SIEMENS

SIMOTION

SIMOTION SCOUT
SIMOTION ST Structured Text

Programming and Operating Manual

Preface

Fundamental safety instructions 1

Introduction 2

Getting Started with ST 3

ST Fundamentals 4

Functions, Function Blocks, and Programs 5

Object-oriented programming - OOP (as of kernel V4.5) 6

Integration of ST in SIMOTION 7

Error Sources and Program Debugging 8

Appendix A

Valid from version 5.7 or higher

03/2025
A5E33437165B

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**
indicates that minor personal injury can result if proper precautions are not taken.

NOTICE
indicates that property damage can result if proper precautions are not taken.

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 Aktiengesellschaft. 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 Aktiengesellschaft
Digital Industries
Postfach 48 48
90026 NÜRNBERG
GERMANY

A5E33437165B
Ⓟ 11/2024 Subject to change

Copyright © Siemens 2025.
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 product family, in product version V5.7 in conjunction with:

- A SIMOTION device with the following versions of a SIMOTION Kernel:
  - V5.7
  - V5.6
  - V5.5
  - V5.4
  - V5.3
  - V5.2
  - V5.1
  - V4.5
  - V4.4
  - V4.3
  - V4.2
  - V4.1 [1]
  - V4.0 [1]
  - V3.2 [1]

  [1] V4.5 is the last product version of SIMOTION SCOUT that will support these versions of the SIMOTION Kernel.
- The relevant version of the following SIMOTION Technology Packages, depending on the kernel:
  - Cam
  - Path (Kernel as of V4.1)
  - 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.

- **Grouped safety messages** (Chapter 1)
- **Introduction** (Chapter 2)
- **Getting Started with ST** (Chapter 3)
  Requirements for creating programs and a sample program
- **ST Basics** (Chapter 4)
  Elements of the ST programming language, variable and data type declarations, statements
- **Functions, Function Blocks and Programs** (Chapter 5)
  Programming and call of the program organization units (POU)
- **Object-Oriented Programming - OOP (as of Kernel V4.5)** (Chapter 6)
  Programming and calling classes and methods
- **Integration of ST into SIMOTION** (Chapter 7)
  Behavior of variables, access to inputs and outputs, libraries, preprocessor
- **Error Sources and Program Test** (Chapter 8)
  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 the SIMOTION Documentation Overview document.

This documentation is included as electronic documentation in the scope of delivery of SIMOTION SCOUT. It comprises ten documentation packages.

The following documentation packages are available for SIMOTION product version V5.7:

- SIMOTION Engineering System Handling
- 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

SIMOTION at a glance

We have compiled an overview page from our range of information about SIMOTION with the most important information on frequently asked topics - which can be opened with only one click.

Whether beginner or experienced SIMOTION user – the most important downloads, manuals, tutorials, FAQs, application examples, etc. can be found at

https://support.industry.siemens.com/cs/ww/en/view/109480700

Additional information

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

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

https://support.industry.siemens.com/cs/ww/en/view/109479653

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 it for the purpose of your own machine documentation:

https://support.industry.siemens.com/My/ww/en/documentation

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**:

https://support.industry.siemens.com/cs/de/en/ps/14505/faq

Technical support

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

https://support.industry.siemens.com/cs/ww/en/sc/2090

Table of contents

Preface ... **3**

1 Fundamental safety instructions ... **17**

1.1 General safety instructions ... 17
1.1.1 Malfunctions of the machine as a result of incorrect or changed parameter settings ... 17
1.2 Cybersecurity information ... 17
1.3 Note regarding the general data protection regulation ... 18
1.4 Danger to life due to software manipulation when using removable storage media ... 18

2 Introduction ... **19**

2.1 High-level programming language ... 19
2.2 Programming language with technology commands ... 19
2.3 Execution levels ... 19
2.4 ST editor with tools for writing and testing programs ... 20

3 Getting Started with ST ... **21**

3.1 Integration of ST in SCOUT ... 21
3.1.1 Getting to know the elements of the workbench ... 23
3.2 Requirements for program creation ... 24
3.3 Working with the ST editor and the compiler ... 25
3.3.1 Insert ST source file ... 25
3.3.2 Opening an existing ST source file ... 27
3.3.3 Changing the properties of an ST source file ... 27
3.3.4 Working with the ST editor ... 29
3.3.4.1 Syntax coloring ... 30
3.3.4.2 Drag&drop ... 30
3.3.4.3 Marking the variable occurrence ... 31
3.3.4.4 Selecting the same level of the control structure ... 31
3.3.4.5 Searching and replacing in Selection ... 31
3.3.4.6 Settings of the ST editor ... 32
3.3.4.7 Indentations and tabs ... 34
3.3.4.8 Folds (show and hide blocks) ... 36
3.3.4.9 Splitting the editor window ... 40
3.3.4.10 Displaying spaces, tabs and line ends ... 41
3.3.4.11 Changing the font size in the ST editor ... 42
3.3.4.12 Select text ... 43
3.3.4.13 Generating a simple series of numbers (generating a sequence) ... 45
3.3.4.14 Use bookmarks ... 47
3.3.4.15 Automatic completion ... 49
3.3.4.16 Opening a called block ... 50
3.3.4.17 Code snippets ... 51
3.3.4.18 Other help for the ST editor ... 53

3.3.4.19 Using the command library 54
3.3.4.20 ST editor toolbar 54
3.3.4.21 ST editor context menu 56
3.3.4.22 Shortcuts 59
3.3.5 Starting the compiler 62
3.3.5.1 Help for the error correction 63
3.3.6 Making settings for the compiler 63
3.3.6.1 Global compiler settings 63
3.3.6.2 Local compiler settings 67
3.3.6.3 Effectiveness of global or local compiler settings 71
3.3.6.4 Meanings of the warning classes 73
3.3.6.5 Display of the compiler options 73
3.3.7 Know-how protection for ST source files 75
3.3.8 Making preprocessor definitions 75
3.3.9 Exporting, importing and printing an ST source file 76
3.3.9.1 Exporting an ST source file as a text file (ASCII) 77
3.3.9.2 Exporting an ST source file in XML format 77
3.3.9.3 Importing a text file (ASCII) as an ST source file 78
3.3.9.4 Importing XML data into ST source files 78
3.3.9.5 Printing an ST source file 79
3.3.10 Using an external editor 79
3.3.11 ST source file menus 81
3.3.11.1 ST source file menu 81
3.3.11.2 ST source file context menu 82

3.4 Creating a sample program 83
3.4.1 Requirements 84
3.4.2 Opening or creating a project 84
3.4.3 Making the hardware known 85
3.4.4 Entering source text with the ST editor 86
3.4.4.1 Functions of the editor 87
3.4.4.2 Source text of the sample program 88
3.4.5 Compiling a sample program 88
3.4.5.1 Starting the compiler 88
3.4.5.2 Correcting errors 89
3.4.5.3 Example of error messages 89
3.4.6 Running the sample program 90
3.4.6.1 Assigning a sample program to an execution level 90
3.4.6.2 Establishing a connection to the target system 91
3.4.6.3 Downloading the sample program to the target system 92
3.4.6.4 Starting and testing the sample program 93

4 ST Fundamentals 95

4.1 Language description resources 95
4.1.1 Syntax diagram 95
4.1.2 Blocks in syntax diagrams 96
4.1.3 Meaning of the rules (semantics) 96

4.2 Basic elements of the language 96
4.2.1 ST character set 97
4.2.2 Identifiers in ST 97
4.2.2.1 Rules for identifiers 97
4.2.2.2 Examples of identifiers 98

4.2.3 Reserved identifiers ... 99
4.2.3.1 Protected identifiers in the ST programming language ... 99
4.2.3.2 Reserved identifiers in the ST programming language ... 105
4.2.3.3 Additional reserved identifiers ... 106
4.2.4 Numbers and Boolean values ... 107
4.2.4.1 Integers ... 107
4.2.4.2 Floating-point numbers ... 108
4.2.4.3 Exponents ... 108
4.2.4.4 Boolean values ... 109
4.2.4.5 Data types of numbers ... 109
4.2.5 Character strings ... 109

4.3 Structure of an ST source file ... 110
4.3.1 Statements ... 112
4.3.2 Comments ... 112

4.4 Data types ... 113
4.4.1 Elementary data types ... 114
4.4.1.1 Value range limits of elementary data types ... 116
4.4.1.2 General data types ... 116
4.4.1.3 Elementary system data types ... 117
4.4.2 User-defined data types ... 117
4.4.2.1 Syntax of user-defined data types (type declaration) ... 118
4.4.2.2 Derivation of elementary or derived data types ... 119
4.4.2.3 Derived data type ARRAY ... 120
4.4.2.4 Derived data type - Enumerator ... 123
4.4.2.5 Derived data type STRUCT (structure) ... 124
4.4.3 Technology object data types ... 132
4.4.3.1 Description of the technology object data types ... 132
4.4.3.2 Inheritance of the properties for axes ... 134
4.4.3.3 Examples of the use of technology object data types ... 134
4.4.4 System data types ... 135

4.5 Variable declaration ... 136
4.5.1 Syntax of variable declaration ... 136
4.5.2 Overview of all variable declarations ... 137
4.5.3 Initialization of variables or data types ... 139
4.5.4 Constants ... 143

4.6 Value assignments and expressions ... 145
4.6.1 Value assignments ... 145
4.6.1.1 Syntax of the value assignment ... 145
4.6.1.2 Value assignments with variables of an elementary data type ... 147
4.6.1.3 Value assignments with variables of the STRING elementary data type ... 147
4.6.1.4 Value assignments with variables of a bit data type ... 148
4.6.1.5 Value assignments with variables of the derived enumerator data type ... 150
4.6.1.6 Value assignments with variables of the derived ARRAY data type ... 150
4.6.1.7 Value assignments with variables of the derived STRUCT data type ... 151
4.6.2 Expressions ... 152
4.6.2.1 Result of an expression ... 152
4.6.2.2 Interpretation order of an expression ... 153
4.6.3 Operands ... 154
4.6.4 Arithmetic expressions ... 155
4.6.4.1 Examples of arithmetic expressions ... 157

4.6.5 Relational expressions ... 158
4.6.6 Logic expressions and bit-serial expressions ... 160
4.6.7 Priority of operators ... 162

4.7 Control statements ... 162
4.7.1 IF statement ... 162
4.7.2 CASE statement ... 164
4.7.3 FOR statement ... 167
4.7.4 WHILE statement ... 169
4.7.5 REPEAT statement ... 170
4.7.6 EXIT statement ... 171
4.7.7 CONTINUE statement ... 171
4.7.8 RETURN statement ... 172
4.7.9 WAITFORCONDITION statement ... 173
4.7.10 GOTO statement ... 175

4.8 Data type conversions ... 176
4.8.1 Elementary data type conversion ... 176
4.8.1.1 Implicit data type conversions ... 176
4.8.1.2 Explicit data type conversions ... 178
4.8.2 Supplementary conversions ... 179

5 Functions, Function Blocks, and Programs ... 181

5.1 Creating and calling functions and function blocks ... 181
5.1.1 Defining functions ... 181
5.1.2 Defining function blocks ... 182
5.1.2.1 Defining classic function blocks ... 182
5.1.2.2 Defining object-oriented function blocks with methods ... 183
5.1.2.3 Defining methods in object-oriented FBs ... 184
5.1.2.4 Access identifiers within object-oriented FBs ... 185
5.1.3 Declaration section of FB and FC ... 186
5.1.4 Statement section of FB and FC ... 191
5.1.5 ARRAY with a dynamic length (as of Kernel V4.2) ... 192
5.1.6 Call of functions and function block calls ... 194
5.1.6.1 Principle of parameter transfer ... 195
5.1.6.2 Parameter transfer to input parameters ... 195
5.1.6.3 Parameter transfer to in/out parameters ... 196
5.1.6.4 Parameter transfer to output parameters ... 197
5.1.6.5 Parameter access times ... 197
5.1.6.6 Calling a function ... 198
5.1.6.7 Calling function blocks (declaring and calling instances) ... 199
5.1.6.8 Accessing the FB's output parameter outside the FB ... 201
5.1.6.9 Accessing the FB's input parameter outside the FB ... 201
5.1.6.10 Access an FB’s public variables outside of the FB ... 202
5.1.6.11 Call methods within and outside of a FB ... 202
5.1.6.12 Error sources in FB calls ... 203

5.2 Comparison of functions and function blocks ... 203
5.2.1 Description of example ... 203
5.2.2 Source file with comments ... 205

5.3 Programs ... 207
5.3.1 Assignment of a program in the execution system ... 207
5.3.2 Calling a program in the program ("program in program") ... 207

5.4 Expressions ... 208

6 Object-oriented programming - OOP (as of kernel V4.5) ... 213

6.1 Important note on object-oriented programming ... 213

6.2 Classes and methods ... 214
6.2.1 Creating classes (basic classes) ... 214
6.2.1.1 Defining classes (basic classes) ... 214
6.2.1.2 Declaration subsection of a class ... 214
6.2.1.3 Access identifier within classes ... 215
6.2.2 Creating methods ... 217
6.2.2.1 Defining methods ... 217
6.2.2.2 Declaration subsection of methods ... 218
6.2.2.3 Statement section of methods ... 220
6.2.3 Calling methods within the class ... 221
6.2.3.1 Calling methods ... 221
6.2.3.2 Parameter transfer to input parameters ... 222
6.2.3.3 Parameter transfer to in/out parameters ... 223
6.2.3.4 Parameter transfer to output parameters ... 224
6.2.4 Example classes and methods ... 224
6.2.4.1 Example: Counter as a class ... 224
6.2.5 Creating and using instances of classes ... 226
6.2.5.1 Declaring an instance of a class ... 226
6.2.5.2 Calling public methods of a class outside of this class ... 227
6.2.5.3 Accessing the public variables of a class outside of the class ... 228
6.2.5.4 Example: Call of the counter method ... 228

6.3 Inheritance of classes and methods ... 229
6.3.1 Inheritance of classes ... 229
6.3.2 Overriding methods ... 230
6.3.3 Example: Counter in steps of 5 through inheritance ... 231
6.3.4 Example: Call of the methods for both counters ... 232
6.3.5 Initialization of derived classes and their instances ... 235

6.4 Abstract classes ... 237

6.5 Object-oriented interface ... 238
6.5.1 Defining an object-oriented interface ... 238
6.5.2 Implementation of object-oriented interfaces in classes ... 240
6.5.3 Variables of object-oriented interfaces ... 242
6.5.4 Example of interface variables ... 245

6.6 Comparison between abstract class and object-oriented interface ... 248

6.7 General references ... 249
6.7.1 Defining general references ... 249
6.7.2 Forming general references ... 250
6.7.3 Operations with general references ... 251

6.8 I/O references (as of kernel V5.1) ... 253
6.8.1 Forming I/O references ... 253
6.8.2 Linking I/O references to I/O variables ... 254

7 Integration of ST in SIMOTION ... 257

7.1 Source file sections ... 257

7.1.1 Use of the source file sections ... 257
7.1.1.1 Interface section ... 258
7.1.1.2 Implementation section ... 260
7.1.1.3 Program organization units (POUs) ... 262
7.1.1.4 Functions (FCs) ... 263
7.1.1.5 Function blocks (FBs) ... 264
7.1.1.6 Programs ... 266
7.1.1.7 Expressions ... 267
7.1.1.8 Classes (as of Kernel V4.5) ... 268
7.1.1.9 Methods ... 270
7.1.1.10 Object-oriented interface (as of Kernel V4.5) ... 271
7.1.1.11 Declaration section ... 272
7.1.1.12 Statement section ... 273
7.1.1.13 Data type definition ... 274
7.1.1.14 Variable declaration ... 275
7.1.2 Declaring ST source files public and using them ... 278
7.1.2.1 Unit identifier ... 278
7.1.2.2 Interface section of a unit with Declare Public function ... 279
7.1.2.3 Example of a unit with Declare Public function ... 280
7.1.2.4 USES statement in a using unit ... 281
7.1.2.5 Example of a using unit ... 282

7.2 Variables in SIMOTION ... 282
7.2.1 Variable model ... 283
7.2.1.1 Unit variables ... 285
7.2.1.2 Non-retentive unit variables ... 286
7.2.1.3 Retentive unit variables ... 287
7.2.1.4 Local variables (static and temporary variables) ... 288
7.2.1.5 Static variables ... 291
7.2.1.6 Temporary variables ... 292
7.2.1.7 Retentive local variables (as of kernel V4.5) ... 292
7.2.2 Use of global device variables ... 294
7.2.3 Memory ranges of the variable types ... 295
7.2.3.1 Example of memory areas ... 297
7.2.3.2 Memory requirement of the variables on the local data stack ... 299
7.2.4 Time of the variable initialization ... 300
7.2.4.1 Initialization of retentive global variables ... 300
7.2.4.2 Initialization of non-retentive global variables ... 302
7.2.4.3 Initialization of local variables ... 304
7.2.4.4 Initialization of static program variables ... 305
7.2.4.5 Initialization of retentive local variables of programs ... 306
7.2.4.6 Initialization of instances of function blocks (FBs) or classes ... 307
7.2.4.7 Initialization of retentive local variables of function blocks (FBs) and classes ... 308
7.2.4.8 Initialization of system variables of technology objects ... 308
7.2.4.9 Version ID of global variables and their initialization during download ... 309
7.2.5 Declaring variables in the Variable declaration dialog box ("on-the-fly" variable declaration) ... 311
7.2.6 Variables and HMI devices ... 315

7.3 Access to inputs and outputs (process image, I/O variables) ... 319
7.3.1 Overview of access to inputs and outputs ... 319
7.3.2 Important features of direct access and process image access ... 320
7.3.3 Direct access and process image of cyclic tasks ... 323

7.3.3.1 Address range of the SIMOTION devices 325
7.3.3.2 Rules for I/O addresses for direct access and the process image of the cyclical tasks 326
7.3.3.3 Creating I/O variables for direct access or process image of cyclic tasks 327
7.3.3.4 Syntax for entering I/O addresses 329
7.3.3.5 Possible data types of I/O variables 330
7.3.3.6 Detailed status of the I/O variables (as of Kernel V4.2) 330
7.3.4 Access to fixed process image of the BackgroundTask 332
7.3.4.1 Common process image (as of Kernel V4.2) 334
7.3.4.2 Separate process image (up to Kernel V4.1) 336
7.3.4.3 Absolute access to the fixed process image of the BackgroundTask (absolute PI access) 338
7.3.4.4 Syntax for the identifier for an absolute process image access 339
7.3.4.5 Symbolic access to the fixed process image of the BackgroundTask (symbolic PI access) 340
7.3.4.6 Possible data types for symbolic PI access 341
7.3.4.7 Example of symbolic PI access 342
7.3.4.8 Creating an I/O variable for access to the fixed process image of the BackgroundTask 342
7.3.5 Accessing I/O variables 343

7.4 Using libraries 343
7.4.1 Compiling a library 345
7.4.2 Know-how protection for libraries 346
7.4.3 Using data types, functions and function blocks from libraries 347

7.5 Use of the same identifiers and namespaces 348
7.5.1 Use of the same identifiers 348
7.5.2 Namespaces 351
7.5.2.1 Namespaces for libraries and technology packages 352
7.5.2.2 Predefined namespaces 353
7.5.2.3 Object-oriented namespace 355

7.6 Reference data 361
7.6.1 Cross-reference list 361
7.6.1.1 Generating and updating a cross-reference list 361
7.6.1.2 Content of the cross-reference list 362
7.6.1.3 Working with a cross-reference list 364
7.6.1.4 Filtering the cross-reference list 364
7.6.2 Program structure 365
7.6.2.1 Content of the program structure 365
7.6.3 Code attributes 366
7.6.3.1 Code attribute contents 367
7.6.4 Reference to variables 367

7.7 Controlling the preprocessor and compiler with pragmas 368
7.7.1 Controlling the preprocessor 369
7.7.1.1 Preprocessor statement 370
7.7.1.2 Example of preprocessor statements 373
7.7.2 Controlling compiler with attributes 374
7.7.3 Issue non-assigned compiler message 378

7.8 SIMOTION devices 379
7.8.1 Rules for identifiers of the SIMOTION devices 379
7.8.2 Making settings on the device (as of Kernel V4.2) 381

7.9 Forward declarations 381

7.10 Jump statement and label 385

8 Error Sources and Program Debugging 387

8.1 Notes on avoiding errors and on efficient programming 387

8.2 Program debugging 387
8.2.1 Operating modes for program testing 388
8.2.1.1 Modes of the SIMOTION devices 388
8.2.1.2 Important information about the life-sign monitoring 390
8.2.1.3 Life-sign monitoring parameters 392
8.2.2 Editing program sources in online mode 392
8.2.3 Symbol Browser 393
8.2.3.1 Properties of the symbol browser 393
8.2.3.2 Using the symbol browser 394
8.2.4 Monitoring variables in watch table 397
8.2.4.1 Variables in the watch table 397
8.2.4.2 Using watch tables 397
8.2.5 Variable status 399
8.2.6 Program run 400
8.2.6.1 Program run: Display code location and call path 400
8.2.6.2 Program run parameters 401
8.2.6.3 Program run toolbar 402
8.2.7 Program status 402
8.2.7.1 Properties of the program status 402
8.2.7.2 Using the status program 404
8.2.7.3 Call path for program status 406
8.2.7.4 Parameter call path status program 407
8.2.8 Breakpoints 407
8.2.8.1 General procedure for setting breakpoints 407
8.2.8.2 Setting the debug mode 408
8.2.8.3 Define the debug task group 409
8.2.8.4 Debug task group parameters 411
8.2.8.5 Debug table parameter 411
8.2.8.6 Setting breakpoints 412
8.2.8.7 Breakpoints toolbar 414
8.2.8.8 Defining the call path for a single breakpoint 415
8.2.8.9 Breakpoint call path / task selection parameters 417
8.2.8.10 Defining the call path for all breakpoints 418
8.2.8.11 Call path / task selection parameters of all breakpoints per POU 420
8.2.8.12 Activating breakpoints 420
8.2.8.13 Display call stack 423
8.2.8.14 Breakpoints call stack parameter 424
8.2.8.15 Resuming program execution 425
8.2.9 Task status function bar 425
8.2.10 Trace 426
8.2.11 Project comparison 426

A Appendix 427

A.1 Formal Language Description 427
A.1.1 Language description resources 427
A.1.1.1 Formatted rules (lexical rules) 427
A.1.1.2 Unformatted rules (syntactic rules) 428
A.1.2 Basic elements (terminals) 429
A.1.2.1 Letters, digits and other characters 429

A.1.2.2 Formatting characters and separators in the rules ... 429
A.1.2.3 Formatting characters and separators for constants ... 430
A.1.2.4 Predefined identifiers for process image access ... 431
A.1.2.5 Operators ... 432
A.1.2.6 Reserved words ... 432
A.1.3 Rules ... 441
A.1.3.1 Identifiers ... 442
A.1.3.2 Notation for constants (literals) ... 442
A.1.3.3 Comments ... 450
A.1.3.4 Sections of the ST source file ... 451
A.1.3.5 Structures of ST source files ... 451
A.1.3.6 Program organization units (POU) ... 454
A.1.3.7 Declaration sections ... 459
A.1.3.8 Structure of the declaration blocks ... 462
A.1.3.9 Data types ... 472
A.1.3.10 Statement section ... 477
A.1.3.11 Value assignments and operations ... 478
A.1.3.12 Call of functions, function blocks and methods ... 484
A.1.3.13 Control statements ... 487
A.1.3.14 Pragmas ... 492

A.2 Compiler Error Messages and Remedies ... 493
A.2.1 File access errors (1000 ... 1100) ... 493
A.2.2 Scanner errors (2001, 2002) ... 494
A.2.3 Declaration errors in POU (3002 ... 3161) ... 494
A.2.4 Declaration errors in data type declarations (4001 ... 4105) ... 497
A.2.5 Declaration errors in variable declarations (5001 ... 5509) ... 498
A.2.6 Errors in the expression (6001 ... 6302) ... 502
A.2.7 Syntax errors, errors in the expression (7000 ... 7014) ... 506
A.2.8 Error when linking a source file (8001 ... 8010) ... 508
A.2.9 Errors while loading the interface of another UNIT or a technology package (10000 ... 10101) ... 508
A.2.10 Implementation restrictions (15001 ... 15700) ... 511
A.2.11 Warnings (16001 .. 16700) ... 512
A.2.12 Non-assigned compiler messages (22000 ... 22002) ... 518
A.2.13 Information (32010 ... 32653) ... 519

A.3 Template for Example Unit ... 521
A.3.1 Preliminary information ... 521
A.3.2 Type definition in the interface ... 522
A.3.3 Variable declaration in the interface ... 523
A.3.4 Implementation ... 524
A.3.5 Function ... 525
A.3.6 Function block ... 527
A.3.7 Program ... 528

Index ... 531

Fundamental safety instructions

1

1.1 General safety instructions

⚠ WARNING

Danger to life if the safety instructions and residual risks are not observed

The non-observance of the safety instructions and residual risks stated in the associated hardware documentation can result in accidents with severe injuries or death.

- Observe the safety instructions given in the hardware documentation.
- Consider the residual risks for the risk evaluation.

1.1.1 Malfunctions of the machine as a result of incorrect or changed parameter settings

⚠ WARNING

Malfunctions of the machine as a result of incorrect or changed parameter settings

As a result of incorrect or changed parameterization, machines can malfunction, which in turn can lead to injuries or death.

- Protect the parameterization against unauthorized access.
- Handle possible malfunctions by taking suitable measures, e.g. emergency stop or emergency off.

1.2 Cybersecurity information

Siemens provides products and solutions with industrial cybersecurity functions that support the secure operation of plants, systems, machines and networks.

In order to protect plants, systems, machines and networks against cyber threats, it is necessary to implement – and continuously maintain – a holistic, state-of-the-art industrial cybersecurity concept. Siemens' products and solutions constitute one element of such a concept.

Customers are responsible for preventing unauthorized access to their plants, systems, machines and networks. Such systems, machines and components should only be connected to an enterprise network or the internet if and to the extent such a connection is necessary and only when appropriate security measures (e.g. firewalls and/or network segmentation) are in place.

For additional information on industrial cybersecurity measures that may be implemented, please visit
https://www.siemens.com/cybersecurity-industry.

Siemens' products and solutions undergo continuous development to make them more secure. Siemens strongly recommends that product updates are applied as soon as they are available and that the latest product versions are used. Use of product versions that are no longer supported, and failure to apply the latest updates may increase customer's exposure to cyber threats.

To stay informed about product updates, subscribe to the Siemens Industrial Cybersecurity RSS Feed under
https://new.siemens.com/cert.

1.3 Note regarding the general data protection regulation

Siemens observes the principles of data protection, in particular the principle of data minimization (privacy by design). For this product this means:

The product does not process / store any personal data, only technical functional data (e.g. time stamp). If the user links this data with other data (e.g. shift plans) or stores personal data on the same medium (e.g. hard disk) and thus establishes a personal reference, the user must ensure compliance with data protection regulations.

1.4 Danger to life due to software manipulation when using removable storage media

⚠ WARNING

Danger to life due to software manipulation when using removable storage media

The storage of files on removable storage media involves a high risk of infection, e.g. via viruses or malware. Incorrect parameter assignment can cause machines to malfunction, which can lead to injuries or death.

- Protect the files on removable storage media against harmful software through appropriate protective measures, e.g. virus scanners.

Introduction 2

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.

2.1 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

2.2 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.

2.3 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.

2.4 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

3

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

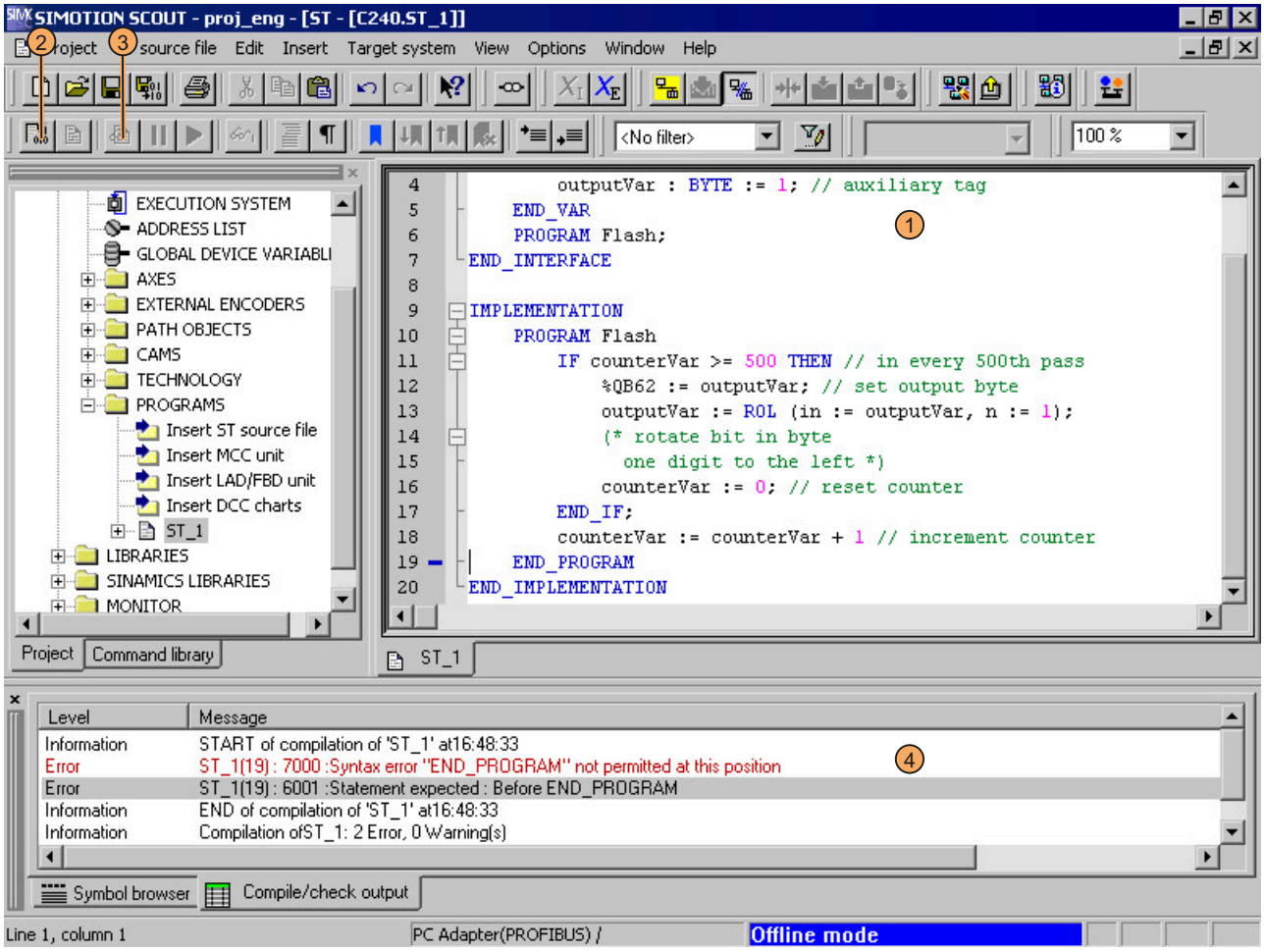
3.1 Integration of ST in SCOUT

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.

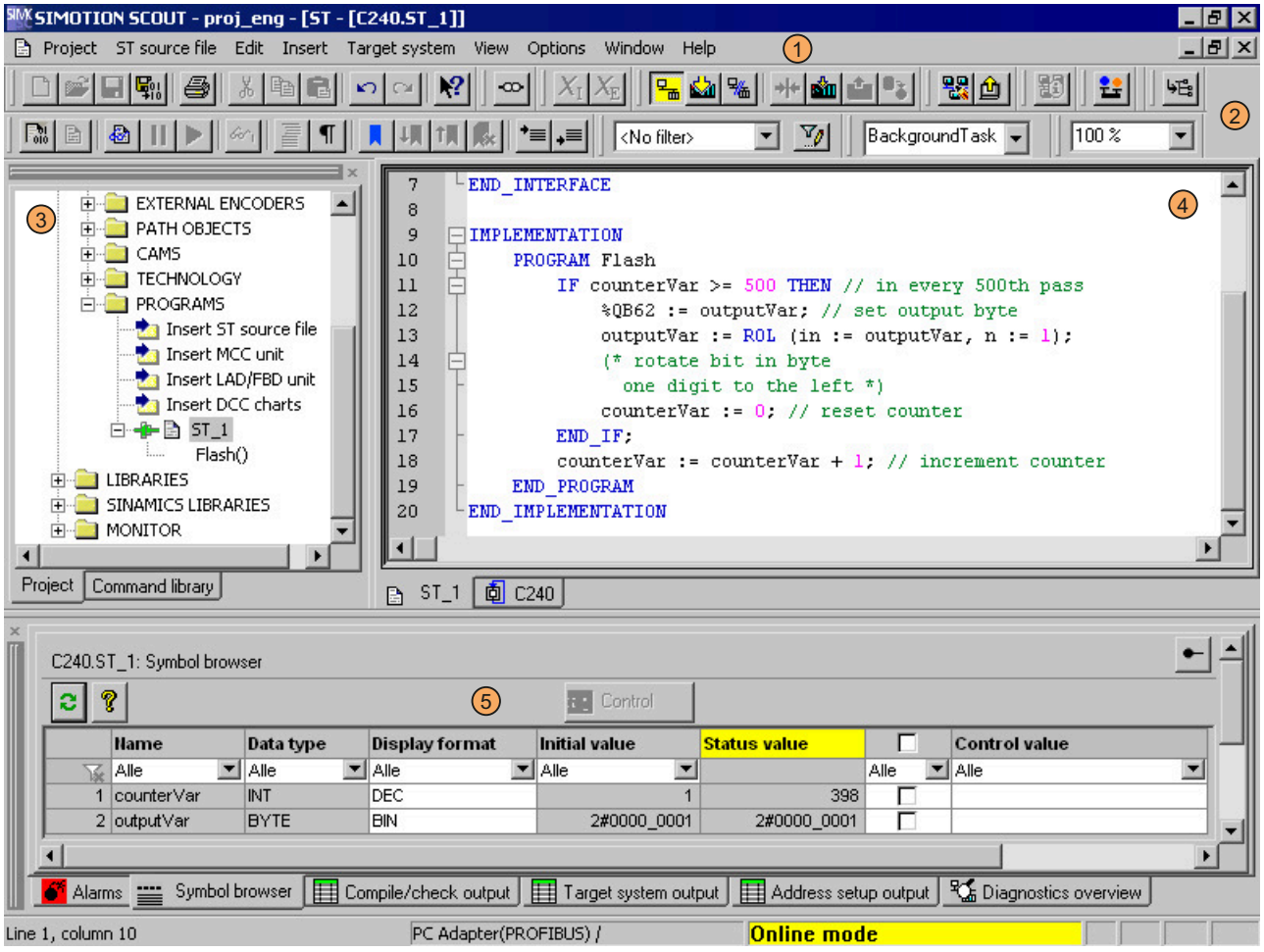


① Editor
② Compiler
③ Program status
④ Detail view (Compile/check output tab)

Figure 3-1 Development environment of ST

3.1.1 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 3-2 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).
- Detail 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.

3.2 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.

3.3 Working with the ST editor and the compiler

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

3.3.1 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 97): 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 99) 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 67) for code generation and message display.
   - **Additional settings** tab: Definitions for preprocessor (Page 75)
8. Confirm with **OK**.

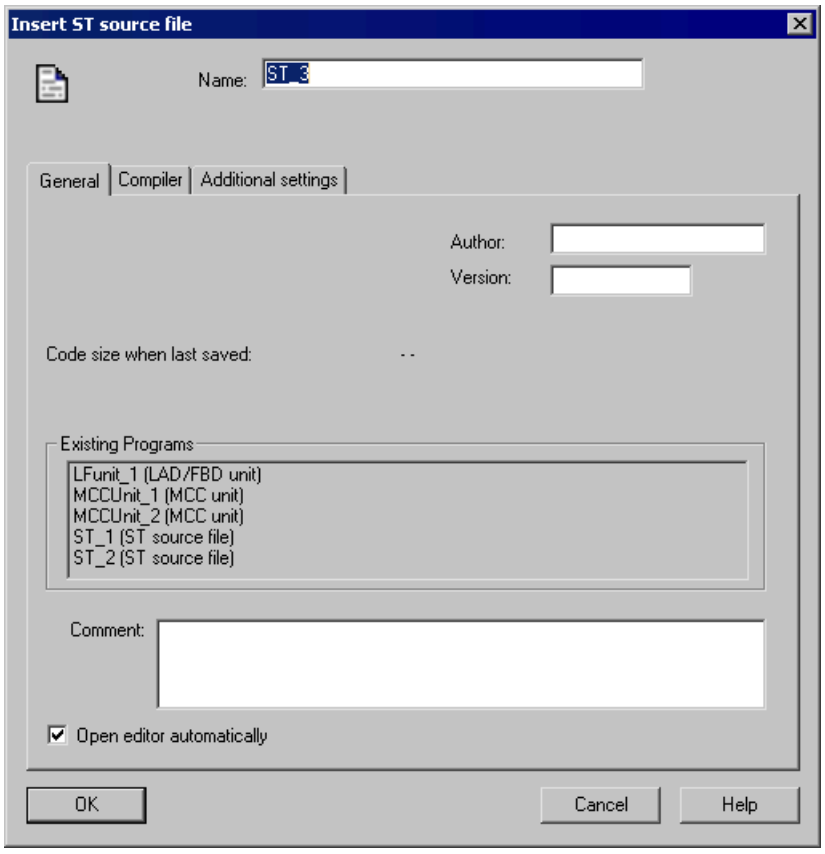


Figure 3-3 Insert ST source file

Note

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**.

3.3.2 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 36) and bookmarks (Page 47). Multiple units can be opened.

Note

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

3.3.3 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 67) for code generation and message display.
   - **Additional settings** tab: Definitions for the preprocessor (Page 75) and display of compiler options (Page 73) as specified in the current settings of the compiler. You also have the possibility of setting the ST source as read-only (Read-Only-mode).
   - **Compilation** tab: Display of the compiler options (Page 73) 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.

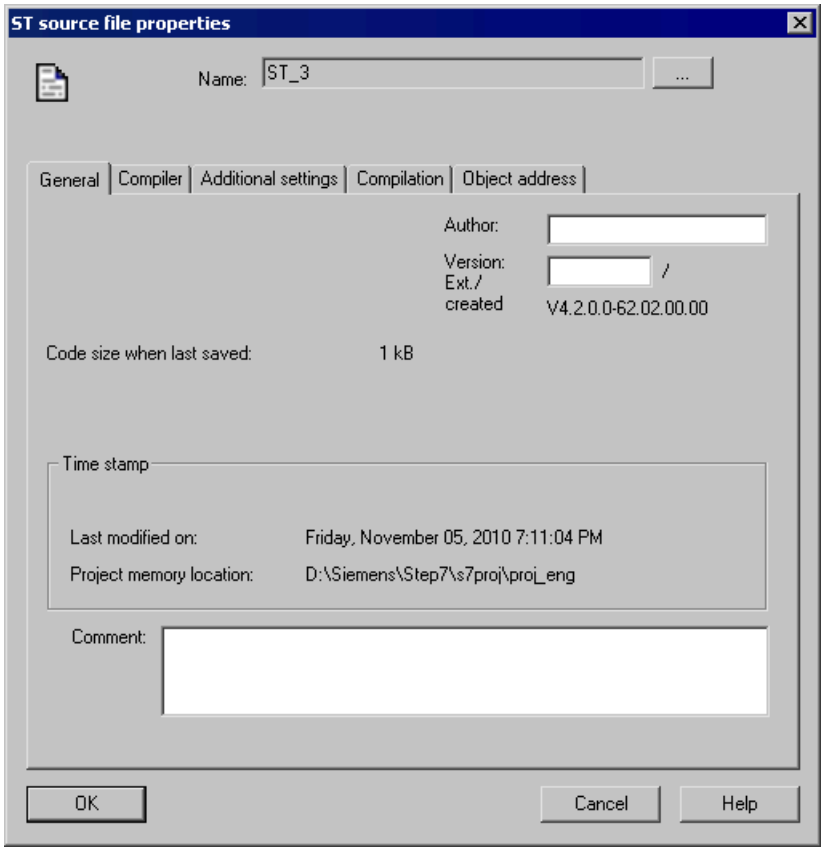


Figure 3-4 Properties of an ST source file

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 99) are not allowed.

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

Note

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!

Read-only ST sources

You can set ST sources as read-only to prevent unintentional code changes during commissioning. To do this, navigate to "Additional settings" and set the option "Read-Only-Mode". Functions that would change the ST source are disabled in read-only mode. For this reason, the corresponding context menu entries and icons are inactive.

The following functions can still be used when an ST source is read-only:

- Status program
- Breakpoints
- Displaying the variable in the tooltip
- Pasting/removing bookmarks, jumping to bookmarks
- Selecting variables
- Searching (Ctrl +F)
- Compiling

3.3.4 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 and drop
- Marking the variable occurrence
- Selecting the same level of the control structure

- Searching and replacing in Selection
- Menu commands and shortcuts

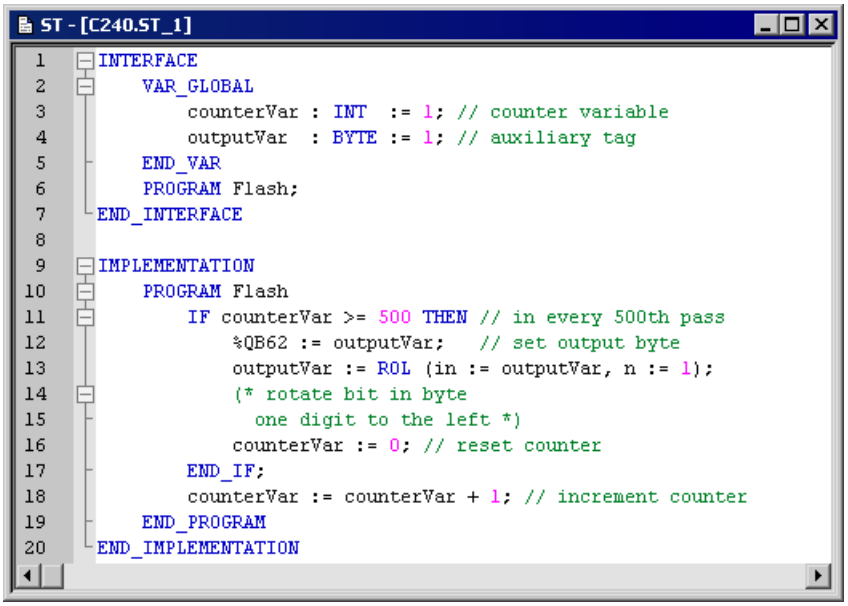


Figure 3-5 Opened ST source file in the ST editor

3.3.4.1 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

3.3.4.2 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.

3.3.4.3 Marking the variable occurrence

The functionality allows you to see all occurrences of a selected variable highlighted.

The functionality is automatically activated when a variable is selected or the cursor position is on a variable name. The background color of the selected occurrence does not change. All other occurrences are marked with a light gray background.

The scope of the variables, the namespaces are also marked if they are explicitly written in front of the variable name.

You can move up and down the line with Alt+Arrow or select content with Ctrl+Shift+Arrow.

3.3.4.4 Selecting the same level of the control structure

If several IF statements or control structures are nested within one another (for example, IF..ELSIF..ELSE..END_IF), it is difficult to make out which keywords belong to one another. You can make keywords that belong to the same level visible. If the cursor is located on one of the keywords (can be any keyword) or if a keyword is selected, all the other keywords that belong to the selected keyword are highlighted in color.

3.3.4.5 Searching and replacing in Selection

The functionality allows you to apply Find and Replace to a limited text area that you have selected.

You select a part of a ST source code and open 'Edit/Replace...' or press CTRL+H. The 'Only in selection' check box is in the open window. If this option is selected, the search and replace feature is applied only to the selection.

3.3.4.6 Settings of the ST editor

Procedure:

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

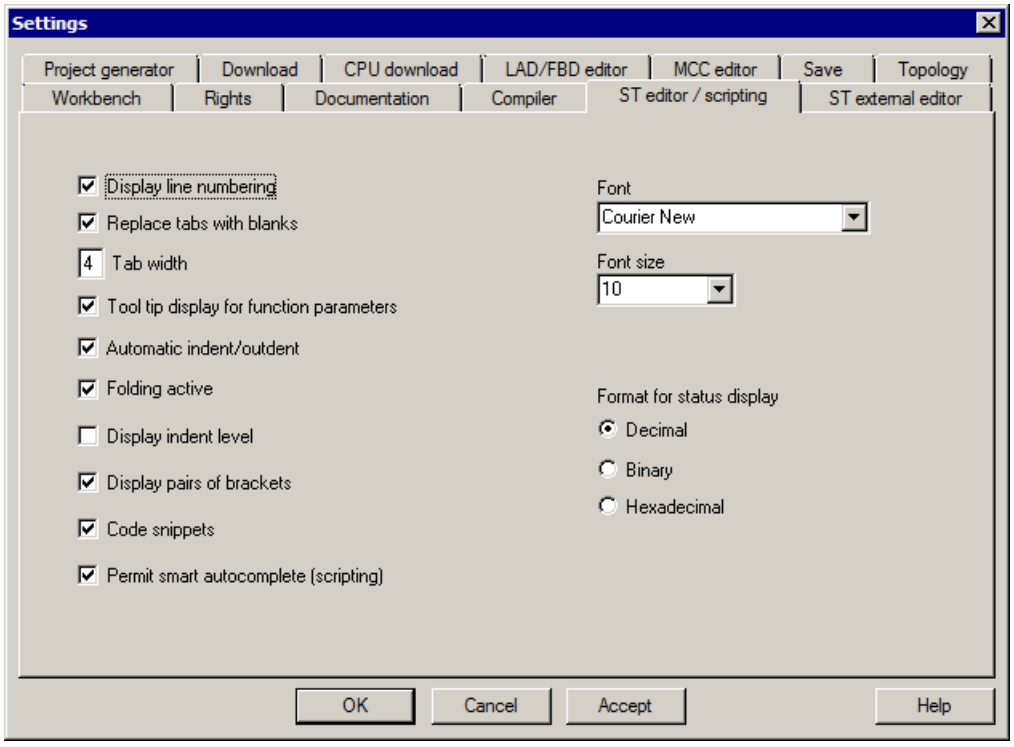


Figure 3-6 ST editor / scripting settings

The settings also apply to the script editor.

The table below contains a description of the individual parameters.

Table 3-1 ST editor / scripting parameter settings

Parameter	Description
Display line numbering	If active, the line numbers are displayed. See: Other ST editor tools (Page 53).
Replace tabs with blanks	You select here how text indentation is performed (for the automatic indentation or by pressing the Tab key): • If active: By adding the appropriate number of space characters ($20). • If inactive: By adding the tab character ($09). See: Indentations and tabs (Page 34).
Tab width	Number of characters skipped by a tab. See: Indentations and tabs (Page 34).
Tooltip display for function parameters	When active, the parameters are displayed as tooltips for the functions.
Automatic indent/outdent	If active, for the text input, source file sections and blocks are indented automatically by the set tab width. See: Indentations and tabs (Page 34).
Folding active	If active, the column with the folding information is displayed at the left-hand side next to the edit area. You can then hide blocks in an ST source file so that only the first line of the block remains visible. See: Folding (showing and hiding blocks) (Page 36).
Display indentation level	If active, you can optically highlight the indent and outdent for blocks using vertical help lines (in accordance with the set tab size). See: Indentations and tabs (Page 34).
Display bracket pairs	If active, the associated bracket of the pair that belongs to the bracket where the cursor is located will be found and optically highlighted. See: Other ST editor tools (Page 53).
Code snippets	If active, after input of certain keywords (such as IF, WHILE, VAR), a window opens to display a code snippet for the associated keyword (e.g. IF (*condition*) ... END_IF). This code snippet can be transferred to the ST source file. See: Code snippets (Page 51).
Permit Smart-autocomplete (scripting) [1]	If active, automatic completion of terms is permitted for the scripting editor.
Font	Font for the display of the text in the ST editor. All non-proportionally spaced fonts installed on the PC are available for selection.
Font size	Font size (in pt) for the display of the text in the ST editor. See: Change the font size in the ST editor (Page 42).

Parameter	Description
Format for status display [2]	Format in which the variable values with bit data type are displayed for the program status or the variable status (for ST editor only). See: Properties of the program status (Page 402), variable status (Page 399)
Vertical margin settings	Addition of a colored line in the ST editor window at a user-defined position.

1 Only for the scripting editor

2 Only for ST editor

3.3.4.7 Indentations and tabs

Specify tab width

The standard tab width for all ST sources is specified in the settings of the ST editor (Page 32).

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 32) 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 32).

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) or the variable block 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).

How to call the function via the shortcut menu:

1. Select the text area in the ST editor that you want to format (see Select text (Page 43)).
2. Select the **View > Format current selection** context menu.

How to call the function using the key combination:

1. In the ST editor, select the text area that you want to format (see Select text (Page 43)).
2. Press **CTRL+SHIFT+F8**.

How to call the function via the shortcut menu:

1. In the ST editor, select the text area that you want to format (see Select text (Page 43)).
2. Press the **"Format current selection"** button.

Note

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 43)).
   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).

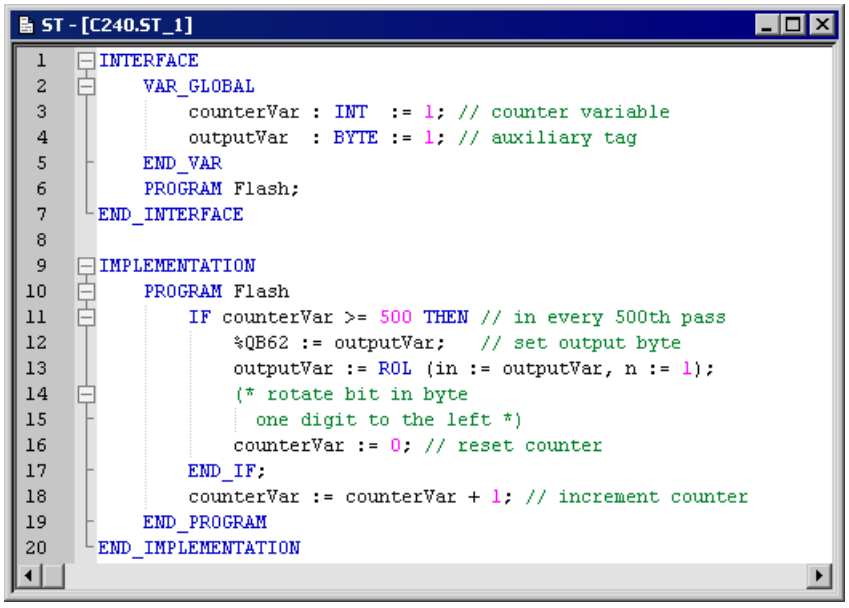


Figure 3-7 ST source with visible indent aid

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 32).

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

3.3.4.8 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.

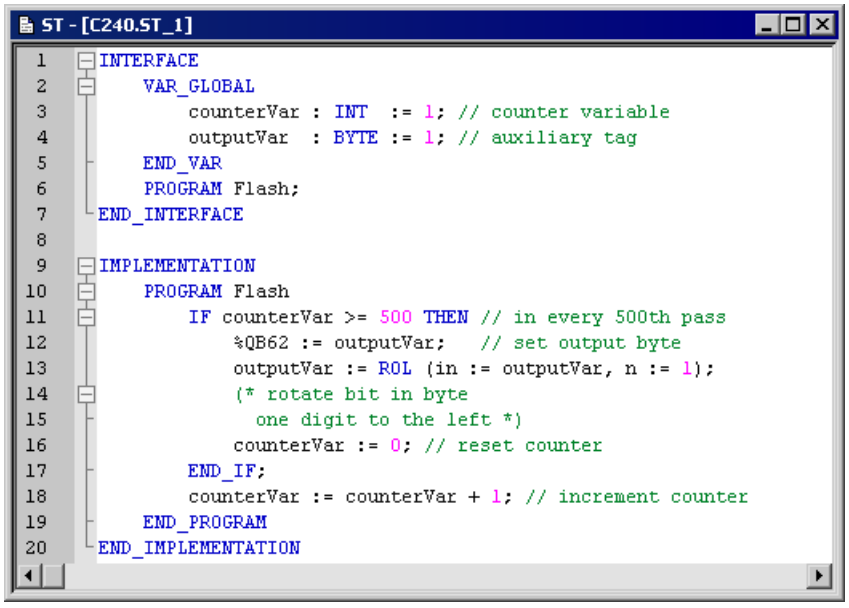


Figure 3-8 ST source file for which all blocks are shown

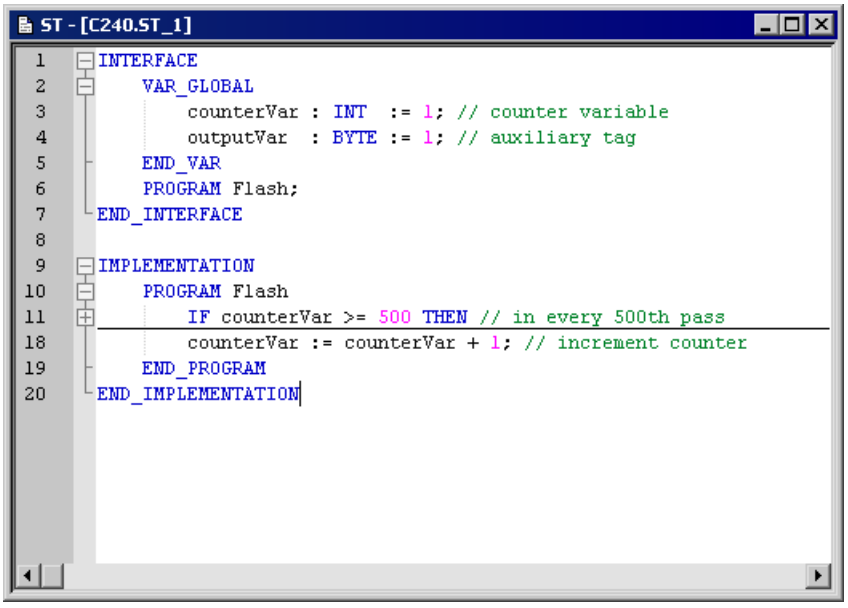


Figure 3-9 ST source file with hidden IF block (including block comment)

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 32).

