficient programming (Page 305)

## 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 124)).

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.

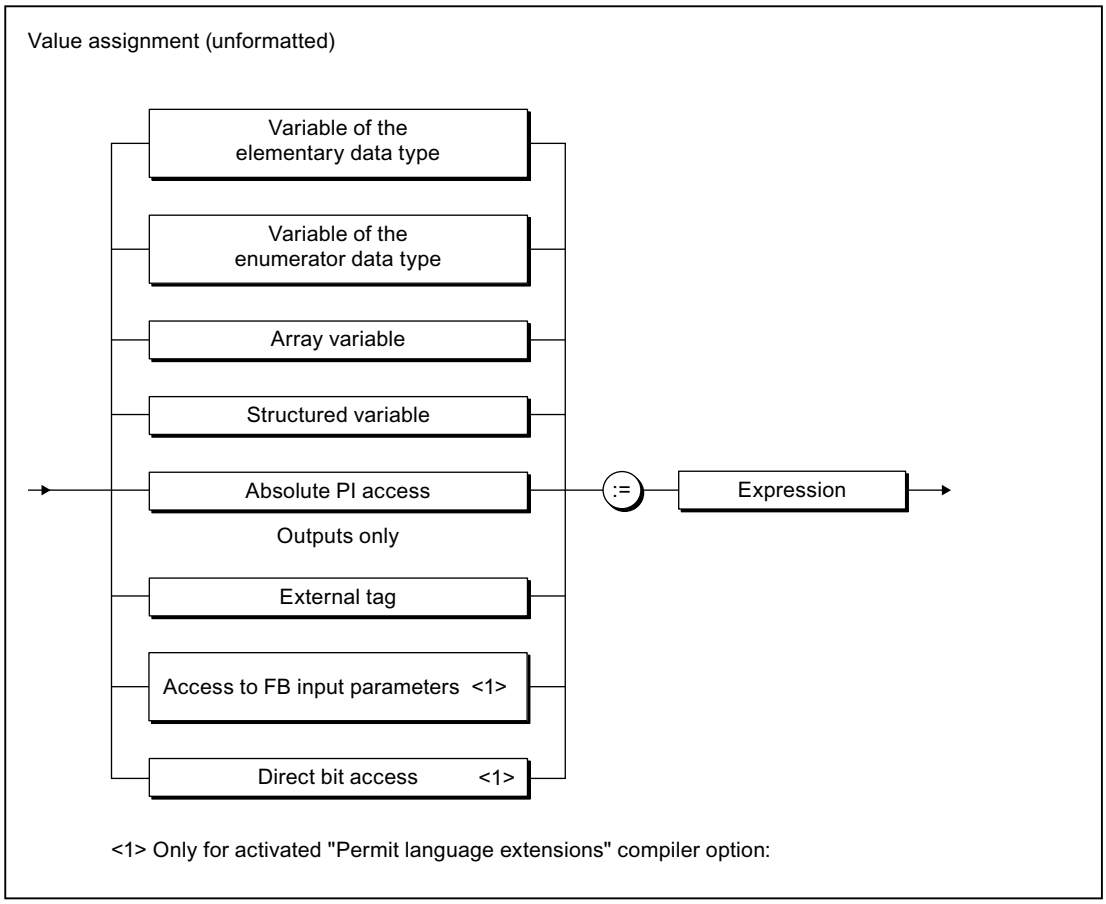



Figure 3-19 Syntax: Value assignments

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

- Value assignments with variables of an elementary data type (Page 135) ,
- Value assignments with variables of the derived enumerator data type (Page 139)
- Value assignments with variables of the derived ARRAY data type (Page 139)
- Value assignments with variables of the derived STRUCT data type (Page 140)
- 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 264).

How the right side of a value assignment, i.e. an expression, is formed, is described in Expressions (Page 141).

### 3.6.1.2 Value assignments with variables of an elementary data type

An expression with an elementary data type (Page 106) 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 135).

- 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 164) 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 137)

### 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 178).

Please also refer to Syntax diagram of STRING data type (Page 106):

**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 conversion between variables of numeric data types and strings; see Converting elementary data types (Page 164) 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 "Permit language extensions" compiler option (see Global compiler settings (Page 56) and Local compiler settings (Page 59)).

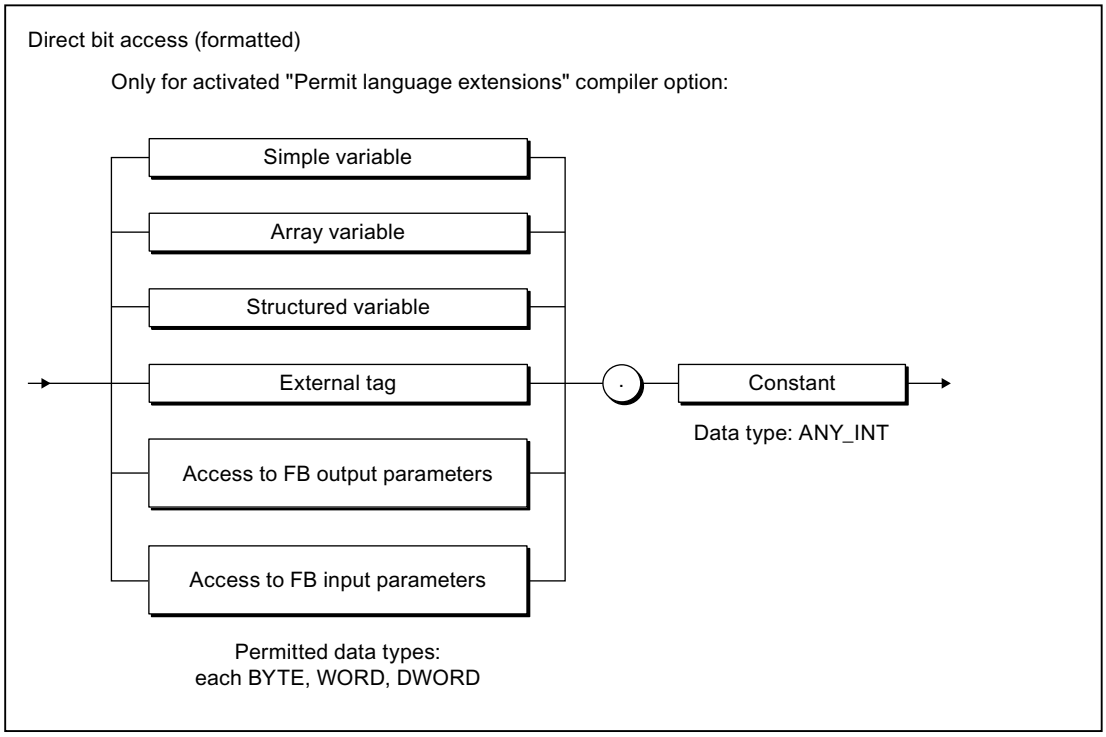



Figure 3-20 Syntax: Direct bit access

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 149).

### 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 enumerator data type (Page 116)) can be assigned to 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 ARRAY data type (Page 114)).

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 STRUCT data type (Page 118)). 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 detailed information about function blocks, refer to the explanations in Defining functions (Page 170) and Defining function blocks (Page 171).

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 (Page 144)
- Relational expressions (Page 147)
- Logic expressions and bit-serial expressions (Page 149)

### 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.

The data type of the result of an arithmetical or bit-serial expression is determined by the data types of the operands. The data type used is the lowest common data type to which both operands can be implicitly converted.

An expression value can only be assigned to a variable (or a parameter of a function or function block) in the following cases:

- The expression calculated 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, see *SIMOTION Basic Functions* Function Manual.

**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 127)):

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

The constant expressions used for initialization are calculated in the data type of the declared variables or in the declared data type.

### 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 Operator priority (Page 151)).
- 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:

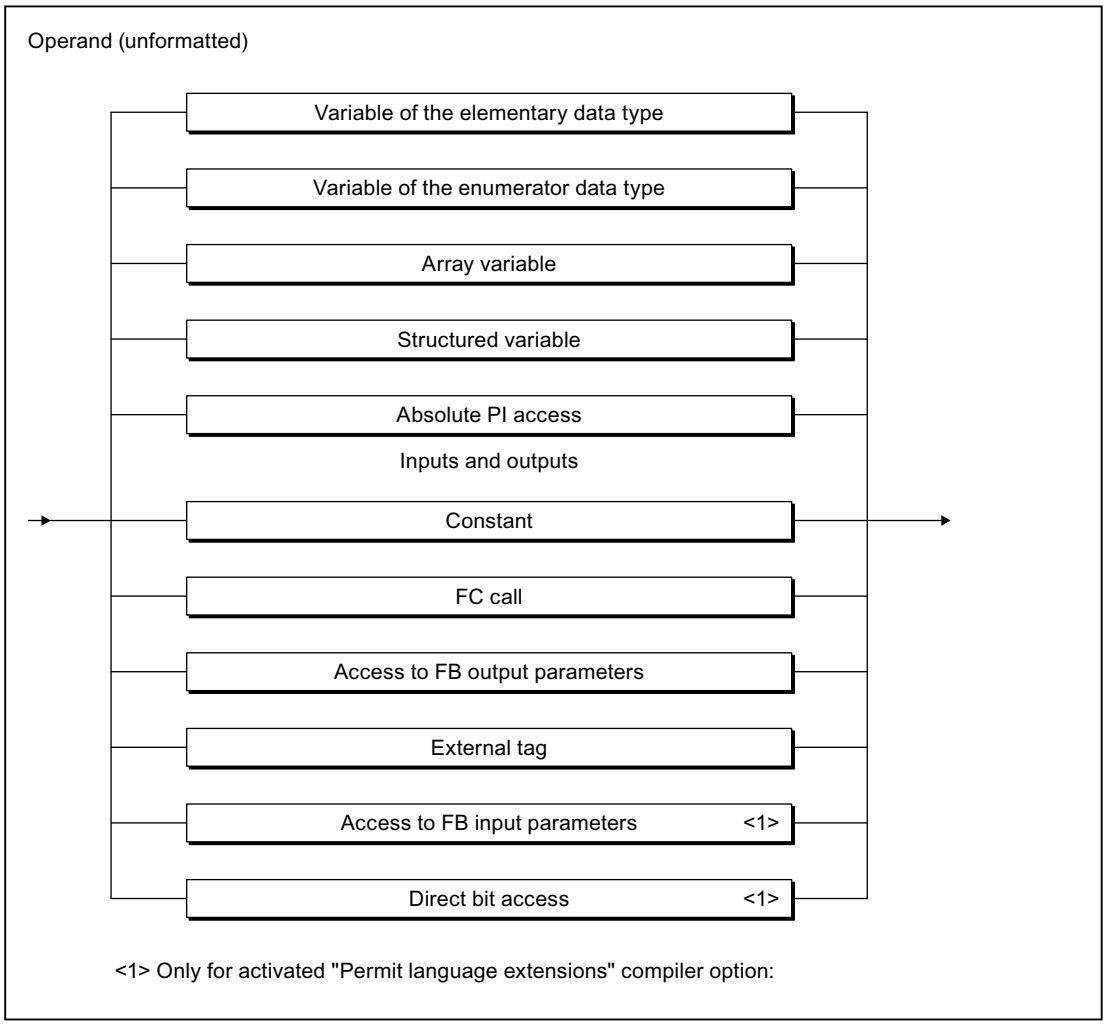



Figure 3-21 Syntax: Operand

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.

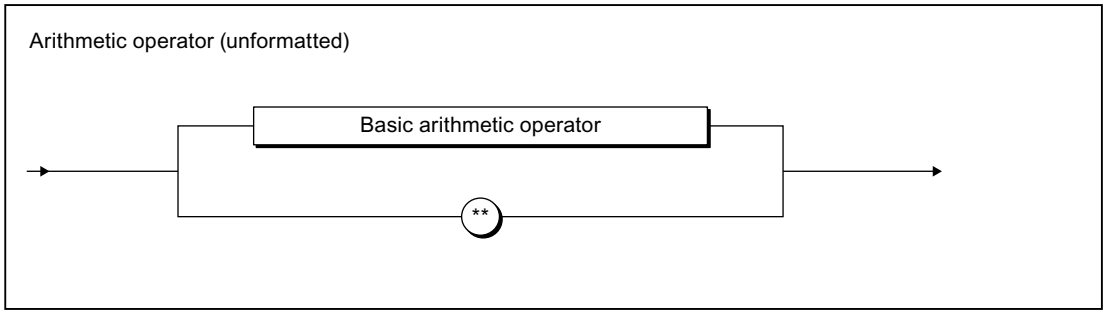



Figure 3-22 Syntax: Arithmetic operator

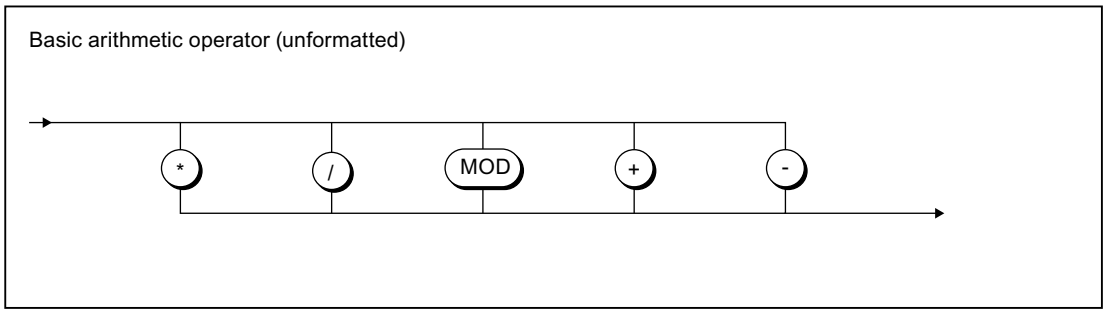



Figure 3-23 Syntax: Basic arithmetic operator

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 109) 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 151).

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 (unless explicitly stated) is the lowest common data type to which both operands can be implicitly converted.

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).

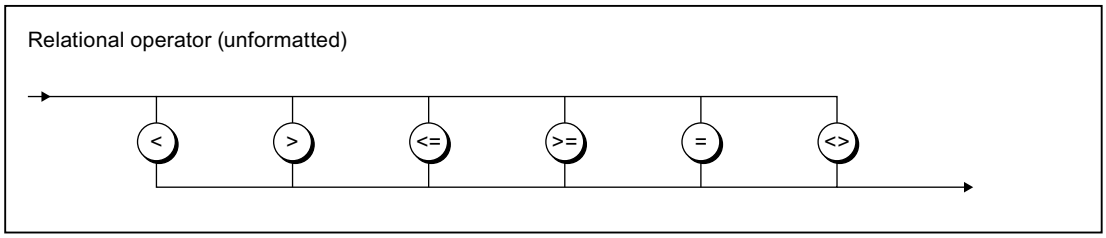



Figure 3-24 Syntax: Relational operators

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 | Field (ARRAY)[3)] | =, <> |
| Structure (STRUCT) | Structure (STRUCT)[3)] | =, <> |
| | | |

1) The comparison is made in the lowest common data type to which both operands can be implicitly converted.

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 the first operand.

Relational expressions and variables or constants of the BOOL data type can be combined with logic operators to form logic expressions (see Logic expressions and bit-serial expressions (Page 149)). 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 Operator priority (Page 151)). 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;

var_3 := 0 < F < 10     // var_3 of BOOL data type, TRUE value
    // The calculation is from left to right
    // in the following form: (0 < F) < 10
    // Initially 0 < F is calculated.
    // The result is from the BOOL data type (FALSE or TRUE)
    // and is compared with BYTE#10.
    // Equivalent expression:
var_3 : = BOOL_TO_BYTE (0 < F) < BYTE#10
```

## 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<br>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 141). The table shows you the priority of the individual operators within an expression.

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

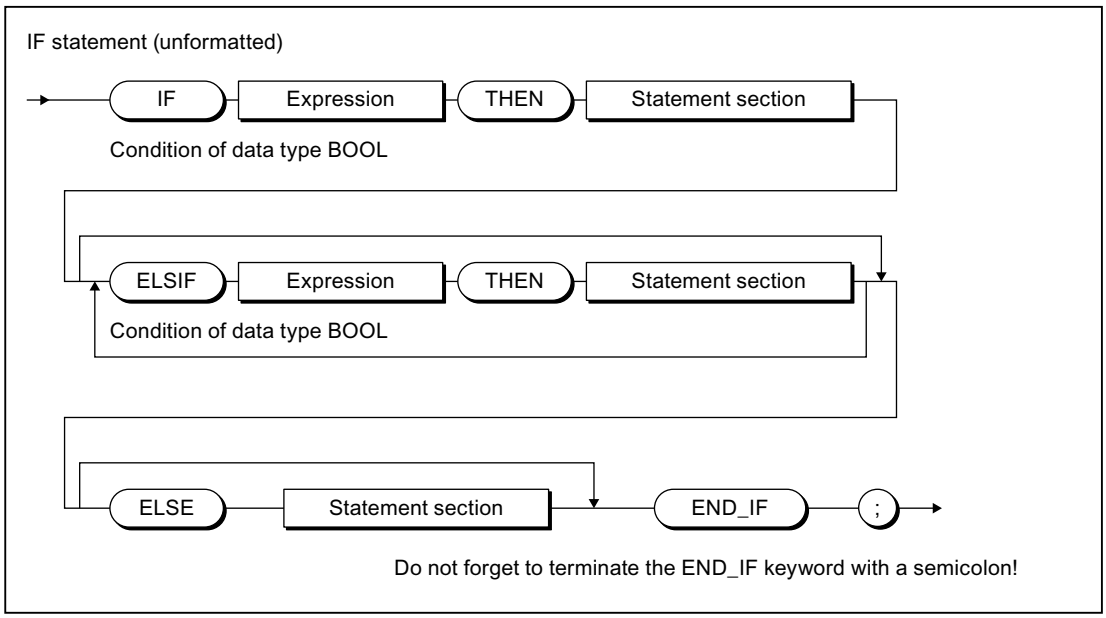



Figure 3-25 Syntax: IF statement

## Sequence of execution

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 can 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.

## Example

The following example illustrates the use of the IF statement:

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.

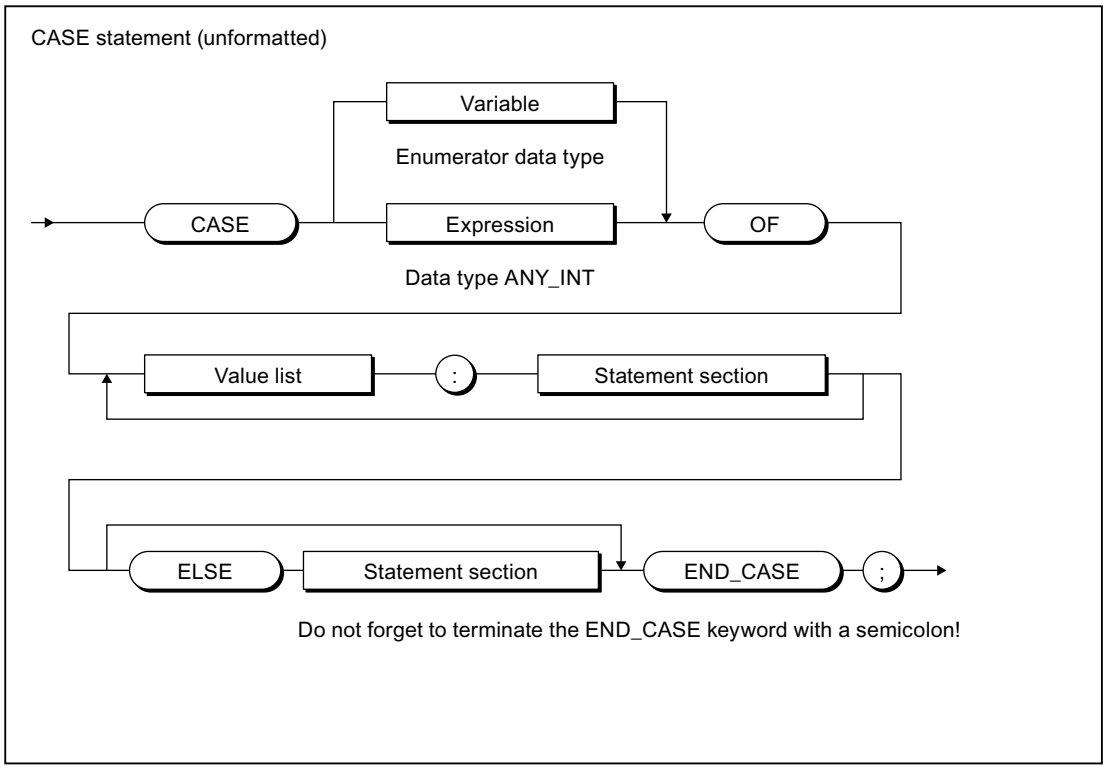



Figure 3-26 Syntax: CASE statement

### Sequence of execution

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 enumeration 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.

## Value list

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

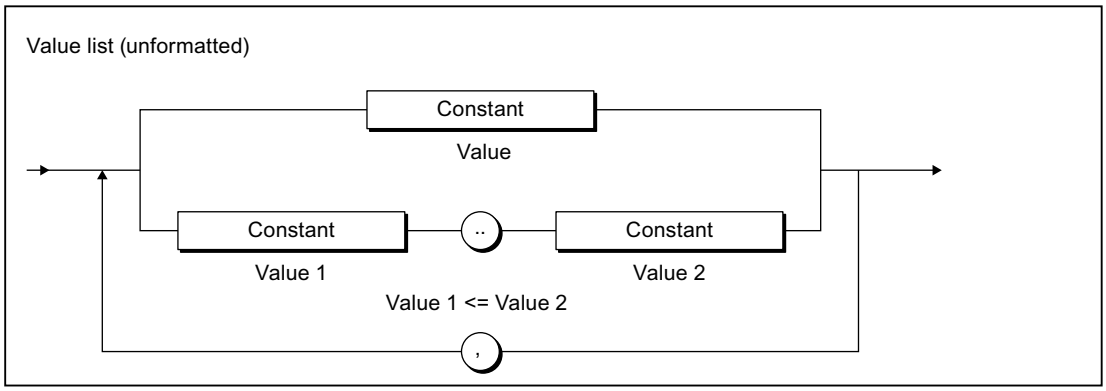



Figure 3-27 Syntax: Value list

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.

## Example

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 158) and REPEAT statement (Page 159)).

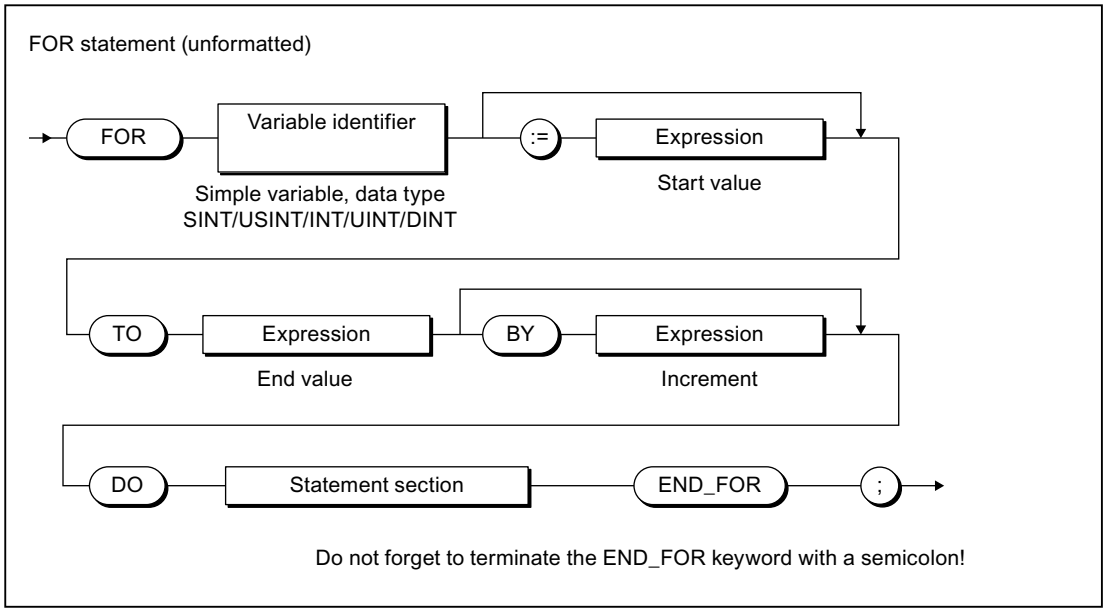



Figure 3-28 Syntax: FOR statement

## Sequence of execution

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.

## Rules

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 141)). 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.

## Example

The following example illustrates the use 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 156)).

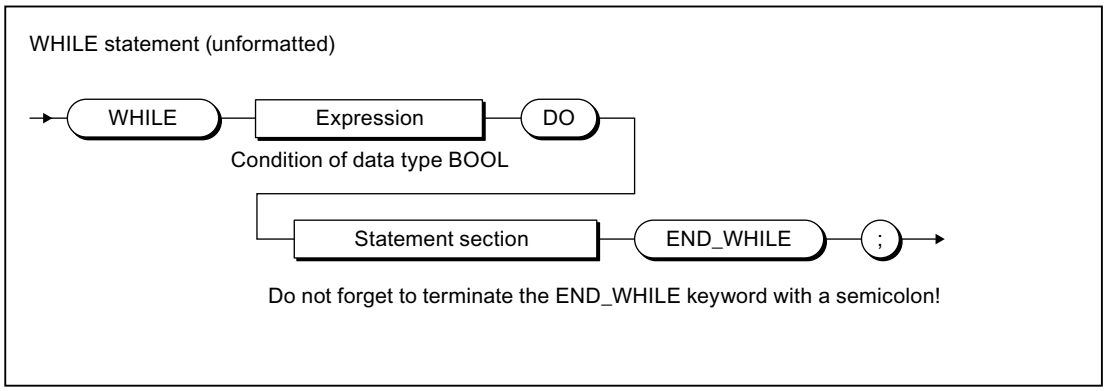



Figure 3-29 Syntax: WHILE statement

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

### Sequence of execution

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.

### Example

The following example illustrates the use of the WHILE statement:

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 156)).

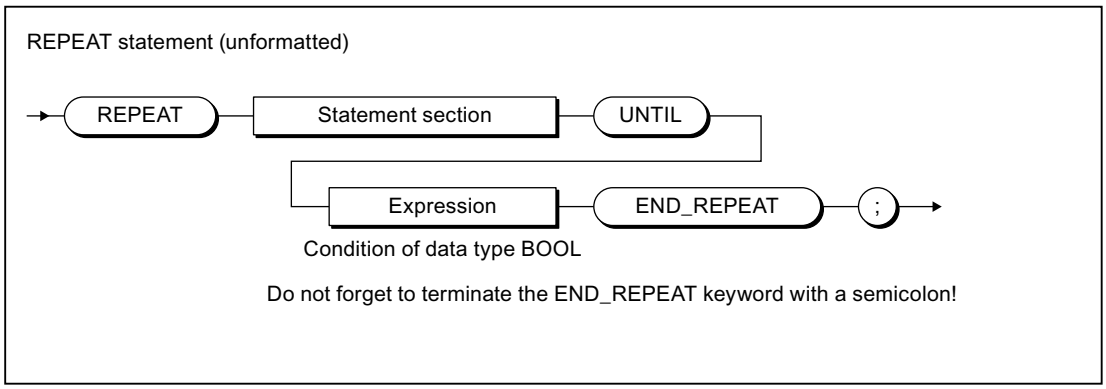



Figure 3-30 Syntax: REPEAT statement

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.

### Example

The following example illustrates the use of the REPEAT statement:

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).

### Example

The following example illustrates the use of the EXIT statement:

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 performed after the execution of EXIT or
after the regular end of the FOR loop.
*)
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.

### Example

The following example illustrates the use of the RETURN statement:

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 executed after the regular end of the
FOR loop, but 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 190). More information about the WAITFORCONDITION and expressions in this regard is contained in the *SIMOTION Motion Control Basic Functions* Function Manual.

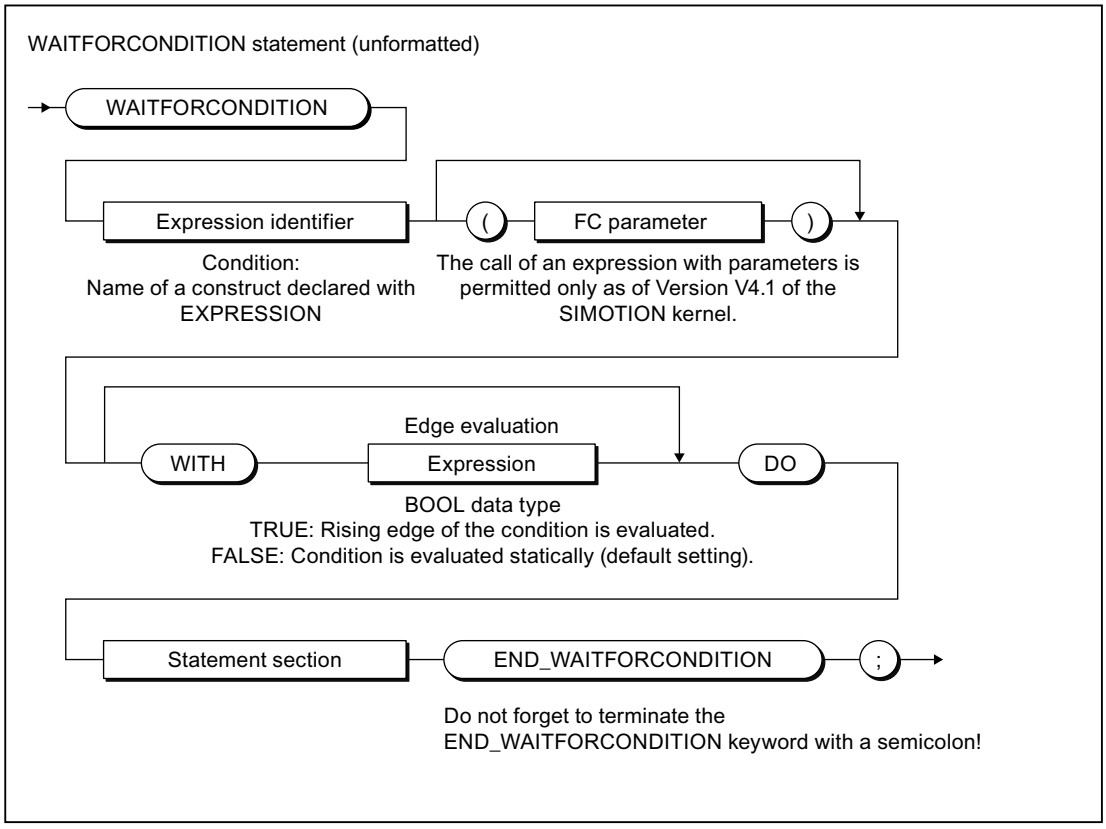



Figure 3-31 Syntax: WAITFORCONDITION statement

*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 evaluated; 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).

### Example

The following example illustrates the use of the WAITFORCONDITION statement:

Table 3- 44 Example of the WAITFORCONDITION statement

```
// ...
// Call of the command with name of the expression
WAITFORCONDITION myExpression WITH TRUE DO
// At least one statement here, 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 190).

## 3.7.9 GOTO statement

The GOTO statement causes a jump to the jump label specified in the command (see Jump statement and labeling (Page 303)).

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.

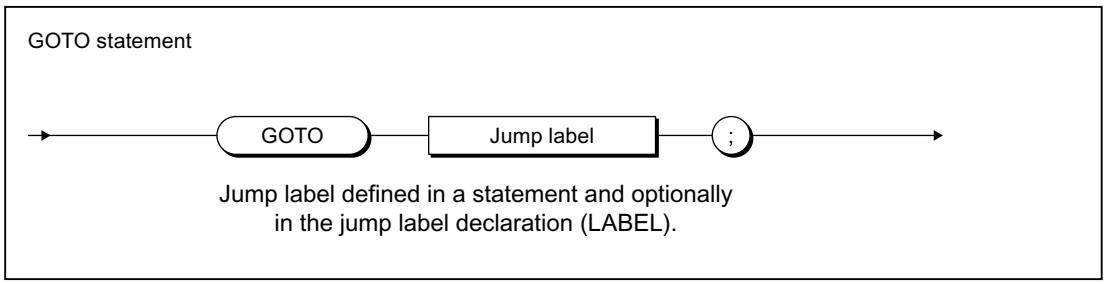



Figure 3-32 Syntax: GOTO statement

**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.

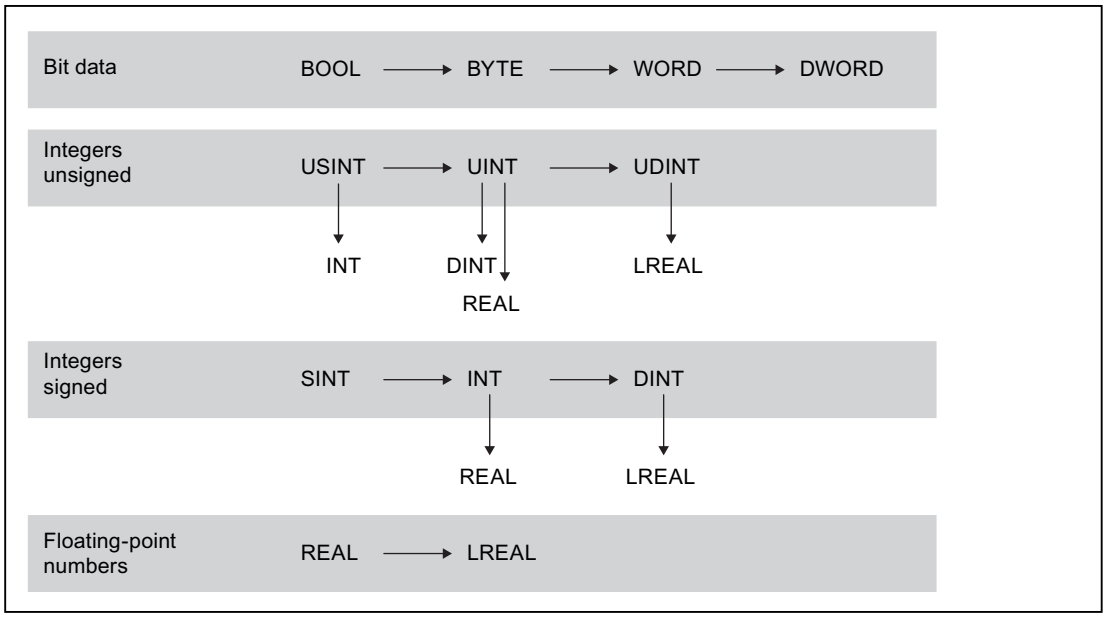



Figure 3-33 Implicit type conversion chains (one or more levels from left to right or one level from top to bottom)

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 193).

## 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 185).

The order in which you must define and call the stipulated source file sections is given in Use of the source file sections (Page 193).

How to export and import FCs and FBs is explained in Section Import and export between ST source files (Page 206).

### 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:

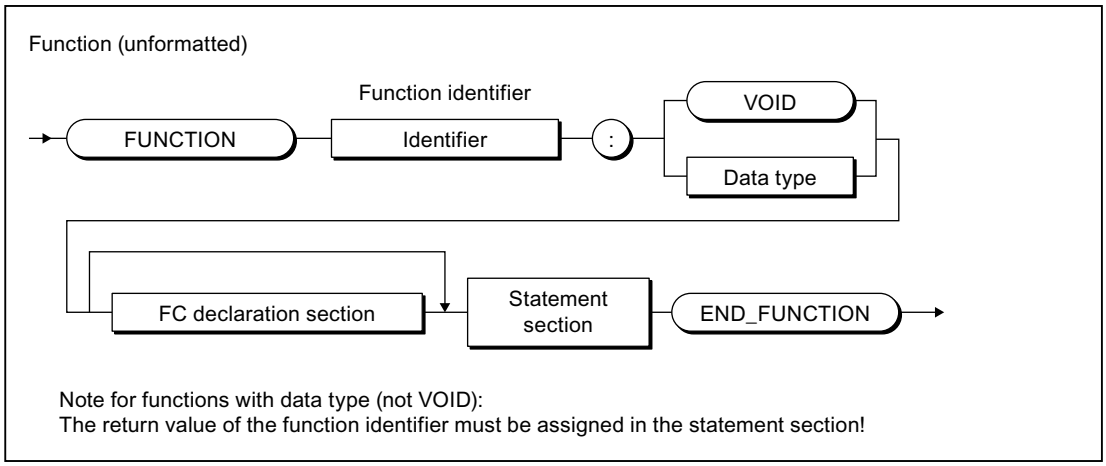



Figure 4-1 Syntax: Function (FC)

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 186)):

- 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:

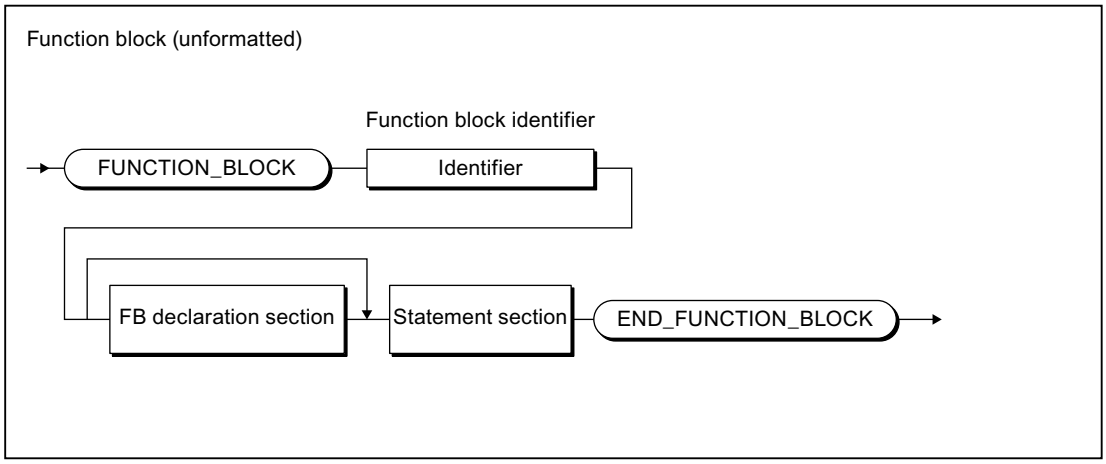



Figure 4-2 Syntax: Function block (FB)

Enter an identifier as the FB name after the FUNCTION_BLOCK keyword.

Then enter (see example in Source file with comments (Page 186)):

- 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 186)):

## Permissible declaration blocks

Table 4- 1 Declaration blocks for FC and FB: Options

| Data | Syntax | FB | FC |
|---|---|---|---|
| Constant | VAR CONSTANT<br>*Declaration list*<br>END_VAR | X | X |
| Input parameters | VAR_INPUT<br>*Declaration list*<br>END_VAR | X | X |
| In/out parameter | VAR_IN_OUT<br>*Declaration list*<br>END_VAR | X | X |
| Output parameters | VAR_OUTPUT<br>*Declaration list*<br>END_VAR | X | – |
| Local variable<br>(for FC and FB) | VAR<br>*Declaration list*<br>END_VAR | X<br>(static) | X<br>(temporary) |
| Local variable<br>(for FB) | VAR_TEMP<br>*Declaration list*<br>END_VAR | X<br>(temporary) | - |
| *Declaration list*: The list of identifiers of the type to be declared | | | |

## Parameter blocks

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.

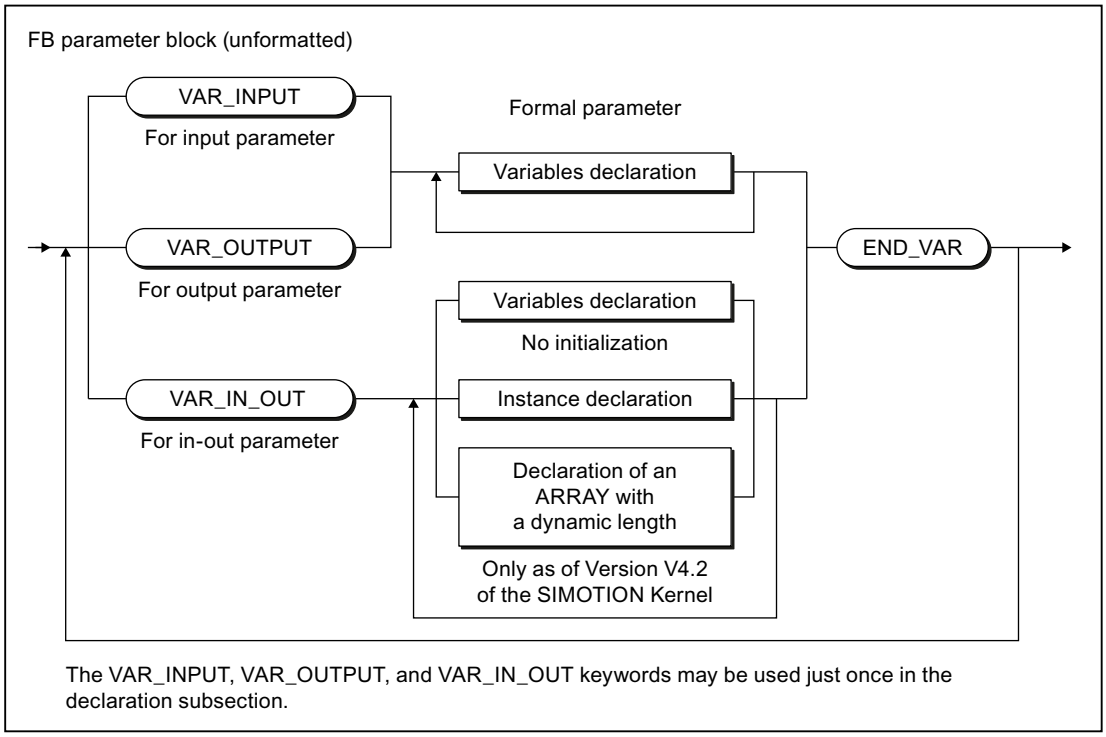



Figure 4-3 Syntax: FB parameter block

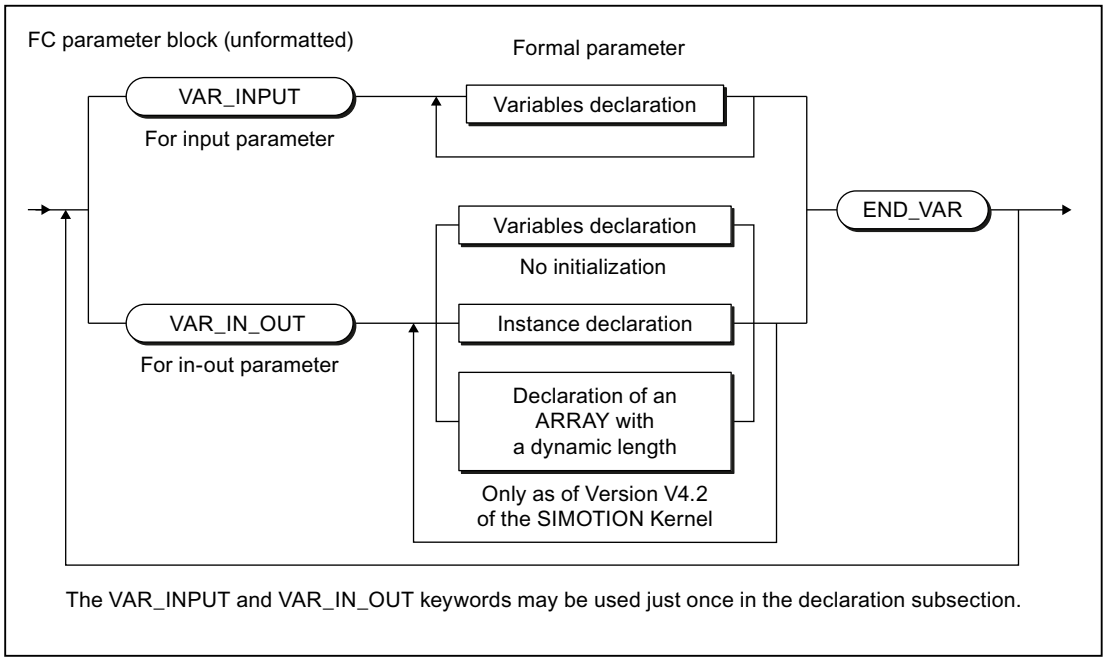



Figure 4-4 Syntax: FC parameter block

As well as variables, you can also declare the following in the parameter block for in-out parameters:

- Instances of function blocks (Page 181)
- Arrays with a dynamic length (Page 175), as of SIMOTION Kernel V4.2.

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 (Page 118).

  Input parameters can only be accessed if the "Permit language extensions" compiler option (Page 56) has been activated.

  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 expression to a variable, for example. Specifying the function name calls the function and returns a result at the same time.

### 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.<br>Only for activated "Permit language extensions" compiler option (see Global compiler settings (Page 56) or Local compiler settings (Page 59)): 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.<br>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. |

| Parameter/variable | Use |
|---|---|
| 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).<br>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.<br>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.<br>A differentiation between static and temporary local variables is made in the FB:<br>• Static variables (VAR ... END_VAR) retain their value when the FB is closed.<br>• Temporary variables (VAR_TEMP ... END_VAR) lose their value when the FB is closed. The next time the FB is called, they are reinitialized.<br>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 ARRAY with a dynamic length (as of Kernel V4.2)

### Declaration as an in-out parameter

As of SIMOTION Kernel Version 4.2, you can declare arrays with a dynamic length in functions and function blocks. This is only possible in the relevant parameter block for in-out parameters (Page 171) (VAR_IN_OUT/END_VAR).

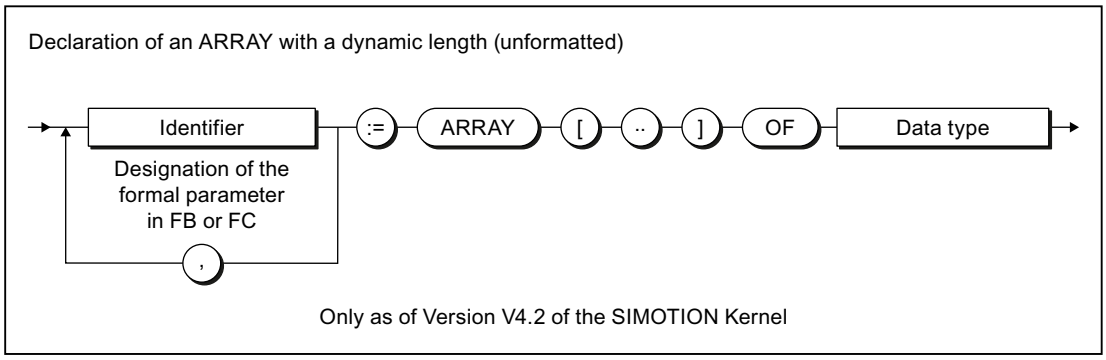



Figure 4-5 Syntax: Declaration of an ARRAY with a dynamic length

## Using an ARRAY with a dynamic length

When the function or an instance of the function block is called, arrays of any length of the declared data type can be transferred. The reference to the transferred array and its index limits are saved on the local data stack (Page 228).

You can use the _firstIndexOf (in := *array-name*) and _lastIndexOf (in := *array-name*) functions within the function or function block to define the lower and upper index limits. The _lengthIndexOf (in := *array-name*) function supplies the number of elements in the ARRAY. The following applies: _lengthIndexOf (x) := _lastIndexOf (x) - _firstIndexOf (x) + 1.

You can define the size of the memory space occupied by the ARRAY with _sizeOf (in := *array-name*). The following applies: _sizeOf (in := *array_of_type*) := _lengthIndexOf (in := *array_of_type*) * _sizeOf (in := *type*).

The syntax of these functions is described in the "SIMOTION Basic Functions" Function Manual.

**Note**

The above functions are executed during runtime for an ARRAY with a dynamic length.

## Example

Table 4- 3 Example of using an ARRAY with a dynamic length

```
FUNCTION_BLOCK example_dyn_array
    VAR_IN_OUT
        flexarray : ARRAY [..] OF DINT;
    END_VAR

    VAR_TEMP
        i : DINT := 0;
    END_VAR

    i := _firstIndexOf (in := flexarray);

    WHILE i <= _lastIndexOf (in := flexArray) DO
        flexArray[i] := i;
        i := i + 1;
    END_WHILE;
END_FUNCTION_BLOCK

PROGRAM test_dyn_array
    VAR
        array_1 : ARRAY [0 .. 29] OF DINT;
        fb_example : example_dyn_array;
    END_VAR
    // ...
    fb_example (flexarray := array_1);
    // ...
END_PROGRAM
```

## 4.1.6 Call of functions and function block calls

This provides an overview of the call of the functions and function blocks.

### 4.1.6.1 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.

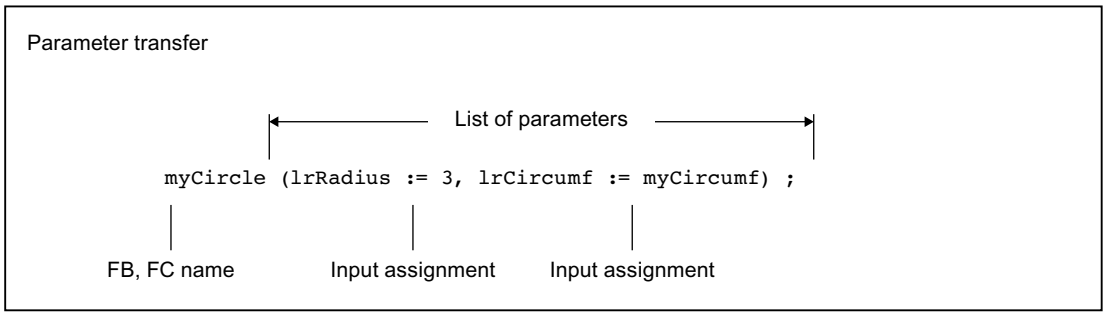



Figure 4-6 Principle of parameter transfer for the call

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).

### 4.1.6.2 Parameter transfer to input parameters

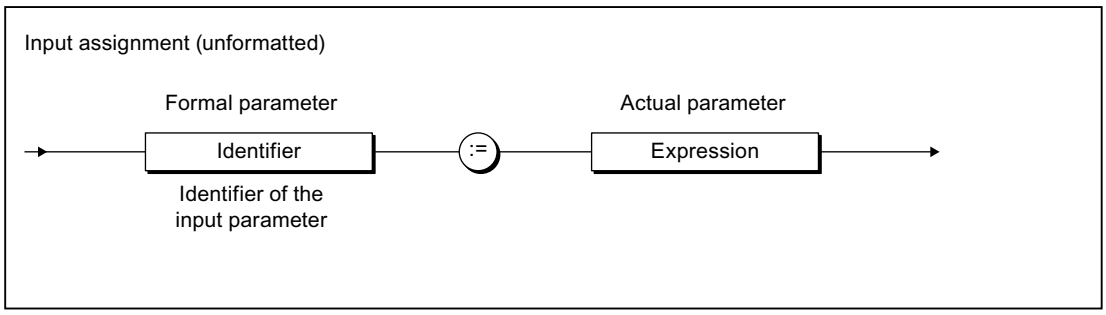



Figure 4-7 Syntax: Input assignment

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 180) and Calling function blocks (instance calls) (Page 181).

You can also gain read and write access to 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 183).

### 4.1.6.3 Parameter transfer to in/out parameters

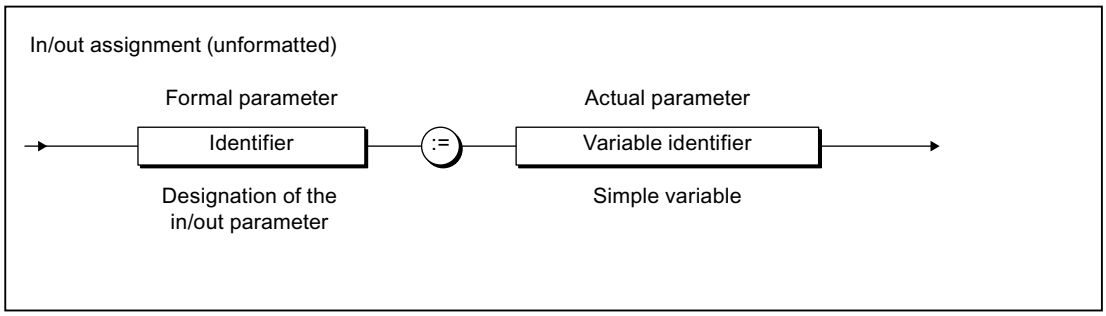



Figure 4-8 Syntax: In/out assignment

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 180) and Calling function blocks (instance calls) (Page 181).

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- 4 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!