  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 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 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 76).

3.3.4.9 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 404) test function is being used.

The snippet editor is disabled when split window is used.

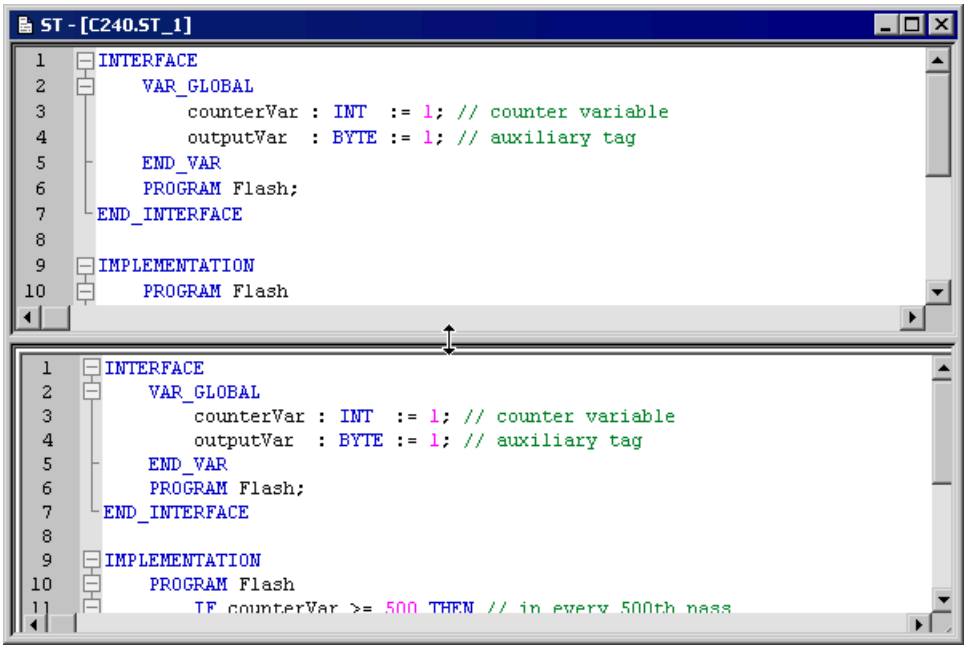


Figure 3-10 ST editor with split window

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.

3.3.4.10 Displaying spaces, tabs and line ends

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

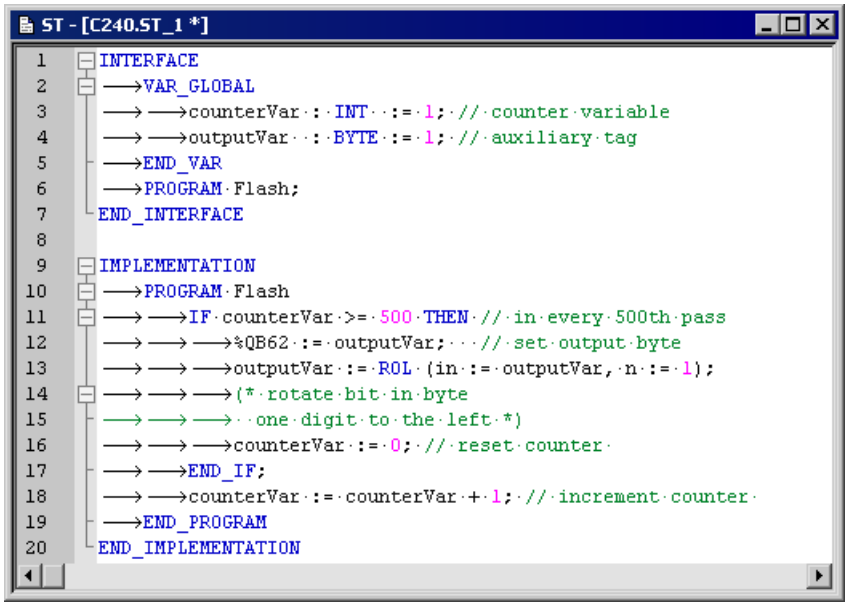


Figure 3-11 ST source file with visible spaces and tabs

In addition, you can display the line end (CR/LF).

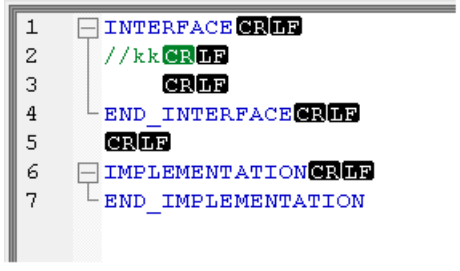


Figure 3-12 ST source with visible line end

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.

3.3.4.11 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.

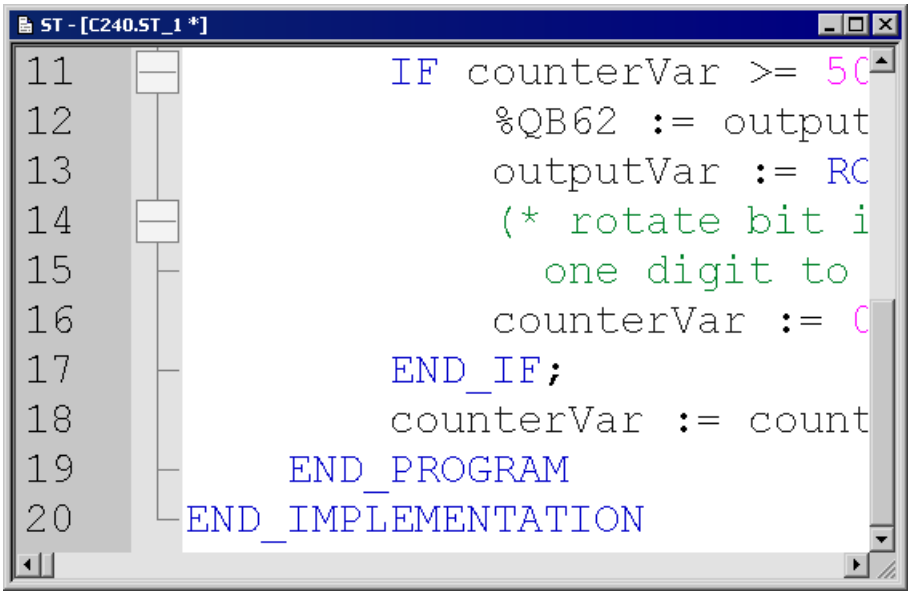


Figure 3-13 Increased size display of the ST source

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 32)).
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.

3.3.4.12 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 59).

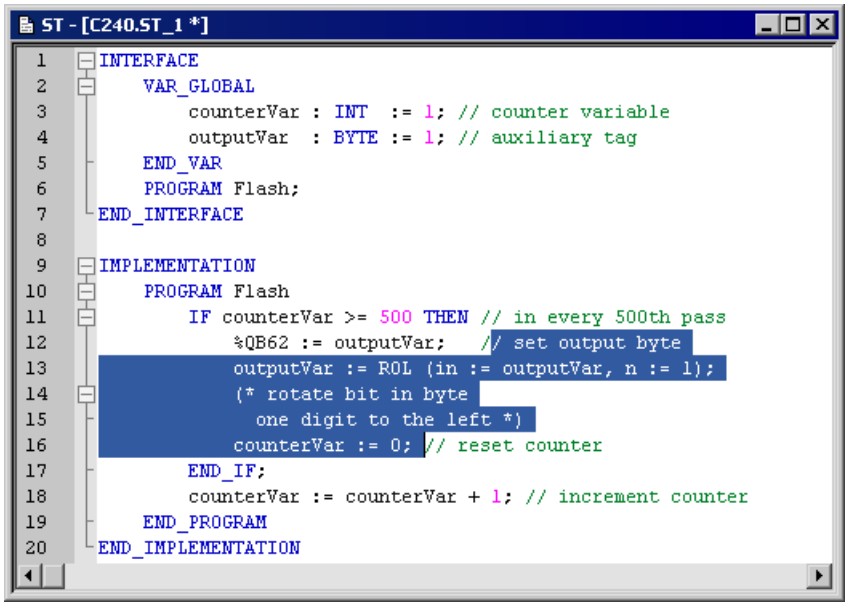


Figure 3-14 ST source with selected lines of text

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 59).

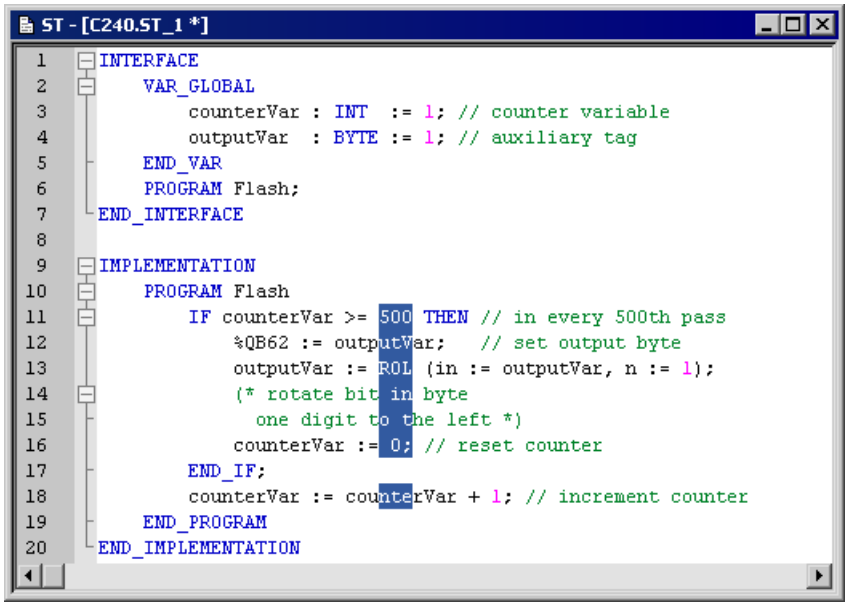


Figure 3-15 ST source with selected columns of text

Selecting a single line

How to select a single line:

- Left-click to the right of 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.

3.3.4.13 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 43).
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.

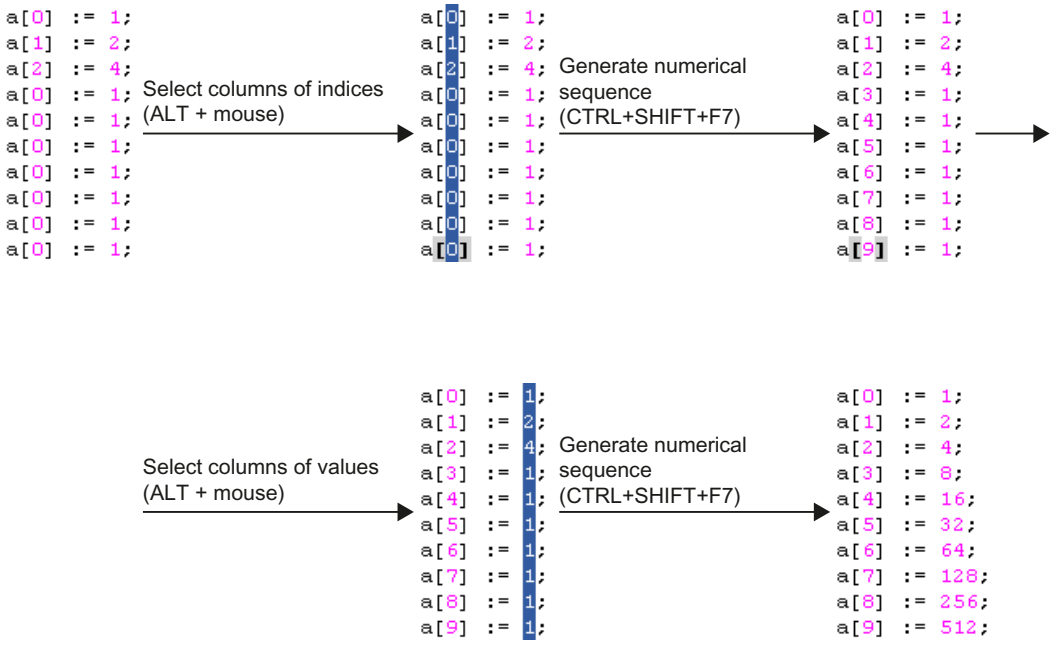


Figure 3-16 Example for generating series of numbers in the ST editor

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 a_3 ($j \geq 0$).
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 3-2 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.						

Note

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.

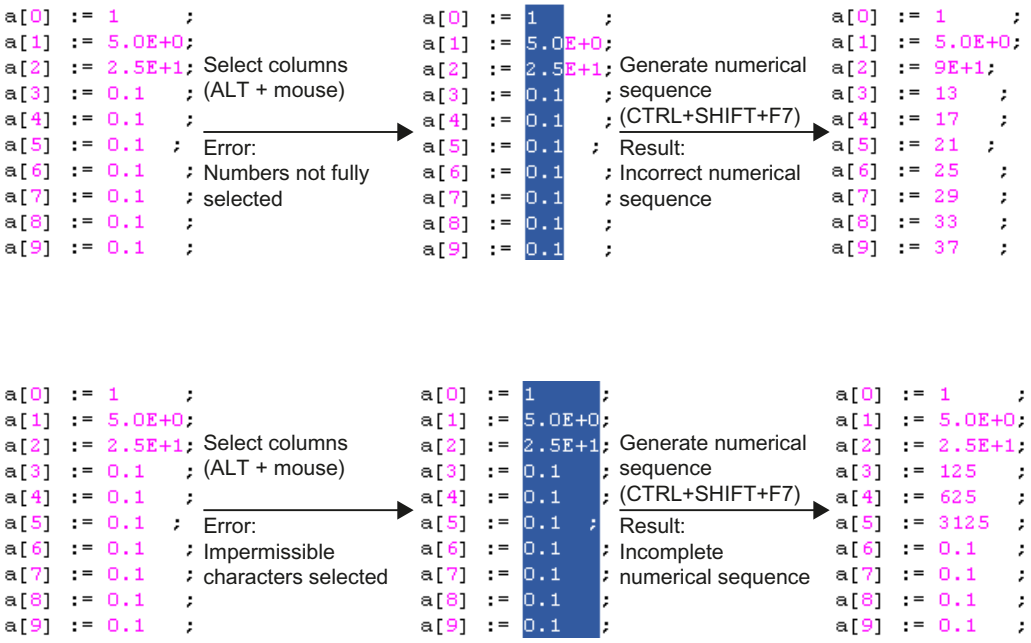


Figure 3-17 Examples of how incorrect selections cause unwanted results when generating a series of numbers in the ST editor

3.3.4.14 Use bookmarks

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

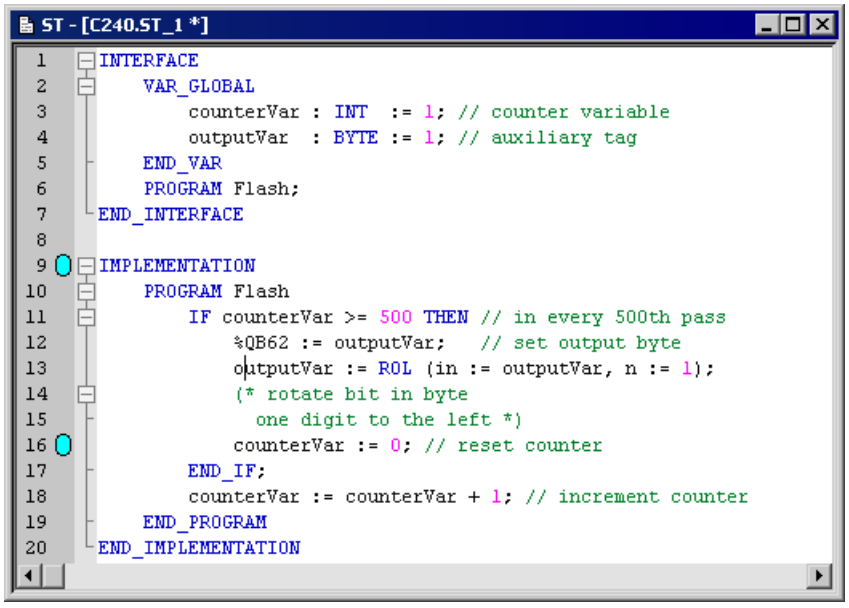


Figure 3-18 ST source with bookmarks

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.

3.3.4.15 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.

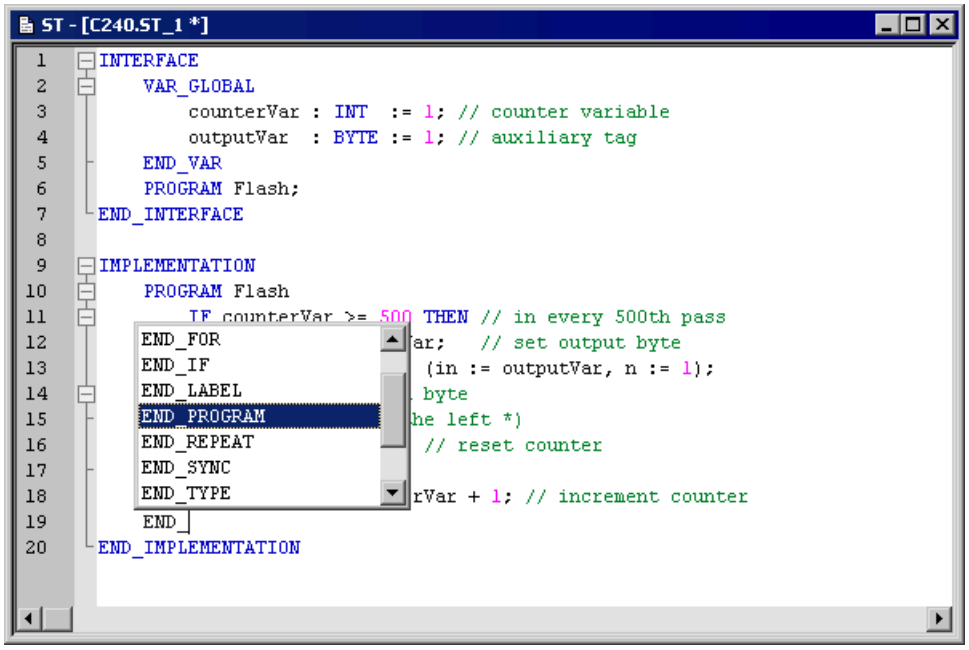


Figure 3-19 ST editor, automatic completion of an identifier (e.g. END_)

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 options 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 program sources used

Note

Identifiers from the own ST source or from program sources used 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

3.3.4.16 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.
  If a library is connected with an online CPU and is consistent online, the online functions are available under this library for all units.
- SIMOTION MCC or SIMOTION LAD/FBD programming languages:
  The corresponding POU (MCC chart, LAD/FBD program) is opened in the associated editor.

3.3.4.17 Code snippets

After the complete input of certain keywords (such as IF, WHILE, VAR), a window opens above the cursor to display a code snippet for the associated keyword (e.g. IF (*condition*) ... END_IF).

Further keywords with the same initial letter with available code snippets are displayed in the lower part of this window.

Pressing the **TAB** key transfers this code snippet to the ST source file. Pressing any other key closes this window.

Activating and deactivating code snippets

Code snippets must be activated before they can be used.

How to activate or deactivate code snippets:

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

  This setting is not saved when the ST source file is closed.
- For all open ST source files:
  - Activate or deactivate the **Code snippets** checkbox in the ST editor settings.

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

Creating user-defined snippets

You can create your own snippets for use in the ST editor. To do this, use the snippet editor, which you can display in the partitioned window at the same time as the ST editor. Observe the following rules for creation:

Rule	Description
Creating and naming snippets	You create new snippets with the snippet editor. Enter a unique name. Permissible characters are A-Z, a-z, 0-9 and _.
Renaming snippets	You can rename snippets that have already been created.
Deleting snippets	You can delete snippets that have already been created. Perform the Save function afterwards. Only then can the name of the deleted snippet be used again for new snippets.

Working with the snippet editor

Display the snippet editor via "ST source file > Snippet editor". Alternatively, you can reach the snippet editor via "ST editor > View > Snippet editor". The snippet editor shares the working area with the associated ST editor. In this way, the snippet editor is always tied to one, and only one, existing ST editor window. The snippet editor consists of three main sections:

- Snippet toolbar
- Snippet tree
- Snippet code editor

Snippet toolbar

The toolbar contains the following functionalities:

Function	Description
Load snippets into the editor – without saving	This function loads the changes that have been made in the snippet editor into the ST editor.
Save	This function saves the selected snippet or all of the snippets under the selected node in the tree.
Save all	This function saves all of the changes that have been made since the last save.
Reject	This function rejects the changes that have been made to the selected snippet or all of the changes in the selected node in the tree.
Reject all	This function rejects all of the changes that have been made since the last save.
Import	This function loads the user-defined snippets from the XML input file. The validity of the file is checked. If there is an error, a corresponding message is output. All existing user-defined snippers are deleted during the import.
Export	This function saves the user-defined snippets into an XML file. For this purpose, you create a new file or overwrite an existing one. The changes in the snippet are saved before the export.
Help	This function calls the context-sensitive online help.
Close	This function closes the snippet editor. Save your snippets before closing the editor.

Snippet tree

The snippet editor shows the hierarchy of all of the available snippets in a clear tree structure. The tree has a root node "Snippets" with two subordinate elements by the name of "STSnippets" for non_OOP- and "STSnippetsOOP for OOP-relevant snippets". Both contain pre-defined and user-defined snippets. The snippet IDs in the snippet tree are sorted alphabetically and all of the pre-defined snippets (recognizable by the lock icon) are listed first, in alphabetical order.

Only user-defined snippets can be created, changed, or deleted. Master elements, scope IDs or pre-defined snippets cannot be changed, deleted, or created. After you have created a new snippet, the new snippet key (name) will be alphabetically sorted into the user-defined snippets in the right position in the tree.

Snippet code editor

You insert the snippets in the snippet tree via the context menu. After inserting, assign the name and edit the snippet in the snippet code editor.

The snippet code editor is a simple text editor field in which you edit the associated part of the code (the currently selected element in the snippet tree). The text can be entered or inserted from the clipboard. Pre-defined snippets can be selected and copied, but are not edited.

Navigating between the identifiers

Snippets can have more identifiers marked with #identifier as placeholders to indicate that an identifier is missing. When a snippet is inserted into the ST source, the first #identifier that can be overwritten by the correct identifier is always marked. You then need to select all other identifiers. You jump to the next identifier with TAB. You jump to the previous identifier with Shift+TAB.

3.3.4.18 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 32).

  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 32).

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

Setting the focus on the source in the project navigation

To navigate more easily between ST source code and PNV, you can select or mark an object in the ST code and set the focus on it in the PNV tree.

- with a key combination: Ctrl+Alt+J (J for jump)
- via the shortcut menu: "Focus on source in PNV"

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.

3.3.4.19 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.

3.3.4.20 ST editor toolbar

This toolbar contains important operating actions for programming:

Table 3-3 ST editor toolbar

Symbol	Meaning
	Accept and compile Click this symbol to transfer the active ST source file to the project and compile it into executable code. See: Starting the compiler (Page 62).
	Insert ST source file Click this icon to create a new ST source file. The icon is active only when the **PROGRAMS** folder where the ST source file is to be saved is selected in the project navigator. See: Insert ST source file (Page 25).

Symbol	Meaning
	Program status Click this icon to start the **program status** test mode. During the program execution, you can monitor the values of the variables marked in the ST source. The following prerequisites are necessary: 1. The program must be compiled with the appropriate compiler option. 2. The project and the program must be loaded into the target system. 3. An online connection to the target system must have been established. Reclick this icon to end the **program status**. See: Using the program status (Page 404). **Note:** Status program and the snippet editor cannot be active at the same time. • Close the snippet editor to open the status program.
	Stop monitoring of the program variables Click this icon in the **program status** test mode to stop the monitoring of the program variables. See Using the program status (Page 404).
	Continue monitoring of the program variables Click this icon in the **program status** test mode to continue the monitoring of the program variables. See: Using the program status (Page 404).
	Refresh 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. See: Using the program status (Page 404).
	Format current selection Click this symbol to indent the blocks in the selected text area by one tab width, in accordance with the block hierarchy. See Indentations and tabs (Page 34).
	Formatting symbols on/off Click this symbol to show or hide blanks and tabs in the active ST source file. See Displaying blanks and tabs (Page 41).
	Insert/remove bookmark Click this symbol to set a bookmark in the current line of the active ST source file or to delete an existing bookmark. See: Using bookmarks (Page 47).
	Next bookmark Click this symbol to jump to the next bookmark in the active ST source file. See: Using bookmarks (Page 47).
	Previous bookmark Click this symbol to jump to the previous bookmark in the active ST source file. See: Using bookmarks (Page 47).
	Remove all bookmarks Click this symbol to remove all bookmarks from the active ST source file. See: Using bookmarks (Page 47).

Symbol	Meaning
	Go to start of block Click this symbol to move the cursor to the start of the current or higher-level block.
	Go to end of block Click this symbol to move the cursor to the end of the current block.
	Insert comment character Click this icon to convert to comment on the selected lines or line fragments or to insert a comment character. See Comments (Page 112).
	Remove comment character Click this icon to reactivate commented lines or line fragments. See Comments (Page 112).

3.3.4.21 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 3-4 ST editor and script editor context menu

Function	Meaning/information
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: Select this command to open and edit the source of this POU (ST source file, MCC chart, LAD/FBD program). See: Open called block (Page 50).
Refactoring	You can change the case or rename. The function provides a list of suggested renames of the symbols used in the actual unit, which differ in terms of notation from the symbols declared in the cross-references. You can decide which changes need to be applied. To avoid inconsistencies, compile the project afterwards.
Surround with	Select this command to surround the selected text with pre-defined code snippets. E.g. surround the selected variable with an If instruction.
Cut	Select this command to cut the selected area from the ST source file and save it to the clipboard.
Copy	Select this command to copy the selected area to the clipboard.
Paste	Select this command to insert the contents of the clipboard into the ST source file at the current cursor position.
Copying text with syntax coloring	Select this command to copy the selected area in RTF format to the clipboard. This allows the insertion of text including syntax coloring into text processing programs. The copied text cannot be inserted into the ST editor.
Delete	Select this command to delete the selected area or the character to the right of the cursor.

Function		Meaning/information
Select all		Select this command to select all of the text in the ST source file. See: Select text (Page 43).
View		
	Line numbering on/off	Select this command to show or hide the line numbers in the active ST source file. See: Other ST editor tools (Page 53).
	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). See: Indentations and tabs (Page 34).
	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. See: Other ST editor tools (Page 53).
	Formatting symbols	Select this command to show or hide blanks and tabs in the active ST source file. See: Displaying blanks and tabs (Page 41).
	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. See: Folding (showing and hiding blocks) (Page 36).
	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. See: Indentations and tabs (Page 34).
	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. See: Splitting the editor window (Page 40).
	Code snippets	Select this command to activate or deactivate code snippets in the activate ST source file. If active, after input of certain keywords (such as IF, WHILE, VAR), a window opens to display a code snippet for the associated keyword (e.g. IF (*condition*) ... END_IF). This code snippet can be transferred to the ST source file. See: Code snippets (Page 51).
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. See: Using bookmarks (Page 47).
	Next bookmark	Select this command to jump to the next bookmark in the active ST source file. See: Using bookmarks (Page 47).
	Previous bookmark	Select this command to jump to the previous bookmark in the active ST source file. See: Using bookmarks (Page 47).
	Remove all bookmarks	Select this command to remove all bookmarks from the active ST source file. See: Using bookmarks (Page 47).
Print		
	Complete source	Select this command to print the complete active ST source file.
	Selection	Select this command to print the selected lines of the active ST source file.
Variable declaration		Select this command to declare a variable "on-the-fly". See: Declaring variables in the Variable declaration dialog box ("on-the-fly" variable declaration) (Page 311).

Function		Meaning/information
Selection in upper case		Select this command to change the selected text to upper case. See: Other ST editor tools (Page 53).
Selection in lower case		Select this command to change the selected text to lower case. See: Other ST editor tools (Page 53).
Code indentation		Select this command to format the text automatically.
Indent selected area		• If no text is selected: Select this command to move the text on the right of the cursor to the next tab position. • If text is selected in one single line: Select this command to delete the selected text and move the subsequent text to the next tab position. • If text is selected in multiple lines: Select this command to indent the selected area (Page 34). A tab character ($09) or the equivalent number of spaces ($20) will be inserted, depending on the settings for the ST editor (Page 32).
Undo selected area		• If no text is selected: Select this command to move the cursor to the previous tab position. • If text is selected in one single line: Select this command to cancel the selection and move the cursor to the previous tab position. • If text is selected in multiple lines: Select this command to outdent the selected area (Page 34).
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.
Go to line		Select this command to move the cursor to a specific line. A dialog box for specifying the line number is opened.
Set/remove breakpoint [1]		Select this command to set a breakpoint at the selected code position or to remove an existing breakpoint. See: Setting breakpoints (Page 412).
Activate/deactivate breakpoint [1]		Select this command to activate or deactivate the breakpoint at the selected code position. See: Activating breakpoints (Page 420).
Add to watch table [1]		
	New watch table	If the cursor is within a variable or if the variable is selected: Select this command to create a new watch table and to take it into this variable.
	(Name of a watch table)	If the cursor is within a variable or if the variable is selected: Select this command to take the variable into the selected watch table
Go to [1]		
	Local use >>	If the cursor is within a variable or if the variable is selected: Select this command to jump to the next use of the variable in the ST source file.

Function		Meaning/information
	Local use <<	If the cursor is within a variable or if the variable is selected: Select this command to jump back to the previous use of the variable in the ST source file.
	Declaration position	If the cursor is within a variable or if the variable is selected: Select this command to jump to the declaration position of the variable. If the variable is declared in a used source or library, the source or library is opened.
	Points of use	If the cursor is within a variable or if the variable is selected: Select this command to list all the points of use of the variable in the detail view.

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

3.3.4.22 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 3-5 Keyboard shortcuts for ST editor and script editor

Shortcuts	Description
F2	Jump to the next bookmark.
F3	Find next (menu **Edit > Find next**). A search is performed on the last text in the search field, even if it has been closed (see CTRL+F).
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: Move the text on the right of the cursor to the next tab position. • If text is selected in one single line: Delete the selected text and move the subsequent text to the next tab position. • If text is selected in multiple lines: Indent selected area. 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**).

Shortcuts	Description
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.
SHIFT+TAB	• If no text is selected: Jump to the preceding tab position. • If text is selected in one single line: Jump to the preceding tab position. • If text is selected in multiple lines: Outdent selected area.
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**) If text is selected in a single line, this is taken into the search screen form.
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)	Scroll one row up or down. 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**)

Shortcuts	Description
CTRL+SHIFT+G	Replace texts within the project (menu **Edit > Replace in the project**)
CTRL+SHIFT+H	Assign parameters to a call. Under development.
CTRL+SHIFT+L	Go to a line in the ST editor.
CTRL+SHIFT+S	Code indentation.
CTRL+SHIFT+U	Change selected text to upper case.
CTRL+SHIFT+F2	Remove all bookmarks from the ST source file.
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	Show 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+N	Declare variable "on-the-fly" in the Variable declaration dialog box (Page 311).
ALT+SHIFT+L	Change selected text to upper case.
ALT+SHIFT+S	Activate or deactivate code snippets (Page 51).
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.

Shortcuts	Description
ALT+SHIFT+PG DN	Move down one page. Select columns of text up to the new cursor position.
ALT+Arrow key	Moves the selected code section up or down in the editor.

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

Table 3-6 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.

3.3.5 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.

3.3.5.1 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.

3.3.6 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, see Global settings of the compiler (Page 63)
- Locally for an individual ST source file within the SIMOTION project, see Local settings of the compiler (Page 67)

3.3.6.1 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**.

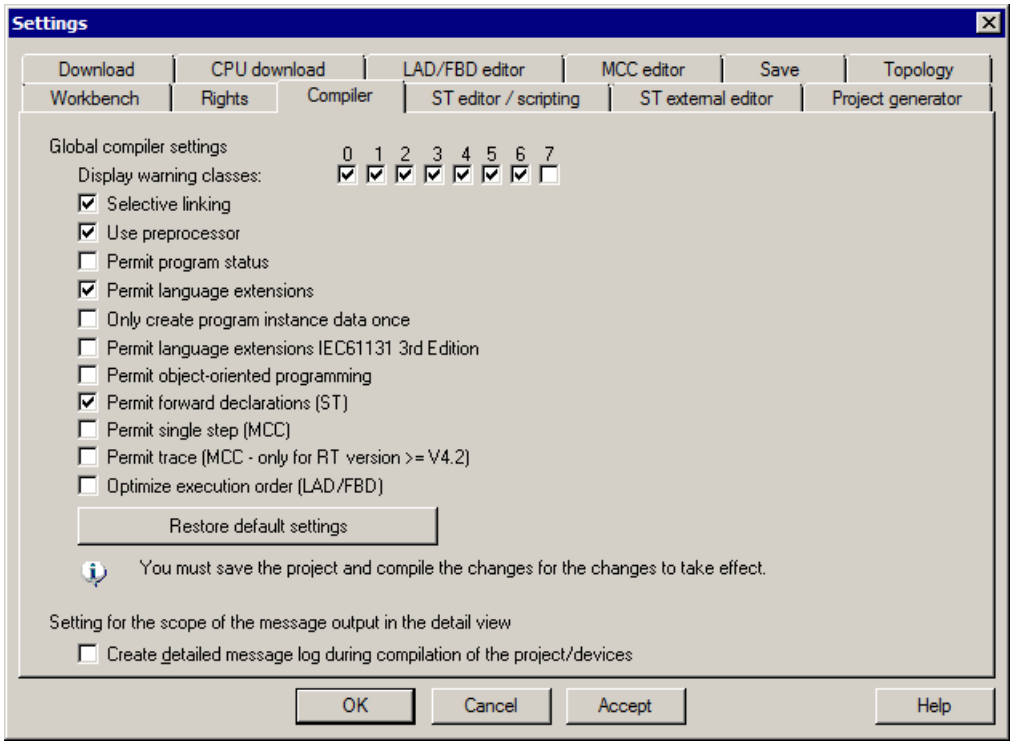


Figure 3-20 Global compiler settings

Parameter

Table 3-7 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: **Active**: The compiler issues warnings and information for the selected class. **Inactive**: The compiler suppresses warnings and information for the respective class. See also Meanings of the warning classes (Page 73).
Selective linking[1]	**Active** (standard): Unused code is removed from the executable program. **Inactive**: Unused code is retained in the executable program.
Use preprocessor[1]	**Active**: Preprocessor is used. **Inactive** (standard): Preprocessor is not used. See Controlling the preprocessor (Page 369).
Enable program status[1]	**Active**: Additional program code is generated to enable monitoring of program variables (including local variables). **Inactive** (standard): Program status not possible. See Properties of the program status (Page 402).

Parameter	Description
Permit language extensions[1]	**Active**: Language elements are permitted that do not comply with IEC 61131-3. • Direct bit access to variables of a bit data type (Page 148) • Accessing the input parameter of a function block while outside the function block (Page 201) • Calling a program while in a different program (Page 207) **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 207). This setting also improves the download in RUN mode, see corresponding section in the Basic Functions Function Manual. **Inactive** (standard): The local variables of a program are stored according to the task assignment in the user memory of the respective task. See Memory areas of the variable types (Page 295).
Permit additional languages, IEC61131 3rd edition[1]	**Active**: Additional language elements can be used in accordance with IEC 61131-3 3rd edition. The corresponding keywords are locked as reserved or protected identifiers. • Nested block comments (Page 112) • CONTINUE statement (Page 171) • Standard-compliant system functions LOWER_BOUND and UPPER_BOUND (Page 192) for determining the limits of a dynamic ARRAY • Additional system functions (see Basic Functions Function Manual): – FROM_BIG_ENDIAN – FROM_LITTLE_ENDIAN – IS_VALID – TO_BIG_ENDIAN – TO_LITTLE_ENDIAN **Inactive** (standard): The additional language elements cannot be used. The corresponding keywords are not locked. **Note** When saving the project in the old project format: Projects in which language elements that require this compiler option are used cannot be compiled correctly in SIMOTION SCOUT versions earlier than V4.5.
Permit object-oriented programming[1]	**Active**: With the ST programming language additional language elements for object-oriented programming can be used in accordance with IEC 61131-3 3rd edition. The corresponding key words are locked as reserved or protected identifiers in all programming languages. **Inactive** (standard): The keywords for object-oriented programming are not locked in all programming languages. The additional language elements cannot be used with the ST programming language. **Note** Independently of the setting the program organization units generated in ST sources using object-oriented programming (e.g. classes, methods) can be used in program sources for all programming languages. When saving the project in the old project format: Projects in which language elements that require this compiler option are used cannot be compiled correctly in SIMOTION SCOUT versions earlier than V4.5.

Parameter	Description
Permit forward declarations (ST)	Only for the ST programming language. Forward declarations enable you to use program organization units (POUs) before they are fully defined. Prototype declarations of program organization units are possible prior to their use, but are only required for an instance declaration of a function block or a class. **Active**: Forward declarations are permitted. **Inactive** (standard): Forward declarations are not permitted. See Forward declarations (Page 381). Forward declarations are always permitted with the MCC and LAD/FBD programming languages. It is also possible to make a local setting on the ST source file (Page 67). Please also refer to the description of the effectiveness of global or local compiler settings (Page 71).
Permit single step (MCC)	Only for the MCC programming language. **Active**: An additional program code is created which enables individual program steps to be monitored. **Inactive**: Single step is not possible. This function facilitates debugging of your program. See "Tracking single steps in the program" in the MCC Programming Manual. 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 71).
Permit trace (MCC - only for RT versions >= 4.2)	Only for the MCC programming language and for SIMOTION Kernel as of version V4.2. **Active**: An additional program code is created which enables monitoring of the program execution in program branches which are executed cyclically. **Inactive**: Trace is not possible. This function facilitates debugging of your program. See "Tracking program execution per trace" in the MCC Programming Manual. 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 71).
Optimize execution order (LAD/FBD)	Only for LAD/FBD programming languages. **Active**: LAD/FBD networks are calculated in the optimized execution order. **Inactive**: LAD/FBD networks are calculated in the non-optimized execution order.
Default setting	Click the button to restore the default setting.
Create detailed message log during compilation of the project/devices[2]	Here, you can control the scope of the message log that will be displayed in the workbench's detail view when you call the **Save and compile changes** command in SIMOTION SCOUT. **Active**: A detailed message log is created that is similar to that for single compilation of an ST source file. **Inactive**: A compressed message log is created.
[1] Local settings also possible, see Local settings of the compiler (Page 67). Please also refer to the description of the effectiveness of global or local compiler settings (Page 71). [2] User-specific settings. Valid for all SIMOTION projects that the user processes.	

Note

You may have to save and compile the project for the settings to take effect.

3.3.6.2 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 27):
   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**.

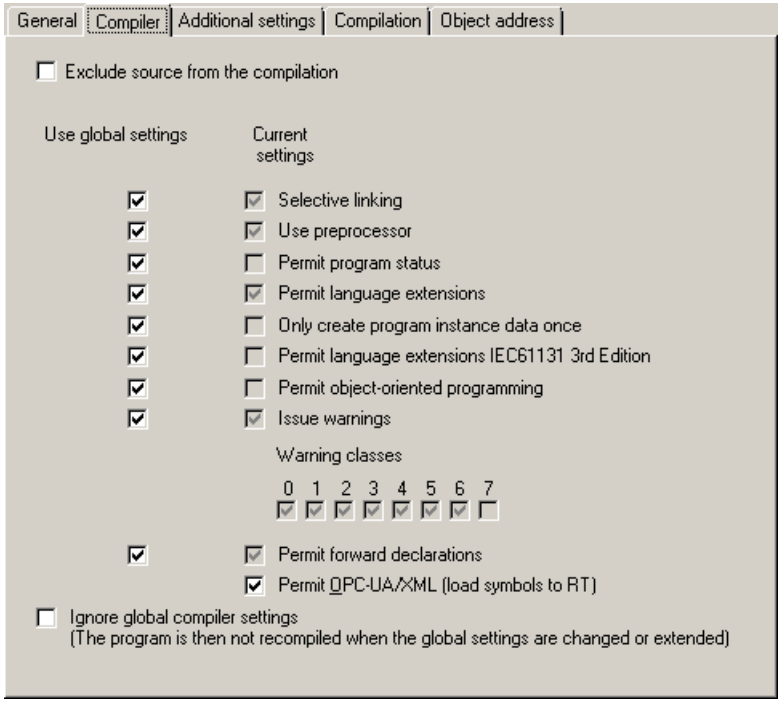


Figure 3-21 Local compiler settings for the ST source file

Parameter

Table 3-8 Parameters for the local compiler settings for the ST source file

Parameter	Description
Exclude source from the compilation	**Active**: This source is not compiled upon compilation of the project, the device or the library (e.g. Menu **Project > Save and recompile all**). The source is marked accordingly in the project navigator. Corresponding information is provided upon compilation. **Inactive** (standard): The source is compiled upon compilation of the project, the device or the library.
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. See the description under "Effectiveness of global or local compiler settings (Page 71)". 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. **Inactive**: Unused code is retained in the executable program. **Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active).
Use preprocessor[1]	**Active**: Preprocessor is used. **Inactive**: Preprocessor is not used. **Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active). See Controlling the preprocessor (Page 369).
Enable program status[1]	**Active**: Additional program code is generated to enable monitoring of program variables (including local variables). **Inactive**: Program status not possible. **Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active). See Properties of the program status (Page 402).
Permit language extensions[1]	**Active**: Language elements are permitted that do not comply with IEC 61131-3. • Direct bit access to variables of a bit data type (Page 148) • Accessing the input parameter of a function block while outside the function block (Page 201) • Calling a program while in a different program (Page 207) **Inactive**: Only language elements that comply with IEC 61131-3 are permitted. **Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active).

Parameter	Description
Permit additional languages, IEC61131 3rd edition[1]	**Active**: Additional language elements can be used in accordance with IEC 61131-3 3rd edition. The corresponding keywords are locked as reserved or protected identifiers. • Nested block comments (Page 112) • CONTINUE statement (Page 171) • Standard-compliant system functions LOWER_BOUND and UPPER_BOUND (Page 192) for determining the limits of a dynamic ARRAY • Additional system functions (see Basic Functions Function Manual): – FROM_BIG_ENDIAN – FROM_LITTLE_ENDIAN – IS_VALID – TO_BIG_ENDIAN – TO_LITTLE_ENDIAN **Inactive**: The additional language elements cannot be used. The corresponding keywords are not locked. **Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active). **Note** When saving the project in the old project format: Projects in which language elements that require this compiler option are used cannot be compiled correctly in SIMOTION SCOUT versions earlier than V4.5.
Permit object-oriented programming[1]	**Active**: Additional language elements for object-oriented programming can be used in accordance with IEC 61131-3 3rd edition. The corresponding keywords are locked as reserved or protected identifiers. **Inactive**: The additional language elements cannot be used. The corresponding keywords are not locked. **Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active). **Note** Independently of the setting the program organization units created in other ST source files using object-oriented programming (e.g. classes, methods) can be used in this source. When saving the project in the old project format: Projects in which language elements that require this compiler option are used cannot be compiled correctly in SIMOTION SCOUT versions earlier than V4.5.
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 207). This setting also improves the download in RUN mode, see corresponding section in the Basic Functions Function Manual. **Inactive**: The local variables of a program are stored according to the task assignment in the user memory of the respective task. **Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active). See Memory areas of the variable types (Page 295). For further information, refer to the SIMOTION Basic Functions Function Manual.

Parameter	Description
Issue warnings 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. "Issue warnings" checkbox: **Active**: The compiler issues the warnings and information according to the warning class selection that follows. **Inactive**: The compiler suppresses all warnings and information concerning this unit. The checkboxes for the warning classes are hidden. **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). "Warning classes" checkboxes (only if "Issue warnings" = active): **Active**: The compiler issues warnings and information for the selected class. **Inactive**: The compiler suppresses warnings and information for the respective class. **Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active). See also Meanings of the warning classes (Page 73).
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. **Active**: Forward declarations are permitted. **Inactive**: Forward declarations are not permitted. **Gray background** (display only): The global setting displayed is adopted (when "Use global settings" = active). See Forward declarations (Page 381).
Permit OPC-UA / -XML (load symbols to RT)	**Active** (standard): Symbol information for the unit variables of the ST source file is available in the SIMOTION device. This is required for: • The *_exportUnitDataSet* and *_importUnitDataSet* functions; see the SIMOTION Basic Functions Function Manual • The watch function of IT DIAG. **Inactive:** Symbol information is not created.

Parameter	Description
Ignore global compiler settings (The program is not compiled again if the global settings are changed or extended.)	**Active:** The global settings of the compiler are ineffective 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 75) with a "High" level: • For all program sources in this library. • For the library itself. **Inactive** (standard): The checkboxes "Use Global Settings" can be selected for all parameters and are presented against 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 ST source file 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. See the description under "Effectiveness of global or local compiler settings (Page 71)".
[1] Global setting also possible, see Global settings of the compiler (Page 63). Please also refer to the description of the effectiveness of global or local compiler settings (Page 71).	

3.3.6.3 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 67).

"Ignore global compiler settings" checkbox

With the "Ignore global compiler settings" checkbox, you specify whether the global properties of the compiler influence the program source.

- **Active:** The global settings of the compiler are ineffective 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 75) with a "High" level:
  - For all program sources in this library.
  - For the library itself.
- **Inactive:** The checkboxes "Use global settings" can be selected for all parameters and are presented against a white background. These checkboxes specify whether the global properties are applied to the corresponding parameter.
  This setting means that in the following case the program source is compiled even though all "Use 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 75) 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:

- "Use global settings" checkbox:
  You use this checkbox to define whether global settings will be used. 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 will be used 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 current (local) settings:
  The function of these checkboxes depends on the "Use global settings" checkbox:
  - If the "Use global settings" checkbox is active:
    The global settings will be used for the corresponding parameter. The checkboxes for the current settings cannot be selected. They are grayed out and show the relevant global setting.
  - If the "Use global settings" checkbox is inactive:
    The checkboxes for the current 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
- Permit additional languages, IEC61131 3rd edition
- Permit object-oriented programming
- Issue 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 73) in the Properties window of the program source.

3.3.6.4 Meanings of the warning classes

The table lists the warning classes and their meanings.

Table 3-9 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

For the detailed description of the compiler error messages, specify which warning classes are assigned to the individual warnings (Page 512) and information (Page 519).

3.3.6.5 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 27) is open.

Procedure

To display the current compiler options using the global or local settings of the compiler (Page 63):

- 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 exclude	Exclude source from the compilation.
-C lang_enable_oop	"Permit object-oriented programming" active.
-C lang_disable_oop	"Permit object-oriented programming" inactive.
-C lang_enable_v3ext	"Permit language extensions IEC61131 3rd Edition" active.
-C lang_disable_V3ext	"Permit language extensions IEC61131 3rd Edition" inactive.
-C lang_ext	"Permit language extensions"[1] active.
-C lang_iec	"Permit language extensions" inactive.
-C opcsym	"Permit OPC-UA/-XML"[1] active.
-C no_opcsym	"Permit OPC-UA/-XML" inactive.
-C opcsym	"Use preprocessor"[1] active.
-C no_preproc	"Use preprocessor" inactive.
-C prog_once	"Only create program instance data once"[1] active.
-C prog_multi	"Only create program instance data once" inactive.
-C scan_twice	"Permit forward declarations"[1] active.
-C scan_once	"Permit forward declarations" inactive.
-D *text*	Preprocessor definition (Page 75).
-e user[2]	Only global settings act.
-e local	"Do not recompile the source if global compiler settings have been changed"[1] active. Only local settings act. 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	"Selective linking"[1] active.
-l no_sel	"Selective linking" inactive.
-s	"Enable program status"[1] active.
-s_off	"Enable program status" inactive.
-w no_warn	"Suppress warnings"[1] active.
-w all_warn	Display all warnings.
-w *n*_off	Warning class *n* inactive[1].
-w *n*_on	Warning class *n* active[1].
Further options	Internal options of the SIMOTION compiler.
[1] Meaning of the compiler option: See "Local compiler settings" (Page 67).	
[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.

3.3.7 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 346)

3.3.8 Making preprocessor definitions

You can make definitions for the preprocessor (see Controlling the preprocessor (Page 369)) 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 75)).

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 27)):
   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**.

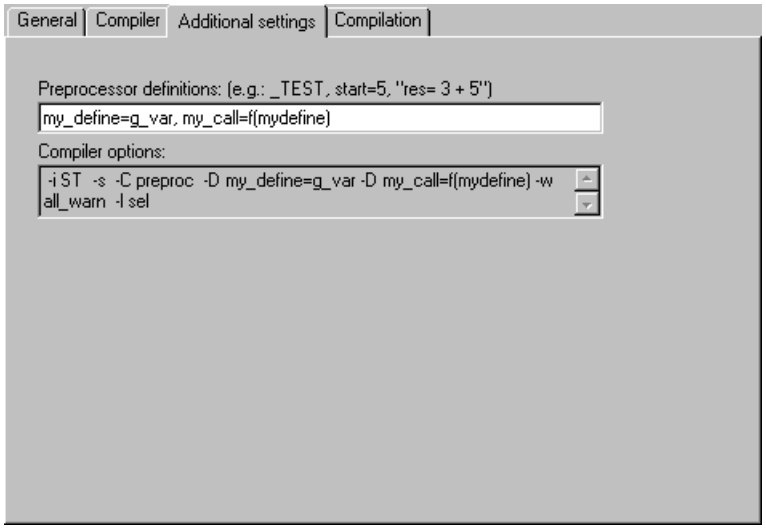


Figure 3-22 Preprocessor definitions

Table 3-10 Syntax of the preprocessor definitions

Syntax	Meaning
Identifier=text	The specified *identifier* is defined and replaced in the ST source file by the specified *text*. Permissible characters: See table footnote. If the expression contains blanks (e.g. in the text), the syntax *"Identifier=text"* must be used.
'Identifier=text'	
"Identifier=text"	
Identifier	The specified *identifier* is defined and replaced in the ST source file by blank text. Permissible characters: See table footnote.
Multiple preprocessor definitions are separated by commas: *Definition_1, Definition_2, ...* Permissible characters: • For *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. • 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 370).

3.3.9 Exporting, importing and printing an ST source file

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

3.3.9.1 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 27), entering the password if necessary (for ST source files with know-how protection (Page 75)).
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 75) 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 36) and bookmarks (Page 47) are not exported.

An ST source file with know-how protection is exported without protection.

3.3.9.2 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 36) and bookmarks (Page 47) are not exported.

3.3.9.3 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 25)).

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

3.3.9.4 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 25)).
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.

3.3.9.5 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.

3.3.10 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.

Program status (Page 404) 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 **Options > 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.

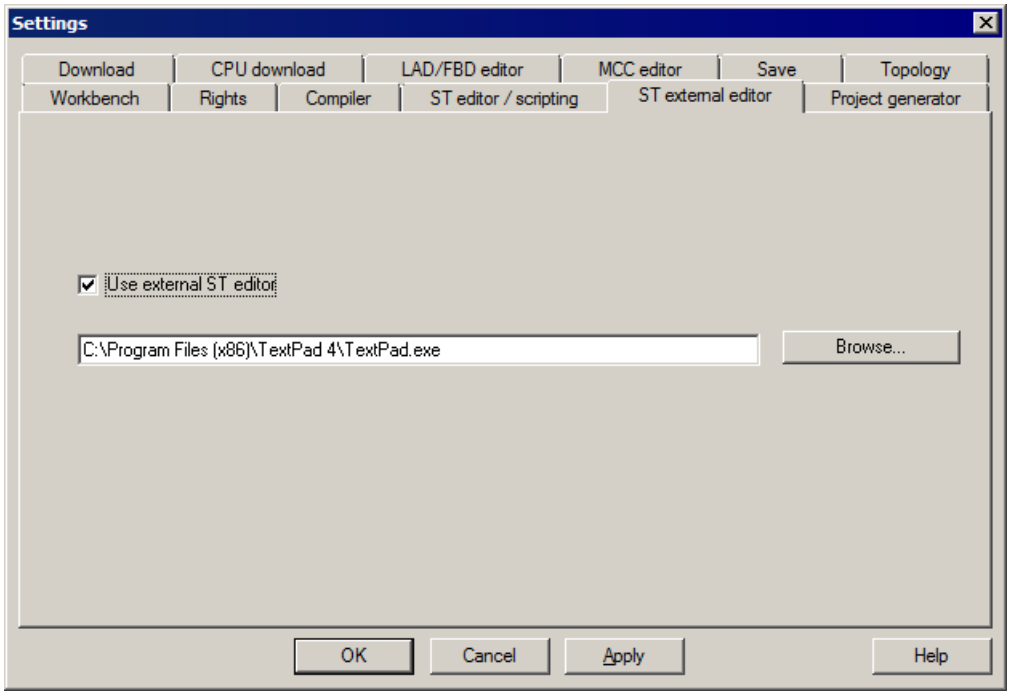


Figure 3-23 Settings for the use of an external editor

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:

NOTICE
Data loss possible If you close a project or exit SIMOTION SCOUT before all windows of the external editor have been closed, data can be lost! Close all windows of the external editor before you close a project or exit SIMOTION SCOUT.