### 4.1.6.4 Parameter transfer to output parameters (for FB only)

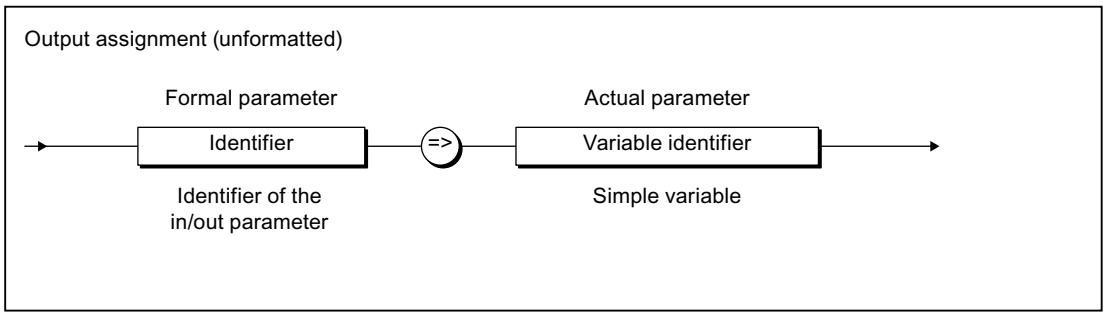



Figure 4-9 Syntax: Output assignment

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 181).

Output assignments are optional for the parameter transfer. You can gain read and write access to 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 183).

### 4.1.6.5 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 212)).

**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.

### 4.1.6.6 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).

### 4.1.6.7 Calling function blocks (declaring and calling instances)

#### Declaring an instance of a function block

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).

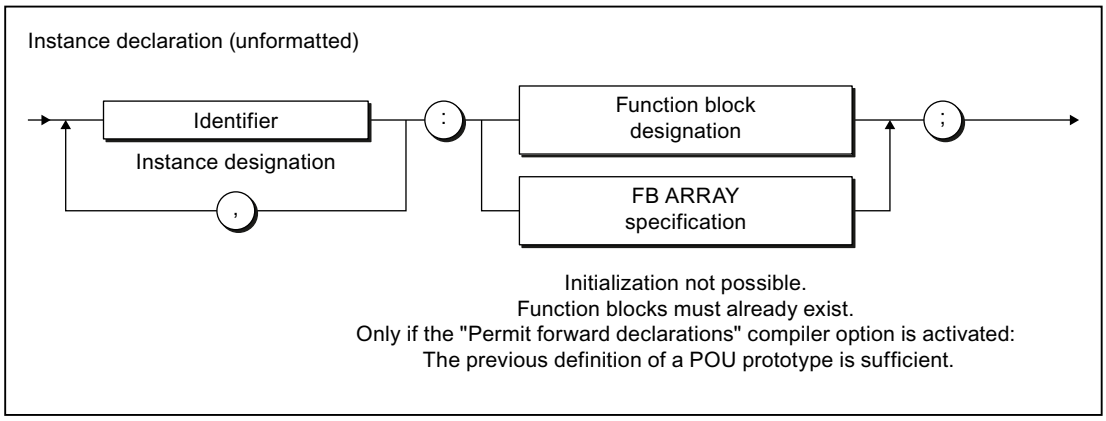



Figure 4-10 Syntax: Instance declaration

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.

## Calling an instance of a function block

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.

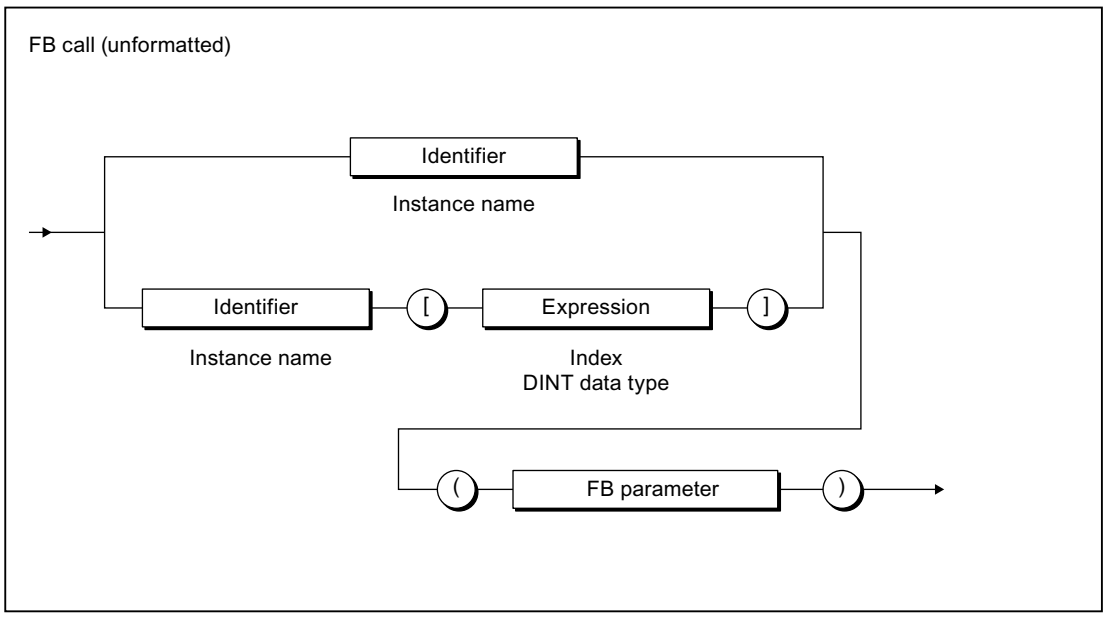



Figure 4-11 FB call syntax

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- 5 Example of instance declaration, FB call, and access to output parameters

```
VAR
    Supply1, Supply2: Supply;
    Motor1 : Motor;
END_VAR

// Parameter transfer (output assignment) when calling the instance of an FB
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);
// ...
// Accessing the FB's output parameter outside the FB
var15 := Supply1.out;
var25 := Supply2.out;
var35 := Motor1.out1;
var36 := Motor1.out2;
var41 := Motor1.out1 * Motor1.out2 * (Supply1.out + Supply2.out);
```

### 4.1.6.8 Accessing the FB's output parameter outside the FB

In addition to the output assignment (Page 179) 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 118) in the *FB instance name.output parameter* format , e.g. *Supply1.out*.

Also refer to the examples in Calling function blocks (instance calls) (Page 181).

The instance name of the FB itself must not be used in a value assignment!

### 4.1.6.9 Accessing the FB's input parameter outside the FB

In addition to the input assignment (Page 177) 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 118) 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 must be activated (see Global compiler settings (Page 56) and Local compiler settings (Page 59)). |

The instance name of the FB itself must not be used in a value assignment!

Table 4- 6 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();
```

### 4.1.6.10 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 are not possible, in particular:

  – 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 complete 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 descriptions in Define functions (Page 170) and Define function blocks (Page 171).

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- 7 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- 8 Example of the differences between FB and FC for the previous example

| Function block (FB) | Function (FC) |
|---|---|
| **Comments** | |
| Reserved words for the definition:<br>FUNCTION_BLOCK and END_FUNCTION_BLOCK | Reserved words for the definition:<br>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.<br>The *Counter* local variable is incremented; its value is retained when the FB is closed. The variable is, therefore, incremented each time the FB is called.<br>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.<br>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.<br>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.<br>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.<br>• With the call, you can assign the output parameters to a variable.<br>• You can read an FB's output parameters, even outside the FB. For this purpose, use structured variables in the following format:<br>*FB-instancename.outputparameter*. | • You assign a variable to the in/out parameters when the FB instance is called.<br>• To obtain the return value of the FC:<br>– Assign the function to a variable.<br>– 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.<br>Exception: If the "Permit language extensions" compiler option is activated (see Global compiler settings (Page 56) or Local compiler settings (Page 59)), the calling program can also access the input parameters of an FB. For this purpose, use structured variables in the following format:<br>*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.

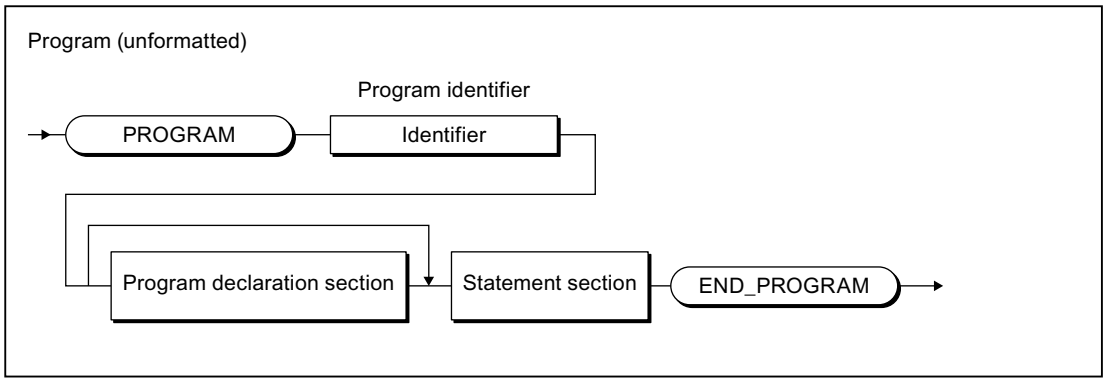



Figure 4-12 Syntax: Program

Programs are declared in the Implementation section (Page 197) 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 186) and Source text of the sample program (Page 81) sections.

## 4.3.1 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 the program in the interface section (Page 194) of the ST source file to be specified as the program organization unit to be exported.

## 4.3.2 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 the following compiler options to be activated (see Global compiler settings (Page 56) and Local compiler settings (Page 59)):

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 223). This also changes the Initialization behavior (Page 234).
- All called programs with the same name use the same program instance data.

Table 4- 9 Example for calling a program in a program

```
PROGRAM my_prog
    ; // ...
END_PROGRAM

PROGRAM main_prog
    ; // ...
    my_prog();
    ; // ...
END_PROGRAM
```

**Note**

Most of the programming work involved in assigning programs to tasks can be done if programs are called from within a program. In the execution system, only one calling program needs to be assigned to the associated tasks in each case.

# 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 162).

An expression can only be declared in the implementation section of the ST source file.

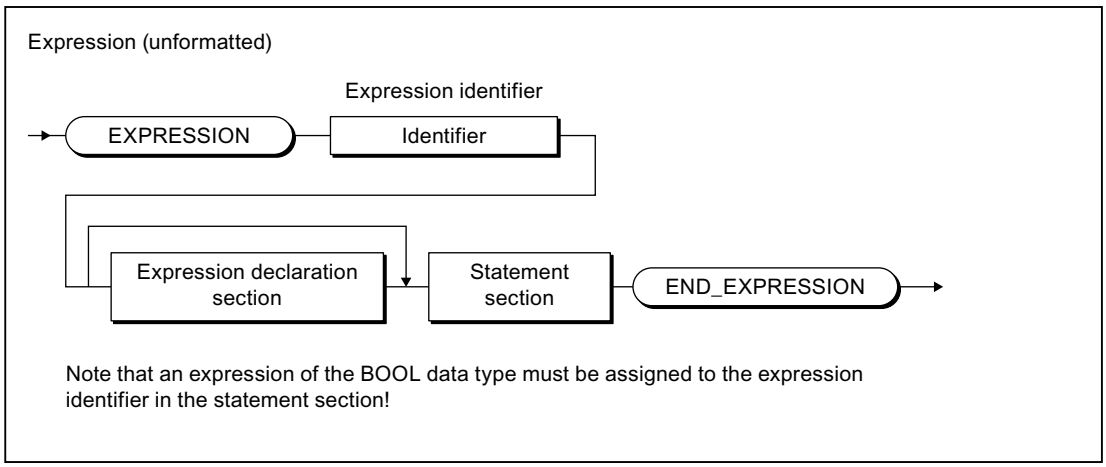



Figure 4-13 Syntax: Expression

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- 10 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 102). 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 89)).

**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`<br>`  // Interface statements (optional)`<br>`END_INTERFACE`<br>An individual identifier of the section cannot be specified.<br>Optionally, interface statements exist in the following order between reserved words INTERFACE and END_INTERFACE.<br>1. Specification of utilized technology package. Syntax:<br>`USEPACKAGE tp-name [AS namespace];`<br>For more details, refer to the *SIMOTION Basic Functions* Function Manual.<br>2. Specification of utilized libraries.<br>Syntax:<br>`USELIB library-name-list [AS namespace];`<br>For more information, see "Using data types, functions and function blocks from libraries (Page 274)".<br>3. Reference to other units in order to use their exported components.<br>Syntax:<br>`USES unit_name-list;`<br>For more information, see "USES statement in an importing unit (Page 209)".<br>4. Declarations and specifications for the export<br>– Data type definitions (Page 203):<br>User-defined data types (UDT) that are valid in the entire ST source file and that are to be exported<br>– Variable declarations (Page 203):<br>Unit variables and unit constants valid in the entire ST source file and exported.<br>Permissible keywords: See table in "Variable declaration (Page 203)".<br>– Specification of the program organization units (POUs) to be exported.<br>Syntax:<br>`FUNCTION fc_name;`<br>`FUNCTION_BLOCK fb_name;`<br>`PROGRAM program_name;`<br>If the "Permit forward declarations" compiler option (Page 56) is activated, they are also interpreted as POU prototypes for the forward declaration (Page 300).<br>– POU prototypes for the forward declaration (Page 300).<br>Specification of the prototypes for program organization units with forward declaration (only effective if the "Permit forward declarations" compiler option (Page 56) is activated).<br>They are also interpreted as POUs to be exported.<br>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 206)". |

| | |
|---|---|
| Sequence | The interface section is the first section of an ST source file[1].<br>The order of the interface statements 1 to 4 is fixed.<br>Within number 4, any order is permitted. The individual declaration blocks for data type definitions (including POU prototypes) and variables declarations can appear more than once.<br>Attention: Identifiers must be declared before they are used.<br>Only if the "Permit forward declarations" compiler option is activated: When declaring an instance of a function block, it is sufficient for the prototype of the function block to have been declared previously. |
| 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 206)". | |

#### 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`<br>`  // Implementation statements (optional)`<br>`END_IMPLEMENTATION`<br>An individual identifier of the section cannot be specified.<br>Optionally, implementation statements (main part of the ST source file) exist in the following order between the reserved words IMPLEMENTATION and END_IMPLEMENTATION:<br>1. Reference to other units in order to use their exported components. Syntax:<br>`USES unit_name-list;`<br>For more information, see "USES statement in an importing unit (Page 209)".<br>2. Declarations<br>– Data type definitions (Page 203):<br>User-defined data types (UDT) that are valid in the entire ST source file<br>– Variable declarations (Page 203):<br>Unit variables and constants that are valid in the entire ST source file<br>Permissible keywords: See table in "Variable declaration (Page 203)".<br>– POU prototypes for the forward declaration (Page 300).<br>Specification of the prototypes for program organization units with forward declaration (only effective if the "Permit forward declarations" compiler option (Page 56) is activated).<br>3. Program organization units (POUs) (Page 198) |
| Sequence | Always follows the interface section.<br>The order of the implementation statements indicated above is mandatory; within number 2 and 3, any order is permitted:<br>Attention: Identifiers must be declared before they are used.<br>Only if the "Permit forward declarations" compiler option (Page 56) is activated: When declaring an instance of a function block, it is sufficient for the prototype of the function block to have been declared previously. |
| Frequency | Once per ST source file |
| Mandatory section | yes |

### 5.1.1.3 Program organization units (POUs)

Program organization units (POUs) are the executable source file sections:

- Functions (FC) (Page 198)
- Function blocks (FB) (Page 199)
- Programs (Page 200)
- Expressions (Page 200)

Usually, the called POUs must precede the calling POUs in the source file so that the former are recognized by the latter.

**Exception:** Only if the "Permit forward declarations" compiler option (Page 56) is activated. Any order is permitted. The following applies to the specification of a prototype for forward declaration (Page 300) in the interface section (Page 194) or the implementation section (Page 197):

- Essential for declaring instances with function blocks
- Optional when calling functions and programs

### 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`<br>`  // Declaration section`<br>`  // Statement section`<br>`END_FUNCTION`<br>*name* stands for the identifier of the function, while *function_data_type* stands for the data type of the return value.<br>Permissible keywords for the variables declaration in the declaration section (Page 201): See table in "Variable declaration (Page 203)".<br>Note the following for functions with *function_data_type* <> VOID: In the body (Page 202), 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 (Page 197).<br>Usually, functions must precede the POU from which they are called.<br>**Exception:** Only if the "Permit forward declarations" compiler option (Page 56) is activated. Any order is permitted. There is an option to specify a prototype for the forward declaration (Page 300) in the interface section (Page 194) or the implementation section (Page 197).<br>The declaration section (Page 201) must precede the statement section (Page 202). |
| 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 169).

### 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`<br>`  // Declaration section`<br>`  // Statement section`<br>`END_FUNCTION_BLOCK`<br>*name* stands for the identifier of the function block.<br>Permissible keywords for the variable declaration in the declaration section: See table in "Variable declaration (Page 203)". |
| 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).<br>You can also declare the instance globally (within VAR_GLOBAL/END_VAR in the interface or implementation section). As standard, this is only possible with function blocks which make imported program sources and libraries available, but not with function blocks which are defined in the same ST source file.<br>You can only declare global instances of function blocks which are defined in the same ST source file if the "Permit forward declarations" compiler option (Page 56) is activated. It is essential that a prototype for the forward declaration (Page 300) is specified in the interface or implementation section.<br>You cannot declare an instance of an FB in FCs. |
| Sequence | FBs can only be defined in the implementation section (Page 197).<br>Pay attention to the order: FBs must precede the POU in which an instance is declared as local variable.<br>**Exception:** Only if the "Permit forward declarations" compiler option (Page 56) is activated. Any order is permitted. It is essential that a prototype for the forward declaration (Page 300) is specified in the interface section (Page 194) or the implementation section (Page 197).<br>The declaration section (Page 201) must precede the statement section (Page 202). |
| 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 169).

### 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`<br>`  // Declaration section`<br>`  // Statement section`<br>`END_PROGRAM`<br>*name* stands for the name of the program.<br>Permissible keywords for the variable declaration in the declaration section: See table in "Variable declaration (Page 203)". |
| Sequence | Programs can only be defined in the implementation section (Page 197).<br>It is advantageous to place programs after expressions, FCs, and FBs. This enables the program to recognize and use the source file sections.<br>**Exception:** Only if the "Permit forward declarations" compiler option (Page 56) is activated.<br>Any order is permitted. When a program is called within a program, there is an option to specify a prototype for the forward declaration (Page 300) in the interface section (Page 194) or the implementation section (Page 197).<br>The declaration section (Page 201) must precede the statement section (Page 202). |
| Frequency | Any number of times per ST source file |
| Mandatory section | no |

For more information about programs, see Programs (Page 188).

### 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`<br>` // Declaration section`<br>` // Statement section`<br>`END_EXPRESSION`<br>*name* stands for the identifier of the expression.<br>Permissible keywords for the variable declaration in the declaration section: See table in "Variable declaration (Page 203)".<br>Attention: In the body (Page 202), an expression of data type BOOL must be assigned to the expression identifier. |
| Sequence | An expression can only be declared in the implementation section (Page 197) of an ST source file.<br>Usually, expressions precede the program in which they are called from a WAITFORCONDITION control structure.<br>**Exception:** Only if the "Permit forward declarations" compiler option (Page 56) is activated. Any order is permitted. There is no option to specify a prototype for the forward declaration (Page 300) in the interface section (Page 194) or the implementation section (Page 197).<br>The declaration section (Page 201) must precede the statement section (Page 202). |
| Frequency | Any number of times per ST source file |
| Mandatory section | no |

For more information on expressions, see Expressions (Page 190). 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 variables declaration of the POU.

The declaration section has the following structure:

Table 5- 7 Structure of the declaration section

| | |
|---|---|
| Structure | `// Data type definition`<br>`// 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.<br>It contains the following in any order:<br>• Data type definitions (Page 203):<br>User-defined data types (UDT) that are valid locally in the POU<br>• Variable declarations (Page 203):<br>Variables and constants that are valid locally in the POU<br>Permissible keywords according to the respective POU: See table in "Variable declaration (Page 203)".<br>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 133)
- Control statements (Page 152)
- Calling functions and function blocks (Page 177)

### 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`<br>`name : data_type_specification;`<br>`// ...`<br>`END_TYPE`<br>*name* represents the name of the individual data type that you use for the Variable declarations.<br>*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:<br>• In the Interface section:<br>The UDTs are recognized within the ST source file and will be exported<br>They can be used in the interface and implementation section for declaration of unit variables and in all POUs for declaration of local variables.<br>In addition, they can be used in all units which import this ST source file (in SIMOTION ST with the USES statement).<br>• In the Implementation section:<br>The UDTs are recognized within the ST source file<br>They can be used in the implementation section for declaration of unit variables and in all POUs for declaration of local variables.<br>• In the Declaration section of a POU (FC, FB, program, expression)<br>The UDTs are only recognized locally within the POU<br>They can only be used within the POU for declaration of local variables.<br>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 110).

### 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`<br>` name_list : data_type;`<br>` // ...`<br>`END_VAR`<br>*variable_type* represents the keyword of the variable type being declared. The permitted keywords depend on the source file section.<br>• In the Interface section or Implementation section of an ST source file:<br>`VAR_GLOBAL`: Non-retentive unit variable<br>`VAR_GLOBAL CONSTANT`: Unit constant<br>`VAR_GLOBAL RETAIN`: Retentive unit variable<br>• In the Declaration section of a function:<br>`VAR`: Local variable<br>`VAR CONSTANT`: Local constant<br>`VAR_INPUT`: Input parameter<br>`VAR_IN_OUT`: In/out parameter<br>• In the Declaration section of a function block:<br>`VAR`: Local variable<br>`VAR CONSTANT`: Local constant<br>`VAR_TEMP`: Temporary variable<br>`VAR_INPUT`: Input parameter<br>`VAR_OUTPUT`: Output parameter<br>`VAR_IN_OUT`: In/out parameter<br>• In the Declaration section of a program:<br>`VAR`: Local variable<br>`VAR CONSTANT`: Local constant<br>`VAR_TEMP`: Temporary variable<br>• In the Declaration section of an expression:<br>`VAR`: Local variable<br>`VAR CONSTANT`: Local constant<br>`VAR_INPUT`: Input parameter (as of Version 4.1 of the SIMOTION kernel)<br>`VAR_IN_OUT`: In/out parameter (as of Version 4.1 of the SIMOTION kernel)<br>*name_list* is the list of identifiers of the *data_type* data type to be declared. |

| | |
|---|---|
| Sequence | The variable is declared:<br>• In the Interface section of the ST source file:<br>Permissible keywords: see table field syntax.<br>The unit variables are recognized within the ST source file and will be exported.<br>They can be used in all POUs of the ST source file.<br>In addition, they can be used in all units which import this ST source file (in SIMOTION ST with the USES statement).<br>• In the Implementation section of the ST source file:<br>Permissible keywords: see table field syntax.<br>The unit variables are recognized within the ST source file.<br>They can be used in all POUs of the ST source file.<br>• In the Declaration section of a POU (FC, FB, program, expression)<br>Permissible keywords according to the type of POU: See table cell *Syntax*.<br>The variables are only recognized locally within the POU.<br>They can only be used within the POU for declaration of local variables.<br>Exceptions:<br>– You can also access the output parameters of a function block outside the FB.<br>– 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 56) and Local settings of the compiler (Page 59).<br>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:<br>• In the interface and implementation section of the ST source:<br>The declaration blocks may appear more than once.<br>• In the declaration section of a POU (FC, FB, program, expression):<br>Each declaration block (other than VAR CONSTANT / END_VAR) may appear just once in the declaration section.<br>Permitted declaration blocks and keywords depending on the associated source file section: See table cell *Syntax*.<br>Any number of variable declarations are possible within a declaration block. |
| Mandatory section | no |

For more information about variable declarations, see Variable declaration (Page 124).

## 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 25).

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 194)".

- The data 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 (optional within the TYPE/END_TYPE construct). Close each entry with a semicolon.

  – FUNCTION_BLOCK *fb_name*;

  – FUNCTION *fc_name*;

  – PROGRAM *program_name*;

  If the "Permit forward declarations" compiler option (Page 56) is activated, they are also interpreted as POU prototypes for the forward declaration (Page 300).

Specifications can be made in any order; the POU itself is programmed in the implementation section (Page 197) 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 209)".

You must adhere to the order presented for the specifications in the interface section of a unit (ST source file), see "Interface section (Page 194)". Otherwise, error-free compilation of the ST source file will not be possible.

Programs which are to be assigned to a task in the execution system must be listed in the interface section (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, function blocks, and programs 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 211)).

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.<br>Example:<br>1. Unit B imports Unit A in the **interface** section.<br>2. Unit C in turn imports Unit B.<br>3. Then Unit C also imports Unit A automatically.<br>A → B → C ⇒ A → C<br>Because of inheritance, Unit A must not be imported explicitly into Unit C. | Inheritance is interrupted.<br>Example:<br>1. Unit B imports Unit A in the **implementation** section.<br>2. Unit C in turn imports Unit B.<br>3. Then Unit C has no automatic access to Unit A.<br>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:<br>• Interface section<br>• 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 208).

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:<br>• Within the SIMOTION device from all programs<br>• From HMI devices<br>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:<br>• Within the SIMOTION device from all programs<br>• From HMI devices<br>You create these variables in the symbol browser after you have selected the I/O element in the project navigator.<br>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.<br>You create these variables in the symbol browser after you have selected the GLOBAL DEVICE VARIABLES element in the project navigator.<br>Global device variables can be defined as retentive. This means that they will remain stored even when the SIMOTION device power supply is disconnected.<br>You can monitor global device variables in the symbol browser. |
| 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.<br>Declare these variables in the unit:<br>• In the interface section:<br>These variables are exported and can be imported into other units (e.g. ST source files, MCC units, LAD/FBD units). They are also available on HMI devices as standard.<br>• In the implementation section:<br>You can only access these variables within the associated unit.<br>You can declare unit variables as retentive. This means that they will remain stored even when the SIMOTION device power supply is disconnected.<br>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:<br>• 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).<br>You can monitor static variables in the symbol browser.<br>• Temporary variables are initialized every time the program in a task is called.<br>Temporary variables cannot be monitored in the symbol browser. |

| Variable type | Meaning |
|---|---|
| Variable of a function (FC variable) | Variable is declared in a function (FC). The variable can only be accessed within this function.<br>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:<br>• 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.<br>You can monitor static variables in the symbol browser.<br>• Temporary variables lose their value when the FB terminates. The next time the FB is called, they are reinitialized.<br>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:
  - Accessing I/O addresses with I/O variables (see Direct access and process image of the cyclic tasks (Page 245))
  - Accessing the process image (see Access to the fixed process image of the BackgroundTask (Page 256))
  - Creating and using global device variables (see Using global device variables (Page 222))
  - 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 228).

### 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 206).

  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 238) and Controlling compiler with attributes (Page 294).

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 131)).

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.

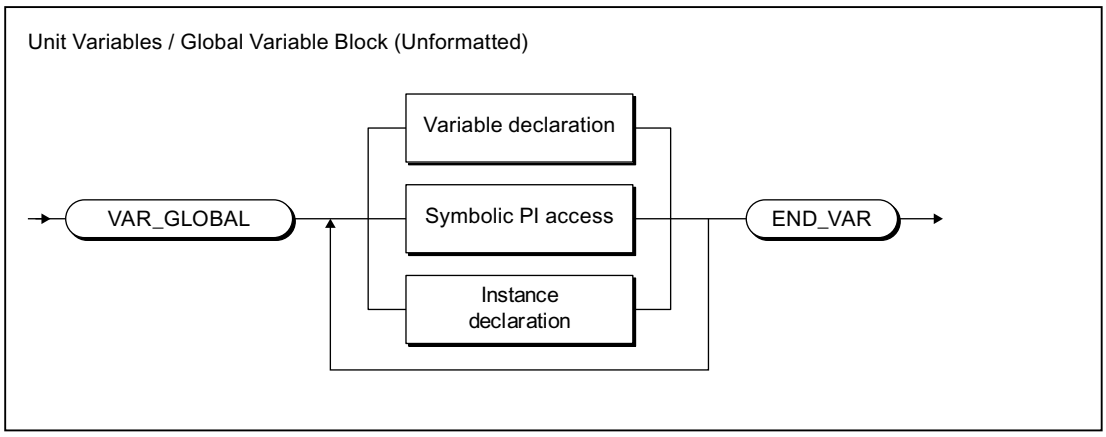



Figure 5-1 Syntax: Unit variables

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 126) and Initialization of variables or data types (Page 127)).

For the scope of the declaration and the HMI export, see Unit variables (Page 215).

**Note**

For initialization of the non-retentive unit variables:

- See Initialization of non-retentive global variables (Page 231).
- The behavior during downloading can be set (**Options > Settings** menu command, **Download** tab, **Initialize non-retentive program data and global device variables** 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 237).

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.
                  //
        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.

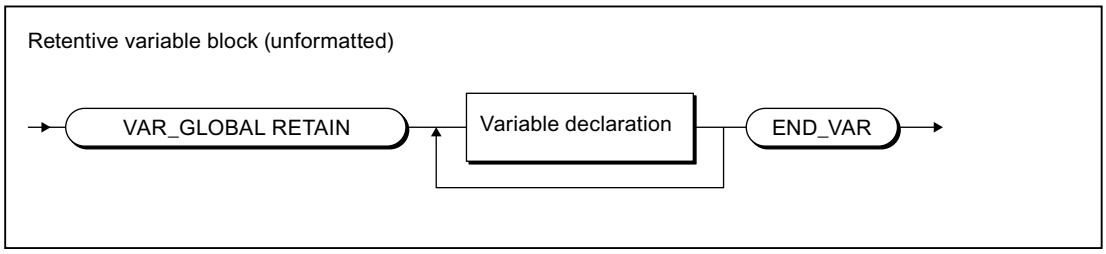



Figure 5-2 Syntax: Retentive variable block

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 126) and Initialization of variables or data types (Page 127)).

For the scope of the declaration and the HMI export, see Unit variables (Page 215).

**Note**

- For initialization of the retentive unit variables:
  - See Initialization of retentive global variables (Page 229).
  - The behavior during downloading can be set (**Options > Settings** menu command, **Download** tab, **Initialize retentive program data and global device variables** 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 237).
- 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..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 220):

  Static variables retain their value over all passes of the source file section (block memory).

- Temporary variables (Page 221):

  Temporary variables are initialized each time the source file section is called again.

See also: Initialization of local variables (Page 233).

**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<br>or<br>VAR_INPUT / END_VAR<br>or<br>VAR_IN_OUT / END_VAR[2] |
| Expression | – | VAR / END_VAR<br>or<br>VAR_INPUT / END_VAR<br>or<br>VAR_IN_OUT / END_VAR[2] |
| Function block | VAR / END_VAR[1]<br>or<br>VAR_INPUT / END_VAR[1]<br>or<br>VAR_OUTPUT / END_VAR[1] | VAR_TEMP / END_VAR<br>or<br>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 235).

[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 234).

**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 228).

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.

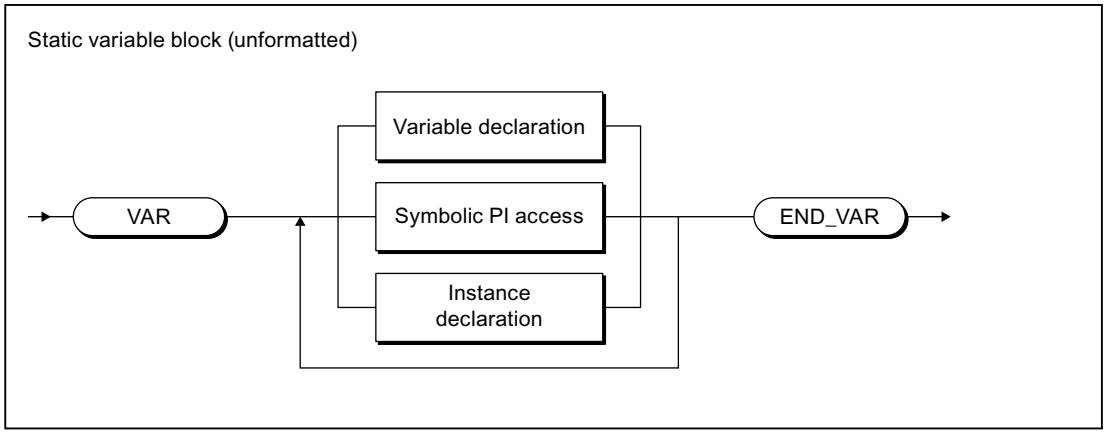



Figure 5-3 Syntax: Static variable block

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 234).
- In function blocks: Depending on the initialization of the declared instance.

  See also Initialization of instances of function blocks (FBs) (Page 235).

### 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):

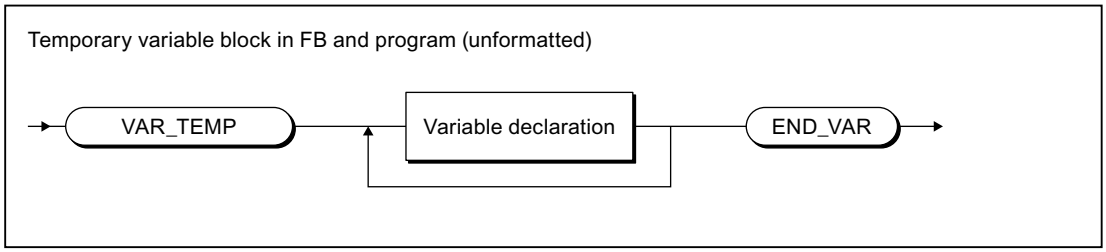



Figure 5-4 Syntax: Temporary variable block in the FB or program

In functions and expressions, you declare temporary variables in the FC temporary variable block (see following figure):

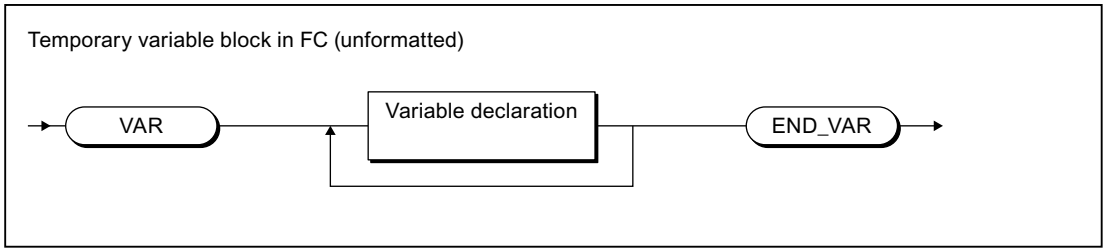



Figure 5-5 Syntax: Temporary variable block in an FC

## 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 source files) 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:
   - Activation of **Retain** checkbox (This declares the variable as retentive, so that its value will be retained after a power failure.)
   - **Field 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 (Page 225) section.

Table 5- 18 Memory areas assigned to different variable types and their initialization

| Memory area | Assigned variable types | Initialization[3] |
|---|---|---|
| Retentive memory | Retentive unit variables | During download using the download settings |
| User memory of unit | • Non-retentive unit variables<br>• Function block instances declared with VAR_GLOBAL, including the associated static variables (VAR, VAR_INPUT, VAR_OUTPUT)<br>Also for the activated "Create program instance data only once" compiler option (Page 56):<br>• Local variables of the unit programs declared with VAR<br>• Function block instances declared with VAR_GLOBAL, including the associated static variables (VAR, VAR_INPUT, VAR_OUTPUT) | • When the device is switched on<br>• During download using the download settings<br>• For transition to the RUN mode:<br>– Version V4.2 and higher of the SIMOTION Kernel:<br>By activating the check box on the SIMOTION device, **Initialization of non-retentive global variables and program data during STOP-RUN transition**.<br>– As of Version V4.1 of the SIMOTION Kernel:<br>If the associated declaration block specifies the following pragma:<br>{ BlockInit_OnDeviceRun := ALWAYS; }<br>See also Controlling compiler with attributes (Page 294) |
| User memory of task | For the deactivated "Create program instance data only once" compiler option (Page 56) (default):<br>• Local variables declared with VAR of the assigned programs<br>• 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:<br>• Sequential tasks:<br>Each time task is started<br>• Cyclic tasks:<br>For CPU transition to the RUN mode |

| Memory area | Assigned variable types | Initialization[3] |
|---|---|---|
| Local data stack of task[2] | • Reference (pointer) to the program called in the task<br>• 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<br>• Local variables of function blocks declared with VAR_TEMP<br>• 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]<br>• Return value of called functions | Each time the function is called |

[1] References (pointers) to the transferred variables.

[2] See also Memory requirement of the variables on the local data stack (Page 228).

[3] For a detailed description of the initialization behavior of the individual variable types, see Time of the variable initialization (Page 228).

#### 5.2.3.1 Example of memory areas

Table 5- 19 Example of memory areas of the variable types - 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 areas of the variable types - 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 areas of the variable types - 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

The variables stored on the local data stack of a task are listed in Memory ranges of the variable types (Page 223). You set the stack size for each task in the **Task configuration** tab. For more information, see the Description for each user task in the Basic Functions Function Manual.

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.

You can obtain information about the memory requirements of a POU in the local data stack using the Program Structure (Page 286) 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 P320 or P350.
- By applying the *_resetUnitData* function, possible selectively for different data segments of the retentive data.
- When a download is performed according to the following description.

## Behavior during download

Table 5- 22 Initializing retentive global variables during download

| Variable type | Time of the variable initialization |
|---|---|
| Retentive global device variables | The behavior during the download depends on the *Initialization of retentive program data and retentive global device variables* setting[1]:<br>• **Yes**[2]: All retentive global device variables are initialized.<br>• **No**[3]: The retentive global device variables are only initialized if their version code is changed.<br>See: Version code of global variables and their initialization during download (Page 237). |
| Retentive unit variables | The behavior during the download depends on the *Initialization of retentive program data and retentive global device variables* setting[1]:<br>• **Yes**[2]: All retentive unit variables (all units) are initialized.<br>• **No**[3]: A data block ( = declaration block)[4] of the retentive unit variables in the interface or implementation section is only initialized[5] if its version code is changed.<br>See: Version code of global variables and their initialization during download (Page 237). |

[1] Default setting in the **Options > Settings** menu, **Download** tab,
or the current setting for the download.

[2] The corresponding check box is active.

[3] The corresponding check box is inactive.

[4] With the **SIMOTION ST** programming language:
A data block of the retentive unit variables corresponds to a VAR_GLOBAL RETAIN/END_VAR declaration block in the interface section or implementation section.
With the **SIMOTION MCC** and **SIMOTION LAD/FBD** programming languages:
A data block of the retentive unit variables is formed as follows from the variables declared with VAR_GLOBAL RETAIN in the interface section or implementation section of the declaration table: Pragma lines within a section of the declaration table separate the variables into different data blocks.

[5] Initialization of a changed data block also occurs during a download in RUN, provided the following condition is fulfilled:
With the **SIMOTION ST** programming language:
The following attribute has been specified within a pragma in the relevant declaration block:
{ BlockInit_OnChange := TRUE; }.
With the **SIMOTION MCC** or **SIMOTION LAD/FBD** programming languages:
A pragma line has been pasted into the declaration table and the following check box is activated: *Initialization of VAR_GLOBAL RETAIN during a change*. All the variables declared with VAR_GLOBAL RETAIN up to the next pragma line or the end of the table form a data block accordingly.
For information on the general conditions for a 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:

- During the initialization of retentive global variables (Page 229), e.g. during a firmware update or overall reset (MRES).
- During switch-on.
- By applying the *_resetUnitData* function, possible selectively for different data segments of the non-retentive data.
- During a download as described in the following table.
- During the transition from STOP to RUN mode as described at the end of the section.

## Behavior during download

Table 5- 23 Initializing non-retentive global variables during download

| Variable type | Time of the variable initialization |
|---|---|
| Non-retentive global device variables | The behavior during the download depends on the *Initialization of non-retentive program data and non-retentive global device variables* setting[1]:<br>• **Yes**[2]: All non-retentive global device variables are initialized.<br>• **No**[3]: The non-retentive global device variables are only initialized if their version code is changed.<br>See: Version code of global variables and their initialization during download (Page 237). |
| Non-retentive unit variables | The behavior during the download depends on the *Initialization of non-retentive program data and non-retentive global device variables* setting[1]:<br>• **Yes**[2]: All non-retentive unit variables (all units) are initialized.<br>• **No**[3]: A data block ( = declaration block)[4] of the non-retentive unit variables in the interface or implementation section is only initialized[5] if its version code is changed.<br>See: Version code of global variables and their initialization during download (Page 237). |

[1] Default in the **Options > Settings** menu, **Download** tab, or the current setting for the download.

[2] The corresponding check box is active.

[3] The corresponding check box is inactive.

[4] With the **SIMOTION ST** programming language:
A data block of the non-retentive unit variables corresponds to a VAR_GLOBAL/END_VAR declaration block in the interface section or implementation section.
With the **SIMOTION MCC** or **SIMOTION LAD/FBD** programming languages:
A data block of the non-retentive unit variables is formed as follows from the variables declared with VAR_GLOBAL in the interface section or implementation section of the declaration table: Pragma lines within a section of the declaration table separate the variables into different data blocks.

[5] Initialization of a changed data block also occurs during a download in RUN, provided the following condition is fulfilled:
With the **SIMOTION ST** programming language:
The following attribute has been specified within a pragma in the relevant declaration block:
{ BlockInit_OnChange := TRUE; }.
With the **SIMOTION MCC** or **SIMOTION LAD/FBD** programming languages:
A pragma line has been pasted into the declaration table and the following check box is activated: *Initialization of VAR_GLOBAL during a change*. All the variables declared with VAR_GLOBAL up to the next pragma line or the end of the table form a data block accordingly.
For information on the general conditions for a download in RUN, see SIMOTION Basic Functions Function Manual.

### Behavior during STOP-RUN transition

The values of non-retentive global variables are retained by default during the transition from STOP to RUN mode.

You can, however, make a setting whereby the non-retentive global variables are initialized during the STOP-RUN transition:

- **As of Version V4.2 of the SIMOTION Kernel**, by activating the (Page 299)**Initialization of non-retentive global variables and program data during STOP-RUN transition** checkbox on the SIMOTION device.

  With non-retentive unit variables, this setting can be overwritten by a pragma or pragma line in the relevant data blocks of the program sources.

- **As of Version V4.1 of the SIMOTION Kernel**, by a pragma or pragma line in the relevant data blocks of the program sources (only with non-retentive unit variables):

  – With the **SIMOTION ST** programming language:

    Specify the following attribute within a pragma in the relevant VAR_GLOBAL/END_VAR declaration block: { BlockInit_OnDeviceRun := ALWAYS; }

  – With the **SIMOTION MCC** or **SIMOTION LAD/FBD** programming languages:

    Paste a pragma line with the following setting into the declaration table: "*Initialization during STOP-RUN transition* = **Always**". All the variables declared with VAR_GLOBAL up to the next pragma line or the end of the table form a data block which is initialized during the STOP-RUN transition.

**Note**

With SIMOTION devices up to SIMOTION Kernel Version V4.0, non-retentive global variables are never initialized during the STOP-RUN transition.

### 5.2.4.3 Initialization of local variables

Local variables are initialized:

- For the initialization of retentive unit variables (Page 229).
- For the initialization of non-retentive unit variables (Page 231).
- 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:<br>• Static variables (VAR) are initialized according to the memory area in which they are stored.<br>See: Initialization of static program variables (Page 234).<br>• 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:<br>• Static variables (VAR, VAR_IN, VAR_OUT) are only initialized when the FB instance is initialized.<br>See: Initialization of instances of function blocks (FBs) (Page 235).<br>• 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 286) 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 "Only create program instance data once" (Page 56) compiler option.

- For the deactivated "Only create program instance data 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 only initialized only during the transition from STOP to RUN operating state.

- For the activated "Only create program instance data once" compiler option:

  This setting is necessary, for example, if a program is to be called within a program.

  The static variables of all programs from the program source (unit) involved are only stored once in the user memory of the unit.

  They are thus initialized together with the non-retentive unit variables, see Initialization of non-retentive global variables (Page 231).

  They are not initialized by default during the transition from STOP to RUN operating state. You can, however, make a setting whereby they are initialized during the STOP-RUN transition:

  – **As of version V4.2 of the SIMOTION Kernel**, by activating the (Page 299) **Initialization of non-retentive global variables and program data during STOP-RUN transition** checkbox on the SIMOTION device.

    This setting can be overwritten by a pragma or pragma line in the data block of the relevant program organization unit (POU).

  – **As of version V4.1 of the SIMOTION Kernel**, by a pragma or pragma line in the data block of the relevant program organization unit (POU):

    With the **SIMOTION ST** programming language:
    Specify the following attribute within a pragma in the VAR/END_VAR declaration block: { BlockInit_OnDeviceRun := ALWAYS; }

    With the **SIMOTION MCC** or **SIMOTION LAD/FBD** programming languages:
    The declaration table starts with a pragma line containing the following setting: "*Initialization during STOP-RUN transition* = **Always**". All the variables declared with VAR in the table are initialized during the STOP-RUN transition.

#### 5.2.4.5 Initialization of instances of function blocks (FBs)

The initialization of a function block instance (Page 181) 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 231).

- Local declaration in a program (within VAR / END_VAR):

  Initialization as for static variables of programs, see Initialization of static variables of programs (Page 234).

- 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 286) 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 229) and Initialization of non-retentive global variables (Page 231).

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]:<br>• **Yes**[2]: All technology objects are initialized.<br>– All technology objects are restructured and all non-retentive system variables are initialized.<br>– All technological alarms are cleared.<br>• **No**[3]: Only technology objects changed in SIMOTION SCOUT are initialized.<br>– The technology objects in question are restructured and all non-retentive system variables are initialized.<br>– All alarms that are pending on the relevant technology objects are cleared.<br>– 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.<br>The retentive system variables of all other technology objects are retained (e.g. absolute encoder calibration). |

[1] Default 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 code of global variables and their initialization during download

| Data segment | | Description of version code |
|---|---|---|
| **Global device variables** | | |
| | Retentive global device variables<br>Non-retentive global device variables | • Separate version code for each data segment of the global device variables<br>• The version code of the data segment changes for:<br>– Add or remove a variable within the data segment<br>– Change of the identifier or the data type of a variable within the data segment<br>• This version code does not change on:<br>– Changes in the other data segment<br>– Changes to initialization values[1]<br>• During downloading[2], the rule is: This data segment is only initialized when the version code of a data segment is changed. |
| **Unit variables of a unit** | | |
| | Retentive unit variables in the interface section<br>Retentive unit variables in the implementation section<br>Non-retentive unit variables in the interface section<br>Non-retentive unit variables in the implementation section | • Several data blocks ( = declaration blocks)[3] in each data segment possible.<br>• **Own version code** for each data block.<br>• The version code of the data block changes for:<br>– Add or remove a variable in the associated declaration block<br>– Change of variable sequence in the relevant declaration block<br>– Change of the identifier or the data type of a variable in the associated declaration block<br>– Change of a data type definition (from a separate or imported[4] unit) used in the associated declaration block<br>– Add or remove declaration blocks within the same data segment **before** the associated declaration block<br>• This version code does not change on:<br>– Add or remove declaration blocks in other data segments<br>– Add or remove declaration blocks within the same data segment **after** the associated declaration block<br>– Changes in other data blocks<br>– Changes to initialization values[1]<br>– Changes to data type definitions that are not used in the associated data block<br>– Changes to functions<br>• During downloading[2], the rule is: This data block is only initialized when the version code of a data block is changed[5]<br>• Functions for data backup and initialization take into account the version code of the data blocks. |

[1] Changed initialization values are not effective until the data block or data segment in question is initialized.

[2] If *Initialization of retentive program data and retentive global device variables* = No and *Initialization of non-retentive program data and non-retentive global device variables* = No.
In the case of other settings: See the sections "Initialization of retentive global variables (Page 229)" and "Initialization of non-retentive global variables (Page 231)".

[3] With the **SIMOTION ST** programming language:
A data block corresponds to a VAR_GLOBAL/END_VAR or VAR_GLOBAL RETAIN/END_VAR declaration block in the interface section or implementation section.
With the **SIMOTION MCC** or **SIMOTION LAD/FBD** programming languages:
A data block of the non-retentive unit variables is formed as follows from the variables declared with VAR_GLOBAL or VAR_GLOBAL RETAIN in the interface section or implementation section of the declaration table: Pragma lines within a section of the declaration table separate the variables into different data blocks.

[4] The import of units depends on the programming language, refer to the associated section (Page 209).

[5] Initialization of a changed data block also occurs during a download in RUN, provided the following condition is fulfilled:
With the **SIMOTION ST**
programming language: The following attribute (Page 289) has been specified within a pragma (Page 294) in the relevant declaration block: { BlockInit_OnChange := TRUE; }.
With the **SIMOTION MCC** or **SIMOTION LAD/FBD** programming languages:
A pragma line has been pasted into the declaration table and the following check box is activated: *Initialization of VAR_GLOBAL during a change*. All the variables declared with VAR_GLOBAL up to the next pragma line or the end of the table form a data block accordingly.
For information on the general conditions for a download in RUN, see 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).

  Change this default as follows:

  – In the SIMOTION ST programming language:

    For each declaration block with the following pragma: `{ HMI_Export := FALSE; }`.

    See also Controlling compiler with attributes (Page 294).

  – In the SIMOTION MCC and SIMOTION LAD/FBD programming languages:

    For the variable declarations following a pragma line, if you deactivate the *VAR_GLOBAL for HMI devices* or *VAR_GLOBAL RETAIN for HMI devices* parameters in this pragma line.

  The unit variables of this kind of data block are not exported to HMI devices. The HMI consistency check is also omitted for them during the download.

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).

  Change this default as follows:

  – In the SIMOTION ST programming language:

    For each declaration block with the following pragma: `{ HMI_Export := TRUE; }`

    See also Controlling compiler with attributes (Page 294).

  – In the SIMOTION MCC and SIMOTION LAD/FBD programming languages:

    For the variable declarations following a pragma line, if you activate the *VAR_GLOBAL for HMI devices* or *VAR_GLOBAL RETAIN for HMI devices* parameters in this pragma line.

  The unit variables of this kind of data block are exported to HMI devices. Consequently, they are subject to the HMI consistency check during the download.

- 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 `{ HMI_Export := FALSE; } /{ HMI_Export := TRUE; }` pragma (or the *VAR_GLOBAL for HMI devices* or *VAR_GLOBAL RETAIN for HMI devices* settings in a pragma line) depends on the SIMOTION Kernel version:

- 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 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.

## Example for the SIMOTION ST programming language

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 access the corresponding I/O address directly.

  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.

  Additional information: Direct access and process image of the cyclic tasks (Page 245).

- 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 range of the SIMOTION device can be used.

  Direct access to this I/O variable is still possible: Specify direct access with *_direct.varname*.

  Additional information: Direct access and process image of the cyclic tasks (Page 245).

- 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.

  Additional information: Access to the fixed process image of the BackgroundTask (Page 256).

A comparison of the most important properties is contained in "Important properties of direct access and process image (Page 242)".

You can use I/O variables like any other variable, see "Access I/O variables (Page 269)".

**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 properties of direct access and process image access

| | Direct access | Access to process image of cyclic tasks | Access to fixed process image of the BackgroundTask |
|---|---|---|---|
| Permissible address range | Entire address range of the SIMOTION device<br>**Exception**: I/O variables comprising more than one byte must not contain addresses 63 and 64 contiguously (example: PIW63 or PQD62 are not permitted). | | 0 .. 63.<br>**Exception** Up to version V4.1 of the SIMOTION Kernel or the "Separate process image" setting:<br>Addresses used for the process image of the cyclic tasks. |
| Address configuration | Necessary. The addresses used must be present in the I/O and appropriately configured. | | Not necessary. Addresses that are not present in the I/O or have not been configured can also be used. |
| Assigned task | None. | Cyclic task for selection:<br>• SynchronousTasks,<br>• TimerInterruptTasks,<br>• BackgroundTask. | BackgroundTask. |
| Memory area for process images | - | Depends on the SIMOTON Kernel version:<br>• Up to version V4.1:<br>Separate memory areas in all cases<br>• As of version V4.2:<br>Option to select a common memory area | |

| | Direct access | Access to process image of cyclic tasks | Access to fixed process image of the BackgroundTask |
|---|---|---|---|
| Updating | • Onboard I/O of SIMOTION devices C230-2, C240, and C240 PN:<br>Update occurs in a cycle clock of 125 µs.<br>• I/O via isochronous PROFIBUS DP, PROFINET and DRIVE-CLiQ as well as Onboard I/O of SIMOTION D devices:<br>The update is performed in the position control cycle clock[1].<br>• I/O via isochronous PROFIBUS DP and PROFINET as well as P-Bus:<br>Update occurs in the interpolator cycle clock[1].<br>Inputs are read at the start of the cycle clock.<br>Outputs are written at the end of the cycle clock. | Update occurs with the assigned task:<br>• Inputs are read before the assigned task is started and transferred to the process input image.<br>• Process output image is written to the outputs after the assigned task has been completed. | An update is made with the *BackgroundTask*:<br>• Inputs are read before the *BackgroundTask* is started and is transferred to the process input image.<br>• Process output image is written to the outputs when the *BackgroundTask* is complete. |
| Consistency | – | During the entire cycle of the assigned task.<br>**Exception**: Direct access to output occurs. | During the entire cycle of the *BackgroundTask*.<br>**Exception**: Direct access to output occurs. |
| | Consistency is only ensured for elementary data types.<br>When using arrays, the user is responsible for ensuring data consistency. | | |
| Use | Preferred in MotionTasks | Preferred in the assigned task | Preferred in the BackgroundTask |
| Declaration as variable | Necessary, for the entire device as an I/O variable in the symbol browser.<br>Each byte of the address range may only be assigned to a single I/O variable.<br>Syntax of I/O address: e.g. PIW1022, PQ63.3. | | Possible, but not necessary:<br>• For the entire device as I/O variable in the symbol browser<br>• As unit variable<br>• As local static variable in a program |

| | Direct access | Access to process image of cyclic tasks | Access to fixed process image of the BackgroundTask |
|---|---|---|---|
| Download of new or changed I/O variables | Only possible in STOP operating state | | - |
| Use the absolute address | Not supported. | | Possible, with the following syntax: e.g. %IW62, %Q63.3. |
| Byte order when forming the process image | - | As supplied by the I/O | Depends on the SIMOTION Kernel version and the memory area setting for the process images:<br>• Up to version V4.1 or the "Separate process image" setting:<br>Always Big Endian<br>• As of version V4.2 and the "Common process image" setting:<br>As supplied by the I/O |
| Byte order during access | Depends on I/O | | Always Big Endian |
| Writeability of inputs | No | Depends on the SIMOTION Kernel version:<br>• Up to version V4.1:<br>No<br>• As of version V4.2:<br>Yes | Yes |
| Write protection for outputs | Possible; **Read only** status can be selected. | Not supported. | Not supported. |
| Declaration of arrays | Possible. | | Not supported. |
| Additional information | Direct access and process image of the cyclic tasks (Page 245). | | Access to the fixed process image of the BackgroundTask (Page 256). |
| Responses in the event of an error | Error during access from user program, alternative reactions available:<br>• CPU stop[2]<br>• Replacement value<br>• Last value<br>Please refer to the SIMOTION Basic Functions Direct function description. | Error during generation of process image, alternative reactions available:<br>• CPU stop[3]<br>• Replacement value<br>• Last value | Error during generation of process image, reaction: CPU stop[3].<br>**Exception**: If a direct access has been created at the same address, the behavior set there applies. |
| | | See SIMOTION Basic Functions Description of Functions. | |

| | Direct access | Access to process image of cyclic tasks | Access to fixed process image of the BackgroundTask |
|---|---|---|---|
| **Access** | | | |
| • In the RUN operating state | Without any restrictions. | Without any restrictions. | Without any restrictions. |
| • During the StartupTask | Possible with restrictions:<br>• Inputs can be read.<br>• Outputs are not written until StartupTask is complete. | Possible with restrictions:<br>• Inputs are read at the start of the StartupTask.<br>• Outputs are not written until StartupTask is complete. | Possible with restrictions:<br>• Inputs are read at the start of the StartupTask.<br>• Outputs are not written until StartupTask is complete. |
| • During the ShutdownTask | Without any restrictions. | Possible with restrictions:<br>• Inputs retain status of last update<br>• Outputs are no longer written. | Possible with restrictions:<br>• Inputs retain status of last update<br>• Outputs are no longer written. |

[1] The following SIMOTION devices are updated in the Servo_fast cycle or IPO_fast cycle, if the cycles are configured: D445-2 DP/PN, D455-2 DP/PN (as of version V4.2) and D435-2 DP/PN (as of version V4.3).

[2] Call the ExecutionFaultTask.

[3] 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 (Page 248) can be used.

A comparison of the most important properties, also in comparison to the fixed process image of the BackgroundTask (Page 256) is contained in "Important properties of direct access and process image (Page 242)".

**Note**

Observe the rules for I/O addresses for direct access and the process image of the cyclical tasks (Page 249).

It is particularly important that every address used in an I/O variable is available in the I/O and configured; each byte in the address range may be assigned to no more than one I/O variable (does not apply to access with data type BOOL).

The detailed status of I/O variables (Page 254) can be read as of Version V4.2 of the SIMOTION Kernel.

### Direct access

Direct access is used to access the corresponding I/O address directly. 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 250) without assigning it a task.

### 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 250) and assign it a task.

**Direct access** to this I/O variable is still possible: Specify direct access with *_direct.var-name*.

**Note**

An access via the process image is more efficient than direct access.

### Additional properties as of Version V4.2 of the SIMOTION Kernel

As of Version V4.2 of the SIMOTION Kernel, direct access to inputs/outputs and the process image of the cyclic tasks offers additional properties:

- As far as the process image of the cyclic tasks is concerned, a common memory area with the fixed process image of the BackgroundTask can be set (standard with newly created devices)
- As far as the process image of the cyclic tasks is concerned, I/O variables for inputs can be written to (i.e. they can be assigned values).
- A detailed status of the I/O variables can be read.

### Memory area with the fixed process image of the BackgroundTask

- **As of Version V4.2 of the SIMOTION Kernel**, selecting a "Common process image" setting on the device (Page 299) ensures the memory area for the fixed process image of the BackgroundTask is a subset of the memory area for the process image of the cyclic tasks.
- **With Version V4.1 and lower of the SIMOTION Kernel** or the "Separate process image" setting on the device (as of Version V4.2 of the SIMOTION Kernel), the fixed process image of the BackgroundTask and the process image of the cyclic tasks occupy different memory areas.

| NOTICE |
| --- |
| If (and only if) you are also using the fixed process image of the BackgroundTask, it is important to consider the effects of the "Common process image" or "Separate process image" settings on the fixed process image of the BackgroundTask (Page 256). |

Table 5- 29 Effect of "Common process image" or "Separate process image" settings on the process image of the cyclic tasks

| | Common process image | Separate process image |
| --- | --- | --- |
| Availability | Only available as of Version V4.2 of the SIMOTION Kernel:<br>• Setting available for selection<br>• Standard for newly created devices | Version V4.1 and lower of the SIMOTION Kernel applies:<br>• System characteristic, not configurable.<br>The following applies as of Version V4.2 of the SIMOTION Kernel:<br>• Setting available for selection<br>• Standard with device upgrades |
| Download of new or changed I/O variables | Only possible in STOP mode | Only possible in STOP mode |
| Byte order when forming the process image and during access | Depends on connected I/O | |
| Effects on the fixed process image of the BackgroundTask | See the relevant table in "Access to the fixed process image of the BackgroundTask (Page 256)". | |
| Additional information | Common process image (Page 259) | Separate process image (Page 262) |

### 5.3.3.1 Address range of the SIMOTION devices

The address range of the SIMOTION devices is specified in the following table according to the version of the SIMOTION Kernel concerned. The complete address range can be used for direct access and process image of the cyclical tasks.

Table 5- 30 Address range of the SIMOTION devices according to the version of the SIMOTION Kernel

| SIMOTION device | Address range for SIMOTION Kernel version | | | | |
|---|---|---|---|---|---|
| | V3.2 | V4.0 | V4.1 | V4.2 | V4.3 |
| C230-2 | 0 .. 2047 [3] | 0 .. 2047 [3] | 0 .. 2047 [3] | – | – |
| C240 | – | 0 .. 4095 [3] | 0 .. 4095 [3] | 0 .. 4095 [3] | 0 .. 4095 [3] |
| C240 PN [1] | – | – | 0 .. 4095 [4] | 0 .. 4095 [4] | 0 .. 4095 [4] |
| D410 DP | – | – | 0 .. 8191 [3] | 0 .. 8191 [3] | 0 .. 8191 [3] |
| D410 PN | – | – | 0 .. 8191 [4] | 0 .. 8191 [4] | 0 .. 8191 [4] |
| D410-2 | – | – | – | – | 0 .. 8191 [3 4] |
| D425 | 0 .. 4095 [3] | 0 .. 16383 [3 4] | 0 .. 16383 [3 4] | 0 .. 16383 [3 4] | 0 .. 16383 [3 4] |
| D425-2 | – | – | – | – | 0 .. 16383 [3 4] |
| D435 | 0 .. 4095 [3] | 0 .. 16383 [3 4] | 0 .. 16383 [3 4] | 0 .. 16383 [3 4] | 0 .. 16383 [3 4] |
| D435-2 | – | – | – | – | 0 .. 16383 [3 4] |
| D445 | 0 .. 4095 [3] | 0 .. 16383 [3 4] | 0 .. 16383 [3 4] | 0 .. 16383 [3 4] | 0 .. 16383 [3 4] |
| D445-1 [1] | – | – | 0 .. 16383 [3 4] | 0 .. 16383 [3 4] | 0 .. 16383 [3 4] |
| D445-2 | – | – | – | 0 .. 16383 [3 4] | 0 .. 16383 [3 4] |
| D455-2 | – | – | – | 0 .. 16383 [3 4] | 0 .. 16383 [3 4] |
| P320 [2] | – | – | 0 .. 4095 [4] | 0 .. 4095 [4] | 0 .. 4095 [4] |
| P350 | 0 .. 2047 [3] | 0 .. 4095 [3] | 0 .. 4095 [3] | 0 .. 4095 [3] | 0 .. 4095 [3] |

1 Available as of V4.1 SP2 HF4

2 Available as of V4.1 SP5

3 For distributed I/O (over PROFIBUS DP), the transmission volume is restricted to 1024 bytes per PROFIBUS DP line.

4 For distributed I/O (over PROFINET), the transmission volume is restricted to 4,096 bytes per PROFINET segment.

### 5.3.3.2 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 245). 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 range 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.

     Please be aware of the following, for example, if you wish to use an I/O variable to read the PROFIdrive telegram transferred to or from the 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 256). These accesses are not taken into account during the consistency check of the project (e.g. during download).

### 5.3.3.3 Creating I/O variables for direct access or process image of cyclic tasks

Create I/O variables for direct access or a process image of the cyclic tasks in the address list of the detail view.

This is only possible in offline mode.

Here is a brief overview of the procedure:

1. Select the "Address list" tab in the detail view and choose the SIMOTION device

   or

   In the project navigator of SIMOTION SCOUT, double-click the "ADDRESS LIST" element in the SIMOTION device subtree.

2. Select the line before which you want to paste the I/O variable and, from the context menu, select **Insert new line**

   or

   Scroll to the end of the table of variables (empty line).

3. In the empty row of the table, enter or select the following:
   - Names of the **I/O variables**
   - **I/O address**

     Select the "IN" or "OUT" entries if you wish to assign symbols to the I/O variable (input or output) (As of Version V4.2 of the SIMOTION Kernel. The symbolic assignment must be activated, menu **Project > Use symbolic assignment**.).

     Or enter a fixed address according to "Syntax for entering I/O addresses (Page 253)".
   - Optional for outputs:

     Activate the **Read only** check box 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 254)".

4. Optionally, you can also enter or select the following (not for data type BOOL):
   - **Field length** (array size).
   - **Process image** or direct access:

     Can only be assigned if the **Read only** check box is deactivated.

     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 behavior in the event of an error, see SIMOTION Basic Functions Function Manual.
   - **Display format** (if array, for each element), when you monitor the variable in the address list
   - **Replacement value** (if array, for each element).
5. Only if you have selected "IN" or "OUT" as the I/O address (symbolic assignment).
   - In the **Assignment** column, click on the **[…]** button.

     A window opens displaying the possible assignment targets of the SIMOTION device and, if necessary, of SINAMICS Integrated. Only those assignment targets are displayed that match the data direction (input/output) and data type.
   - Select the assignment target.

     The **Assignment status** column indicates whether the assignment was successful or not.

   For details regarding symbolic assignment, refer to the SIMOTION Basic Functions Function Manual.

You can now access this variable using the address list or any program of the SIMOTION device.

Details on how to manage the address list can be found in the online help.

**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 269)".

### 5.3.3.4 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 245), 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- 31 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:<br>x: | 0 .. *MaxAddr*<br>0 .. 7 | | -[1] | n:<br>x: | 0 .. 16383<br>0 .. 7 |
| BYTE | PIBn | PQBn | n: | 0 .. *MaxAddr* | n: | 0 .. *MaxAddr* | n: | 0 .. 16383 |
| WORD | PIWn | PQWn | n: | 0 .. 62<br>64 .. *MaxAddr* - 1 | n: | 0 .. 62<br>64 .. *MaxAddr* - 1 | n: | 0 .. 62<br>64 .. 16382 |
| DWORD | PIDn | PQDn | n: | 0 .. 60<br>64 .. *MaxAddr* - 3 | n: | 0 .. 60<br>64 .. *MaxAddr* - 3 | n: | 0 .. 60<br>64 .. 16380 |
| n = logical address<br>x = bit number | | | | | | | | |
| *MaxAddr* = | Maximum I/O address of the SIMOTION device, depending on the version of SIMOTION Kernel, see address range of the SIMOTION devices in "Direct access and process image of the cyclic tasks (Page 245)". | | | | | | | |
| [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 249).

### 5.3.3.5 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 245). 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- 32 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 253)".

### 5.3.3.6 Detailed status of the I/O variables (as of Kernel V4.2)

As of Version V4.2 of the SIMOTION Kernel, the status of an I/O variable can be queried using *_quality.var-name*. It is supplied as an OR logic operation of the following status values in the DWORD data type and can be assigned to an appropriate variable, for example. The value 16#0000_0000 indicates the connected I/O are operating without error.

The same value is supplied for every I/O variable within an address range (slot or subslot) configured in HW Config.

Table 5- 33 Meanings of status values of I/O variables

| Value (DWORD) | Bit x = 1 | Meaning |
|---|---|---|
| 16#0000_0000 | - | No error occurred. |
| 16#0000_0001 | 0 | **Maintenance required**<br>The connected module signals that it requires maintenance. The component needs to be checked within a foreseeable period (e.g. the printer cartridge must be changed within a period of several days). |
| 16#0000_0002 | 1 | **Maintenance demanded**<br>The connected module demands maintenance. The component needs to be checked soon (e.g. the printer cartridge must be changed immediately). |
| 16#0000_0004 | 2 | **Warning pending** (drive warning, TM17 warning, etc.)<br>The connected module has signaled a warning. This has been entered in the diagnostics buffer. The precise cause can be determined from the documentation for the relevant module. |
| 16#0000_0008 | 3 | **Fault pending** (diagnostic interrupt, drive fault, TM17 fault, etc.)<br>The connected module has signaled an error. This has been entered in the diagnostics buffer. The precise cause can be determined from the documentation for the relevant module. |

| Value (DWORD) | Bit x = 1 | Meaning |
|---|---|---|
| 16#0000_0010 | 4 | **This parameter assignment does not match the parameter assignment being compared.**<br>A difference was detected when this parameter assignment was compared with the parameter assignment of the connected module. As such, the required functionality cannot be guaranteed.<br>Remedy: Save the project and compile changes, reload both application and counterpart. |
| 16#0000_0020 | 5 | **Application and counterpart are not isochronous** (error involving the dynamic life-sign).<br>Certain message frames (axis, synchronous operation, output cam, measuring input message frames) are synchronized by exchanging cyclic life-signs. Errors are detected when the cyclic life-sign is checked. This invalidates the data in the message frame.<br>Remedy: Await synchronization, check the parameter assignment (e.g. does the master application cycle set on the device in HW Config match the position control cycle clock), save the project and compile changes, reload both application and counterpart. |
| 16#0000_0040 | 6 | **I/O cannot be used synchronously in all cycles.**<br>A fast application cycle (Servo_fast) and a slow application cycle (Servo) are running asynchronously in relation to one another. The I/O can only be used synchronously in the cycles associated with the bus cycle. Access from other cycles is asynchronous and inconsistent.<br>Remedy: Call the _synchroniseDpInterfaces() function. |
| 16#0000_0080 | 7 | **I/O cannot be used synchronously**<br>The SIMOTION control is the sync slave on a bus. The bus connection is running synchronously in relation to the sync master, but is not yet running synchronously in relation to the application cycles of the SIMOTION control. Access to the I/O is asynchronous and inconsistent.<br>Remedy: Call the _synchroniseDpInterfaces() function. |
| 16#0000_0100 | 8 | **Bus connection (sync slave) is not isochronous in relation to the sync master.**<br>The SIMOTION control is the sync slave on a bus and has not yet synchronized its bus connection with the sync master.<br>The isochronous I/O on this bus cannot be used yet.<br>Remedy: Switch on/connect the sync master. |
| 16#0000_0200 | 9 | **DP station is deactivated.**<br>The partner module has been deactivated.<br>Remedy: Activate the partner module (_activateDpSlave() function). |
| 16#0000_0400 | 10 | **The partner of the inputs (e.g. I-device, I-slave) is in STOP.**<br>The connected module is in STOP mode and not sending any new data as a result.<br>Remedy: Switch the connected module to RUN. |
| 16#0000_0800 | 11 | **PROFINET: Failure detected by submodule** (e.g. channel error)<br>The connection to the connected device is OK. The error must be searched for in the connected device.<br>Troubleshooting: Diagnostics buffer, device diagnostics with HW Config |
| 16#0000_1000 | 12 | **PROFINET: Failure detected by module** (e.g. submodule failed, removed, etc.)<br>The connection to the connected device is OK. The error must be searched for in the connected device.<br>Troubleshooting: Diagnostics buffer, device diagnostics with HW Config |
| 16#0000_2000 | 13 | **PROFINET: Failure detected by device** (e.g. device in STOP, module removed, etc.)<br>The connection to the connected device is OK. The error must be searched for in the connected device.<br>Troubleshooting: Diagnostics buffer, device diagnostics with HW Config |

| Value (DWORD) | Bit x = 1 | Meaning |
|---|---|---|
| 16#0000_4000 | 14 | **PROFINET: Failure detected by controller** (e.g. not connected, etc.)<br>There is no connection to a partner on PROFINET.<br>Possible causes: Partner is switched off, cable pulled out, incorrect parameter assignment for connection<br>Troubleshooting: Best to use the PROFINET topology editor in HW Config. |
| 16#0000_8000 | 15 | **Slot/subslot is not connected** (disconnection alarm).<br>The connection to the connected device is OK. The error must be searched for in the connected device (e.g. module/submodule removed).<br>Troubleshooting: Diagnostics buffer, device diagnostics with HW Config |
| 16#0001_0000 | 16 | **Device is not connected** (station failure).<br>There is no connection to a partner.<br>Possible causes: Partner is switched off, cable pulled out. |
| 16#0002_0000 | 17 | **Replacement value behavior during access**<br>There is no connection to the counterpart (sum signal from bits 9 to 16), i.e. there is no valid input data or the output data is not reaching the terminal. The replacement value behavior set (replacement value, last value) takes effect during direct access to this address or during process image updates. |
| 16#4000_0000 | 30 | **Diagnostics address only**<br>No cyclic I/O data is configured for this address. It is possible, however, to query submodule diagnostic information. |
| 16#8000_0000 | 31 | **Address gap**<br>There is no hardware configured for this logical address. |

### 5.3.4 Access to fixed process image of the BackgroundTask

The fixed 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. Preferably, it should be 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).

**Note**

The fixed process image of the BackgroundTask can be used to access addresses that are not available in the I/O or not configured in HW Config. These are treated like normal memory addresses.

## Memory area

- **As of Version V4.2 of the SIMOTION Kernel**, selecting a "Common process image" setting on the device (Page 299) ensures the memory area for the fixed process image of the BackgroundTask is a subset of the memory area for the process image of the cyclic tasks.

  I/O addresses can be read and written to using both the fixed process image of the BackgroundTask and the process image of the cyclic tasks.

- **With Version V4.1 and lower of the SIMOTION Kernel** or the "Separate process image" setting on the device (as of Version V4.2 of the SIMOTION Kernel), the fixed process image of the BackgroundTask and the process image of the cyclic tasks occupy different memory areas.

  I/O addresses accessed using the process image of the cyclic tasks cannot be read or written to using the fixed process image of the BackgroundTask. They are treated like normal memory addresses.

Table 5- 34 Effect of "Common process image" or "Separate process image" settings on the fixed process image of the BackgroundTask

| | Common process image | Separate process image |
|---|---|---|
| Availability | Only available as of Version V4.2 of the SIMOTION Kernel:<br>• Setting available for selection<br>• Standard for newly created devices | Version V4.1 and lower of the SIMOTION Kernel applies:<br>• System characteristic, not configurable.<br>The following applies as of Version V4.2 of the SIMOTION Kernel:<br>• Setting available for selection<br>• Standard with device upgrades |
| Memory area | Subset of the memory area for the process image of the cyclic tasks | Separate memory area for the process image of the cyclic tasks |
| Using I/O addresses accessed using the process image of the cyclic tasks | Possible.<br>Updates use the configured cyclic tasks. | Not supported.<br>The addresses are treated like normal memory addresses. |
| Byte order when forming the process image | As supplied by the I/O | Always Big Endian |
| Byte order when accessing the process image | Always Big Endian | Always Big Endian |
| Access to I/O operating in the Little Endian byte order | Same result as during direct access or for the process image of cyclic tasks (apart from WORD or DWORD data types). | Results differ depending on the I/O variables created for direct access. |
| Effects on the process image of the cyclic tasks | See the relevant table in "Direct access and process image of the cyclic tasks (Page 245)". | |
| Additional information | Common process image (Page 259) | Separate process image (Page 262) |

For information on the order of the Little Endian and Big Endian bytes, please refer to the SIMOTION Basic Functions Function Manual.

A comparison of the most important properties in comparison to the direct access and process image of the cyclical tasks (Page 245) is contained in "Important properties of direct access and process image (Page 242)".

**Note**

The rules for I/O addresses for direct access and the process image of the cyclical tasks (Page 249) do **not** apply. Access to the fixed process image of the BackgroundTask is 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 264): The absolute PI access identifier contains the address of the input/output and the data type.
- Using a symbolic PI access (Page 266): 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 268): In the symbol browser, you define a valid I/O variable for the entire device that references the corresponding absolute PI access.

#### 5.3.4.1 Common process image (as of Kernel V4.2)

As of Version V4.2 of the SIMOTION Kernel, the "Common process image" setting (Page 299) can be selected on the SIMOTION device. This means addresses 0 .. 63 of the process image of the cyclic tasks and the fixed process image of the BackgroundTask occupy the same memory area.

This is the default for SIMOTION devices newly created in the project as of Version V4.2.

### Property of the common process image

1. The memory area for the fixed process image of the BackgroundTask (Page 256) is a subset of the memory area for the process image of the cyclic tasks (Page 245).
2. This means I/O addresses already accessed using the process image of the cyclic tasks may also continue to be used for the fixed process image of the BackgroundTask. Updates, however, use the configured cyclic tasks.
3. The following applies when forming the fixed process image of the BackgroundTask:

   The byte order is the same as supplied by the I/O:

   – Big Endian, e.g. for I/O via PROFIBUS DP, PROFINET, P-Bus, DRIVE-CLiQ

   – Little Endian, e.g. for onboard I/O of C240, C240 PN SIMOTION devices

   Any I/O variable created for the relevant addresses for the purpose of direct access or the process image of the cyclic tasks has no effect on the byte order.
4. Access to the fixed process image of the BackgroundTask always takes place using the Big Endian byte order.
5. These last two properties (nos. 3 and 4) affect access to inputs and outputs operating with the Little Endian byte order (e.g. onboard I/O of C240, C240 PN SIMOTION devices).

   If the fixed process image of the BackgroundTask is used for access, this leads to the following behavior, regardless of whether I/O variables have been created for the relevant addresses for the purpose of direct access or the process image of the cyclic tasks:

   – Access to individual bytes always supplies the same result via an I/O variable or the fixed process image of the BackgroundTask.

   – With the fixed process image of the BackgroundTask, bytes only change places if data type WORD is used for access.

   Please also refer to the example below.

For information on the order of the Little Endian and Big Endian bytes, please refer to the SIMOTION Basic Functions Function Manual.

### Example for common process image: Access to I/O operating with the Little Endian byte order

The digital inputs of the C240 SIMOTION device operate with the Little Endian byte order and occupy addresses 66 (bits 0 ..7) and 67 (bits 0.. 3) by default. The start address is changed to 60 in HW Config to ensure it is in the range occupied by the fixed process image of the BackgroundTask. Addresses 60 and 61 are now accessed using various I/O variables and the process image of the BackgroundTask.

The following three scenarios are considered, which differ in terms of whether and which I/O variables are created for direct access or the process image of the cyclic tasks:

1. Scenario A:

   **No I/O variables** are created for addresses 60 and 61.

2. Scenario B:

   **Two I/O variables** with data type BYTE are created for addresses 60 and 61: io_byte_60 (PIB60) and io_byte_61 (PIB61).

3. Scenario C:

   For adresss 60, **one I/O-Variable** with data type WORD is created; this also covers address 61: io_word_60 (PIW60).

Two additional I/O variables are also created in each of the three scenarios, making it possible to access bit 3: io_bit_60_3 (PI60.3) and io_bit_61_3 (PI61.3).

The table below lists which values are generated with the following access types:

- Direct access or access to the process image of the cyclic tasks:
  - Access to individual bytes or the word using the relevant I/O variables
  - Access to each individual byte using the _getInOutByte function (direct access only).
  - Access to the respective bit 3 using the relevant I/O variables
- Access to the fixed process image of the BackgroundTask:
  - Access to individual bytes using an absolute name
  - Access to the word using an absolute name
  - Access to the respective bit 3 using an absolute name

Table 5- 35 "Common process image" setting (as of Kernel V4.2): Different types of access to the process images of an input operating with the Little Endian byte order

| | Access using | Scenario A [1] | Scenario B [1] | Scenario C [1] |
|---|---|---|---|---|
| Direct access or access to the process image of the cyclic tasks | io_byte_60 (PIB60) | - | 16#08 | - |
| | io_byte_61 (PIB61) | - | 16#00 | - |
| | io_word_60 (PIW60) | - | - | 16#0008 |
| | _getInOutByte (IN, 60) | 16#08 | 16#08 | 16#08 |
| | _getInOutByte (IN, 61) | 16#00 | 16#00 | 16#00 |
| | io_bit_60_3 (PI60.3) | TRUE | TRUE | TRUE |
| | io_bit_61_3 (PI61.3) | FALSE | FALSE | FALSE |
| Access to the fixed process image of the BackgroundTask | %IB60 | 16#08 | 16#08 | 16#08 |
| | %IB61 | 16#00 | 16#00 | 16#00 |
| | %IW60 | 16#0800 [2] | 16#0800 [2] | 16#0800 [2] |
| | %I60.3 | TRUE | TRUE | TRUE |
| | %I61.3 | FALSE | FALSE | FALSE |

[1] Scenarios A, B, or C determine whether and which I/O variables are created for direct access or the process image of the cyclic tasks; see the explanation provided in the body of the document.

[2] The two bytes in the word change places, as a value saved in the Little Endian byte order is being read using Big Endian.

### 5.3.4.2 Separate process image (up to Kernel V4.1)

With Version V4.1 and below of the SIMOTION Kernel, the process image of the cyclic task and the fixed process image of the BackgroundTask are stored in different memory areas (separate process image).

As of Version V4.2 of the SIMOTION Kernel, the "Separate process image" setting (Page 299) can be selected on the SIMOTION device. This setting ensures there is compatibility with earlier Kernel versions.

It is the default for SIMOTION devices upgraded to Version V4.2 or higher.

#### Property of the separate process image

1. The fixed process image of the BackgroundTask (Page 256) and the process image of the cyclic tasks (Page 245) are stored in different memory areas.
2. This means I/O addresses that are already accessed using the process image of the cyclic tasks cannot be read or written to using the fixed process image of the BackgroundTask. They are treated like normal memory addresses.
3. I/O variables for direct access influence the fixed process image of the BackgroundTask:
   - The fixed process image of the BackgroundTask is always formed for the relevant addresses in the Big Endian byte order.
4. Access to the fixed process image of the BackgroundTask always takes place using the Big Endian byte order.
5. These last two properties (nos. 3 and 4) affect access to inputs and outputs operating with the Little Endian byte order (e.g. onboard I/O of C230-2, C240, C240 PN SIMOTION devices ).

   If an I/O variable is created for the relevant addresses for the purpose of direct access using data type WORD and access takes place using the fixed process image of the BackgroundTask, this leads to the following behavior:

   - Access with the data type WORD supplies the same result via the I/O variable and the fixed process image of the BackgroundTask.
   - Access to individual bytes using 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.

   Please also refer to the example below.

For information on the order of the Little Endian and Big Endian bytes, please refer to the SIMOTION Basic Functions Function Manual.

### Example for separate process image: Access to I/O operating with the Little Endian byte order

The digital inputs of the C240 SIMOTION device operate with the Little Endian byte order and occupy addresses 66 (bits 0 ..7) and 67 (bits 0.. 3) by default. The start address is changed to 60 in HW Config to ensure it is in the range occupied by the fixed process image of the BackgroundTask. Addresses 60 and 61 are now accessed using various I/O variables and the process image of the BackgroundTask.

The following three scenarios are considered, which differ in terms of whether and which I/O variables are created for direct access:

1. Scenario A:

   **No I/O variables** are created for addresses 60 and 61.

2. Scenario B:

   **Two I/O variables** with data type BYTE are created for addresses 60 and 61: io_byte_60 (PIB60) and io_byte_61 (PIB61).

3. Scenario C:

   For adresss 60, **one I/O-Variable** with data type WORD is created; this also covers address 61: io_word_60 (PIW60).

Two additional I/O variables are also created in each of the three scenarios, making it possible to access bit 3: io_bit_60_3 (PI60.3) and io_bit_61_3 (PI61.3).

The table below lists which values are generated with the following access types:

- Direct access:
  - Access to individual bytes or the word using the relevant I/O variables
  - Access to each individual byte using the _getInOutByte (Page 245) function.
  - Access to the respective bit 3 using the relevant I/O variables
- Access to the fixed process image of the BackgroundTask:
  - Access to individual bytes using an absolute name
  - Access to the word using an absolute name
  - Access to the respective bit 3 using an absolute name

Table 5- 36 "Separate process image" setting or Kernel up to Version V4.1: Different types of access to the process images of an input operating with the Little Endian byte order

| | Access using | Scenario A [1] | Scenario B [1] | Scenario C [1] |
|---|---|---|---|---|
| Direct access | io_byte_60 (PIB60) | - | 16#08 | - |
| | io_byte_61 (PIB61) | - | 16#00 | - |
| | io_word_60 (PIW60) | - | - | 16#0008 |
| | _getInOutByte (IN, 60) | 16#08 | 16#08 | 16#08 |
| | _getInOutByte (IN, 61) | 16#00 | 16#00 | 16#00 |
| | io_bit_60_3 (PI60.3) | TRUE | TRUE | TRUE |
| | io_bit_61_3 (PI61.3) | FALSE | FALSE | FALSE |
| Access to the fixed process image of the BackgroundTask | %IB60 | 16#08 | 16#08 | **16#00** [3] |
| | %IB61 | 16#00 | 16#00 | **16#08** [3] |
| | %IW60 | **16#0800** [2] | **16#0800** [2] | 16#0800 |
| | %I60.3 | TRUE | TRUE | **FALSE** [3] |
| | %I61.3 | FALSE | FALSE | **TRUE** [3] |

[1] Scenarios A, B, or C determine whether and which I/O variables are created for direct access; see the explanation provided in the body of the document.

[2] The two bytes in the word change places, as a value saved in the Little Endian order is being read using Big Endian.

[3] The two adjacent bytes change places, as the relevant word is saved in the Big Endian order.

### 5.3.4.3 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 256) by directly using the identifier for the address (with implicit data type). The syntax of the identifier (Page 265) 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 265).

**Note**

Outputs can be read and written to, but inputs can only be read.

### 5.3.4.4 Syntax for the identifier for an absolute process image access

For the absolute access to the fixed process image of the BackgroundTask (Page 264), 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 266).
- For the creation of an I/O variables for accessing the fixed process image of the BackgroundTask (Page 268).

Table 5- 37 Syntax for the identifier for an absolute process image access

| Data type | Syntax for | | Permissible address range | |
|---|---|---|---|---|
| | Input | Output | | |
| BOOL | %In.x<br>or<br>%IXn.x[1] | %Qn.x<br>or<br>%QXn.x[1] | n:<br>x : | 0 .. 63 [2]<br>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<br>x = bit number | | | | |
| [1] The syntax %IXn.x or %QXn.x is not permitted when defining I/O variables.<br>[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**

With Version V4.1 and below of the SIMOTION Kernel or the "Separate process image" setting on the device (as of Version V4.2 of the SIMOTION Kernel), the following applies:

- Addresses accessed using the process image of the cyclic tasks cannot be read or written to using the fixed process image of the BackgroundTask.

This restriction no longer applies as of Version V4.2 of the SIMOTION Kernel or with the "Common process image" setting on the device.

**Note**

The rules for I/O addresses for direct access and the process image of the cyclical tasks (Page 249) do **not** apply. Access to the fixed process image of the BackgroundTask is 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- 38 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
```