3.3.11 ST source file menus

3.3.11.1 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 3-11 ST Source File Menu

Function	Meaning/information
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 and whether the changes should be saved in the snippet editor.
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. See: Changing the properties of an ST source file (Page 27).
Accept and compile	Choose this command to transfer the current ST source file to the project and compile into executable code. See: Starting the compiler (Page 62).
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). See: Exporting an ST source file as a text file (ASCII) (Page 77).
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. See: Splitting the editor window (Page 40).
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. The following prerequisites are necessary: 1. The program must be compiled with the appropriate compiler option. 2. The project and the program must be loaded into the target system. 3. An online connection to the target system must have been established. Select the command again to close the **program status**. See: Using the program status (Page 404).
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.

3.3.11.2 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 3-12 ST source file context menu

Function		Meaning/information
Open		Select this command to open the selected ST source file. See: Opening an existing ST source file (Page 27).
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 97).
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. See: Importing XML data into an ST source file (Page 78).
	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. See: Exporting an ST source file in XML format (Page 77).
Accept and compile		Choose this command to transfer the current ST source file to the project and compile into executable code. See: Starting the compiler (Page 62).
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.

Function	Meaning/information
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. The following requirements are necessary: 1. The program must be compiled with the appropriate compiler option. 2. The project and the program must be loaded into the target system. 3. An online connection to the target system must have been established. Select the command again to close the **program status**. See: Using the program status (Page 404).
Export	Select this command to export the selected ST source file as a text file (ASCII). See: Exporting an ST source file as a text file (ASCII) (Page 77).
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. See: Know-how protection for ST source files (Page 75).
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. See: Know-how protection for ST source files (Page 75).
Display reference data	An error-free compilation is required for a correct, consistent display of the reference data. If required, compile the project, the CPU, the program or the library first. See: Reference data (Page 361), creating a cross-reference list (Page 361).
Cross references	The cross-reference list of the selected ST source file is generated and displayed. The cross-reference list contains the declaration and uses all the identifiers for the selected ST source file. See: Content of the cross-reference list (Page 362).
Program structure	The program structure of the selected ST source file is generated and displayed. The program structure contains all the subprogram calls and their nesting within the selected ST source file. See: Content of the program structure (Page 365).
Code attributes	The code attributes of the selected ST source file are generated and displayed. The code attributes contain information about the storage requirements of various data areas of the selected ST source file. See: Code attribute contents (Page 367).
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. See: Changing the properties of an ST source file (Page 27).

3.4 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 387).

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.

3.4.1 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.

3.4.2 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.

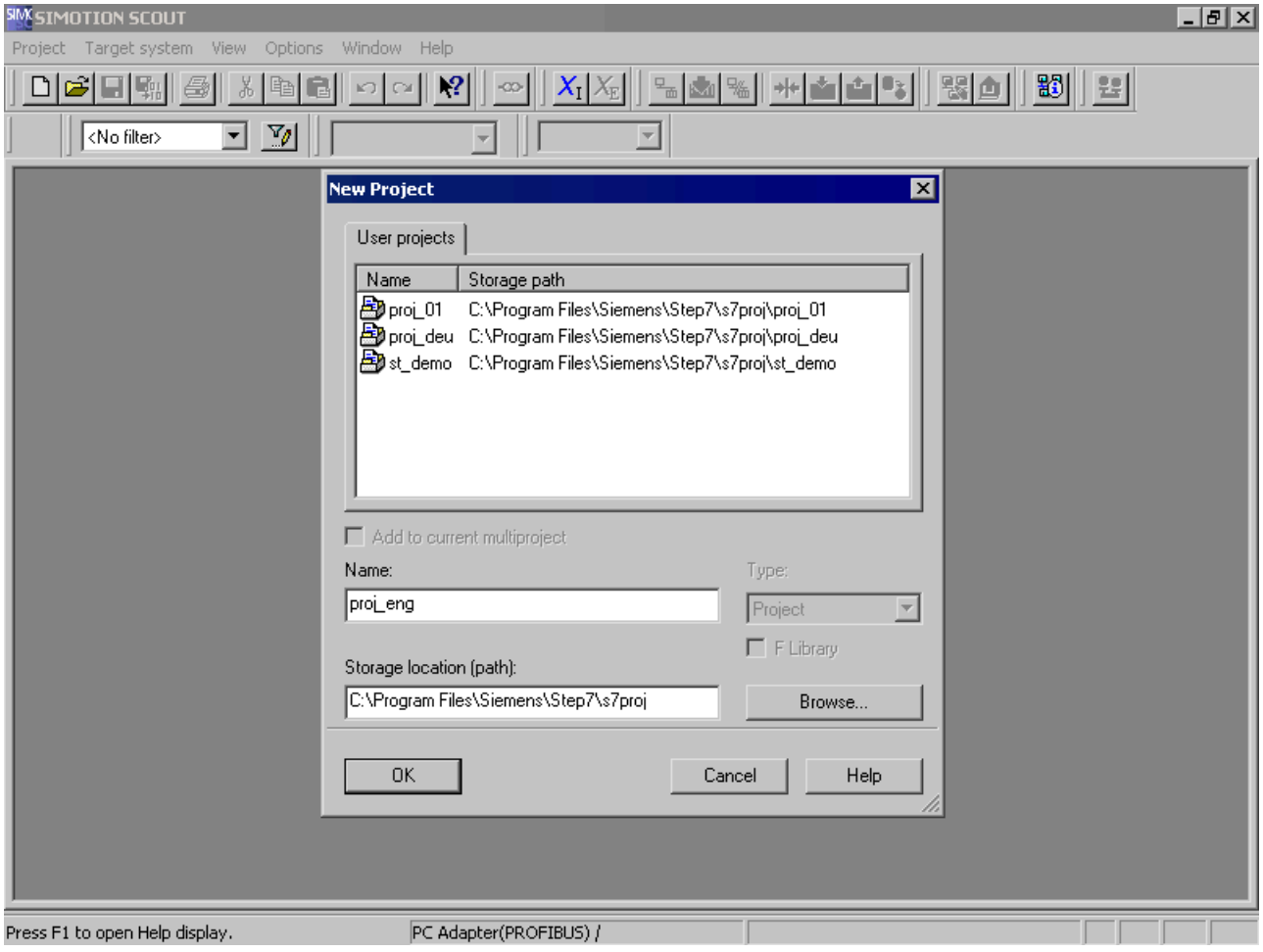


Figure 3-24 Creating a new project

3.4.3 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.

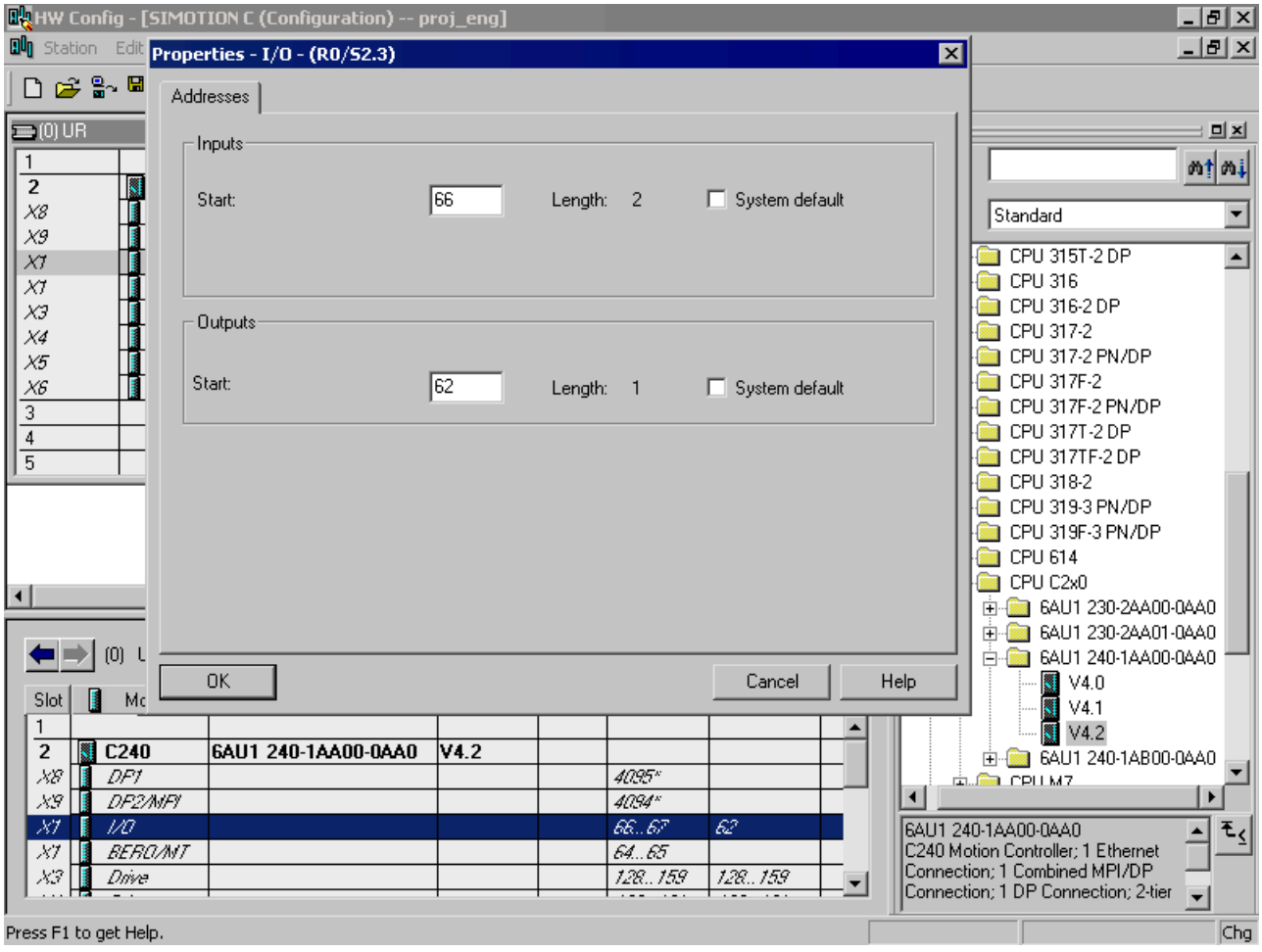


Figure 3-25 Change in HW Config

3.4.4 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 88), 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 29); the structure of an ST source file is described in detail in Structure of the ST source file (Page 110) and in Source file sections (Page 257).

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**.

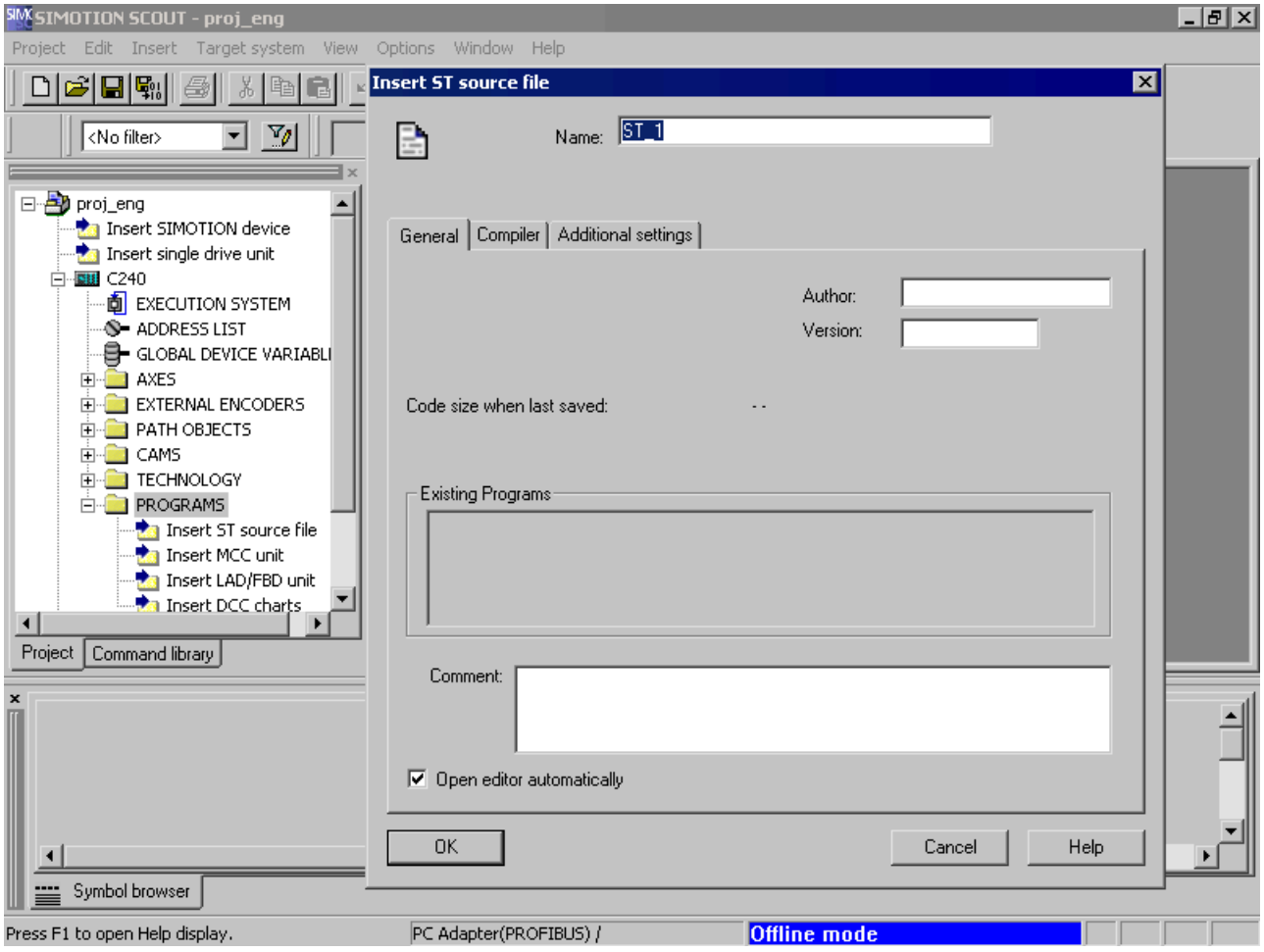


Figure 3-26 Naming the ST source file

3.4.4.1 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 29) and in Making settings for the compiler (Page 63).

3.4.4.2 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 3-13 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
```

3.4.5 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.

3.4.5.1 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.

3.4.5.2 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 89).
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.

3.4.5.3 Example of error messages

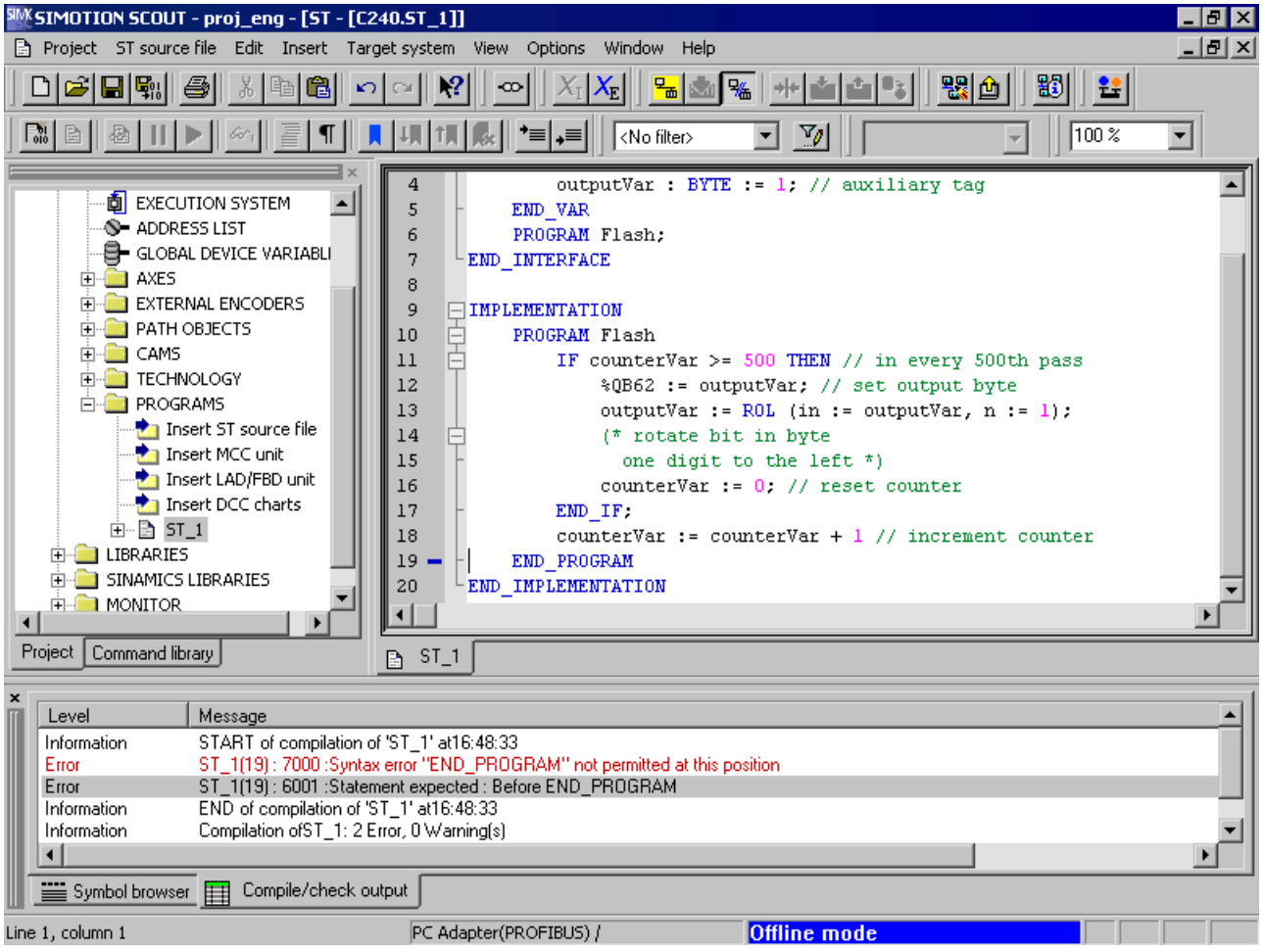


Figure 3-27 Error messages during ST source file compilation

The figure shows an example of compiling the ST source file ST_1 (see Source text of the sample program (Page 88), 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 493).

3.4.6 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.

3.4.6.1 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.

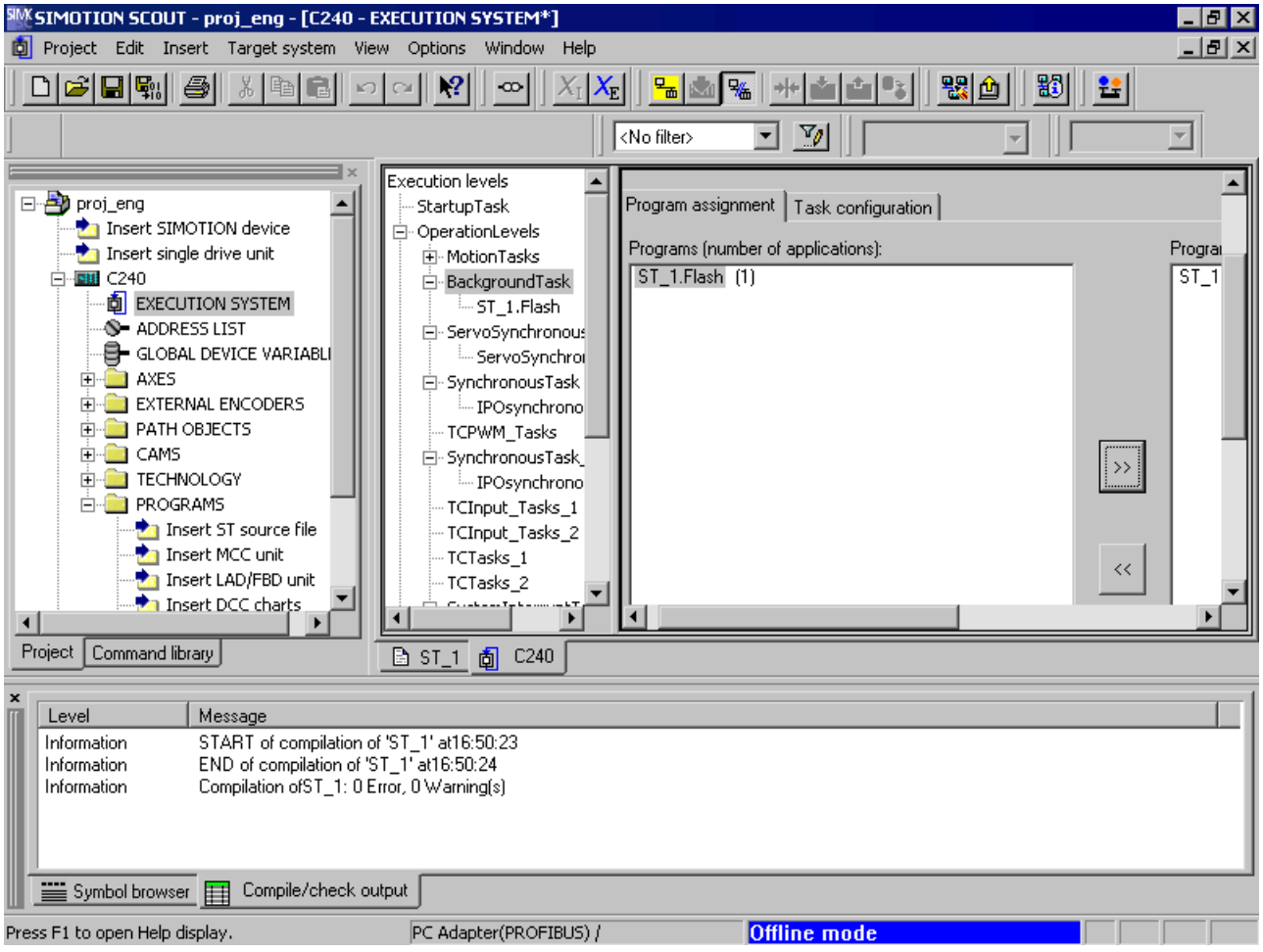


Figure 3-28 Assigning the sample program to the BackgroundTask

3.4.6.2 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.

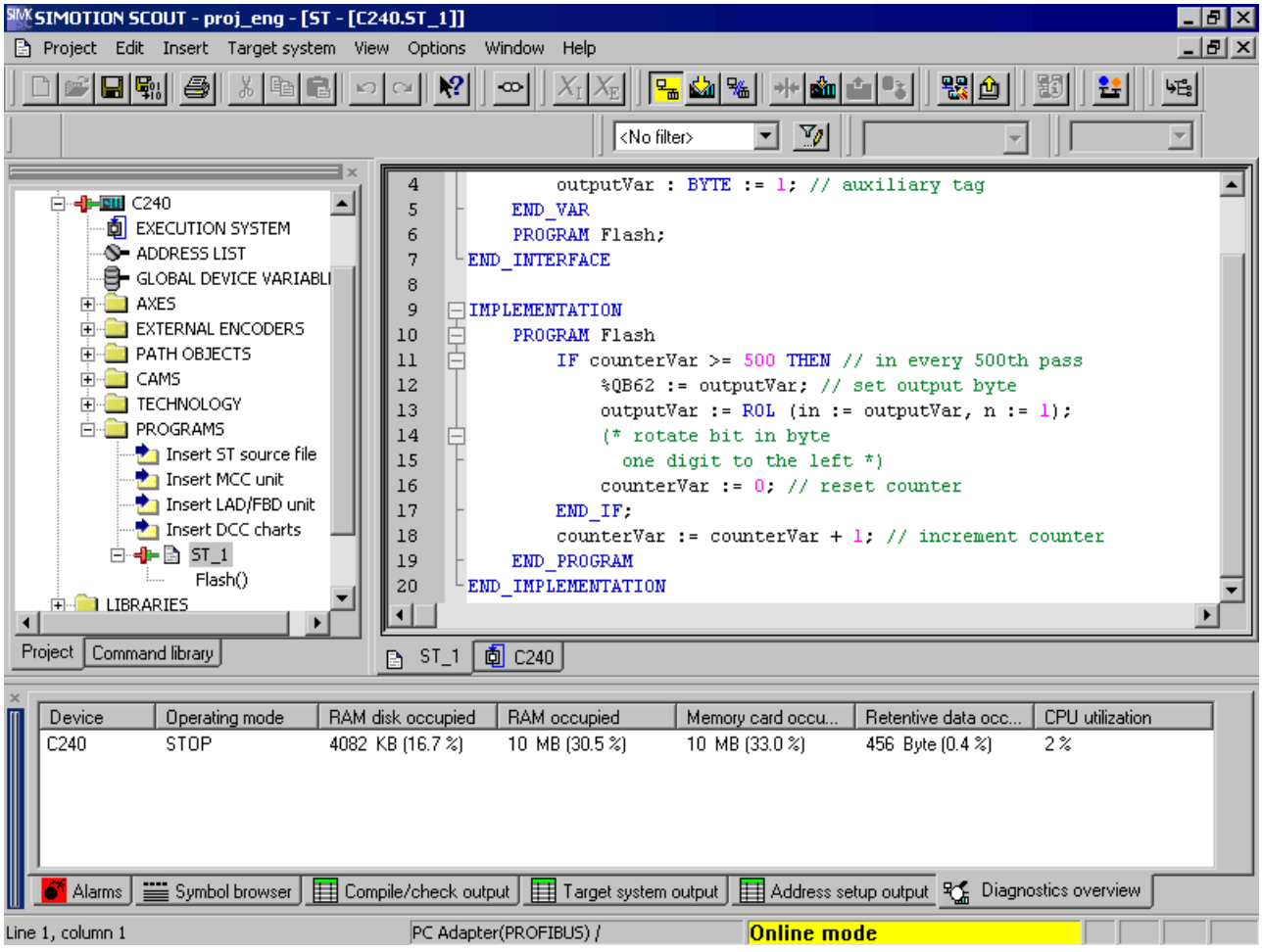


Figure 3-29 Establishing a connection to the target system

3.4.6.3 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.

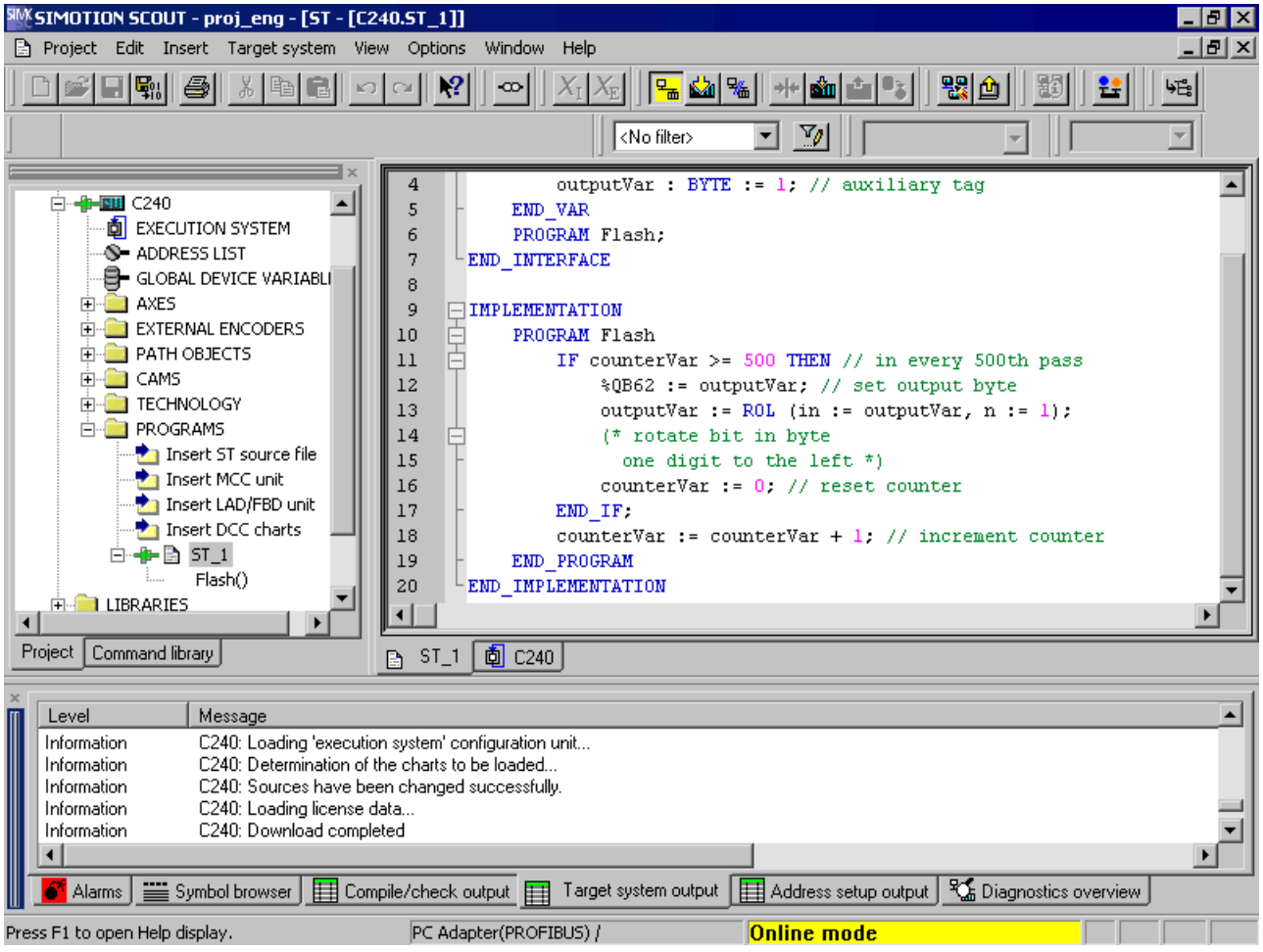


Figure 3-30 Downloading the sample program to the target system

3.4.6.4 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 387).

ST Fundamentals

4

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 441).

4.1 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.

4.1.1 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.

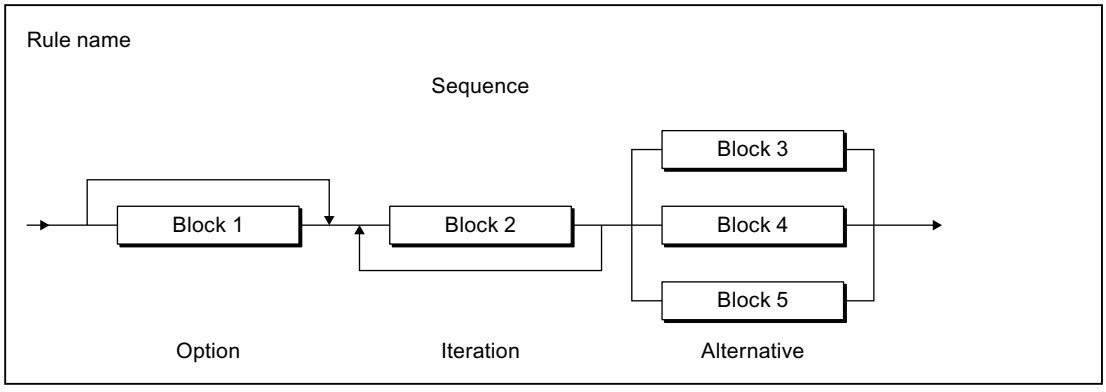


Figure 4-1 Syntax diagram

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

4.1.2 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:

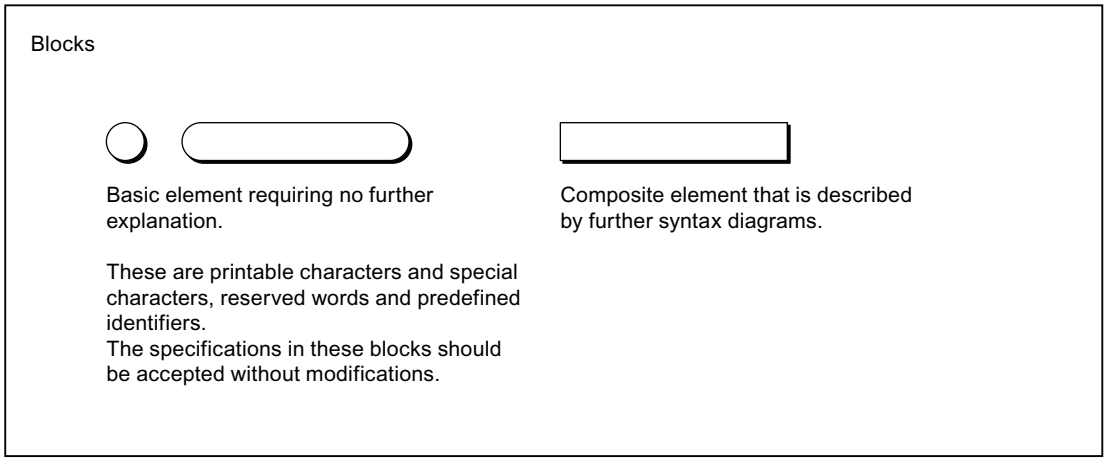


Figure 4-2 Blocks

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 427).

4.1.3 Meaning of the rules (semantics)

Only the formal structure of the language can be represented in the rules. 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 include:

- 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 442). 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 442).

4.2 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 96). Below is a

selection of the main terminals; for a complete overview, refer to Basic elements (terminals) (Page 429).

4.2.1 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 97)) 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 Basic elements (terminals) (Page 429).

4.2.2 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.

4.2.2.1 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:

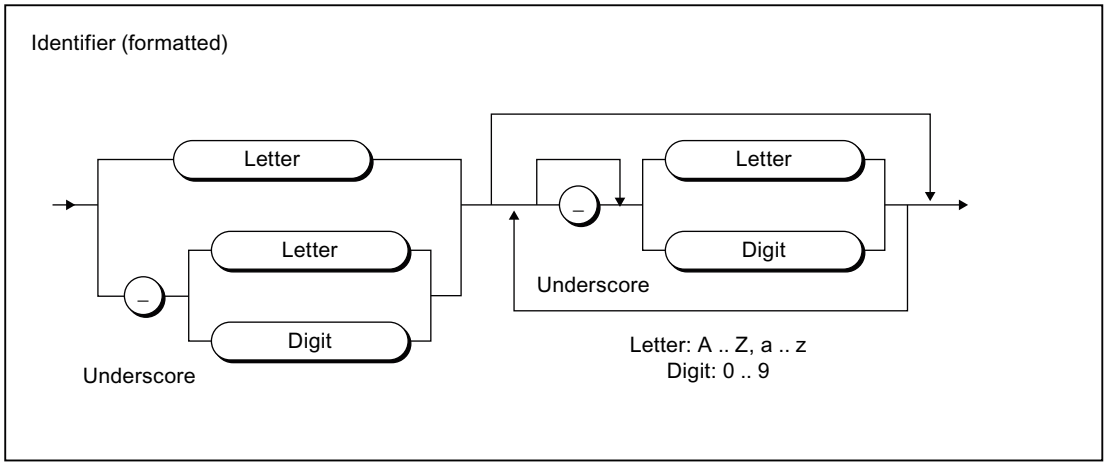


Figure 4-3 Syntax: Identifier

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

4.2.2.2 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.

Invalid identifier	Reason
`Array`	ARRAY, REAL, and _sizeOf are valid identifiers from a technical point of view, but they are also reserved identifiers (Page 99), 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`	

4.2.3 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 99)
- Reserved identifiers in the ST programming language (Page 105)
- Other reserved identifiers (Page 106), 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.

4.2.3.1 Protected identifiers in the ST programming language

The protected identifiers of the ST language are listed in the table. They can only be used as predefined. A brief explanation can be found in the Appendix, under Reserved words (Page 432). The associated syntax diagrams (Page 95) can be found in the Appendix, under Rules (Page 441).

Table 4-1 Protected identifiers in ST programming language

A	
ABS	ANYTYPE_TO_LITTLEBYTEARRAY
ABSTRACT [2]	ARRAY
ACOS	AS
AND	ASIN
ANYOBJECT	AT
ANYOBJECT_TO_OBJECT	ATAN
ANYTYPE_TO_BIGBYTEARRAY	
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_DINT
CLASS [2]	CTD_UDINT
CONCAT	CTU
CONCAT_DATE_TOD	CTU_DINT
CONSTANT	CTU_UDINT
CONTINUE [1]	CTUD
COS	CTUD_DINT
CTD	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_PROGRAM
ELSIF	END_REPEAT
END_CASE	END_STRUCT
END_CLASS [2]	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_METHOD [2]	EXPT
END_NAMESPACE [2]	EXTENDS [2]
F	
F_TRIG	FROM_BIG_ENDIAN [1]
FALSE	FROM_LITTLE_ENDIAN [1]
FINAL [2]	FUNCTION
FIND	FUNCTION_BLOCK
FOR	
G	
GOTO	
I	
IF	INT_TO_SINT
IMPLEMENTATION	INT_TO_TIME
IMPLEMENTS [2]	INT_TO_UDINT
INSERT	INT_TO_UINT
INT	INT_TO_USINT
INT_TO_BYTE	INT_TO_WORD
INT_TO_DINT	INT_VALUE_TO_BOOL
INT_TO_DWORD	INTERFACE
INT_TO_LREAL	INTERNAL [2]
INT_TO_REAL	IS_VALID [1]
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
LOWER_BOUND [1]	LREAL_VALUE_TO_BYTE
LREAL	LREAL_VALUE_TO_DWORD
LREAL_TO_DINT	LREAL_VALUE_TO_WORD
LREAL_TO_INT	
M	
MAX	MIN
METHOD [2]	MOD
MID	MUX
N	
NAMESPACE [2]	NULL [2]
NOT	
O	
OF	OVERLAP
OR	OVERRIDE [2]
P	
PRIVATE [2]	PROTECTED [2]
PROGRAM	PUBLIC [2]
R	
R_TRIG	REAL_VALUE_TO_DWORD
REAL	REAL_VALUE_TO_WORD
REAL_TO_DINT	REF [2]
REAL_TO_DWORD	REF_TO [2]
REAL_TO_INT	REPEAT
REAL_TO_LREAL	REPLACE
REAL_TO_SINT	RETAIN
REAL_TO_STRING	RETURN
REAL_TO_TIME	RIGHT
REAL_TO_UDINT	ROL
REAL_TO_UINT	ROR
REAL_TO_USINT	RS
REAL_VALUE_TO_BOOL	RTC
REAL_VALUE_TO_BYTE	
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	SUPER [2]
T	
TAN	TOF
THEN	TON
THIS [2]	TO_BIG_ENDIAN [1]
TIME	TO_LITTLE_ENDIAN [1]
TIME_OF_DAY	TP
TIME_TO_INT	TRUE
TIME_TO_REAL	TRUNC
TO	TYPE
TOD	
U	

UDINT	UINT_TO_USINT
UDINT_TO_BYTE	UINT_TO_WORD
UDINT_TO_DINT	UINT_VALUE_TO_BOOL
UDINT_TO_DWORD	UNIT
UDINT_TO_INT	UNTIL
UDINT_TO_LREAL	UPPER_BOUND [1]
UDINT_TO_REAL	USELIB
UDINT_TO_SINT	USEPACKAGE
UDINT_TO_STRING	USES
UDINT_TO_UINT	USING [2]
UDINT_TO_USINT	USINT
UDINT_TO_WORD	USINT_TO_BYTE
UDINT_VALUE_TO_BOOL	USINT_TO_DINT
UINT	USINT_TO_DWORD
UINT_TO_BYTE	USINT_TO_INT
UINT_TO_DINT	USINT_TO_LREAL
UINT_TO_DWORD	USINT_TO_REAL
UINT_TO_INT	USINT_TO_SINT
UINT_TO_LREAL	USINT_TO_UDINT
UINT_TO_REAL	USINT_TO_UINT
UINT_TO_SINT	USINT_TO_WORD
UINT_TO_UDINT	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	

1 Only with active compiler option "Permit language extensions, IEC61131 3rd edition".

2 Only with active compiler option "Permit object-oriented programming".

4.2.3.2 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 4-2 Reserved identifiers in the ST programming language

A	
ACTION ADD ADD_DT_TIME	ADD_TIME ADD_TOD_TIME
B	
BCD_TO_BYTE BCD_TO_DINT BCD_TO_DWORD BCD_TO_INT	BCD_TO_LWORD BCD_TO_SINT BCD_TO_WORD BYTE_TO_BCD
C	
CONFIGURATION CTD_LINT CTD_ULINT CTU_LINT	CTU_ULINT CTUD_LINT CTUD_ULINT
D	
DINT_TO_BCD DIV	DIVTIME DWORD_TO_BCD
E	
EN END_ACTION END_CONFIGURATION END_RESOURCE	END_STEP END_TRANSITION ENO EQ
F	
F_EDGE	FROM
G	
GE	GT
I	
INITIAL_STEP	INT_TO_BCD
L	
LE LINT LT	LWORD 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	

4.2.3.3 Additional reserved identifiers

User definitions must avoid not only protected identifiers (Page 99) and reserved identifiers (Page 105) 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.

4.2.4 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.

4.2.4.1 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
```

4.2.4.2 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.

4.2.4.3 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.

4.2.4.4 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;         // corresponds to a := TRUE
b := FALSE;     // corresponds to b := 0
```

4.2.4.5 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.

4.2.5 Character strings

What is a character string?

A character string is a sequence of 0 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 $27), as these have a special function within the string
- As the two-digit hexadecimal ASCII code of the relevant character preceded by the dollar sign ($)
- As a combination of two characters according to the following table:

Table 4-3 Two-character combinations for special characters in strings

Character combination			Meaning
$$			Dollar sign $ ($24)
$'			Apostrophe ' ($27)

Character combination			Meaning
$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.
'RL' '$0D$0A'	Two equivalent representations for a string of length 2 containing the characters CR and LF.
'$$1.00'	String of length 5 containing $1.00.
'Text$R$L'	String of length 6 containing the word Text followed by the characters CR and LF.
'ÄÖÜ' '$C4$D6$FC'	Two equivalent representations for a string of length 3 containing the German umlauts ÄÖü (A, O, u with diaresis).

4.3 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 257).

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:

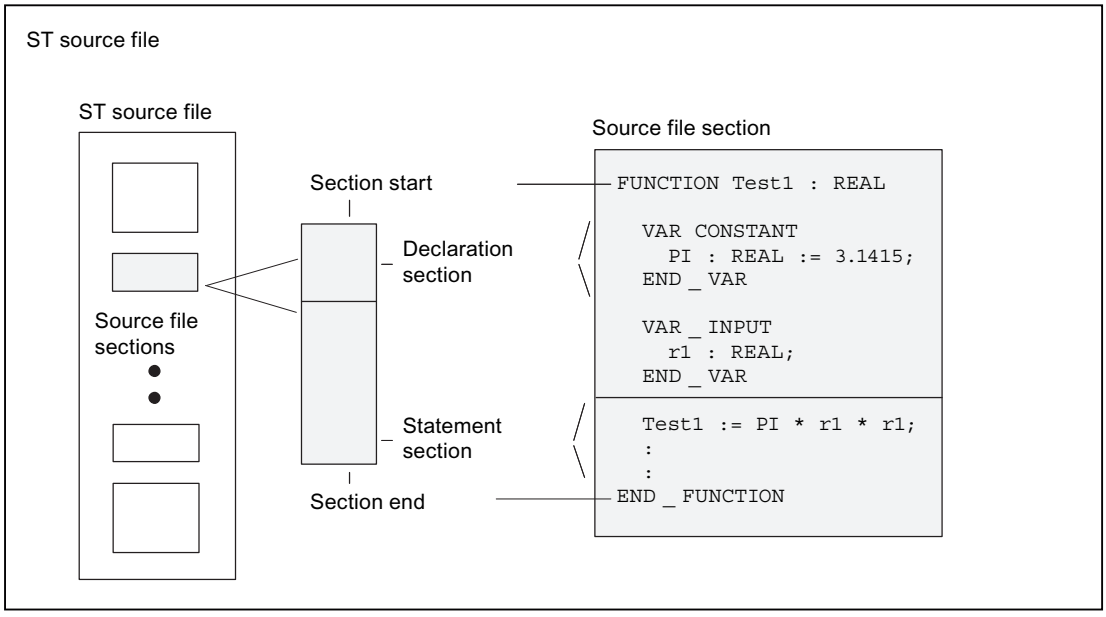


Figure 4-4 Structure of an ST source file

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 4-4 Major sections of an ST source file

Source file section	Description
Unit statement (optional)	Contains the name of the ST
Interface section	Contains instructions for making variables, data types and program organization units (POU) public, and for using other program sources.
Implementation section	Contains instructions for using other program sources and the 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 data 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.

The section "Template for example unit" (Page 521) contains a copy of this template.

4.3.1 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 136)
- Control statements
  Repetition or branching of statements; see Control statements (Page 162)
- Subroutine execution
  Functions (FC) and function blocks (FB); see Functions, function blocks, programs (Page 181)

Table 4-5 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);
...
```

4.3.2 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 *).

You can also use the corresponding buttons in the ST editor toolbar to show or hide comments. This is helpful for showing or hiding code positions.

Please note the following when inserting comments:

- The character pairs (* and *) are ignored within the line comment.
- Nesting of block comments is not allowed as standard. However, you can nest line comments in block comments.
  With activated compiler option “Permit language extensions IEC61131 3rd edition” only: block comments can be nested.
- You can use the complete extended ASCII character set in 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 427).

Table 4-6 Examples of comments

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

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

(* The example above is easier to maintain as a block comment.
*)
c := 87;

(* Block comments can only be nested with compiler option "Admit language
extensions IEC61131 3rd edition".
(* Nested block comment *)
*)
```

4.4 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 (Page 114)
- User-defined data types (UDT) (Page 117)
  - Simple derivatives
  - Arrays
  - Enumerators
  - Structures (Struct)
- Technology object data types (Page 132)
- System data types (Page 135)

4.4.1 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 4-7 Bit widths and value ranges of the elementary data types

Type		Reserv. word	Bit width	Range of values
Bit data type Data of this type 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 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_967_295 (0 to 2**32-1)
	Floating-point number (per IEEE -754)	REAL	32	-3.402_823_466E+38 to -1.175_494_351E-38, 0.0, +1.175_494_351E-38 to +3.402_823_466E+38 Accuracy: 23-bit mantissa (corresponds to 6 decimal places), 8-bit exponent, 1-bit sign.
	Long floating-point number (in accordance with IEEE-754)	LREAL	64	-1.797_693_134_862_315_7E+308 to -2.225_073_858_507_201_4E-308, 0.0, +2.225_073_858_507_201_4E-308 to +1.797_693_134_862_315_7E+308 Accuracy: 52-bit mantissa (corresponds to 15 decimal places), 11-bit exponent, 1-bit sign.
Time types These data types are used to represent various date and time values.				

Type		Reserv. word	Bit width	Range of values
	Duration in increments of 1 ms	TIME	32	T#0d_0h_0m_0s_0ms to T#49d_17h_2m_47s_295ms Maximum of 2 digits for the values day, hour, minute, second and a maximum of 3 digits for milliseconds Initialization with T#0d_0h_0m_0s_0ms
	Date in increments of 1 day	DATE	32	D#1992-01-01 to D#2200-12-31 Leap years are taken into account, year has four digits, month and day are two digits each Initialization with D#0001-01-01
	Time of day in increments of 1 ms	TIME_OF_DAY (TOD)	32	TOD#0:0:0.0 to TOD#23:59:59.999 Maximum of two digits for the values hour, minute, second and maximum of three digits for milliseconds Initialization with TOD#0:0:0.0
	Date and time	DATE_AND_TIME (DT)	64	DT#1992-01-01-0:0:0.0 to DT#2200-12-31-23:59:59.999 DATE_AND_TIME consists of the data types DATE and TIME Initialization with DT#0001-01-01-0:0:0.0
String type Data of this type represents character strings, in which each character is encoded with the specified number of bytes. The length of the string can be defined at the declaration. Indicate the length in "[" and "]", e.g. STRING[100]. The default setting consists of 80 characters. The number of assigned (initialized) characters can be less than the declared length.				
	String with 1 byte/character	STRING	8	You may use all the characters in the extended ASCII character set (ASCII code $00 to $FF). Default ' ' (empty string)

Note

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

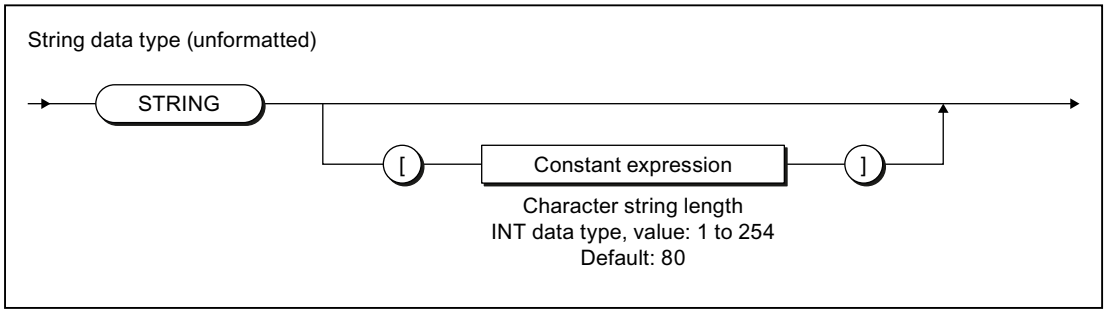


Figure 4-5 Syntax: STRING data type

4.4.1.1 Value range limits of elementary data types

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

Table 4-8 Symbolic constants for the value range limits of elementary data types

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

4.4.1.2 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 4-9 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).

4.4.1.3 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 4-10 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. Please refer to the *SIMOTION Basic Functions* Function Manual. Initialization with STRUCTALARMID#NIL
StructTaskId	32	Data type of the taskId for the project-wide unique identification of the tasks in the execution system. Please refer to the *SIMOTION Basic Functions* Function Manual. Initialization with STRUCTTASKID#NIL

Table 4-11 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

4.4.2 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 110) and Source file modules (Page 257)):

- 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 118)

Derivation of elementary or derived data types (Page 119)

Derived data type ARRAY (Page 120)

Derived data type - Enumerator (Page 123)

Derived data type STRUCT (structure) (Page 124)

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

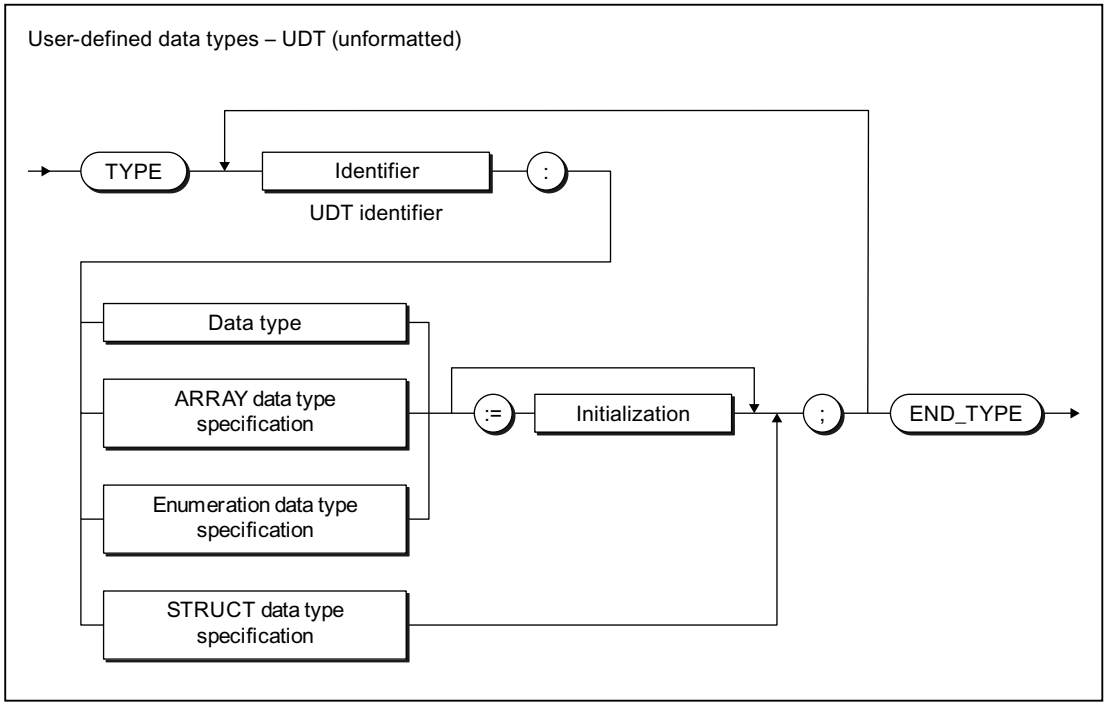


Figure 4-6 Syntax: User-defined data type

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 119):
   - 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 - array (Page 120)
   - Enumerator data type specification, see Derived data type enumeration – enumerator (Page 123)
   - STRUCT data type specification, see Derived data type STRUCT – structure (Page 124)

   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 139).

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 137), and how to assign values with UDT in Syntax for value assignment (Page 145).

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

4.4.2.2 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 4-12 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;
```

4.4.2.3 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

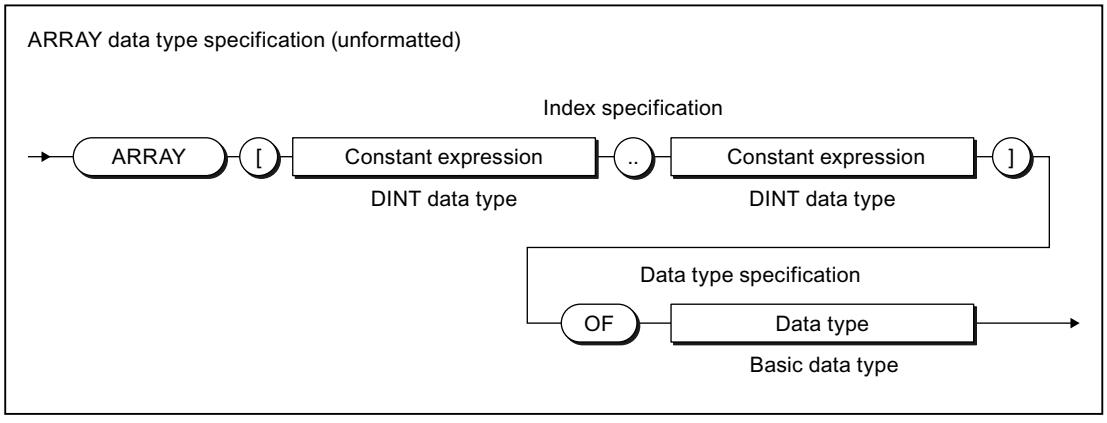


Figure 4-7 Syntax: ARRAY data type specification

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 176)).
- 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 176)).

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** (basic data type). 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.

The _arrayCopy function can be used to copy an area of an array into another array with the same basic data 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 192) for the FB and FC declaration subsection (Page 186).

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 4-13 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 4-14 Examples of multi-dimensional arrays

```
TYPE
    a      : ARRAY[1..3] OF INT;     // Single-dimensional array
                                     // (3 columns)
    matrix1: ARRAY[1..4] OF a;       // Two-dimensional array
                                     // (4 lines with 3 columns)
    b      : ARRAY[4..8] OF INT;     // single-dimensional field
                                     // (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-dimensional array
    n : matrix2;     // Variable n of data type
                     // Two-dimensional array
END_VAR

m[4][3] := 9;       // Write in matrix1 in line 4, column 3
n[16][8] := 10;     // Write in matrix2 in 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 matrix1[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 139).

4.4.2.4 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

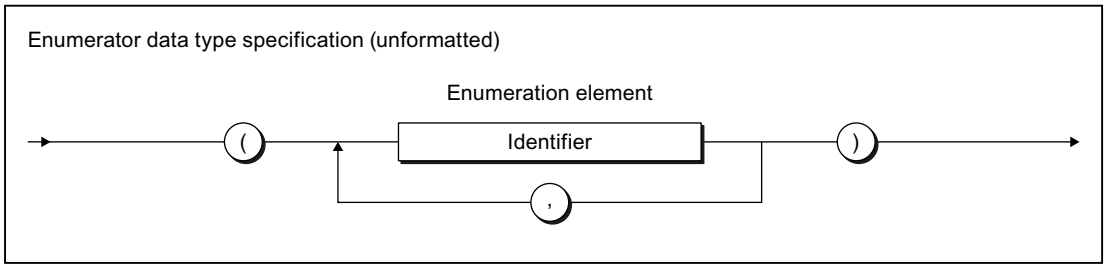


Figure 4-8 Syntax: Enumeration data type specification

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

Example

Table 4-15 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 139).

4.4.2.5 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

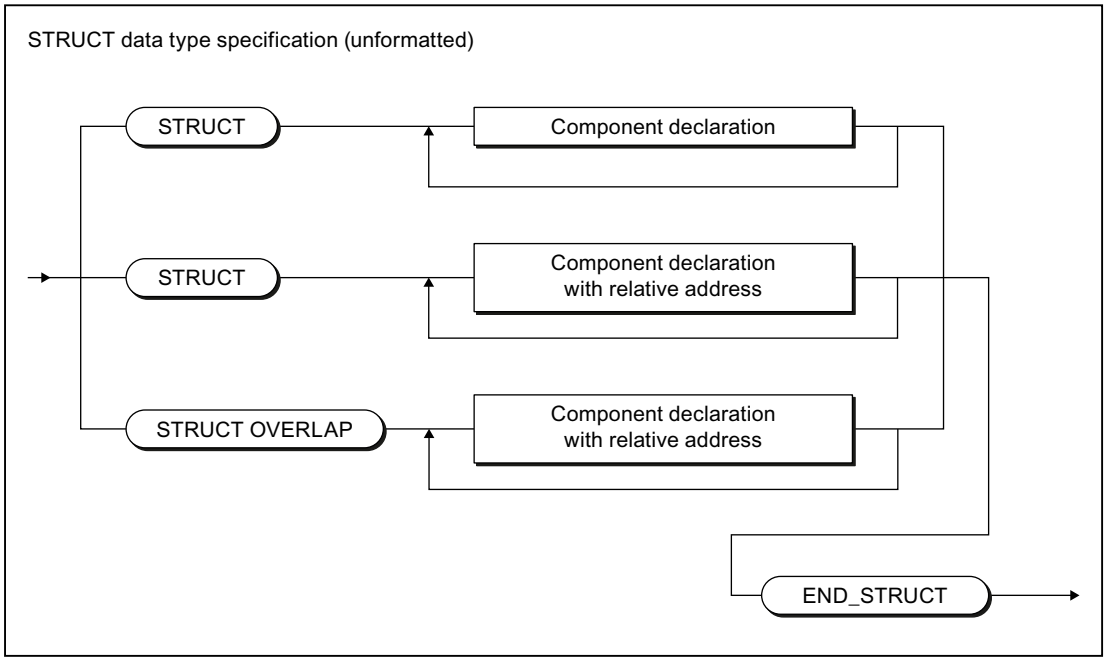


Figure 4-9 Syntax: STRUCT data type specification

Component declaration

Usually, a structure contains the definitions of the individual components between the keywords STRUCT and END_STRUCT.

The syntax of the component declaration is shown in the following figure. The components are arranged in the order of their declaration with this syntax. The relative addresses of the components within the structure are assigned automatically by the compiler.

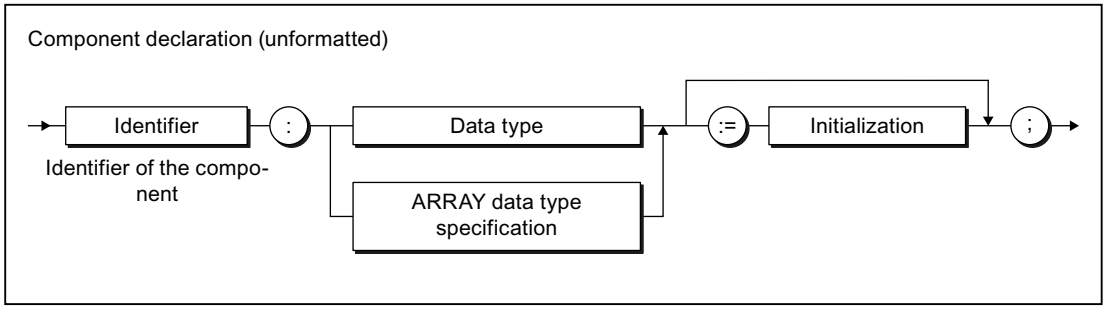


Figure 4-10 Syntax: Component declaration

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 type 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 4-16 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 139).

Component declaration with specification of the relative addresses

You can assign the relative addresses of the components within the structure with the keyword AT in the component declarations, see following syntax diagram.

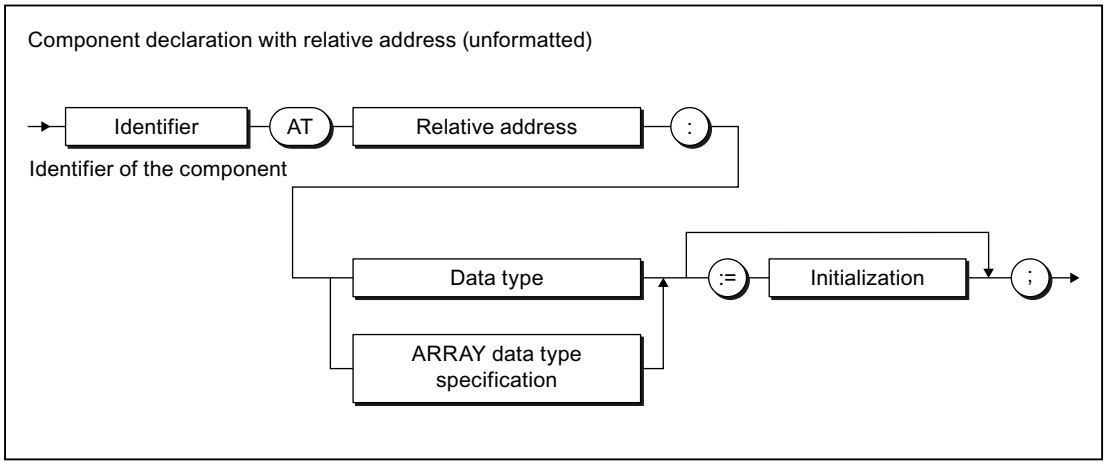


Figure 4-11 Syntax: Component declaration with relative address

Specify the relative address of the component (byte offset) as follows:

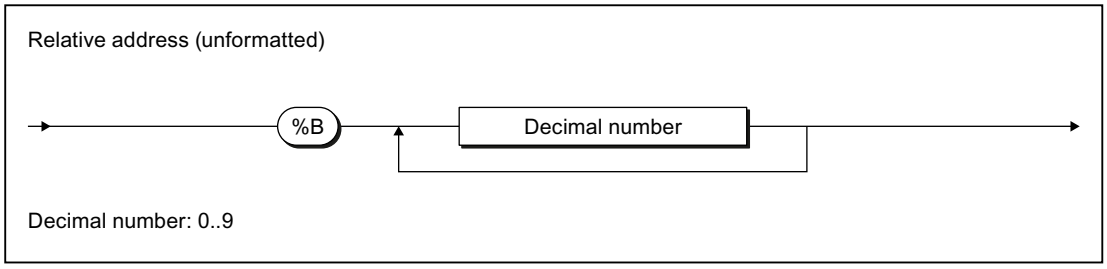


Figure 4-12 Syntax: Relative address

If you specify relative addresses of the structure components, you must do this for all components of this structure.

The permissible relative addresses of the components depends on their data type, see following table:

Table 4-17 Permissible addresses and length of various data types

Data type	Permissible address	Length
Elementary data types		
BOOL, BYTE, SINT, USINT	Any	1 byte
WORD, INT, UINT	Can be divided by 2	2 bytes
DWORD, DINT, UDINT, REAL TIME, DATE, TIME_OF_DAY (TOD)	Can be divided by 4	4 bytes
LREAL DATE_AND_TIME (DT)	Can be divided by 8	8 bytes
STRING	Any address	Declared length + 2 bytes (Default: 82 bytes)
User-defined data types (UDT)		
Derived data type	Corresponding to the basic data type	
ARRAY	Corresponding to the basic data type	Corresponding to the index specification and the basic data type
Enumerator	Can be divided by 4	4 bytes
STRUCT (structure)	Corresponding to the largest data type within the structure	Corresponding to the declaration of the structure
Further data types		
StructAlarmId, StructTaskId	Can be divided by 4	4 bytes
System data type	Corresponding to its declaration as STRUCT or enumerator	
TO data type, ANYOBJECT	Can be divided by 8	8 bytes

If the above addressing rules are violated, the compiler issues an error message which indicates the required divisor.

Table 4-18 Example of a structure with specification of the relative address

```
myLocatedStruct : STRUCT
    memb1 AT %B0 : INT;     // Relative address 0
                            // Bytes 0 and 1 assigned
    memb2 AT %B8 : REAL;    // Relative address 8
                            // Bytes 8 .. 11 assigned
                            // Bytes 2 .. 7 not assigned (gap)
END_STRUCT
```

The declaration order of the components is arbitrary when specifying the relative addresses. The structure in the following example is identical to that in the previous example.

Table 4-19 Identical structure to the previous example

```
myLocatedStruct : STRUCT
    memb2 AT %B8 : REAL;    // Relative address 8
                            // Bytes 8 .. 11 assigned
    memb1 AT %B0 : INT;     // Relative address 0
                            // Bytes 0 and 1 assigned
                            // Bytes 2 .. 7 not assigned (gap)
END_STRUCT
```

The address ranges of the components must not overlap except when the structure is declared with the keyword STRUCT OVERLAP, see following section.

Structure with overlapping address ranges (UNION)

If you make the component declarations with specification of the relative address (see above) between the keywords STRUCT OVERLAP and END_STRUCT, the address ranges can overlap.

The syntax diagram for the component declaration with relative address (see above) applies for the component declaration. See Table "Permissible addresses and length of various data types" for the permissible addresses.

Table 4-20 Example of a structure with overlapping address ranges

```
myOverLapStruct : STRUCT OVERLAP
    memb1 AT %B0 : REAL;    // Relative address 0
                            // Bytes 0 .. 3
    memb2 AT %B0 : INT;     // Also on relative address 0
                            // Bytes 0 and 1
    ar    AT %B0 : ARRAY [0..3] OF BYTE;
                            // Relative address 0 again
                            // Bytes 0 .. 3
END_STRUCT
```

If the address ranges of two components overlap, the following data types are **not** permitted in these components:

- STRING
- ANYOBJECT
- TO data type

They are permitted in components whose address ranges do not overlap. For example, the following declaration is permitted:

```
myOverLapStruct_2 : STRUCT OVERLAP
    dint_1  AT %B0 : DINT;
    udint_1 AT %B4 : UDINT;
    ar      AT %B0 : ARRAY [0..7] OF BYTE;
            //Bytes 0 .. 7 overlapping
    pos_1   AT %B8 : PosAxis;
            // Bytes 8 .. 15 not overlapping
END_STRUCT
```

Note

The declaration of a structure with overlapping ARRAY OF BYTE, as specified in the examples, does not replace the marshalling functions. The byte order (Little Endian or Big Endian) depends on the respective SIMOTION device. For information on the Little Endian and Big Endian byte order as well as the marshalling functions, see SIMOTION Basic Functions Function Manual.

Visibility of the components with overlapping address ranges

Symbol information (OPC-XML data) is only generated for the following components:

- Components without overlapping of address ranges
- For several components with overlapping address ranges:
  Only for the last declared component that uses the respective address range. The compiler issues a warning for the hidden components.

Only these components are visible:

- For the watch function of IT DIAG
- For the *_exportUnitDataSet* and *_importUnitDataSet* functions, see SIMOTION Basic Functions Function Manual
- For automatic displays in SIMOTION SCOUT (e.g. symbol browser (Page 393), program status (Page 402)).

Note

Hidden components can used in the programming. In the ST editor, they are shown in the "Automatic completion" (Page 49) function.

You can monitor individual hidden components in a watch table (Page 397), see the appropriate information in Section "Using a watch table" (Page 397).

As an example, the visible components of the following structure declaration are explained in the table below.

The following examples of a structure declaration explain the behavior of the visible and hidden components. The visible components and the associated relative addresses are specified in the following table.

```
mystru : STRUCT OVERLAP
    a  AT %B0 : INT;    // Warning: Not visible for OPC-XML
    b  AT %B6 : INT;    // Warning: Not visible for OPC-XML
    c  AT %B8 : INT;    // Visible
    ar AT %B0 : ARRAY [0..6] OF BYTE; // Visible
END_STRUCT
```

	Relative address									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Declared components	a						b		c	
	ar[0]	ar[1]	ar[2]	ar[3]	ar[4]	ar[5]	ar[6]			
Visible components	ar[0]	ar[1]	ar[2]	ar[3]	ar[4]	ar[5]	ar[6]	-	c	

A change in the declaration order has an effect on the visible components:

```
mystru1 : STRUCT OVERLAP
    ar AT %B0 : ARRAY [0..6] OF BYTE; // Warning: For OPC-XML
                                      // Not visible
    a  AT %B0 : INT;    // Visible
    b  AT %B6 : INT;    // Visible
    c  AT %B8 : INT;    // Visible
END_STRUCT
```

	Relative address									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Declared components	ar[0]	ar[1]	ar[2]	ar[3]	ar[4]	ar[5]	ar[6]			
	a						b		c	
Visible components	a		-	-	-	-	b		c	

Initialization of the components with overlapping addresses

Initialization values are ignored for components that are not visible. The compiler issues an appropriate warning.

If initialization values are specified for the visible components, they are taken into account. These initialization values are also valid for the commonly used addresses of hidden components. The following applies to the addresses of hidden components that are not overlapped by the other components: They are assigned the default initialization value that was defined in the declaration of the appropriate data type, or zero.

The following examples of a structure declaration explain this initialization behavior. The initialization values of the respective relative addresses are specified in the following table.

```
mystru_init : STRUCT OVERLAP
    a  AT %B0 : INT := 16#AAAA;     // Initialization value warning
                                    // OPC-XML warning
    b  AT %B6 : INT := 16#BBBB;     // Initialization value warning
                                    // OPC-XML warning
    c  AT %B8 : INT := 16#CCCC;
    ar AT %B0 : ARRAY [0..6] OF BYTE := [7(1)];
END_STRUCT
```

	Relative address									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Declared components	a						b		c	
	ar[0]	ar[1]	ar[2]	ar[3]	ar[4]	ar[5]	ar[6]			
Initialization value	16#01	16#01	16#01	16#01	16#01	16#01	16#01	16#00	16#CCCC	

A change in the declaration order has an effect on the initialization values:

```
mystru_init1 : STRUCT OVERLAP
    ar AT %B0 : ARRAY [0..6] OF BYTE := [7(1)];
                                    // Initialization value warning
                                    // OPC-XML warning
    a  AT %B0 : INT := 16#AAAA;
    b  AT %B6 : INT := 16#BBBB;
    c  AT %B8 : INT := 16#CCCC;
END_STRUCT
```

	Relative address									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Declared components	ar[0]	ar[1]	ar[2]	ar[3]	ar[4]	ar[5]	ar[6]			
	a						b		c	
Initialization value	16#AAAA		16#00	16#00	16#00	16#00	16#BBBB		16#CCCC	

4.4.3 Technology object data types

4.4.3.1 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 4-21 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
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 4-22 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 134)

Examples of the use of technology object data types (Page 134)

4.4.3.2 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.

4.4.3.3 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 4-23 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 4-24 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.

4.4.4 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)

4.5 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 99)) must not be used as identifiers.

4.5.1 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 137)).
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).

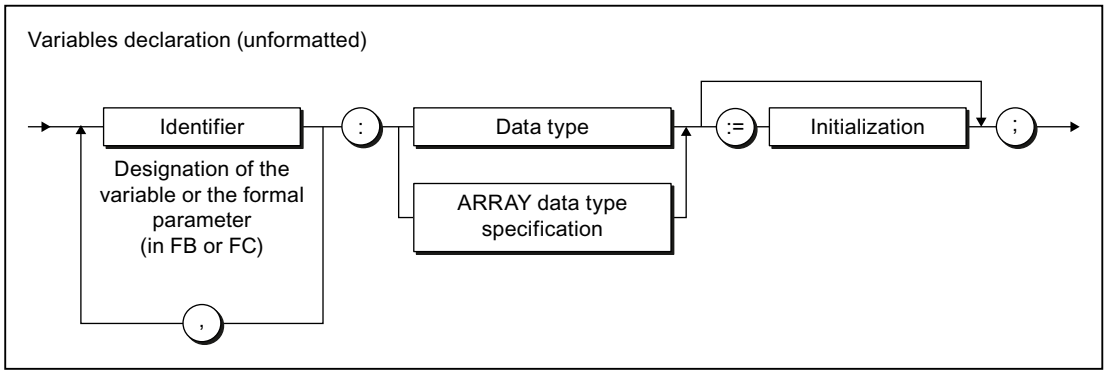


Figure 4-13 Syntax: Variable declaration

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 194)).
- 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 139)).

Deviations from the pattern presented are as follows:

- For constant declarations (a constant **must** be initialized with a value, see Constants (Page 143)),
- For process image access (see Overview of all variable declarations (Page 137)):
  - A variable declaration is not required for absolute process image access,
  - Initialization is not permitted for symbolic process image access.

Table 4-25 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
```

4.5.2 Overview of all variable declarations

You declare 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 sections, refer to Breakdown of ST source file (Page 110) and Source file sections (Page 257).

Table 4-26 Keywords for declaration blocks

Keyword	Meaning	Declaration in the following declaration sections
VAR	Declaration of temporary or static variables See Variable model (Page 283)	Any POU
VAR_GLOBAL	Declaration of unit variables See Variable model (Page 283)	Interface section Implementation section

Keyword	Meaning	Declaration in the following declaration sections
VAR_IN_OUT	Variable declaration of in/out parameter; the POU accesses this variable directly (using a reference) and can change it immediately. See Defining functions (Page 181), Defining function blocks (Page 182), Defining methods (Page 217)	Function Function block Expression Method [1]
VAR_INPUT	Variable declaration of input parameter, value is externally supplied and cannot be changed within the POU. See Defining functions (Page 181), Defining function blocks (Page 182), Defining methods (Page 217)	Function Function block Expression Method [1]
VAR_OUTPUT	Variable declaration of output parameter; value is transmitted from the function block See Defining functions (Page 181), Defining function blocks (Page 182), Defining methods (Page 217)	Function Function block Method [1]
VAR_TEMP	Declaration of temporary variables See Variable model (Page 283)	Program Function Function block Method [1]
RETAIN	Declaration of retentive variables See Variable model (Page 283)	Only as a supplement: • To VAR [2] in POU: – Program – Function block – Class [1] • To VAR_GLOBAL in the: – Interface section – Implementation section
CONSTANT	Declaration of constants See Constants (Page 143)	Only as a supplement: • To VAR in POU: – Program – Function – Function block – Class [1] [2] – Method [1] • To VAR_GLOBAL in the – Interface section – Implementation section
[1] Only with compiler option (Page 63) "Permit object-oriented programming". [2] Only as of version V4.5 of the SIMOTION Kernel.		

4.5.3 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 136) and Syntax of user-defined data types (Page 118).

- 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 an array initialization list to an array (ARRAY) in accordance with the Figure *Syntax: Array initialization list*.
- Assign a structure initialization list to a structure (STRUCT) in accordance with Figure *Syntax: Structure initialization list* (even when the structure is used in other declarations).
  See also the explanation at the end of this section.
- Assign an enumeration element to an enumeration data type.

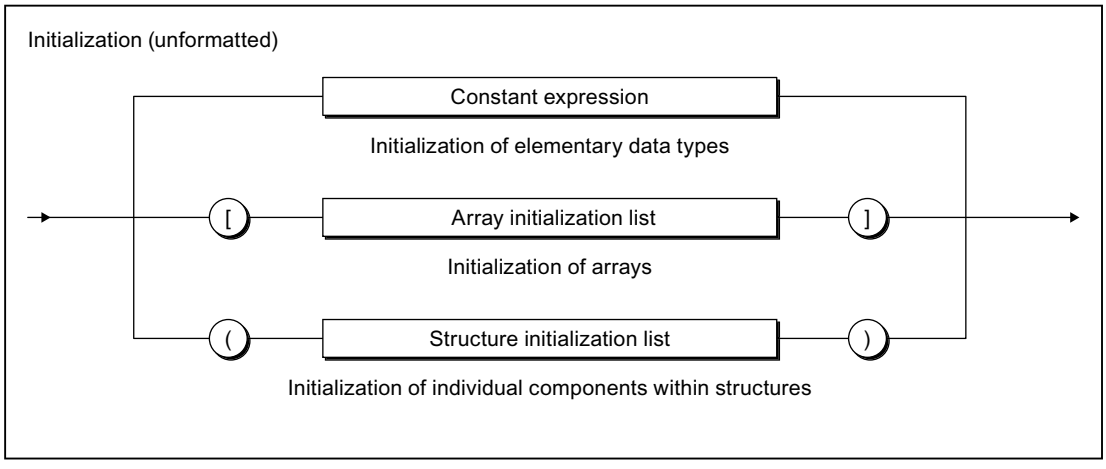


Figure 4-14 Syntax: Variable initialization

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.

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

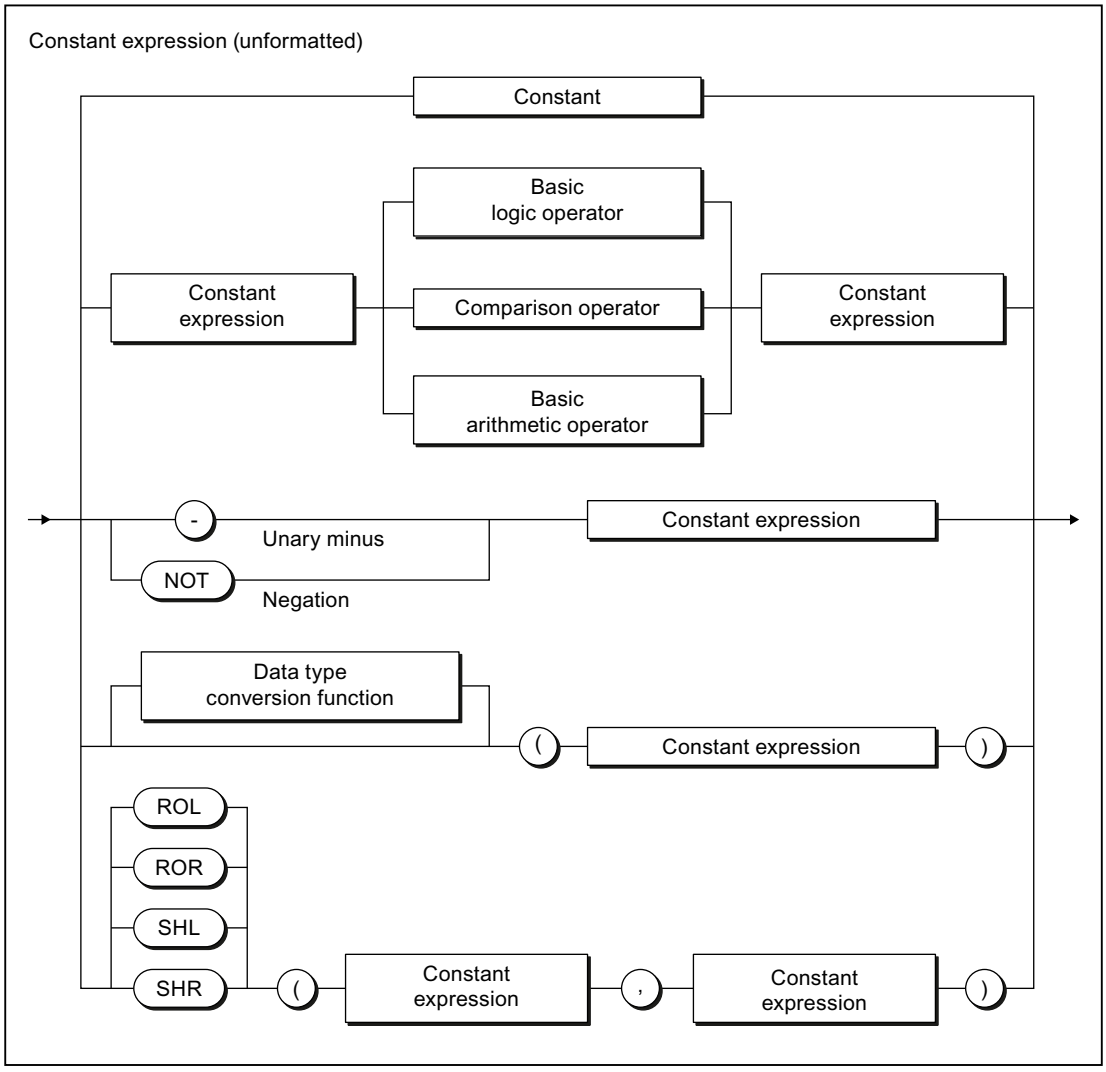


Figure 4-15 Syntax: Constant expression

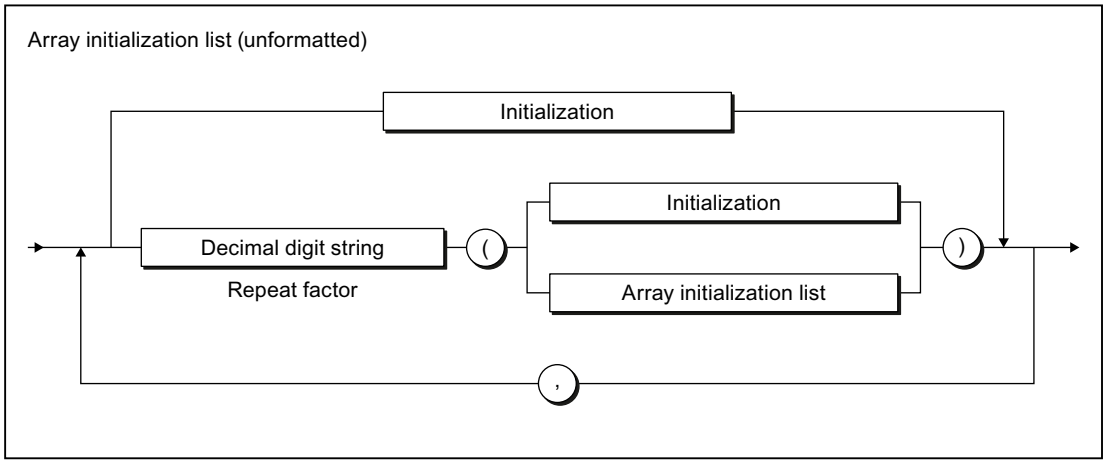


Figure 4-16 Syntax: Array initialization list

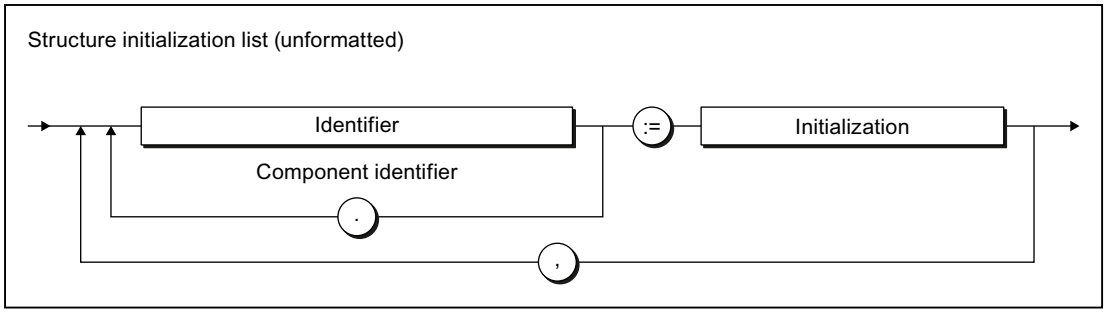


Figure 4-17 Syntax: Structure initialization list

Table 4-27 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 4-28 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
```

Information about the circumstances when variables are initialized can be found in Section "Time of the variable initialization" (Page 300).

Initialization of structures

To initialize a structure (data type STRUCT), assign to it a structure initialization list enclosed in brackets "(" ")". Assign the initialization value to the desired component in this initialization list, see the syntax diagram above.

The following methods can be used to initialize components of lower-level structures:

- You assign another structure initialization list enclosed in brackets to a component and then "nest" the lists inside one another.
- You aggregate the identifiers of the component, subcomponent, etc. as follows (use periods as separators): *comp_name.subcomp_name.subsubsomp_name* and assign the initialization value to this structure.

Note

Saving in an old project format:

Projects in which aggregated identifiers are used cannot be compiled correctly in SIMOTION SCOUT versions earlier than V4.5.

You can use both notations within the same structure initialization list provided that you are initializing different components.

You can initialize arrays within structures by assigning an array initialization list enclosed in square brackets "[" "]" to the component. See example below:

Table 4-29 Examples of structure initialization

```
TYPE
    s_base : STRUCT
        i : INT;
        j : DINT;
    END_STRUCT
    s_deri : STRUCT
        x : INT;
        v_b : s_base := (i := 10);
        arr_b : ARRAY [0..2] OF s_base
           := [(i := 10, j := 15), (i := 12), (j := 13)];
    END_STRUCT
    s_up : STRUCT
        vup_d : s_deri := (x := 3);
        vup_a : ARRAY [0..1] OF s_deri;
    END_STRUCT
END_TYPE
VAR
    // Initialization with nested
    // structure initialization lists
    var_1 : s_up := (vup_d := (v_b := (i := 1) , x := 2));
    // Alternative with aggregated component identifiers
    var_2 : s_up := ( vup_d.v_b.i := 1, vup_d.x := 2);
    // Initialization of arrays of structures
    var3 : s_up := (vup_a := [ (v_b := (i := 5)), (v_b.i := 6) ] );
    var4 : ARRAY [0..7] OF s_up := [ 5((vup_d := (v_b := (i := 1)))),
                                     3((vup_d.v_b.i := 1)) ];
END_VAR
```

4.5.4 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 283)).

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

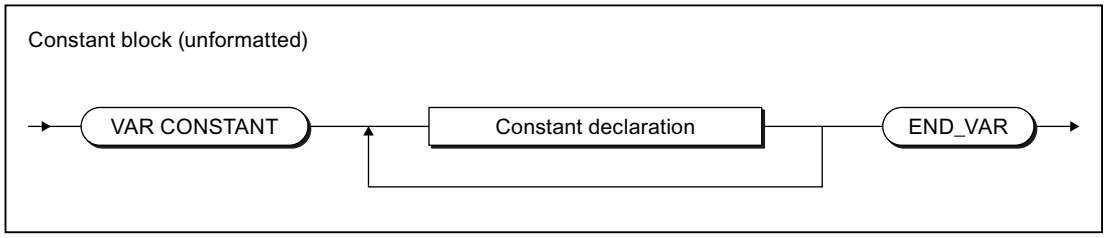


Figure 4-18 Syntax: Constant block in a POU

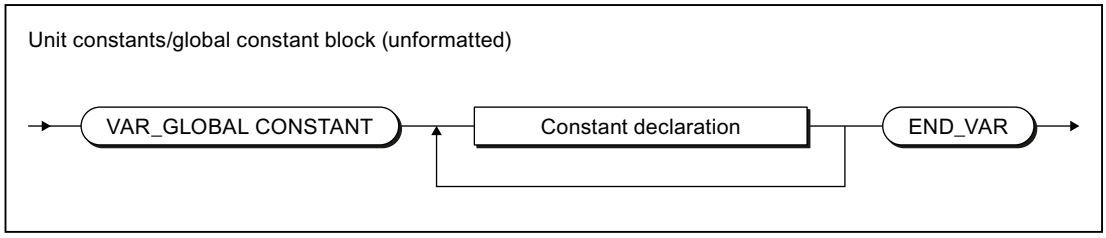


Figure 4-19 Syntax: Unit constants in interface or implementation section

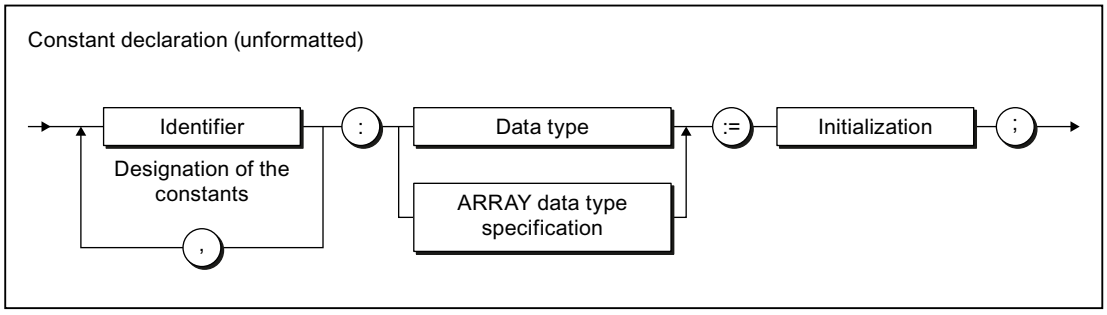


Figure 4-20 Syntax: Constant declaration

To initialize a constant, see "Initialization of variables or data types (Page 139)". The value assigned to a constant is calculated from the constant expression at the time of compilation.

Table 4-30 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
```

4.6 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 112)) 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 387)

4.6.1 Value assignments

4.6.1.1 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 136)).

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

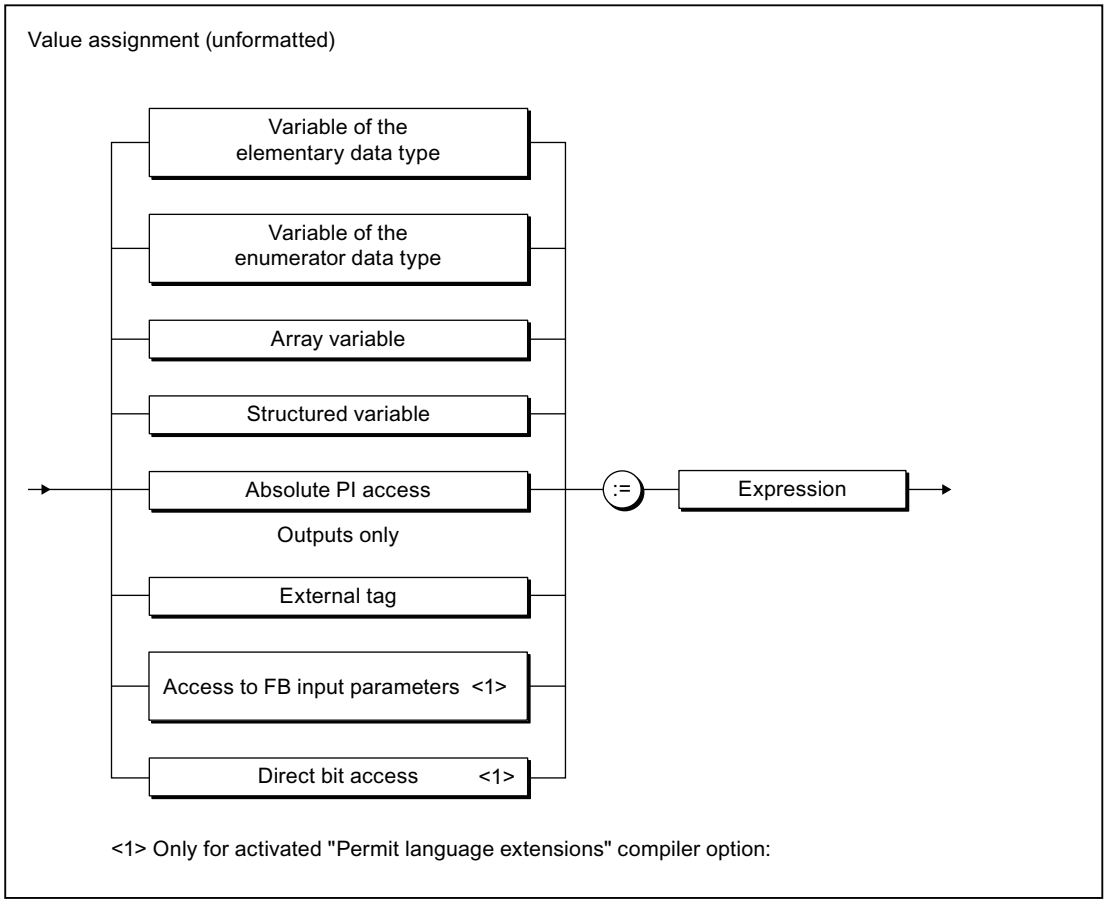


Figure 4-21 Syntax: Value assignment

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

- Value assignments with variables of an elementary data type (Page 147) ,
- Value assignments with variables of the derived enumerator data type (Page 150)
- Value assignments with variables of the derived ARRAY data type (Page 150)
- Value assignments with variables of the derived STRUCT data type (Page 151)
- 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 338).

How the right side of a value assignment, i.e. an expression, is formed, is described in Expressions (Page 152).

4.6.1.2 Value assignments with variables of an elementary data type

An expression with an elementary data type (Page 114) 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 147).
- 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 176) 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 148)

4.6.1.3 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 196).

Please also refer to Syntax diagram of STRING data type (Page 114):

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 176) and the *SIMOTION Basic Functions* Function Manual.

4.6.1.4 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 63) and Local compiler settings (Page 67).

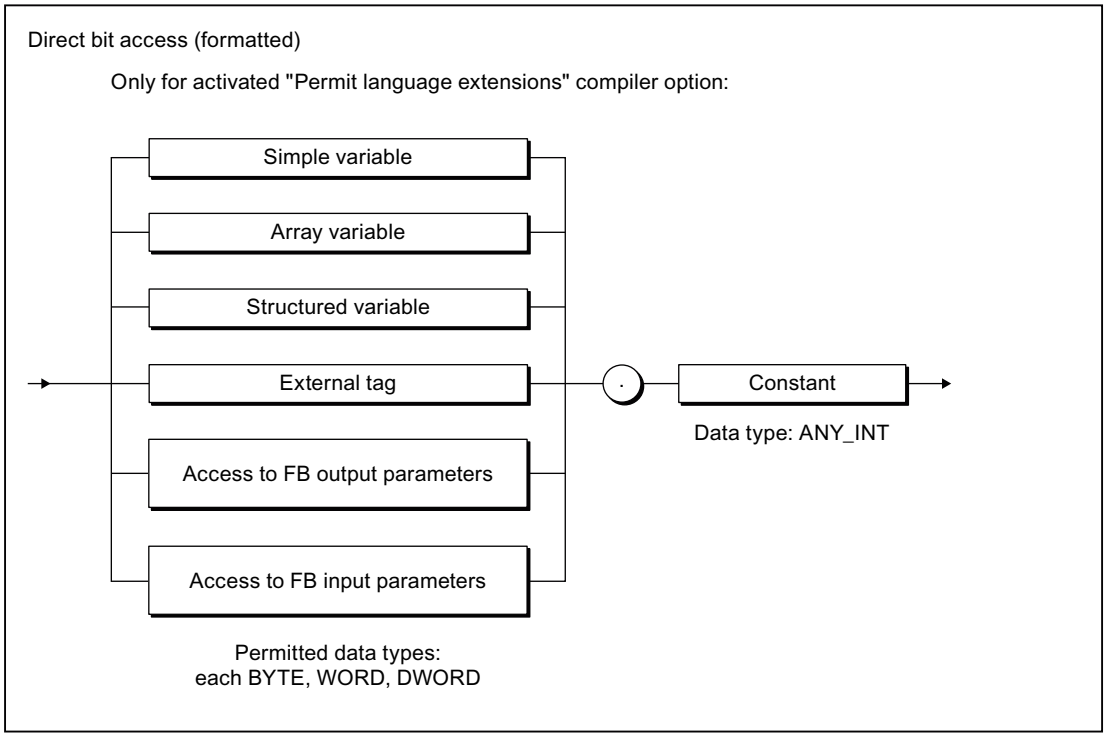


Figure 4-22 Syntax: Direct bit access

Table 4-31 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.
    dw.BIT_7 := b   // Access to Bit 7
    dw.3     := NOT dw.3  // Access to Bit 3
END_FUNCTION
```

Note

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 string 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 160).

4.6.1.5 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 123)) can be assigned to another variable of the same type.

```
type1        := BLUE;
```

4.6.1.6 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 120)).

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];
```

4.6.1.7 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 124)). 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 181) and Defining function blocks (Page 182).

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

4.6.2 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 155)
- Relational expressions (Page 158)
- Logic expressions and bit-serial expressions (Page 160)

4.6.2.1 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 139)):

- 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.

4.6.2.2 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 162)).
- 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 4-32 Examples of expressions

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

4.6.3 Operands

Definition

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

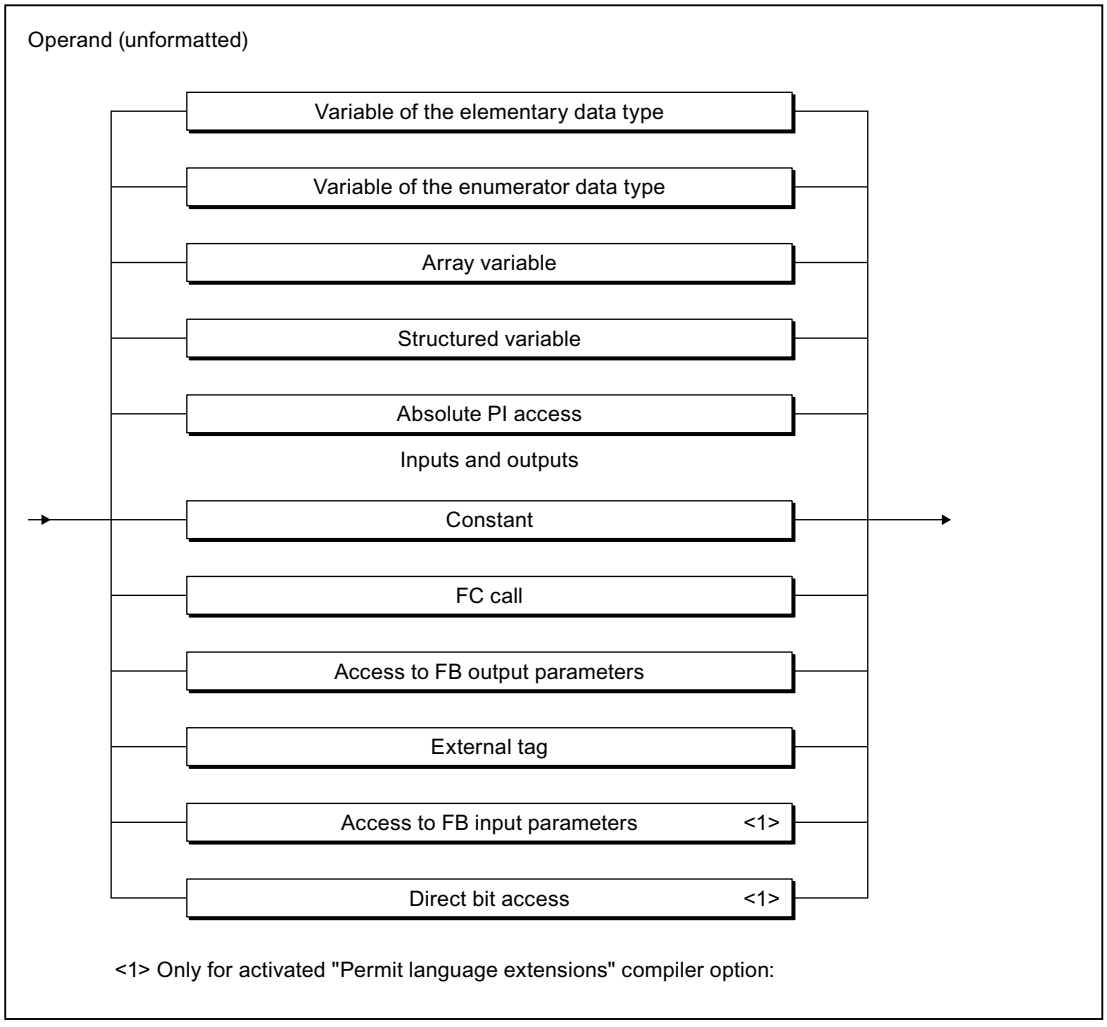


Figure 4-23 Syntax: Operand

Table 4-33 Examples of operands

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

4.6.4 Arithmetic expressions

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

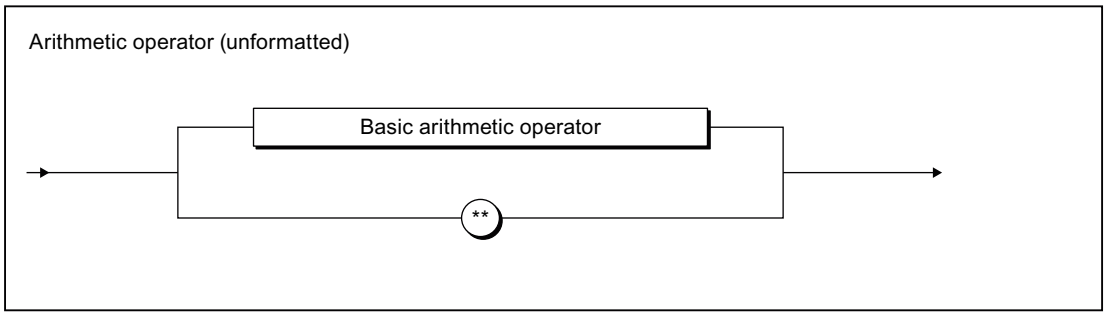


Figure 4-24 Syntax: Arithmetic operator

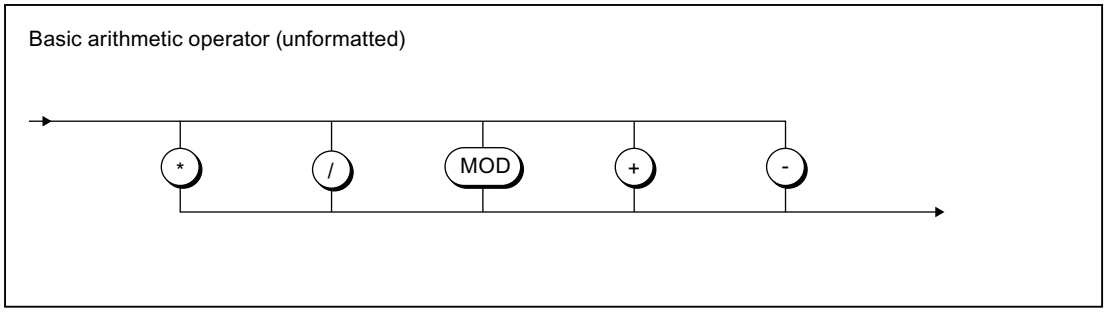


Figure 4-25 Syntax: Basic arithmetic operator

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 116) 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 162).

Table 4-34 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 1st operand must be greater than zero.
Exceptions as of version V4.1 of the SIMOTION Kernel:
– If the 2nd operand is an integer, the 1. operand can be less than zero.
– If the 2nd operand is positive, the 1st operand can be equal to zero.
The following applies up to version V4.0 of the SIMOTION Kernel: If the 1st operand is equal to zero, an error message can be caught with ExecutionFaultTask.

3) Data type of the 1st operand.

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

5) The 2nd 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).

4.6.4.1 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
```

4.6.5 Relational expressions

Definition

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

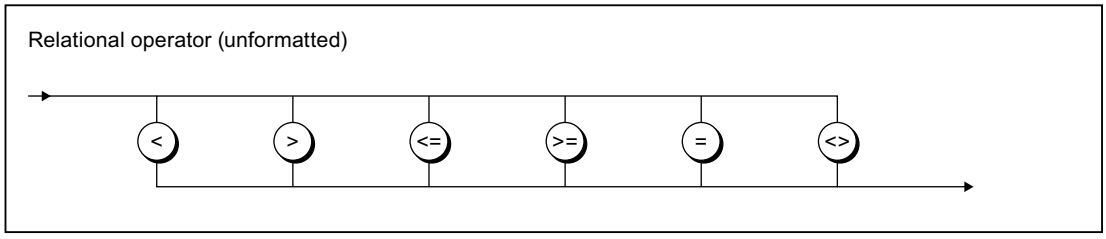


Figure 4-26 Syntax: Relational operators

Relational operators compare the values of 2 operands (see table) and return a Boolean value as result.

1st Operand Operator 2nd Operand -> Boolean value

Table 4-35 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 4-36 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	<, >, <=, >=, =, <>

Data type		Permissible relational operators
1. Operand	2. Operand	
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 1. 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 160)). This enables the implementation of queries such as *If a < b and b < c, then ...*.

Note

Relational operators have a higher priority than logic operators in an expression (see Operator priority (Page 162)). 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 4-37 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
```

4.6.6 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 4-38 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 4-39 Truth table of the logic operators

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

Examples

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`

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.

4.6.7 Priority of operators

Some general rules for the formulation of expressions were described in Expressions (Page 152). The table shows you the priority of the individual operators within an expression.

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

4.7 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.

4.7.1 IF statement

Description

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.