### 5.3.4.5 Symbolic access to the fixed process image of the BackgroundTask (symbolic PI access)

You can access the fixed process image of the BackgroundTask (Page 256) 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:

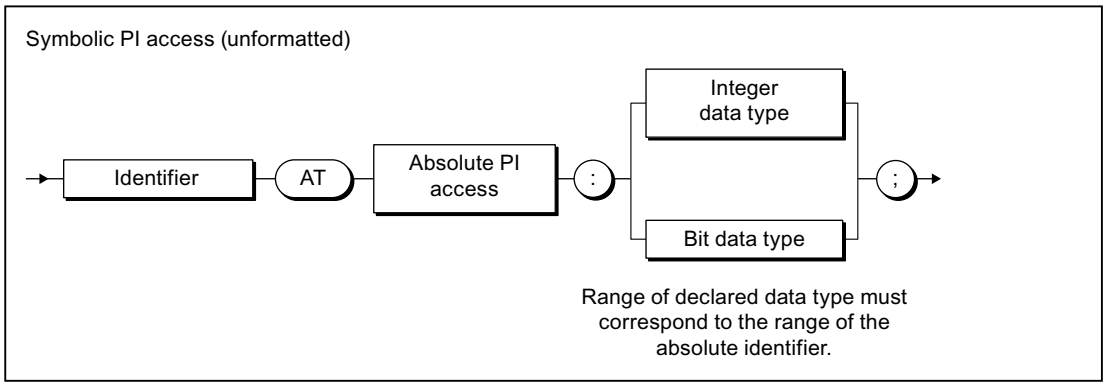



Figure 5-6 Declaration of a symbolic access to the process image

For the absolute PI access, see "Syntax for the identifier for an absolute PI access (Page 265)".

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 267)". 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 267).

### 5.3.4.6 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 256). 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 266).
- For the creation of an I/O variable (Page 268).

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- 39 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 265)".

### 5.3.4.7 Example of symbolic PI access

If, for example, you want to access the CPU memory area (absolute PI access (Page 265)) %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.

### 5.3.4.8 Creating an I/O variable for access to the fixed process image of the BackgroundTask

You create I/O variables for access to the fixed process image for the background task in the symbol browser in the detail view; you must be in offline mode to do this.

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 265)"
     (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 267)".
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 269)".

## 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 245).
- Access to the fixed process image of the BackgroundTask (Page 256).

You can use this I/O variable just like any other variable.

| NOTICE |
| --- |
| Consistency is only ensured for elementary data types.<br>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 name space, see the "Predefined name spaces" table in "Name spaces").

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!

There is also the option of having a library make programs available, which can be called from other programs or function blocks. Please refer to the conditions which apply when calling a "program in a program" (Page 188). In each case, the static data for the program called is stored once in the user memory of the device on which the library program is called. The same program instance data is used every time the program is called on the same device. A library program cannot be assigned to the execution system.

## 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- 40 Illegal ST commands and variable access in libraries

| |
|---|
| **Prohibited commands:**<br>• *_getTaskId* function (see *SIMOTION Basic Functions* Function Manual).<br>• *_getAlarmId* function (see *SIMOTION Basic Functions* Function Manual).<br>• *_checkEqualTask* function (see *SIMOTION Basic Functions* Function Manual).<br>• If the library is not device-dependent (i.e. compiled without reference to a SIMOTION device or SIMOTION Kernel version):<br>– System functions of SIMOTION devices (see the Parameter Manual for SIMOTION devices)<br>– Version-dependent system functions |
| **Prohibited variable accesses:**<br>• Unit variables (retentive and non-retentive)<br>• Global device variables (retentive and non-retentive)<br>• I/O variables<br>• Instances of the technology objects and their system variables<br>• Variables of task names and configured messages (*_task* and *_alarm* namespaces, see Namespaces (Page 278), *Predefined namespaces* table)<br>• If the library is not device-dependent (i.e. compiled without reference to a SIMOTION device or SIMOTION Kernel version):<br>– System variables of SIMOTION devices (see the Parameter Manual for SIMOTION devices)<br>– 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:<br>1. For the imported library, at least the same devices and SIMOTION kernel versions must be selected as for the importing library.<br>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).<br>2. The imported library must already be compiled individually with reference to all configured devices, kernel versions and technology packages.<br>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:<br>1. The system automatically identifies dependencies between libraries and selects the appropriate compilation sequence.<br>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.<br>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.

The SIMOTION online help provides additional information on know-how protection.

**Note**

If you export in XML format, the libraries or sources are exported in an encrypted form. When importing the encrypted XML files, the know-how protection, including login and password, is retained.

### 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 (Page 278).

- 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- 41 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 POU, 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. |

### Identifier 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 must not be identical to the following identifiers either:

- 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 (POUs) 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

Exported identifiers of all units (unit variables, data types, and POUs) must be unique throughout the device.

All the following identifiers must be unique within a unit:

- Unit variables (declared in the interface or implementation section)
- Data types (declared in the interface or implementation section)
- Program organization units (POUs)

They 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 (POUs) 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.

They must not be identical to the following identifiers either:

- 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- 42 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 275). In order to still access these hidden identifiers, you can use namespaces in certain cases.

### User-defined namespace

In the import statement for libraries and technology packages, you can define namespaces in order to access 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- 43 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- 44 Predefined namespaces

| Namespace | 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 cyclic tasks (Page 245).<br>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.<br>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). |
| _quality | As of Version V4.2 of the SIMOTION Kernel: For the detailed status of I/O variables (Page 254). A value with data type DWORD is supplied.<br>Local namespace for _device. Nesting as in _device._quality.*name* is permitted. |
| _to | For technology objects configured on the SIMOTION device, and their system variables and configuration data.<br>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 |

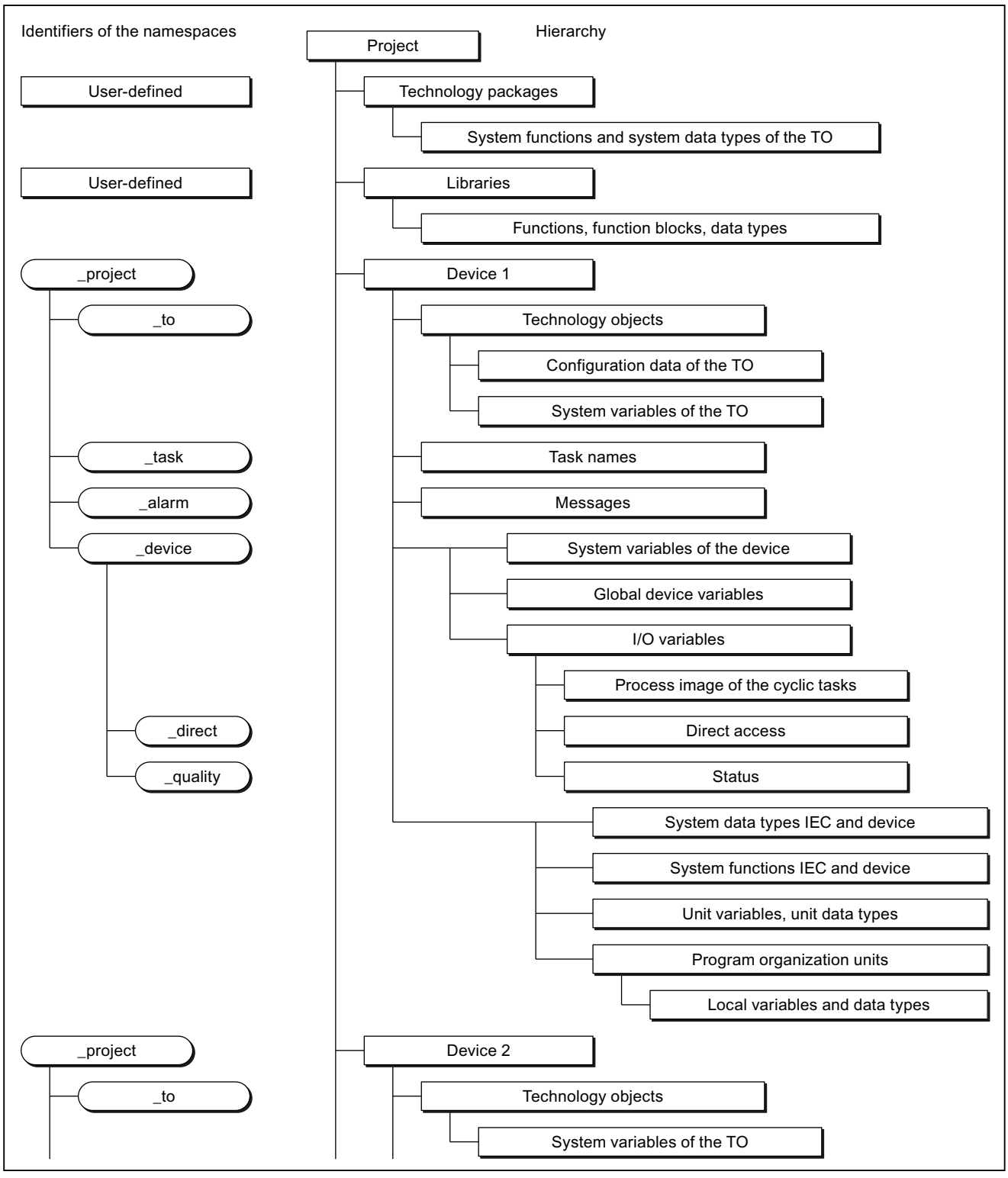



Figure 5-7 Namespaces and identifier hierarchy

# 5.6 Reference data

The reference data provide you with an overview of:

- on utilized identifiers with information about their declaration and use (Cross-reference list (Page 282)).
- on function calls and their nesting (Program structure (Page 286))
- on the memory requirement for various data areas of the program sources (Code attributes (Page 288))

## 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 types in declarations
- Used as variables in the statement section of a program organization unit.

You can generate the cross-reference list 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)

### 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.

**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.

### 5.6.1.2 Content of the cross-reference list

The cross-reference list contains all the identifiers assigned to the element selected in the project navigator. The applications for the identifiers are also listed in a table:

Details of how to work with the cross-reference list are provided in the section titled "Working with the cross-reference list (Page 285)".

Table 5- 45 Meanings of columns and selected entries in the cross-reference list

| Column | Entry in column | Meaning |
|---|---|---|
| **Name** | | Identifier name |
| **Type** | | Identifier type |
| | *Name* | • Data type of a variable (e.g. REAL, INT)<br>• POU type (e.g. PROGRAM, FUNCTION) |
| | DERIVED | Derived data type |
| | DERIVED ANY_OBJECT | TO data type |
| | ARRAY … | ARRAY data type |
| | ENUM … | Enumerator data type |
| | STRUCT … | STRUCT data type |
| **Declaration** | | Location of declaration |
| | *Name* (unit) | Declaration in the program source *name* |
| | *Name* (LIB) | Declaration in the library *name* |
| | *Name* (TO) | System variable of the technology object *name* |
| | *Name* (TP) | Declaration in the default library specified:<br>• Technology package *name*<br>• std_fct = IEC library<br>• device = device-specific library |
| | *Name* (DV) | Declaration on the SIMOTION device *name* (e.g. I/O variable or global device variable) |
| | _project | Declaration in the project (e.g. technology object) |
| | _device | Internal variable on the SIMOTION device (e.g. TaskStartInfo) |
| | _task | Task in the execution system |

| Column | Entry in column | Meaning |
|---|---|---|
| **Use** | | Use of identifier |
| | CALL | Call as subprogram |
| | ENUM *name* | As element when declaring the enumerator data type *name* |
| | I/O | Declaration as I/O variable |
| | R | Read access |
| | R (TYPE) | As data type in a declaration |
| | R/W | Read and write access |
| | STRUCT *name* | As component when declaring the structure *name* |
| | TYPE | Declaration as data type or POU |
| | *Variable type* (e.g. VAR, VAR_GLOBAL) | Declaration as variable of the variable type specified |
| | W | Write access |
| **Path specification** | | Path specification for the SIMOTION device or program source |
| | Name | SIMOTION device *name* |
| | *Name1*\|*Name2* | • Program source *name2* on SIMOTION device *name1*<br>• Program source *name2* in library *name1* |
| | *Name*\taskbind.hid | Execution system of the SIMOTION device *name* |
| **Range** | | Range within the SIMOTION device or program source |
| | IMPLEMENTATION | Implementation section of the program source |
| | INTERFACE | Interface section of the program source |
| | *POU type name* (e.g. FUNCTION *name*, PROGRAM *name*) | Program organization unit (POU) *name* within the program source (also MCC chart, LAD/FBD program) |
| | *I/O address* | I/O variable |
| | TASK *name* | Assignment for the task *name* |
| | _device | Global device variable |
| **Language** | | Programming language of the program source |
| **Line/Block** | | Line number of the program source (e.g. ST source file)<br>With MCC units or MCC charts, the following is also shown:<br>• Number: Serial numbers for the command (block numbers) or<br>• DT: Declaration table |

**Note**

**Single-step tracking and trace diagnostic functions in MCC programming**

Additional variables and functions are created or used for these diagnostics functions:

- The variables TSI#dwuser_1 and TSI#dwuser_2 of the TaskStartInfo are used for the single-step tracking diagnostic function.
- Various internal functions and variables, whose identifier begins with an underscore, are automatically created by the compiler for the trace diagnostic function. The TSI#currentTaskId variable of the TaskStartInfo is also used.

With activated diagnostic function, these variables and functions are used for the control of the diagnostics function. These variables and functions must not be used in the user program.

### 5.6.1.3 Working with a cross-reference list

In the cross-reference list you are able to:

- Sort the column contents alphabetically:
  - To do this, click the header of the appropriate column.
- Search for an identifier or entry:
  - Click the "Search" button and enter the search term.
- Filter (Page 286) the identifiers and entries displayed.
- Copy contents to the clipboard in order to paste them into a spreadsheet program, for example.
  - Select the appropriate lines and columns.
  - Press the CTRL+C shortcut.
- Print the content (**Project > Print**).
- Open the referenced program source and position the cursor on the relevant line of the ST source file (or MCC command 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.

#### 5.6.1.4 Filtering the cross-reference list

You can filter the entries in the cross-reference list so that only relevant entries are displayed:

1. Click the "Filter settings" button.

   The "Filter Setting for Cross References" window will appear.
2. Activate the "Filter active" checkbox.
3. If you also want to display system variables and system functions:
   - Deactivate the "Display user-defined variables only" checkbox.
4. Set the desired filter criterion for the relevant columns:
   - Select the relevant entry from the drop-down list box or enter the criterion.
   - If you want to search for a character string within an entry: Deactivate the "Whole words only" checkbox.
5. Click **OK** to confirm.

The contents of the cross-reference list will reflect the filter settings selected.

**Note**

A filter is automatically activated after the cross-reference list has been created.

### 5.6.2 Program structure

The program structure contains all the 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 file)
- 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- 46 Elements of the display for the program structure

| Element | Description |
|---|---|
| Base<br>(declared POU or task used)) | List separated by a comma<br>• Identifier of the program organization unit (POU) or task<br>• Identifier of the program source in which the POU or task was declared, with add-on [UNIT]<br>• Minimum and maximum stack requirement (memory requirement of the POU or task on the local data stack), in bytes [Min, Max]<br>• 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:<br>• Identifier of called POU<br>• Optionally: Identifier of the program source / technology package in which the POU was declared:<br>Add-on (UNIT): User-defined program source<br>Add-on (LIB): Library<br>Add-on (TP): System function from technology package<br>• Only for function blocks: Identifier of instance<br>• Only for function blocks: Identifier of program source in which the instance was declared:<br>Add-on (UNIT): User-defined program source<br>Add-on (LIB): Library<br>• 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 file)
- 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 sources:

- Identifier of program source,
- Memory requirement, in bytes, for the following data areas of the program source:
  - **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,
- the **Code size** during the last compilation in bytes,
- the **Number of referenced sources**:

  The maximum number of connected sources is displayed (including system libraries), regardless of whether they are downloaded to the target system at a later date.

## 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 290).

  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 294).

  The pragmas with attributes for compiler options contained in an ST source file are evaluated by the compiler and interpreted as control statements.

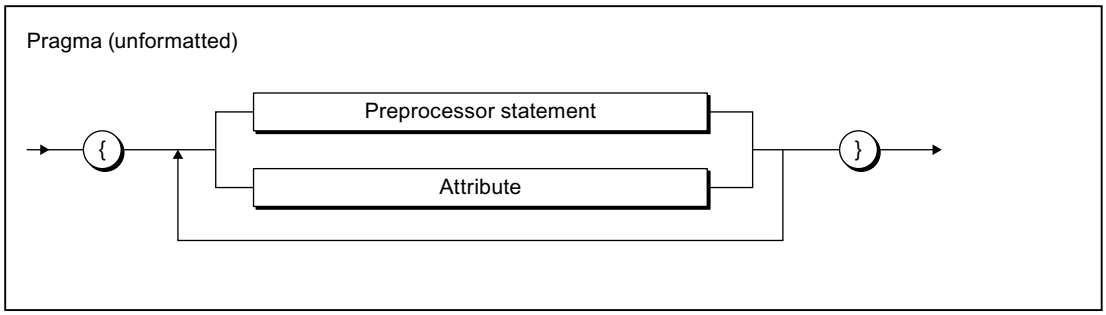



Figure 5-8 Pragma syntax

| NOTICE |
| --- |
| Be sure to use the correct pragma syntax (e.g. upper and lower case notation of attributes).<br>Unrecognized pragmas are ignored with no warning message. |

## 5.7.1 Controlling the 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 56)".
- Local for a program source file, see "Local compiler settings (Page 59)".

During the compilation of a program source file, you will be informed about the preprocessor actions. This requires, however, that the display of class 7 warnings is activated, see Meanings of the warning classes (Page 65). 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 294)".

Like all compiler messages, information about the preprocessor actions is shown on 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 also make definitions for the preprocessor in the Properties dialog box of the ST source file (see Making preprocessor definitions (Page 67)). This enables you to control the preprocessor with ST source files with know-how protection too (see Know-how protection for ST source files (Page 67)).

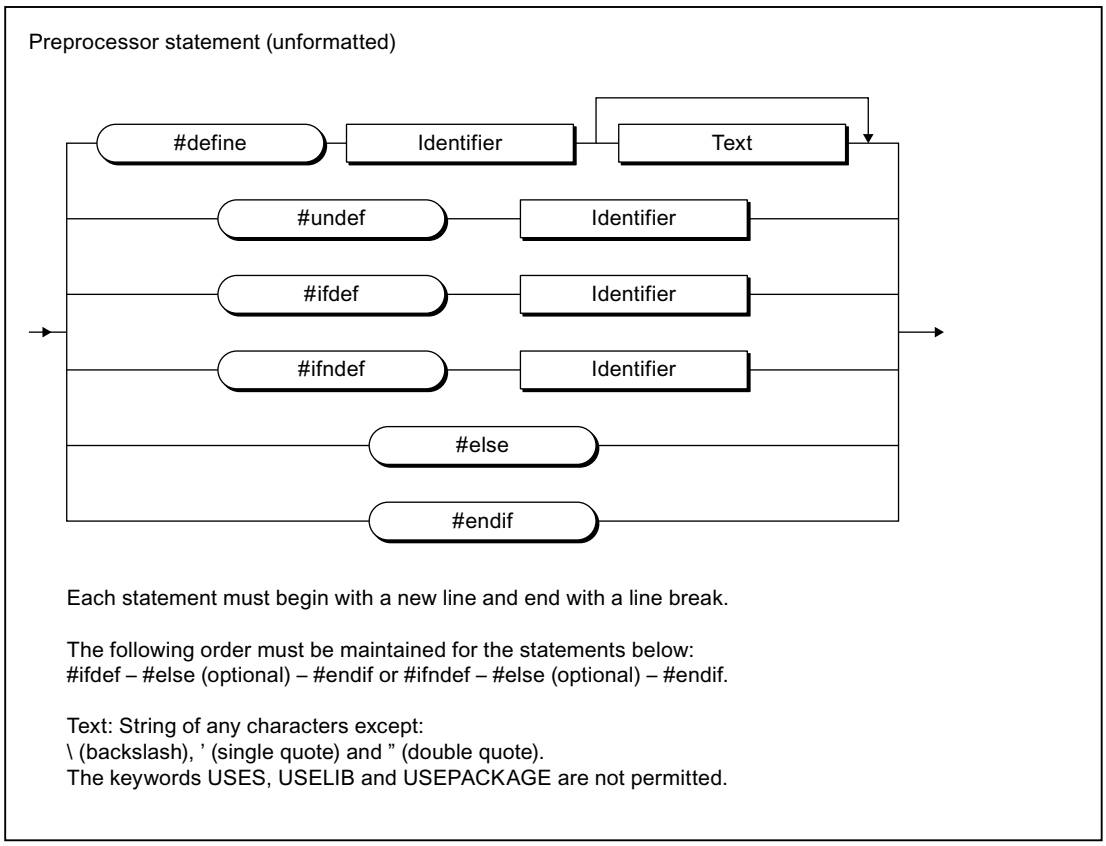



Figure 5-9 Syntax of a preprocessor statement

Table 5- 47 Preprocessor statements

| Statement | Meaning |
|---|---|
| #define | The specified identifier will be replaced below by the specified text.<br>Permissible characters: See table footnote. |
| #undef | The replacement rule for the identifier is cancelled. |
| #ifdef | For variant formation (conditional compilation)<br>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)<br>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)<br>Alternative branch to #ifdef or #ifndef.<br>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:<br>• For identifiers: In accordance with the rules for identifiers (Page 89).<br>• 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 Properties 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- 48 Example of preprocessor statements

**ST source file**

**With preprocessor statements**

```
INTERFACE
    FUNCTION_BLOCK fb1;
    VAR_GLOBAL
        g_var     : INT;
    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

IMPLEMENTATION
    FUNCTION f : INT
        VAR_INPUT
            i : INT;
        END_VAR
        f := i;
    END_FUNCTION

    FUNCTION_BLOCK fb1
        VAR_INPUT
            i_var : INT;
        END_VAR
        VAR_OUTPUT
            o_var : INT;
        END_VAR
        my_define := i_var;
// Delete the preprocessor definition
// For my_define
{
#undef my_define
}
        o_var := my_call + 1;
{
#ifdef my_define
}
        my_define := i_var;
{
#endif
}
    END_FUNCTION_BLOCK
END_IMPLEMENTATION
```

**Preprocessor output**

```
INTERFACE
    FUNCTION_BLOCK fb1;
    VAR_GLOBAL
        g_var     : INT;
    END_VAR

{

}


END_INTERFACE

IMPLEMENTATION
    FUNCTION f : INT
        VAR_INPUT
            i : INT;
        END_VAR
        f := i;
    END_FUNCTION

    FUNCTION_BLOCK fb1
        VAR_INPUT
            i_var : INT;
        END_VAR
        VAR_OUTPUT
            o_var : INT;
        END_VAR
        g_var := i_var;


{

}
        o_var := f(g_var) + 1;
{



}
    END_FUNCTION_BLOCK
END_IMPLEMENTATION
```

## 5.7.2 Controlling compiler with attributes

You can use attributes within a pragma to control the compiler.

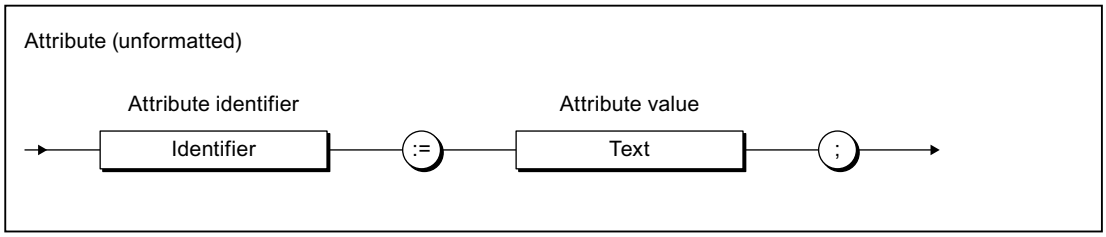



Figure 5-10 Syntax of an attribute for compiler options

Table 5- 49 Permissible attributes for compiler options

| Attribute identifier | Attribute value | Meaning |
|---|---|---|
| _U7_PoeBld_CompilerOption | The attribute affects the output of compiler warnings and information within an ST source file, as well as all subsequent lines of the ST source file. | |
| | warning:*n*:err | Outputs the warning or information specified by the number *n* as an error. Permissible values for *n*:<br>*n* = 16000 and higher: Number for a warning or information: |
| | warning:*n*:off | Warnings or information specified by the number *n* are not displayed. Permissible values for *n*:<br>*n* = 0 to 7: Warning class, see Meanings of the warning classes (Page 65).<br>*n* = 16000 and higher: Number for a warning or information: |
| | warning:*n*:on | Warnings specified by the number *n* are displayed<br>Permissible values for *n*:<br>*n* = 0 to 7: Warning class, see Meanings of the warning classes (Page 65).<br>*n* = 16000 and higher: Number for a warning or information: |
| 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:<br>• VAR_GLOBAL<br>• VAR_GLOBAL RETAIN<br>It affects only the unit variables declared in the associated declaration block.<br>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 238). | |
| | 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. |

| Attribute identifier | Attribute value | Meaning |
|---|---|---|
| BlockInit_OnChange | 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:<br>• VAR_GLOBAL (in the interface and implementation section)<br>• VAR_GLOBAL RETAIN (in the interface and implementation section)<br>• VAR (only for programs in a unit when the "Create program instance data only once" compiler option is active).<br>It affects only the variables declared in the associated declaration block.<br>See also Version ID of global variables and their initialization during download (Page 237). | |
| | FALSE | Download in RUN mode is **not** possible when the version identification of the declaration block is changed (default). |
| | TRUE | Download in RUN mode is possible despite the change to the version identification of the declaration block. The variables of the declaration block are initialized in the process. |
| BlockInit_OnDeviceRun | Only as of Version V4.1 of the SIMOTION kernel.<br>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:<br>• VAR_GLOBAL (in the interface and implementation section)<br>• VAR (only for programs in a unit when the "Create program instance data only once" compiler option is active).<br>It affects only the variables declared in the associated declaration block.<br>See also Memory ranges of the variable types (Page 223). | |
| | DISABLE | The variables declared in the associated declaration block are **not** initialized during the transition of the operating mode from STOP to RUN. |
| | ALWAYS | The variables declared in the associated declaration block are initialized in the transition of the mode from STOP to RUN. |
| | Default:<br>Version V4.1 and lower of the SIMOTION Kernel: DISABLE<br>Version V4.2 and higher of the SIMOTION Kernel: In accordance with the setting on the device | |

**NOTICE**

Be sure to use the correct upper- and lower-case notation for attributes!

**Note**

The insertion, deletion or changing of the HMI_Export, BlockInit_OnChange or BlockInit_OnDeviceRun attributes in a declaration block does not change its version identification!

Table 5- 50 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 SIMOTION devices

## 5.8.1 Rules for identifiers of the SIMOTION devices

### General device identifiers

Identifiers for SIMOTION devices in the project navigator do not have to comply with the general Rules for identifiers (Page 89).

All characters of the extended ASCII character set (ASCII code $20 to $7E, $80 to $FF) that can be displayed are permitted for the identifier of a SIMOTION device, except the following special characters:

- " (double inverted commas, ASCII code $22),
- & (Et character, ampersand, ASCII code $26),
- * (star, ASCII code $2A),
- : (colon, ASCII code $3A),
- < (less than character, ASCII code $3C),
- > (greater than character, ASCII code $3E),
- ? (question mark, ASCII code $3F),
- \ (backslash, ASCII code $5C),
- | (vertical line, ASCII code $7C).

Identifiers for the SIMOTION devices can only be specified in SIMOTION SCOUT or HW Config and only used in the programming languages.

Examples of valid device identifiers

- C240.2
- D455-2-DP (1)
- D445-2.PN-1

Device identifiers that do not comply with the general rules for identifiers can be used for SIMOTION devices of all versions (or all versions of the SIMOTION Kernel).

| NOTICE |
| --- |
| Projects that contain device identifiers, which do not comply with the general Rules for identifiers (Page 89), cannot be saved in the old project format (up to and including V4.2). |

### Device identifiers for PROFINET IO (name of station)

Device identifiers that are used as device names in PROFINET IO (name of station), must comply with the following- DNS conventions:

- Permissible lengths: 1 to 127 characters
- Organization using points "." is permissible in several labels; length of a single label: 1 to 63 characters
- Characters permitted within a label:
  - Letters "A" to "Z" and "a" to "z".
  - Numbers "0" to "9" (not at the beginning of the label)
  - Special character hyphen "-" (not at the beginning or end of a label)
  - Other special characters (such as accented characters, blank spaces, brackets, underscores) are not permitted.
- The following identifiers are not permitted for device names:
  - Identifiers that start with "port-xyz-" (x, y, z = 0 … 9),
  - Identifiers of the form n.n.n.n (n = 0 … 999).

### Use of device identifiers in SIMOTION SCOUT

Identifiers for SIMOTION devices that do not comply with the general rules for identifiers **must** be enclosed in double inverted commas (", ASCII code $22) when used in SIMOTION SCOUT (e.g. in the programming languages).

Example:

- "D455-2 DP (1)".axis_1. motionStateData.actualVelocity

  Access to the system variable *motionStateData.actualVelocity* of the technology object *axis_1* on the device *D455-2 DP (1)*.

---

**Note**

Device identifiers that comply with the general Rules for identifiers (Page 89) **can** also be enclosed in double inverted commas.

Example: The following notations are permitted for access to the system variable *motionStateData.motionState* of the technology object *axis_2* on the device *D435_2*:

- D435_2.axis_2.motionStateData.motionState
- "D435_2".axis_2.motionStateData.motionState

---

## 5.8.2 Making settings on the device (as of Kernel V4.2)

As of version V4.2 of the SIMOTION Kernel, you can make the following settings, among others, on the SIMOTION device:

- Memory area for the process image of the cyclic tasks and the fixed process image of the BackgroundTasks:
  - Separate memory areas for both process images - separate process image (Page 262)
  - Common process image (Page 259)
- Initialization of non-retentive global variables and program data during STOP-RUN transition
- Perform time synchronization with SINAMICS drive units
- Permit OPC-XML for global device variables or I/O variables

  The symbol information of these variables is available in the SIMOTION device. This is necessary for the watch function of IT DIAG, for example.
- Restrict area for automatic address assignment during the message frame configuration (as of version V4.3 of the SIMOTION Kernel).

  When the checkbox is activated, the specified address area for the automatic message frame configuration is blocked. Specification of the blocked area has no effect when the message frames have already been configured.

You will find a detailed description in the online help.

### Procedure

Settings on the device can be made as follows:

1. Select the SIMOTION device in the project navigator.
2. Select the **Edit > Object Properties** menu command.
3. Select the **Settings** tab.
4. Enter the settings.
5. Confirm with **OK**.

# 5.9 Forward declarations

When creating the source file, you should always pay attention to the order of the source file modules. A module that is to be called must always precede the calling module so that the former is recognized by the latter.

For example, variables must be declared before they are used, functions must be defined before they are called, and function blocks must be defined prior to instance declaration.

## Forward declaration for program organization units (POUs)

If the "Permit forward declarations" compiler option (Page 56) is activated, the program organization units (POUs) can be used for the following purposes in a source file, before the corresponding POUs have been fully defined:

- Instance declaration of a function block (Page 181)
- Call of a function (Page 180)
- Call of a program within a program (Page 188), provided that the other relevant requirements have been met.

If this compiler option is activated, the following statements within the TYPE / END_TYPE construct in the interface section (Page 194) or the implementation section (Page 197) of an ST source file are interpreted as prototypes:

- FUNCTION_BLOCK *fb-name*;
- FUNCTION *fc-name*;
- PROGRAM *prog-name*;

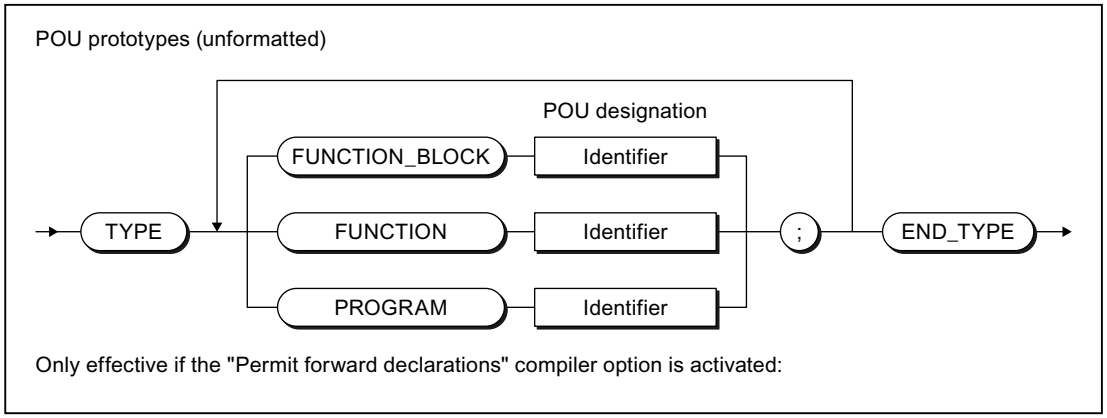



Figure 5-11 Syntax: POU prototypes

Once the prototype of a function block has been declared, an instance of the function block can be declared; the function block is not implemented in the ST source file until after these steps have been performed.

If a function or a program is called prior to being implemented, the declaration of the prototype is optional; the call can still be carried out even if the prototype is not declared. Please note the additional relevant requirements when calling a program within a program.

**Note**

If the POU prototypes within the interface section have been declared, the corresponding POUs are exported.

Export statements for POUs within the interface section (e.g. FUNCTION_BLOCK *fb-name*;) are also interpreted as prototypes.

If the "Permit forward declarations" compiler option (Page 56) is not activated, the POU prototypes are ignored. Only the prototypes in the interface section are interpreted as export statements for the corresponding POUs.

## Example

Table 5- 51 Sample program with forward declaration

```
(*
Only if the "Permit forward declarations" compiler option is activated.
*)

(*
The following compiler options also need to be activated
due to a program being called within the program:
"Permit language extensions" and
"Only create program instance data once".
*)

INTERFACE
    PROGRAM prog_main;
END_INTERFACE

IMPLEMENTATION
    TYPE                      // POU prototypes
        FUNCTION_BLOCK fb_1; // Required for instance declaration
        FUNCTION fc_1;       // Optional
        PROGRAM prog_1;      // Optional
    END_TYPE

    // Instance declaration prior to implementation
    VAR_GLOBAL
        var_fb_1 : fb_1;
    END_VAR
```

```
    PROGRAM prog_main
        VAR
            var_1, var_2 : DINT;
            var_3, var_4 : INT;
        END_VAR

        var_fb_1 (x_in := var_3, x_out => var_4);

        // Call prior to implementation
        var_2 := fc_1 (var_1);
        prog_1();
    END_PROGRAM

    // Implementations

    FUNCTION_BLOCK fb_1
        VAR_INPUT
            x_in : INT;
        END_VAR

        VAR_OUTPUT
            x_out : INT;
        END_VAR

        x_out := x_in;
    END_FUNCTION_BLOCK

    FUNCTION fc_1: DINT
        VAR_INPUT
            x_in : DINT;
        END_VAR

        fc_1 := x_in;
    END_FUNCTION

    PROGRAM prog_1
        VAR
            var_int1, var_int2 : INT;
        END_VAR

        var_int1 := var_int2;
    END_PROGRAM
END_IMPLEMENTATION
```

## 5.10 Jump statement and label

In addition to control statements (see Control statements (Page 152)), 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- 52 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. For all possible compilation error messages, i.e. compiler errors, see Compiler error messages and their remedies (Page 407). 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 76) is recommended.

## 6.2.1 Operating modes for program testing

### 6.2.1.1 Modes of the SIMOTION devices

Various SIMOTION device operating modes are available for program testing.

Table 6- 1 Operating modes of a SIMOTION device

| Operating mode | Meaning |
|---|---|
| Process mode | Program execution on the SIMOTION device is optimized for maximum system performance.<br>The following diagnostic functions are available, although they may have only restricted functionality because of the optimization for maximum system performance:<br>• Monitor variables in the symbol browser or a watch table<br>• Program status (only restricted):<br>– Restricted monitoring of variables (e.g. variables in loops, return values for system functions).<br>– Maximum of 1 program source (e.g. ST source file, MCC source file, LAD/FBD unit)[1] can be monitored.<br>• Trace tool (only restricted) with measuring functions for drives and function generator, see online help:<br>– Maximum of **1** trace on each SIMOTION device. |
| Test mode | The diagnostic functions of the process mode are available to the full extent:<br>• Monitor variables in the symbol browser or a watch table<br>• Program status:<br>– Monitoring of all variables possible.<br>– As of version V4.0 of the SIMOTION Kernel:<br>Several program sources (e.g. ST source files, MCC source files, LAD/FBD units)[1] can be monitored per task.<br>– For version V3.2 of the SIMOTION Kernel:<br>Maximum of 1 program source (e.g. ST source file, MCC source file, LAD/FBD unit)[1] can be monitored per task.<br>• Trace tool with measuring functions for drives and function generator, see online help:<br>– Maximum of **4** traces on each SIMOTION device.<br>In addition, the following diagnostics function is available:<br>• Trace for monitoring the program execution in program branches which are executed cyclically (only for the MCC programming language and for SIMOTION Kernel V4.2 and higher).<br>**Note**<br>Runtime and memory utilization increase as the use of diagnostic functions increases. |

| Operating mode | Meaning |
|---|---|
| Debug mode | In addition to the diagnostic functions of the test mode, you can use the following functions:<br>• Breakpoints<br>Within a program source, you can set breakpoints (Page 327). When an activated breakpoint is reached, selected tasks will be stopped.<br>• Controlling MotionTasks<br>On the "Task Manager" tab of the device diagnostics, you can use task control commands for MotionTasks; see the SIMOTION Basic Functions Function Manual.<br>No more than 1 SIMOTION device of the project can be switched to debug mode.<br>SIMOTION SCOUT is in online mode, i.e. connected to the target system.<br>Observe the following section: Important information about the life-sign monitoring (Page 308). |

[1] Each with 1 MCC chart or 1 LAD/FBD program in a program source.

## Selecting the operating mode

How to select the operating mode of a SIMOTION device:

1. Make sure a connection to the target system has been established (online mode).
2. Highlight the SIMOTION device in the project navigator.
3. Select the "Operating mode" context menu.
4. Select the required operating mode (see the table above).

   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 308).
5. Confirm with **OK**.

   The SIMOTION device switches to the selected operating mode (apart from with debug mode; see the explanation below).

## Special features with debug mode

Debug mode can only be selected for one SIMOTION device.

If you have selected debug mode, only SIMOTION SCOUT switches to it; the SIMOTION device is in test mode.

- The project navigator indicates that debug mode is activated for SIMOTION SCOUT by means of a symbol next to the SIMOTION device.
- The breakpoints toolbar is displayed.

Debug mode is not enabled for the SIMOTION device until at least one set breakpoint is activated. If all breakpoints are deactivated, debug mode is canceled for the SIMOTION device.

The status bar indicates that debug mode is activated for the SIMOTION device.

#### 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.

In the following cases, the SIMOTION device and SIMOTION SCOUT regularly exchange life-signs to ensure a correctly functioning connection:

- In debug mode with activated breakpoints.
- When controlling an axis or a drive via the control panel (control priority at the PC):

If the exchange of the life-signs is interrupted longer than the set monitoring time, the following reactions are triggered:

- In debug mode for activated breakpoints:
  - The SIMOTION device switches to the STOP operating state.
  - 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.

### 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 Parameters" 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**

The life-sign monitoring also responds in the following cases:

- Pressing the spacebar.
- Switching to a different Windows application.
- Too high a communication load between the SIMOTION device and SIMOTION SCOUT (e.g. by uploading task trace data).

The following reactions are triggered:

- In debug mode for activated breakpoints:
  - The SIMOTION device switches to the STOP operating state.
  - 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 states. 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-signs to ensure a correctly functioning connection. If the exchange of the life-signs is interrupted longer than the set monitoring time, the following reactions are triggered:<br>• In debug mode for activated breakpoints:<br>– The SIMOTION device switches to the STOP operating state.<br>– The outputs are deactivated (ODIS).<br>• For controlling an axis or a drive using the control panel (control priority for the PC):<br>– The axis is brought to a standstill.<br>– The enables are reset.<br>The following parameterizations are possible:<br>• Checkbox **active**:<br>If the checkbox is activated, life-sign monitoring is active.<br>The deactivation of the life-sign monitoring is not always possible.<br>• Monitoring time:<br>Enter the timeout.<br><br>**Prudence**<br>Do not make any changes to the defaults for life-sign monitoring, if possible.<br>Changes should only be made in special circumstances and in observance of all danger warnings. |
| Safety information | **Please observe the warning!**<br>Click the button to obtain further safety information.<br>See: Important information about the life-sign monitoring (Page 308) |

## 6.2.2 Editing program sources in online mode

### Online editing in process or test mode

If SIMOTION SCOUT is connected to a target system which is in the "process mode" or "test mode" operating mode, program sources (e.g. ST source files, MCC units with MCC charts) can generally be edited, compiled, and loaded to the target system in STOP operating mode. For information on downloading in RUN operating mode, see the corresponding section in the "SIMOTION Basic Functions" Function Manual.

However, you can only activate the "program status", "monitor program execution" (only for MCC), and trace (only for MCC) test functions for a program source or a program organization unit (POU) if the following conditions are met:

1. This program source or any POU of this source (e.g. MCC chart) does not contain any changes which have not been saved.
2. The program source (unit) in SCOUT is consistent with the target system.

**Note**

If the "program status" test function is activated, editing of the corresponding program source or one of its POUs is disabled.

If an MCC unit or MCC chart is changed and the "monitor program execution" or trace test functions are active for that unit or chart, the test functions are canceled.

### Online editing in debug mode

If SIMOTION SCOUT is in debug mode, editing is possible as long as the SIMOTION device is not in debug mode, i.e. no breakpoints are activated.

You can only activate breakpoints and, as a result, switch the SIMOTION device to debug mode if the corresponding program source and all its POUs are saved, compiled so they are up to date, and consistent with the target system.

If you attempt to edit a program source or POU when the SIMOTION device is in debug mode, you are requested to deactivate all breakpoints and, as a result, to switch the SIMOTION device out of debug mode.

| NOTICE |
| --- |
| If breakpoints have been activated and the SIMOTION device is in debug mode:<br>Entering a space switches the SIMOTION device to STOP operating mode and deactivates all outputs (ODIS). |

## 6.2.3 Symbol Browser

### 6.2.3.1 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.

### 6.2.3.2 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 83)".
- 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.

#### Procedure

Follow these steps:

1. Select the appropriate element in the project navigator in accordance with the following table.
2. In the detail view, click the **Symbol browser** tab.

   The corresponding variables are displayed in the symbol browser.
3. Select how each variable in the "Display format" column should be displayed.

Table 6- 3 Elements in the project navigator and variables to be monitored in the symbol browser

| Variables to be monitored in the symbol browser | Element to be selected in the project navigator |
|---|---|
| For variables in the user memory of the unit or in the retentive memory, refer to Memory areas of the variable types (Page 223):<br>• Retentive and non-retentive unit variables of the interface section of a program source (unit)<br>• Retentive and non-retentive unit variables of the implementation section of a program source (unit)<br>• Static variables of the function blocks whose instances are declared as unit variables.<br>• In addition, if the program source (unit) has been compiled with the "Only create program instance data once" compiler option (Page 56):<br>– Static variables of the programs.<br>– Static variables of the function blocks whose instances are declared as static variables of programs. | Program source (unit) |
| Variables in the user memory of the task, refer to Memory areas of the variable types (Page 223):<br>If the program source (unit) was compiled without the "Only create program instance data once" (default) compiler option (Page 56), the user memory of the task to which the program was assigned contains the following variables:<br>• Static variables of the programs.<br>• Static variables of the function blocks whose instances are declared as static variables of programs. | EXECUTION SYSTEM |
| System variables of a SIMOTION device | SIMOTION device |
| System variables of a technology object | Instance of the technology object |
| Global device variables | GLOBAL DEVICE VARIABLES |
| I/O variables (in the Address list tab of the detail view).<br>The Address list tab of the detail view can be opened by double-clicking the ADDRESS LIST element in the project navigator. | ADDRESS LIST |

**Note**

You can monitor temporary variables (together with unit variables and static variables) with **Program status** (see Properties of the program status (Page 321)).

**Note**

**Trace diagnostic function for MCC programming**

Various internal variables, whose identifier begins with an underscore, are automatically created by the compiler for the trace diagnostic function. These variables are displayed in the symbol browser.

With activated diagnostic function, these variables are used for the control of the diagnostics function. These variables must not be used in the user program.

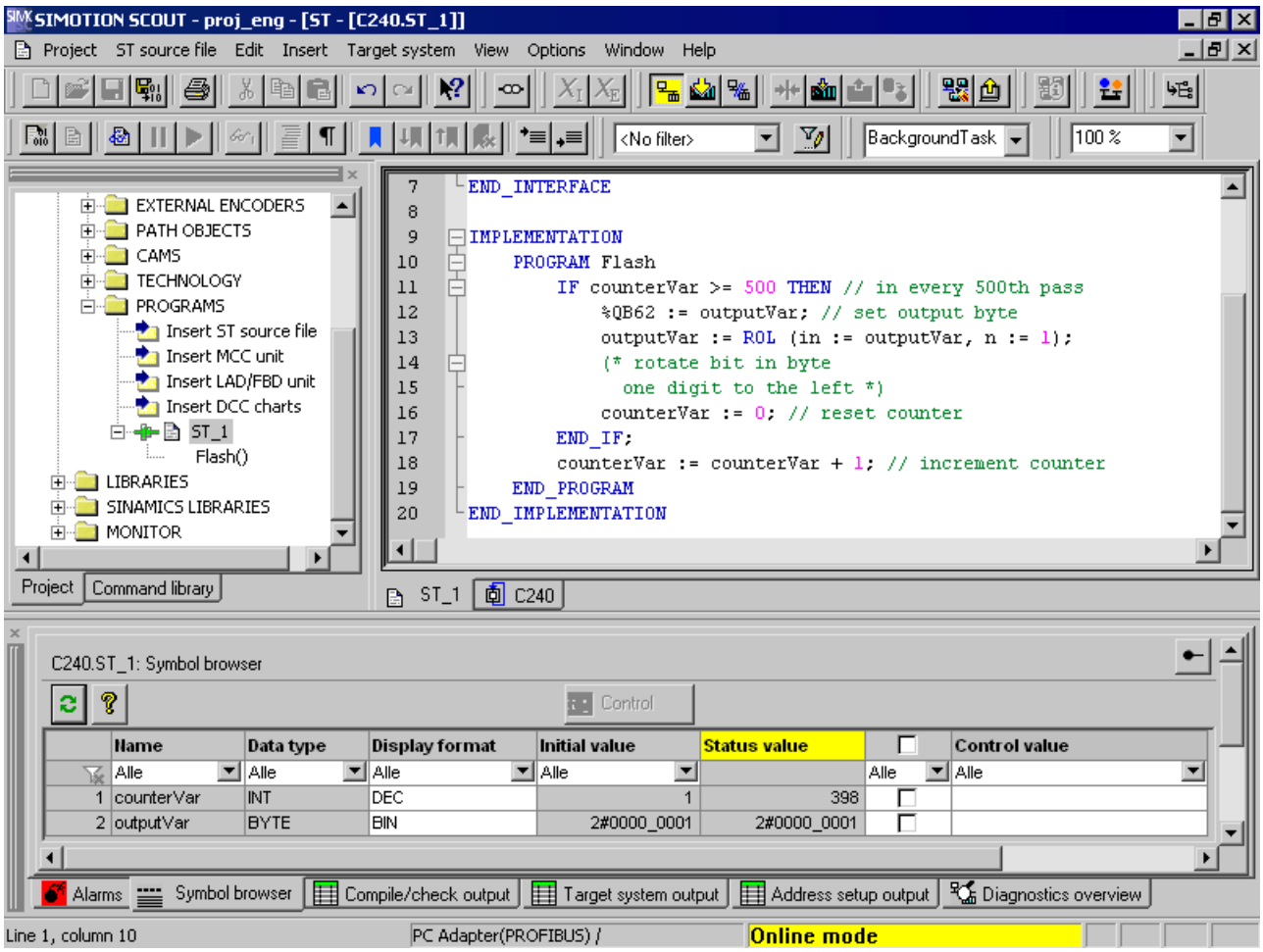



Figure 6-1 Viewing variables in the symbol browser

## 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 adjacent to this column
3. Click the **Control** button.

The values you entered are written to the selected variables.

| ⚠ WARNING |
| --- |
| You assign the entered values to the variables immediately. This can result in dangerous plant states, e.g. unexpected axis motion. |

| NOTICE |
| --- |
| Note when you change the values of several variables:<br>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. |

### Working with the symbol browser

The functions of the symbol browser and how you work with them are described in detail in the online help.

### Display invalid floating-point numbers

Invalid floating-point numbers are displayed as follows in the symbol browser (independently of the SIMOTION device):

Table 6- 4 Display invalid floating-point numbers

| Display | Meaning |
| --- | --- |
| 1.#QNAN<br>-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<br>-1.#INF | Bit pattern for + infinity in accordance with IEEE 754<br>Bit pattern for – infinity in accordance with IEEE 754 |
| -1.#IND | Bit pattern for indeterminate |

## 6.2.4 Monitoring variables in watch table

### 6.2.4.1 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.

### 6.2.4.2 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 **Add 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.

#### Status and controlling variables

If you are connected to the target system, you can monitor the variable contents.

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.

| ⚠ WARNING |
| --- |
| You assign the entered values to the variables immediately. This can result in dangerous plant states, e.g. unexpected axis motion. |

| NOTICE |
| --- |
| Note when you change the values of several variables:<br>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. |

### Working with the watch table

The functions of the symbol browser and how you work with them, are described in detail in the online help.

### Display invalid floating-point numbers

Invalid floating-point numbers are displayed as follows in the watch table (independently of the SIMOTION device):

Table 6- 5 Display invalid floating-point numbers

| LED | Meaning |
| --- | --- |
| 1.#QNAN<br>-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<br>-1.#INF | Bit pattern for + infinity in accordance with IEEE 754<br>Bit pattern for – infinity in accordance with IEEE 754 |
| -1.#IND | Bit pattern for indeterminate |

## 6.2.5 Variable status

"Variable status" enables you to monitor the current value for an individual variable, selected using the cursor, in an open program source or program organization unit (e.g. ST source file, MCC chart, LAD program).

### Requirements

- Make sure that a connection to the target system has been established and a project has been downloaded to the target system. For information on loading a project, see "Running the sample program (Page 83)".
- The program source containing the program organization unit (POE) whose variables you want to monitor must be consistent with the target system.
- The associated source (e.g. ST source file, MCC chart, LAD program) must be open.
- With the MCC programming language only: The parameter screen form for the command in which the variable you want to monitor is being used must be open.
- 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.

## Procedure

To monitor an individual variable using variable status:

1. Position the cursor above the identifier for a variable.
   - With the ST programming language: in the open ST source file
   - With the ST programming language: within an input field in the open parameter screen form
   - With the LAD/FBD programming language: within a network of the LAD/FBD program
2. Briefly position the cursor above the identifier.

The tool tip shows the current value of the variable. If you keep the cursor above the identifier for a longer period, the value is updated on an ongoing basis.

**Note**

With "variable status", the current value for the variable is displayed, wherever the selected variable is being used.

The "variable status" function enables you to monitor all those variables you are also able to monitor in the symbol browser (Page 312) or the address list. These are:

- System variables of SIMOTION devices
- System variables of technology objects
- Global device variables
- Retentive and non-retentive unit variables of the interface section of a program source (unit)
- Retentive and non-retentive unit variables of the implementation section of a program source (unit)
- Static variables of the programs
- Static variables of the function blocks whose instances are declared as unit variables
- Static variables of the function blocks whose instances are declared as static variables of programs
- I/O variables

## 6.2.6 Program run

### 6.2.6.1 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 320)" 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 sources:

Table 6- 6 SIMOTION RT program sources

| 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,<br>e.g. cam.pck for the library of the CAM technology package |

### 6.2.6.2 Program run parameters

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- 7 Program run parameter description

| Array | 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.<br>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, 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, 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 319).

### 6.2.6.3 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- 8 Program run toolbar

| Symbol | Meaning |
|---|---|
| | Display program run<br>Click this symbol to open the Program run call stack window. In this window, you can display the currently active code position with its call path.<br>See: Program run: Display code position and call path (Page 319) |

## 6.2.7 Program status

### 6.2.7.1 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 values which have been made available at regular intervals and displays them in the format you selected in the ST editor settings (Page 29) under "Format for status display".

In the case of functions and function blocks, you can select a location in an ST source file where 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- 9 Differences between process mode and test mode in Program Status

| | Process mode | Test mode |
|---|---|---|
| Optimization of program execution | For maximum system performance, only restricted diagnostics are possible. | For full diagnosis options |
| Maximum number of monitored program sources (e.g. ST source files, MCC source files, LAD/FBD units) | Maximum of 1 program source[1] | • As of version V4.0 of the SIMOTION kernel:<br>Multiple program sources[1] per task<br>• For version V3.2 of the SIMOTION Kernel:<br>Maximum of 1 program source[1] per task |
| Loops (e.g. WHILE, REPEAT, FOR) | The recording is interrupted on the first repeat loop.<br>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.<br>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) | The recording is interrupted during the execution of the system function.<br>Values are not correctly displayed in some cases. | The recording is performed correctly during the call of the system function.<br>Values are displayed correctly. |

[1] Each with 1 MCC chart or 1 LAD/FBD program in a program source.

**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 306))
- Up to version V4.0 of the SIMOTION Kernel, the programs must be assigned to various tasks.

### 6.2.7.2 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 59).
   - 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 56).
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 48) to start this test mode.

The ST editor window is now divided vertically:

- In the left-hand pane of the window, you can see the ST source file. You can select an area here.
- The right-hand pane of the window displays the variables for the selected area and their values.

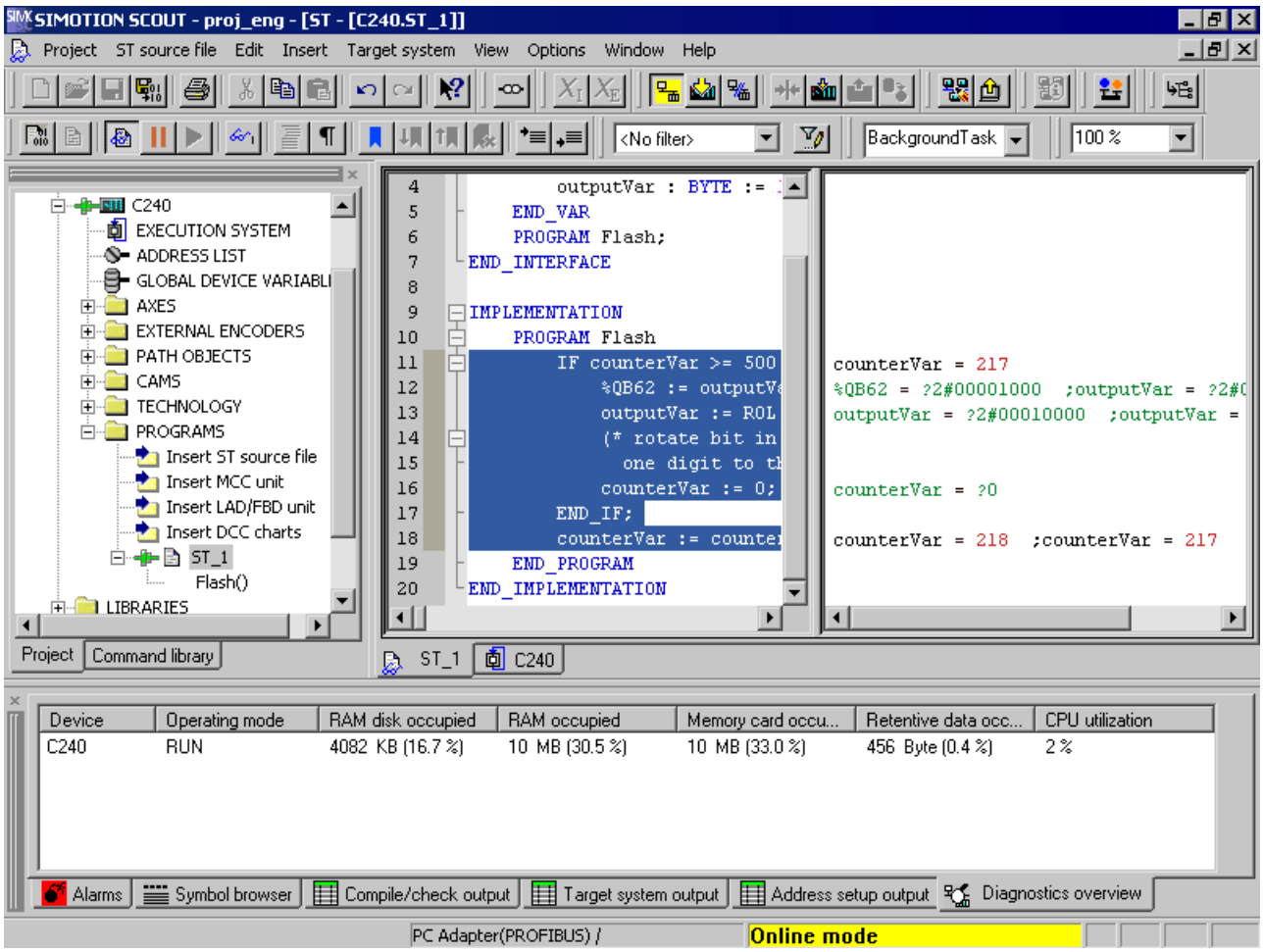



Figure 6-2 Part of an ST program in program status test mode

Follow the procedure below to test with program status:

1. In the left-hand pane, select the section of the ST source file you want to test.
2. 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 325).

The right-hand pane of the window displays the variables for the selected area and their values. The display of the variable values is updated cyclically. Variables with bit data type are updated cyclically and correspond to the format you selected in the ST editor settings (Page 29) under "Format for status display":

- 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 48) to stop the display.
- Click the button for **Continue monitoring of program variables** in the ST editor toolbar (Page 48) to continue the display.

You can force the update of the displayed values:

- Click the button for **Update** on the ST editor toolbar (Page 48).

  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.

### 6.2.7.3 Call path for program status

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.

### 6.2.7.4 Parameter call path status program

Table 6- 10 Program status call path parameter description

| Field | Description |
|---|---|
| Calling task | Select the task.<br>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.<br>The following are available:<br>• The code positions to be called within the selected task (with the name of the program source, line number, name of the POU).<br>If the selected calling code position is in turn called by several code positions, further lines are displayed in which you proceed similarly.<br>• **All:**<br>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. |

## 6.2.8 Breakpoints

### 6.2.8.1 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.

#### Procedure

Follow these steps:

1. Select "Debug mode" for the associated SIMOTION device; see Setting debug mode (Page 327).
2. Specify the tasks to be stopped, see Specifying the debug task group (Page 329).
3. Set breakpoints, see Setting breakpoints (Page 332).
4. Define the call path, see Defining a call path for a single breakpoint (Page 335).
5. Activate the breakpoints, see Activating breakpoints (Page 341).

### 6.2.8.2 Setting the debug mode

**⚠ WARNING**

You must observe the appropriate safety regulations.

Use the debug mode only with activated life-sign monitoring (Page 308) 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.

## Requirement

1. A connection to the target system must have been established (online mode)
2. Debug mode must not be selected for any SIMOTION device.

## Procedure

To set the debug mode, proceed as follows:

1. Highlight the SIMOTION device in the project navigator.
2. Select **Operating mode** from the context menu.
3. Select **Debug** mode (Page 306).
4. Accept the safety information
5. Parameterize the sign-of-life monitoring.

   See also section: Important information about the life-sign monitoring (Page 308).
6. Confirm with **OK**.

SIMOTION SCOUT switches to debug mode for this device; the SIMOTION device itself remains in "test mode", as long as at least one breakpoint is activated:

The project navigator indicates that debug mode is activated for SIMOTION SCOUT by means of a symbol next to the SIMOTION device.

The breakpoints toolbar (Page 334) is displayed.

As long as no breakpoints are activated, you can edit program sources in debug mode (Page 311).

Debug mode is not enabled for the SIMOTION device until at least one set breakpoint is activated. If all breakpoints are deactivated, debug mode is canceled for the SIMOTION device. The status bar indicates that debug mode is activated for the SIMOTION device.

**NOTICE**

Pressing the spacebar or switching to a different Windows application causes the following to happen if the SIMOTION device is in debug mode (breakpoints activated):

- 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.

### 6.2.8.3 Define the debug task group

On reaching an activated breakpoint, all tasks that are assigned to the debug task group are stopped.

#### Requirement

1. A connection to the target system must have been established (online mode).
2. SIMOTION SCOUT is in debug mode for the corresponding SIMOTION device; see Setting debug mode (Page 327).

#### Procedure

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 operating state): 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 HOLD operating state): 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- 11 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 state | 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 state | 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.<br>• Active: ODIS will be deactivated. All outputs and technology objects are released.<br>• Inactive: ODIS remains activated. All outputs and technology objects are only enabled for one STOP-RUN transition. |
| | Technology | Closed-loop control active | |

**Note**

You can only make changes to the debug task group if no breakpoints are active.

The settings of the debug task group are retained after exiting "Debug mode".

## Proceed as follows:

1. Set breakpoints (see Setting breakpoints (Page 332)).
2. Define the call path (see Defining a call path for a single breakpoint (Page 335)).
3. Activate the breakpoints (see Activating breakpoints (Page 341)).

### 6.2.8.4 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 306).

Table 6- 12 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.<br>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).<br>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.<br>Activate the checkbox, to release again the outputs and technology objects after program execution has been resumed.<br>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 329).<br>You can only make changes to the debug task group if no breakpoints are active. |

### 6.2.8.5 Debug table parameter

The debug table shows all breakpoints in the program sources of a SIMOTION device.

Table 6- 13 Debug table parameter description

| Field | | Description |
|---|---|---|
| **Debug points (table)** | | |
| | Active | The activation state of the corresponding breakpoint is displayed and can be modified:<br>**Active**: The breakpoint is activated.<br>**Inactive**: The breakpoint is deactivated.<br>See: Activating breakpoints (Page 341). |
| | Source, line (POU) | The code position with the set breakpoint is shown (together with the name of the program source, line number, name of the POU). |
| | Call path | Click the button to define the call path for the breakpoint.<br>See: Defining the call path for a single breakpoint (Page 335). |
| **All breakpoints ...** | | |
| | Activate | Click the button to activate all breakpoints (in all program sources) of the SIMOTION device.<br>See: Activating breakpoints (Page 341). |
| | Deactivate | Click the button to deactivate all breakpoints (in all program sources) of the SIMOTION device.<br>See: Activating breakpoints (Page 341). |
| | Delete | Click the button to clear all breakpoints (in all program sources) of the SIMOTION device.<br>See: Setting breakpoints (Page 332). |

### 6.2.8.6 Setting breakpoints

**Requirements:**

1. The program source with the POU (e.g. ST source file, MCC chart, LAD/FBD program) is open.
2. A connection to the target system must have been established (online mode).
3. SIMOTION SCOUT is in debug mode for the corresponding SIMOTION device; see Setting debug mode (Page 327).
4. The tasks to be stopped are specified, see Specifying the debug task group (Page 329).

## Procedure

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. Perform the following (alternatives):
   - Select the **Debug > Set/remove breakpoint** menu command (shortcut F9).
   - Click the button in the Breakpoints toolbar.

To remove a breakpoint, proceed as follows:

1. Select the code position with the breakpoint.
2. Perform the following (alternatives):
   - Select the **Debug > Set/remove breakpoint** menu command (shortcut F9).
   - Click the button in the Breakpoints toolbar.

To remove all breakpoints (in all program sources) of the SIMOTION device, proceed as follows:

- Perform the following (alternatives):
  - Select the **Debug > Remove all breakpoints** menu command (shortcut CTRL+F5).
  - 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".

The breakpoints set also remain saved after leaving debug mode; they are displayed in debug mode only.

---

You can use the program status (Page 323) diagnosis functions and breakpoints together in a program source 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 335).
2. Activate the breakpoints, see Activating breakpoints (Page 341).

## 6.2.8.7 Breakpoints toolbar

This toolbar contains important operator actions for setting and activating breakpoints:

Table 6- 14 Breakpoints toolbar

| Symbol | Meaning |
|---|---|
| | Set/remove breakpoint<br>Click this icon to set at breakpoint for the selected code position or to remove an existing breakpoint.<br>See: Setting breakpoints (Page 332). |
| | Activate/deactivate breakpoint<br>Click this icon to activate or deactivate the breakpoint at the selected code position.<br>See: Activating breakpoints (Page 341). |
| | Edit the call path<br>Click this icon to define the call path for the breakpoints:<br>• If a code position **with** breakpoint is selected: The call path for this breakpoint.<br>• If a code position **without** breakpoint is selected: The call path for all breakpoints of the POU.<br>See: Defining the call path for a single breakpoint (Page 335), Defining the call path for all breakpoints (Page 338). |
| | Activate all breakpoints of the active POU<br>Click this symbol to activate all breakpoints in the active program source or POU (e.g. ST source file, MCC chart, LAD/FBD program).<br>See: Activating breakpoints (Page 341). |
| | Deactivate all breakpoints of the active POU<br>Click this symbol to deactivate all breakpoints in the active program source or POU (e.g. ST source file, MCC chart, LAD/FBD program).<br>See: Activating breakpoints (Page 341). |
| | Remove all breakpoints of the active POU<br>Click this symbol to remove all breakpoints from the active program source or POU (e.g. ST source file, MCC chart, LAD/FBD program).<br>See: Setting breakpoints (Page 332). |

| Symbol | Meaning |
|---|---|
| | Debug table<br>Click this icon to display the debug table.<br>See: Debug table parameters (Page 332). |
| | Display call stack<br>Click this icon after reaching an activated breakpoint to:<br>• View the call path at the current breakpoint.<br>• View the code positions at which the other tasks of the debug task group have been stopped together with their call path.<br>See: Displaying the call stack (Page 343). |
| | Resume<br>Click this icon to continue the program execution after reaching an activated breakpoint.<br>See: Activating breakpoints (Page 341), Displaying the call stack (Page 343). |

### 6.2.8.8 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. A connection to the target system must have been established (online mode).
3. SIMOTION SCOUT is in debug mode for the corresponding SIMOTION device; see Setting debug mode (Page 327).
4. The tasks to be stopped are specified, see Specifying the debug task group (Page 329).
5. Breakpoint is set, see Setting breakpoints (Page 332).

#### Procedure

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, 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 341).

**Note**

You can use the "Display call stack (Page 343)" 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 338)

### 6.2.8.9 Breakpoint call path / task selection parameters

Table 6- 15 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.<br>The following are available:<br>• **All calling locations starting at this call level**<br>The user program will always be started when the activated breakpoint in any task of the debug task group is reached.<br>• The individual tasks from which the POU with the selected breakpoint can be reached.<br>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.<br>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:<br>Select the call path, i.e. the code position to be called (in the calling POU).<br>The following are available:<br>• **All calling locations starting at this call level**<br>**No** call path is specified. The user program will always be stopped at the activated breakpoint when the POU in the tasks is reached.<br>• 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).<br>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.<br>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. |

### 6.2.8.10 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

1. The program source with the POU (e.g. ST source file, MCC chart, LAD/FBD program) is open.
2. A connection to the target system must have been established (online mode).
3. SIMOTION SCOUT is in debug mode for the corresponding SIMOTION device; see Setting debug mode (Page 327).
4. The tasks to be stopped are specified, see Specifying the debug task group (Page 329).

## Procedure

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, 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 341).

**Note**

You can use the "Display call stack (Page 343)" 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 335)

## 6.2.8.11 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- 16 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.<br>The following are available:<br>• **All calling locations starting at this call level**<br>The user program will always be started when an activated breakpoint of the POU in any task of the debug task group is reached.<br>• The individual tasks from which the selected breakpoint can be reached.<br>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.<br>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:<br>Select the call path, i.e. the code position to be called (in the calling POU).<br>The following are available:<br>• **All calling locations starting at this call level**<br>**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.<br>• 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).<br>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.<br>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. |

### 6.2.8.12 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. A connection to the target system must have been established (online mode).
3. SIMOTION SCOUT is in debug mode for the corresponding SIMOTION device; see Setting debug mode (Page 327).
4. The tasks to be stopped are specified, see Specifying the debug task group (Page 329).
5. Breakpoints are set, see Setting breakpoints (Page 332).
6. Call paths are defined, see Defining a call path for a single breakpoint (Page 335).

## 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 file.
   - 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. Perform the following (alternatives):
   - Select the **Debug > Activate/deactivate breakpoint** menu command (shortcut F12).
   - Click the button in the Breakpoints toolbar.

To activate all breakpoints (in all program sources) of the SIMOTION device, proceed as follows:

- Perform the following (alternatives):
  - Select the **Debug > Activate all breakpoints** menu command.
  - Click the button in the Breakpoints toolbar.

Once the first breakpoint has been activated, the SIMOTION device switches to debug mode. It remains in this mode until the last breakpoint is deactivated.

In the Task status function bar, (Page 345) the tasks with activated breakpoints are highlighted in gray ( ).

**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.

In the case of activated breakpoints, the "Single step" test function of the SIMOTION MCC programming language cannot be used.

Breakpoints cannot be activate if the control priority is at the axis control panel. Conversely, you cannot fetch the control priority for the axis control panel when a breakpoint activated.

## Behavior at the activated breakpoint

On reaching an activated breakpoint (possibly using the selected call path (Page 335)), 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 329)". The breakpoint is highlighted.

In the Task status function bar, (Page 345) the task in which the breakpoint was reached is highlighted in red ( ).

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 312).

You can use the "Display call stack (Page 343)" 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:

- Perform the following (alternatives):
  - Select the **Debug > Continue** menu command (shortcut CTRL+F8).
  - Click the button on the Breakpoint toolbar to "Continue".

### Deactivate breakpoints

To deactivate a single breakpoint, proceed as follows:

1. Select the code position with the activated breakpoint.
2. Perform the following (alternatives):
   - Select the **Debug > Activate/deactivate breakpoint** menu command (shortcut F12).
   - Click the button in the Breakpoints toolbar.

To deactivate all breakpoints (in all program sources) of the SIMOTION device, proceed as follows:

- Perform the following (alternatives):
  - Select the **Debug > Deactivate all breakpoints** menu command.
  - Click the button in the Breakpoints toolbar.

Once the last breakpoint has been deactivated, the SIMOTION device switches to "test mode"; SIMOTION SCOUT continues to run in debug mode.

## 6.2.8.13 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 329) have been stopped.

## Procedure

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) on 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 box.

For names of the SIMOTION RT program sources, refer to the table in "Program run (Page 319)".

#### 6.2.8.14 Breakpoints call stack parameter

When an activated breakpoint (Page 341) is reached, you can display the following for each task in the debug task group (Page 329):

- 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- 17 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.<br>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 319)".

### 6.2.9 Task status function bar

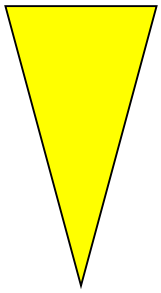
In a combo box, the Task status function bar displays all tasks of the active SIMOTION device, to which a program is assigned.

They are displayed under the following conditions:

1. SIMOTION SCOUT is in online mode.
2. The affected SIMOTION device is active, e.g.
   - In the project navigator, the SIMOTION device or an element in its subtree is selected (such as program source, technology object).
   - In the working area, an open window is active that belongs to an element in the subtree of the SIMOTION device.
3. The SIMOTION device is consistent.

A background color highlights the occurrence of specific events in the affected task, see the following table. The task in question is displayed in the combo box of the function bar according to the event priority.

Table 6- 18 Meaning of background colors in the Task status function bar

| Background color | Meaning | Priority |
|---|---|---|
| Cyan | The affected task waits for a command at the "Single step" test function (only for SIMOTION MCC programming language). | Highest  Lowest |
| Red | The affected task is located at a breakpoint (Page 341). | |
| Blue | In the affected task, the "Single step" test function is activated (only for SIMOTION MCC programming language) | |
| Gray | In the affected task, at least 1 breakpoint (Page 341) is activated. | |
| Yellow | In the affected task, the "Monitoring" test function is activated (only for SIMOTION MCC programming language). | |
| White | In the affected task, none of the above-mentioned test functions are activated. | |

**Note**

Tasks can only be selected in the combo box for the following test functions of the SIMOTION MCC programming language:

- Monitoring
- Single step

## 6.2.10 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.

## 6.2.11 Project comparison

SIMOTION SCOUT has a **project comparison** function (start this via the **Start object comparison** button) for comparing objects within the same project and/or objects from different projects (online or offline).

Project comparison allows you to establish any differences and, if necessary, run a data transfer to rectify them.

Objects are devices and their sub-objects, programs, technology objects (TOs) or drive objects (DOs), and libraries. Comparing projects is useful if you need to carry out service work on the system.

Further information on project and detail comparisons can be found in the SIMOTION Project Comparison Function Manual.