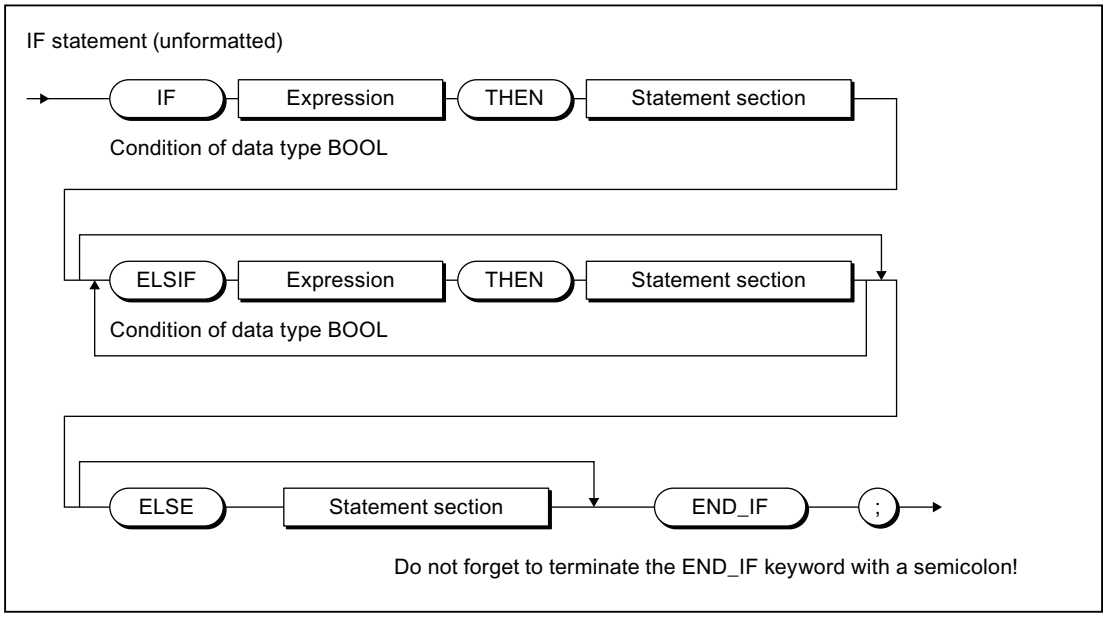


Figure 4-27 Syntax: IF statement

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:

Example 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;
```

4.7.2 CASE statement

Description

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.

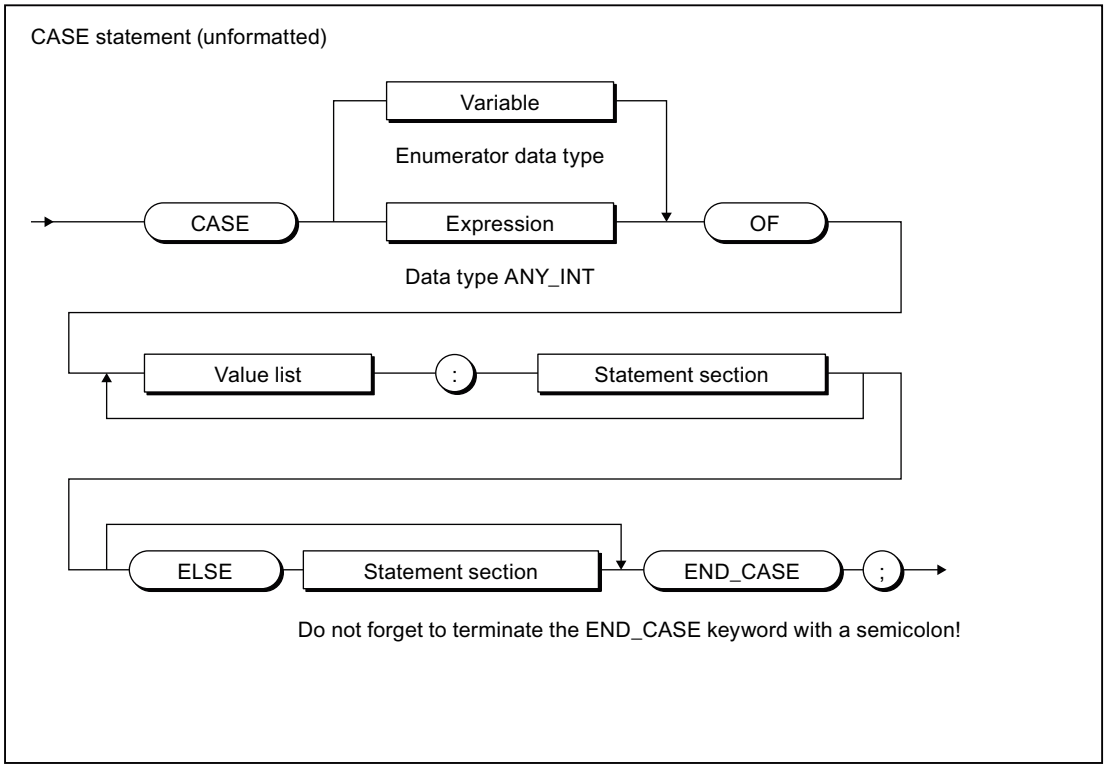


Figure 4-28 Syntax: CASE statement

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.

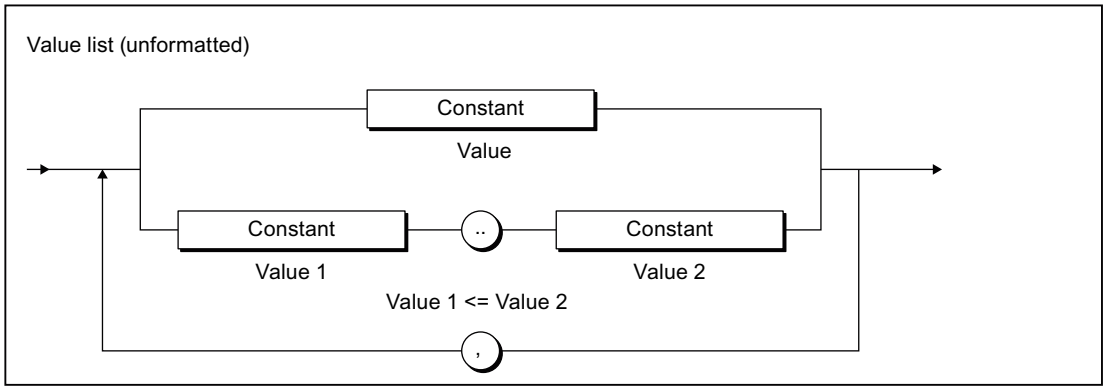


Figure 4-29 Syntax: Value list

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 constants or 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:

Example 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;
```

4.7.3 FOR statement

Description

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 cycles is not known, the WHILE or REPEAT statement is more suitable, see WHILE statement (Page 169) and REPEAT statement (Page 170).

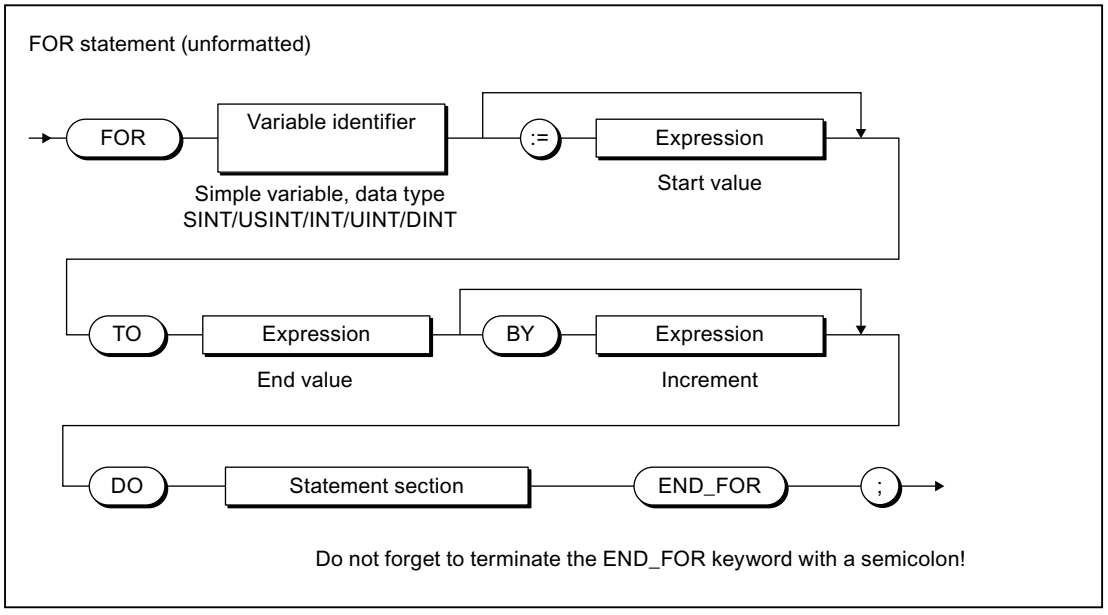


Figure 4-30 Syntax: FOR statement

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 content of the count variable 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 152). The expression is evaluated once at the beginning of the FOR statement.
- If the start value and end value are of the data type DINT, the amount from (end value - start value) must be less than DINT#MAX (2**31 - 1), see also Value range limits of elementary data types (Page 116).
- The count variable contains the value which triggers the loop exit, i.e. it is incremented before the loop is exited.
- During the loop execution, the count variable (current value) as well as the start value, the end value and the increment must not be changed.

Example

The following example illustrates the use of the FOR statement:

Example of the FOR statement

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

4.7.4 WHILE statement

Description

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 (Page 167) is more suitable.

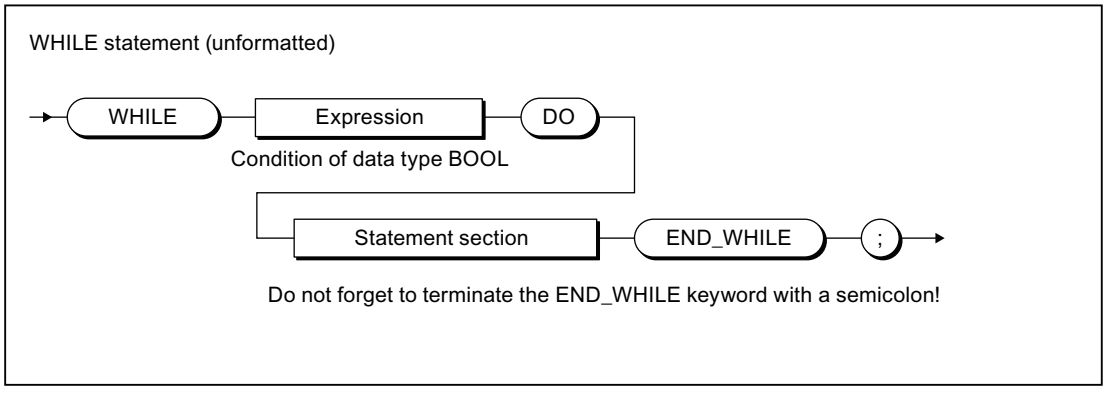


Figure 4-31 Syntax: WHILE statement

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:

Example of the WHILE statement

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

4.7.5 REPEAT statement

Description

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 (Page 167) is more suitable.

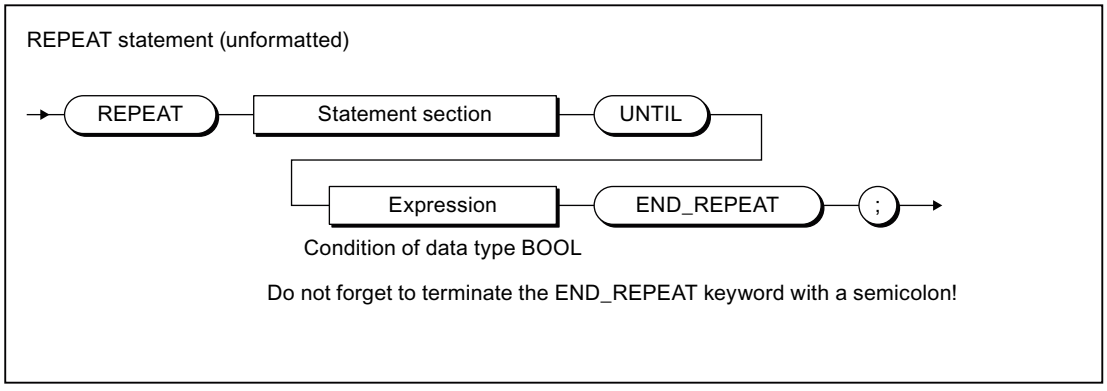


Figure 4-32 Syntax: REPEAT statement

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.

Sequence of execution

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:

Example of the REPEAT statement

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

4.7.6 EXIT statement

Description

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:

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;
```

4.7.7 CONTINUE statement

Description

The CONTINUE statement is only available if the compiler option **Language extensions IEC61131 3rd edition** is enabled, see Global settings of the compiler (Page 63) and Local settings of the compiler (Page 67).

It acts as a jump to the start of a loop (FOR, WHILE or REPEAT loop) from any point. Use this to jump to the start of the repeat statement which immediately surrounds the CONTINUE statement.

- With a FOR loop the run tag is increased by the programmed increment and compared with the full-scale value.
- With a WHILE loop the execution condition is reviewed.

Example

The following example illustrates the use of the CONTINUE statement:

Example of the CONTINUE statement:

```
sum := 0;
FOR i:= 1 TO 3 DO
    FOR j:= 1 TO 2 DO
        sum:= sum + 1;
        IF %I0.0 THEN
            CONTINUE;
        END_IF;
        sum:= sum + 1;
    END_FOR;
    sum:= sum + 1;
END_FOR;
(*
Once the loop has run the "sum" variable has the following value:
With %I0.0 = FALSE: sum = 15
With %I0.0 = TRUE:  sum =  9
*)
```

4.7.8 RETURN statement

Description

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:

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;
```

4.7.9 WAITFORCONDITION statement

Description

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 208).

More information about the WAITFORCONDITION and expressions in this regard is contained in the *SIMOTION Basic Functions* Function Manual.

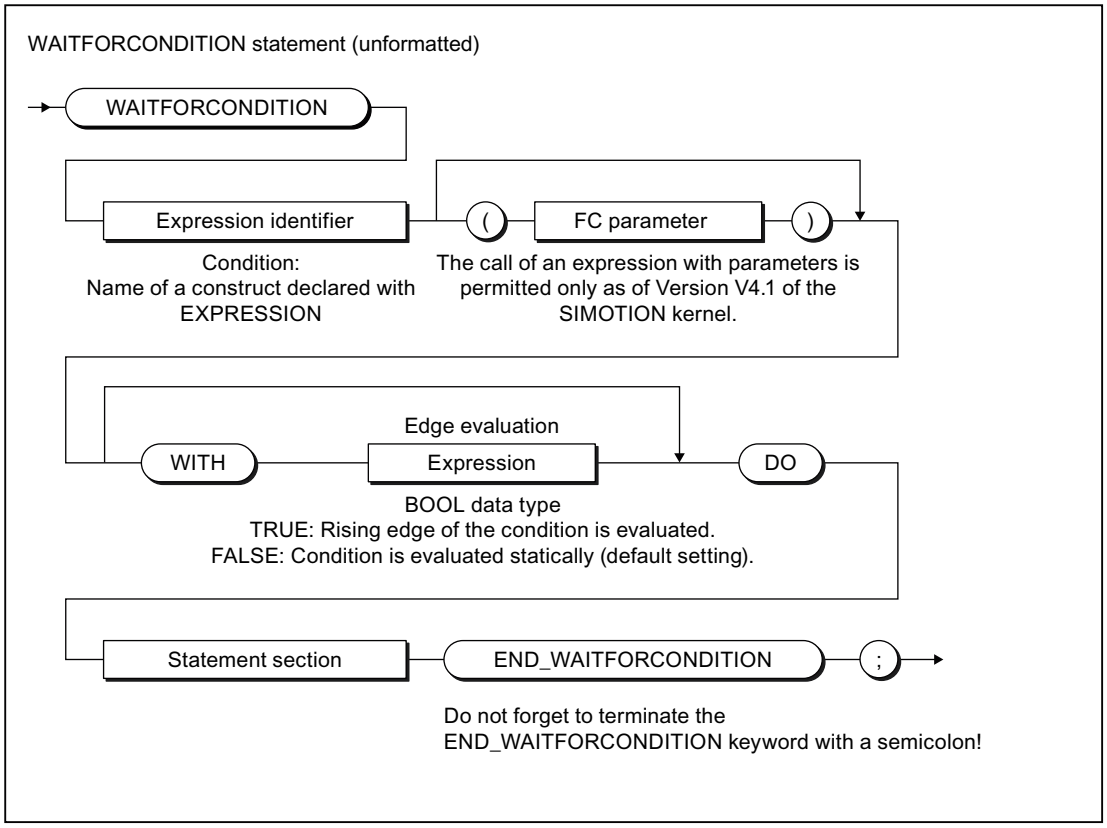


Figure 4-33 Syntax: WAITFORCONDITION statement

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:

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 208).

4.7.10 GOTO statement

The GOTO statement causes a jump to the jump label specified in the command (see Jump statement and labeling (Page 385)).

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.

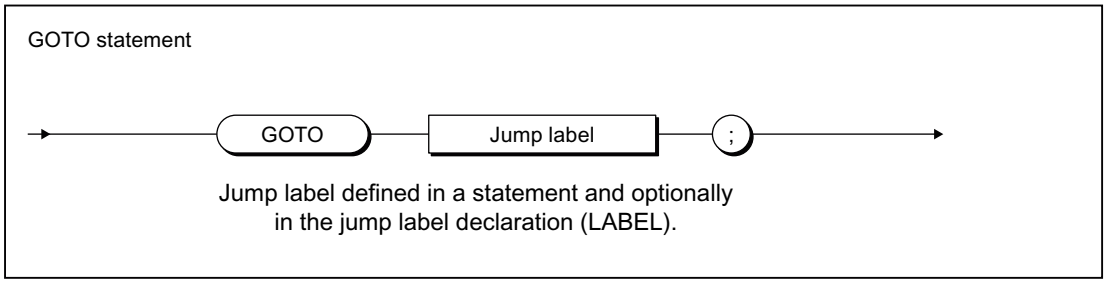


Figure 4-34 Syntax: GOTO statement

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.

4.8 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.

4.8.1 Elementary data type conversion

The table presents an overview of the conversion options between numerical data types and bit data types. A distinction is made between:

- Implicit conversion (Page 176): Conversion is automatic when different data types are used in an expression or when values are assigned by the compiler.
- Explicit conversion (Page 178): Conversion is carried out when the user calls a conversion function (see *SIMOTION Basic Functions* Function Manual).

Table 4-40 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 by means of type conversion function *source data type*_TO_*target data type*

Val: Explicit data type conversion possible by means of type conversion function*source data type*_VALUE_TO_*target data type*

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.

4.8.1.1 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.

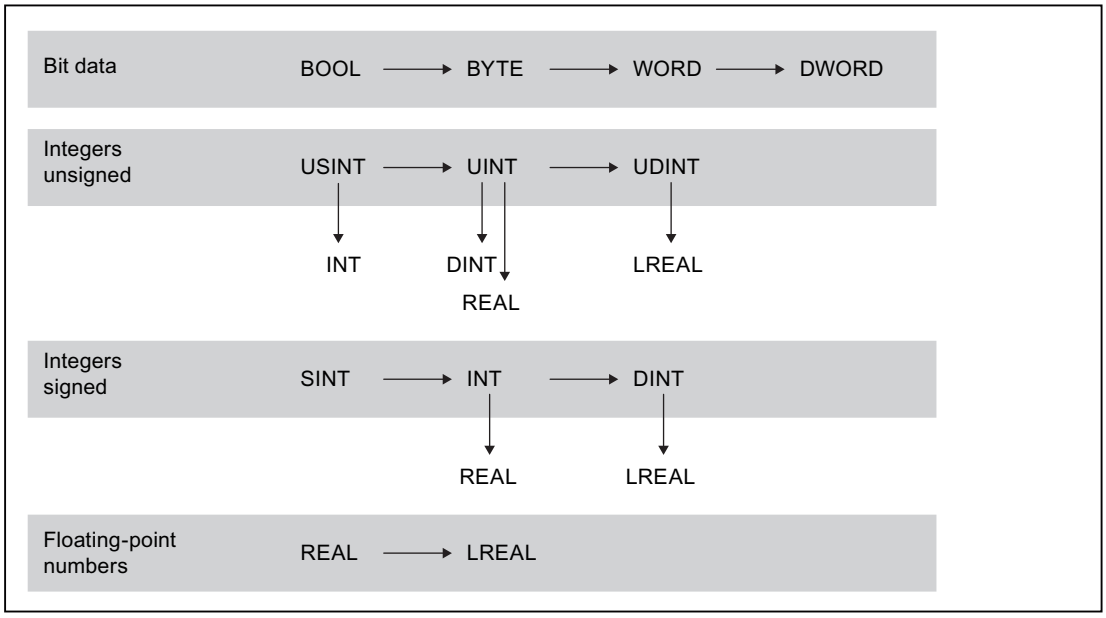


Figure 4-35 Implicit type conversion chains (one or more levels from left to right or one level from top to bottom)

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 (Page 178) (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 4-41 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).

4.8.1.2 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.

Note

The type conversion may cause errors when the program is running, which will trigger the error response set in the task configuration (see "Execution errors in programs" in the *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 can be triggered (FPU exception) 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.

4.8.2 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 5

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 257).

5.1 Creating and calling functions and function blocks

The following description explains how to create and call functions (FCs) and function blocks (FBs). For a complete example showing the differences between FCs and FBs see Comparison of functions and function blocks (Page 203).

The order in which you must define and call the stipulated source file sections is given in Use of the source file sections (Page 257).

For a description of how to make FCs and FBs public at other program sources or make them available for use by other program sources, see Declaring ST source files public and using them (Page 278).

5.1.1 Defining functions

You can define a function (FC) within the implementation section of an ST source file in the section for program organization units (POE). If the compiler option (Page 63) “Allow forward declaration" is not activated a FC must be defined before the POE (program, FB or FC) in which it is called.

Use the following syntax:

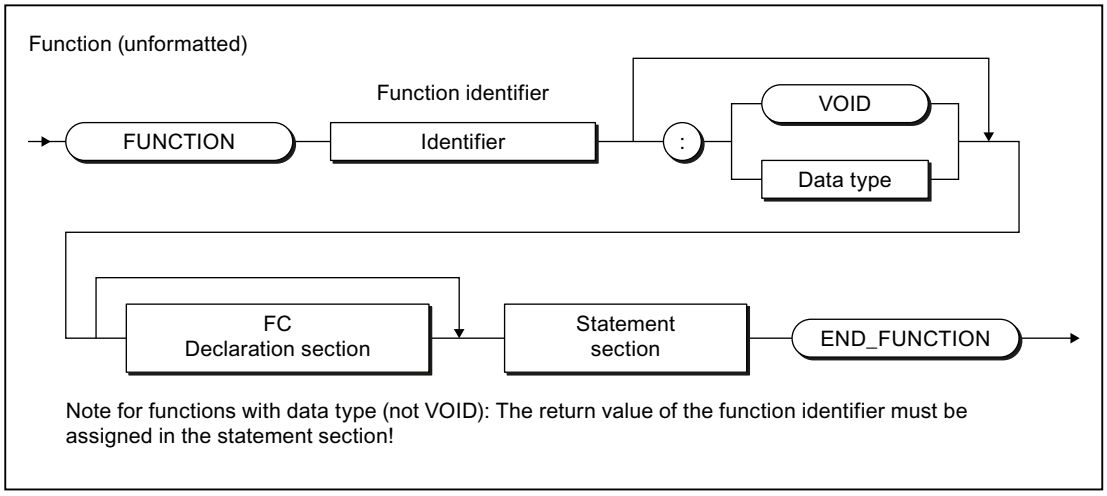


Figure 5-1 Syntax: Function (FC)

Enter in the following sequence, see example in Source file with comments (Page 205)):

- The FUNCTION keyword
- An identifier as FC name
- If the function has a return value, the data type for the return value following a colon
  The symbolic data type VOID can also be entered for functions with no return value.

This is then followed by:

- The optional declaration section (Page 186)
- The statement section (Page 191)
- The END_FUNCTION keyword

5.1.2 Defining function blocks

5.1.2.1 Defining classic function blocks

You can define a function block (FB) within the implementation section of an ST source file in the section for program organization units (POE). If the compiler option (Page 63) "Allow forward declaration" is not activated a FB must be defined before the POE (program, FB or FC) in which it is called.

Use the following syntax:

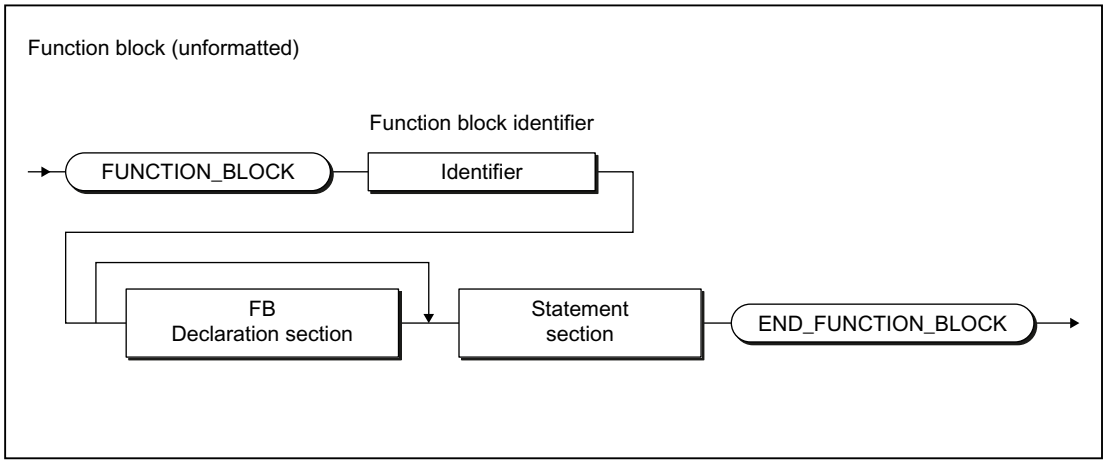


Figure 5-2 Syntax: Function block (FB)

Enter in the following sequence, see example in Source file with comments (Page 205)):

- The FUNCTION_BLOCK keyword
- An identifier as FB name

This is then followed by:

- The optional declaration section (Page 186)
- The statement section (Page 191)
- The END_FUNCTION_BLOCK keyword

5.1.2.2 Defining object-oriented function blocks with methods

Function blocks can also be defined with methods. Reduced object-oriented programming is possible with this irrespective of the SIMOTON Kernel version. Only the compiler option (Page 63) “Permit object-oriented programming" needs to be activated.

You can define object-oriented function blocks (FB) with methods and public variables within the implementation section of an ST source file in the section for program organization units (POE). If the compiler option (Page 63) “Allow forward declaration" is not activated a FB must be defined before the POE (program, FB or FC) in which it is called.

Use the following syntax:

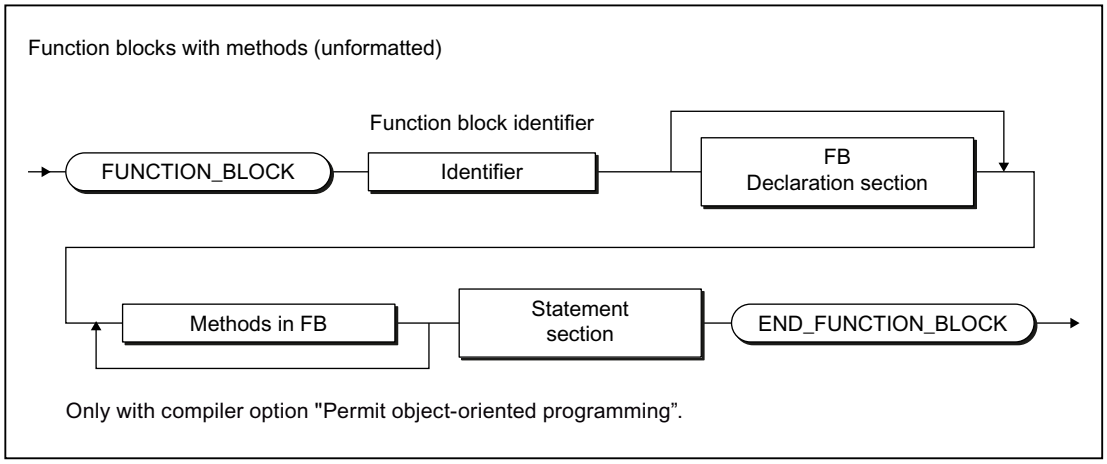


Figure 5-3 Syntax: Function block (FB) with methods

Enter in the following sequence:

- The FUNCTION_BLOCK keyword
- An identifier as FB name

This is then followed by:

- The optional declaration section (Page 186)
- the methods in the function block
- The statement section (Page 191)
- The END_FUNCTION_BLOCK keyword

Note

Object-oriented function blocks cannot be derived. Inheritance is therefore not possible for methods.

5.1.2.3 Defining methods in object-oriented FBs

Executable statements for an object-oriented function block can be summarized as methods. The methods are defined within a function block before the statement section for the FB.

A method roughly corresponds to a function (Page 181). Specification of an access identifier determines the environment from which the method can be called.

If the compiler option (Page 63) "Allow forward declaration" is not activated, a method must be defined before the program organization unit (program, FB, FC or class) or method in which it is called.

Use the following syntax:

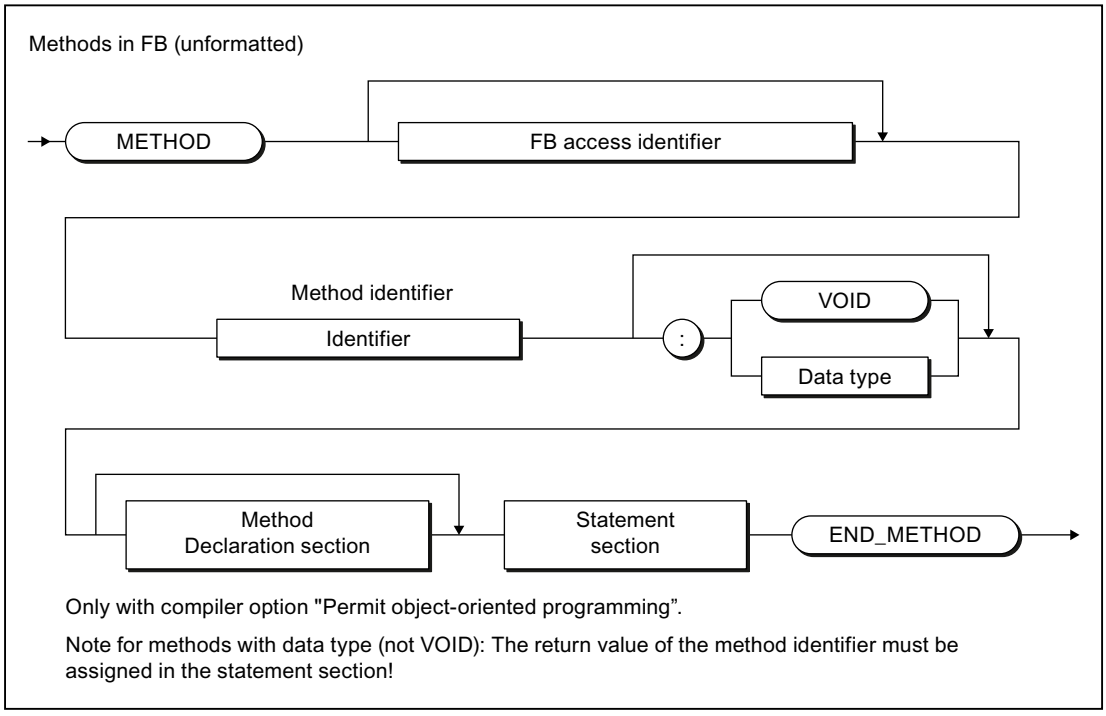


Figure 5-4 Syntax: Methods in the function block

Enter in the following sequence:

- The METHOD keyword.
- The optional FB access identifier (PUBLIC, PRIVATE or INTERNAL), default PRIVATE.
  For access identifier, see section "Access identifiers within object-oriented FBs" (Page 185).
- An identifier as method name.
- if the method has a return value, the data type for the return value following a colon.
  The symbolic data type VOID can also be entered for a method with no return value.

This is then followed by:

- The optional declaration section (Page 218)
- The statement section (Page 220)
- The END_METHOD keyword

5.1.2.4 Access identifiers within object-oriented FBs

An access identifier may be specified within object-oriented function blocks for declaration blocks or methods:

- With declaration blocks:
  Directly after the keyword for the relevant block (e.g. TYPE, VAR).
  The access identifier is valid for all identifiers within this declaration block.
- With methods:
  Directly after the keyword METHOD.
  The access identifier is valid for the relevant method.

You use the access identifier to determine whether the identifiers or the method can be used only inside or outside of the function block, see the following table.

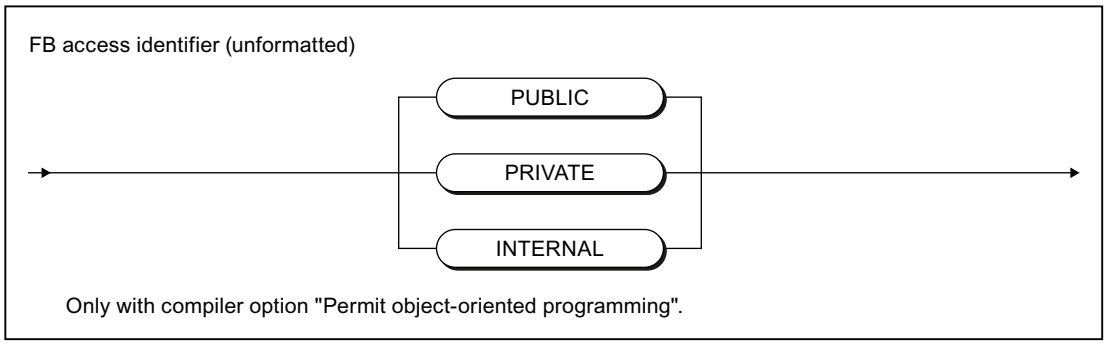


Figure 5-5 Syntax: FB access identifier

Table 5-1 Possible access identifiers within an object-oriented function block

Access identifier	Meaning
PRIVATE	The identifiers for the relevant declaration block (variables, data types, etc.) or the method can be used only within this function block. This is the default when no access identifier is specified.
INTERNAL	Only when the identifier is defined within an object-oriented namespace (Page 355). The identifiers of the relevant declaration block (variables, data types, etc.) or the method can be used only within the higher-level namespace. Within the higher-level namespace, the following apply: • The declared identifiers can also be accessed outside the function block. This does not apply to retentive local variables (VAR RETAIN). • The method can also be called outside the function block.
PUBLIC	The identifiers for the relevant declaration block (variables, data types, etc.) or the method are public. • The declared identifiers can also be accessed outside the function block and outside the higher-level object-oriented namespace. • The method can also be called outside the function block and outside the higher-level object-oriented namespace. This access identifier is not permitted in declaration blocks for retentive local variables (VAR RETAIN / END_VAR).

5.1.3 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 205):

Permissible declaration blocks

Table 5-2 Declaration blocks for FC and FB: Options

Data	Syntax	FB	FC	Method in FB
Data type	**TYPE**[1] *Declaration list*[4] **END_TYPE**	X	X	–
Constant	**VAR CONSTANT**[1] *Declaration list*[4] **END_VAR**	X	X	X
Input parameter	**VAR_INPUT** *Declaration list*[4] **END_VAR**	X	X	X
In/out parameter	**VAR_IN_OUT** *Declaration list*[4] **END_VAR**	X	X	X
Output parameters	**VAR_OUTPUT** *Declaration list*[4] **END_VAR**	X	X	X
Local static variable	**VAR**[1 3] *Declaration list*[4] **END_VAR**	X	–	–
Local temporary variable	**VAR** *Declaration list*[4] **END_VAR**	–	X	X
	VAR_TEMP *Declaration list*[4] **END_VAR**	X	X	X
Retentive variable (as of version 4.5 of the SIMOTION Kernel)	**VAR RETAIN**[2 3] *Declaration list*[4] **END_VAR**	X	–	–

[1] Additional specification of the FB access identifier possible with FB and compiler option (Page 63) "Permit object-oriented programming" activated: **PUBLIC**, **PRIVATE** or **INTERNAL**, default **PRIVATE**; see section "Access identifier within object-oriented FBs (Page 185)". The relevant blocks may occur multiple times.

[2] Additional specification of the access identifier **PRIVATE** or **INTERNAL** possible with FB and activated compiler option (Page 63) "Permit object-oriented programming". The relevant blocks may occur multiple times.

[3] Additional specification of the keyword **OVERRIDE** possible with FB and activated compiler option (Page 63) "Permit object-oriented programming". This keyword enables the initialization values of variables to be overridden with the **PRIVATE** access identifier when the instance is declared.

[4] *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, a function or a method in FB. 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 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, an FC or a method in the FB.

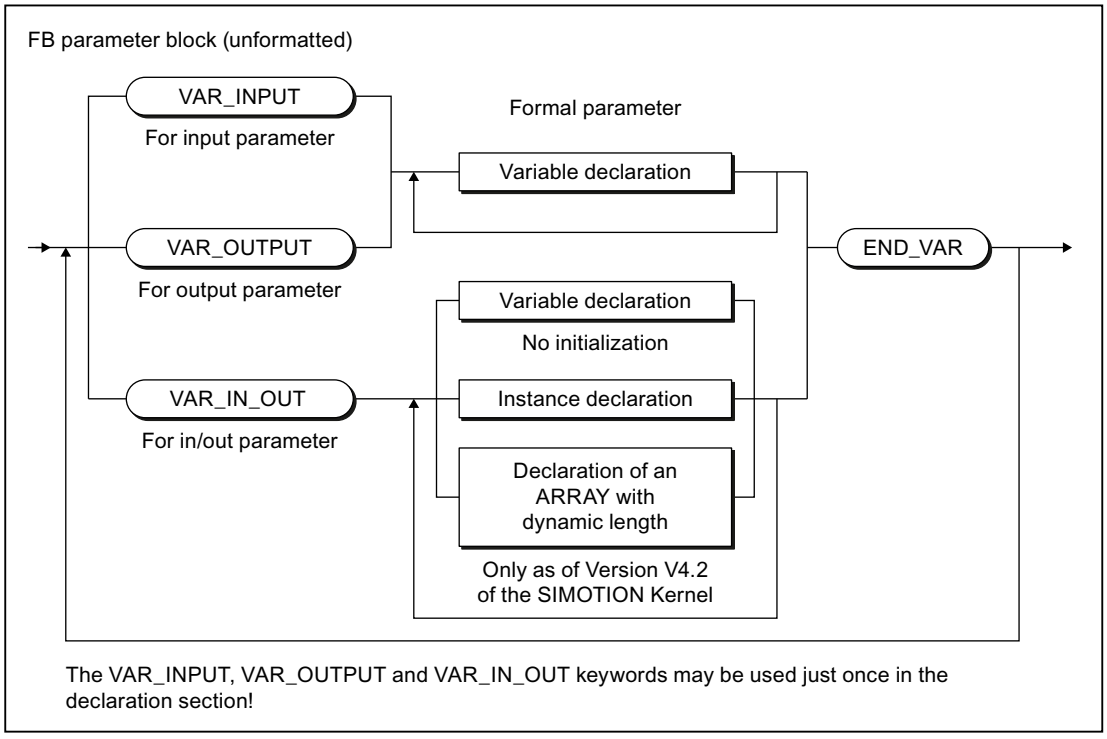


Figure 5-6 Syntax: FB parameter block

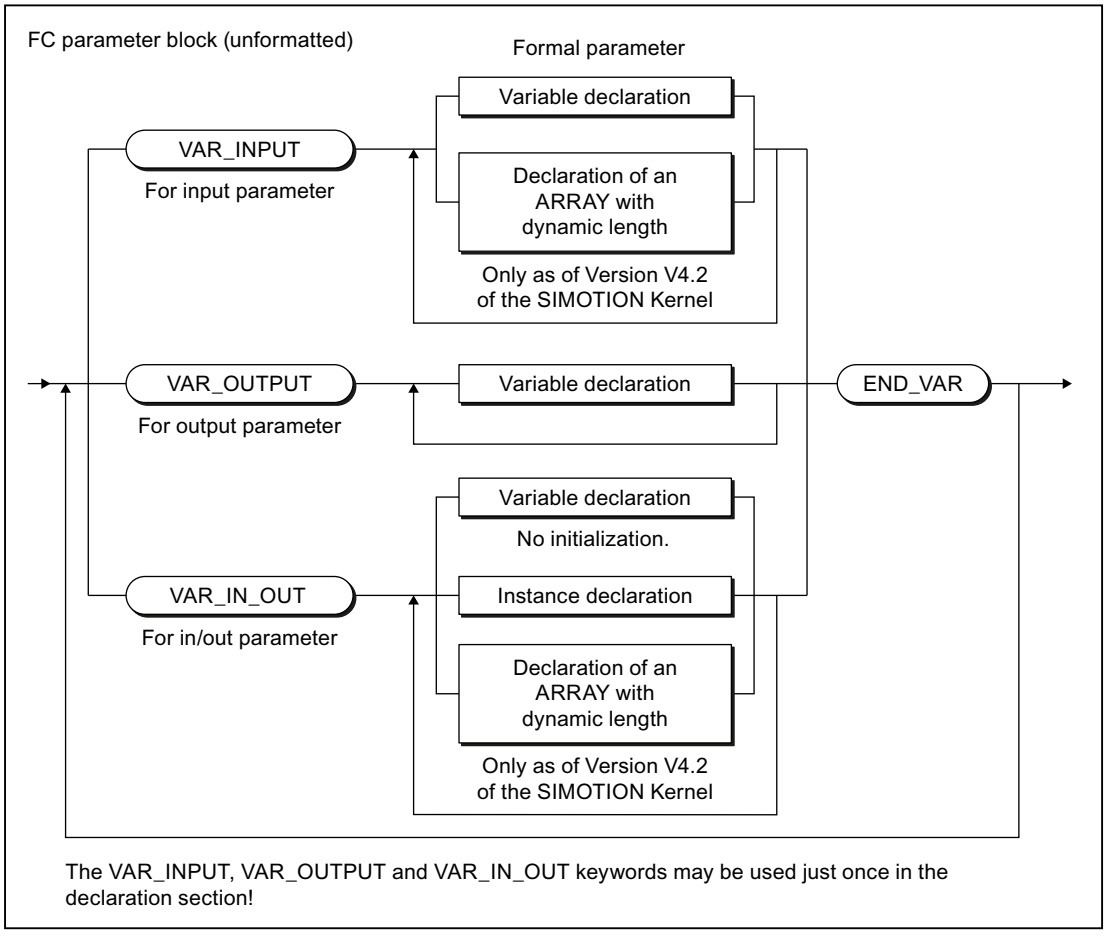


Figure 5-7 Syntax: FC parameter block

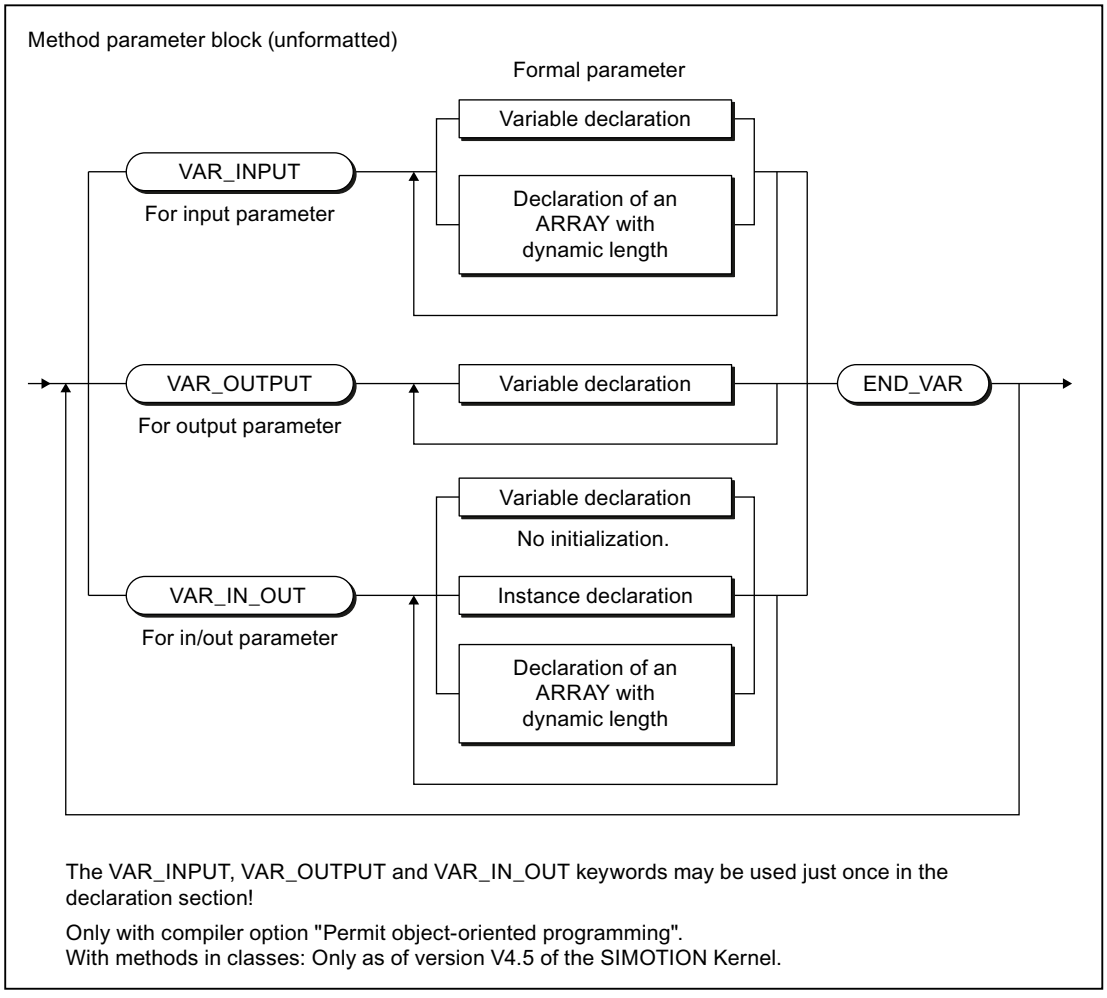


Figure 5-8 Syntax: Method parameter block

Apart from variables you can also declare the following in the parameter blocks:

- in the parameter block for in/out parameters:
  - Instances of function blocks (Page 199)
  - Arrays with a dynamic length (Page 192), as of version V4.2 of the SIMOTION Kernel.
- with functions in the parameter block for input parameters:
  - Arrays with a dynamic length (Page 192), as of version V4.2 of the SIMOTION Kernel.

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 124). Input parameters can only be accessed when the compiler option (Page 63) "Permit language extensions" 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.

Note

When saving the project in the old project format:

Projects in which the output parameters for a function are declared cannot be compiled correctly in SIMOTION SCOUT versions earlier than V4.5.

5.1.4 Statement section of FB and FC

The statement section of the FC or FB contains instructions 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 the Runtime-optimized Programming section in the SIMOTION Basic Functions Function Manual.

Table 5-3 Use of parameters and variables in FCs and FBs

Parameter/variable	Use
Input parameters	With the call of an FC or an FB, assign the current values to the input parameters. These values are used for data processing within the FC or the FB, for example, for calculations, but cannot be modified themselves. Only with "Permit language extensions" compiler option enabled (see Global compiler settings (Page 63) or Local compiler settings (Page 67)): The input parameters of an FB can be read and written using structured variables, including outside the FB (e.g. in the calling source file module).
In/out parameter	You assign a variable to an in/out parameter for the call of the FC or FB. The FC or the FB accesses this variable directly and can change this directly. Type conversions are not supported. The variable assigned to an in/out parameter must be able to be read and written directly. 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	You assign a variable to an output parameter for the call of an FC or FB using the **=>** operator. The value of the output parameter (result) is transferred to the variables when the FC or FB is closed. The following also applies to FBs: The output parameters of an FB can be read using structured variables, including outside the FB (e.g. in the calling source file section).
Local variables	Local variables are variables that are declared and used only within the block. All local variables (VAR ... END_VAR or VAR_TEMP ... END_VAR) are temporary in an FC, i.e. they lose their value when the FC is terminated. The next time the FC is called, they are reinitialized. A differentiation between static and temporary local variables is made in the FB: • Static variables (VAR ... END_VAR) retain their value when the FB is closed. • Temporary variables (VAR_TEMP ... END_VAR) lose their value when the FB is closed. The next time the FB is called, they are reinitialized. The value of a local variable cannot be queried directly by the calling block. This is only possible using an output parameter.

5.1.5 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 possible:

- With functions:
  - in the parameter block for in/out parameters (Page 186) (VAR_IN_OUT / END_VAR).
  - in the parameter block for input parameters (Page 186) (VAR_INPUT / END_VAR).
- With function blocks:
  in the parameter block for in/out parameters (Page 186) (VAR_IN_OUT / END_VAR).

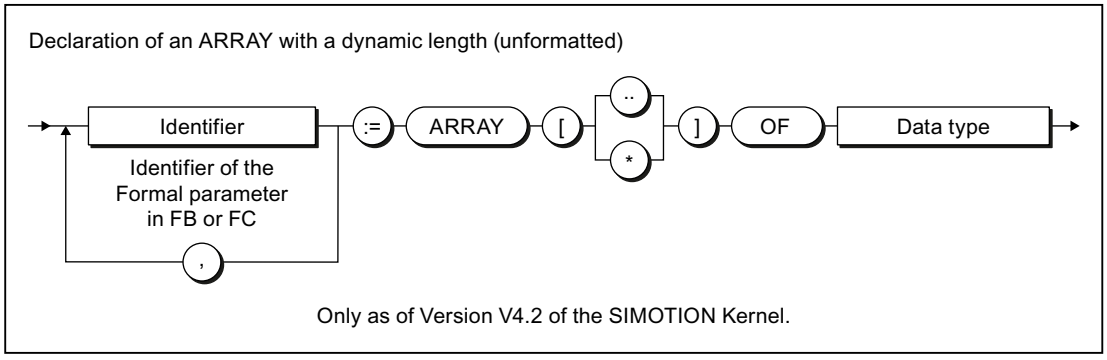


Figure 5-9 Syntax: Declaration of an ARRAY with a dynamic length

Note

When saving the project in the old project format:

Projects in which [*] is used to identify a dynamic array cannot be compiled correctly in SIMOTION SCOUT versions earlier than V4.5.

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 299).

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.

Note

With compiler option (Page 63) "Permit language extensions IEC61131 3rd edition" enabled the standardized functions LOWER_BOUND (in := *array-name*) or UPPER_BOUND (in := *array-name*) can be used to determine the lower and upper index limits respectively.

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 5-4 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
```

5.1.6 Call of functions and function block calls

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

5.1.6.1 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.

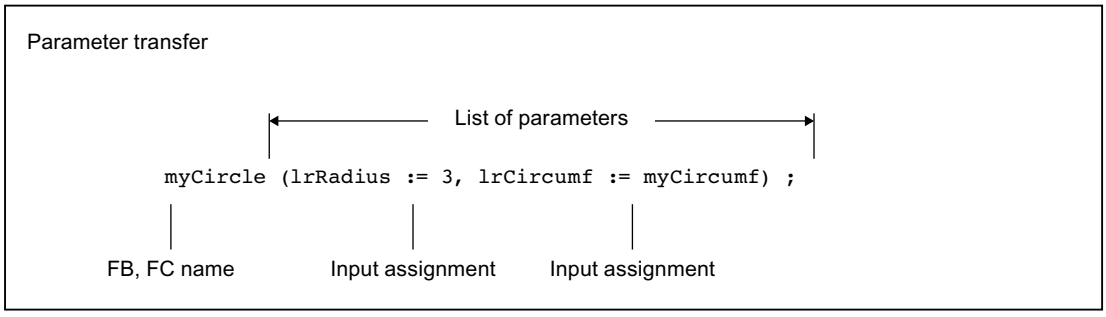


Figure 5-10 Principle of parameter transfer for the call

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).

5.1.6.2 Parameter transfer to input parameters

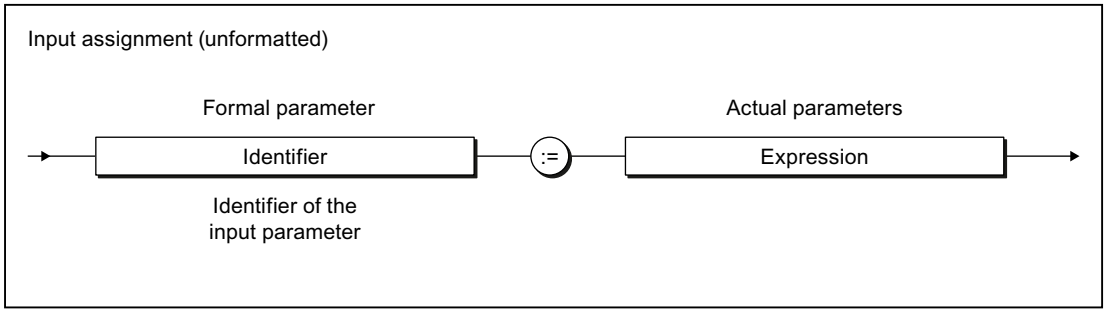


Figure 5-11 Syntax: Input assignment

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 198) and Calling function blocks (instance calls) (Page 199).

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 201).

5.1.6.3 Parameter transfer to in/out parameters

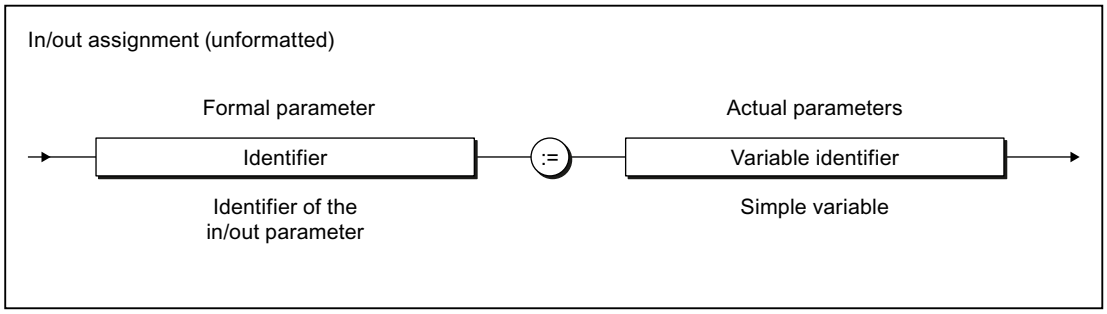


Figure 5-12 Syntax: In/out assignment

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 directly.

See also the examples in Calling functions (Page 198) and Calling function blocks (instance calls) (Page 199).

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 5-5 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);    // Permissible call
    my_fb (io := str2);    // Illegal call,
                          // Compiler error message
END_FUNCTION_BLOCK
```

The variable assigned to an in/out parameter must be able to be read and written directly. 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!

5.1.6.4 Parameter transfer to output parameters

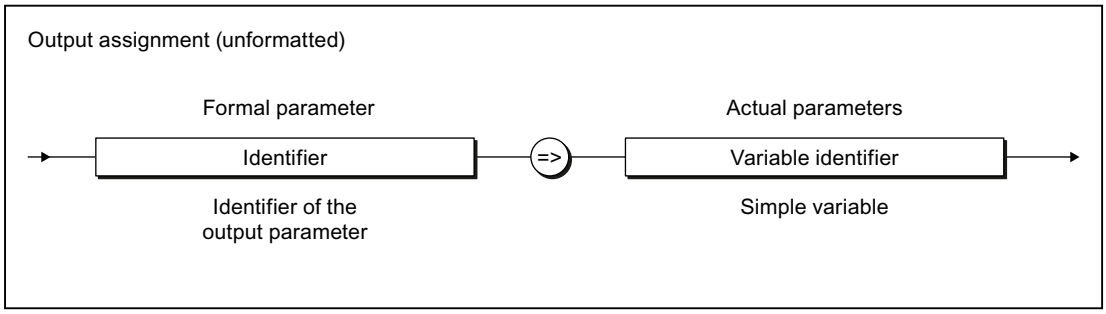


Figure 5-13 Syntax: Output assignment

You use an output assignment to assign the formal output parameters of an FC or 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 FC or the FB.