# 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.

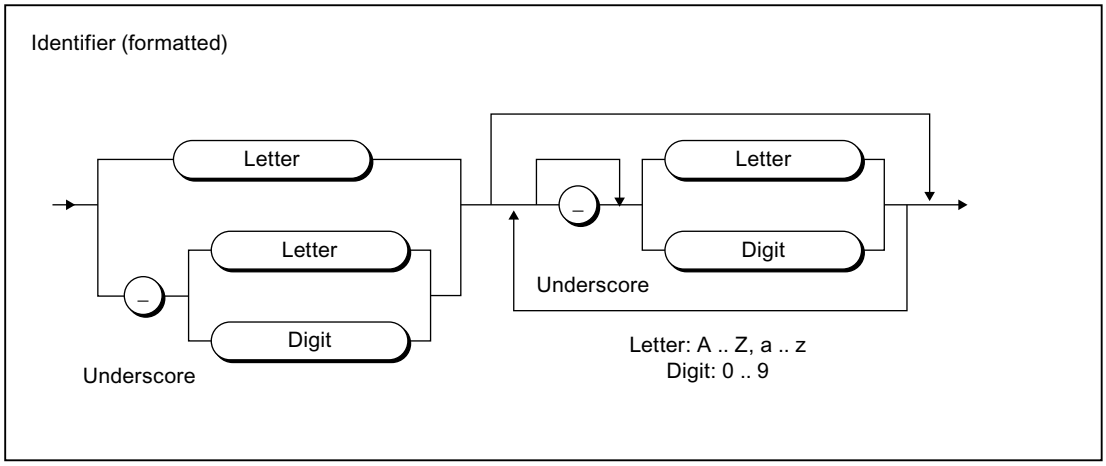



Figure A-1 Example of a lexical rule

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.

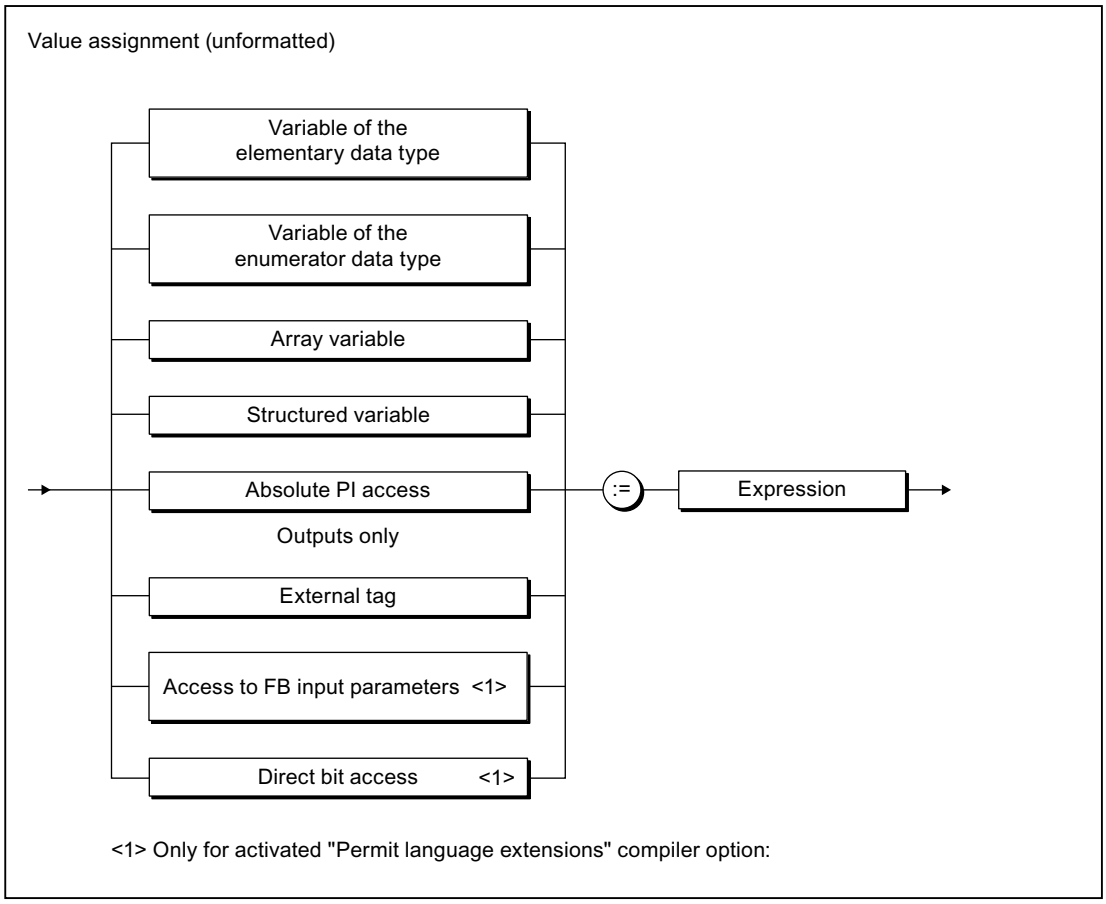



Figure A-2 Example of a syntactic rule

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 349) 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 365)).

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 87)

### 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<br>or<br>%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<br>or<br>%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 |
| +, –, *, /<br>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 |
| BYTE | Elementary data type | Bit data type |

| Keyword/identifier | Description | Rule |
|---|---|---|
| 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 | Time type |
| 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 |
| DINT_TO_UINT | Standard function for type conversion | Function call |
| DINT_TO_USINT | Standard function for type conversion | Function call |

| Keyword/identifier | Description | Rule |
|---|---|---|
| DINT_TO_WORD | Standard function for type conversion | Function call |
| DINT_VALUE_TO_BOOL | Standard function for type conversion | Function call |
| DO | Introduces the body for a FOR statement, WHILE statement, or WAITFORCONDITION statement | FOR statement, WHILE statement, WAITFORCONDITION statement |
| DT | Shorthand notation for DATE_AND_TIME | Time type |
| 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 | Expression |
| 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 | Program |
| 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 definition | User-defined data type |
| END_VAR | Ends a declaration block | Variable block, parameter block, constant block |
| END_WAITFORCONDITION | Ends the control statement for a task waiting for a programmable event | WAITFORCONDITION statement |
| END_WHILE | Ends the WHILE statement | WHILE statement |

| Keyword/identifier | Description | Rule |
|---|---|---|
| ENUM_TO_DINT | Standard function for type conversion | Function call |
| EXIT | Direct exit from loop execution | EXIT statement |
| EXP | Standard numeric function | Function call |
| EXPD | Standard numeric function | Function call |
| EXPRESSION | Programmable event for waiting task | Expression |
| 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 | GOTO statement |
| 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 |
| 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 |

| Keyword/identifier | Description | Rule |
|---|---|---|
| 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 |
| 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 | 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 |
| 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 |

| Keyword/identifier | Description | Rule |
|---|---|---|
| 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 (set as priority) | 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 | – |
| 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 |

| Keyword/identifier | Description | Rule |
|---|---|---|
| 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 the UDT definition | 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 |
| 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 |

| Keyword/identifier | Description | Rule |
|---|---|---|
| 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 or constants | Static variable block, temporary variable block in FC, constant block |
| VAR_GLOBAL | Introduces a declaration block for unit variables (global variables) or unit constants | Unit variables, unit constants |
| 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 | Temporary variable block in FB/program |
| VOID | No return value for function | 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 |
| 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

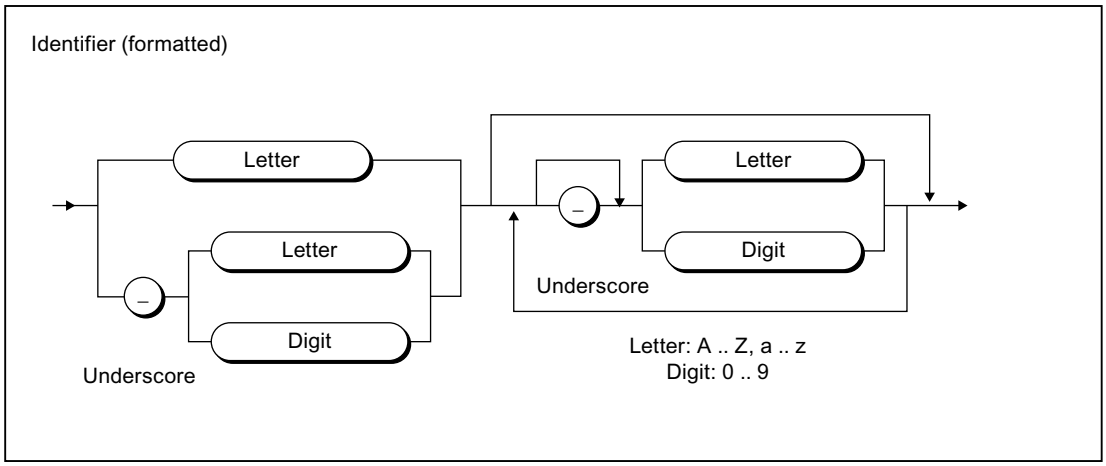



Figure A-3 Identifier

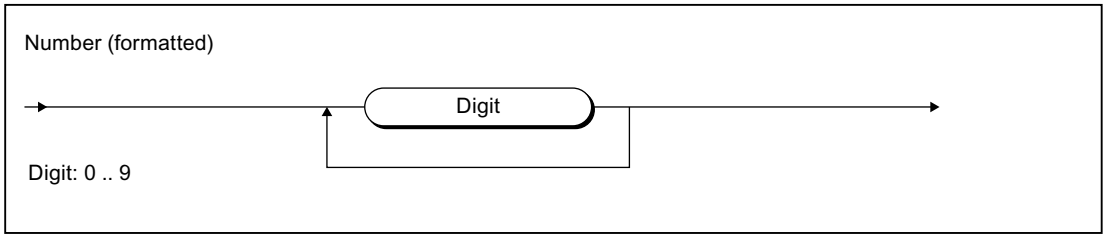



Figure A-4 Number

### A.1.3.2 Notation for constants (literals)

## Literals

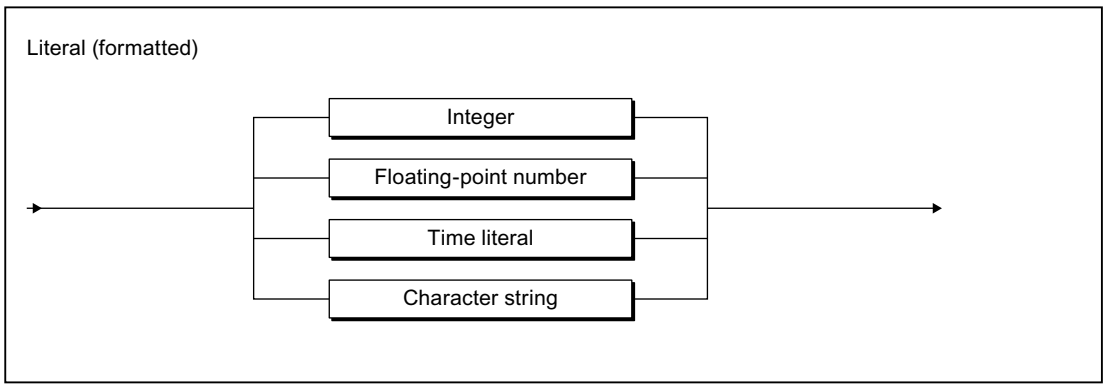



Figure A-5 Literal

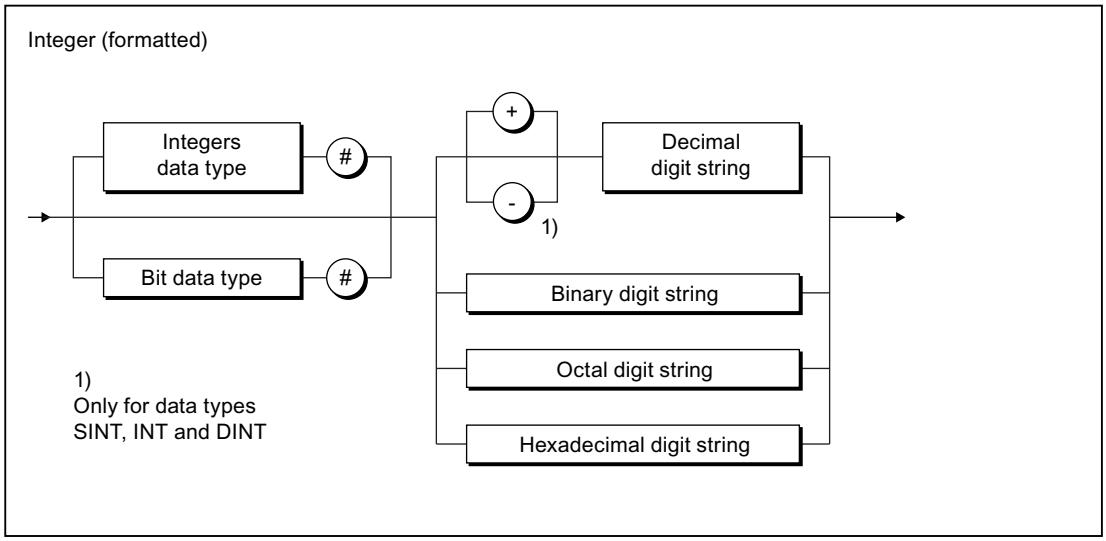



Figure A-6 Integer

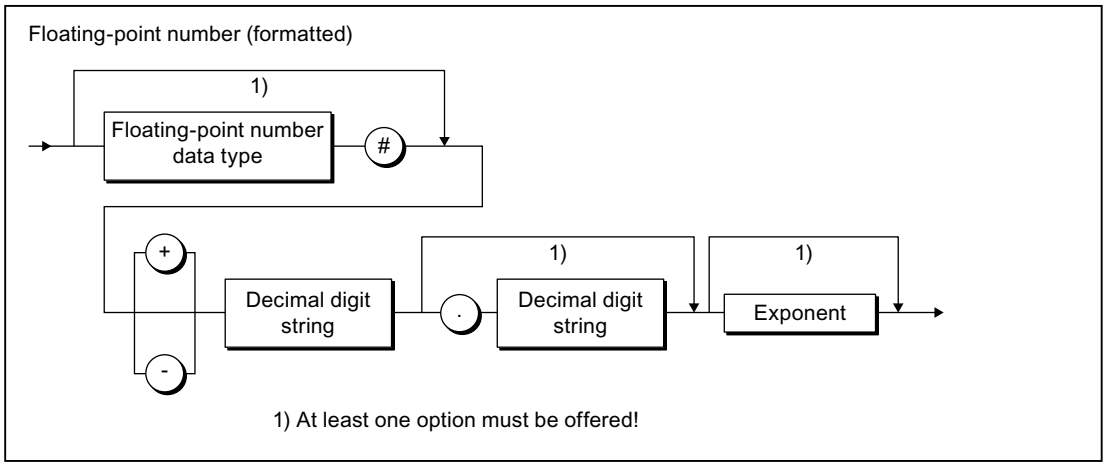



Figure A-7 Floating-point number

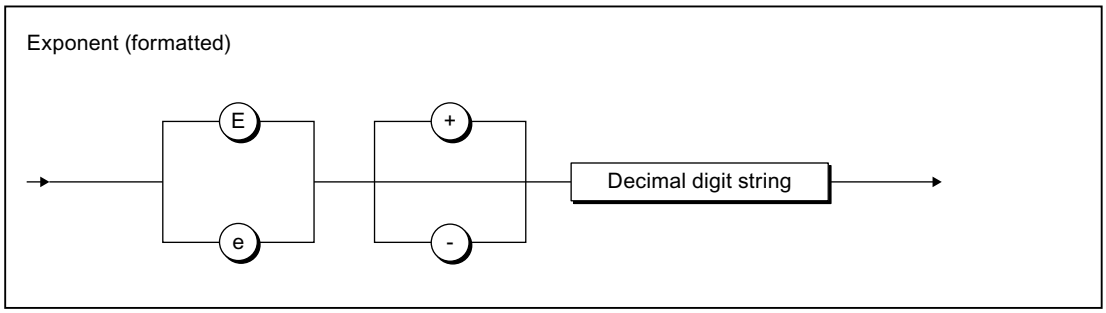



Figure A-8 Exponent

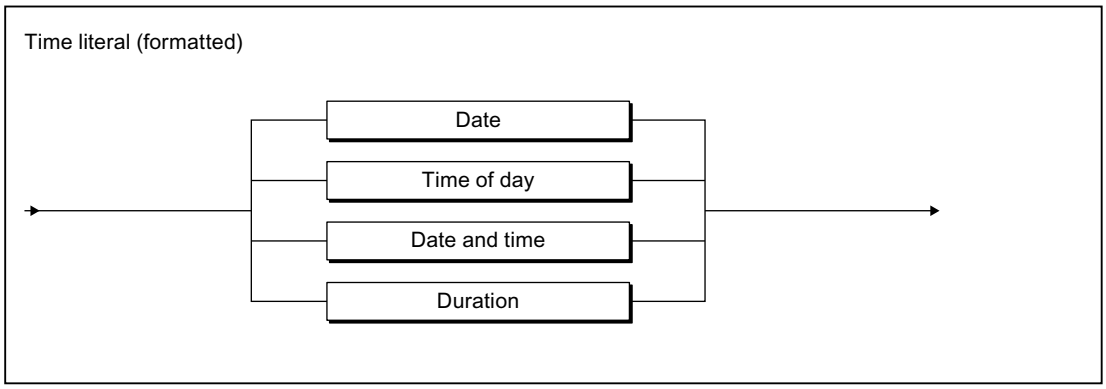



Figure A-9 Time literal

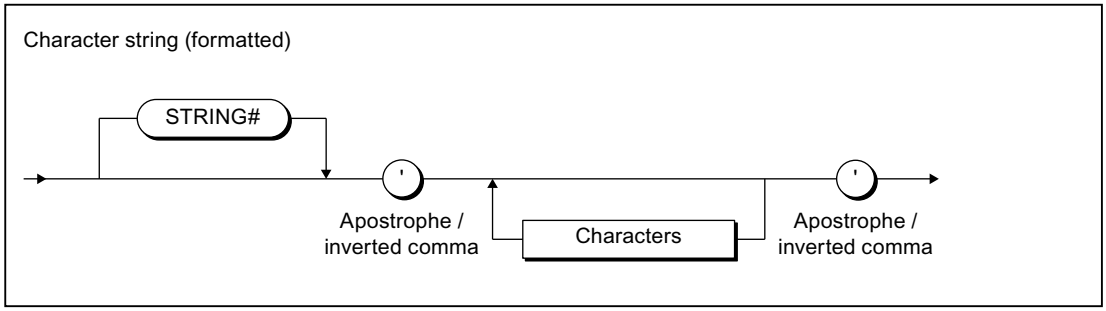



Figure A-10 Character string

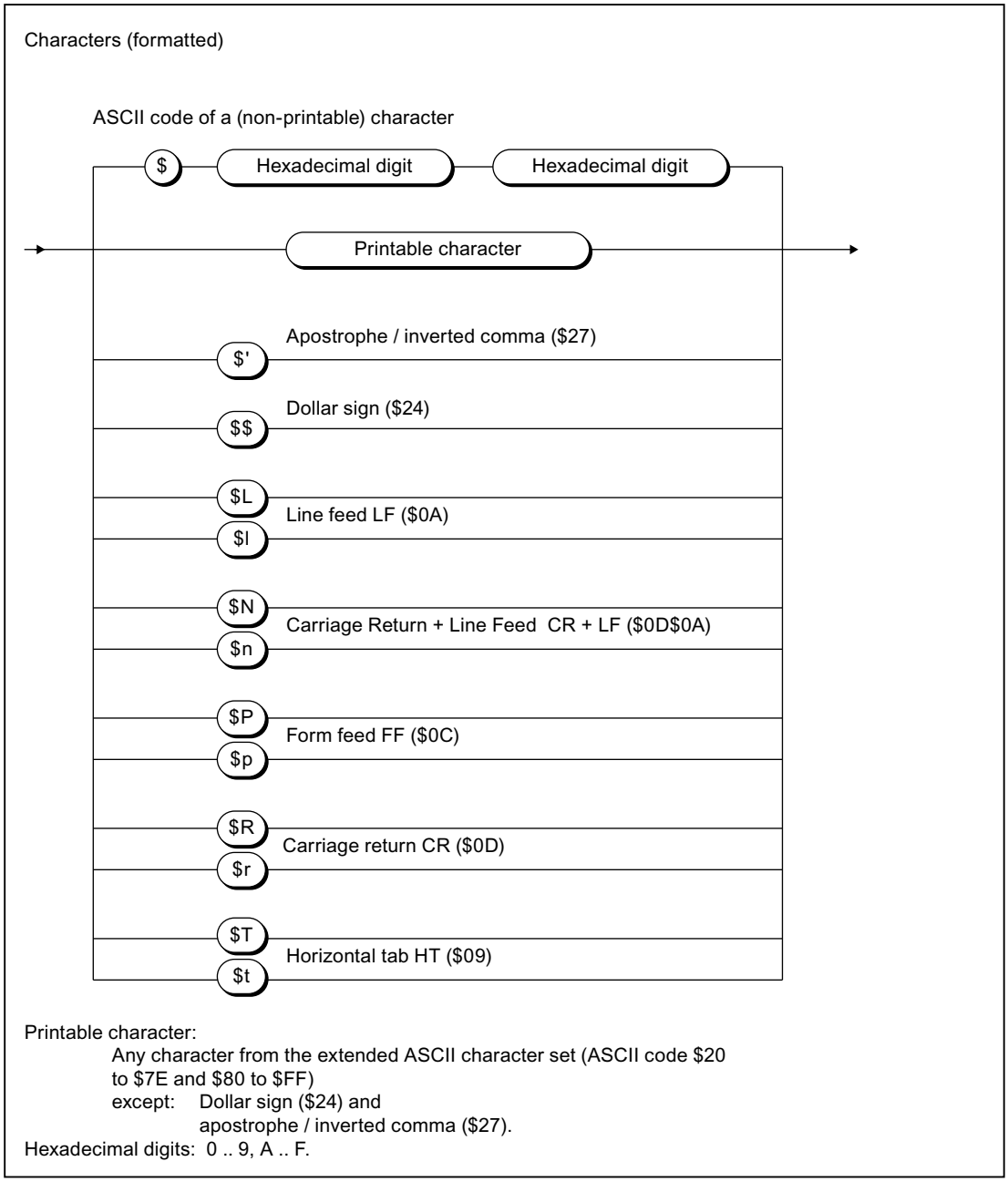



Figure A-11 Character

## Digit string

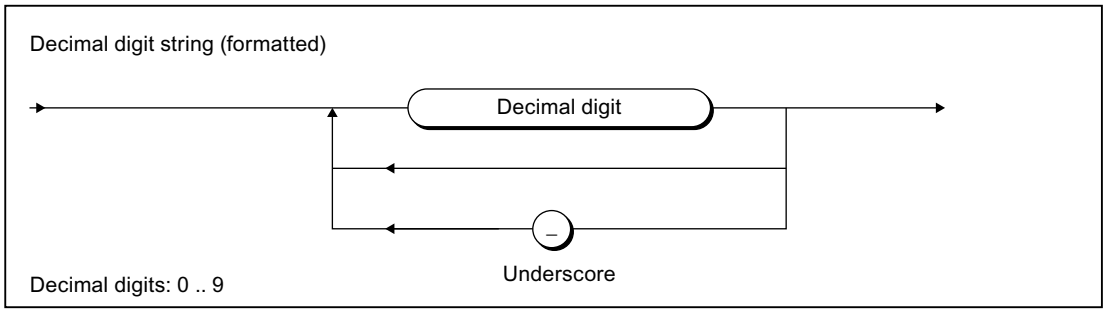



Figure A-12 Decimal digit string

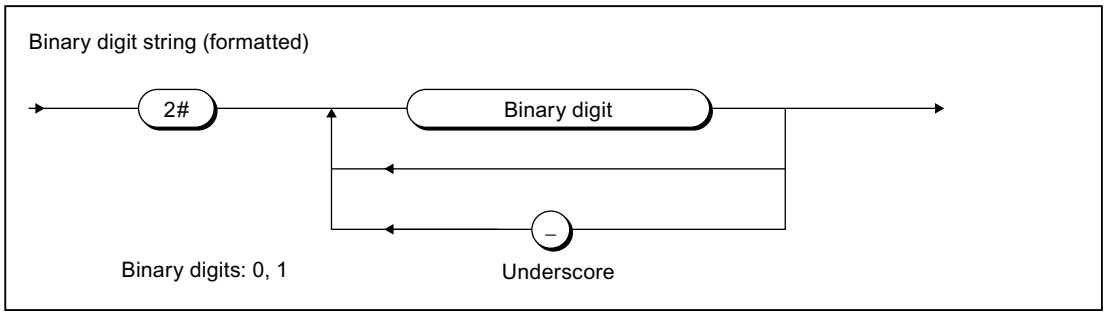



Figure A-13 Binary digit string

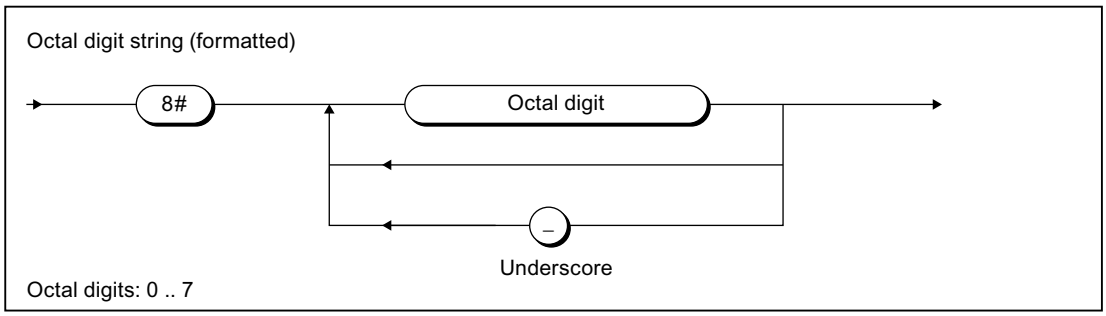



Figure A-14 Octal digit string

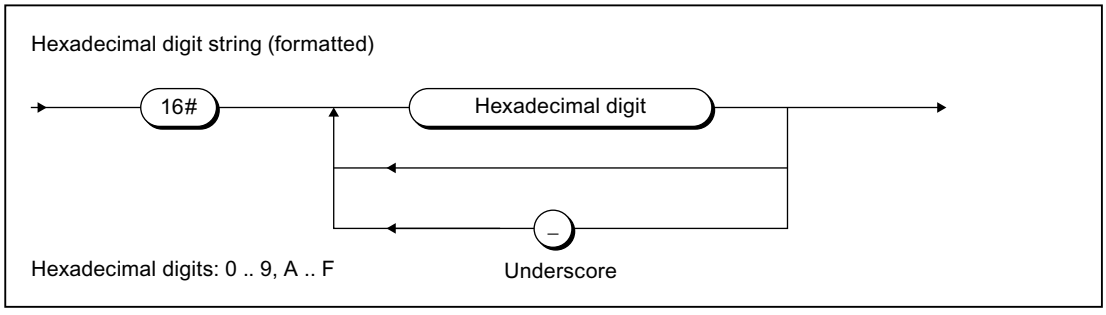



Figure A-15 Hexadecimal digit string

## Date and time

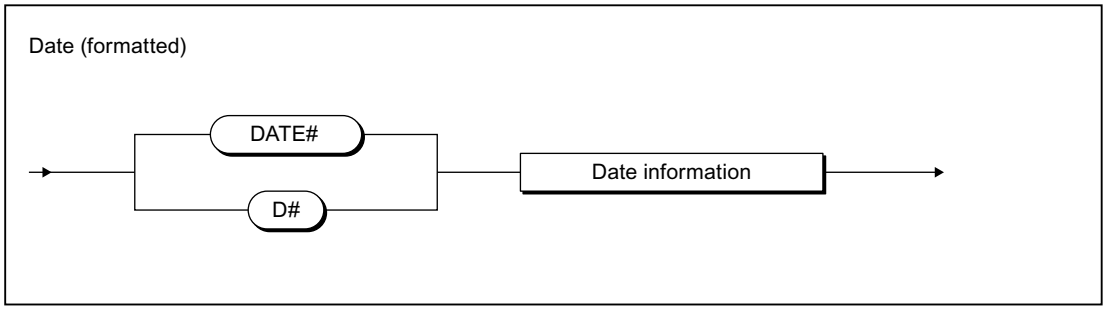



Figure A-16 Date

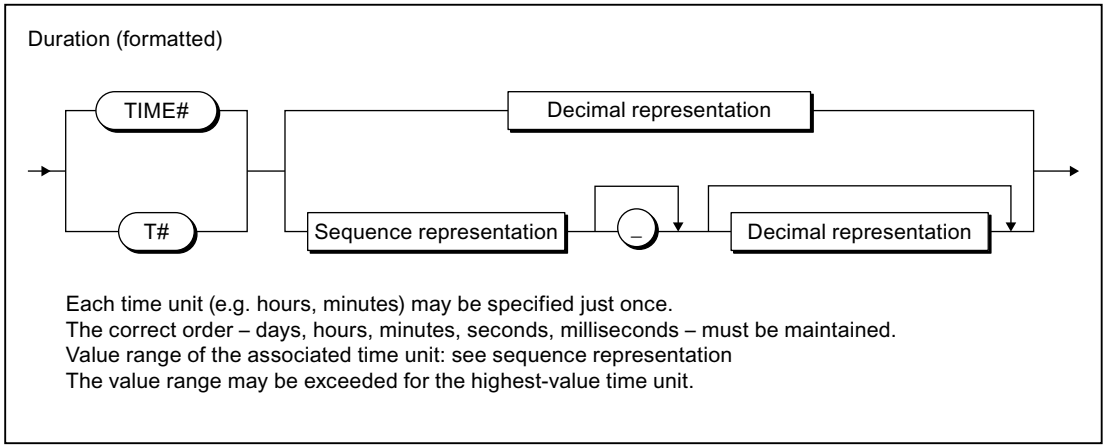



Figure A-17 Time

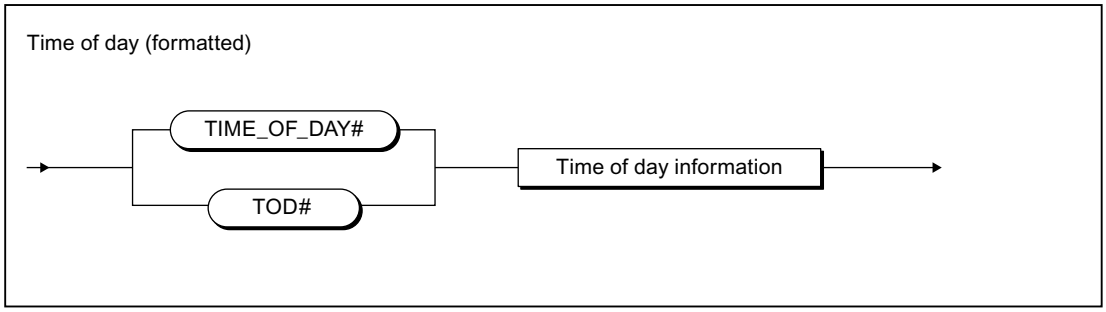



Figure A-18 Time

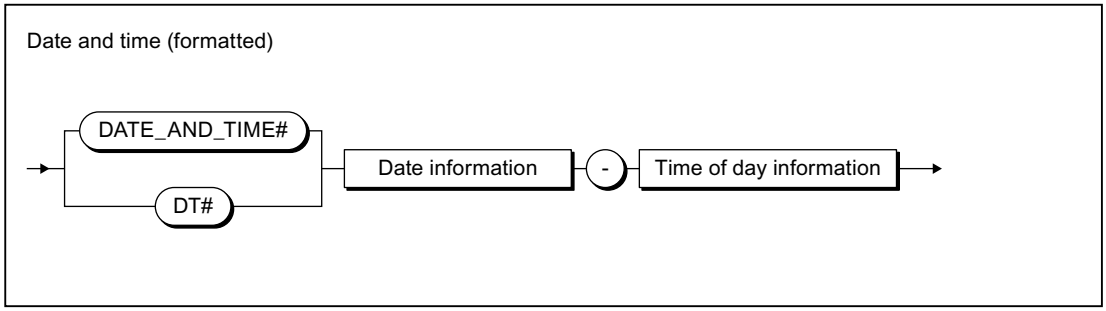



Figure A-19 Date and time

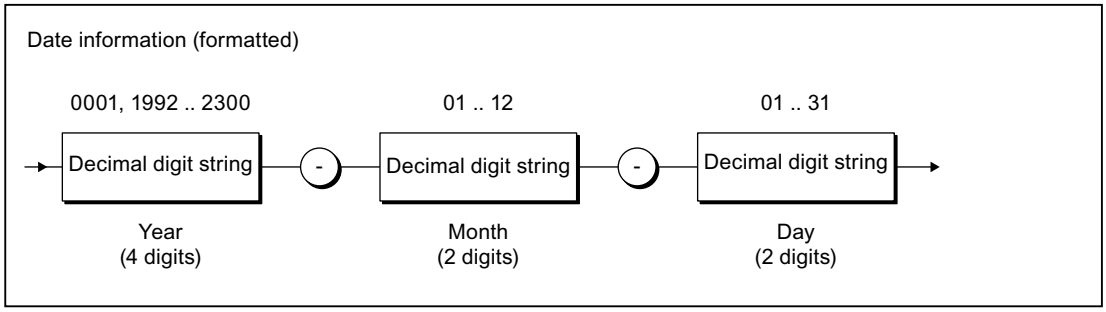



Figure A-20 Date information

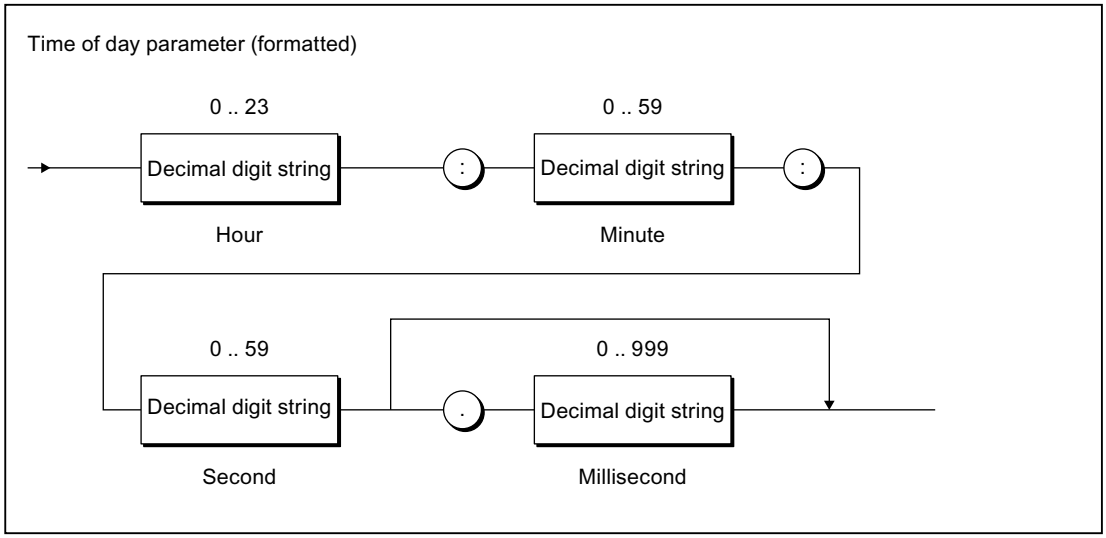



Figure A-21 Time of day information

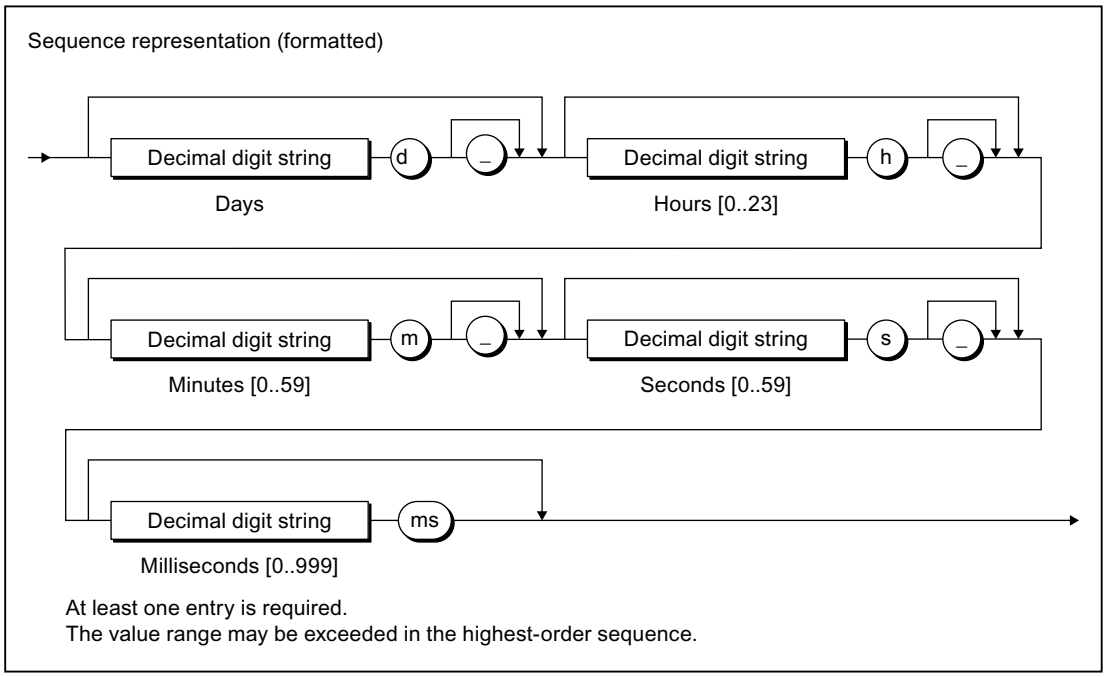



Figure A-22 Sequence representation

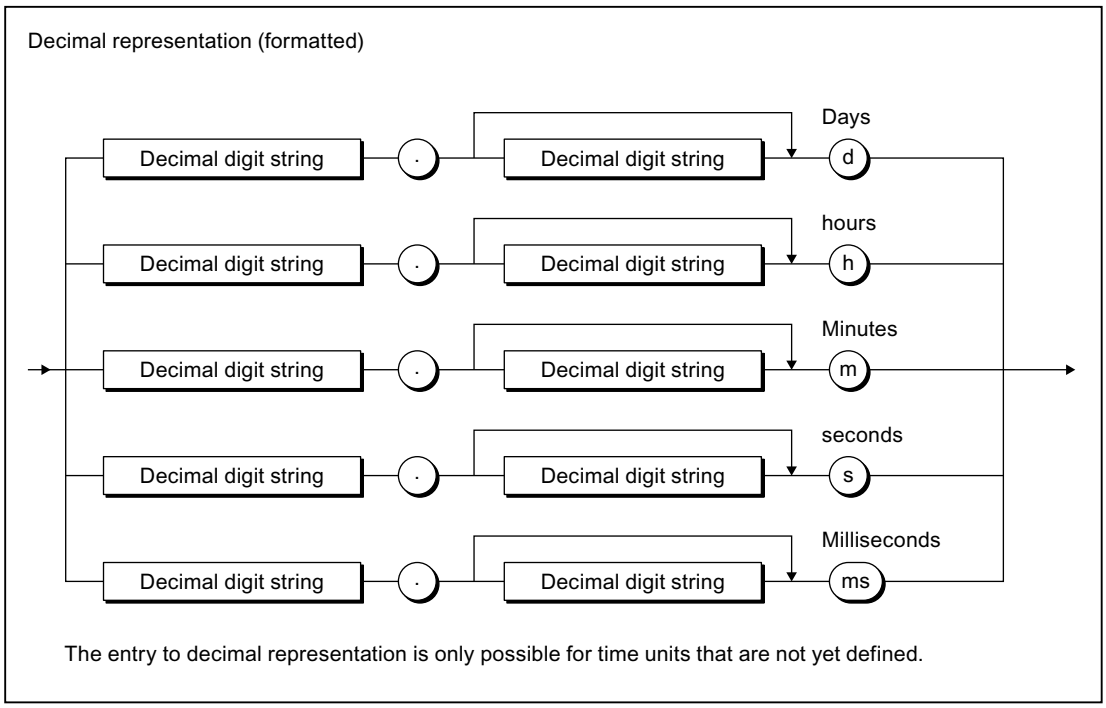



Figure A-23 Decimal representation

### 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.

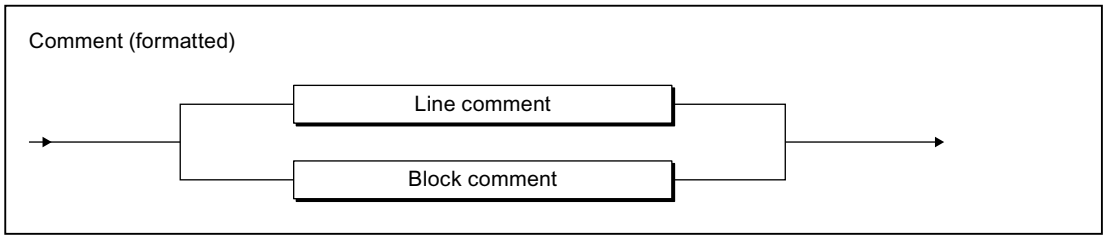



Figure A-24 Comments

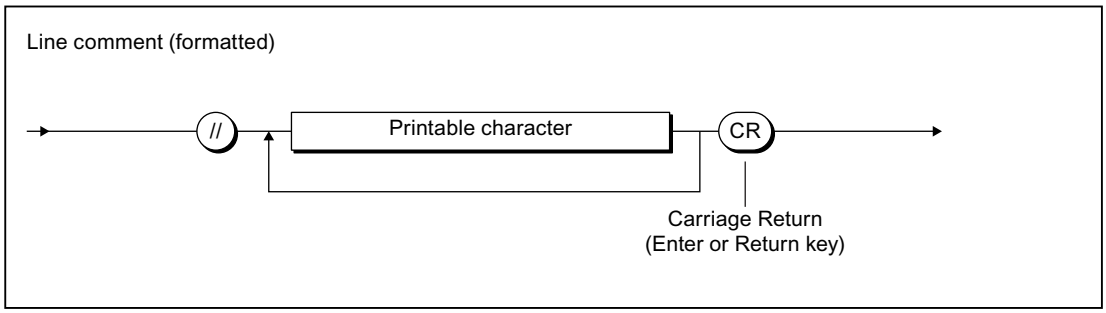



Figure A-25 Line comment

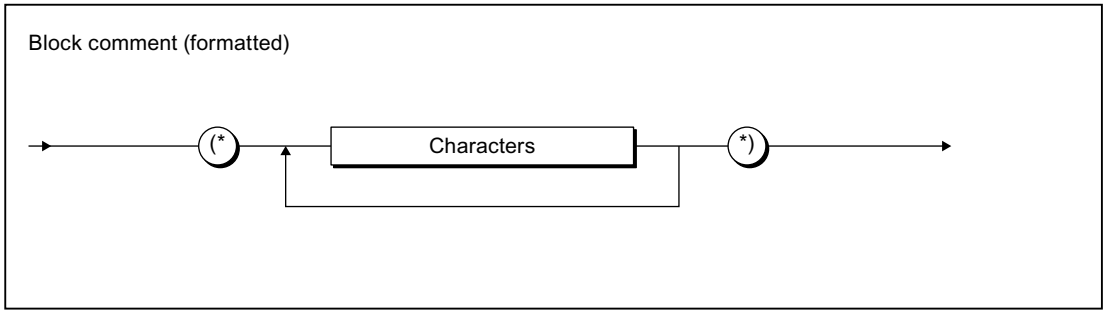
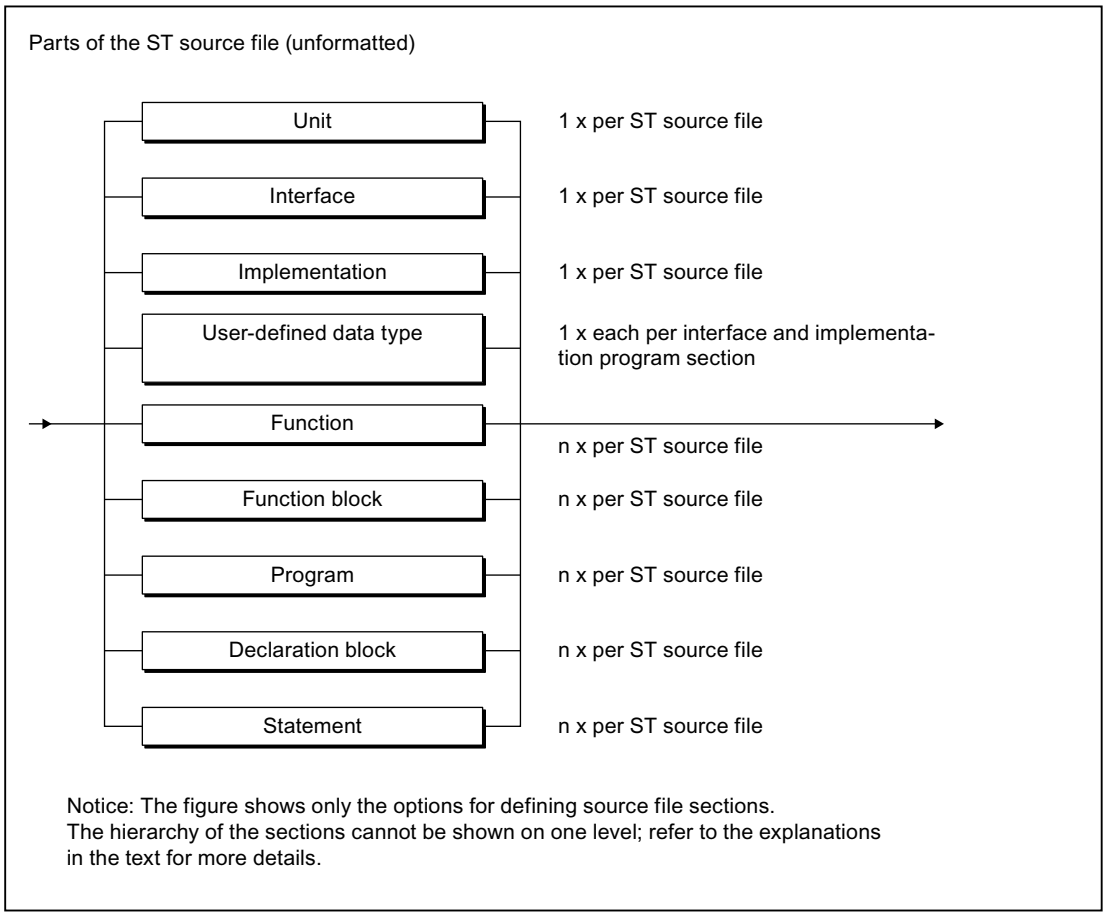



Figure A-26 Block comment

### A.1.3.4 Sections of the ST source file




Sections of the ST source file

### A.1.3.5 Structures of ST source files

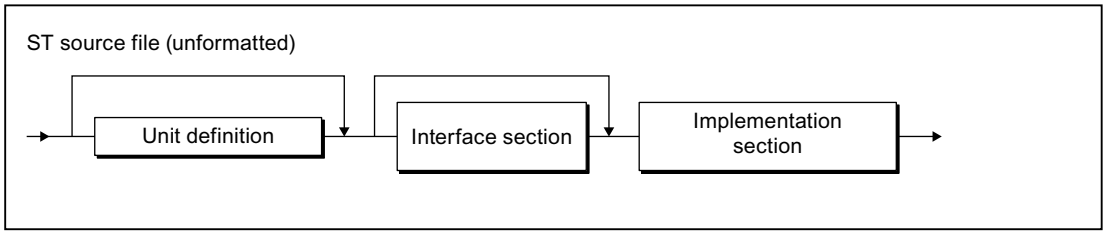



Figure A-27 ST source file

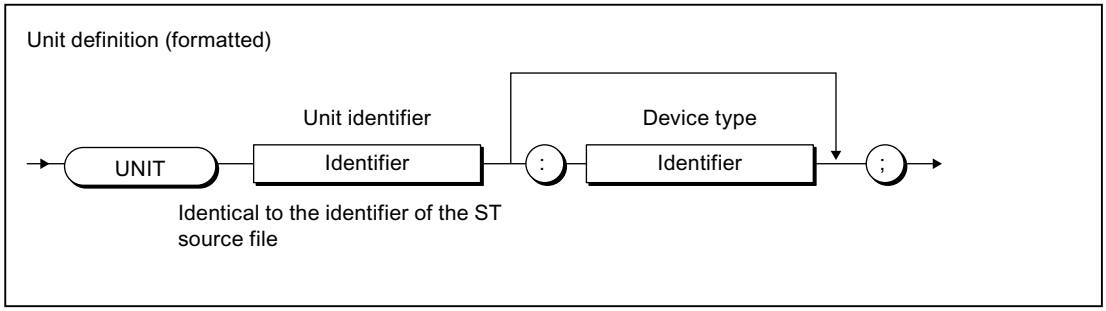



Figure A-28 Unit definition

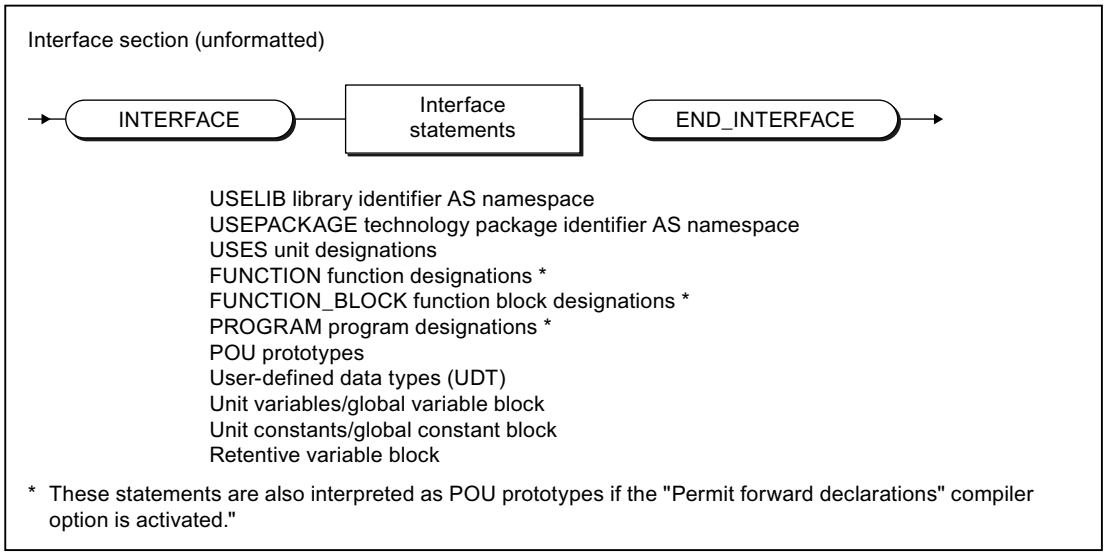



Figure A-29 Interface section

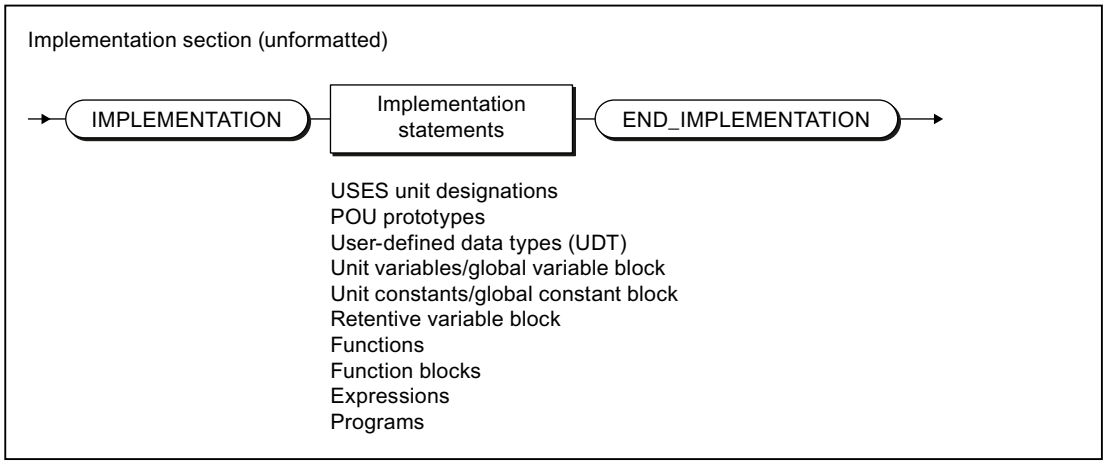



Figure A-30 Implementation section

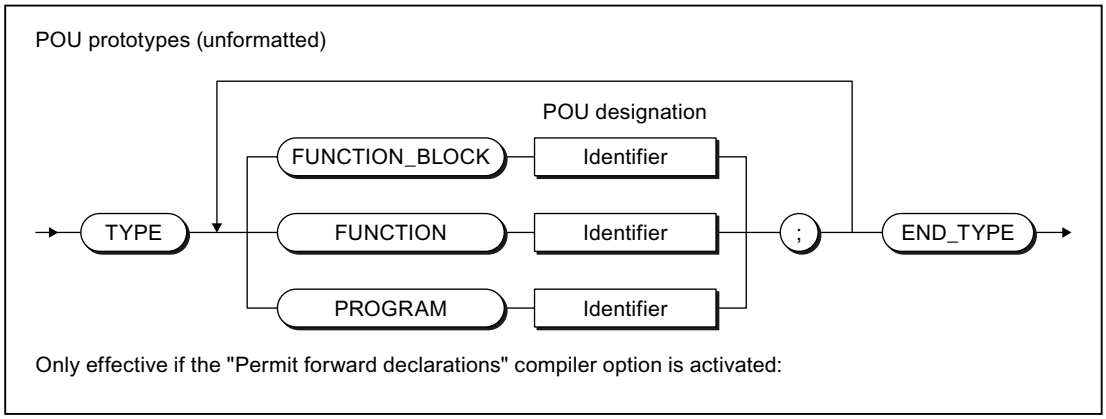



Figure A-31 POU prototypes

### A.1.3.6 Program organization units (POU)

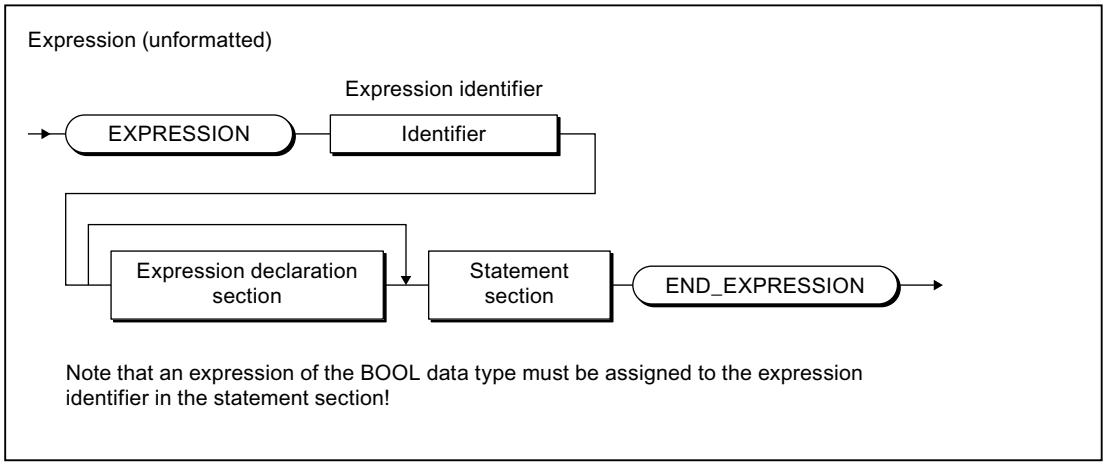



Figure A-32 Expression

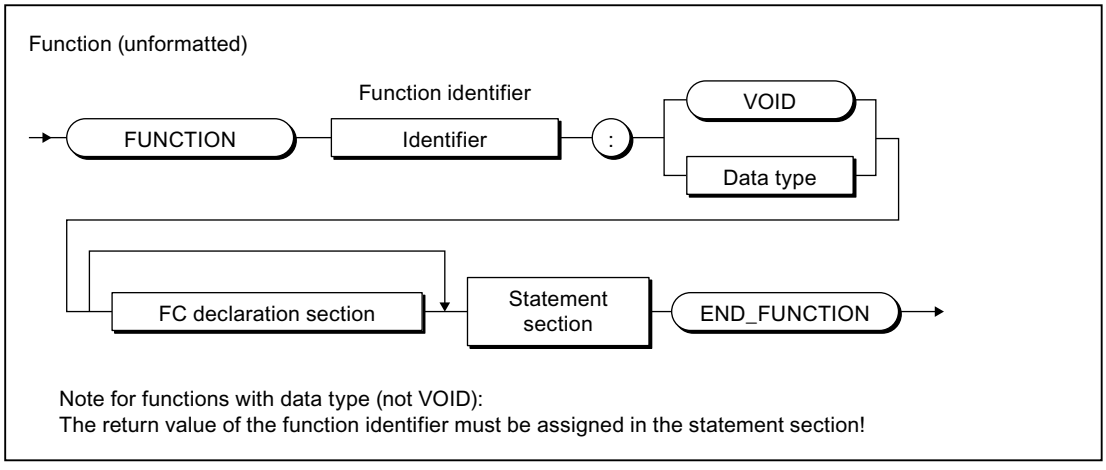



Figure A-33 Function (FC)