See also the examples in Calling functions (Page 198) and Calling function blocks (instance calls) (Page 199).

Output assignments are optional for the parameter transfer. If there is no output assignment for an output parameter, the following applies:

- With functions (FC):
  The values for the output parameters are lost after the FC is terminated.
- With function blocks (FB):
  You can gain read and write access to an FB's output parameter at any time, including outside the FB. For further details, see: Accessing the FB's output parameter outside the FB (Page 201).

5.1.6.5 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 283)).

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.

5.1.6.6 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 an 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));
```

Note

In the function itself, the result (return value) is assigned to the function name.

Function without return value (no data type or VOID data type)

The assignment consists only of the function call.

The following example is valid provided a funct1 function has already been defined with the in1 and in2 input parameters along with the inout in/out parameter and the out1 and out2 output parameters.

Example:

```
funct1 (in1 := var11, in2 := var12, inout1 := var13, out1 => var14, out2 =>
var15);
```

5.1.6.7 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).

Instance-specific initialization of individual local variables of the function block is possible in the case of function blocks declared in ST source files by means of the compiler option (Page 67) "Permit object-oriented programming". In this case, you override the initialization values that were stipulated in the function block declaration. Specification of the variables to be initialized is similar to the process for initializing a structure, and involves specifying a structure initialization list (Page 141) that is enclosed in brackets.

You can in all cases initialize public variables of the function block (i.e. variables with the PUBLIC access identifier) specific to the instance. In order to be able to initialize private variables of the function block (i.e. variables with PRIVATE access identifier), the keyword OVERRIDE must be stated at their declaration block after the access identifier.

The following generally applies: The function block must be declared in the ST source file before an instance can be declared.

Exception: It is sufficient to define a POU prototype (Page 381) beforehand if initialization is not specified and when compiler option (Page 63) "Permit forward declarations" is activated.

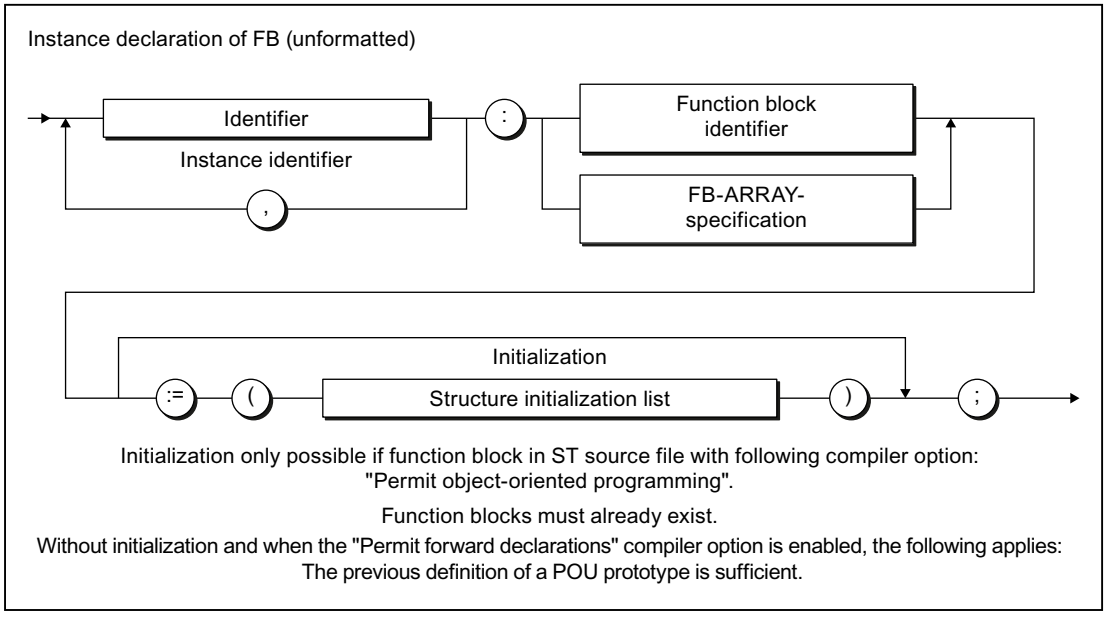


Figure 5-14 Syntax: Instance declaration of FB

Example of the instance declaration of an FB with initialization:

```
FB_inst : FB_name := (Var_publ1 := 12,
                      Var_publ2 := 123.456);
```

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 the input, in-out and output assignments separated by commas.

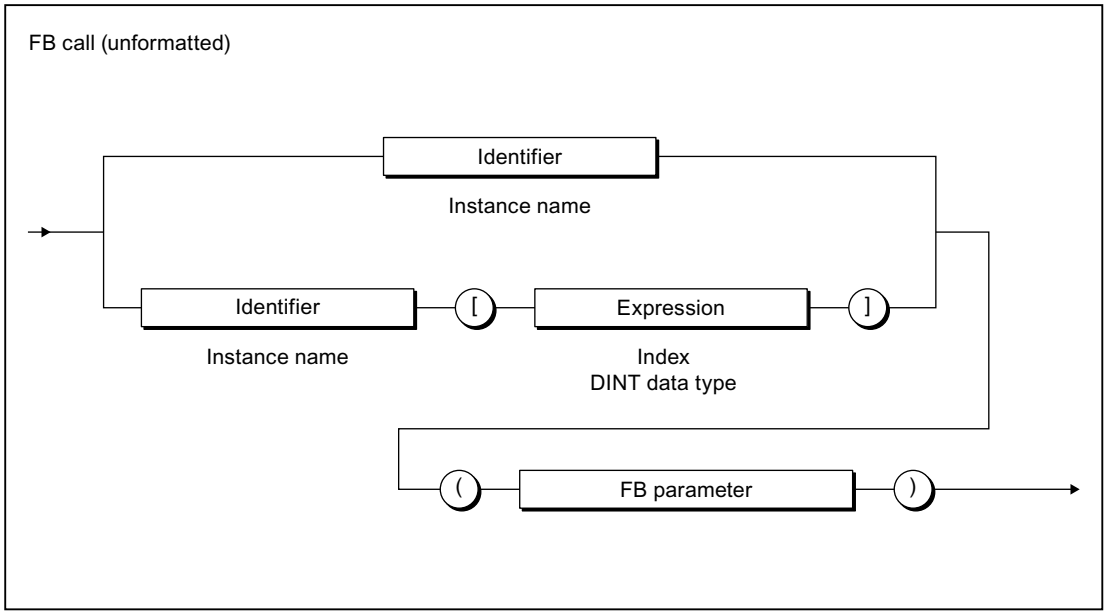


Figure 5-15 FB call syntax

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 5-6 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);
```

5.1.6.8 Accessing the FB's output parameter outside the FB

In addition to the output assignment (Page 197) 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 124) in the *FB instance name.output parameter* format, e.g. *Supply1.out*.

Also refer to the examples in Calling function blocks (instance calls) (Page 199).

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

5.1.6.9 Accessing the FB's input parameter outside the FB

In addition to the input assignment (Page 195) 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 124) in the *FB instance name.input parameter* format, e.g. *Supply1.in1*.

Note

To be able to use this option, the "Permit language extensions" compiler option must be activated, see Global compiler settings (Page 63) and Local compiler settings (Page 67).

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

Table 5-7 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();
```

5.1.6.10 Access an FB's public variables outside of the FB

How to access a public local variable of a function block (i.e. variable with PUBLIC access identifier):

1. Declare an instance of the FB (unless this has already been done).
   See Calling function blocks (declaring and calling instances) (Page 199).
2. For access purposes use a structured variable (Page 151) in the format *FB-instancename.variablename*, e.g. *fb_inst.var_name*.

You can have read/write access to the public variable.

You access the public constants or data types of a function block in a similar manner.

5.1.6.11 Call methods within and outside of a FB

Inside a FB you can call all methods defined within this, irrespective of the access identifier (PUBLIC or PRIVATE). The call works in the same way as with a function, see Calling a function (Page 198).

You can also call the public methods for a function block (i.e. methods with PUBLIC access identifier) outside of this FB, e.g. in programs, functions, function blocks or methods from other function blocks. Proceed as follows:

1. Declare an instance of the FB (unless this has already been done).
   See Calling function blocks (declaring and calling instances) (Page 199).
2. To call the method, write the name of the FB instance and a period before the method name (similar to a structured variable (Page 151)).
   The parameter transfer takes place as described in the Calling a function (Page 198) section.

5.1.6.12 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.**

5.2 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.

5.2.1 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 181) and Define function blocks (Page 182).

A block will be created as an FB and an FC in the implementation section of an ST source file 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.

5.2.2 Source file with comments

Table 5-8 Example of differences between FB and FC

Function block (FB)	Function (FC)

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
	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 5-9 Example of the differences between FB and FC for the previous example

Function block (FB)	Function (FC)
Comments	
Reserved words for the definition: FUNCTION_BLOCK and END_FUNCTION_BLOCK	Reserved words for the definition: FUNCTION and END_FUNCTION
No return value permitted.	The data type of the return value must be specified after the name (VOID data type, if no return value).
Input parameters can be used to transfer values to the FB.	Input parameters can be used to transfer values to the FC.
In/out parameters can be used to read and write the transferred variables in the FB.	In/out parameters can be used to read and write the transferred variables in the FC.
Output parameters can be used to return values from an FB.	No output parameters permitted.
The local variables are static, i.e. they retain their value between FB calls. The *Counter* local variable is incremented; its value is retained when the FB is closed. The variable is, therefore, incremented each time the FB is called. To see this behavior: Assign the value of the local variables to a global variable in the FB. Monitor the value of the global variable after repeated FB calls.	The local variables are temporary, i.e. they lose their value when the function is terminated. Although the *Counter* local variable is incremented, its value is lost when the FC is exited. The variable is reinitialized (to 0 in the example) at the next FC call. To see this behavior: Assign the value of the local variables to a global variable in the FC. The value of the global variable remains unchanged after repeated FC calls.
In the statement section, the results (return values) are assigned to the output or in/out parameters.	In the statement section, the result (return value) is assigned to the function name (except when VOID data type is specified).
In the declaration section of the block that executes the call, an instance of the FB is declared: you declare a variable and specify the name of the FB as its data type. You use the declared instance name to call the FB and to access its output parameters. The name of the FB itself must not be used in the statement section.	
• You assign a variable to the in/out parameters when the FB instance is called. • With the call, you can assign the output parameters to a variable. • You can read an FB's output parameters, even outside the FB. For this purpose, use structured variables in the following format: *FB-instancename.outputparameter*.	• You assign a variable to the in/out parameters when the FB instance is called. • To obtain the return value of the FC: – Assign the function to a variable. – Use the function in an expression on the right side of a value assignment.
The program that executes the call cannot access variables other than the in/out variables and output parameters of the FB. Exception: If the "Permit language extensions" compiler option is activated (see Global compiler settings (Page 63) or Local compiler settings (Page 67)), the calling program can also access the input parameters of an FB. For this purpose, use structured variables in the following format: *FB-instancename.inputparameter*.	The program that executes the call cannot access any variables other than the return value.

5.3 Programs

Programs are a series of statements placed between the PROGRAM and END_PROGRAM keywords.

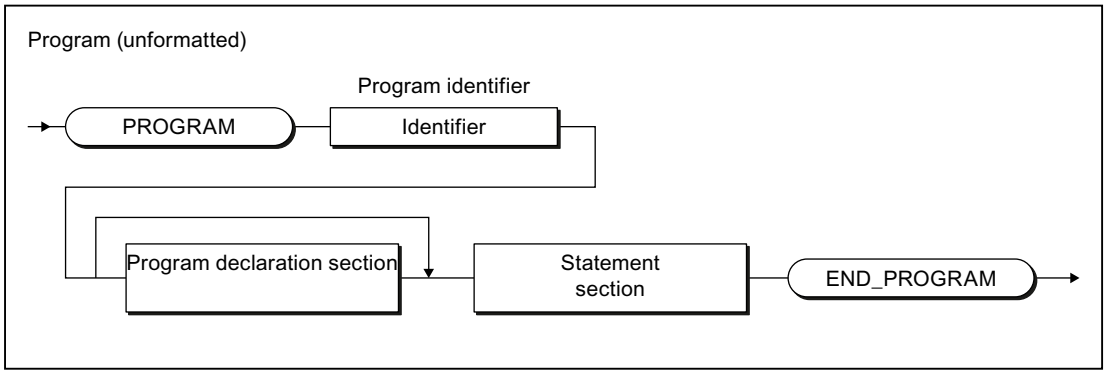


Figure 5-16 Syntax: Program

Programs are declared in the implementation section (Page 260) of an ST source file and are comparable with the FB. Static local variables (VAR ... END_VAR), retentive local variables (VAR RETAIN ... 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 of programs are contained in the Source file with comments (Page 205) and Source text of the sample program (Page 88) sections.

5.3.1 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. initialization of variables, is determined by the relevant task. 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 258) of the ST source file to be specified as a program organization unit to be declared public.

5.3.2 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 63) and Local compiler settings (Page 67):

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.

Note

The activated "Create program instance data only once" compiler option causes:

- The static variables of the programs (program instance data) to be stored in a different memory area (Page 295). This also changes the initialization behavior (Page 305).
- All called programs with the same name to use the same program instance data.

Table 5-10 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.

5.4 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 173).

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

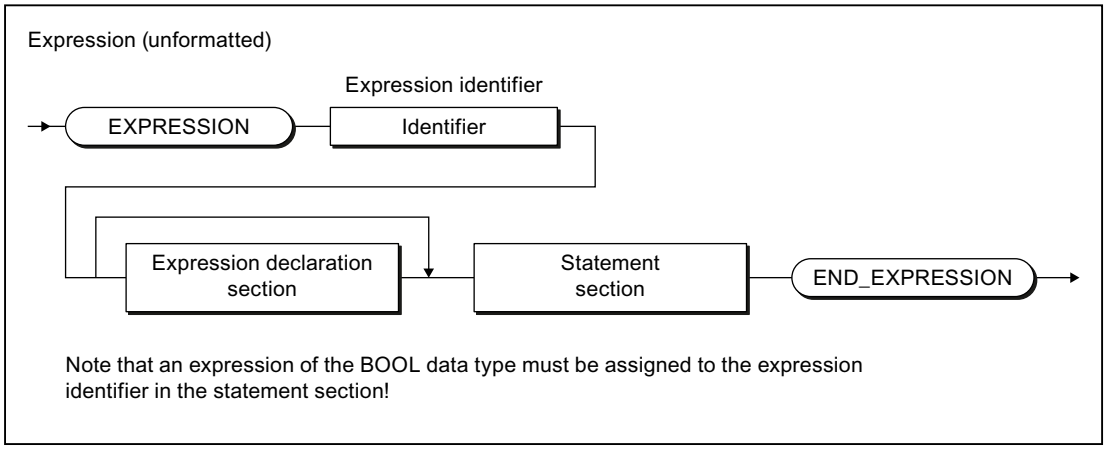


Figure 5-17 Syntax: Expression

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:

- The local variables of the expression
- 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 Description of Functions).

Table 5-11 Example 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.

Object-oriented programming - OOP (as of kernel V4.5)

6

6.1 Important note on object-oriented programming

As of version V4.5 the Structured Text programming language provides options for object-oriented programming.

You can use the object-oriented programming in individual or in all ST source files under the following conditions.

Conditions for object-oriented programming

The following conditions must be met in order to use the object-oriented programming in an ST source file:

- The higher-level SIMOTION device must be configured with a version of the SIMOTION Kernel as of V4.5.
- The corresponding version of the SIMOTION Kernel must be installed on the target device.
- The ST source file must be compiled with the compiler option **Permit object-oriented programming**, see Global compiler settings (Page 63) and Local compiler settings (Page 67).

Note

When using the compiler option **Permit object-oriented programming** there is an expanded list of protected identifiers (Page 99) and of reserved identifiers (Page 105)in the ST programming language.

The aforementioned conditions apply to the entire section **Object-Oriented Programming – OOP** in this Manual.

Further information and usage examples of object-oriented programming can be found in the following references:

Braun, Michael / Horn, Wolfgang: Objektorientiertes Programmieren mit SIMOTION (Object-Oriented Programming with SIMOTION) Grundlagen, Programmbeispiele und Softwarekonzepte nach IEC 61131-3 (Basic Principles, Example Programs and Software Concepts according to IEC61131-3) 1st edition, Erlangen: Publicis Publishing 2016. ISBN 978-3-89578-455-2.

6.2 Classes and methods

6.2.1 Creating classes (basic classes)

6.2.1.1 Defining classes (basic classes)

You define a class in the declaration part of the implementation section wherever possible before the section of the source file (program, FB, FC or class) in which it is called.

Use the following syntax:

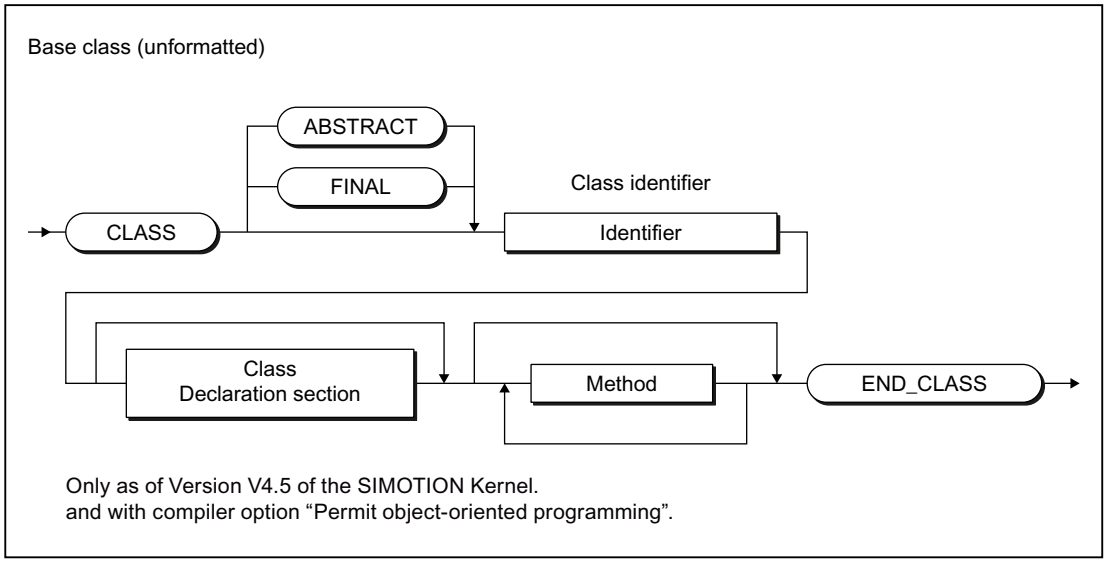


Figure 6-1 Syntax: Base class

Enter an identifier as the class name after the keyword CLASS and optionally FINAL or ABSTRACT.

Then enter, see example in Table 6-7 Example: Up and Down counter as a class (Page 225):

- The optional declaration section (Page 214)
- The methods
- Keyword END_CLASS

6.2.1.2 Declaration subsection of a class

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 and local variables. Each block type may occur multiple times, any sequence is possible for the blocks.

Permissible declaration blocks

Table 6-1 Declaration blocks of a class: Options

Data	Syntax
Data type	**TYPE** *Accessidentifier_class* (optional) *Declaration list* **END_TYPE**
Constant	**VAR CONSTANT** *Accessidentifier_class* (optional) *Declaration list* **END_VAR**
Local variable (static)	**VAR** *Accessidentifier_class* (optional) **OVERRIDE** (optional) *Declaration list* **END_VAR**
Retentive variable	**VAR RETAIN** *access identifier* (optional) **OVERRIDE** (optional) *Declaration list* **END_VAR** (Only PROTECTED or PRIVATE access identifier permitted, default PROTECTED.)
Accessidentifier_class: **PUBLIC**, **PROTECTED**, **PRIVATE** or **INTERNAL**, default **PROTECTED**; see section "Access identifiers within classes" (Page 215). **OVERRIDE**: Specifying this keyword enable an override of the initialization values for values with the access identifier PROTECTED or PRIVATE for the instance declaration (Page 226). *Declaration list*: The list of identifiers of the type to be declared	

6.2.1.3 Access identifier within classes

An access identifier may be stated within classes with declaration blocks or methods:

- With declaration blocks:
  Directly after the keyword for the relevant block (e.g. TYPE, VAR).
  The access identifier is valid for all identifiers within this declaration block.
- With methods:
  Directly after the keyword METHOD.
  The access identifier is valid for the relevant method.

You use the access identifier to determine whether the identifier or the method can only be used within the class, in its derivations or outside of the class, see following table.

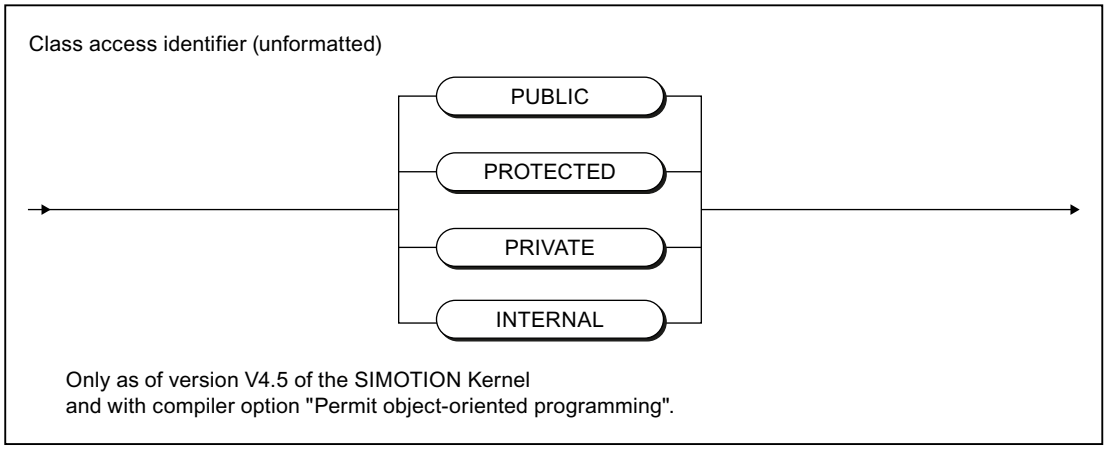


Figure 6-2 Syntax: Class access identifier

Table 6-2 Possible access identifiers within a class

Access identifier	Meaning
PRIVATE	The identifiers for the relevant declaration block (variables, data types, etc.) or the method can only be used within this class. These identifiers or the method cannot be used in derived classes.
PROTECTED	The identifier for the relevant declaration block (variables, data types, etc.) or the method can be used within this class and in classes derived from this. This is the default if there is no access identifier stated.
INTERNAL	Only when the identifier is defined within an object-oriented namespace (Page 355). The identifiers of the relevant declaration block (variables, data types, etc.) or the method can be used only within the higher-level namespace. Within the higher-level namespace, the following apply: • The declared identifiers can also be accessed outside of the class. This does not apply to retentive local variables (VAR RETAIN). • The method can also be called outside of the class.
PUBLIC	The identifiers for the relevant declaration block (variables, data types, etc.) or the method are public. • The declared identifiers can also be accessed outside the class and outside the higher-level object-oriented namespace. • The method can also be called outside the class and outside the higher-level object-oriented namespace. This access identifier is not permitted in declaration blocks for retentive local variables (VAR RETAIN / END_VAR).

6.2.2 Creating methods

6.2.2.1 Defining methods

The executable instructions of a class are summarized in methods. Within a class the methods are defined after the declaration subsection for the class.

A method roughly corresponds to a function (Page 181). Specification of an identifier determines the environment from which the method can be called.

If the compiler option (Page 63) “Allow forward declaration" is not activated a method must be defined before the program organization unit (program, FB, FC or class) or method in which it is called.

Use the following syntax:

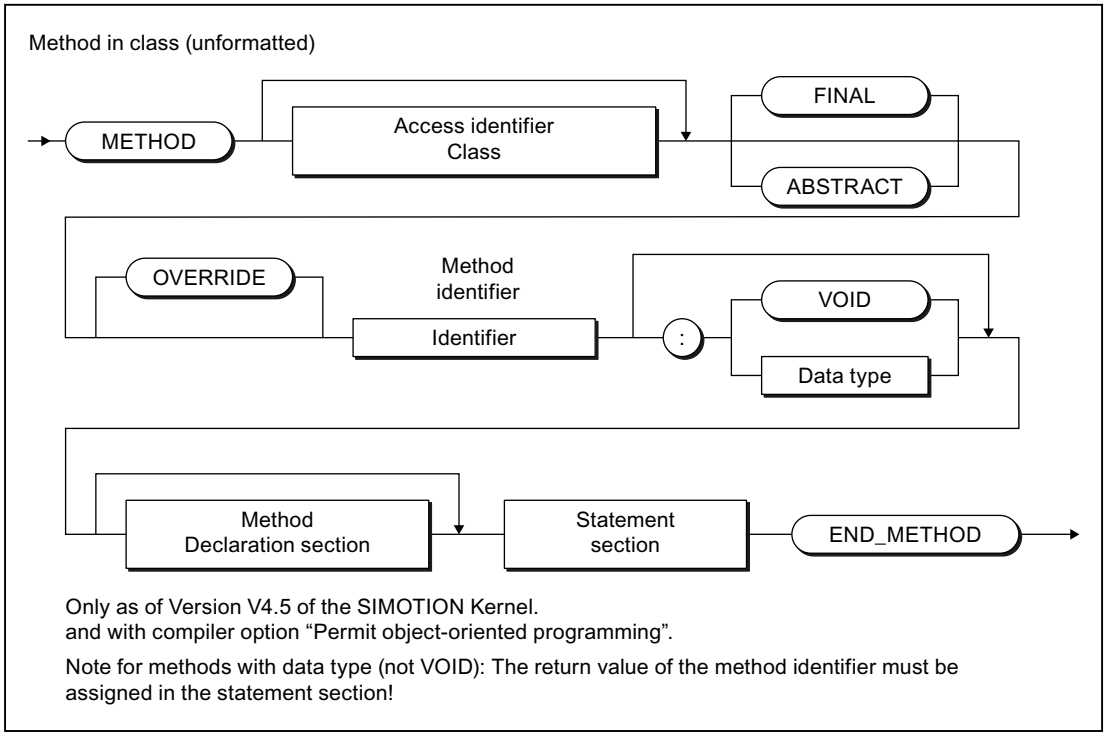


Figure 6-3 Syntax: Method

Enter in the following sequence, see Table 6-7 Example: Up and Down counter as a class (Page 225):

- The METHOD keyword.
- the optional class access identifier (PUBLIC, PROTECTED or PRIVATE), default PROTECTED. See section “Access identifiers within Classes" (Page 215) for access identifiers.

- The keywords FINAL, ABSTRACT and OVERRIDE are only relevant for methods in classes that are derived or are to be derived:
  - FINAL: Method cannot be overridden in a derived class.
  - ABSTRACT: Method cannot be called, it must be overridden in a class to be derived. Furthermore, this keyword renders the class in which the method is defined an abstract class. In this case, it is not possible to declare an instance of this class, see "Abstract classes" (Page 237).
  - OVERRIDE: Method overrides the existing method with the same name in a derived class.
- An identifier as method name.
- if the method has a return value, the data type for the return value following a colon. The symbolic data type VOID can also be entered for a method with no return value.

This is then followed by:

- The optional declaration section (Page 218)
- The statement section (Page 220) (except in methods with keyword ABSTRACT)
- The END_METHOD keyword

6.2.2.2 Declaration subsection of methods

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 a method:

Permissible declaration blocks

Table 6-3 Declaration blocks for methods: Options

Data	Syntax
Constant	**VAR CONSTANT** *Declaration list* **END_VAR**
Input parameters	**VAR_INPUT** *Declaration list* **END_VAR**
In/out parameter	**VAR_IN_OUT** *Declaration list* **END_VAR**
Output parameters	**VAR_OUTPUT** *Declaration list* **END_VAR**

Data	Syntax
Local variable (temporary)	**VAR** or **VAR_TEMP** *Declaration list* **END_VAR**
Declaration list: The list of identifiers of the type to be declared	

Parameter blocks

Parameters are local data and are formal parameters of a method. When the method 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 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 a method. This is identical to the syntax for the parameter declaration of a function (FC).

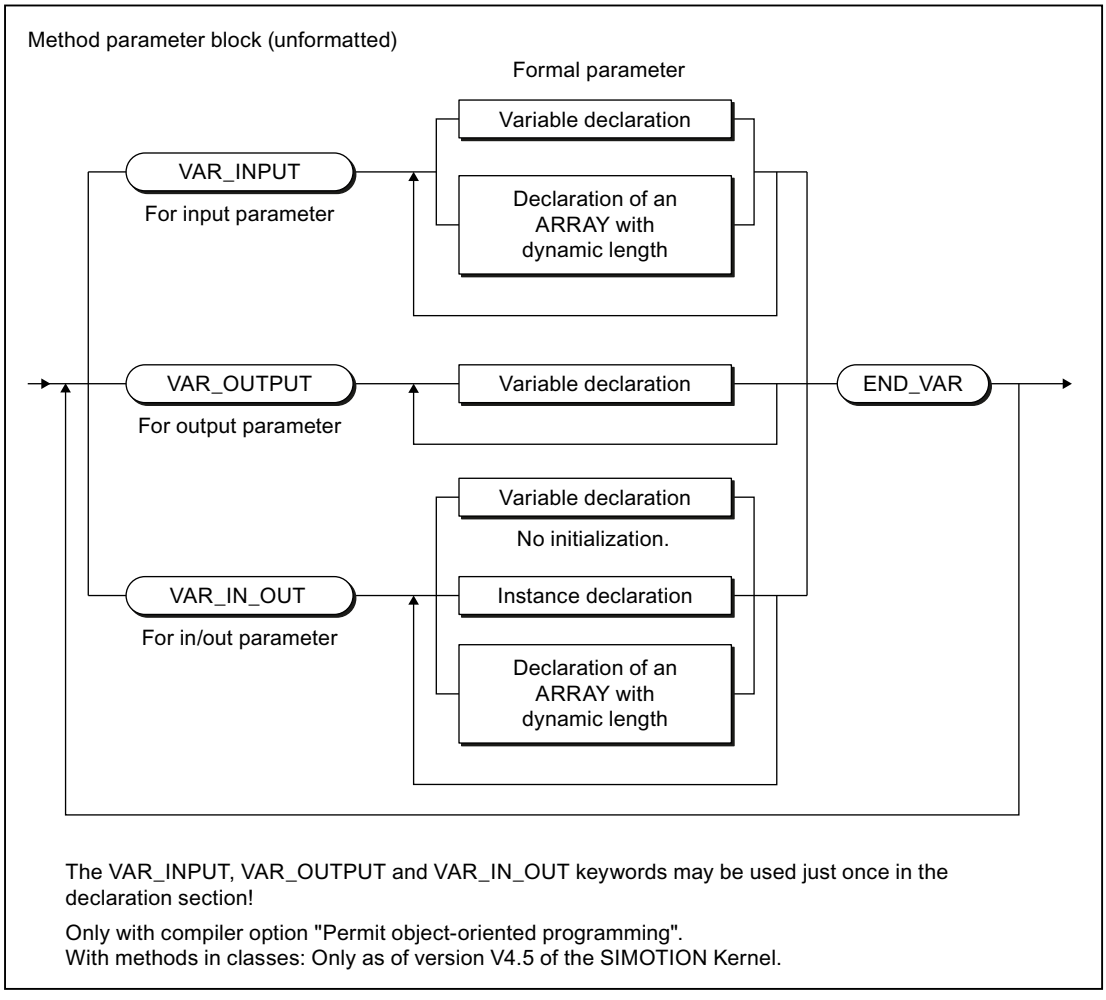


Figure 6-4 Syntax: Method parameter block

Apart from variables you can also declare the following in the parameter blocks:

- in the parameter block for in/out parameters:
  - Instances of function blocks (Page 199)
  - Instances of classes (Page 226)
  - Arrays with a dynamic length (Page 192), as of version V4.2 of SIMOTION Kernel.
- in the parameter block for input parameters:
  - Arrays with a dynamic length (Page 192), as of version V4.2 of SIMOTION Kernel.

You can use the declared parameters in the same way as other variables within a method, with the following exception: You cannot assign values to input parameters.

From outside the method you can only access the parameters by calling the method (including return value).

6.2.2.3 Statement section of methods

The statement section of a method contains statements that are executed when the method 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 the Runtime-optimized Programming section in the SIMOTION Basic Functions Function Manual.

Table 6-4 Use of parameters and variables in methods

Parameter/variable	Use
Input parameters	With the call of a method, assign the current values to the input parameters. These values are used for data processing within the method, for example, for calculations, but cannot be modified themselves.
In/out parameter	You assign a variable to an in/out parameter for the call of the method. The method accesses this variable directly and can change it directly. Type conversions are not supported. The variable assigned to an in/out parameter must be able to be read and written directly. Therefore, system variables (of the SIMOTION device or a technology object), I/O variables or process image accesses cannot be assigned to an in/out parameter.
Output parameters	You assign a variable to an in/out parameter for the call of a method using the **=>** operator. The value of the output parameter (result) is transferred to the variables when the method is closed.
Local variables	Local variables are variables that are declared and used only within the block. All local variables (VAR ... END_VAR or VAR_TEMP ... END_VAR) are temporary in methods, i.e. they lose their value when the method is terminated. The next time the method is called, they are reinitialized.

Note

The statement section is omitted from methods that are declared with the keyword ABSTRACT.

6.2.3 Calling methods within the class

6.2.3.1 Calling methods

Methods can be called in other methods within a class. As with functions, the call is made by specifying the method identifier and the parameter list (FC parameters).

The method can also be specified more precisely by stating the keywords THIS or SUPER or by stating a class identifier, see syntax in the following figure as well as the following table.

- The following applies to methods with no return value or with data type VOID:
  The method call is an instruction within a different method.
- The following applies to methods with return values:
  The method call is part of an expression on the right side of a value assignment.

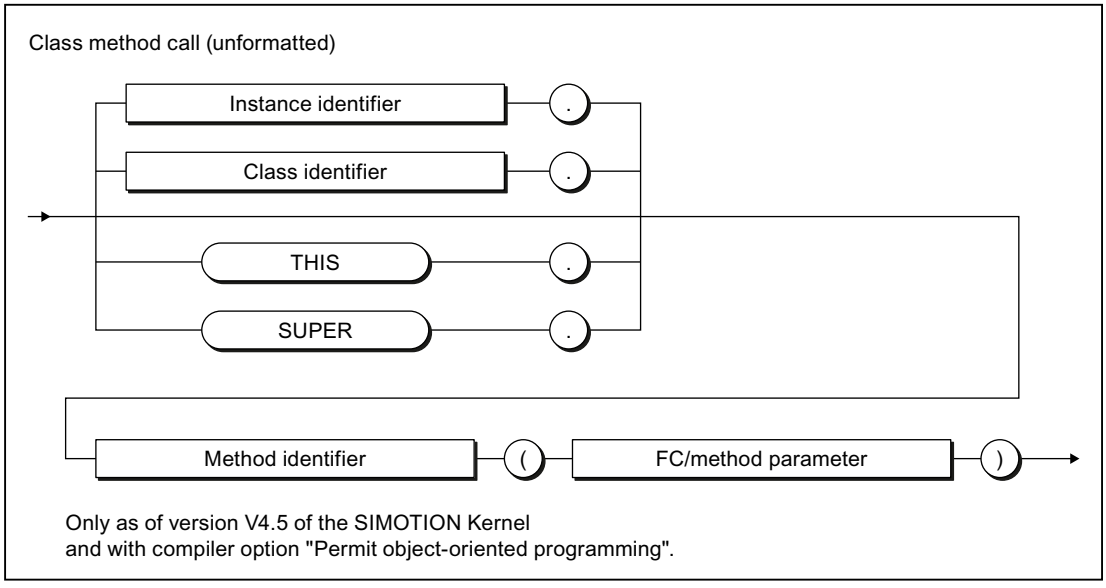


Figure 6-5 Syntax: Class method call

Table 6-5 Description of the method specification

Method specification	Description
(without)	Call of a method in a separate class with static binding. This means: If the stated method is overridden and called in a derived class, the original method is still called. Only methods that are defined in a separate class can be specified (i.e. not methods inherited from the base class).
THIS.	Call of a method in a separate class with dynamic binding. This means: If the stated method in a derived class is overridden and called, the current (overriding) method is called. All methods that are recognized in the class, including methods inherited from the base class, can be specified.
SUPER.	With derived classes: Explicit call of a method of the base class (static binding).
Class identifier.	Explicit call of a method of the stated class (static binding). Permitted class identifiers: Separate class and all base classes within a method implementation
Instance identifier.	Call of a method outside of the class, see Calling public methods of a class outside of this class (Page 227).

6.2.3.2 Parameter transfer to input parameters

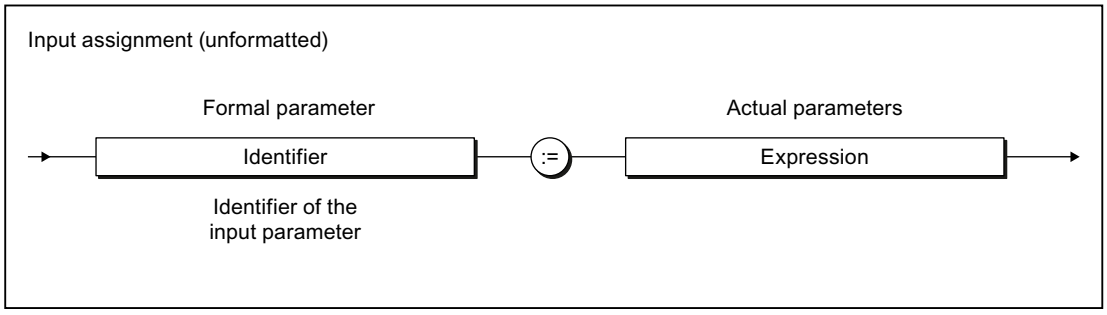


Figure 6-6 Syntax: Input assignment

You transfer the data (actual parameters) to the formal input parameters of a method 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 method, but you cannot modify their values.

The assignment of an actual parameter is optional when an initialization expression was specified for the declaration of the formal parameter.

See also Table 6-7 Example: Up and Down counter as a class (Page 225).

6.2.3.3 Parameter transfer to in/out parameters

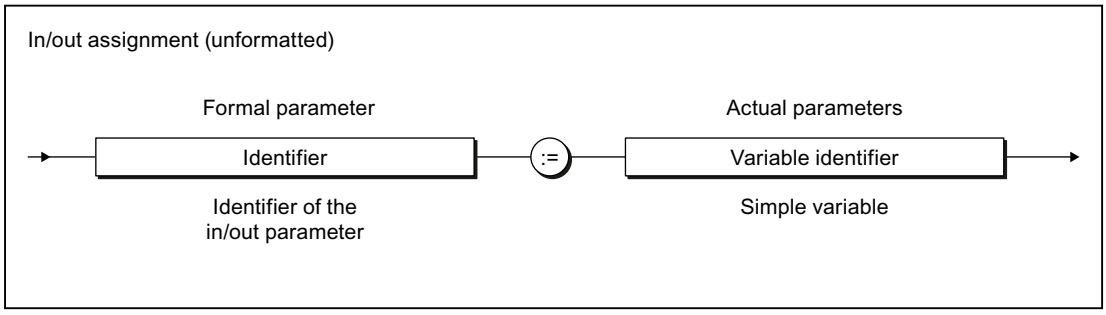


Figure 6-7 Syntax: In/out assignment

You transfer the data (actual parameters) to the formal in/out parameters of a method by means of 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 method. The method accesses the variable of the actual parameter directly and can change it directly.

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.

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 6-6 Example of the use of the STRING data type in in/out assignments

```
METHOD REF_STRING
    VAR_IN_OUT
        io : STRING[80];
    END_VAR
    ;    // Statements
END_METHOD

METHOD test
    VAR
        str1 : STRING [100];
        str2 : STRING [50] ;
    END_VAR
    REF_STRING (io := str1);    // Permissible call
    REF_STRING (io := str2);    // Illegal call,
                                // Compiler error message
END_METHOD
```

Please note the different parameter access times!

6.2.3.4 Parameter transfer to output parameters

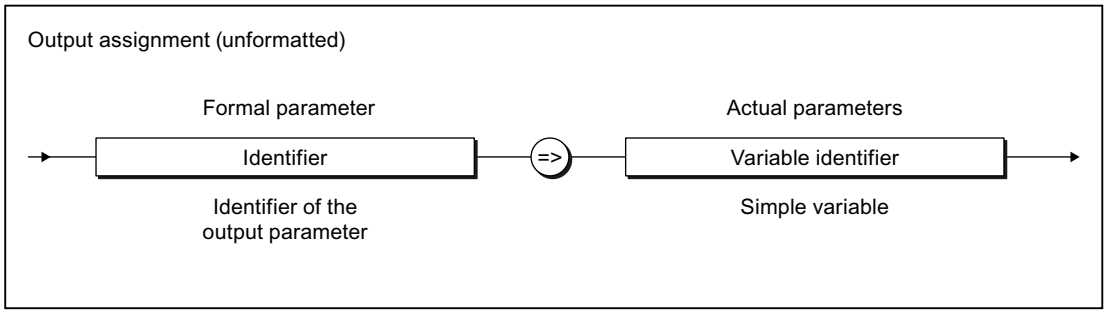


Figure 6-8 Syntax: Output assignment

You use an output assignment to assign the formal output parameters of a method to the variables (actual parameter) that adopt the value of the formal output parameter when the method is closed.

You can use and change the formal output parameters in statements within the method.

See also example: Table 6-8 Example: Call of the methods in the COUNTER class (Page 228)

Output assignments are optional for the parameter transfer. If there is no output assignment for an output parameter, the values for the output parameters are lost once the method is terminated.

6.2.4 Example classes and methods

6.2.4.1 Example: Counter as a class

The classes and methods are explained in the following example using a counter.

In the COUNTER class the two methods UP and DOWN are defined for counting a variable CV up or down. They have the access identifier PUBLIC so that they can be called from outside, e.g. from a program.

The UP method increases the CV counter variable with every run-through by the input parameter INC (default = 1) until the upper limit value MAX_Val is reached.

The DOWN method decreases the CV counter variable with every run-through by the input parameter DEC (default = 1) until the lower limit value MIN_Val is reached. It calls the UP method internally and transfers the negative value of the input parameter DEC to it. Adding THIS. before the call for the UP method means that if the UP method is exceeded in a derived class the UP method effective there is called.

The declaration of the variables MAX_Val and MIN_Val with the keyword OVERRIDE allows the initialization values for the instance declaration of the class to be changed.

The return value for both methods contains the current value of the counter variables CV. The output parameter QU signals whether the upper limit value MAX_Val has been reached or the lower limit value MIN_Val has not been reached as relevant.

Table 6-7 Example: Up and Down counter as a class

```
CLASS COUNTER
    VAR
        CV : INT; // Current value of the counter
    END_VAR
    VAR OVERRIDE
        MAX_Val : INT := 100;
        MIN_Val : INT := 0;
    END_VAR

    // Method for incrementing by INC
    METHOD PUBLIC UP: INT
        VAR_INPUT
            INC : INT := 1;
        END_VAR
        VAR_OUTPUT
            QU : BOOL;
        END_VAR
        // Detection of upper limit
        IF CV <= MAX_Val - INC THEN
            CV := CV + INC; // Increment of the counter CV
            QU := FALSE;
        ELSE
            QU := TRUE; // Upper level limit reached
        END_IF;
        UP := CV; // Result of the method
    END_METHOD
    // Method for decrementing by DEC
    METHOD PUBLIC DOWN : INT
        VAR_INPUT
            DEC : INT := 1;
        END_VAR
        IF CV > MIN_Val THEN
            // Internal call for the UP method
            DOWN := THIS.UP (INC := - DEC);
        END_IF;
    END_METHOD
END_CLASS
```

6.2.5 Creating and using instances of classes

6.2.5.1 Declaring an instance of a class

Declaring an instance of a class

Before using methods or variables of a class you must declare an instance of the class. You declare a variable and enter the name of the class as the data type. You declare this instance:

- locally:
  within VAR / END_VAR in the declaration section of a program, a function block or a class
- globally:
  within VAR_GLOBAL / END_VAR in the interface section or implementation section of the ST source file
- as an in/out parameter:
  within VAR_IN_OUT / END_VAR in the declaration section of a function block, a function or a method

It is in all cases possible to declare the instance of a class in a program source at which the compiler option (Page 63) "Permit object-oriented programming" is **not** active.

Exception: The compiler option (Page 63) "Permit object-oriented programming" **must** be active at the ST source file in the following cases:

- With local declaration within classes.
  The access identifier class (Page 215) (PUBLIC, PROTECTED or PRIVATE) can also be stated as an option after the keyword VAR.
- With local declaration within function blocks as long as the access identifier FB (PUBLIC or PRIVATE) is stated after the keyword VAR.
- With declaration as an in/out parameter in methods.

Instance-specific initialization of individual local variables of the class is possible. You thereby override the initialization values that were stipulated with the declaration of the class. Specification of the variables to be initialized is similar to the process for initializing a structure, and involves specifying a structure initialization list (Page 141) in brackets.

You can in all cases initialize public variables of the class (i.e. variables with the PUBLIC access identifier) specific to the instance. In order to be able to initialize protected or private variables of the class (i.e. variables with PROTECTED or PRIVATE access identifiers), the keyword OVERRIDE must be stated at their declaration block after the access identifier.

The following generally applies: The class must be declared in the ST source file before an instance can be declared.

Exception: It is sufficient to define a POU prototype (Page 381) beforehand if initialization is not specified and when compiler option (Page 63) "Permit forward declarations" is activated.

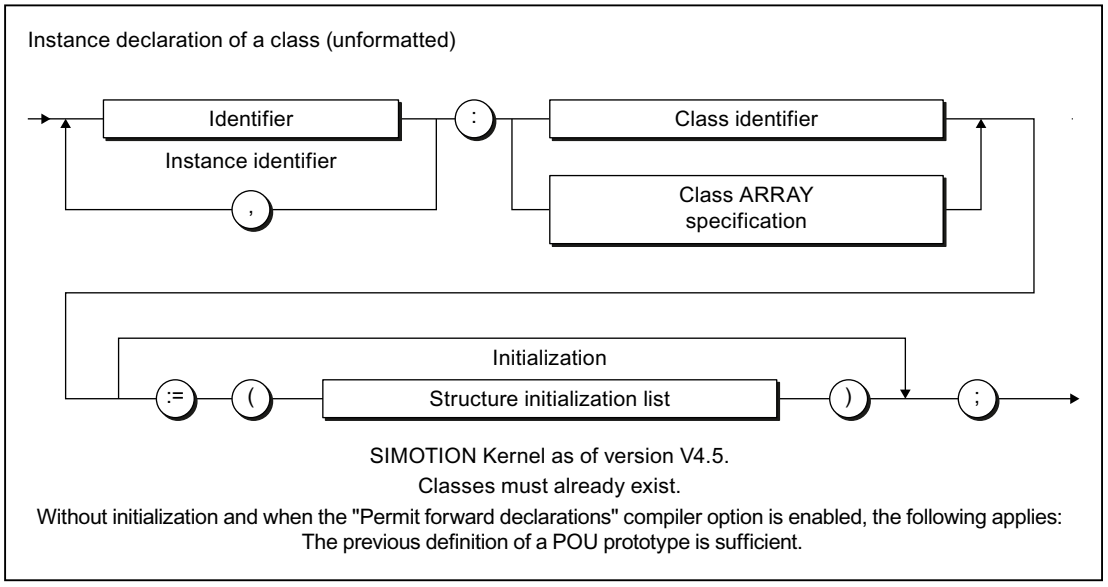


Figure 6-9 Syntax: Instance declaration of a class

Example of the instance declaration of a class:

```
Class_inst : Class_name := (Var_publ1 := 12,
                            Var_publ2 := 123.456);
```

The instance declaration can also be an array, e.g.:

```
Class_inst : ARRAY [1..2] OF Class_name;
```

Note

Pay attention to the different initialization times for different variable types.

6.2.5.2 Calling public methods of a class outside of this class

You can also call the public methods for a class (i.e. methods with PUBLIC access identifier) outside of this class, e.g. in programs, functions, function blocks or methods from other classes. Proceed as follows:

1. Declare an instance of the class (unless this has already been done).
   See Declaring an instance of a class (Page 226).
2. To call the method, write the name of the method instance and a period before the method name (similar to a structured variable (Page 151)).
   The parameter transfer takes place as described in the Calling methods within the class (Page 221) section.

6.2.5.3 Accessing the public variables of a class outside of the class

How to access a public local variable of a class (i.e. variable with PUBLIC access identifier):

1. Declare an instance of the class (unless this has already been done).
   See Declaring an instance of a class (Page 226).
2. For access purposes use a structured variable (Page 151) in the format *class-instancename.variablename*, e.g. *Class_inst.var_name*.

You can have read/write access to the public variable.

Access the public constants or data types of a class in a similar manner.

6.2.5.4 Example: Call of the counter method

The following program uses the COUNTER class declared in Table 6-7 Example: Up and Down counter as a class (Page 225). It calls the methods UP and DOWN defined there. A C1 instance of the COUNTER class is declared for this, with the initialization values of the variables MAX-Val and MinVal modified in this process. Provided that Locking = FALSE, method C1.UP is called. Provided that the upper value is reached (UpValreached = TRUE), Locking becomes TRUE and the method C1.DOWN is called until the lower value is reached.

This program must be assigned to a cyclic task of the execution system (e.g. BackgroundTask).

The complete ST source file includes the class COUNTER from Table 6-7 Example: Up and Down counter as a class (Page 225) in front of the program CallCounter.

Table 6-8 Example: Call of the methods in the COUNTER class

```
PROGRAM CallCounter_ST
    VAR
         C1 : COUNTER := (MAX_Val := 1000, MIN_Val := 0); // Instance
         CountOut     : INT;
         Locking      : BOOL;
         UpValreached : BOOL;
    END_VAR

    // Call of methods for incrementing and decrementing
    IF UpValreached = TRUE THEN
        Locking := TRUE;
    END_IF;
    IF Locking = FALSE THEN
        CountOut := C1.UP (QU => UpValreached); // Increment
    END_IF;
    IF Locking = TRUE THEN
        CountOut := C1.DOWN (); // Decrement
    END_IF;
    IF CountOut <= 0 THEN
        Locking      := FALSE;
        UpValreached := FALSE;
    END_IF;
END_PROGRAM
```

6.3 Inheritance of classes and methods

6.3.1 Inheritance of classes

In object-oriented programming, it is possible to derive subclasses from an existing class (base class). When a subclass is derived, it inherits all the properties of the base class, although the source code is not provided in the derived class:

- all data types that were declared in the base class,
- all variables and constants that were declared in the base class,
- all methods that were declared in the base class,
- all instance declarations of classes and function blocks within the base class.

The complete syntax diagram of a class definition is provided below.

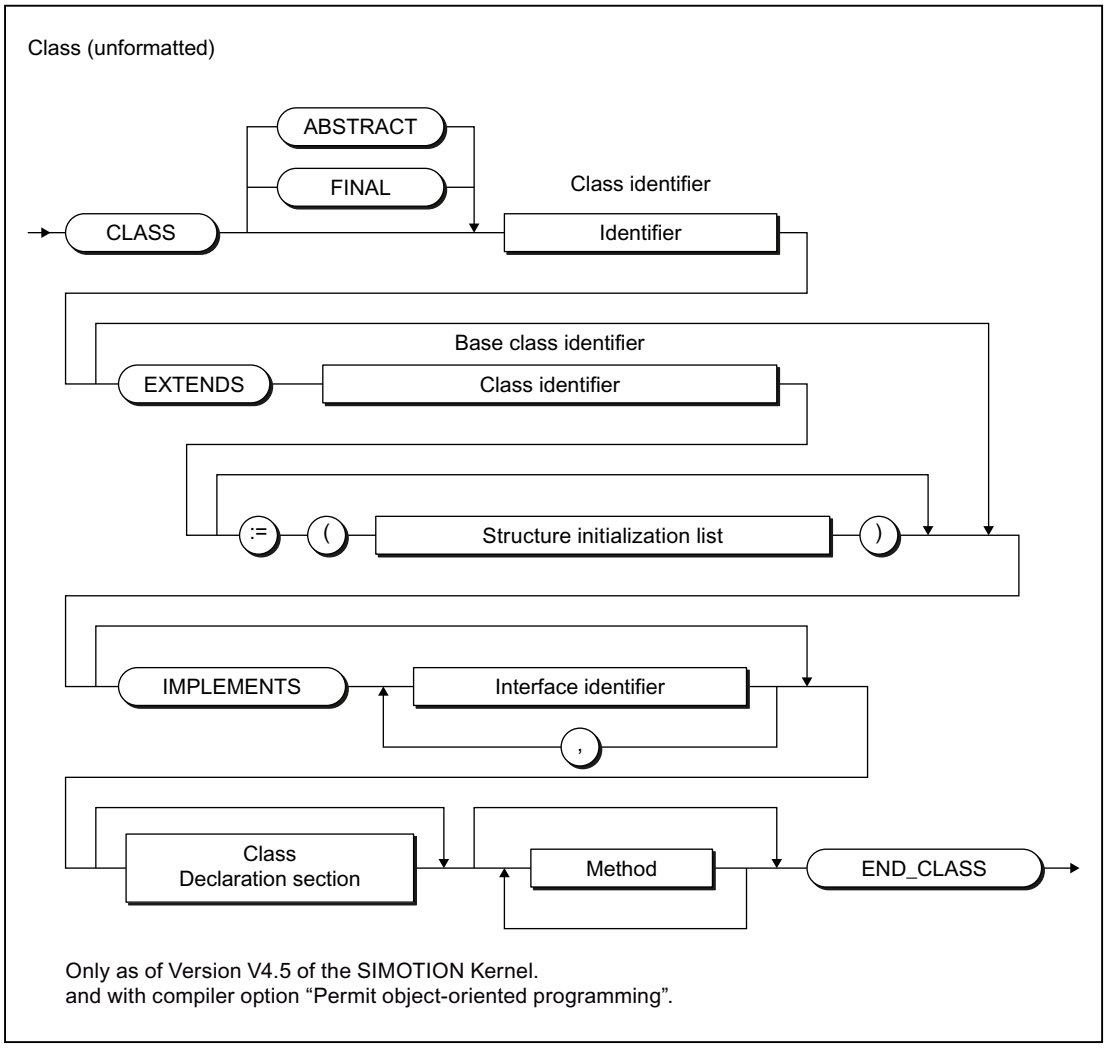


Figure 6-10 Syntax: Class

Use the keyword EXTENDS with the identifier for the base class in order to derive a class.

The properties for the derived class can now be adapted to the new circumstances:

- The initialization values for the variables from the base class can be modified:
  - public or protected variables (access identifiers PUBLIC or PROTECTED) without restrictions.
  - private variables (access identifier PRIVATE) only when the keyword OVERRIDE has been stated at their declaration block after the access identifier.
- Additional data types, variables and constants can be declared.
- Additional instances of classes and function blocks can be declared.
- Additional methods can be defined.
- Existing methods can be overridden.

Note

Data types, variables and constants that have been declared with the access identifier PRIVATE in the base class cannot be accessed in the derived class. Access is only possible via methods inherited from the base class.

Methods that have been defined with the access identifier PRIVATE in the base class cannot be called in the derived class. They can only be called from methods inherited from the base class.

6.3.2 Overriding methods

Public or protected methods of the base class (access identifiers PUBLIC or PROTECTED) can be overridden in a derived class.

For this proceed as described in the “Defining methods" (Page 217) section and use the keyword OVERRIDE.

The following must be consistent in the overriding method and the methods from the base class:

- Access identifier
- Method name
- Data type of the return value
- Parameters in the declaration subsection of the method:
  - Type (input, in/out, output parameters)
  - Number and sequence
  - Identifier
  - Data type

The overriding method and the method from the base class may be different:

- with the temporary variables (VAR / END_VAR or VAR_TEMP / /END_VAR)
- In the statement section
  You can also call the same method from the base class in the statement section. Use the keyword SUPER or the identifier for the base class. See “Calling methods" (Page 221).

The following methods from the base class cannot be overridden:

- Methods with the keyword FINAL.
- Methods with the access identifier PRIVATE.

The keyword ABSTRACT can only be used in the overriding method if it was also used in the method for the base class. Note that a class is abstract if it contains at least one abstract method. No instances can be declared from abstract classes.

6.3.3 Example: Counter in steps of 5 through inheritance

The following example is based on the COUNTER class from the Table 6-7 Example: Up and Down counter as a class (Page 225) example. It shows how to obtain a COUNTER_5STEP class whose methods count up and down in steps of 5 by deriving from the COUNTER class and overriding its methods.

With the definition of the COUNTER_5STEP class the keyword EXTENDS signifies that this class is derived from the COUNTER class (= base class). The derived class inherits from the base class all variables, constants, instance declarations and methods that have been defined with the PUBLIC or PROTECTED access identifiers. In the example, these are:

- the variables CV, MAX_Val and MIN_Val,
- the methods UP and DOWN

The UP method is now overridden so that it counts up in steps of 5. With the method definition the keyword OVERRIDE signals that the method for the base class is overridden. The overriding method and the method from the base class must be consistent on the following points:

- Access identifier (PUBLIC in the example)
- Method identifier (UP in the example)
- Data type of the return value (INT in the example)
- Parameters in the declaration subsection of the method (input parameter INC and output parameter QU in the example)

The statement section for the overriding method UP is entirely new. The UP method is called from the base class with the keyword SUPER and 5x of the input parameter IC of the overriding class is transferred to it as the input parameter. Similarly the return value and the output parameter QU of the base function are transferred to the overriding method. It may be confusing to see the same parameter identifiers in the input and output assignments on both sides. However, these are functions independent of the parameters. The formal

parameters of the method of the base class are on the left side of the assignments, the formal parameters of the overriding class are on the right side as actual parameter.

Table 6-9 Up and Down counter in steps of 5

```
CLASS COUNTER_5STEP EXTENDS COUNTER
    METHOD PUBLIC OVERRIDE UP: INT // Method override
        VAR_INPUT
            INC : INT := 1;
        END_VAR
        VAR_OUTPUT
            QU : BOOL;
        END_VAR

        UP := SUPER.UP (INC := 5 * INC, QU => QU);

    END_METHOD
END_CLASS
```

The DOWN method does not need to be overridden. Since the COUNTER_5STEP classes is derived from the COUNTER class then it also inherits the DOWN method. The DOWN method calls the UP method with the following statement: `DOWN := THIS.UP (INC := - DEC);`. The keyword THIS means that the method applicable there is called in derived classes. Consequently the DOWN method calls the following methods:

- in the COUNTER class:
  the UP method for counting upwards in steps of 1
- in the COUNTER_5STEP class:
  the UP method for counting upwards in steps of 5

This means that the DOWN method

- in the COUNTER class counts downwards in steps of 1
- in the COUNTER_5STEP class counts downwards in steps of 5.

6.3.4 Example: Call of the methods for both counters

Call in a program

The following program extends the Table 6-8 Example: Call of the methods in the COUNTER class (Page 228) program to include the call of the methods in the COUNTER_5STEP class. A C2 instance of the COUNTER_5STEP classes is declared for this: The statements for calling the UP and DOWN methods of the C1 instance of the COUNTER classes are duplicated and modified accordingly.

This program must be assigned to a cyclic task of the execution system (e.g. BackgroundTask).

The complete ST source file also includes the classes COUNTER and COUNTER_5_STEP from Table 6-9 Up and Down counter in steps of 5 (Page 232) and Table 6-7 Example: Up and Down counter as a class (Page 225) respectively in front of the program CallCounter_ST2.

Table 6-10 Call of the methods of the COUNTER and COUNTER_5STEP classes in stages

```
PROGRAM CallCounter_ST2
    VAR
        C1 : COUNTER;       // COUNTER instance
        C2 : COUNTER_5STEP; // COUNTER_5STEP instance
        CountOut      : INT;
        CountOut2     : INT;
        Locking       : BOOL;
        Locking2      : BOOL;
        UpValreached  : BOOL;
        UpValreached2 : BOOL;
    END_VAR
    // Call of methods from COUNTER (C1)
    IF UpValreached = TRUE THEN
        Locking := TRUE;
    END_IF;
    IF Locking = FALSE THEN
        CountOut := C1.UP (QU => UpValreached); // Increment
    END_IF;
    IF Locking = TRUE THEN
        CountOut := C1.DOWN (); // Decrement
    END_IF;
    IF CountOut <= 0 THEN
        Locking      := FALSE;
        UpValreached := FALSE;
    END_IF;
    // Call of methods from COUNTER_5STEP (C2)
    IF UpValreached2 = TRUE THEN
        Locking2 := TRUE;
    END_IF;
    IF Locking2 = FALSE THEN
        CountOut2 := C2.UP (QU => UpValreached2); // Increment
    END_IF;
    IF Locking2 = TRUE THEN
        CountOut2 := C2.DOWN (); // Decrement
    END_IF;
    IF CountOut2 <= 0 THEN
        Locking2      := FALSE;
        UpValreached2 := FALSE;
    END_IF;
END_PROGRAM
```

Call via function

The following example moves the same types of statements which call the methods of both classes in the above example to the CallSingleCounter function. The class and additional variables are transferred to the function via in/out parameters. Program maintenance is essentially simplified because the individual statements for calling the UP and DOWN methods do not need to be maintained twice.

The program CallCounter_ST3 must be assigned to a cyclic task of the execution system (e.g. BackgroundTask).

The complete ST source file also includes the classes COUNTER and COUNTER_5_STEP from Table 6-9 Up and Down counter in steps of 5 (Page 232) and Table 6-7 Example: Up and Down counter as a class (Page 225) respectively in front of the function CallSingleCounter.

Table 6-11 Call of the methods of the COUNTER and COUNTER_5STEP classes via a function

```
FUNCTION CallSingleCounter : VOID
    VAR_IN_OUT
        C : COUNTER;
        CountOut     : INT;
        Locking      : BOOL;
        UpValreached : BOOL;
    END_VAR
    // Call COUNTER ( C ) for increment/decrement
    IF UpValreached = TRUE THEN
        Locking := TRUE;
    END_IF;
    IF Locking = FALSE THEN
        CountOut := C.UP (QU => UpValreached); // increment
    END_IF;
    IF Locking = TRUE THEN
        CountOut := C.DOWN (); // decrement
    END_IF;
    IF CountOut <= 0 THEN
        Locking := FALSE;
    END_IF;
END_FUNCTION
```

```
PROGRAM CallCounter_ST3
    VAR
        C1 : COUNTER;
        C2 : COUNTER_5STEP;
        CountOut      : INT;
        CountOut2     : INT;
        Locking       : BOOL;
        Locking2      : BOOL;
        UpValreached  : BOOL;
        UpValreached2 : BOOL;
    END_VAR
    // Call COUNTER (C1) for increment/decrement
    CallSingleCounter (
        C := C1,
        CountOut     := CountOut,
        Locking      := Locking,
        UpValReached := UpValReached);
    // Call COUNTER_5STEP (C2) for increment/decrement
    CallSingleCounter (
        C := C2,
        CountOut     := CountOut2,
        Locking      := Locking2,
        UpValReached := UpValReached2);
END_PROGRAM
```

6.3.5 Initialization of derived classes and their instances

Initialization of derived classes

As stated in the syntax diagram Figure 6-10 Syntax: Class (Page 229) the initialization values of the following variables of the base class can be modified when deriving from a class:

- Public and protected variables (access identifiers PUBLIC or PROTECTED) without restrictions.
- Private variables (access identifier PRIVATE) only when the keyword OVERRIDE has been stated at their declaration block after the access identifier.

How to change the initialization values of the variables of a derived class:

1. Enter the assignment operator := after the name of the base class.
2. Next state the variables to be amended in brackets and their initialization values in the form of a structure initialization list (Page 141) (similar process as for the initialization with the user-defined data type structure).
   If the base class is itself a derived class, you can modify the initialization values of the variables inherited from its base class in the following way (as an alternative):
   - Enter the base class itself as an element of the structure initialization list or
   - Aggregate the identifiers of the base class and the variables in the form *class_name*.*var_name*.

   See also section "Initialization of structures" in chapter Initialization of variables or data types (Page 139) and the example below.

Example

The example below supplements the example Table 6-9 Up and Down counter in steps of 5 (Page 232).

Table 6-12 Derivation and initialization of the COUNTER class

```
CLASS COUNTER_5STEP EXTENDS COUNTER
                := (MAX_Val := 500)
    // MAX_Val = 500, MIN_Val = 0
    // ...
    // Override of method UP
END_CLASS

CLASS COUNTER_10STEP EXTENDS COUNTER_5STEP
                := (COUNTER := (MIN_Val := -500))
             // Alternative:
             // := (COUNTER.MIN_Val := -500)
    // MAX_Val = 500, MIN_Val = -500
    // ...
    // Override of method UP
END_CLASS
```

Initialization of instances of derived classes

As stated in the syntax diagram Figure 6-9 Syntax: Instance declaration of a class Instance declaration of a classSyntax (Page 227) the initialization values of the following variables can be modified with the instance declaration of a class:

- Public variables (access identifier PUBLIC) without restrictions.
- Protected and private variables (access identifier PROTECTED or PRIVATE) only when the keyword OVERRIDE has been stated at their declaration block after the access identifier.

How to change the initialization values of the variables of an instance:

1. Enter the assignment operator := after the name of the class.
2. Next state the variables to be amended in brackets and their initialization values in the form of a structure initialization list (Page 141) (similar process as for the initialization with the user-defined data type structure).
   If the class is itself a derived class, you can modify the initialization values of the variables inherited from its base class in the following way (as an alternative):
   - Enter the base class itself as an element of the structure initialization list.
   - Aggregate the identifiers of the base class and the variables in the form *class_name.var_name*.

   See also section "Initialization of structures" in chapter Initialization of variables or data types (Page 139) and the example below.

Example

The following example supplements the examples Table 6-8 Example: Call of the methods in the COUNTER class (Page 228) and Table 6-10 Call of the methods of the COUNTER and COUNTER_5STEP classes in stages (Page 233).

Table 6-13 Initialization of instances of the COUNTER class and the derivations

```
VAR
    C1 : COUNTER := (MIN_Val := -100);
         // MAX_Val = 100, MIN_Val = -100

    C2 : COUNTER_5STEP
            := (COUNTER := (MIN_Val := -100));
         // Alternative:
         // := (COUNTER.MIN_Val := -100);
         // MAX_Val = 500, MIN_Val = -100

    C3 : COUNTER_10STEP
            := (COUNTER_5STEP := (COUNTER := (MAX_Val := 1000)));
         // Alternative:
         // := (COUNTER_5STEP.COUNTER.MAX_Val := 1000);
         // MAX_Val = 1000, MIN_Val = -500
END_VAR
```

6.4 Abstract classes

Definition

A class is described as abstract if one of the two following conditions is met:

- The class is marked with the keyword ABSTRACT, see Class syntax diagram (Page 229).
- At least one method of the class is abstract, i.e. is marked with the keyword ABSTRACT, see Method syntax diagram (Page 217).

No instances can be declared of an abstract class.

In order to be able to declare instances, derived classes must be declared from abstract classes with all abstract methods overridden and their statement sections formulated. Derived classes in which not all inherited methods are implemented remain abstract.

Use

Abstract classes are a means of structuring and abstraction often used in object-oriented programming. The common variables and common methods are defined with their interfaces (formal parameters) in an abstract class. The methods and specified in the derived classes and overridden and implemented in accordance with the relevant requirements.

There are different motor types in a machine, e.g. motors that can be switched on directly via a contactor or via a star-delta switch and speed-controlled drives. The concrete methods for switching the motors on and off differ. Nevertheless the methods for switching on and off

in an abstract class can be stipulated with their signatures (identifier, access identifier, formal parameters) with variables also defined for the status of the motor (e.g. running, stationary, starting, stopping, actual speed, error). These methods and variables should be subject to uniform usage in all classes. In the derived classes the methods are specified in accordance with the motor type with the corresponding values assigned to the variables.

6.5 Object-oriented interface

6.5.1 Defining an object-oriented interface

Object-oriented interfaces define the call interface for public methods (access identifier PUBLIC). Method prototypes are defined for this purpose in the object-oriented interface. These methods with the definition "prototype" are formulated in classes which implement the object-oriented interfaces.

Object-oriented interfaces are defined in the interface section (Page 258) or implementation section (Page 260) of an ST source file. In most cases, they are declared in the interface section because they are generally declared PUBLIC.

They must always be declared in full before they are used in the declaration of an implementing class or an interface variable. This applies irrespective of the compiler option (Page 63) "Permit forward declarations".

Note

While object-oriented interfaces start and end with the same keywords as the interface section (Page 258) of a unit (INTERFACE / END_INTERFACE), they have different meanings and are used in different ways.

Syntax of the object-oriented interface

Use the following syntax:

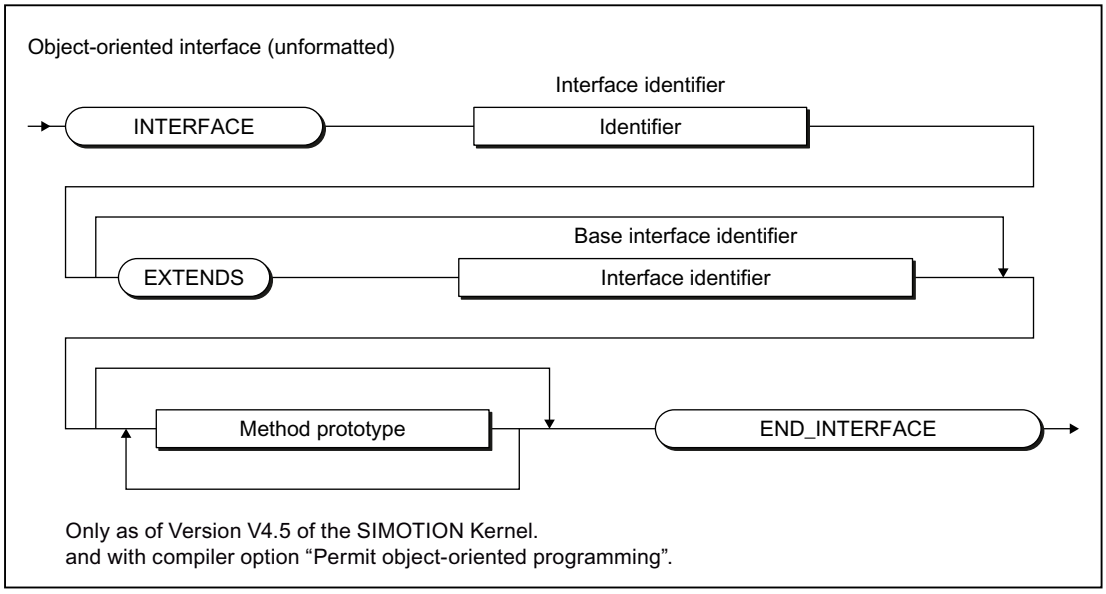


Figure 6-11 Syntax: Object-oriented interface

Enter an identifier as the interface name after the keyword INTERFACE.

This is then followed by:

- Optionally the keyword EXTENDS and the identifier of a different interface.
  This is how you derive an interface from an existing interface (base interface).
- The method prototypes in accordance with the following syntax
- The keyword END_INTERFACE

Method prototype

A method prototype determines the call interface for a method. This call interface includes:

- the method identifier (name),
- optionally the data type of the return value
- the input, in-out parameters and output parameters for the method

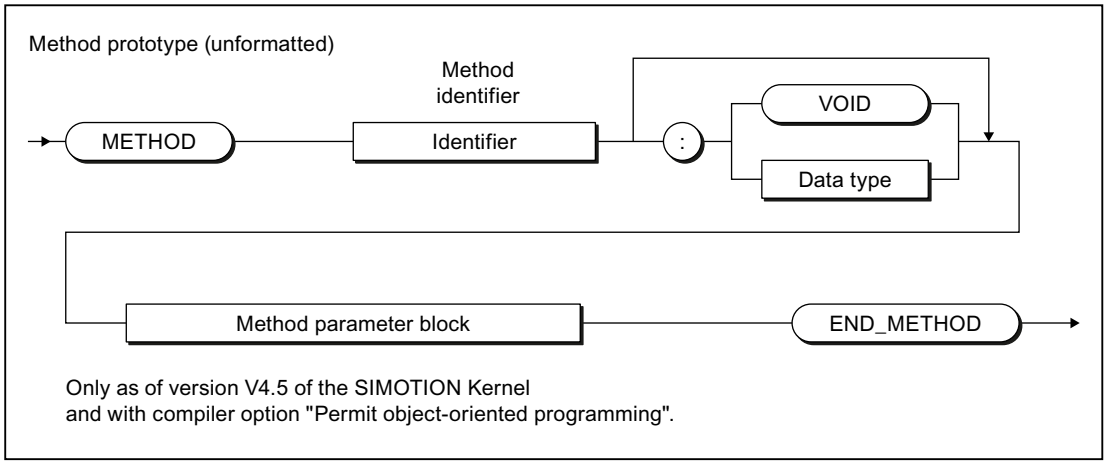


Figure 6-12 Syntax: Method prototype

These method prototypes must be formulated as public methods (access identifier PUBLIC) in the classes which implement the interface. Observe the following here:

- The following must still be adopted here:
  - the method identifier
  - the data type of the return value (if present)
  - the input, in/out and output parameters in the parameter block
- The following must or can be added:
  - Optional: Property of the method (FINAL, ABSTRACT)
  - Mandatory: Access identifier PUBLIC
  - Optional: Declaration of variables and data types
  - Mandatory: Statement section (except for methods with the property ABSTRACT: the statement section is omitted in this case).

Derivation of an object-oriented interface

With the keyword EXTENDS you can derive an object-oriented interface from an existing one. Only single derivations are possible, i.e. it is not possible to derive multiple interfaces.

Existing method prototypes cannot be changed or overridden in the derived object-oriented interface. Only method prototypes can be added.

6.5.2 Implementation of object-oriented interfaces in classes

Object-oriented interfaces are implemented in classes It is essential to declare the interfaces to be implemented before the implementing class in the ST source file.

The figure below repeats the syntax diagram for the class from the “Inheritance of classes" (Page 229) section.

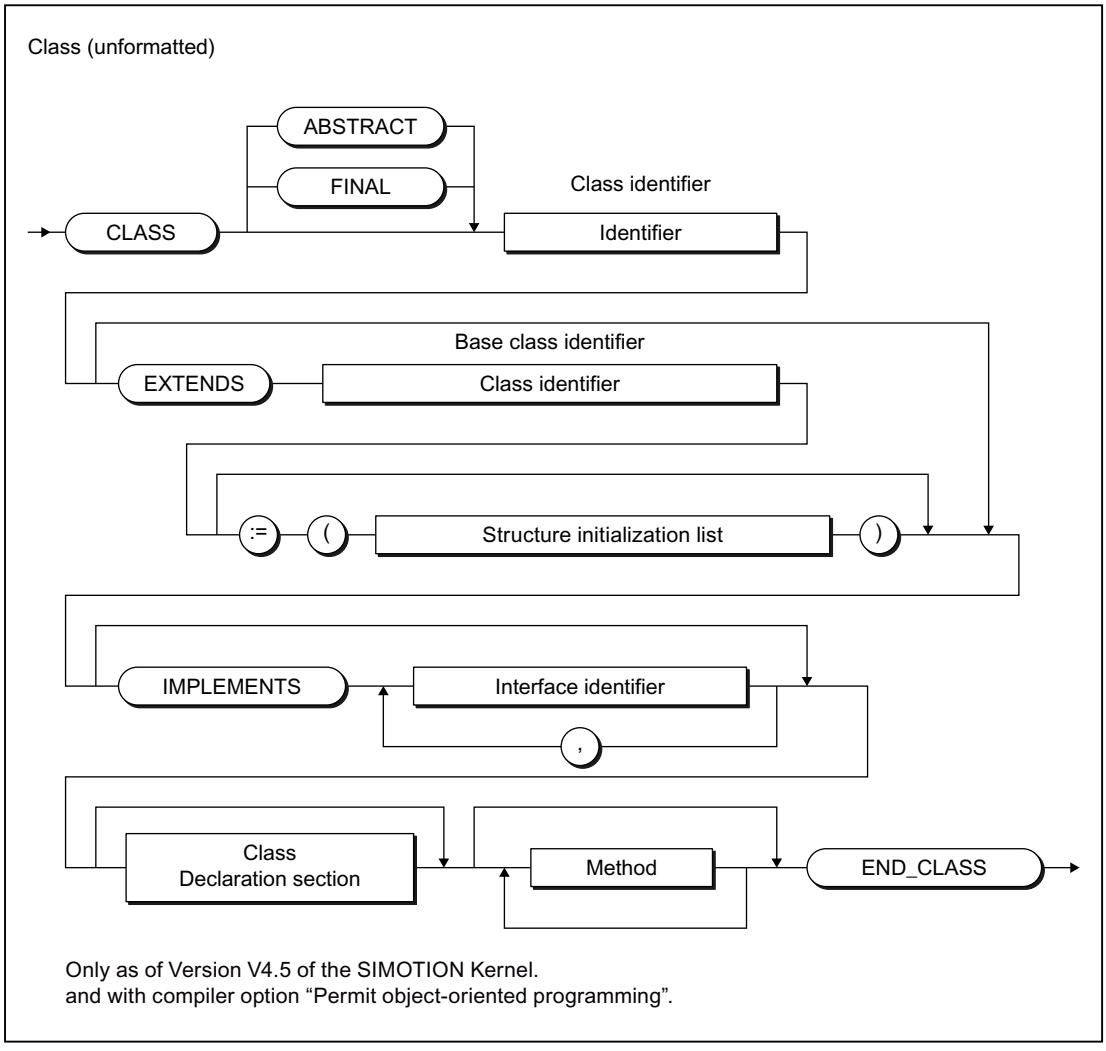


Figure 6-13 Syntax: Class

To implement one or more object-oriented interfaces use the keyword IMPLEMENTS, followed by the lists of the interfaces to be implemented separated by commas. A class can implement multiple interfaces.

The following must or can be supplemented for the definition of the implementing class:

- Optional: the class property (FINAL, ABSTRACT)
- Mandatory: the class identifier (name)
- Optional: the keyword EXTENDS with the identifier of a base class to be derived as well as the initialization (if applicable)
- Optional: the declaration subsection with the local variables and data type for the class

- Mandatory: all methods that have been defined as prototypes in the interfaces to be implemented.
  The following must still be adopted with this:
  - the method identifier
  - the data type of the return value (if present)
  - the input, in/out and output parameters in the parameter block

  The following must or can be added:
  - Optional: Property of the method (FINAL, ABSTRACT)
  - Mandatory: Access identifier PUBLIC
  - Optional: Declaration of local variables and data types
  - Mandatory: Statement section (except for methods with the property ABSTRACT: the statement section is omitted in this case).
- Optional: Additional methods (any access identifier possible).
  See: Creating methods (Page 217).

All method identifiers must be unique within the class. This means that the object-oriented interface to be implemented

Derived classes and derived object-oriented interfaces

An object-oriented interface (base interface) is implemented in a class (base class). A further object-oriented interface containing additional method prototypes is derived from the base interface. This derived object-oriented interface is now implemented in another class that is also derived from the base class.

This derived class then inherits all the methods of the base class including those that have been defined as prototypes in the base interface. The additional method prototypes of the derived object-oriented interface are then the only elements still to be formulated in the derived class.

6.5.3 Variables of object-oriented interfaces

You can declare variables of object-oriented interfaces in program organization units and classes (e.g. in programs, function blocks, functions). Enter the name of the interface as its data type. Interface variables are initialized with "NULL". Only "NULL" is permissible for explicit specification of the initialization.

All variable types for the relevant program organization unit are permitted with the exception of:

- VAR_IN_OUT.
  Arrays, structures, function blocks or classes which contain interface variables may nonetheless be transferred as VAR_IN_OUT.
- VAR RETAIN or VAR_GLOBAL RETAIN is not meaningful.
  The interface variables are non-retentive when saved.
- VAR CONSTANT or VAR_GLOBAL CONSTANT is not meaningful.
  The interface variables must be initialized with "NULL".

Using the interface variables

Interface variables contain references to classes which implement the corresponding object-oriented interface. They permit access to the methods of the interface with the following syntax: *var_name.method-name*.

Once an instance of an implementing class has been assigned to the interface variable with the assignment operator :=, the method of the referencing class is called with the call in accordance with the syntax stated above.

An invalid reference is marked with the value "NULL" as the content of the variable. A check for a valid reference is easy therefore with the sequence `IF var_name <> NULL THEN (* … *) ; END_IF`, see example below.

Dynamic type conversion with the operator "?="

the operator ?= allows a dynamic type conversion for interface variables. It takes place during the program execution time.

A dynamic type conversion is performed if the following two conditions are satisfied:

1. The interface variable to the right of the operator contains the reference to the instance of a class.
2. This class implements the object-oriented interface that is the data type of the interface variable to the left of the operator.

In this case, the reference on the right-hand side is converted and transferred to the left-hand side.

If the type conversion fails, the variable on the left-hand side contains the "NULL" value.

This dynamic type conversion of references makes it easy to determine whether classes have implemented specific interfaces. The methods defined in the relevant interface can then be called. It is not necessary to know how the methods have actually been programmed.

Forcing instance-specific initialization in classes and function blocks

When instances of classes and function blocks are set up, their static variables can be initialized specific to the instance.

Instance-specific initialization can be forced for the variables of object-oriented interfaces. To do this, specify " `:= *` " after the identifier of the object-oriented interface when declaring these variables in classes or function blocks. The relevant static variable block must

be assigned the OVERRIDE or PUBLIC identifier so that it can be initialized from an external source.

If a variable is declared in this way, the compiler checks whether all variables of this kind have been initialized at least with instances in VAR_GLOBAL or in the VAR declaration blocks of programs. In this case, initialization can take place even the instance was previously included in other classes or function blocks.

If a class or a function block possesses this kind of variable, the following needs to be noted when the instance is declared in the VAR blocks of other classes or function blocks:

- Either the instance itself is initialized,
- Or the variable block in which the instance is declared must be capable of initialization (OVERRIDE or PUBLIC identifier).

Initialization with NULL is permissible.

This feature can be used to define links between different instances via interfaces and to monitor by means of the compiler whether they have been meaningfully initialized.

Table 6-14 Example of forcing instance-specific initialization

```
INTERFACE IfTest
    // ...
END_INTERFACE
CLASS cl IMPLEMENTS IfTest
    // ...
END_CLASS
CLASS clTest
    VAR PROTECTED OVERRIDE
        myInterf : IfTest := *;
    END_VAR
END_CLASS
PROGRAM prog
    VAR
        clInst : cl;
        testInst : clTest := (myInterf := clInst);
    END_VAR
    ; // ...
END_PROGRAM
```

6.5.4 Example of interface variables

The example below illustrates the use of interface variables.

```
INTERFACE ITF1
    METHOD m1 END_METHOD
END_INTERFACE
INTERFACE ITF2
    METHOD m2 END_METHOD
END_INTERFACE
CLASS A IMPLEMENTS ITF1
    METHOD PUBLIC m1 (* ... *) ; END_METHOD
    // ...
END_CLASS
CLASS B IMPLEMENTS ITF2
    METHOD PUBLIC m2 (* ... *) ; END_METHOD
    // ...
END_CLASS
CLASS C IMPLEMENTS ITF1, ITF2
    METHOD PUBLIC m1 (* ... *) ; END_METHOD
    METHOD PUBLIC m2 (* ... *) ; END_METHOD
    // ...
END_CLASS
FUNCTION func_if1 : VOID
    VAR_INPUT i : ITF1; END_VAR
    IF i <> NULL THEN
        i.m1();
    END_IF;
END_FUNCTION
FUNCTION func_if2 : VOID
    VAR_INPUT i    : ITF2; END_VAR
    VAR       tmp  : ITF1; END_VAR
    IF i <> NULL THEN
        i.m2();
    END_IF;
    tmp ?= i;
    (* Dynamic type conversion to ITF1 *)
    IF tmp <> NULL THEN
        tmp.m1();
    END_IF;
END_FUNCTION
```

```
PROGRAM P
    VAR
        inst_a : A; // Instance of class A
        inst_b : B; // Instance of class B
        inst_c : C; // Instance of class C
        interf1: ITF1; // interf1 has value NULL
        interf2: ITF2; // interf2 has value NULL
    END_VAR

    interf1 := inst_a;
    (* interf1 contains a valid reference to inst_a. *)
    func_if1(interf1);
    (* The method inst_a.m1() is therefore called within the function. *)

    func_if1(inst_a);
   (* Equivalent call to the last two code lines *)

    interf2 := inst_b;
    (* interf2 contains a valid reference to inst_b. *)
    func_if2(interf2);
    (* The method inst_b.m2() is therefore called within the function.
       The method m1() is not called. The variable tmp contains the value NULL after
execution of the operator ?= because the class B does not implement the interface ITF1. *)

    interf2 := inst_c;
    (* interf2 contains a valid reference to inst_c. *)
    func_if2(interf2);
    (* The method inst_b.m2() is therefore called within the function.
      The method inst_c.m1() is also called. The variable tmp contains a valid reference to
inst_c after execution of the operator ?= because the class C also implements the interface
ITF1. *)
END_PROGRAM
```

Two object-oriented interfaces (ITF1 and ITF2) are defined in the example shown. Each interface contains one method as relevant: m1() in ITF1 and m2() in ITF2.

The following 3 classes implement both of these interfaces:

- Class A implements interface ITF1
- Class B implements interface ITF2
- Class C implements both interfaces ITF1 and ITF2.

For both functions func_if1 and func_if2:

- You have the option of adopting an interface variable via an input variable i with the relevant interface type.
- You can use the query " i<> NULL" to check whether i contains a valid reference.
  If the reference is valid, the functions call the methods of the relevant interfaces.
- Function func_if2 has the additional option of transferring (tmp ?= i) the reference of i to the tmp variable of type ITF1.
  - If i contains a reference that matches interface ITF1, the reference is transferred by the ?= operator.
  - If the reference does not match, a NULL is entered in tmp.

  If the reference in tmp is valid, method m1() is called.

The instances of classes A, B and C are generated and the interface variables interf1 and interf2 are declared in program P.

In the program statement section, the class instances are assigned to the relevant interface variables interf1 and interf2. Following each of these assignments one of the two functions is called with the corresponding interface variable also transferred. This results in different calls for the interface methods within the functions:

- First call of func_if1 (with transfer of the interface variables interf1):
  Since interf1 contains a valid reference to the instance of class A (inst_a); the function calls the method m1().
- Second call of func_if1 (with the transfer of instance inst_a):
  Since class A implements the interface ITF1, this call is functionally identical to the first call.
- First call of func_if2 (with transfer of the interface variables interf2):
  Since interf2 contains a valid reference to the instance of class B (inst_b); the function calls the method m2().
  The method m1() is not called, since the class B only implements the interface ITF2.
- Second call of func_if2 (with transfer of the interface variables interf2):
  Since interf2 contains a valid reference to the instance of class C (inst_c); the function calls both methods m1() and m2().
  Class C implements both interfaces (ITF1 and ITF2). The operator ?= therefore transfers the reference to inst_c successfully to the variable tmp of data type ITF1. The function therefore also calls the method m1().

This example demonstrates perfectly how interface variables with corresponding check functions can be handled and the degree of programming flexibility offered by this approach.

Note

This mechanism of course works in exactly the same way in methods, But we have used functions here instead of methods because the example was simpler.

This option of forming object references and interface variables makes it possible to independently develop and test program modules. Interfaces are a description of the interfaces shared by the program modules containing the program code of the methods to be implemented. This data description can then be used, for example, to perform a test in a program area with dummy methods and data supplied by reference transfer. When

the formulated program sections are created later on, the tested sections will function automatically. Of course, this arrangement will work successfully only if the interfaces are not changed retrospectively. But the compiler monitors interfaces carefully for any changes.

6.6 Comparison between abstract class and object-oriented interface

If we compare the definition of an abstract class (Page 237) and compare it to the definition of an object-oriented interface (Page 238), we will find that both constructs are very similar. Both constructs contain prototype methods for which the appropriate functions must be programmed in a class.

The programmer must now decide which construct to use for the specific task in hand. The following information should provide a useful decision-making guide.

The table below compares the properties of the abstract class with the properties of the object-oriented interface.

Table 6-15 Comparison between abstract class and object-oriented interface

Abstract class	Object-oriented interface
Abstract classes can contain abstract methods as well as fully programmed methods, i.e. real methods.	An interface may contain only prototype methods (keyword ABSTRACT does not need to be specified).
Abstract classes can possess defined properties (attributes).	An interface cannot possess any properties (attributes).
The properties of methods can be selected (PUBLIC, PRIVATE, PROTECTED).	All methods in the interface are PUBLIC and cannot be changed retrospectively.
An abstract class cannot be instantiated.	Interface variables can be created.
The methods of an abstract class must be fully programmed when they are derived unless the derivation itself is also ABSTRACT.	All the methods of an interface must be programmed in the class in which the method is implemented, or identified as ABSTRACT
A class can be derived from an (abstract) class.	An interface can be derived from a base interface.
An (abstract) class can implement multiple interfaces.	-

Both constructs define methods as prototypes. This guarantees that the methods can be used according to their definition in the derived or implementing classes. A programmer can thus feel confident that the use (call) of methods defined as prototypes will function reliably when the entire program is finished and the individual program sections are joined together.

The basic difference between the two constructs can be summarized as follows:

- An abstract class constitutes a contract for use in the derivation chain, i.e. within the class hierarchy.
- Object-oriented interfaces constitute an external contract between different program sections.

The following decision-making guide can be derived from the above information:

- Object-oriented interfaces are used if the following two conditions are satisfied:
  - A more generalized definition between different programs is required.
  - The methods should or may be accessed by anyone.
- Abstract classes are used if one of the following conditions is satisfied:
  - A structure must be defined for a program area..
  - Methods must be concealed (i.e. must not be PUBLIC).

Abstract classes can be combined very effectively with object-oriented interfaces in object-oriented software and thus provide a wonderful tool for planning and designing the software structure.

The benefits of abstract classes are:

- Abstract classes allow full programming of methods. thereby reducing the time and effort involved in writing program sections that can be used by everyone in the same way. The methods are simply inherited when real classes are derived.
- The specification of access rights to methods enhances the security of programs.

The benefits of object-oriented interfaces are:

- Object-oriented interfaces allow different program sections to be connected in a highly flexible manner,
- so making it possible for software development teams to work more independently of one another.

6.7 General references

6.7.1 Defining general references

General references contain the address assignment to a variable or an instance of a function block or class. To do this, the REF_TO keyword creates a data type that can contain the address for an existing data type (reference data type).

- In a data type declaration:
```
TYPE
    ref_type : REF_TO data_type
END_TYPE
```
The declaration is also possible in an ARRAY or as element of a structure.
- In a variable declaration:
A reference already declared as data type or the REF_TO statement can be used directly.
```
VAR
    ref_var : ref_type // or
    ref_var := REF_TO data_type
END_VAR
```
The declaration is also possible in an ARRAY.

The associated declared data type is "Reference to the data type" (REF_TO *data_type*).

The following are permitted as data types to which references can be formed:

- Elementary data types (e.g. INT, DINT, REAL, WORD, TIME, STRING)
- User-defined data types (UDT)
- System data types
- Function blocks, provided they contain at least one static variable
- Classes

No references can be formed to the following data types because references exist already:

- Technology object data types
- Object-oriented interfaces
- General references
- I/O references

Variables of general references can be declared in the interface and implementation sections as well as in all program organization units and classes. The data type to be referenced must be declared beforehand and lie within the scope of the POU. The declaration as element of a structure or an ARRAY is also possible.

All variable types for the relevant program organization unit are permitted with the exception of:

- VAR_IN_OUT.
  Arrays, structures, function blocks or classes that contain general references may nonetheless be transferred as VAR_IN_OUT.
- VAR RETAIN or VAR_GLOBAL RETAIN is not permitted.
  However, structures that contain at least one element other than reference types (general references, data types of the technology objects, object-oriented interfaces), can be stored retentive.
- VAR CONSTANT or VAR_GLOBAL CONSTANT is not meaningful.
  General references must be initialized with "NULL".

6.7.2 Forming general references

A reference to a variable is formed with the REF standard function (in := *var_name*). This function is described in detail in the "SIMOTION Basic Functions" Function Manual.

The REF function can be used on the following variables:

- Retentive and non-retentive unit variables or instances of classes or function blocks or their elements declared as such.
- In methods within classes or function blocks:
  - Static variables (VAR) of the higher-level class or function block (including instance variables for classes or function blocks).
  - The THIS keyword. Enables access to the instance variable of the higher-level class or function block.
- In function blocks:
  - Static variables (VAR) of the function block (including instance variables of classes or function blocks).
  - The THIS keyword. Permits access to the instance variable of the function block.

The variable must be able to be read and written.

Not permitted are:

- Variables of technology object data types
- Variables of object-oriented interfaces
- Variables of general references
- Retentive local variables within classes or function blocks
- I/O variables
- Global device variables
- Constants

The return value has the data type "Reference to the data type of the input parameter *in*". The return value can be assigned with the assignment operator ":=" to a variable that has REF_TO *type* data type. A data type conversion is not possible.

Note

Note for instances of function blocks that are passed as in/out parameter (VAR_IN_OUT):

- The REF function cannot be used for static variables (VAR) of function blocks imported from technology packages or device-independent libraries.

The restriction does not apply to class instances.

6.7.3 Operations with general references

Assignment operator ":="

For the assignment of a reference variable, the address of the variable rather than the variable value is passed. Consequently, the reference data types must be identical on both sides of the operator. No implicit data type conversion is possible.

The only exception is for references to classes. A reference to a derived class of a reference to the base class can be assigned here.

For the assignment, no check is made for the validity of the reference; the assignment is always possible.

An explicit assignment with NULL is also possible.

The validity of references should be checked by comparing with NULL before use.

Dereferencing with operator "^"

The reference content can be accessed with the postfix operator "^", e.g. *ref_var*^.

If the reference does not contain a valid value at runtime, the processing task is aborted and the ExecutionFaultTask called.

Before access, a check for validity by comparison with NULL is recommended.

Multilevel dereferencing is possible, e.g. reference to a structure that contains further references. Dereferenced variables can be used anywhere in expressions.

If a method is called via a reference to a class or function block, the class or function block must be dereferenced.

Dereferenced values cannot be used as actual parameters for the _releaseSemaphore() and _testandSetSemaphore() system functions.

Dynamic type conversion with the operator "?="

The operator "?=" can be used for dynamic type conversion only with references to classes and with interface variables (variables for object-oriented interfaces). It supports the following variants:

- Assignment between references to classes.
- Assignment of a reference to a class for an interface variable.
  Whereby, the reference to a class must be specified with the dereferencing operator "^".
- Assignment of an interface variable for a reference to a class.
- Assignment between interface variables

A dynamic type conversion is performed if the following two conditions are satisfied:

1. The variable to the right of the operator contains the reference to the instance of a class.
2. For the variable to the left of the operator, the following applies:
   - The variable is an interface variable:
     The object-oriented interface that this variable has as data type must implement the class on the right-hand side.
   - The variable is a reference to a class:
     This class corresponds to the class on the right-hand side or is a base class for this class.

In this case, the reference on the right-hand side is converted and transferred to the left-hand side.

If the type conversion fails, the variable on the left-hand side contains the "NULL" value.

Comparison operators

Only the comparison operators "=" and "<>" can be used to check for equality or inequality.

The operators ">", "<", ">=", "<=" as well as the MIN and MAX standard functions are not permitted.

6.8 I/O references (as of kernel V5.1)

6.8.1 Forming I/O references

Declaration of I/O references

I/O references allow symbolic access to I/O variables within classes and function blocks. This allows the classes and function blocks also to be used in libraries.

The I/O references are declared exclusively in classes and function blocks in the declaration block for static variables (VAR). The declaration as retentive variable (VAR RETAIN) or constant VAR CONSTANT is not possible.

var_name AT %I : data_type* ; // for inputs
var_name AT %Q* : *data_type ;* // for outputs

The following are permissible as data types:

- Bit data types (e.g. BOOL, WORD)
- Integer data types (e.g. SINT, UINT)
- Arrays of these data types, other than arrays of the BOOL data type

Provided the variable block is not identified as PUBLIC, it must be identified with the OVERRIDE keyword so that the variables can be initialized instance-specific for the instance formation of the class or function block.

Example

```
FUNCTION_BLOCK myFB
  VAR OVERRIDE
    myInput AT %I*  : INT; // Declaration input
    myOutput AT %Q* : INT; // Declaration output
    myOutArr AT %Q* : ARRAY[0..3] OF BYTE;
                           // Array of outputs
  END_VAR
  ; // ...
END_FUNCTION_BLOCK
```

Initialization for the definition

I/O references are initialized with NULL. Only "NULL" is permissible for explicit specification of the initialization. With the specification of NULL as initialization value, for the instance declaration of the class or of the function block, the I/O reference must not necessarily be linked with an I/O variable from the address list. This allows the software to be structured so that, if necessary, it adapts itself to non-linked I/O references.

An I/O reference identified with NULL in the initialization must be checked for validity before use, as also usual for all other reference types. A comparison with NULL can be used for this purpose.

```
FUNCTION_BLOCK myFB
  VAR OVERRIDE
    myInput AT %I* : INT := NULL;
  END_VAR
  IF (myInput <> NULL) THEN
    // Validity check of the I/O reference
    Testvar := myInput;
    // Access to the value
  END_IF;
  // ...
END_FUNCTION_BLOCK
```

Principle

The following restrictions apply to I/O reference variables:

- I/O references cannot be forwarded as in/out parameters (VAR_INOUT) to other program organization units (POU).
- I/O references cannot be assigned general references and vice versa. In particular, the REF() standard function cannot be used on I/O variables to assign I/O references.
- I/O references cannot be passed to input parameters (VAR_INPUT) defined as variable-length ARRAY.
- I/O references are not displayed in the Symbol browser and in watch tables.
- I/O references cannot be read or written per HMI, Web access or OPC access.
- I/O references must not be declared in retentive variables (VAR RETAIN).
- Bit access is possible to I/O references of the BYTE, WORD or DWORD data type. This also applies to the associated standard functions (e.g. _setbit()).
- Use as argument for marshalling and endian functions is not possible.

6.8.2 Linking I/O references to I/O variables

For the instance declaration of the higher-level class or function block, I/O references are linked with an I/O variable. This is done with the instance-specific initialization of the class or function block. Whereby, the specified I/O reference is assigned an I/O variable from the address list rather than a constant.

The following apply:

- I/O references for inputs (%I*) can be linked with any I/O variable of an appropriate data type.
- I/O references for outputs (%Q*) can be linked only with an I/O variable for an output of an appropriate data type. This variable may not be identified with the attribute "read only".

For partial initialization, as usual for structures, the initialization value of the associated data type is used for non-explicitly specified components.

I/O references declared as ARRAY must be linked completely with an appropriate I/O ARRAY variable of the address list. Individual variables as I/O reference can also be linked with the individual elements of an I/O ARRAY variable by specifying a constant index value.

```
VAR_GLOBAL
  myFbInst1 : myFB := (myInput  := InputVarW3,
                       myOutput := OutputVarW10);
  myFbInst2 : myFB := (myInput  := InputVarW4,
                       myOutput := OutputVarW14);
END_VAR
```

The compiler checks whether all I/O references are linked with I/O variables at least with instances in VAR_GLOBAL or in the VAR declaration blocks of programs. Linking is also possible even such an instance was included previously in other classes or function blocks.

If a class or a function block possesses an I/O reference, the following must be observed when the instance is declared in the VAR blocks of other classes or function blocks:

- Either the instance itself is initialized,
- Or the variable block in which the instance is declared must be capable of initialization (OVERRIDE or PUBLIC identifier).

The above-mentioned checks are omitted when the I/O reference for the declaration was initialized with NULL.

Integration of ST in SIMOTION

7

This section describes the interoperability of ST programs and SIMOTION SCOUT.

7.1 Source file sections

An overview of the meaning of the source file sections has been provided in Structure of an ST source file (Page 110). There you will find detailed information about, for example:

- the syntax of source file sections (Page 257)
- how you can declare public and use (Page 278) data between different ST source files.

7.1.1 Use of the source file sections

You must follow certain structure and syntax rules in your source file sections 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 so that the former is recognized by the latter.
  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, see Identifiers in ST (Page 97).

Note

A template with all possible source file sections (template for example unit) is available in the online help.

7.1.1.1 Interface section

The interface section contains statements for declaring data public and using data (data types, variables, function blocks, functions, programs). Technology packages and libraries can also be downloaded.

The interface section has the following syntax:

Syntax

```
INTERFACE
// Interface statements (optional)
END_INTERFACE
```

An individual identifier of the section cannot be specified.

Optionally, interface statements exist in the following order between reserved words INTERFACE and END_INTERFACE.

1. Specification of utilized technology package. Syntax:
   `USEPACKAGE tp-name [AS namespace];`
   For more details, refer to the *SIMOTION Basic Functions* Function Manual.
2. Specification of utilized libraries.
   Syntax:
   `USELIB library-name-list [AS namespace];`
   For more information, see "Using data types, functions and function blocks from libraries (Page 347)".

3. Reference to other units in order to use their public components.
   The syntax is:
   `USES unit_name-list;`
   For more information, see "USES statement in a using unit (Page 281)".
4. Declarations and information regarding declaration as public to other program sources:
   - Data type definitions (Page 274):
     User-defined data types (UDT) that are public and valid within the entire ST source file
   - Variable declarations (Page 275):
     Unit variables and unit constants that are public and valid within the entire ST source file.
     Permissible keywords: See Section "Variable declaration" (Page 275).
   - Object-oriented interfaces (Page 271):
     Object-oriented interfaces that are public and valid within the entire ST source file.
     As of version V4.5 of the SIMOTION Kernel and with compiler option (Page 63) "Permit object-oriented programming" enabled.

     Note

     Object-oriented interfaces utilize the same keywords as the interface section (INTERFACE / END_INTERFACE).

   - Specification of the program organization units (POUs) to be declared public.
     Syntax:
     `FUNCTION fc_name;`
     `FUNCTION_BLOCK fb_name;`
     `PROGRAM program_name;`
     `CLASS class_name;` (as of version V4.5 of the SIMOTION Kernel and with compiler option (Page 63) "Permit object-oriented programming" enabled)
     If the "Permit forward declarations" compiler option (Page 63) is activated, they are also interpreted as POU prototypes for the forward declaration (Page 381).
   - POU prototypes for the forward declaration (Page 381).
     Specification of the prototypes for program organization units with forward declaration (only effective if the "Permit forward declarations" compiler option (Page 63) is activated). They are also interpreted as POUs to be exported.

All technology packages, libraries, units used, data type declarations, variable declarations, object-oriented interfaces and program organization units listed in the interface section are declared public. For further information about declaring public, see "Interface section of a unit with Declare Public function (Page 279)".

Sequence

The interface section is the first section of an ST source file.

Note

Optionally, the unit statement can precede the interface section, see "Identifier of the unit (Page 278)".

The order of the interface statements 1 to 4 is fixed.

Within number 4, any order is permitted. The individual declaration blocks for data type definitions (including POU prototypes) and variables declarations can appear more than once.

Please note: Identifiers must be declared before they are used.

Only if the "Permit forward declarations" compiler option (Page 63) is activated: When declaring an instance of a function block or a class, it is sufficient for the prototype of the function block to have been declared beforehand. It is not possible to initialize the function block or the class.

Frequency

Once per ST source file

Mandatory section

Yes

7.1.1.2 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:

Syntax

```
IMPLEMENTATION
// Implementation statements (optional)
END_IMPLEMENTATION
```

An individual identifier of the section cannot be specified.

Optionally, implementation statements (main part of the ST source file) exist in the following order between the reserved words IMPLEMENTATION and END_IMPLEMENTATION:

1. Reference to other units in order to use their public components. Syntax:
   `USES unit_name-list;`
   For more information, see "USES statement in a using unit (Page 281)".
2. Declarations
   - Data type definitions (Page 274):
     User-defined data types (UDT) that are valid within the entire ST source file
   - Variable declarations (Page 275):
     Unit variables and unit constants that are valid within the entire ST source file.
     Permissible keywords: See Section "Variable declaration" (Page 275).
   - Object-oriented interfaces (Page 271):
     Object-oriented interfaces that are valid within the entire ST source file
     As of version V4.5 of the SIMOTION Kernel and with compiler option (Page 63) "Permit object-oriented programming" activated.
   - POU prototypes for the forward declaration (Page 381).
     Specification of the prototypes for program organization units with forward declaration (only effective if the "Permit forward declarations" compiler option (Page 63) is activated).

   Note

   Data types, variables or object-oriented interfaces that are to be declared public and used in other program sources must be fully declared in the interface section (Page 258).

3. Program organization units (Page 262).

Sequence

Always follows the interface section (Page 258).

If the "Permit forward declarations" compiler option (Page 63) is **not** activated, the following applies:

- The order of the implementation statements indicated above is mandatory; within numbers 2 and 3, any order is permitted.
- All identifiers must be declared before they are used.

If the "Permit forward declarations" compiler option (Page 63) is activated, the following applies:

- As regards the order of the interface statements, the following applies:
  Numbers 2 and 3 need no longer be strictly separated. Declarations may be freely programmed between program organization units.
  Exception: POU prototypes must be declared before the relevant program organization unit is defined or used.
- As long as the following conditions are satisfied, the identifiers of program organization units (Page 262) can be used before they are defined.
  - There are no restrictions with respect to functions (Page 263), programs (Page 266), expressions (Page 267) or methods (Page 270). Any order is permitted.
    A POU prototype may be optionally declared in advance for programs or functions.
  - This is possible for function blocks (Page 264) or classes (Page 268) subject to the following restrictions (e.g. when the instance is declared):
    Any order is permitted if initialization is not specified. In this case, however, it is absolutely essential that the prototype of the function block or class is declared beforehand.
    If initialization is specified, the function block or the class must be fully defined beforehand.
- All other identifiers (e.g. data types, variables) must be declared before they are used.

Frequency

Once per ST source file

Mandatory section

Yes

7.1.1.3 Program organization units (POUs)

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

- Functions (FC) (Page 263)
- Function blocks (FB) (Page 264)
- Programs (Page 266)
- Expressions (Page 267)
- Classes (Page 268) (as of version V4.5 of the SIMOTION Kernel and with compiler option (Page 63) "Permit object-oriented programming" activated)

The following applies for the sequence of the POUs in an ST source file:

- 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 63) is activated. The sequence is arbitrary: The following applies to the specification of a prototype for forward declaration (Page 381) in the interface section (Page 258) or the implementation section (Page 260):
  - Essential for declaring instances with function blocks and classes
  - Optional when calling functions and programs

7.1.1.4 Functions (FCs)

Functions (FC) are classified as program organization units (POUs). Functions are parameterized 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:

Syntax

```
FUNCTION name : function_data_type
// Declaration section
// Statement section
END_FUNCTION
```

name stands for the identifier of the function, while *function_data_type* stands for the data type of the return value.

Permissible keywords for the variable declaration in the declaration section (Page 272): See Section "Variable declaration" (Page 275).

Note the following for functions with *function_data_type* <> VOID: In the body (Page 273), 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 260).

- If the "Permit forward declarations" compiler option (Page 63) is **not** activated, the following applies:
  In the source file, functions must precede the POU from which they are called.
- If the "Permit forward declarations" compiler option (Page 63) is activated, the following applies:
  The sequence is arbitrary: There is an option to specify a prototype for the forward declaration (Page 381) in the interface section (Page 258) or the implementation section (Page 260).

The declaration section (Page 272) must precede the statement section (Page 273).

Frequency

Any number of times per ST source file

Mandatory section

No

Further information on functions (FC)

See Section "Creating and calling functions and function blocks" (Page 181).

7.1.1.5 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:

Syntax