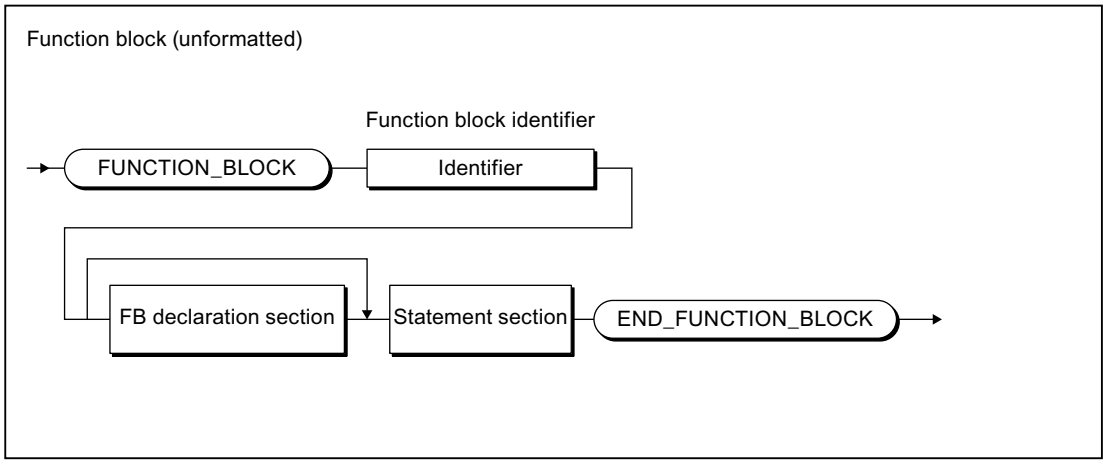



Figure A-34 Function block (FB)

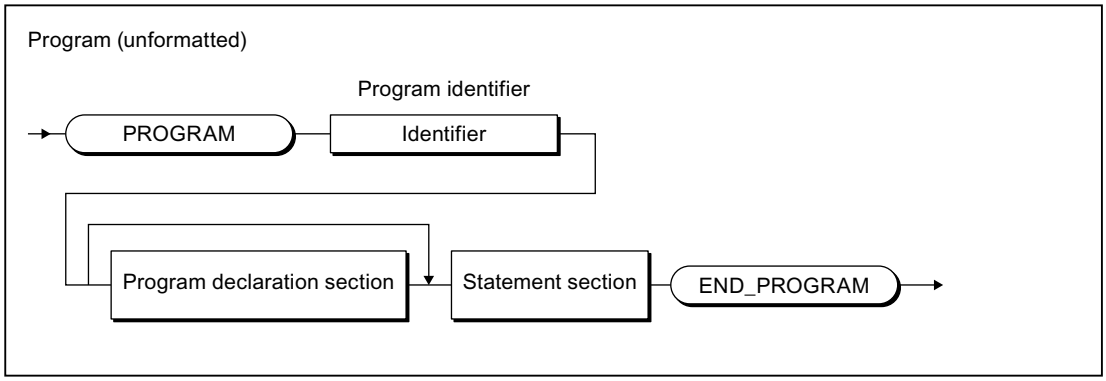



Figure A-35 Program

### A.1.3.7 Declaration sections

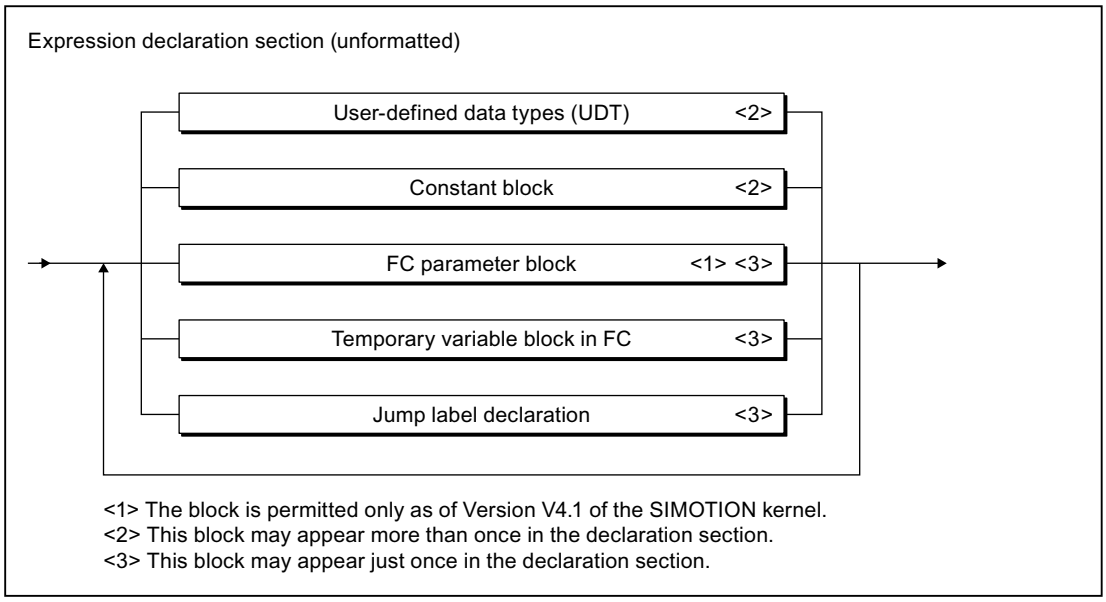



Figure A-36 Expression declaration section

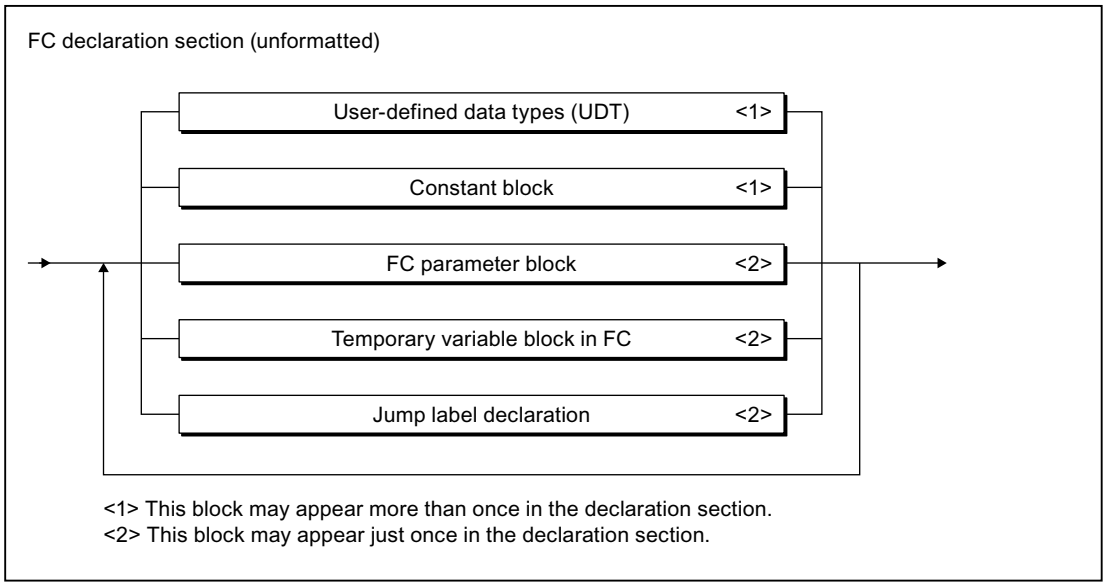



Figure A-37 FC declaration section

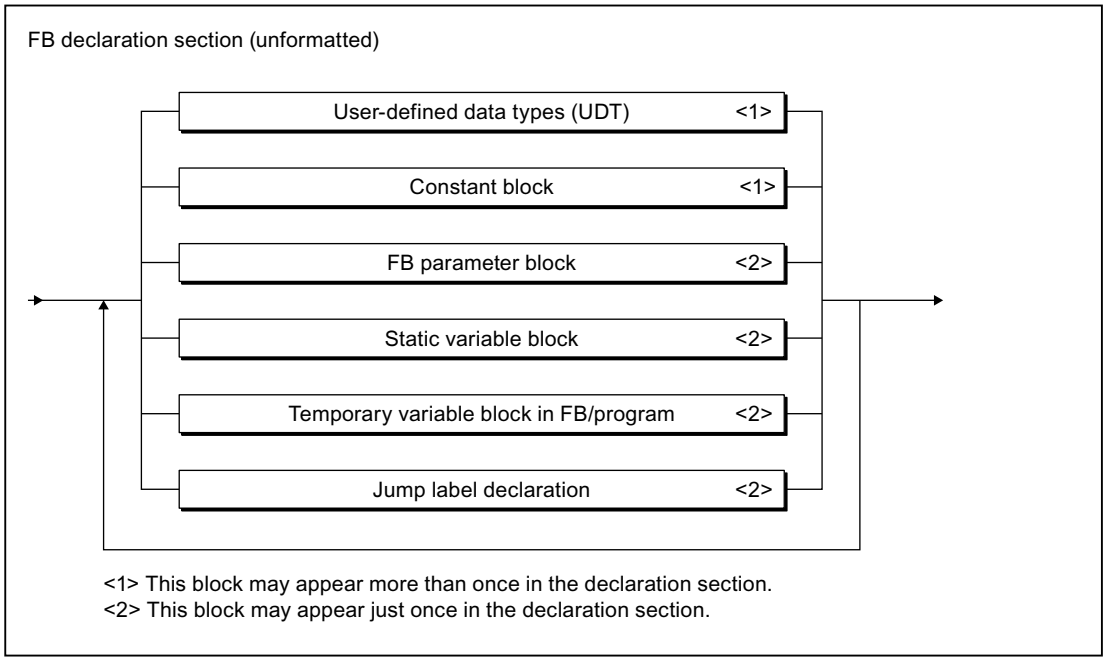



Figure A-38 FB declaration section

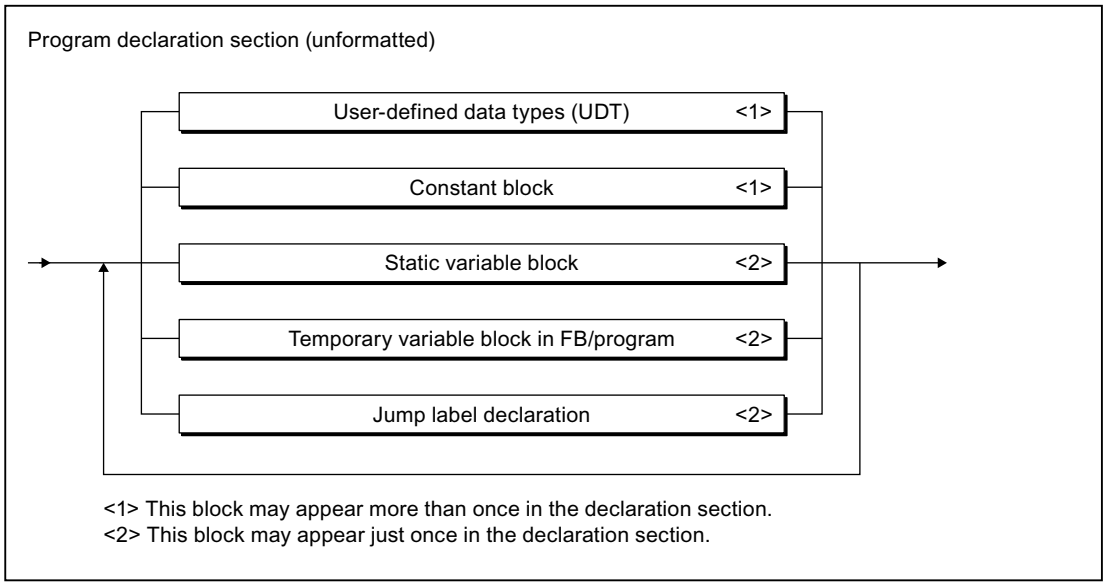



Figure A-39 Program declaration section

### A.1.3.8 Structure of the declaration blocks

## Constant blocks

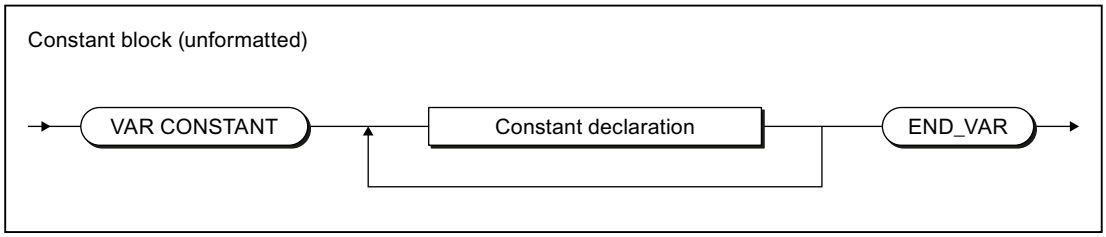



Figure A-40 Constant block

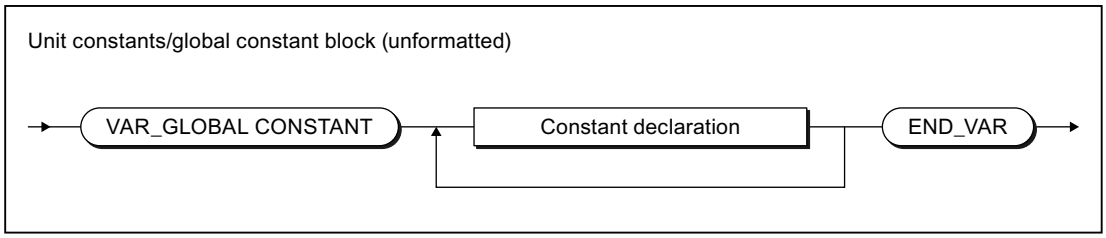



Figure A-41 Unit constants / global constant block

## Variable blocks

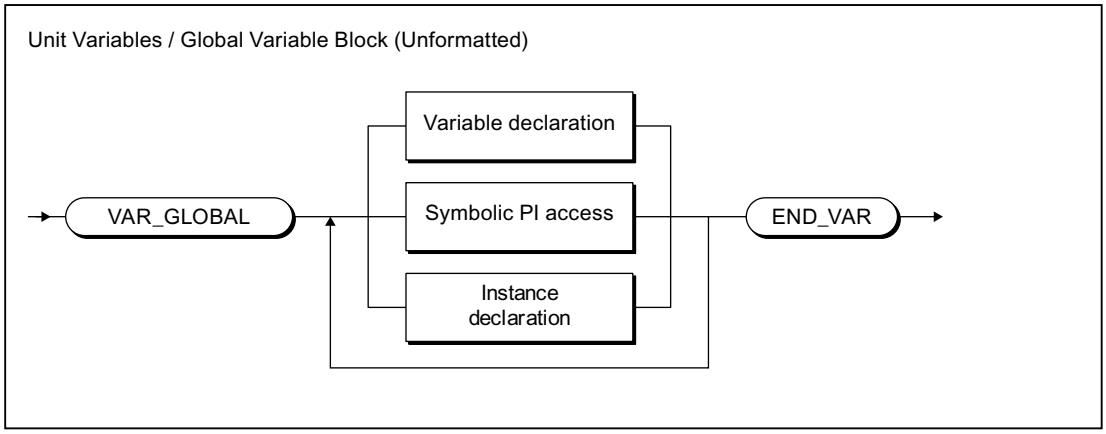



Figure A-42 Unit variables / global variable block

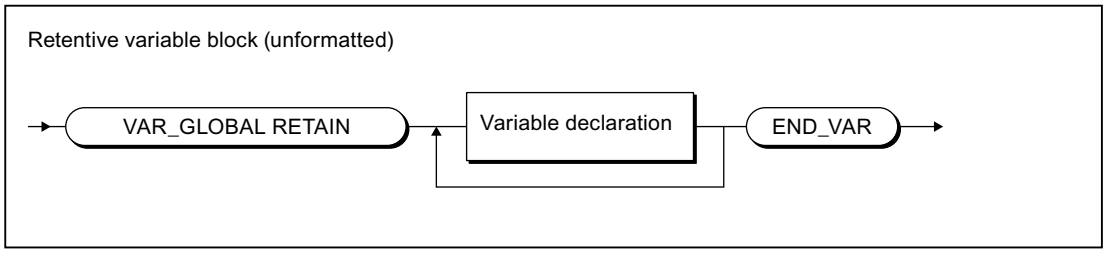



Figure A-43 Retentive variable block

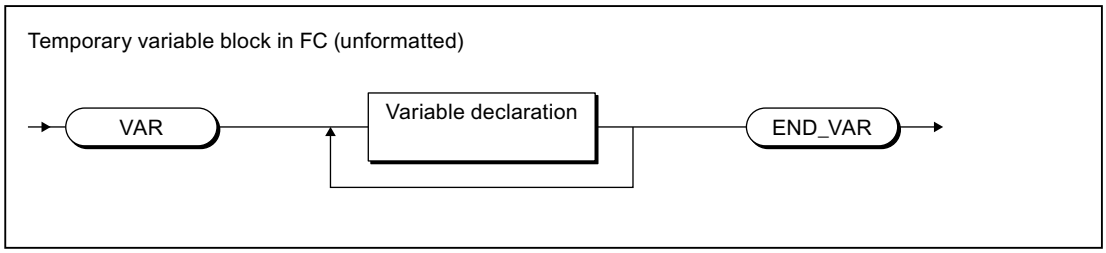



Figure A-44 Temporary variable block in FC

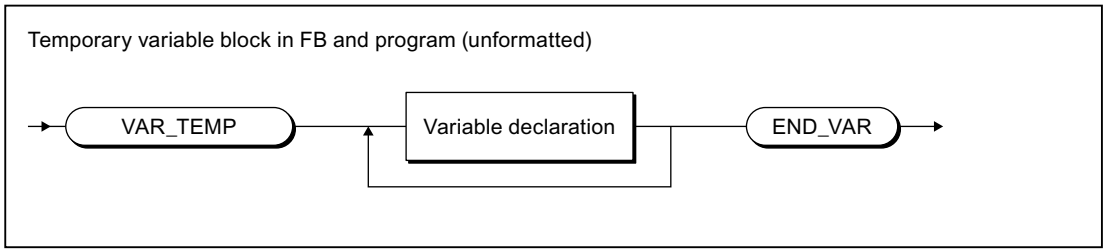



Figure A-45 Temporary variable block in the FB/program

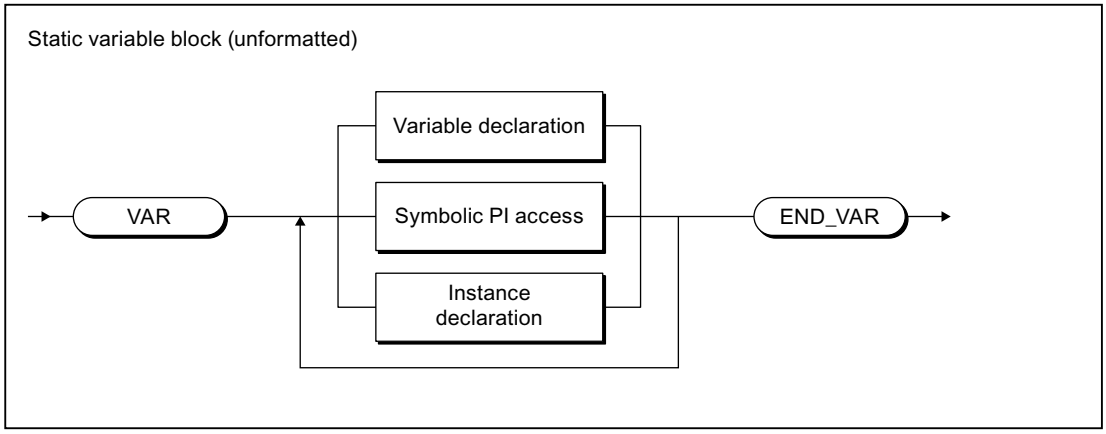



Figure A-46 Static variable block

## Parameter fields

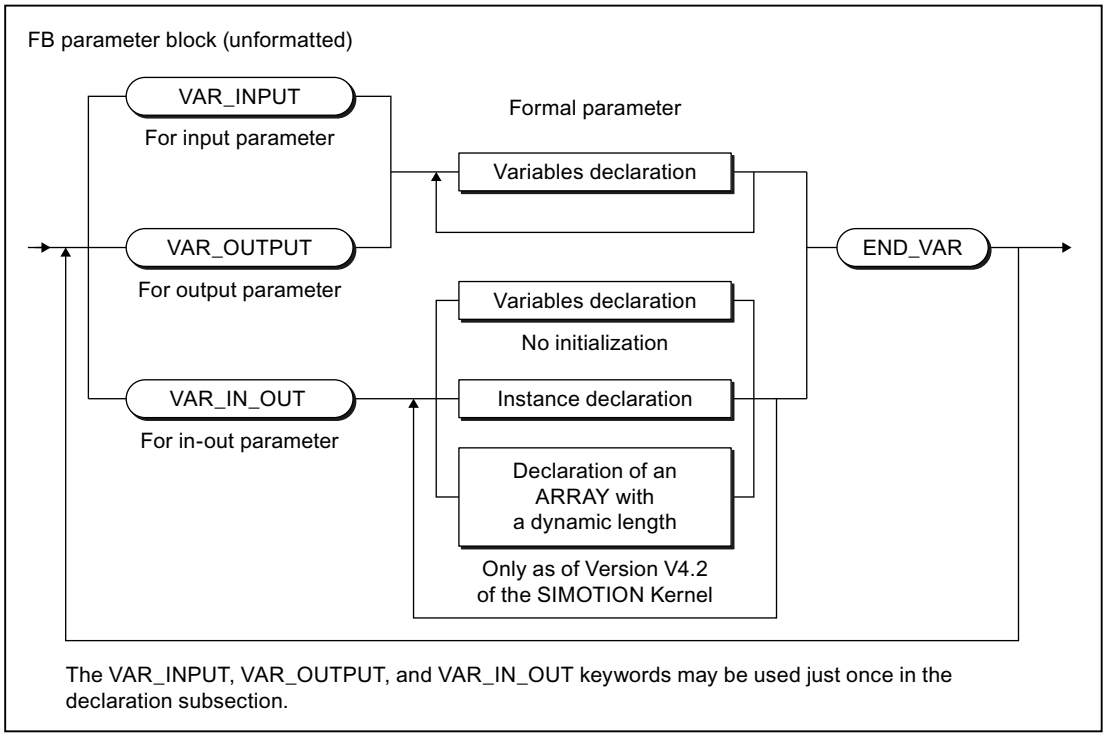



Figure A-47 FB parameter block

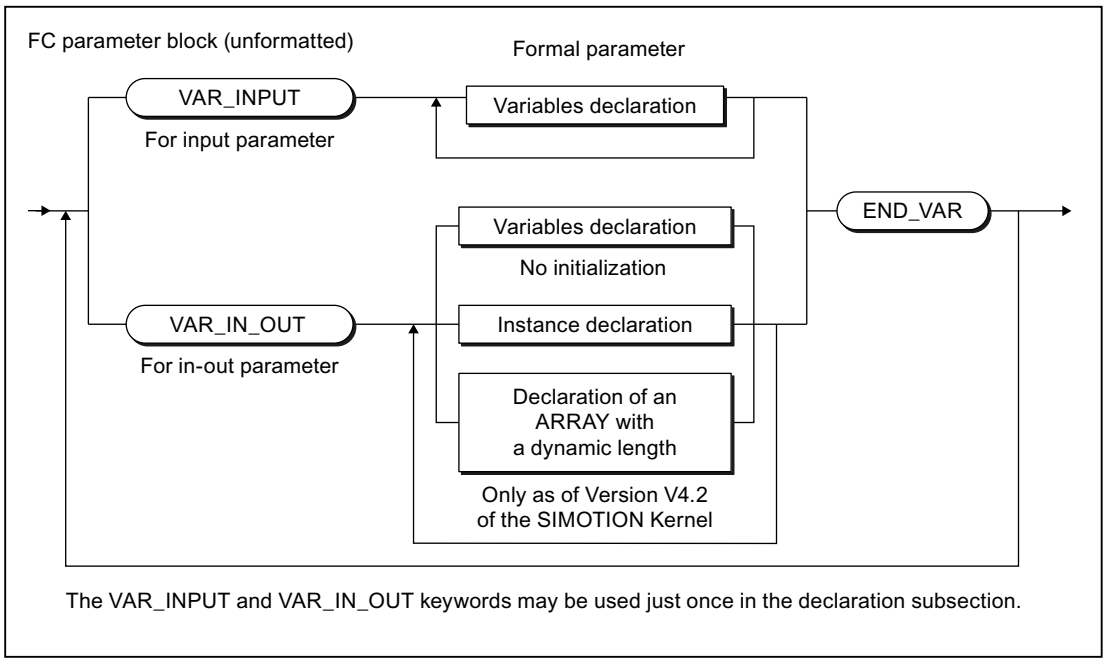



Figure A-48 FC parameter block

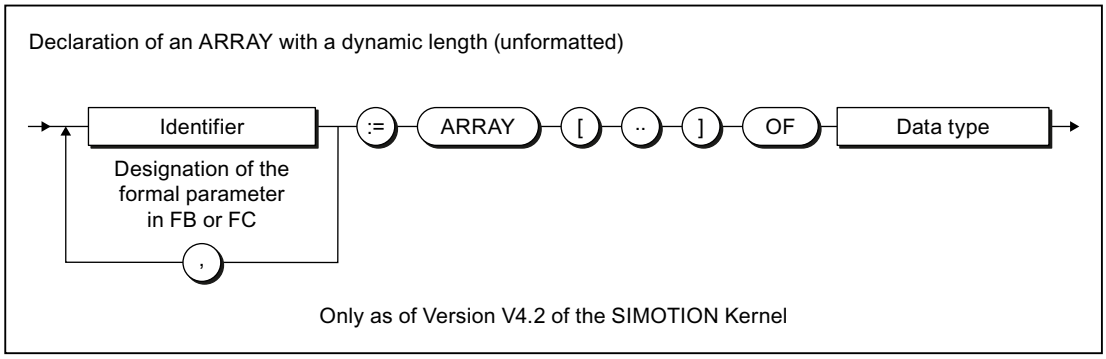



Figure A-49 Declaration of an ARRAY with a dynamic length

## Jump labels

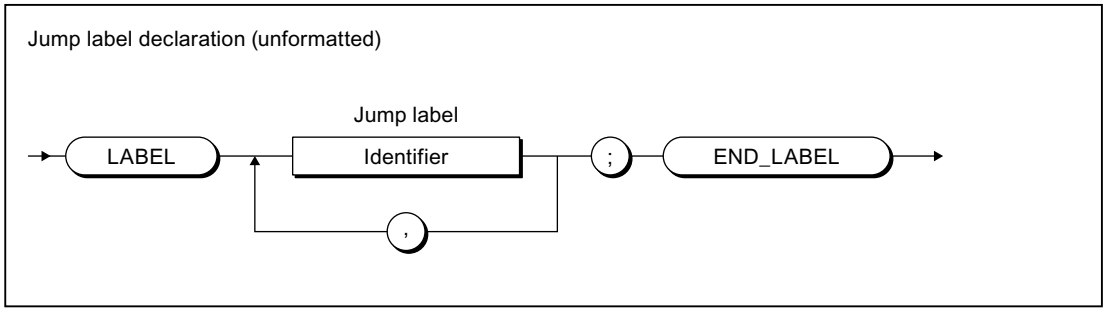



Figure A-50 Jump label declaration

## Declarations

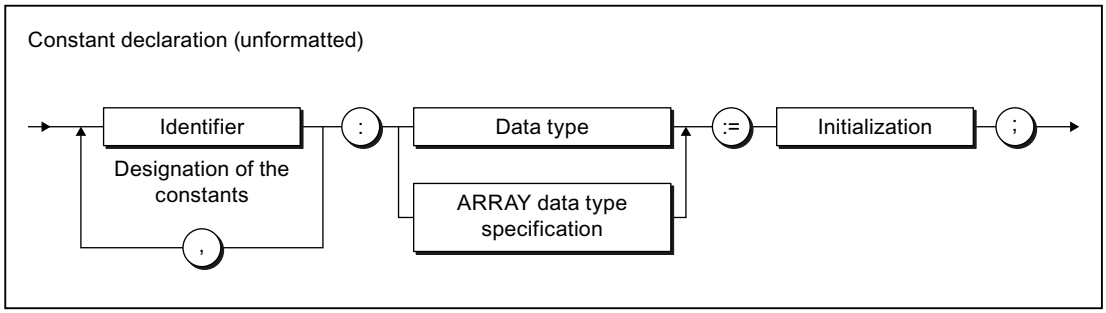



Figure A-51 Constant declaration

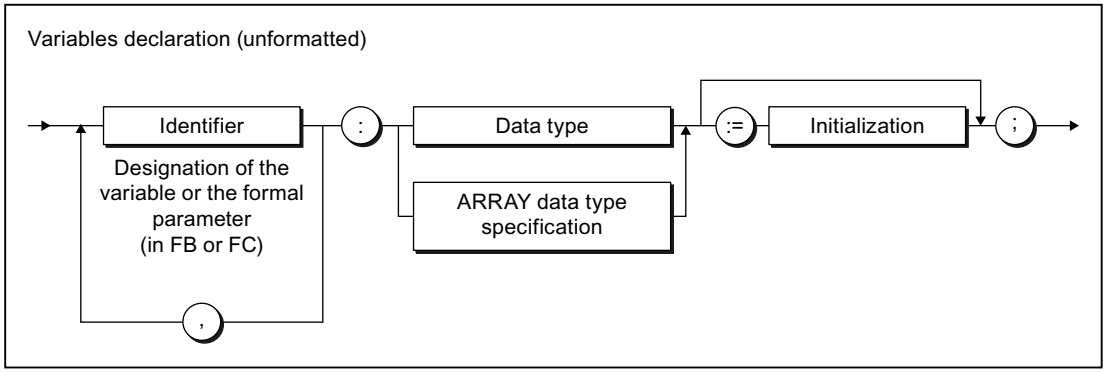



Figure A-52 Variable declaration

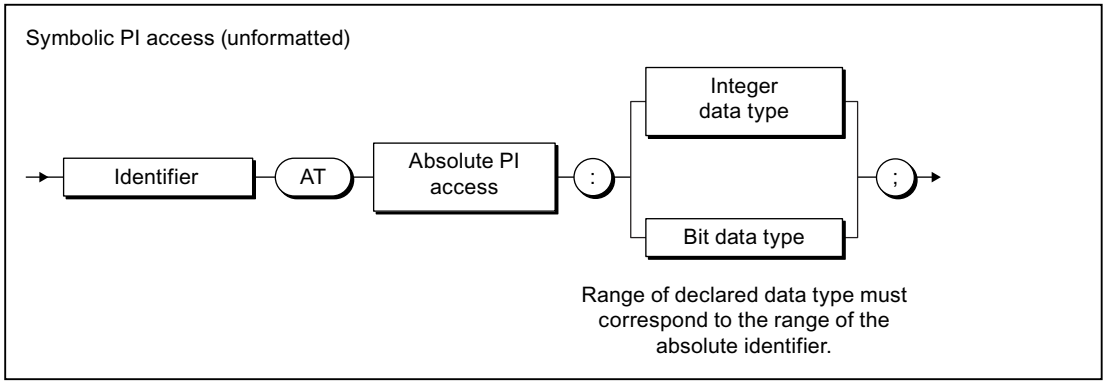



Figure A-53 Symbolic PI access

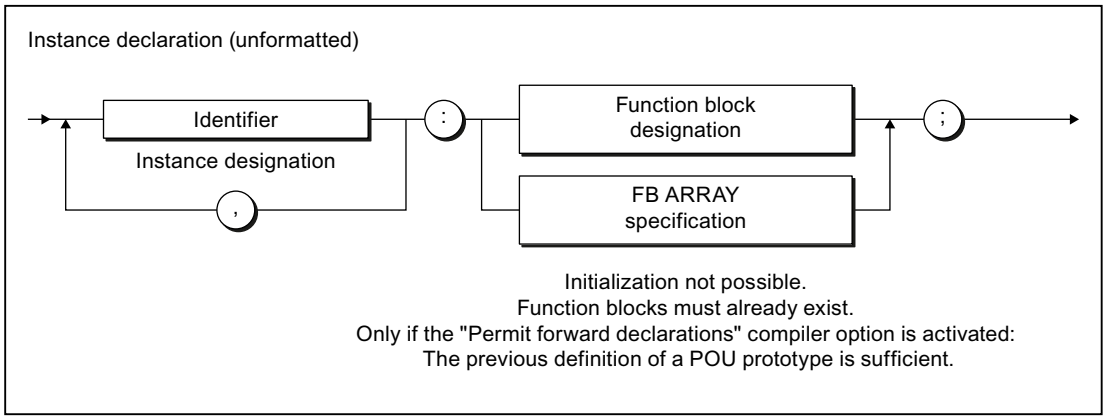



Figure A-54 Instance declaration

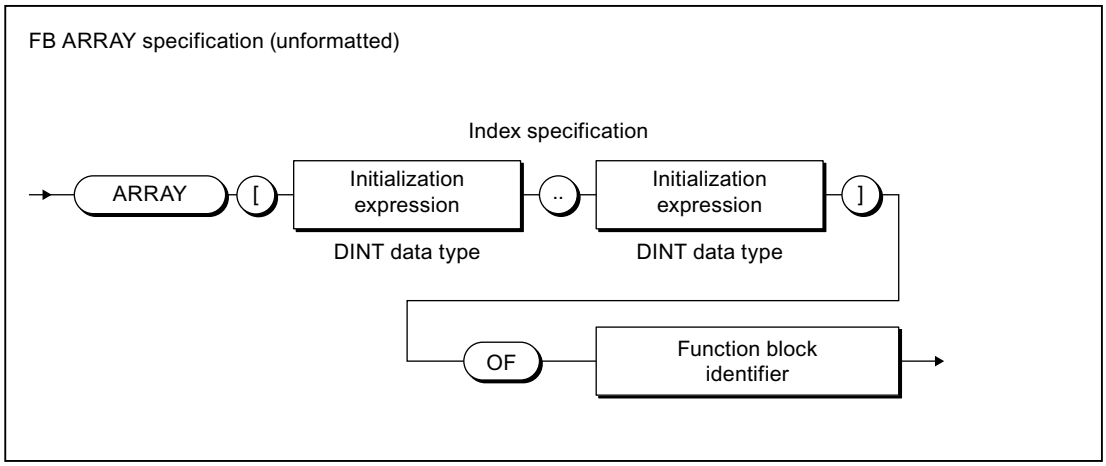



Figure A-55 FB ARRAY specification

## Initialization

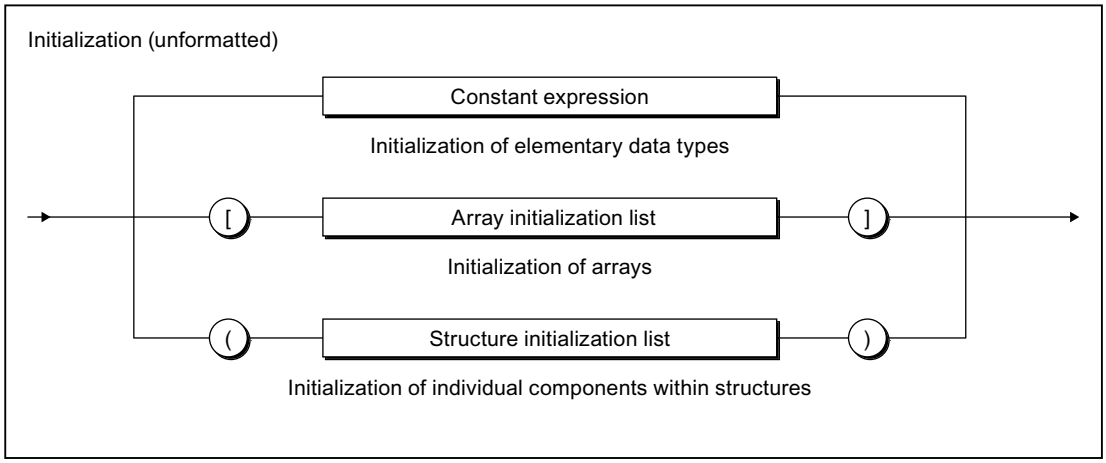



Figure A-56 Initialization

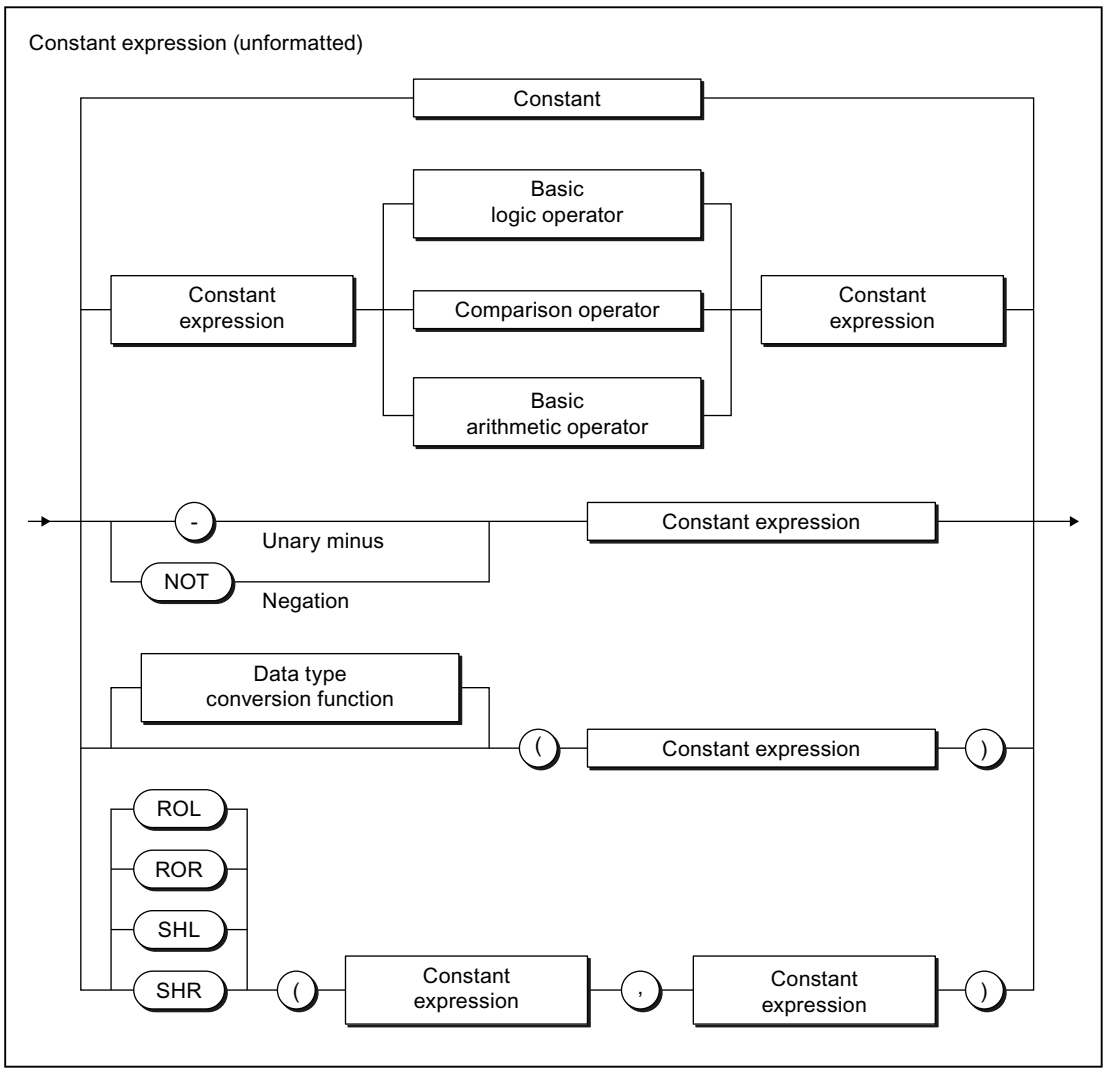



Figure A-57 Constant expression

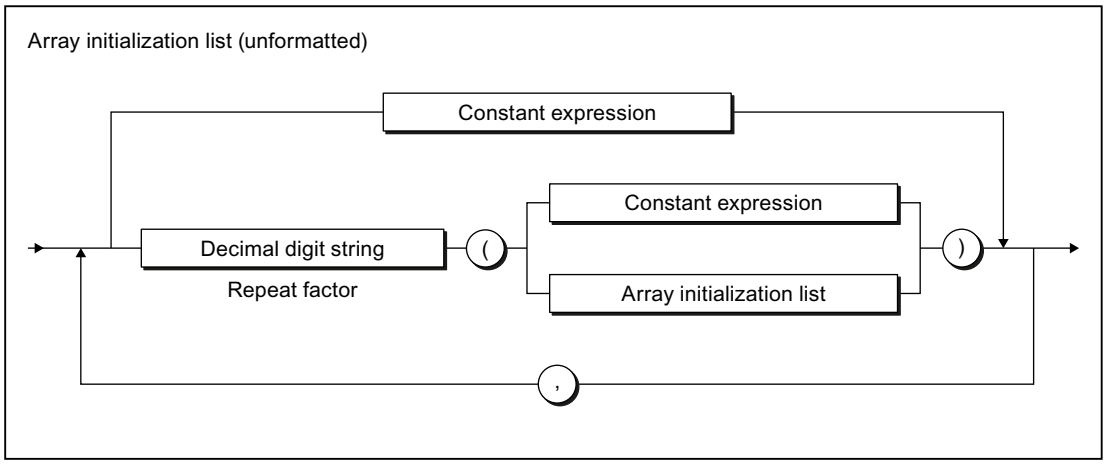



Figure A-58 Array initialization list

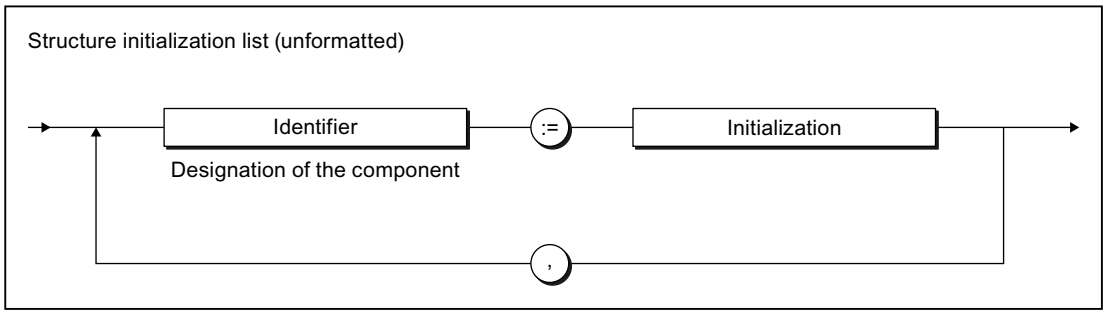



Figure A-59 Structure initialization list

### A.1.3.9 Data types

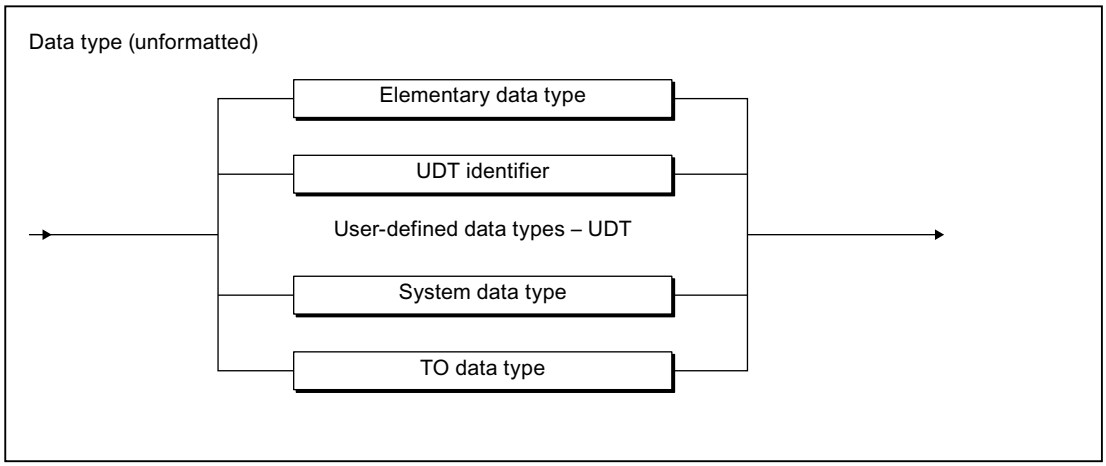



Figure A-60 Data type

## Elementary data types

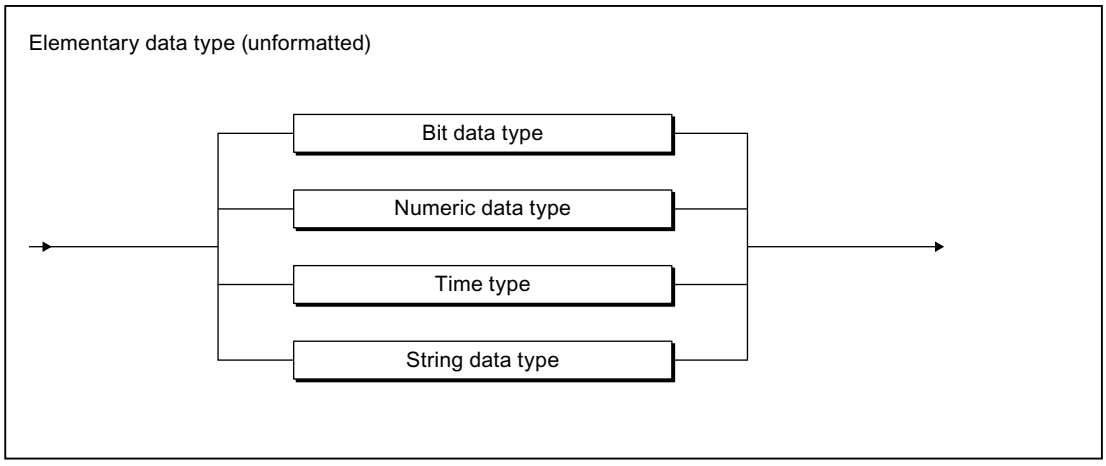



Figure A-61 Elementary data type

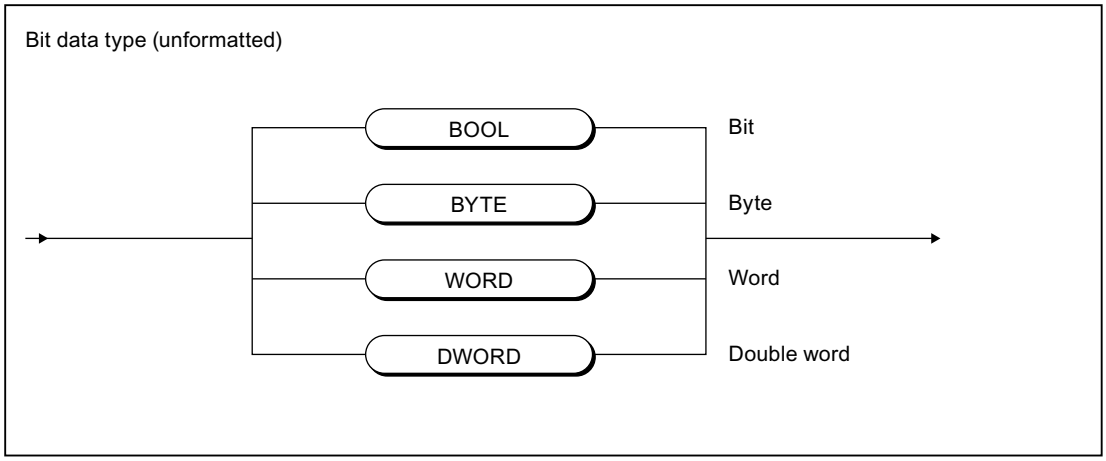



Figure A-62 Bit data type

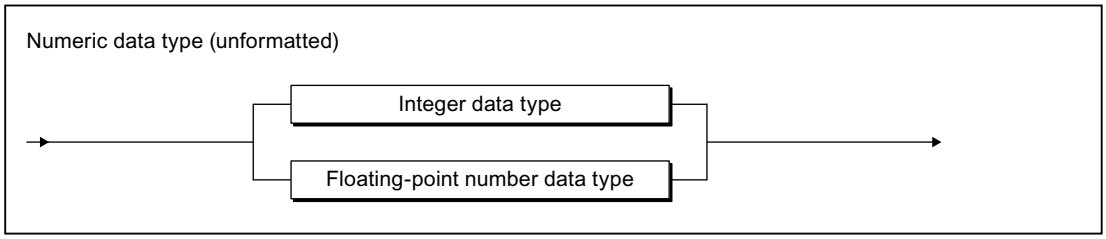



Figure A-63 Numeric data type

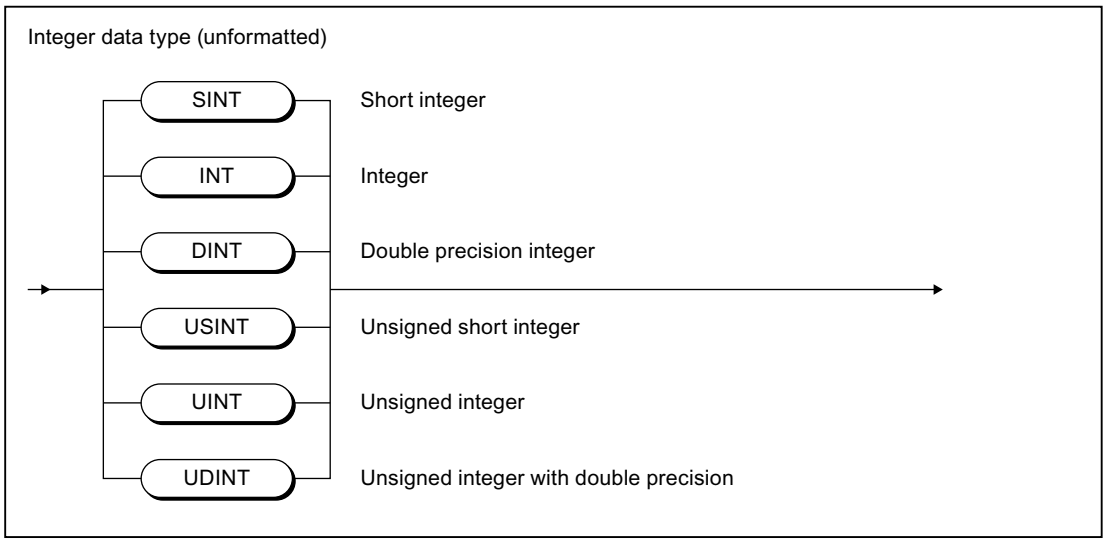



Figure A-64 Integer data type

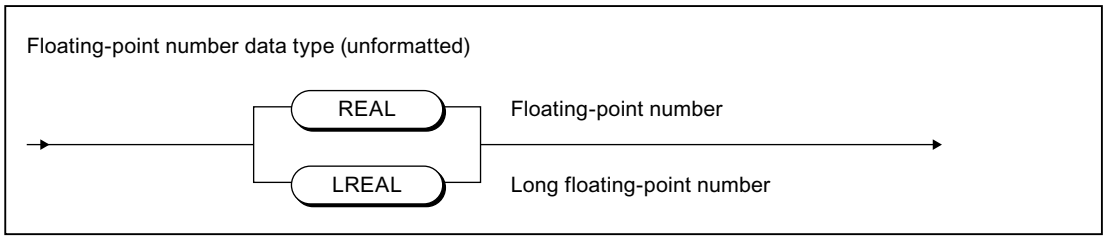



Figure A-65 Floating-point number data type

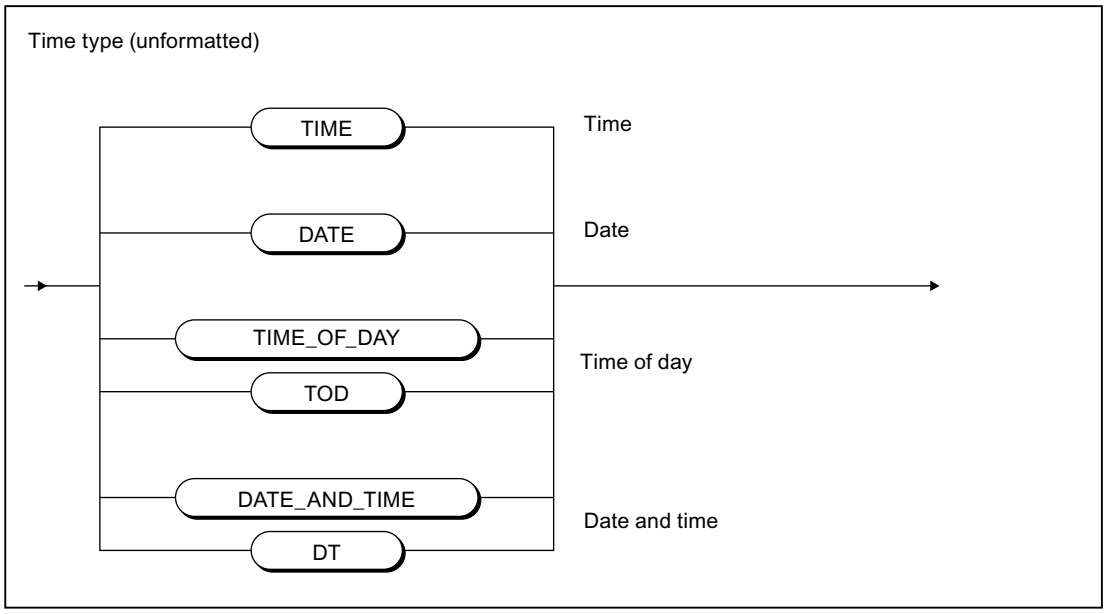



Figure A-66 Time data type

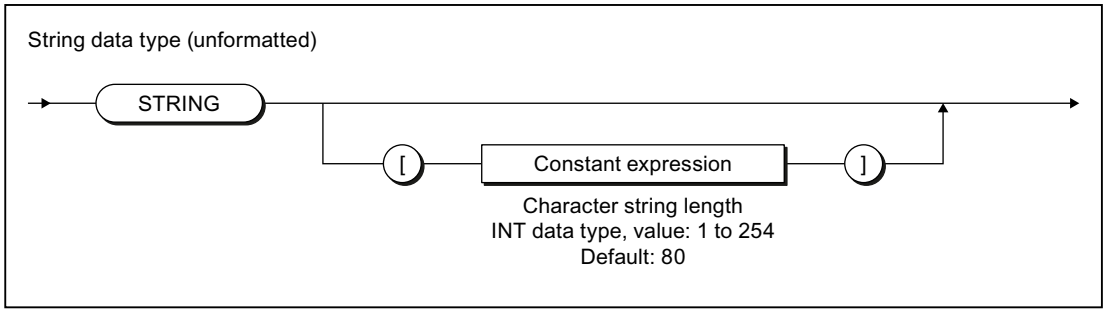



Figure A-67 String data type

## User-defined data types

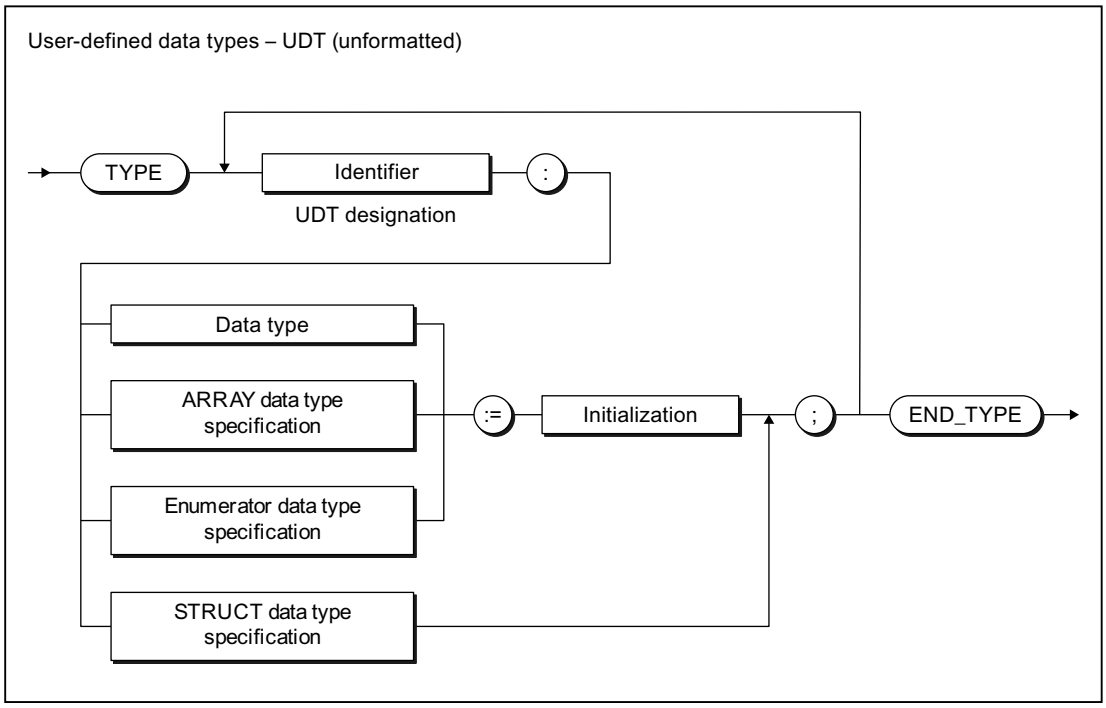



Figure A-68 User-defined data type

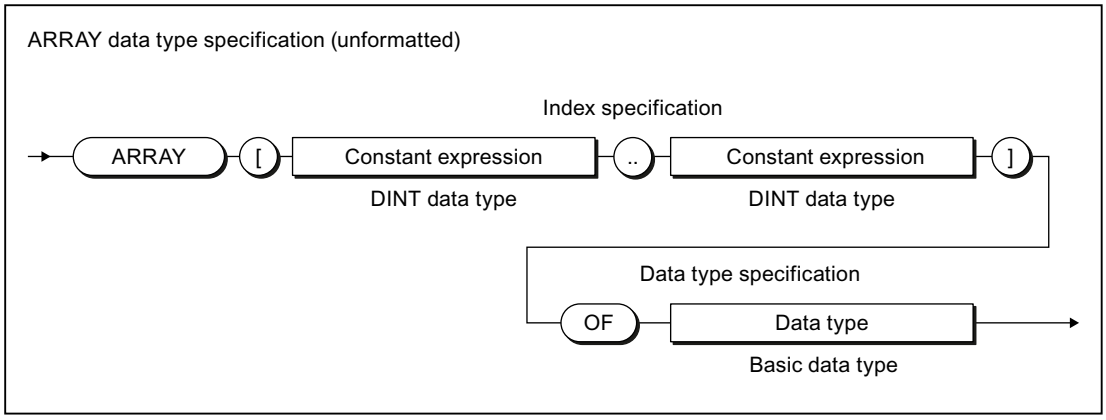



Figure A-69 ARRAY data type specification

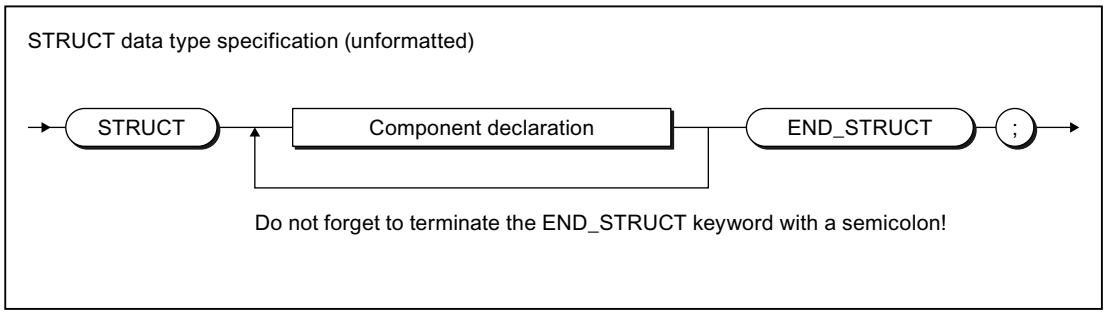



Figure A-70 STRUCT data type specification

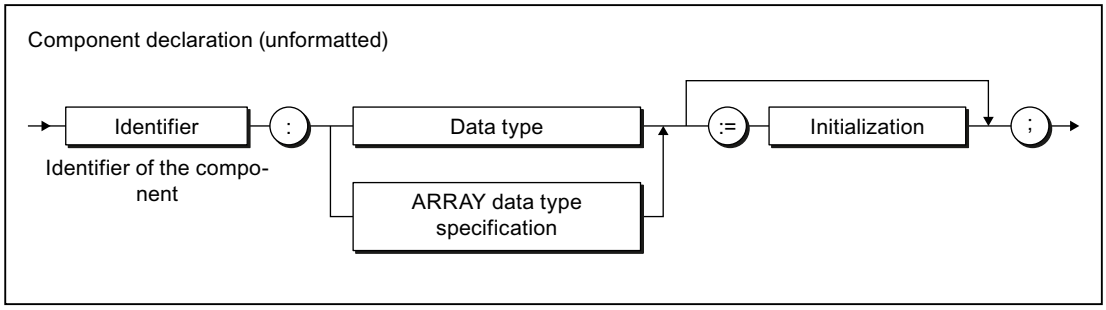



Figure A-71 Component declaration

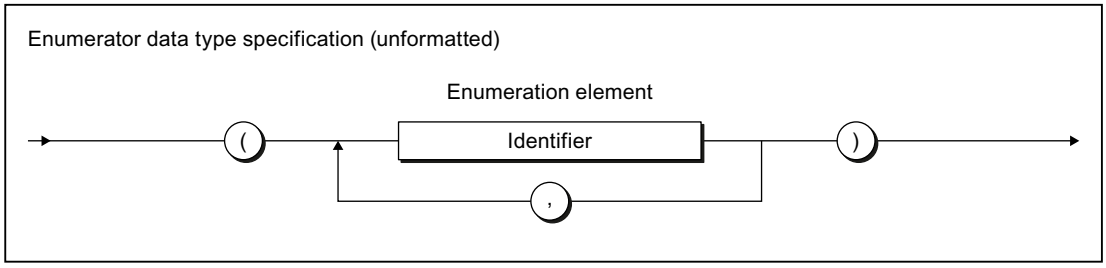



Figure A-72 Enumerator data type specification

### A.1.3.10 Statement section

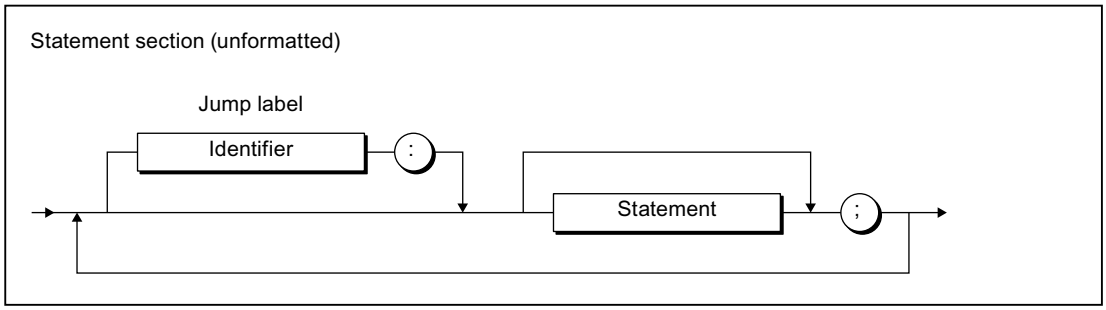



Figure A-73 Statement section

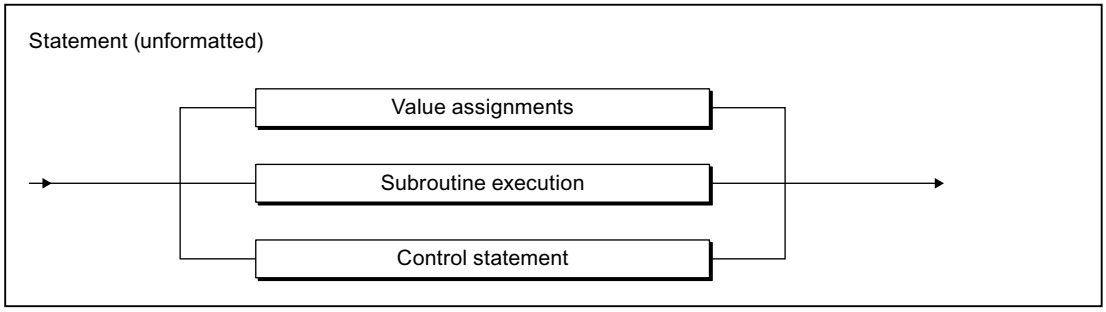



Figure A-74 Statement

### A.1.3.11 Value assignments and operations

#### Value assignment and expression

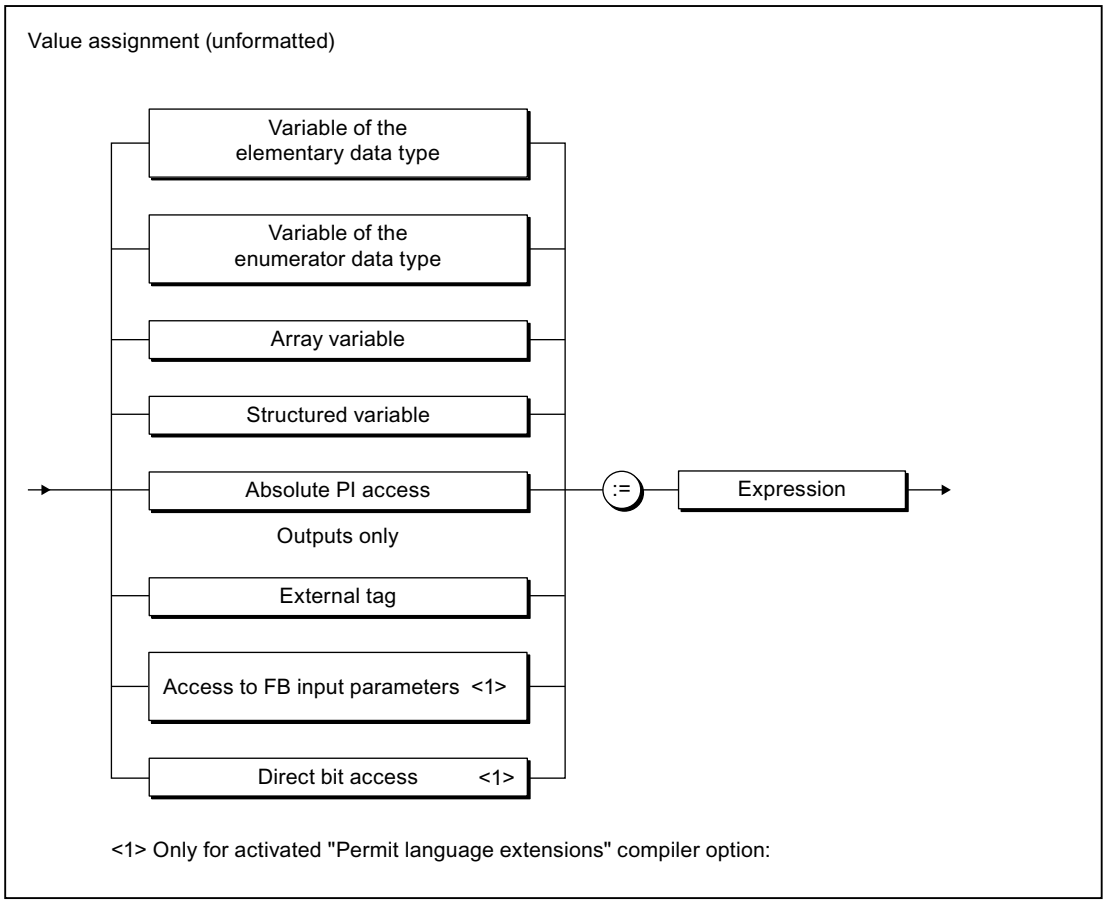



Figure A-75 Value assignments

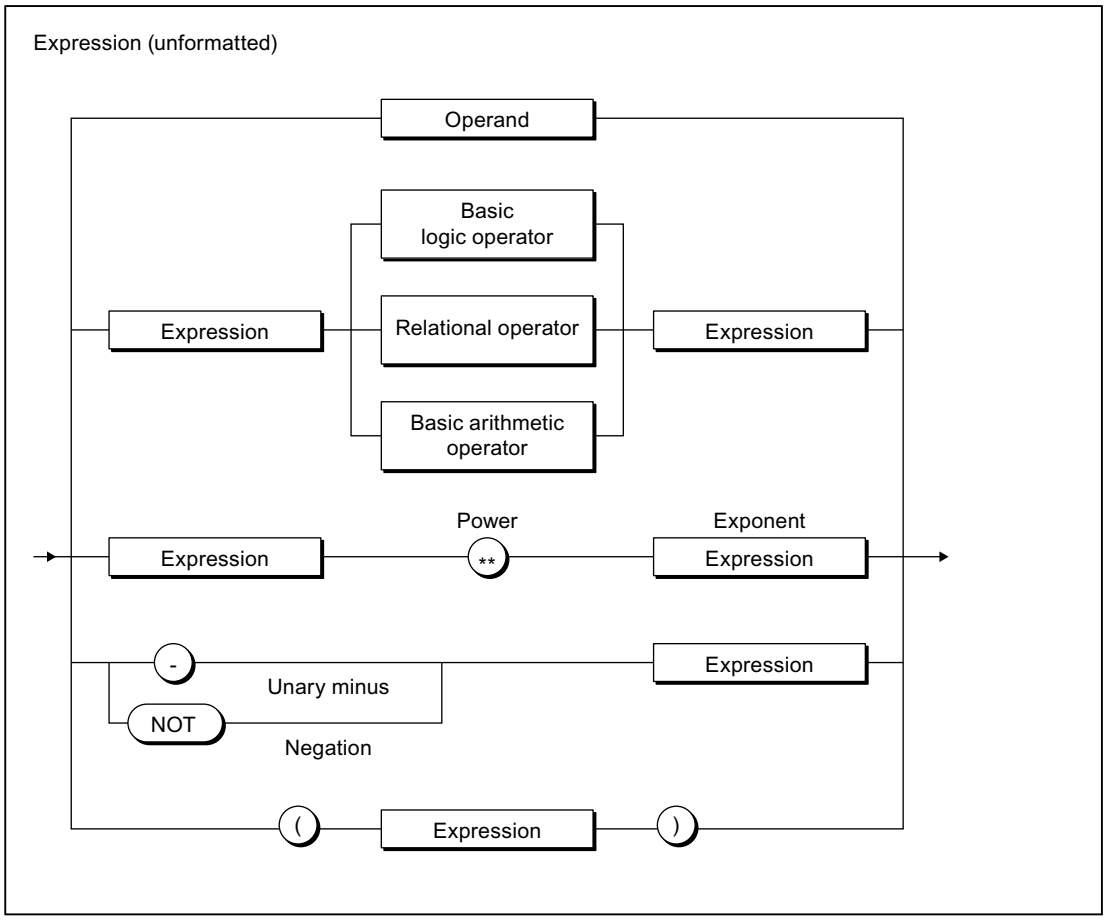



Figure A-76 Expression

## Operands

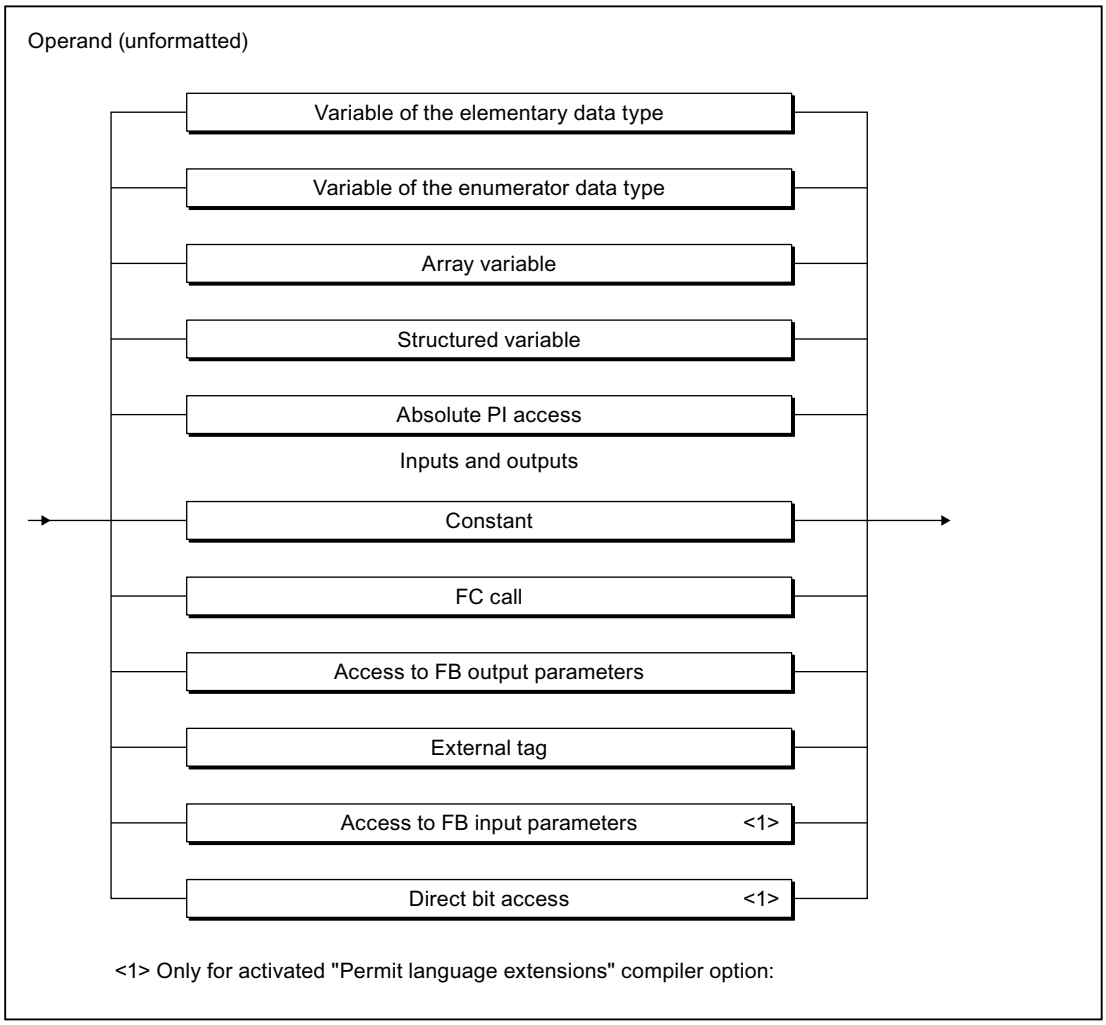



Figure A-77 Operand

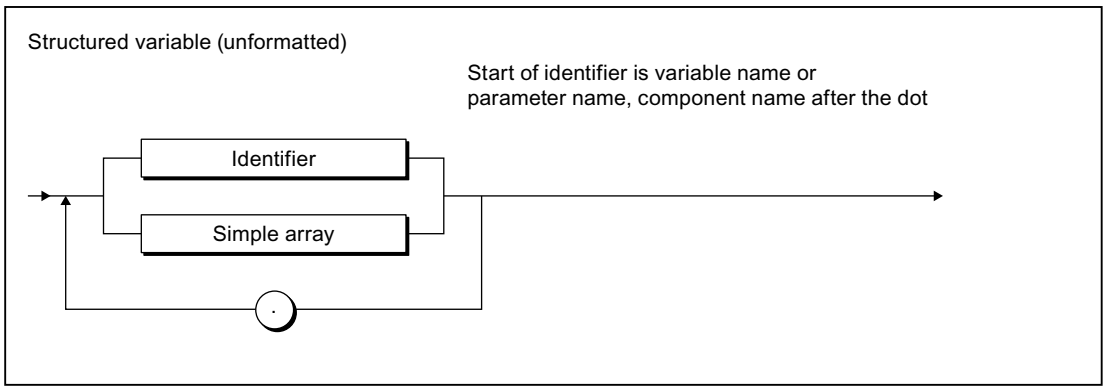



Figure A-78 Structured variable

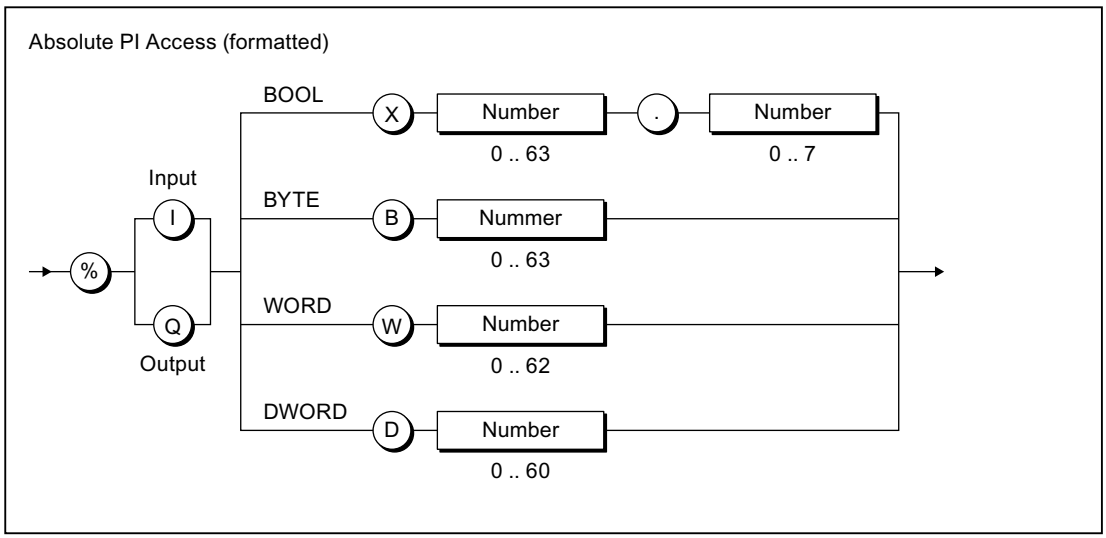



Figure A-79 Absolute PI access

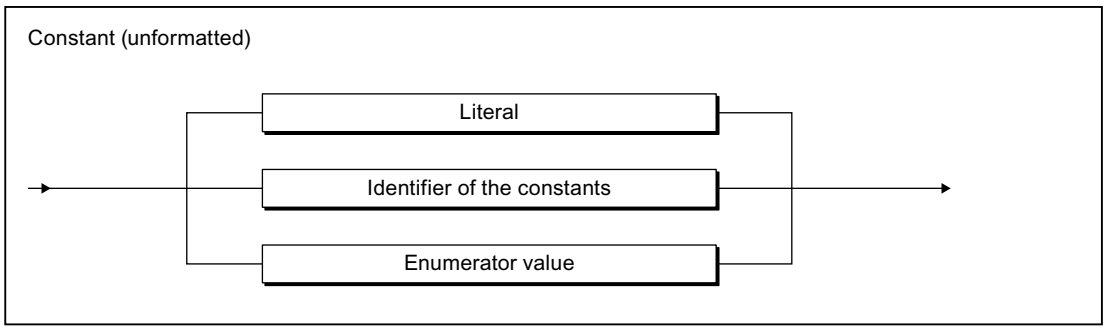



Figure A-80 Constant

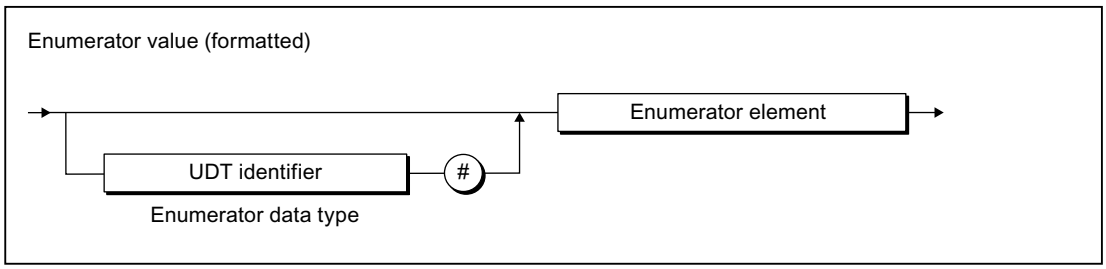



Figure A-81 Enumerator value

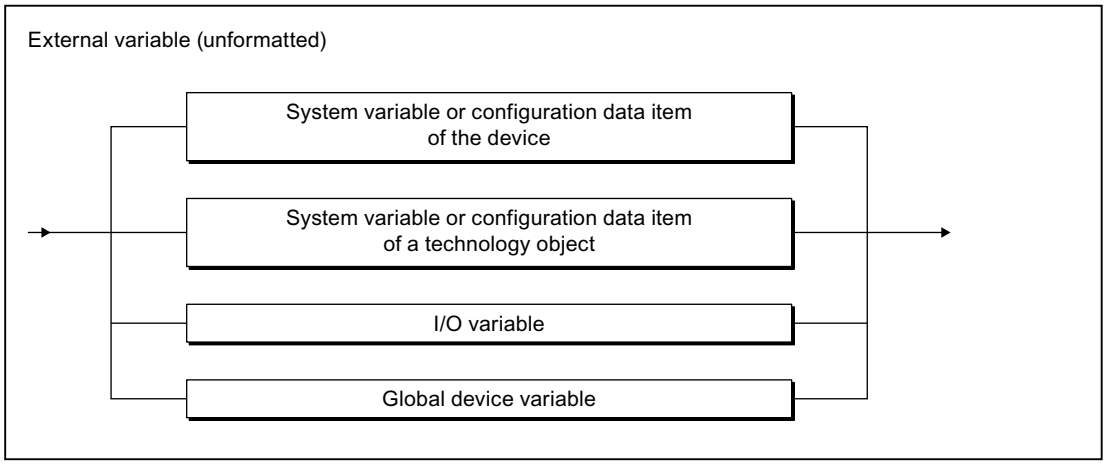



Figure A-82 External tag

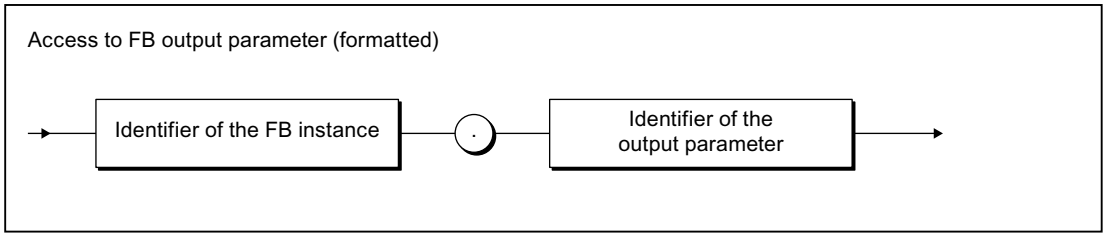



Figure A-83 Access to FB output parameters

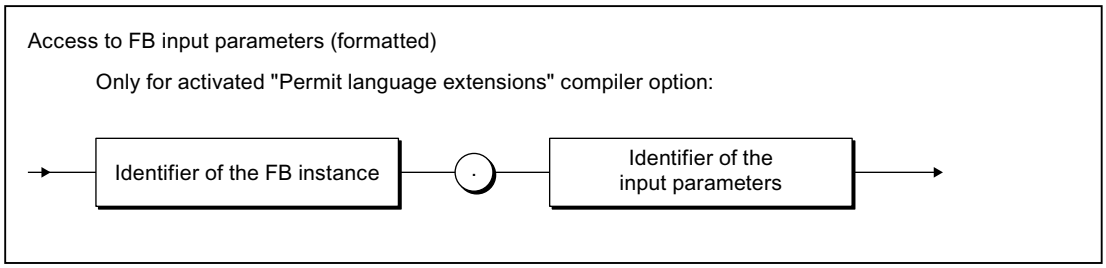



Figure A-84 Access to FB input parameters

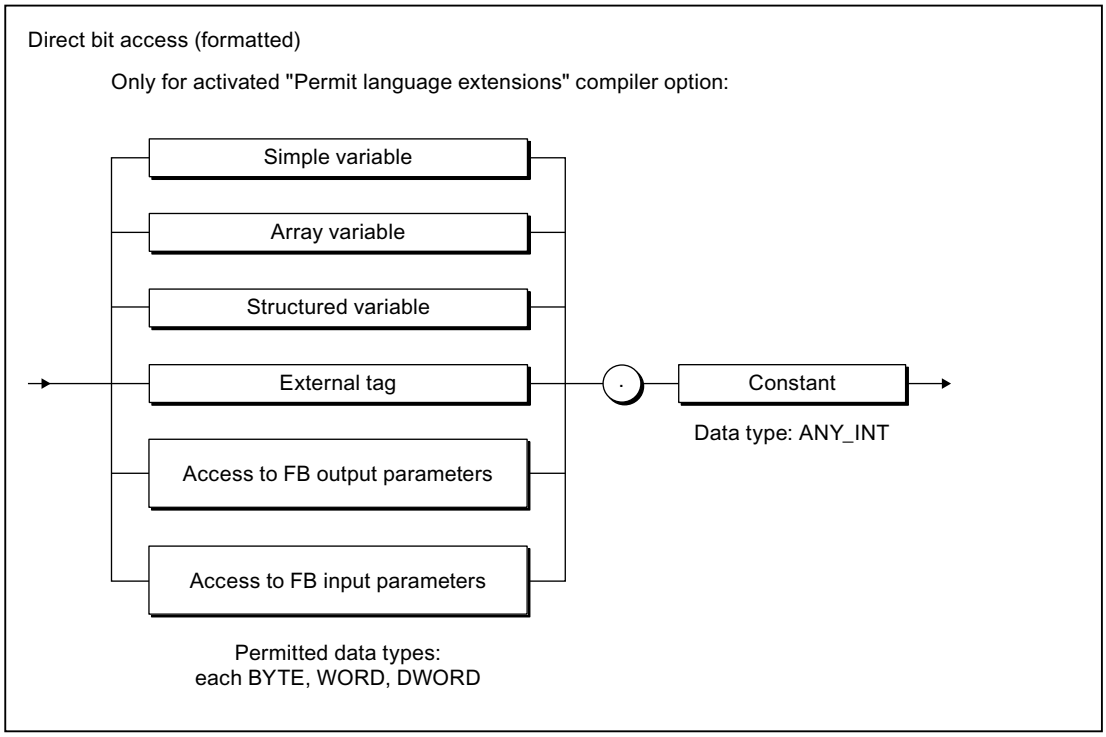



Figure A-85 Bit access

## Operators

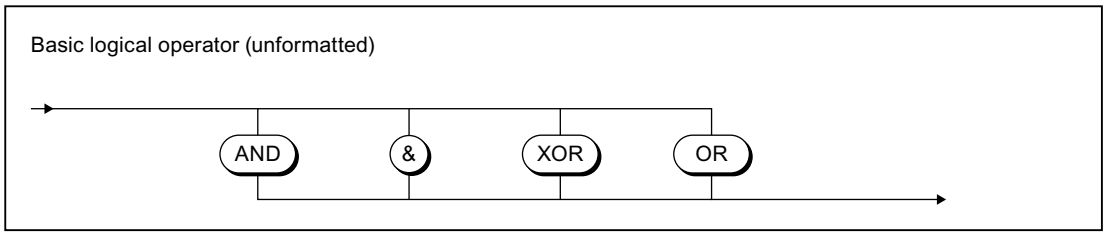



Figure A-86 Basic logic operator

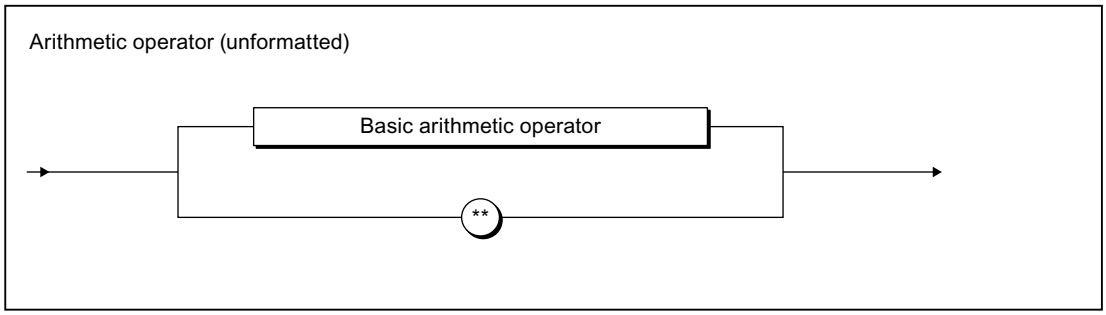



Figure A-87 Arithmetic operator

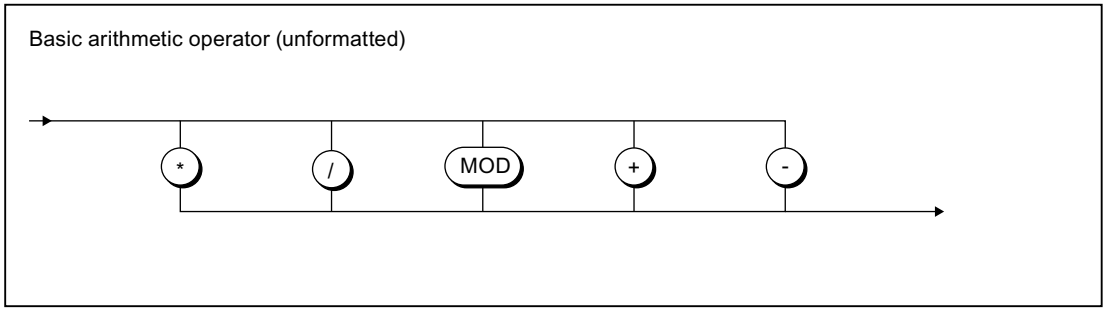



Figure A-88 Basic arithmetic operator

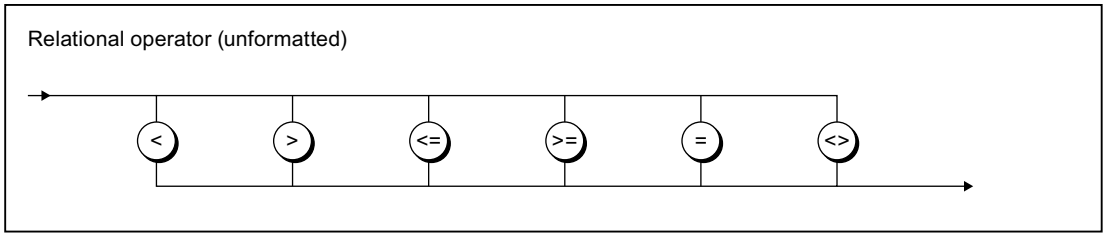



Figure A-89 Relational operators

### A.1.3.12 Call of functions and function block calls

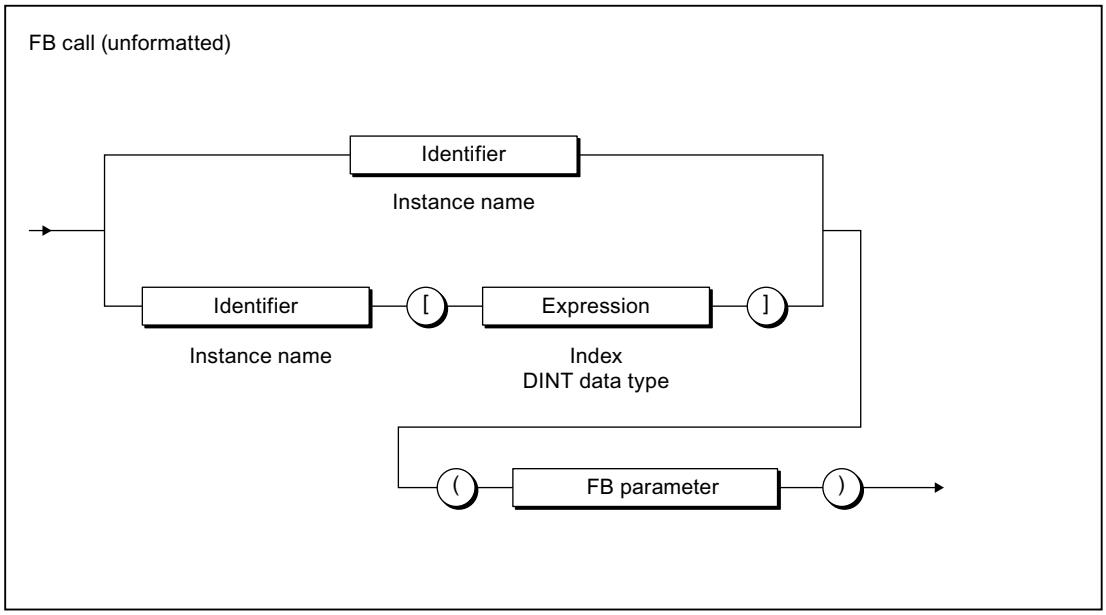



Figure A-90 FB call

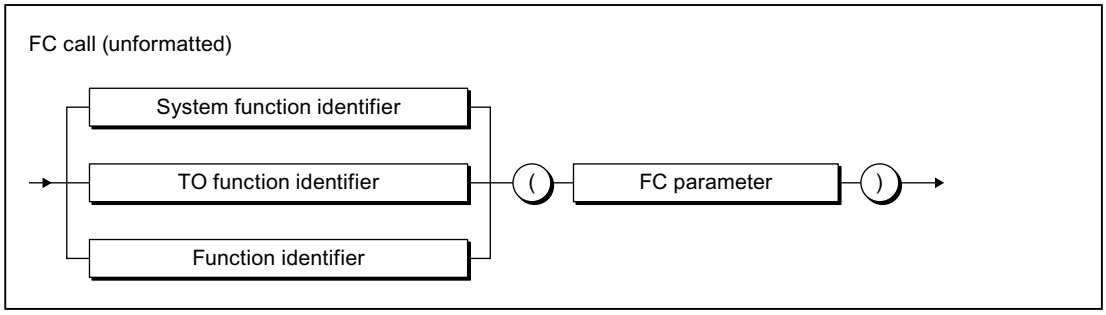



Figure A-91 FC call

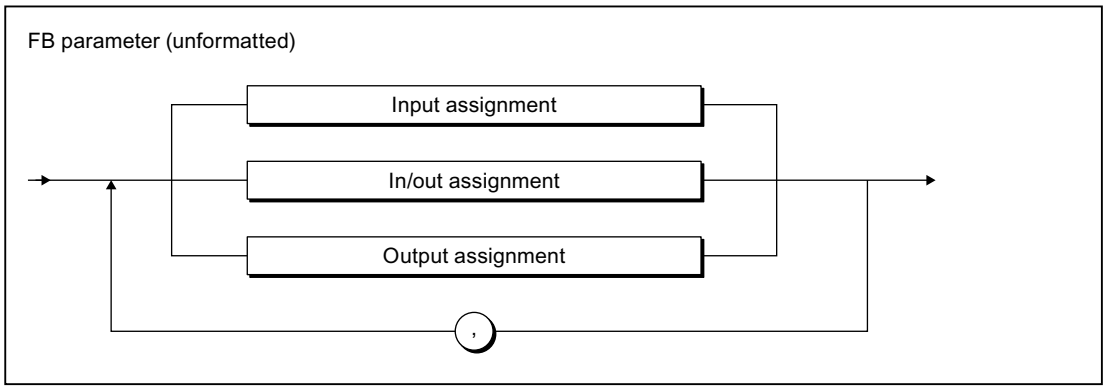



Figure A-92 FB parameter

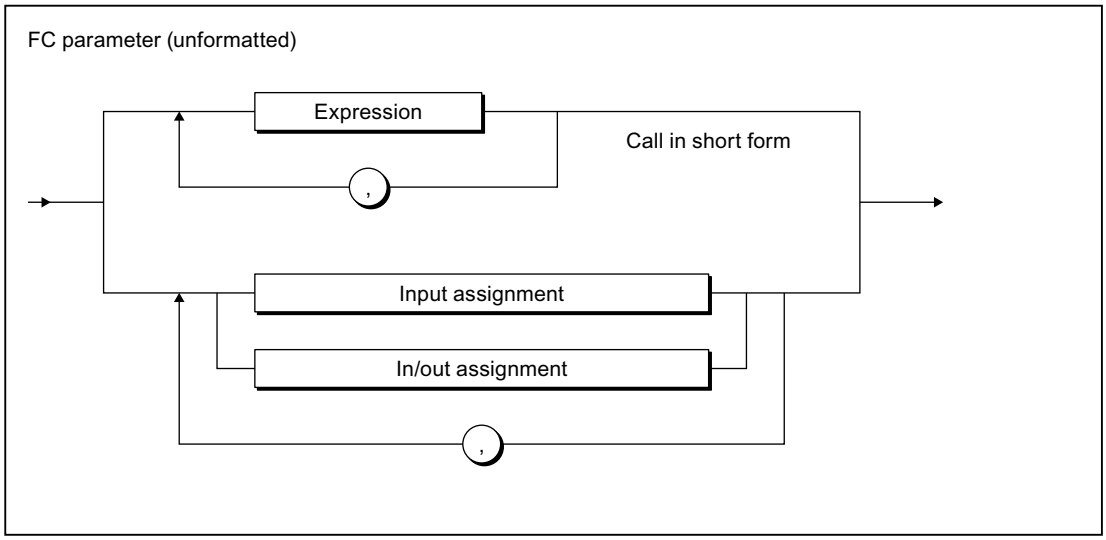



Figure A-93 FC parameter

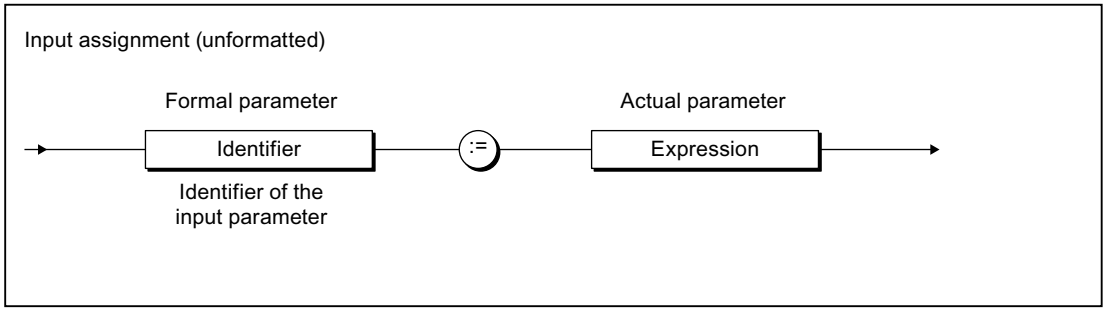



Figure A-94 Input assignment

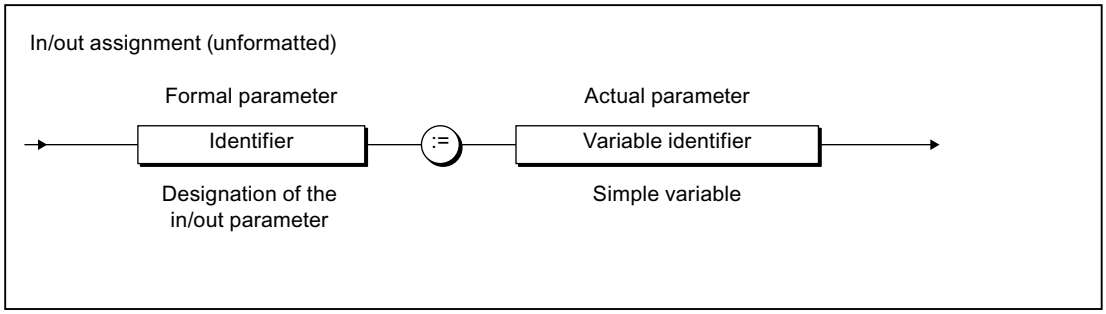



Figure A-95 In/out assignment

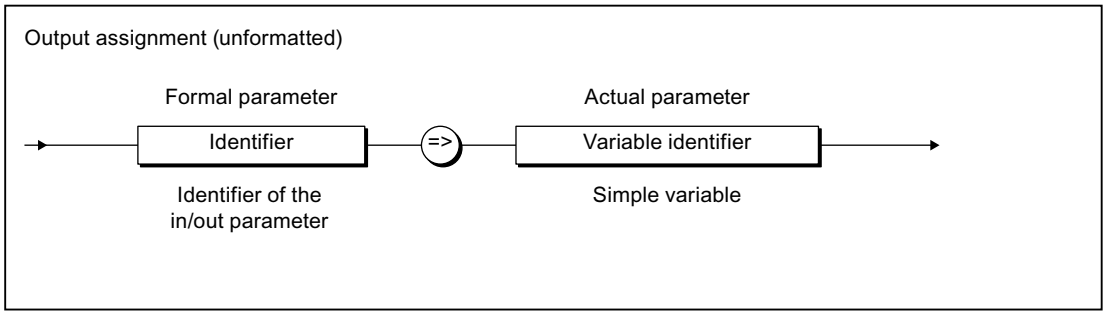



Figure A-96 Output assignment

### A.1.3.13 Control statements

## Branches

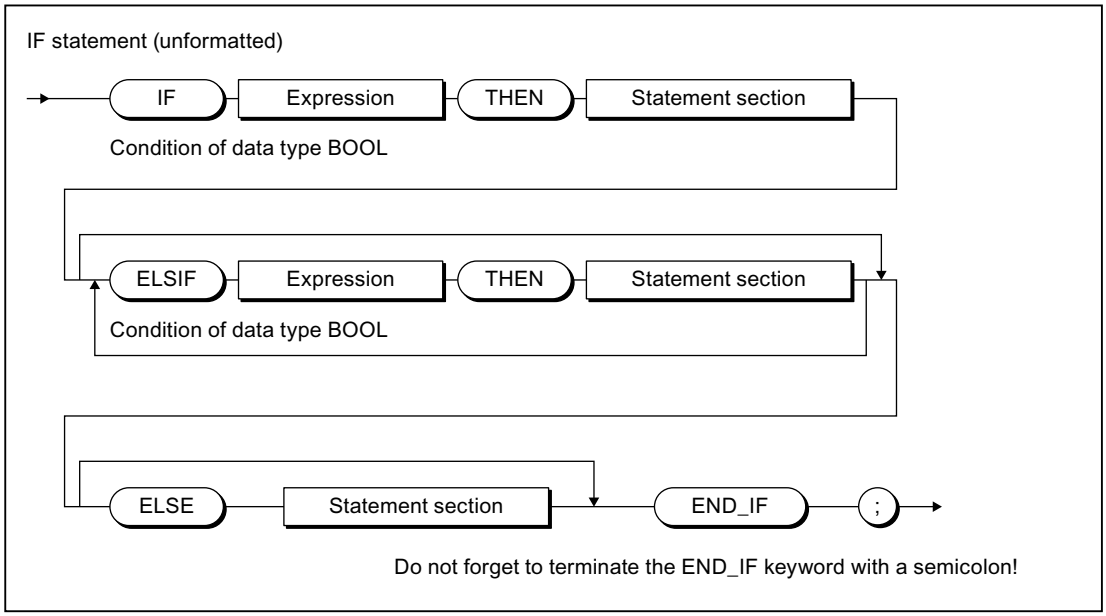



Figure A-97 IF statement

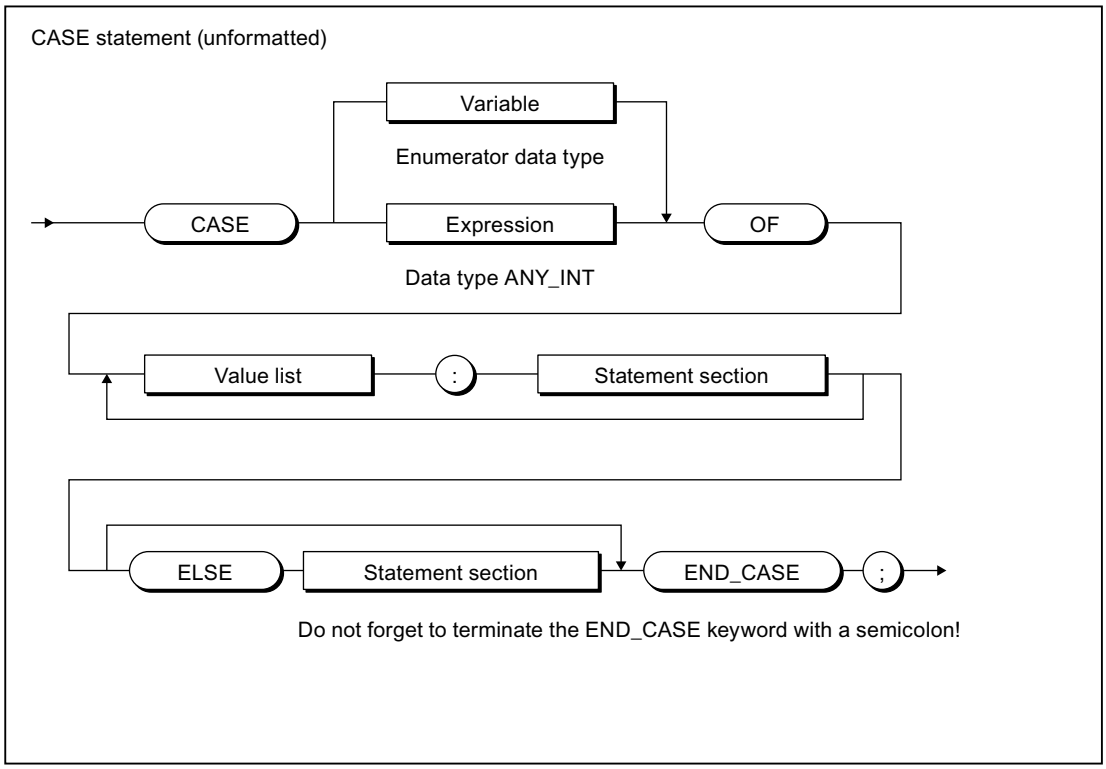



Figure A-98 CASE statement

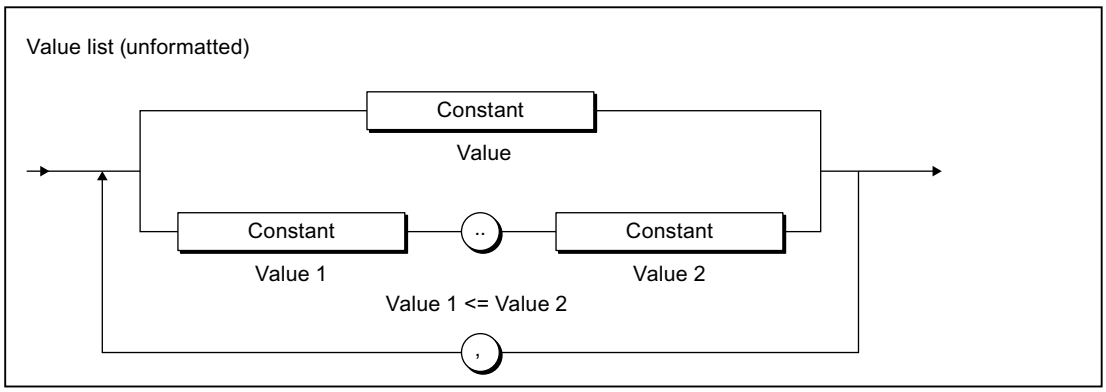



Figure A-99 Value list

## Repetition statements and jump statements

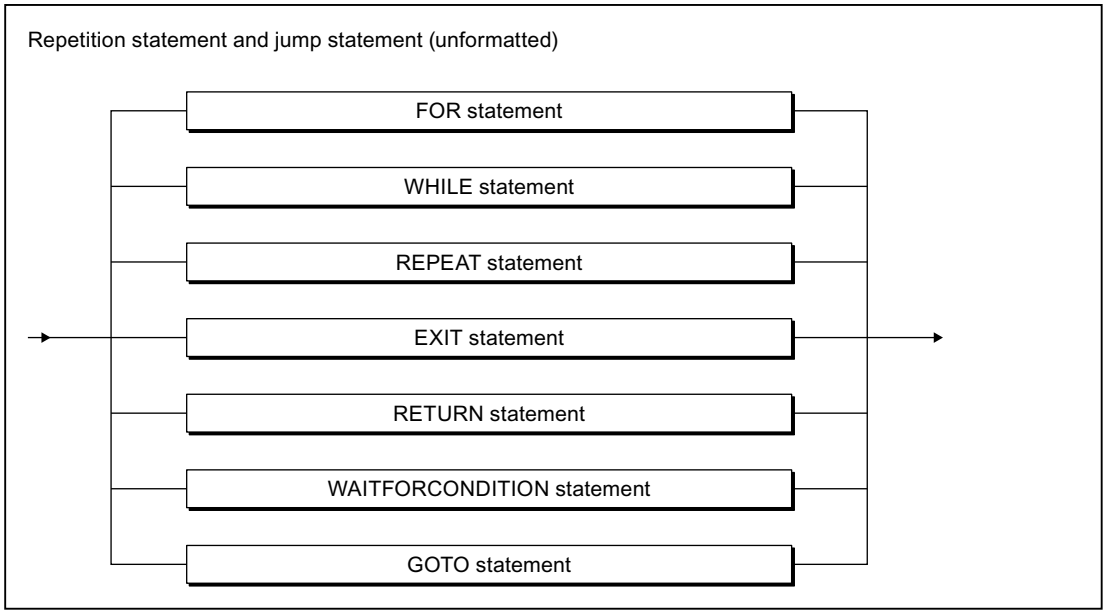



Figure A-100 Repetition statement and jump statements

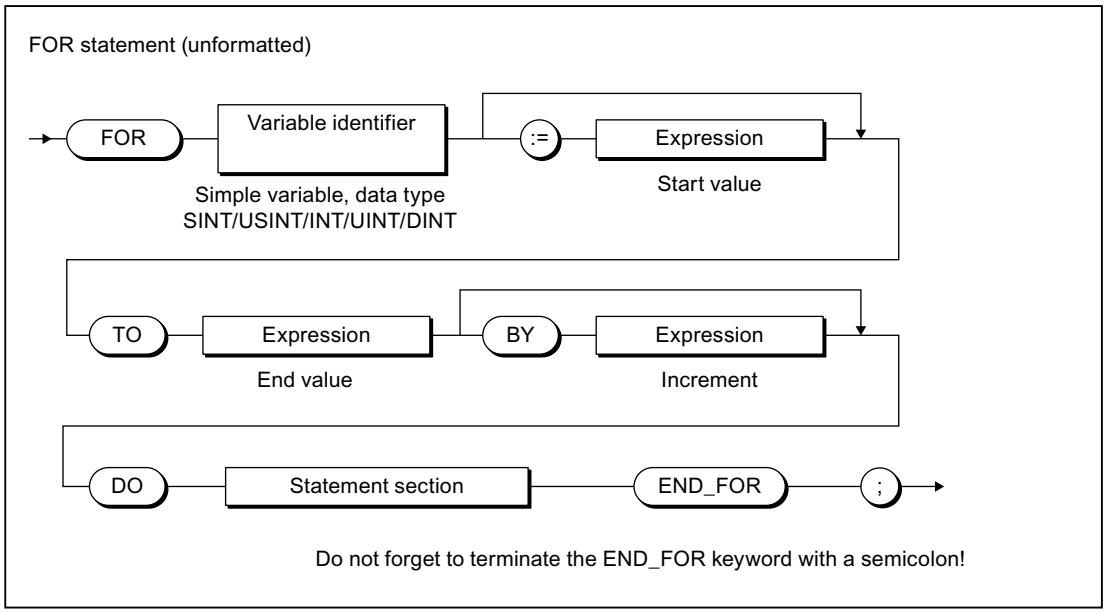



Figure A-101 FOR statement

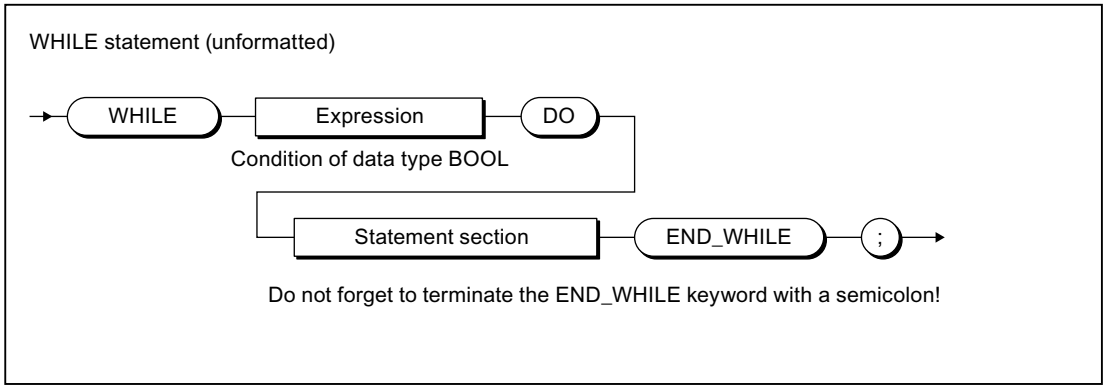



Figure A-102 WHILE statement

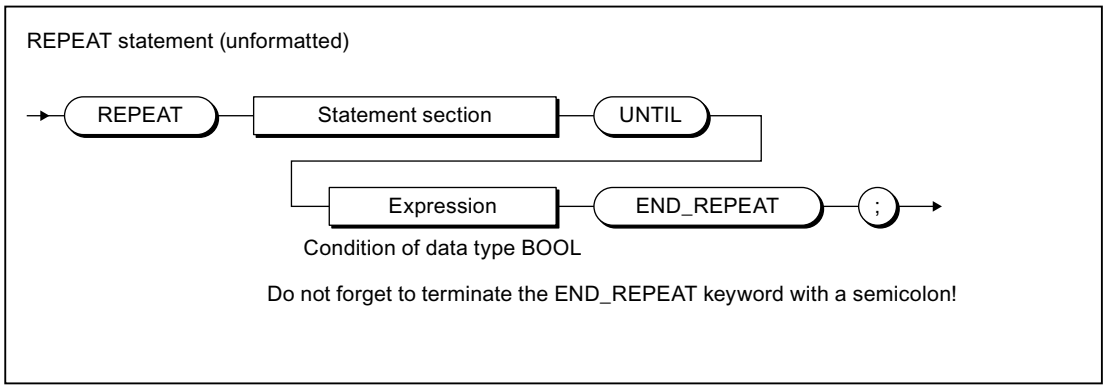



Figure A-103 REPEAT statement

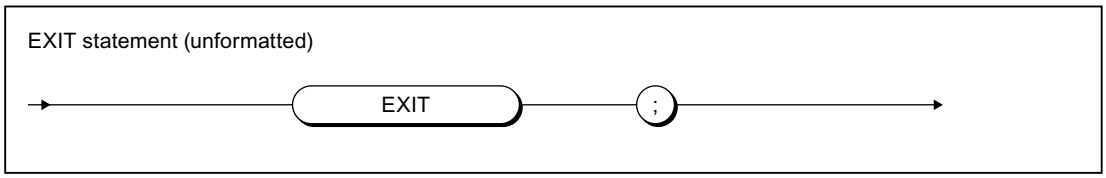



Figure A-104 EXIT statement

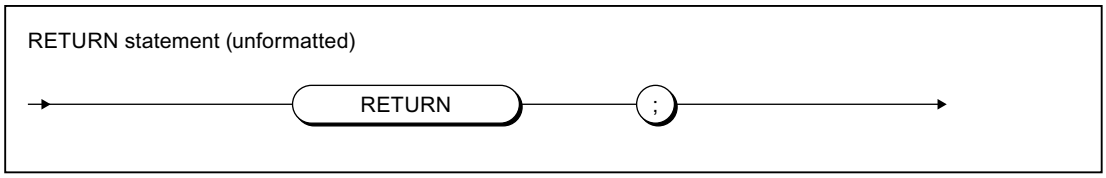



Figure A-105 RETURN statement

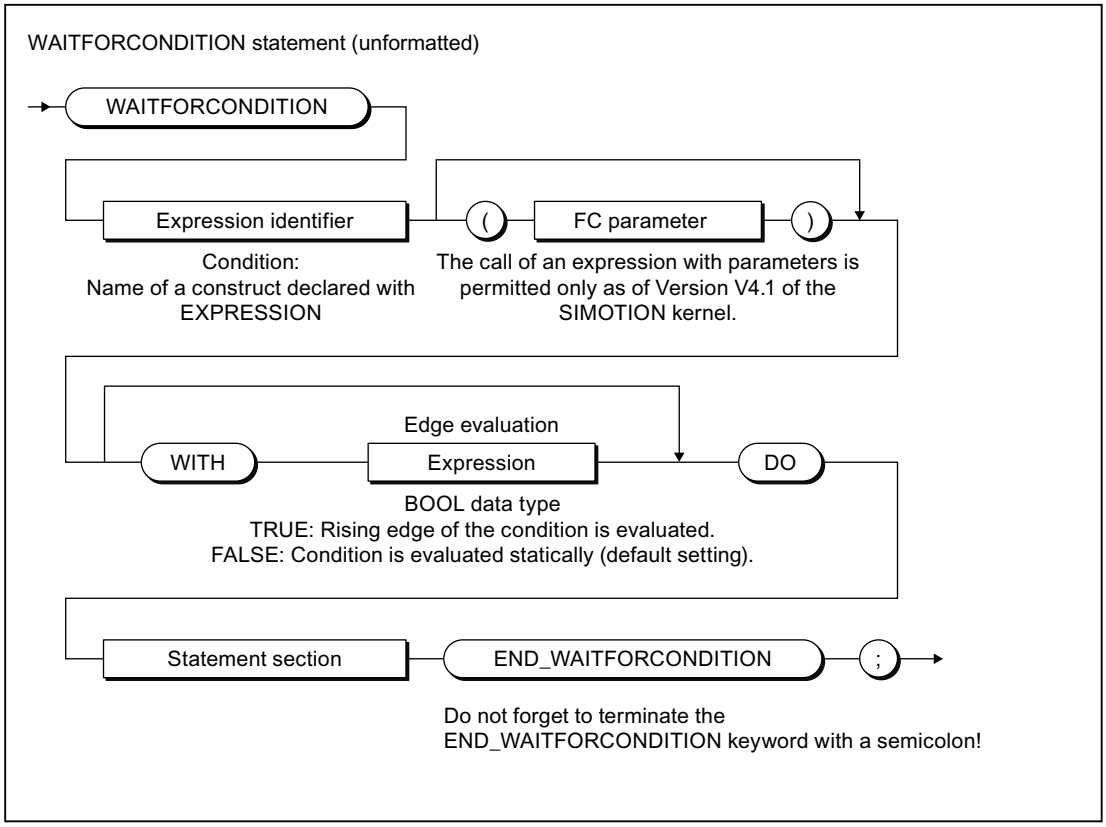



Figure A-106 WAITFORCONDITION statement

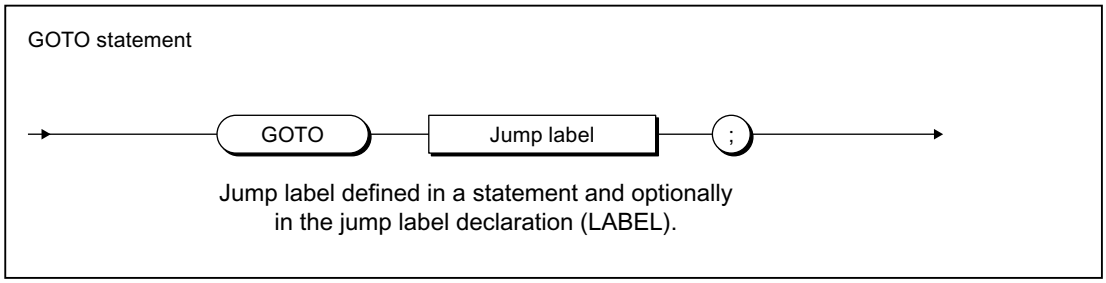



Figure A-107 GOTO statement

# 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 (1000 – 1100)

Table A- 9 File access errors (1000 – 1100)

| 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]]<br>Examples:<br>• -D myident // Definition of myident; this can be queried using #ifdef.<br>• -D myident= // myident is defined as empty character string<br>• -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 (2001, 2002)

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 (3002 – 3027)

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. |

| Error | Description |
|---|---|
| 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:<br>• UNIT myunit;<br>• UNIT myunit : dvtype;<br>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. |
| 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 data type declarations (4001 – 4051)

Table A- 12 Declaration errors in data 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. |
| 4020 | The specified identifier has already been used as a data type identifier. However, the definition differs from the current definition. This is not permitted.<br>Either choose a different identifier or adapt the type definitions. If this message appears on loading libraries or technology packages, you can use namespaces here too (e.g. USELIB mylib AS Namespace_1). |
| 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 variables declarations (5001 – 5509)

Table A- 13 Declaration errors, variables declarations (5001 – 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, $R$L. |
| 5200 | The data type definition contains a recursion, either directly or indirectly. This is not permitted. Do not use this data type at the position concerned. |
| 5201 | The function call creates a recursion, either directly or indirectly. This is not permitted. Do not call the function at the position concerned. |
| 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. |

| Error | Description |
|---|---|
| 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 the expression (6001 - 6201)

Table A- 14 Errors in the expression (6001 - 6201)

| Error | Description |
|---|---|
| 6001 | Syntax error: A statement terminated with a semicolon is expected,<br>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.<br>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 the 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:<br>• Variable cannot be read.<br>• 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. |

| Error | Description |
|---|---|
| 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. |
| 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. |

| Error | Description |
|---|---|
| 6036 | An operation is applied to a constant. The value of the constant is outside the definition range for the function. Examples are:<br>• Application of SQRT to a negative number.<br>• Use of logarithmic functions on a number <= 0.<br>• 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:<br>• The definition of the POU in the implementation section is missing. Only the prototype was specified in the interface section.<br>• 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.<br>• 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. |
| 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 the 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 the 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:<br>• f (x, y); or<br>• 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. |
| 6063 | 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. |
| 6071 | Access to configuration data is only possible for variables that have been specified completely. Therefore, array indices, which cannot be resolved until runtime, may not be used. |

| Error | Description |
|---|---|
| 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. |
| 6114 | A function block from a different source of the same library has been used. This is only possible within a library if the declaring source is compiled with the "Permit forward declarations" compiler setting activated. |
| 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. |
| 6160 | The specified array data type without a defined length is only permitted when VAR_IN_OUT parameters are declared in functions and function blocks. |
| 6200 | Only for "Permit language extensions" compiler option (-C lang_ext):<br>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):<br>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 the expression (7000 - 7014)

Table A- 15 Syntax errors, errors in the expression (7000 - 7014)

| Error | Description |
|---|---|
| 7000 | A syntax error has occurred. Possible causes:<br>• Incorrectly ended control structures (e.g. END_IF missing)<br>• Statements not terminated with ;<br>• Missing brackets |
| 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. |

| Error | Description |
|---|---|
| 7004 | An initialization value is expected. The value must be a constant. Constants can be assigned as follows:<br>• Directly per value<br>• Symbolically via a preceding constant declaration<br>• As an expression containing constants only |
| 7005 | If identical data types are to be initialized in different sources, this requires an identical initialization value too. Adapt the initialization values. |
| 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:<br>• Incorrectly terminated control structures (e.g. END_IF missing)<br>• Statements not terminated with ;<br>• Missing brackets |
| 7011 | A syntax error has occurred. Possible causes:<br>• Incorrectly terminated control structures (e.g. END_IF missing)<br>• Statements not terminated with ;<br>• Missing brackets |
| 7012 | A syntax error in the statement, that starts at the specified line, has occurred. Possible causes:<br>• Incorrectly terminated control structures (e.g. END_IF missing)<br>• Statements not terminated with ;<br>• Missing brackets |
| 7013 | A syntax error has occurred. An illegal construct is being used. |
| 7014 | A syntax error has occurred. Possible causes:<br>• Incorrectly terminated control structures (e.g. END_IF missing)<br>• Statements not terminated with ;<br>• Missing brackets |

## A.2.8 Error when linking a source file (8001, 8100)

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 a technology package (10000 - 10101)

Table A- 17 Errors while loading the interface of another UNIT or a technology package (10000 - 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:<br>• Up to Version V4.0 of the SIMOTION Kernel: 8 characters.<br>• 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.<br>• Remove a unit, library or technology package from the import list or<br>• 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".<br>If a package is involved, a newer version should be installed.<br>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. |
| 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. |

| Error | Description |
|---|---|
| 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. |
| 10024 | The specified technology package does not contain any components which can be used in the programming environment. Therefore, it cannot be used in the USEPACKAGE statement either. |
| 10025 | The specified identifier is not an identifier for a valid or an installed technology package. Therefore, it cannot be used in the USEPACKAGE statement either. |
| 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:<br>• The library has not yet been compiled.<br>• The library has not been compiled for all device types specified for the library container (e.g. in project-wide compilation).<br>• An error occurred in the last compilation.<br>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. |
| 10038 | A DCC library can only be used in the DCC. It is not permissible to integrate such a library into different programming languages. |
| 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 (15001 - 15700)

Table A- 18 Implementation restrictions (15001 - 15700)

| 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, it is 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 not permitted. 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. |
| 15154 | It is not permitted to apply a line comment and to use multi-line comments in the definition section. |

| Error | Description |
|---|---|
| 15200 | The specification of a bit offset for a bitstring variable requires the "Permit language extensions" compiler option (-C lang_ext). |
| 15700 | The specified construct is not supported by the version of SIMOTION SCOUT into which conversion is to be performed. |

## A.2.11 Warnings (16001 - 16700)

You can control the output of warnings and information:

- In the global compiler settings (Page 56)
- In the local compiler settings (Page 59)
- In an ST source file by specifying the following attribute (Page 294) within a pragma (Page 289): { _U7_PoeBld_CompilerOption := warning:*n*:on } or { _U7_PoeBld_CompilerOption := warning:*n*:off }, where *n* is the warning class or the number for the warning or information.

You can also redefine individual warnings and information as errors:

- In an ST source file by specifying the following attribute (Page 294) within a pragma (Page 289): { _U7_PoeBld_CompilerOption := warning:*n*:err }, where *n* is the number for the warning or information.

Table A- 19 Warnings (16001 - 16700)

| Error | Description |
|---|---|
| 16001 | (Warning class: 0)<br>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)<br>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)<br>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)<br>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)<br>During type conversion, the dependency of the variable value can cause the sign to change. |
| 16006 | (Warning class: 2)<br>The specified value will be rounded to the next displayable value due to insufficient display width. |
| 16007 | (Warning class: 2)<br>A loss of accuracy occurred during type conversion. Not all decimal places are considered. |

| Error | Description |
|---|---|
| 16008 | (Warning class: 2)<br>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)<br>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)<br>Specified program not exported to unit; therefore unable to use it in configuration of the execution level. |
| 16011 | (Warning class: 0)<br>The source file does not contain any exported global variables. No data are loaded to the target system. |
| 16012 | (Warning class: 0)<br>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)<br>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)<br>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. |
| 16015 | (Warning class: 2)<br>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)<br>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:<br>• Specify the data type of the used constants explicitly with <type>#<value>.<br>• Use an explicit data type conversion. |
| 16017 | (Warning class: 2)<br>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.<br>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)<br>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. |

| Error | Description |
|---|---|
| 16020 | (Warning class: 1)<br>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)<br>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)<br>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)<br>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)<br>The declaration hides the access to the technology object on the device. You can access this TO with _to.<name>. |
| 16025 | (Warning class: 1)<br>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)<br>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. |
| 16027 | (Warning class: 1)<br>The specified identifier, which is reserved for access to I/O qualities, is already assigned on the device. This means that I/O qualities cannot be accessed. |
| 16030 | (Warning class: 1)<br>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)<br>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. |
| 16103 | (Warning class: 3)<br>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. |
| 16113: | (Warning class: 3)<br>The pragma contains a syntax error in an attribute declaration. The valid syntax is as follows: <attribute identifier> := <attribute value>; |
| 16150 | (Warning class: 7)<br>A new definition has been made for the specified identifier. Consequently, the previous definition is invalid.<br>This warning enables the work of the preprocessor to be tracked. |
| 16151 | (Warning class: 7)<br>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.<br>This warning enables the work of the preprocessor to be tracked. |

| Error | Description |
|---|---|
| 16152 | (Warning class: 7)<br>The specified definition is not considered during code generation. This may be caused by the preprocessor being deactivated for the compiled source. |
| 16153 | (Warning class: 7)<br>The preprocessor is not active in the current source, even though preprocessor statements are used. Activate the preprocessor or remove the statements. |
| 16154 | (Warning class: -)<br>The preprocessor cannot be used to control the contents of USEPACKAGE, USELIB, and USES statements. It is no longer possible to automatically determine the dependencies between sources for the "Save project and compile changes" and "Save project and recompile all" functions. |
| 16170 | (Warning class: -)<br>The definitions from sources imported using USES are not considered during code generation. |
| 16171 | (Warning class: -)<br>The definition from the specified source imported using USES could not be loaded. Compile the specified source file beforehand. |
| 16200 | (Warning class: 4)<br>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)<br>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:<br>1. It can be used on a device with a version of the SIMOTION kernel as of V4.1.<br>2. The exponent is an integer.<br>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)<br>The condition of an IF statement, WHILE statement or REPEAT statement is a constant expression. |
| 16230 | (Warning class: 4)<br>The expression with the specified values does not cause any change to the result; optimized code will be created. |
| 16240 | (Warning class: 4)<br>The expression with the specified values exceeds the definition range of the operation. The result may be incorrect. |
| 16300 | (Warning class: 5)<br>The auxiliary value has a data type that cannot be converted to the data type configured for the message. |
| 16301 | (Warning class: 5)<br>The specified auxiliary value is not evaluated during output of the message. |
| 16302 | (Warning class: 5)<br>The data type of the auxiliary value cannot be determined from the message configuration. The specified data type is used. |
| 16303 | (Warning class: 5)<br>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. |

| Error | Description |
|---|---|
| 16304 | (Warning class: 5)<br>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:<br>• Specify the data type of the used constants explicitly with <type>#<value> or<br>• Use an explicit data type conversion. |
| 16400 | (Warning class: 6)<br>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)<br>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)<br>A variable that has neither been assigned nor read in the code has been declared. |
| 16450 | (Warning class: –)<br>A global variable has been created in the retentive memory range. This declaration is not permissible at the specified position. |
| 16451 | (Warning class: –)<br>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: –)<br>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: -)<br>The specified construct does not conform to the language standard; however, for compatibility reasons, it is supported for old platforms. Convert the usage to the specified alternative. |
| 16600 | (Warning class: 6)<br>The specified variable is not contained in the initialization list. The default initialization value is used. |
| 16601 | (Warning class: 6)<br>The specified variable is not contained in the initialization list. The default initialization value is used. |
| 16602 | (Warning class: 6)<br>The specified variable is not contained in the initialization list. The default initialization value is used. |
| 16603 | (Warning class: 6)<br>The specified function block does not contain any instance data and, as a result, has a size of 0 bytes. When transferred as a non-dimensional array for referencing purposes, the size of the element is defined as 1 element. |
| 16700 | (Warning class: 3)<br>The SIMOTION device can also be processed with previous versions of the SIMOTION SCOUT. The specified construct is not supported by all the earlier versions of the compiler. |

## A.2.12 Information (32010 - 32653)

You control this output of information together with the warnings (Page 419):

Table A- 20 Information (32010 - 32653)

| Error | Description |
|---|---|
| 32010 | (Warning class: 6)<br>The specified jump label identifier has been declared but not used. |
| 32020 | (Warning class: –)<br>The specified variable was declared globally in this source file or in another source file with the indicated data type.<br>This information helps when searching for the cause of compilation errors. It is issued together with error messages. |
| 32021 | (Warning class: –)<br>The specified variable was declared on the device as an I/O variable, a global device variable, or a system variable.<br>This information helps when searching for the cause of compilation errors. It is issued together with error messages. |
| 32022 | (Warning class: –)<br>The specified variable was declared in the project as a global identifier.<br>This information helps when searching for the cause of compilation errors. It is issued together with error messages. |
| 32023 | (Warning class: –)<br>Until now, no valid declaration has been found for the specified identifier.<br>This information is issued together with error messages. |
| 32024 | (Warning class: 0)<br>The specified variable has been declared as a global identifier in the current unit or in an importing unit.<br>This information helps when searching for the cause of compilation errors. It is issued together with error messages. |
| 32030 | (Warning class: 0)<br>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:<br>x : ARRAY [0 to 1] OF INT := [1, 2]; |
| 32050 | (Warning class: 0)<br>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)<br>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)<br>The specified identifier will be replaced thereafter by the output text.<br>This information enables the work of the preprocessor to be tracked. |

| Error | Description |
|---|---|
| 32651 | (Warning class: 7)<br>The definition of the specified identifier has been deleted with #undef.<br>This information enables the work of the preprocessor to be tracked. |
| 32652 | (Warning class: 7)<br>The identifier will be used with the specified replacement text in the source file. Compilation takes place with the replacement text.<br>This information enables the work of the preprocessor to be tracked. |
| 32653 | (Warning class: 7)<br>The specified identifier will be replaced thereafter by the output text. This information appears if additional replacements are loaded with a USES statement.<br>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.

```
//============================================================================
//(organization)
//(division / place)
//(c)Copyright 2009  All Rights Reserved
//----------------------------------------------------------------------------
//  project name: (name)
//  file name: (name as soon as saved)
//  library: (that the source is dedicated to)
//  system: (target system)
//  version: (SIMOTION / SCOUT version)
//  application: (relation to project/ product/ usage)
//  restrictions:
//  requirements: (hardware, technological package, memory needed, etc.)
//  search items: (with the purpose of browser usage)
//  functionality: (that is implemented)
//----------------------------------------------------------------------------
//  change log table:
//  version    date      expert in charge     changes applied
//
//============================================================================

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
// exported content imported and made usable in "sttemp_1_de".
// 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.
```

## A.3.2 Type definition in the interface

```
// ******************************************************
// * Type definition in the INTERFACE                   *
// ******************************************************

VAR_GLOBAL CONSTANT
       PI             : REAL := 3.1415;
       ARRAY_MAX1     : INT  := 4;
       ARRAY_MAX2     : INT  := 4;
       COLLECTION_MAX : INT  := 6;
       GLOBARRAY_MAX  : INT  := 12;
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
    ai16Dim1 : ARRAY [0..ARRAY_MAX1-1] OF INT;
    // Definition of a one-dimensional array with four array elements from
    // type INT under the name "ai16Dim1". With "ai16Dim1" as the data type
    // in all source file segments, one-dimensional arrays can now
    // be declared by type INT.
```

```
    aaDim2 : ARRAY [0..ARRAY_MAX2-1] OF ai16Dim1;
    // 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 "aaDim2"

    eTrafficLight : (RED, YELLOW, GREEN);
    // Definition of enumerator "eTrafficLight" as a
    // user-defined variable type. Variables of this type can
    // only accept the values "RED", "YELLOW" and "GREEN".

    sCollection : STRUCT
        toAxisX          : posaxis;
        aInStructDim1    : ai16Dim1;
        eTrafficInStruct : eTrafficLight;
        i16Counter       : INT;
        b16Status        : 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 "eTrafficLight") into
    // one structure. In addition, types
    // of technology objects can also be used.
    // In the example, the structure contains an element of
    type // a position axis (posaxis).
    // In the definition, make certain to sort the variables
    // by size in increasing sequence
    // (ARRAY, STRUCT, LREAL, DWORD, INT, BOOL ...)

    aCollection : ARRAY [0..5] OF sCollection;
    // Nesting is also possible. The type "aCollection"
    // contains a field comprising six elements of type "sCollection"

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.

    gaMyArray : ARRAY [0..GLOBARRAY_MAX-1] 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
    // "gaMyArray" three times in succession.
    // The field elements are thus assigned as follows:
    // 4, 4, 18, 18, 4, 4, 18, 18, 4, 4, 18, 18.

    gaMy2dim : aaDim2;
    // Example of a declaration of a two-dimensional array

    gaMy1dim : ai16Dim1;
    // Example of a declaration of a one-dimensional array with
    // use of a type declaration.

    gsMyStruct : sCollection;
    // Variable of the type or with the structure of
    // user_struct.

    gaMyArrayOfStruct : aCollection;
    // The variable generated here contains a field from
    // structural elements as declared in section TYPE/END_TYPE
    //.

    gtMyTime : TIME := T#0d_1h_5m_17s_4ms;
    // ...as elementary time types and derived data types.

    geMyTraffic : eTrafficLight := RED;
    // An enumerator of type "eTrafficLight" is created here and
    // assigned the value "RED".

    gi16MyInt : 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 "sttemp_l_de" into another source file
(UNIT)
    .

    FUNCTION FCmyFirst;
    FUNCTION_BLOCK FBmyFirst;
    PROGRAM myPRG;
    // The function blocks (FBs),
    // functions (FCs) and programs defined in the interface part are exported
    // 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
    //.

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 "sttemp_l_de". The definition of types must
    // be performed before the declaration of a variable.
```

```
VAR_GLOBAL  // In the user memory of the UNIT
    gboDigInput1 : 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 := gboDigInput1;
END_EXPRESSION
// Definition of an EXPRESSION.
// An EXPRESSION is a special function case, which recognizes only the
// return value 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 "gboDigInput1" (usual in a digital input or a
// condition in the program) takes on the value 1, the return value of the
// EXPRESSIONTRUE.
```

## 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 FCmyFirst : 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.

    geMyTraffic := YELLOW; // e.g. change the traffic light.

    FCmyFirst := 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 FBmyFirst
    // The statement section of the POU FUNCTION_BLOCK 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 *
    // ******************************************

    geMyTraffic := 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 : FBmyFirst;
        // 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.
        //
        // In 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 := FCmyFirst();
        // FC call and assignment of return value.

        WAITFORCONDITION xCond WITH TRUE DO
            // The statements programmed here come immediately for
            // execution if the condition EXPRESSION defined in the associated
            // "xcond" 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
```

# Index


## k

–, 145

## #

#define, 292
#else, 292
#endif, 292
#ifdef, 292
#ifndef, 292
#undef, 292

## *

*, 145
**, 145

## /

/, 145

## :

:, 111, 124
:=, 134, 177, 178

## _

_AdditionObjectType, 121
_alarm, 280
_CamTrackType, 120
_ControllerObjectType, 121
_device, 269, 280
_direct, 241, 246, 269, 280
_FixedGearType, 121
_FormulaObjectType, 121
_getSafeValue
  Application, 269
_PathAxis, 121
_PathObjectType, 121
_project, 280
_quality, 254, 280
_SensorType, 121
_setSafeValue
  Application, 269
_task, 280
_to, 280
_U7_PoeBld_CompilerOption, 294

## +

+, 145

## <

<, 147
<=, 147
<>, 147

## =

=, 147
=>, 179

## >

>, 147
>=, 147

## 1

-1.#IND, 315, 317
1.#INF, 315, 317
-1.#INF, 315, 317
-1.#INF, 315, 317
1.#QNAN, 315, 317
-1.#QNAN, 315, 317
-1.#QNAN, 315, 317

## A

Absolute identifier
  Overview, 354
Access times
  Parameter, 180
ANY, 109
ANY_BIT, 109

ANY_DATE, 109
ANY_ELEMENTARY, 109
ANY_INT, 109
ANY_NUM, 109
ANY_REAL, 109
ANYOBJECT, 121
Arithmetic operators, 144
ARRAY
  Data type, 114
  With a defined length, 114
  With a dynamic length, 176
Arrays
  Data type, 114
  Value assignments, 139, 140
  With a defined length, 114
  With a dynamic length, 176
Attribute
  Compiler option, 294

## B

Basic elements
  Of ST, 89
Basic functions, 144
Bit constants, 100
Bit data types, 106
BlockInit_OnChange, 295
BlockInit_OnDeviceRun, 295
Blocks, 88
Bookmarks, 43
BOOL, 106
Boolean data, 100
Branches
  Syntax, 402
Breakpoint, 327
  Activating, 341
  Call path, 335, 338
  Call stack, 344
  Deactivating, 343
  remove, 332
  Setting, 332
  Toolbar, 334
BYTE, 106

## C

Call path
  Breakpoint, 335, 338
  Call stack, 344
  Program run, 319
  Program status, 325
CamType, 121
CASE statement
  Description, 154
Character set, 89, 351
Code attributes, 288
Commands
  Overview of the basic system, 357
  ST programming language overview, 96
Comments, 104
  Source file section, 104
  Syntax, 374
Compiler, 81
  Attribute, 294
  Correcting errors, 55, 82
  Declaration errors, 407
  Declaration errors in type declarations, 409
  Declaration errors in variable declarations, 410
  Error when linking a source file, 415
  Errors while loading the interface of another UNIT or technology package, 416
  File access errors, 407
  Implementation restrictions, 418
  Information, 424
  Scanner errors, 407
  Setting, 56
  Start, 81
  starting, 55
  Syntax errors, errors in expression, 414
  Warnings, 419
Compiler option, 56, 66
Compiling
  Library, 271
Compound data types, 114, 118
CONSTANT, 127, 131
Constant block
  Syntax, 381
Constants
  Bit, 100
  Data types for constants, 105
  Date and time, syntax, 371
  Digit strings, syntax, 370
  Floating-point number, 99
  Formatting characters and separators, 353
  Globally valid, 215
  Integer, 98
  Literals, syntax, 366
  Symbolic names, 131
  Time specifications, 107
  Unit constants, 215
Control statements, 152
CPU memory access
  Identifiers for process image access, 354

Variable model, 212
Cross-reference list, 282
Displayed data, 283
Filtering, 286
Generating, 282
Single-step monitoring (MCC), 283
Sorting, 285
Trace (MCC), 283
TSI#currentTaskId, 283
TSI#dwuser_1, 283
TSI#dwuser_2, 283
Cyclic program execution
Effect on I/O access, 241, 246, 256
Effect on variable initialization, 228

## D

Data model, 212
Data type specification
ARRAY, 114
elementary, 113
Enumeration, 116
STRUCT, 118
Data types
ARRAY, 114
Bit data type, 106
Conversions, 164
Derivation of simple types, 113
elementary, 106
Elements, syntax, 388
Enumeration, 116
Enumerators, 116
Explicit conversions, 167
Implicit conversions, 165
Inheritance, 121
Initialization, 127
Numeric, 106
STRING, 107
STRUCT, 118
Structure, 118
Syntax, 388
Technology object, 120
Time, 107
TYPE, 111
User-defined, 111
User-defined, syntax, 391
DATE, 107
DATE_AND_TIME, 107
Debug mode, 307, 328
Declaration
Parameter, 126
Variables, 126
Declaration section
Syntax, 379
Declarations
Syntax, 384
Derivation of simple data types, 113
Derived data type
Array, 114
ARRAY, 114
Enumeration, 116
Enumerator, 116
STRUCT, 118
Structure, 118
Device
Rules for identifiers, 297
Settings, 299
DINT, 106
DINT#MAX, 108
DINT#MIN, 108
Direct access, 241, 246
Properties, 242
Variable model, 212
Download
Effect on variable initialization, 228
DriveAxis, 120
DT, 107
DWORD, 106

## E

Editor, 26
Example for program, 79
Operation, 80
Toolbar, 48
Elementary data types
Overview, 106
Enumeration
Defining, 116
Example, 117
Enumeration data types, 116
Enumerators, 116
Error
FB or FC call, 184
Error messages
Declaration errors, 407
Declaration errors in type declarations, 409
Declaration errors in variable declarations, 410
Error when linking a source file, 415
Errors while loading the interface of another UNIT or technology package, 416
File access errors, 407
Implementation restrictions, 418
Information, 424

Scanner errors, 407
Syntax errors, errors in expression, 414
Warnings, 419
Example, complete
FBs and FCs, 185
Rotate bit in output byte, 76
ST source file (template), 426
User-defined data types, 119
Using data types of TOs, 122
EXIT statement
Description, 160
Explicit data type conversions, 167
Exponent
Description, 99
Exponentiation, 145
Export
ST source file, 69
EXPRESSION
Description, 200
Syntax, 190
Logic expression; bit-serial expression
logic,
Expressions
Arithmetic, 144
Logic, 151
Relational expressions, 147, 151
Rules for formulation, 142, 151
ExternalEncoderType, 120


## F


FB, 169
FB/FC variables
Definition, 218
Variable model, 212
FC, 169
File
See Source file, 102
Floating-point number
Data types, 106
Description, 99
Notation, 99
FollowingAxis, 120
FollowingObjectType, 120
FOR statement
Description, 156
Formatting characters, 351
Forward declaration, 300
Function, 169
Call path, 325
Calling, 180
defining, 170
Error sources during a call, 184
Example, 185
Input parameters, 174
Local variables, 175
Source file section, 198
Structure, 170
Syntax, 170
Function block, 169
Call path, 325
Call, syntax, 182
Calling, 181
defining, 171
Difference to the FC, 185
Error sources during a call, 184
Example, 185
In/out parameter, 174
Input parameters, 174
Instances, 181
Local variables, 175
Names, 181
Output parameters, 175
Source file section, 199
Structure, 171
Syntax, 171


## G


Global device user variables
Defining, 222
Variable model, 212
GOTO statement, 303


## H


Hardware
Setting up, 78
Hiding validity ranges, 275
HMI_Export, 294


## I


I/O variable
create, 250, 268
Creating, 250, 268
Direct access, 241, 246
Process image, 241, 246
Process image of the BackgroundTask, 258
Status, 254
Variable model, 212
Identifier
Predefined, 354

Reserved for ST, 96, 357
Rules for formulating, 89
Rules for SIMOTION devices, 297
Syntax, 89
Identifiers
Syntax, 365
IF statement
Description, 152
Implementation
Source file section, 197
Implicit data type conversions, 165
Import
ST source file, 70
In/out assignment
Syntax, 179
In/out parameter
Function block, 174
Transfer, 178
Inheritance
During import/export, 210
For technology objects, 121
Initialization
Data types, 127
Syntax, 386
Time of the variable initialization, 228
Variables, 127
Input assignment
Syntax, 177
Input parameters
Access in the function block, 183
Function, 174
Function block, 174
Transfer, 177
Instance declaration of FB
Syntax, 181
INT, 106
INT#MAX, 108
INT#MIN, 108
Integer
Data types, 106
Description, 98
Notation, 98
Integer number
See Integer, 98
Interface
Source file section, 194

## J

Jump labels
Syntax, 384

## K

Keyboard shortcuts, 52
Know-how protection
Libraries, 273
Source files, 67

## L

LABEL declaration, 303
Language description
Resources, 87, 349, 351
Library, 270
Compiling, 271
Using, 274
Line numbering, 47
Local data stack, 223, 228
Local variables
Variable model, 212
Lower case, 47
LREAL, 106

## M

MeasuringInputType, 120
Memory requirement, 223, 228
MOD, 145
Monitoring variables
Variable status, 317
Multi-element variables, 139, 140

## N

Names, 89
Namespace
Predefined, 280
User-defined, 278
New
I/O variable, 250, 268
Number systems
Notation, 99
Numbers
Data types for numbers, 105
Description, 98
Notation, 98
Numeric data types, 106

## O

Opening
ST source file, 24

Operands
Syntax, 395
Operating mode
Debug mode, 307, 328
Process mode, 306
Test mode, 306, 322
Operators, 356
Priority, 151
Relational operators, 147
Syntax, 398
Output parameters
Access in the function block, 183
Function block, 175
Transfer, 179
OutputCamType, 120

## P

Parameter
Access times, 180
Block (syntax), 172
Declaration, 171
Declaration, general, 126
Function and function block, 171
Transfer (in/out parameter), 178
Transfer (input parameter), 177
Transfer (output parameter), 179
Transfer (principle), 177
Parameter blocks
Syntax, 383
PosAxis, 120
Pragma
Attribute, 294
Preprocessor statement, 291
Preprocessor
Activating, 57, 60
Controlling, 290
Preprocessor statement, 291
Using, 57, 60
Warning class, 65
Preprocessor statement
Example, 293
Printing
ST source file, 71
Process image
Cyclic tasks, 246
principle and use, 241, 256
Properties, 242
Symbolic access, 266
Process image of the BackgroundTask, 241
Process image of the cyclic tasks, 241
Process mode, 306
Program
Assigning tasks, 83
Call path, 325
Compiling, 81
Connecting to target system, 84
Creating (example), 79
Download, 85
Executing, 83, 86
Locating errors, 305
Source file section, 200
starting, 83, 86
Status (test tool), 321
Testing, 305
Program organization units
Source file section, 198
Syntax, 377
Program run, 319
Toolbar, 320
Program section
See Source file section, 193
Program structure, 286
Program structuring, 152
program variables
Definition, 218
In the data model, 217
Variable model, 212
Programming environment, 17
Project
Opening, 77
Project comparison
Overview, 347
Prototype
Program organization unit, 300
Prototypes, 206

## R

REAL, 106
Real number
See Floating-point number, 99
Reference, 120
Reference data, 282
References, 5
Relational expressions, 147
REPEAT statement
Description, 159
Repetition statements and jump statements
Syntax, 403
Reserved identifiers, 91, 357
RETAIN, 127, 217
Retentive variables
Definition, 217

Variable model, 212
RETURN statement
Description, 161
Rules
Formatted, 349, 365
Semantics, 88
Unformatted, 350, 365
RUN
Effect on variable initialization, 228

## S

SCOUT Workbench > See Workbench, 17
sections
Syntax, 375
Separators, 351
Sequential program execution
Effect on I/O access, 241, 246
Effect on variable initialization, 228
Setting
Compiler, 56
Shortcut
Script editor, 52
ST editor, 52
Shortcuts, 52
SIMOTION device
Rules for identifiers, 297
Settings, 299
Simple data types
Derivation, 113
Single-element variables, 135
SINT, 106
SINT#MAX, 108
SINT#MIN, 108
Source file
Structure, 102
Source file section, 193
Data type declaration, 203
Declaration section, 201
Function, 198
Function block, 199
Implementation, 197
Interface, 194
Program, 200
Program organization unit, 198
Statement, 103
Statement section, 202
Unit statement, 206
Variable declaration, 203
ST compiler. See Compiler, 55
ST editor, 26
Pairs of brackets, 47
ST source file
Exporting, 69
Importing, 70
Opening, 24
Printing, 71
See Source file, 102
Template (example), 426
ST source file section
See Source file section, 193
Standard functions, 144
Statement
Source file section, 103, 202
Statement section
Syntax, 392
Status
I/O variable, 254
Program (test tool), 321
STOP to RUN
Effect on variable initialization, 228
STRING, 107
assignment, 135
Edit, 136
Element, 136
Syntax diagram, 107
StructAlarmId, 109
STRUCTALARMID#NIL, 110
StructTaskId, 109
STRUCTTASKID#NIL, 110
Structure
Defining, 118
Example, 119
Structured variables, 139, 140
Structures
Syntax, 375
Symbol Browser, 312
Symbolic access to I/O address space
Process image, 266
Syntax diagram, 87
System functions
Inheritance, 121
System variables
Inheritance, 121
Variable model, 212

## T

T#MAX, 108
T#MIN, 108
Target variable, 133
Task
Assigning programs, 83
Effect on variable initialization, 228

Technology object
Data type, 120
Inheritance, 121
TemperatureControllerType, 121
Template
ST source file, 426
Terminals, 89
Test mode, 306, 322
Testing a program, 305
TIME, 107
Time types
Conversions, 164
Functions, 144
Overview, 107
TIME#MAX, 108
TIME#MIN, 108
TIME_OF_DAY, 107
TIME_OF_DAY#MAX, 108
TIME_OF_DAY#MIN, 108
TO#NIL, 121
TOD, 107
TOD#MAX, 108
TOD#MIN, 108
Tool tip, 47
Trace tool, 346
TSI#currentTaskId
Cross-reference list, 283
TSI#dwuser_1
Cross-reference list, 283
TSI#dwuser_2
Cross-reference list, 283
TYPE, 111
Type conversion functions, 164
Type declaration, 111

## U

UDINT, 106
UDINT#MAX, 108
UDINT#MIN, 108
UDT
See User-defined data type, 110
UINT, 106
UINT#MAX, 108
UINT#MIN, 108
Unit
Source file section, 206
Template (example), 426
UNIT, 206
Unit constants
Definition;, 215
Unit variables, 215
Definition, 215
Non-retentive, 215
Variable model, 212
Upper case, 47
USELIB, 195, 278
USEPACKAGE, 195, 278
User-defined data type
Syntax, 111
USES, 195, 197, 209
USINT, 106
USINT#MAX, 108
USINT#MIN, 108

## V

Value assignments
Description, 133
Syntax, 393
VAR, 126, 220, 221
VAR CONSTANT, 127, 131
VAR_GLOBAL, 126, 215
VAR_GLOBAL CONSTANT, 127, 132
VAR_GLOBAL RETAIN, 127, 217
VAR_IN_OUT, 126, 173
VAR_INPUT, 126, 173
VAR_OUTPUT, 126, 173
VAR_TEMP, 126, 221
Variable blocks
Syntax, 381
Variable status
Monitoring variables, 317
Variables, 124
ARRAY, 139, 140
Battery-backed, 217
Declaration, 126
Declaration (source file section), 203
elementary, 135
Enumerator data type, 139
Function block, 175
Functions, 175
Hiding validity ranges, 275
Identical names, 275
Initialization, 127
Instance declaration of FB, 181
Local, 218
Parameter declaration, 171
Process image, 241, 256
Retentive, 217
Static, 218
structured, 140
Temporary, 218
timing of initialization, 228

Unit variable, 215
Validity, 212
Watch tables, 315

## W

Warning class, 65, 290
Watch tables, 315
WHILE statement
Description, 158
WORD, 106
Workbench
Elements, 19
Programming environment, 17