```
FUNCTION_BLOCK name
// Declaration section
    // Methods (Only with compiler option (Page 63) "Permit object-oriented programming")
// Statement section
END_FUNCTION_BLOCK
```

name stands for the identifier of the function block.

Permissible keywords for the declaration of variables in the declaration section: See Section "Variable declaration" (Page 275).

Special features

Before you call a function block (FB), you must declare an instance: You declare a variable and enter the identifier of the function block as the data type. You can declare the instance locally (within VAR / END_VAR in the declaration sections of a program or a function block).

You can also declare the instance globally (within VAR_GLOBAL/END_VAR in the interface or implementation section).

- If the "Permit forward declarations" compiler option (Page 63) is **not** activated, the following applies:
  This is only possible with FBs which make used program sources and libraries available, but not with FBs which are defined in the same ST source file.
- If the "Permit forward declarations" compiler option (Page 63) is activated, the following applies:
  You can declare global instances of FBs which are defined in the same ST source file.
  - For instance declaration without specification of initialization:
    The instance may be declared before the FB is fully defined. In this case, it is essential that a prototype for the forward declaration (Page 381) is specified in the interface section (Page 258) or the implementation section (Page 260).
  - For instance declaration with specification of initialization:
    The instance must always be declared after the FB is fully defined.

You cannot declare an instance of an FB in functions or methods.

Sequence

FBs can only be defined in the implementation section (Page 260).

- If the "Permit forward declarations" compiler option (Page 63) is **not** activated, the following applies:
  In the source file, an FB must precede the POU in which it is used, e.g. in which an instance of the FB is declared as local variable.
- If the "Permit forward declarations" compiler option (Page 63) is enabled, the following applies:
  - Without specification of initialization:
    The POU which uses the FB may precede the definition of the FB. In this case, it is essential that a prototype for the forward declaration (Page 381) is specified in the interface section (Page 258) or the implementation section (Page 260).
  - With specification of initialization (only with compiler option (Page 63) "Permit object-oriented programming"):
    The FB must be fully defined before the POU in which it is used.

The declaration section (Page 272) must precede the statement section (Page 273).

Frequency

Any number of times per ST source file

Mandatory section

No

Further information about function blocks (FB)

See Section "Creating and calling functions and function blocks" (Page 181).

7.1.1.6 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 functions (FC) and function blocks (FB), as well as methods in classes and function blocks.

Programs have the following syntax:

Syntax

```
PROGRAM name
// Declaration section
// Statement section
END_PROGRAM
```

name stands for the name of the program.

Permissible keywords for the variable declaration in the declaration section: See Section "Variable declaration" (Page 275).

Sequence

Programs can only be defined in the implementation section (Page 260).

- If the "Permit forward declarations" compiler option (Page 63) is **not** activated, the following applies:
  Programs are expediently placed after FCs, FBs, expressions and classes. The program then recognizes these source file sections and can make use of them.
- If the "Permit forward declarations" compiler option (Page 63) is enabled, the following applies:
  The sequence is arbitrary: When a program is called within a program, there is an option to specify a prototype for the forward declaration (Page 381) in the interface section (Page 258) or the implementation section (Page 260).

The declaration section (Page 272) must precede the statement section (Page 273).

Frequency

Any number of times per ST source file

Mandatory section

No

Further information on programs

See Section "Programs" (Page 207).

7.1.1.7 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 (Page 173) 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:

Syntax

```
EXPRESSION name
// Declaration section
// Statement section
END_EXPRESSION
```

name stands for the identifier of the expression.

Permissible keywords for the variable declaration in the declaration section: See Section "Variable declaration" (Page 275).

Please note: In the body (Page 273), 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 260) of an ST source file.

- If the "Permit forward declarations" compiler option (Page 63) is **not** activated, the following applies:
  Expressions must precede the program organization unit in which they are called from a WAITFORCONDITION control structure.
- If the "Permit forward declarations" compiler option (Page 63) is activated, the following applies:
  The sequence is arbitrary: There is no option to specify a prototype for the forward declaration (Page 381) in the interface section (Page 258) or the implementation section (Page 260).

The declaration section (Page 272) must precede the statement section (Page 273).

Frequency

Any number of times per ST source file

Mandatory section

No

Further information on expressions

See Section "Expressions" (Page 208).

Examples in conjunction with the WAITFORCONDITION statement: See SIMOTION Basic Functions Function Manual

7.1.1.8 Classes (as of Kernel V4.5)

Classes belong to the program organization units (POE). They form a capsule for variables and methods (Page 270). Only variables and methods that have the access identifier PUBLIC are visible outside of the class. Another class can be derived from a class (base class). This derived class inherits all the variables and methods from the base class.

A SIMOTION Kernel as of version V4.5 is mandatory for the purposes of defining and using classes. Classes can only be defined when compiler option (Page 63) "Permit object-oriented programming" is enabled.

Classes have the following syntax:

Syntax (simplified)

```
CLASS name
// Declaration section
    // methods
END_CLASS
```

name stands for the identifier of the class.

Permissible keywords for the variable declaration in the declaration section: See Section "Variable declaration" (Page 275).

For a complete syntax diagram see Figure 6-10 Syntax: Class (Page 229).

Special features

Before using public methods or variables of a class you must declare an instance of the class: You declare a variable and enter the identifier of the class as the data type. You can declare the instance locally (within VAR / END_VAR in the declaration sections of a program, a function block or a class).

You can also declare the instance globally (within VAR_GLOBAL/END_VAR in the interface or implementation section).

- If the "Permit forward declarations" compiler option (Page 63) is **not** activated, the following applies:
  This is only possible with classes which make used program sources and libraries available, but not with classes which are defined in the same ST source file.
- If the "Permit forward declarations" compiler option (Page 63) is activated, the following applies:
  You can declare global instances of classes which are defined in the same ST source file.
  - For instance declaration without specification of initialization:
    The instance may be declared before the class is fully defined. In this case, it is essential that a prototype for the forward declaration (Page 381) is specified in the interface section (Page 258) or the implementation section (Page 260).
  - For instance declaration with specification of initialization:
    The instance must always be declared after the class is fully defined.

You cannot declare an instance of a class in functions or methods.

Sequence

Classes can only be defined in the implementation section (Page 260).

- If the "Permit forward declarations" compiler option (Page 63) is **not** activated, the following applies:
  In the source file, a class must precede the POU in which it is used, e.g. in which an instance of the class is declared as a local variable.
- If the "Permit forward declarations" compiler option (Page 63) is enabled, the following applies:
  - Without specification of initialization:
    The POU which uses the class may precede the definition of the class. In this case, it is essential that a prototype for the forward declaration (Page 229) is specified in the interface section (Page 258) or the implementation section (Page 260).
  - With specification of initialization:
    The class must be fully defined before the POU in which it is used.

The declaration section (Page 272) must precede the methods (Page 270).

Frequency

Any number of times per ST source file

Mandatory section

No

Additional information on classes

See “Object-Oriented Programming - OOP (as of kernel V4.5)” (Page 213) section

7.1.1.9 Methods

Methods belong to the program organization units (POE). They are callable and parameterizable source file sections with temporary data within classes (Page 268) or function blocks (Page 264). All internal variables lose their values when the method is exited and are reinitialized the next time the method is called.

Methods can only be defined when compiler option (Page 63) “Permit object-oriented programming" is enabled.

Methods have the following syntax:

Syntax

```
METHOD access-spec name : method_data_type
    // Variable declaration
// Statement section
END_METHOD
```

access-spec stands for the optional access identifier (Page 215):

- In classes: PUBLIC, PROTECTED, PRIVATE. Default setting: PROTECTED.
- In function blocks: PUBLIC, PRIVATE. PRIVATE is default setting.

name stands for the identifier of the method, while *method_data_type* stands for the data type of the return value.

Permissible keywords for the variable declaration in the declaration section (Page 272): See Section "Variable declaration" (Page 275).

Note the methods with *method_data_type* <> VOID: In the statement section (Page 273), an expression of data type *method_data_type* must be assigned to the method identifier!

For a complete syntax diagram see Figure 6-3 Syntax: Method (Page 217).

Special feature

The compiler option (Page 63) “Permit object-oriented programming" needs to be enabled.

Sequence

Methods can only be defined within classes (Page 268) and function blocks (Page 264).

- If the "Permit forward declarations" compiler option (Page 63) is **not** activated, the following applies:
  In the source file, methods must precede the POU from which they are called.
- If the "Permit forward declarations" compiler option (Page 63) is activated.
  The sequence is arbitrary:

The declaration section (Page 272) must precede the statement section (Page 273).

Frequency

Any number of times per class or function block.

Mandatory section

No

Additional information on methods

See “Object-Oriented Programming - OOP (as of kernel V4.5)” (Page 213) section

7.1.1.10 Object-oriented interface (as of Kernel V4.5)

Object-oriented interfaces define the call interface for public methods (Page 270) (access identifier PUBLIC). Method prototypes are defined for this purpose in the object-oriented interface. These methods with the definition "prototype" are formulated in classes (Page 268) which implement the object-oriented interfaces.

Another interface can be derived from an object-oriented interface (base interface). This derived interface inherits all of the method prototypes of the base interface.

Note

While object-oriented interfaces start and end with the same keywords as the interface section (Page 258) of a unit (INTERFACE / END_INTERFACE), they have different meanings and are used in different ways.

Object-oriented interfaces can only be defined and used with a SIMOTION Kernel as of version V4.5. The compiler option (Page 63) "Permit object-oriented programming" also needs to be activated.

Object-oriented interfaces have the following syntax:

Syntax (simplified)

```
INTERFACE name
    // Method prototype
    METHOD method_name : method_data_type
        // Variable declaration (parameters only)
    END_METHOD
    // Other method prototypes
END_INTERFACE
```

name stands for the identifier of the object-oriented interface.

For complete syntax diagrams see Figure 6-11 Syntax: Object-oriented interface (Page 239) and Figure 6-12 Syntax: Method prototype (Page 240).

Special features

In classes (Page 268) that implement object-oriented interfaces, all methods (Page 270)that are defined as prototypes in the interfaces must be formulated. These methods must always be declared public (access identifier PUBLIC). The call interface (input, output and in/out parameters) specified in the interface must be transferred unchanged.

Variables of object-oriented interfaces can also be declared. Instances of classes that implement the same object-oriented interface can be assigned to these variables.

Sequence

You can define object-oriented interfaces in the interface section (Page 258) and the implementation section (Page 260). In most cases, they are declared in the interface section because they are generally declared PUBLIC.

They must always be declared in full before they are used in the declaration of an interface variable or an implementing class. This applies irrespective of the compiler option (Page 63) "Permit forward declarations".

Frequency

Any number of times per ST source file

Mandatory section

No

Further information about object-oriented interfaces

See section "Object-oriented interfaces" (Page 238).

7.1.1.11 Declaration section

The declaration section of a program organization unit (POU) contains the data type definition and the variable declaration of the POU.

The declaration section has the following structure:

Structure

```
// Data type definition
// Variable declaration
```

Sequence

The declaration section has no explicit keywords at the start or end. It begins after the keyword of the respective program organization unit (POU) and ends with the first executable statement of the statement section.

It contains the following in any order:

- Data type definitions (Page 274):
  User-defined data types (UDT) that are valid locally in the POU.
  Data types cannot be defined within methods.
- Variable declarations (Page 275):
  Variables and constants that are valid locally in the POU
  Permissible keywords according to the respective POU: See Section "Variable declaration" (Page 275).

Please note: Identifiers must be declared before they are used.

Frequency

Once per POU

Mandatory section

No

7.1.1.12 Statement section

The statement section of a POU consists of the individual (executable) statements.

The statement section has the following structure:

Configuration

```
// 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

Further information on statements

See following sections:

- Value assignments and expressions (Page 145)
- Control statements (Page 162)

- Call of functions and function block calls (Page 194)
- Calling methods (Page 221)

7.1.1.13 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:

Syntax

```
TYPE
name : data_type_specification;
    // ...
END_TYPE
```

name represents the name of the individual data type that you use for the Variable declarations.

data_type_specification stands for any data type or a structure. Any number of individual data types can appear between TYPE and END_TYPE.

Sequence

You can define UDTs as follows:

- In the Interface section:
  The UDTs are recognized within the ST source file and will be exported
  They can be used in the interface and implementation section for declaration of unit variables and in all POUs for declaration of local variables.
  In addition, they can be used in all units which import this ST source file (in SIMOTION ST with the USES statement).
- In the Implementation section:
  The UDTs are recognized within the ST source file
  They can be used in the implementation section for declaration of unit variables and in all POUs for declaration of local variables.
- In the declaration section of a POU (FC, FB, program, expression, class):
  The UDTs are only recognized locally within the POU
  They can only be used within the POU for declaration of local variables.

UDTs must be defined before they are used in a variable declaration.

Frequency

The TYPE / END_TYPE 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

Further information on user-defined data types (UDT)

See Section "User-defined data types" (Page 117).

7.1.1.14 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:

Syntax

```
variable_type
   name_list : data_type;
    // ...
END_VAR
```

variable_type represents the keyword of the variable type being declared. The permitted keywords depend on the source file section.

- In the Interface section or Implementation section of an ST source file:
  `VAR_GLOBAL`: Non-retentive unit variable
  `VAR_GLOBAL CONSTANT`: Unit constant
  `VAR_GLOBAL RETAIN`: Retentive unit variable
- In the Declaration section of a function:
  `VAR`: Local variable
  `VAR CONSTANT`: Local constant
  `VAR_INPUT`: Input parameter
  `VAR_OUTPUT`: Output parameters
  `VAR_IN_OUT`: In/out parameter

- In the Declaration section of a function block:
  `VAR`: Local variable
  - With activated compiler option (Page 63) "Permit object-oriented programming", the following applies: An access identifier (Page 185) (PUBLIC, PRIVATE, INTERNAL) can be optionally specified after the keyword. Default: PRIVATE.

  `VAR CONSTANT`: Local constant
  - With activated compiler option (Page 63) "Permit object-oriented programming", the following applies: An access identifier (Page 185) (PUBLIC, PRIVATE, INTERNAL) can be optionally specified after the keyword. Default: PRIVATE.

  `VAR RETAIN`: Retentive variable
  - With activated compiler option (Page 63) "Permit object-oriented programming", the following applies: An access identifier (Page 185) (PRIVATE, INTERNAL) can be optionally specified after the keyword. Default: PRIVATE.

  `VAR_TEMP`: Temporary variable
  `VAR_INPUT`: Input parameter
  `VAR_OUTPUT`: Output parameter
  `VAR_IN_OUT`: In/out parameter
- In the Declaration section of a program:
  `VAR`: Local variable
  `VAR CONSTANT`: Local constant
  `VAR_TEMP`: Temporary variable
- In the Declaration section of an expression:
  `VAR`: Local variable
  `VAR CONSTANT`: Local constant
  `VAR_INPUT`: Input parameter (as of Version 4.1 of the SIMOTION Kernel)
  `VAR_IN_OUT`: In/out parameter (as of Version 4.1 of the SIMOTION Kernel)
- In declaration section of a class (Page 268) (as of version 4.5 of the SIMOTION Kernel):
  `VAR`: Local variable (static)
  - An access identifier (Page 215) (PUBLIC, PROTECTED, PRIVATE, INTERNAL) can be optionally specified after the keyword. Default: PROTECTED.

  `VAR CONSTANT`: Local constant
  - An access identifier (Page 215) (PUBLIC, PROTECTED, PRIVATE, INTERNAL) can be optionally specified after the keyword. Default: PROTECTED.

  `VAR RETAIN`: Retentive variable
  - An access identifier (Page 215) (PROTECTED, PRIVATE, INTERNAL) can be optionally specified after the keyword. Default setting: PROTECTED.

- In the declaration section of a method (Page 270):
  VAR: Local variable
  VAR CONSTANT: Local constant
  VAR RETAIN: Retentive variable
  VAR_INPUT: Input parameters
  VAR_OUTPUT: Output parameters
  VAR_IN_OUT: In/out parameter
- In the method prototypes within an object-oriented interface (Page 271):
  VAR_INPUT: Input parameter
  VAR_OUTPUT: Output parameter
  VAR_IN_OUT: In-out parameter

name_list is the list of identifiers of the *data_type* data type to be declared.

Sequence

The variable is declared:

- In the Interface section of the ST source file:
  Permissible keywords: See above at *Syntax*.
  The unit variables are recognized within the ST source file and will be exported.
  They can be used in all POUs of the ST source file.
  In addition, they can be used in all units which import this ST source file (in SIMOTION ST with the USES statement).
- In the Implementation section of the ST source file:
  Permissible keywords: See above at *Syntax*.
  The unit variables are recognized within the ST source file.
  They can be used in all POUs of the ST source file.
- In the declaration section of a class:
  Permissible keywords: See above under *Syntax*.
  The variables are only recognized locally within the class.
- In the declaration section of a POU (FC, FB, program, expression):
  Permissible keywords according to the type of POU: See above at *Syntax*.
  The variables are only recognized locally within the POU.
  Exceptions:
  - You can also access the output parameters of a function block outside the FB.
  - You can access the input parameters of a function block from outside the FB when the compiler option (Page 63) "Permit language extensions" has been activated.
- In the declaration section of a method:
  Permissible keywords: See above under *Syntax*.
  The variables are only recognized locally within the method.
- In the method prototypes within an object-oriented interface (Page 271):
  Permissible keywords: See above under *Syntax*.
  The parameters define the call interface of the method.

Variables must be declared before they are used.

Frequency

The number of times the *variable_type* / END_VAR declaration block of a specific variable type can appear depends on the associated source file section:

- In the interface and implementation section of the ST source:
  The declaration blocks may appear more than once.
- In the declaration section of a POU (FC, FB, program, expression, class, method):
  Each declaration block (other than VAR CONSTANT / END_VAR) may appear just once in the declaration section.
- In the method prototypes within an object-oriented interface:
  Each declaration block may only occur once in each method prototype.

Permitted declaration blocks and keywords depending on the associated source file section: See above at *Syntax*.

Any number of variable declarations are possible within a declaration block.

Mandatory section

No

Further information about variable declaration

See Section "Variable declaration" (Page 136).

7.1.2 Declaring ST source files public and using them

ST has a unit concept which can be used to access global variables, data types, functions (FC), function blocks (FB) and programs from other source files. Thus, for example, you can compile reusable subroutines and make them available.

7.1.2.1 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 25) or Change the properties of an ST source file (Page 27).

The unit statement is ignored if the name specified there differs from the name of the ST source file.

7.1.2.2 Interface section of a unit with Declare Public function

You can enter the following constructs in the interface section of a unit with Declare Public function. The syntax of the constructs is only implied here, for details, see "Interface section (Page 258)".

- The data type declarations to be declared public
  TYPE
  User-defined data types with their complete declaration.
- The variable declarations to be declared public
  VAR_GLOBAL, VAR_GLOBAL RETAIN, or VAR_GLOBAL CONSTANT
  Non-retentive and retentive unit variables and unit constants with their complete declaration.
- The POU to be declared public (functions, function blocks, programs and classes)
  Specify each POU (function, function block, program or class) to be declared public 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*;
  - CLASS *class_name*; (as of version V4.5 of the SIMOTION Kernel and with compiler option (Page 63) "Permit object-oriented programming" activated)

  If the "Permit forward declarations" compiler option (Page 63) is activated, they are also interpreted as POU prototypes for the forward declaration (Page 381).
  You program the POU itself in the implementation section (Page 260) of the ST source file.
- The object-oriented interfaces to be declared public as of version V4.5 of the SIMOTION Kernel with compiler option (Page 63) "Permit object-oriented programming" activated.
  INTERFACE *if_name* ... END_INTERFACE
  Object-oriented interfaces with their complete declaration.

The information can be specified in any sequence:

Note

The following further specifications are possible in the interface section, they are listed **before** the data types, variables and POUs to be declared public:

1. Specification of utilized technology packages (USEPACKAGE ...).
2. Specification of utilized libraries (USELIB ...).
3. Reference to other units in order to use their public components (USES ...).

These imported technology packages, libraries and units are also declared public. For inheritance, see "USES statement in a using unit (Page 281)".

You must adhere to the order presented for the specifications in the interface section of a unit (ST source file), see "Interface section (Page 258)". 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 are not declared public 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.

7.1.2.3 Example of a unit with Declare Public function

Below is an example of a unit with Declare Public function (*myUnit_A*). It is imported by *myUnit_B* (see Example of a using unit (Page 282)).

Table 7-1 Example of a unit with Declare Public function

```
UNIT myUnit_A;    // Optional, name of the ST source file

INTERFACE
    // ... USES statement also possible here
    TYPE        // Declaration of data types to be declared public
        color : (RED, GREEN, BLUE);
    END_TYPE
    VAR_GLOBAL
        cycle : INT := 1;  // Declaration of unit variables to be declared public
                           // unit variables to be exported
    END_VAR
    FUNCTION myFC;         // DECLARE PUBLIC statement of an FC
    FUNCTION_BLOCK myFB;   // DECLARE PUBLIC statement of an FB
    CLASS myClass;         // DECLARE PUBLIC statement of a class
                           // (as of version V4.5 of the SIMOTION Kernel)
                           // and activated compiler option
                           // "Permit object-oriented programming")
    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 formulated
        ;     // ... (Statements)
    END_FUNCTION_BLOCK

    CLASS myClass          // Class formulated
                           // (as of version V4.5 of the SIMOTION Kernel)
                           // and activated compiler option
                           // "Permit object-oriented programming")
        ;     // ... (Statements)
    END_CLASS;

    PROGRAM myProgram_A    // Program formulated
        ;     // ... (Statements)
    END_PROGRAM
END_IMPLEMENTATION
```

7.1.2.4 USES statement in a using unit

Enter the following statement in the interface section or implementation section of a using unit:

```
USES unit_name-list
```

unit_name-list is a list of the units to be used separated by commas.

Example:

```
USES unit_1, unit_2, unit_3;
```

This enables you to access the following elements that are declared public in the interface section of the units used (e.g. ST source file, MCC unit):

- User-defined data types (UDT)
- Unit variables and unit constants
- Programs, functions, function blocks and classes
- Used technology packages, libraries and units

You can use the used 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 used, 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 7-2 Implications regarding placement of USES statement in interface section or in implementation section of the ST source file

Effect	USES statement in the interface section	USES statement in the implementation section
Inheritance	The current unit continues declaring the used unit public; the used unit is inherited by all other units that access the current unit. Example: 1. Unit B uses Unit A in the **interface** section. 2. Unit C in turn uses Unit B. 3. Then Unit C also uses Unit A automatically. A → B → C ⇒ A → C Because of inheritance, Unit A need not be used explicitly in Unit C.	Inheritance is interrupted. Example: 1. Unit B uses Unit A in the **implementation** section. 2. Unit C in turn uses Unit B. 3. Then Unit C has no automatic access to Unit A. Unit C must explicitly use Unit A if it wants to access Unit A.
Variable declaration	The declaration of a unit variable of a used data type is possible in: • Interface section • Implementation section	The declaration of a unit variable of a used 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.

7.1.2.5 Example of a using unit

Below is an example of a using unit (*myUnit_B*). This uses the *myUnit_A* unit from the Example of a unit with Declare Public function (Page 280).

Table 7-3 Example of a using unit

```
UNIT myUnit_B;        // Optional, name of the ST source file
INTERFACE
    // ...  if applicable USES statement
    PROGRAM myProgram_B;
    // Specification of declared public programs, FBs, FCs, classes,
    // Data types and unit variables
END_INTERFACE

IMPLEMENTATION
    USES myUnit_A;              // Specification of used unit

    VAR_GLOBAL
        myInstance : myFB;      // Declaration of an instance
                                // of the public FB of the used unit
        mycolor : color;        // Declaration of a variable
                                // of the public data type of the used unit
    END_VAR

    PROGRAM myProgram_B
        mycolor := GREEN;       // Value assignment to variables
                                // of the public data type of the used unit
        cycle := cycle + 1;     // Value assignment to
                                // public variables of the used unit
    END_PROGRAM
END_IMPLEMENTATION
```

7.2 Variables in SIMOTION

This summarizes the variables available in ST.

7.2.1 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, a method, a function block or a class)

System variables

Variable type	Meaning
System variables of the SIMOTION device	Each SIMOTION device and technology object has specific system variables. These can be accessed as follows: • Within the SIMOTION device from all programs • From HMI devices You can monitor system variables in the symbol browser.
System variables of technology objects	

Global user variables

Variable type	Meaning
I/O variables	You can assign symbolic names to the I/O addresses of the SIMOTION device or the peripherals. This allows you to have the following direct accesses and process image accesses to the I/O: • Within the SIMOTION device from all programs • From HMI devices You create these variables in the symbol browser after you have selected the I/O element in the project navigator. You can monitor I/O variables in the symbol browser.
Global device variables	User-defined variables which can be accessed by all SIMOTION device programs and HMI devices. You create these variables in the symbol browser after you have selected the GLOBAL DEVICE VARIABLES element in the project navigator. Global device variables can be defined as retentive. This means that they will remain stored even when the SIMOTION device power supply is disconnected. You can monitor global device variables in the symbol browser.
Unit variables	User-defined variables that all programs, function blocks, and functions (e.g. ST source, MCC source, LAD/FBD source) can access within a unit. Declare these variables in the unit: • In the interface section: These variables are declared public and can be used in other units (e.g. ST source files, MCC units, LAD/FBD units). They are also available on HMI-device as standard. • In the implementation section: You can only access these variables within the associated unit. You can declare unit variables as retentive. This means that they will remain stored even when the SIMOTION device power supply is disconnected. You can monitor unit variables in the symbol browser.

Local user variables

Variable type	Meaning
	User-defined variables which can be accessed from within the program (or function, function block) in which they were defined.
Variable of a program (program variable)	Variable is declared in a program. The variable can only be accessed within this program. A differentiation is made between static, temporary and retentive variables: • Static variables are initialized according to the memory area in which they are stored. Specify this memory area by means of a compiler option. By default, the static variables are initialized depending on the task to which the program is assigned (see *SIMOTION Basic Functions* Function Manual). You can monitor static variables in the symbol browser. • Temporary variables are initialized every time the program in a task is called. Temporary variables cannot be monitored in the symbol browser. • Retentive variables (as of version V4.5 of the SIMOTION Kernel) retain their value even when the SIMOTION device power supply is disconnected. You can monitor retentive variables in the symbol browser.
Variable of a function or method	Variable is declared in a function (FC) or method. The variable can only be accessed within this FC or method. Variables in functions or methods are temporary; they are initialized each time the FC or method is called. They cannot be monitored in the symbol browser.
Variable of a function block	Variable is declared in a function (FB). The variable can only be accessed within this function block. A differentiation is made between static, temporary and retentive variables: • Static variables retain their value when the FB terminates. They are initialized only when the instance of the FB is initialized; this depends on the variable type with which the instance of the FB was declared. You can monitor static variables in the symbol browser. Only when compiler option "Permit object-oriented programming" is activated: Static variables with the PUBLIC access identifier can be accessed externally. • Temporary variables lose their value when the FB terminates. The next time the FB is called, they are reinitialized. Temporary variables cannot be monitored in the symbol browser. • Retentive variables (as of version V4.5 of the SIMOTION Kernel) retain their value even when the SIMOTION device power supply is disconnected. You can monitor retentive variables in the symbol browser.
Variables of a class (as of version V4.5 of the SIMOTION Kernel)	Variable is declared in a class. A differentiation is made between static and retentive variables: • Static variables are initialized only when the instance of the class is initialized; this depends on the variable type with which the instance of the class has been declared. Static variables can only be accessed from within this class and derived classes. Only static variables with the PUBLIC access identifier can be accessed from outside the class. • Retentive variables retain their value even when the SIMOTION device power supply is disconnected. Retentive variables can only be accessed from within this class and derived classes. They cannot be accessed from outside the class. You can monitor the variables 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 323))
  - Accessing the process image (see Access to the fixed process image of the BackgroundTask (Page 332))
  - Creating and using global device variables (see Using global device variables (Page 294))
  - 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 300).

7.2.1.1 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). More information about declaring data public and using public data in other program sources can be found in Declaring ST source files public and using them (Page 278).
  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 315) and Controlling compiler with attributes (Page 374).

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 143)).

You will find tips for the efficient use of unit variables in the *SIMOTION Basic Functions* Function Manual.

7.2.1.2 Non-retentive unit variables

Non-retentive unit variables lose their value in the event of a power failure.

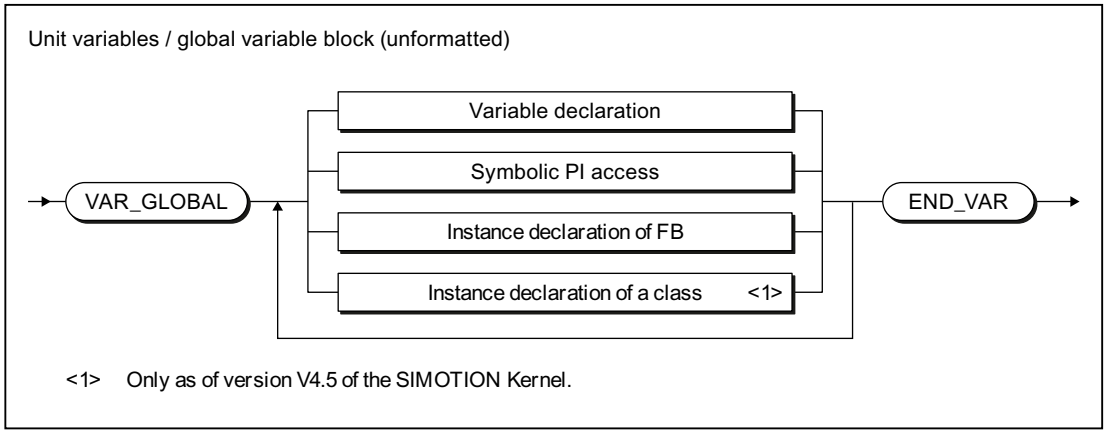


Figure 7-1 Syntax: Unit variables

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 137) and Initialization of variables or data types (Page 139)).

For the scope of the declaration and the HMI export, see Unit variables (Page 285).

Note

For initialization of the non-retentive unit variables:

- See Initialization of non-retentive global variables (Page 302).
- 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 309).

Table 7-4 Examples of non-retentive unit variables

```
INTERFACE
    VAR_GLOBAL    // These variables are declared public.
        rotation1        : INT;
        field1           : ARRAY [1..10] OF REAL;
        flag1            : BOOL;
        motor1           : motor;    // Instance declaration
    END_VAR
END_INTERFACE

IMPLEMENTATION
    VAR_GLOBAL    // These variables are not declared public.
        rotation2        : INT;
        field2           : ARRAY [1..10] OF REAL;
        flag2            : BOOL;
        motor2           : motor;    // Instance declaration
    END_VAR
END_IMPLEMENTATION
```

7.2.1.3 Retentive unit variables

Retentive unit variables permit permanent storage of variable values even throughout a power failure.

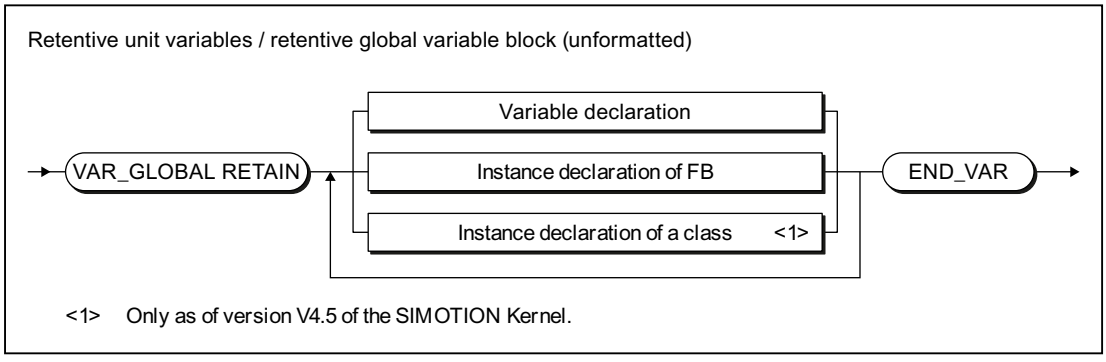


Figure 7-2 Syntax: Retentive global variable block

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 137) and Initialization of variables or data types (Page 139)).

For the scope of the declaration and the HMI export, see Unit variables (Page 285).

Note

- For initialization of the retentive unit variables:
  - See Initialization of retentive global variables (Page 300).
  - 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 309).
- 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 7-5 Examples of retentive unit variables

```
VAR_GLOBAL RETAIN
    Measuring field : ARRAY[1..10] OF REAL;
    Pass : INT;
    Switch : BOOL;
END_VAR
```

7.2.1.4 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 291):
  Static variables retain their value over all passes of the source file section (block memory).
- Temporary variables (Page 292):
  Temporary variables are initialized each time the unit section is called again.
- Retentive local variables (Page 292)
  SIMOTION Kernel as of version V4.5.
  They retain their value:
  - in the event of a power failure
  - when the task is started
  - over all passes of the source file section

See also: Initialization of local variables (Page 304).

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 retentive, 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 7-6 Keywords for declaring retentive, static and temporary variables depending on source file section

Source file section	Keywords for the declaration		
	Static variables	**Temporary variables**	**Retentive local variables (as of kernel V4.5)**
Function	–	VAR / END_VAR or VAR_TEMP / END_VAR or VAR_INPUT / END_VAR or VAR_IN_OUT / END_VAR[2] or VAR_OUTPUT / /END_VAR	–
Expression	–	VAR/ END_VAR or VAR_INPUT / END_VAR or VAR_IN_OUT / END_VAR[2]	–
Function block	VAR / END_VAR[1] or VAR_INPUT / END_VAR[1] or VAR_OUTPUT / END_VAR[1]	VAR_TEMP / END_VAR or VAR_IN_OUT / END_VAR[2]	VAR RETAIN / END_VAR
Program	VAR / END_VAR[3]	VAR_TEMP / END_VAR	VAR RETAIN / END_VAR
Methods (within a class or a function block)[4]	–	VAR / END_VAR or VAR_TEMP / END_VAR or VAR_INPUT / END_VAR or VAR_IN_OUT / END_VAR[2] or VAR_OUTPUT / /END_VAR	–
Class[4]	VAR / END_VAR[1]	–	VAR RETAIN / END_VAR

[1] The initialization of the variable depends on initialization of the declared instance. See Initialization of instances of function blocks (FBs) or classes (Page 307).

[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 305).

[4] Only with active compiler option “Permit object-oriented programming”.

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 300).

Table 7-7 Examples of static and temporary variables

```
IMPLEMENTATION
    FUNCTION testFkt
        VAR            // Declaration of temporary variables
            flag : BOOL;
        END_VAR
    END_FUNCTION
    FUNCTION_BLOCK testFbst
        VAR RETAIN     // Declaration of retentive variables
            position1 : LREAL
        END_VAR

        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 RETAIN     // Declaration of retentive variables
            position2 : LREAL
        END_VAR

        VAR           // Declaration of static variables
            rotation2 : INT;
        END_VAR

        VAR_TEMP      // Declaration of temporary variables
            help1, help2 : REAL;
        END_VAR
    END_PROGRAM
    CLASS testcls
        VAR RETAIN     // Declaration of retentive variables
            position3 : LREAL
        END_VAR

        VAR           // Declaration of static variables
            rotation3 : INT;
        END_VAR
    END_CLASS
END_IMPLEMENTATION
```

7.2.1.5 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
- Classes (as of version V4.5 of the SIMOTION Kernel and with compiler option "Permit object-oriented programming" enabled)

Static variables are declared in a static variable block.

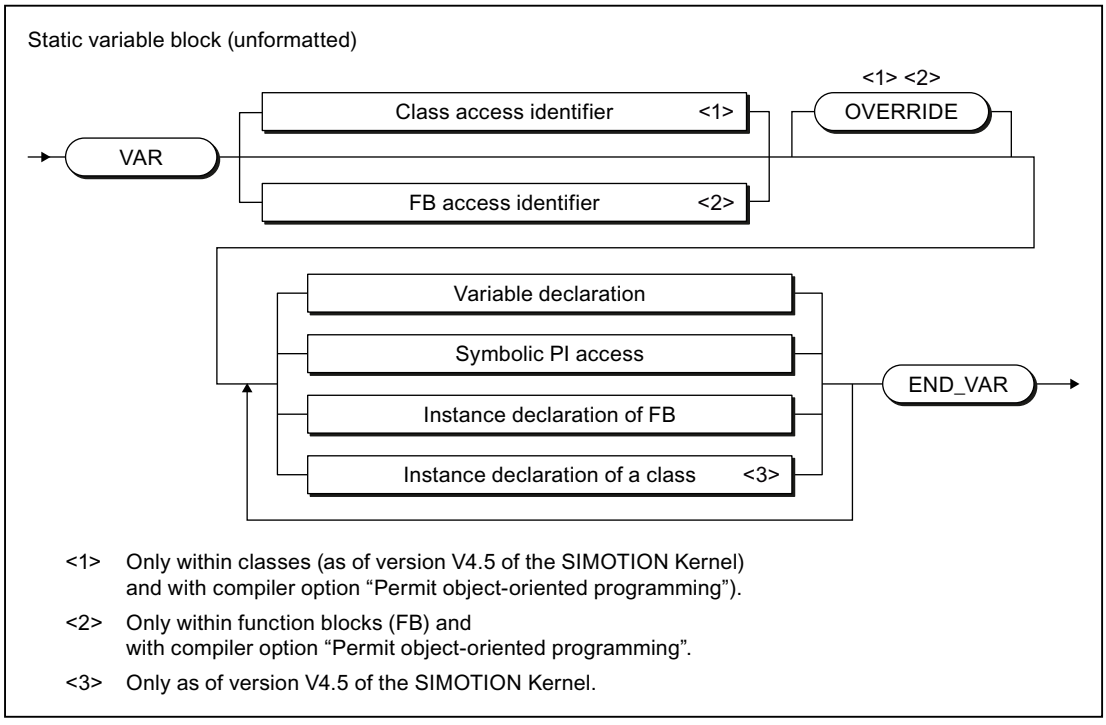


Figure 7-3 Syntax: Static variable block

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.
- Declaring instances of classes (SIMOTION Kernel as of version V4.5)

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 305).
- In function blocks: Depending on the initialization of the declared instance.
  See also Initialization of instances of function blocks (FBs) (Page 307).

7.2.1.6 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
- Methods (within classes or function blocks, with compiler option “Permit object-oriented programming”)
- Expressions

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

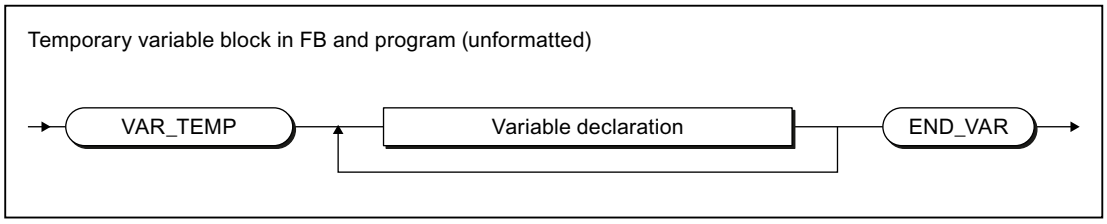


Figure 7-4 Syntax: Temporary variable block in the FB or program

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

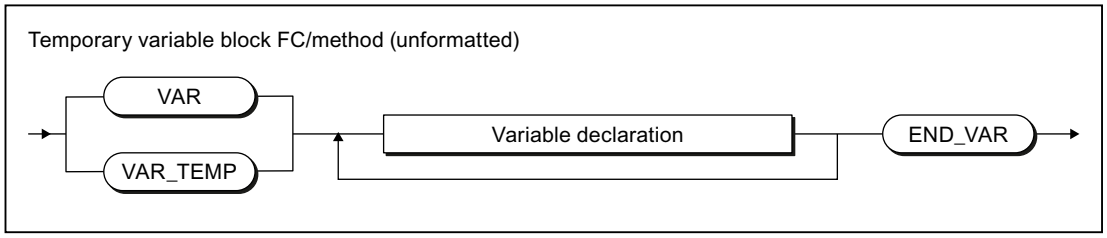


Figure 7-5 Syntax: Temporary variable block in an FC or a method

7.2.1.7 Retentive local variables (as of kernel V4.5)

You can declare retentive local variables in SIMOTION Kernel as of version V4.5.

They retain their value over all passes of the source file section in the event of a power failure or if the task is started.

The following source file sections contain retentive local variables:

- Programs
- Function blocks
- Classes (with compiler option “Permit object-oriented programming")

They are declared in the retentive local variable block.

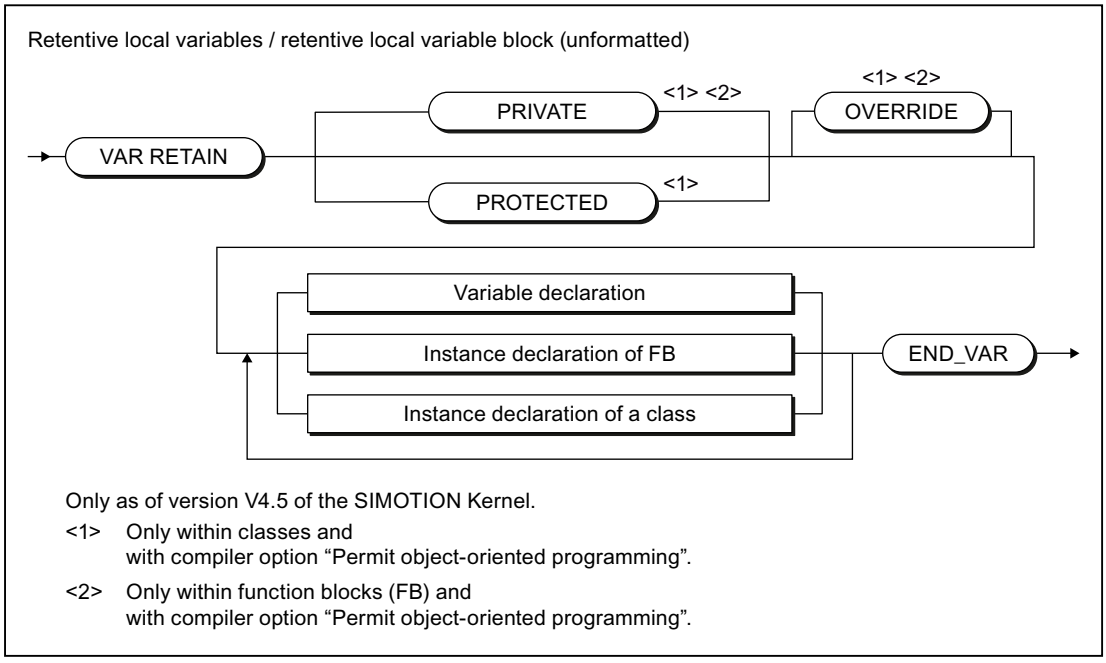


Figure 7-6 Syntax: Retentive local variable block

You can do the following in the retentive local variable block, according to the syntax in the figure:

- Declare variables (name and data type), optionally with initialization.
  Exception. No variables of data types can be declared in the retentive local variable block if they exclusively contain elements that cannot be saved (e.g. variables of data types of the technology objects).
- Declare instances of the function blocks.
  Exception: No instances of function blocks can be declared in the retentive local variable block if they contain the following:
  - Retentive local variables
  - Instances of system function blocks
- Declaring instances of classes (SIMOTION Kernel as of version V4.5)
  Exception: No instances of classes can be declared in the retentive local variable block if they contain the following:
  - Retentive local variables
  - Instances of system function blocks

For initialization of retentive local variables:

- In programs: dependent on the compiler option "Only create program instance data once”.
  See also Initialization of retentive local variables of programs (Page 306).
- In function blocks or classes: dependent on the initialization of the retentive variables of the declared instance.
  See also Initialization of retentive local variables of function blocks and classes (Page 308).

7.2.2 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.

7.2.3 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 297) section.

Table 7-8 Memory areas assigned to different variable types and their initialization

Memory area	Assigned variable types	Initialization[3]
Retentive memory	• Retentive unit variables • As of version V4.5 of the SIMOTION Kernel: Retentive local variables of programs and function blocks	During download using the download settings
User memory of unit	• Non-retentive unit variables • Function block instances declared with VAR_GLOBAL, including the associated static variables (VAR, VAR_INPUT, VAR_OUTPUT) • As of version V4.5 of the SIMOTION Kernel: Instances of classes declared with VAR_GLOBAL, including the associated variables (VAR) Also for the activated "Create program instance data only once" compiler option (Page 63): • Local variables of the unit programs declared with VAR • Function block instances declared with VAR_GLOBAL, including the associated static variables (VAR, VAR_INPUT, VAR_OUTPUT) • As of version V4.5 of the SIMOTION Kernel: Instances of classes declared with VAR within the programs of the unit, including the associated variables (VAR)	• When the device is switched on • During download using the download settings • For transition to the RUN mode: – Version V4.2 and higher of the SIMOTION Kernel: By activating the check box on the SIMOTION device, **Initialization of non-retentive global variables and program data during STOP-RUN transition**. – As of Version V4.1 of the SIMOTION Kernel: If the associated declaration block specifies the following pragma: { BlockInit_OnDeviceRun := ALWAYS; } See also Controlling compiler with attributes (Page 374)
User memory of task	For the deactivated "Create program instance data only once" compiler option (Page 63) (default): • Local variables declared with VAR of the assigned programs • Function block instances declared with VAR within the assigned programs, including the associated static variables (VAR, VAR_INPUT, VAR_OUTPUT) • As of version V4.5 of the SIMOTION Kernel: Instances of classes declared with VAR within the programs of the unit, including the associated variables (VAR)	According to execution behavior of task: • Sequential tasks: Each time task is started • Cyclic tasks: 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 • Local variables declared with VAR_TEMP of the program called in the task	On each call of the program in the task
	• Reference (pointer) to called function block instances • Local variables of function blocks declared with VAR_TEMP • In/out parameters of function blocks declared with VAR_IN_OUT[1]	Each time the function block instance is called
	• Variables of called functions or methods declared with VAR, VAR_INPUT, VAR_OUTPUT or VAR_IN_OUT[1] • Return value of called functions or methods	Each time the function or method is called

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

[2] See also Memory requirement of the variables on the local data stack (Page 299).

[3] For a detailed description of the initialization behavior of the individual variable types, see Time of the variable initialization (Page 300).

7.2.3.1 Example of memory areas

Table 7-9 Example of memory areas of the variable types - Part 1

```
INTERFACE
// The statements in the interface section specify,
// which source contents are declared public.
    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 7-10 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 7-11 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
```

7.2.3.2 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 295). 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 1 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 365) function.

7.2.4 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).

7.2.4.1 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 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.
- With a firmware update (upgrade) or activation of a kernel in accordance with the following description.

Behavior during download

Table 7-12 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]: • **Yes**[2]: All retentive global device variables are initialized. • **No**[3]: The retentive global device variables are only initialized if their version code is changed. See: Version code of global variables and their initialization during download (Page 309).
Retentive unit variables	The behavior during the download depends on the *Initialization of retentive program data and retentive global device variables* setting[1]: • **Yes**[2]: All retentive unit variables (all units) are initialized. • **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. See: Version code of global variables and their initialization during download (Page 309).

[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 mode, provided the following condition is fulfilled:
With the **SIMOTION ST** programming language
The following attribute has been specified within a pragma: { BlockInit_OnChange := TRUE; }.
With the **SIMOTION MCC** or **SIMOTION LAD/FBD** programming languages:
A pragma line is inserted in the declaration table with the following check box enabled in this line: *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.

Behavior during upgrade or configuration change

When the SIMOTION device is upgraded to a new version of the SIMOTION Kernel or if the configuration is changed, the retentive variables are initialized as described below:

Table 7-13 Initialization of retentive global variables during upgrade or configuration change

Variable type	Time of the variable initialization
Retentive global device variables	This data is always initialized.
Retentive unit variables	This data can be retained. See section "Retaining retentive data" in the "Basic Functions for Modular Machines" Function Manual.

7.2.4.2 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 300), 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 7-14 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]: • **Yes**[2]: All non-retentive global device variables are initialized. • **No**[3]: The non-retentive global device variables are only initialized if their version code is changed. See: Version code of global variables and their initialization during download (Page 309).
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]: • **Yes**[2]: All non-retentive unit variables (all units) are initialized. • **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. See: Version code of global variables and their initialization during download (Page 309).

[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 381) **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.

7.2.4.3 Initialization of local variables

Local variables are initialized:

- For the initialization of retentive unit variables (Page 300).
- For the initialization of non-retentive unit variables (Page 302).
- Also, according to the following description:

Table 7-15 Initialization of local variables

Variable type	Time of the variable initialization
Local program variables	Local variables of programs are initialized differently: • Static variables (VAR) are initialized according to the memory area in which they are stored. See: Initialization of static program variables (Page 305). • Temporary variables (VAR_TEMP) are initialized every time the program of the task is called.
Local variables of function blocks (FB)	Local variables of function blocks are initialized differently: • Static variables (VAR, VAR_IN, VAR_OUT) are only initialized when the FB instance is initialized. See: Initialization of instances of function blocks (FBs) (Page 307). • 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 365) function.

7.2.4.4 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 63) 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 302).
  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 381) **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.

7.2.4.5 Initialization of retentive local variables of programs

The following versions relate to the following retentive variables:

- Local variables of a unit program declared with VAR RETAIN,
- Function block or class instances declared with VAR RETAIN within a unit program, including the associated static variables (VAR, VAR_INPUT, VAR_OUTPUT).
- Function block instances declared with VAR within a unit program, or classes for their associated retentive variables (VAR RETAIN).

These retentive variables are initialized:

- When retentive unit variables are initialized (Page 300).
- And also when a download is performed according to the following description.

The initialization behavior when downloading is determined by the compiler option (Page 63) "Only create program instance data once".

- For the deactivated "Only create program instance data once" compiler option (default):
  The retentive variables are initialized in the following cases:
  - The data structure of the VAR RETAIN declaration blocks of the instances involved changes.
  - Additional tasks are activated in the execution system.
  - The assignment or sequence of the programs with retentive variables changes within a task.
- For the activated "Only create program instance data once" compiler option:
  The retentive variables are initialized in the following cases:
  - The data structure of the VAR RETAIN declaration blocks of the instances involved changes.
  - The sequence of the VAR_GLOBAL data blocks or of the programs (PROGRAM) changes in the implementation section of the source.

7.2.4.6 Initialization of instances of function blocks (FBs) or classes

The initialization of a function block instance (Page 199) or a class instance (as of version V4.5 of the SIMOTION Kernel) (Page 226) 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 302).
- Local declaration in a program (within VAR / END_VAR):
  Initialization as for static variables of programs, see Initialization of static variables of programs (Page 305).
- 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 365) function.

7.2.4.7 Initialization of retentive local variables of function blocks (FBs) and classes

The initialization of retentive variables of a function block (FB) or a class (as of version V4.5 of the SIMOTION Kernel) is determined by the location at which the instance of a function block (Page 199) or the instance of a class (Page 226) is declared:

- Global declaration (within VAR_GLOBAL/END_VAR in the interface of implementation section):
  The retentive variables are initialized:
  - When retentive unit variables are initialized (Page 300).
  - In addition to the download:
    The data structure of the VAR RETAIN declaration blocks of the instances involved changes.
    The sequence of the VAR_GLOBAL declaration blocks in the current source changes.
- Local declaration in a program (within VAR / END_VAR):
  Initialization as for retentive local variables of programs, see Initialization of retentive local variables of programs (Page 306).
- Local declaration in a function block (within VAR / END_VAR):
  Initialization as with the retentive variables of 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 365) function.

7.2.4.8 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 300) and Initialization of non-retentive global variables (Page 302).

The behavior during the download is shown below for:

- Non-retentive system variables
- Retentive system variables

Table 7-16 Initializing technology object system variables during download

Variable type	Time of the variable initialization
Non-retentive system variables	Behavior during download, depending on the *Initialization of all non-retentive data for technology objects* setting[1]: • **Yes**[2]: All technology objects are initialized. – All technology objects are restructured and all non-retentive system variables are initialized. – All technological alarms are cleared. • **No**[3]: Only technology objects changed in SIMOTION SCOUT are initialized. – The technology objects in question are restructured and all non-retentive system variables are initialized. – All alarms that are pending on the relevant technology objects are cleared. – If an alarm that can only be acknowledged with *Power On* is pending on a technology object that will not be initialized, the download is aborted.
Retentive system variables	Only if a technology object was changed in SIMOTION SCOUT, will its retentive system variables be initialized. The retentive system variables of all other technology objects are retained (e.g. absolute encoder calibration).
[1] Default 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.	

7.2.4.9 Version ID of global variables and their initialization during download

Table 7-17 Version code of global variables and their initialization during download

Data segment		Description of version code
Global device variables		
	Retentive global device variables Non-retentive global device variables	• Separate version code for each data segment of the global device variables. • The version code of the data segment changes for: – Add or remove a variable within the data segment – Change of the identifier or the data type of a variable within the data segment • This version code does not change on: – Changes in the other data segment – Changes to initialization values[1] • 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		

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

Data segment		Description of version code
	Non-retentive unit variables in the interface section	Only as of version V4.5 of the SIMOTION Kernel. • The retentive variables of all instances within a declaration block (VAR_GLOBAL / END_VAR) are summarized as a separate retentive data block • to which a separate version code is assigned. • The version code of this retentive data block changes: – The data structure of the retentive local variables for the instances within the declaration block changes. – The sequence of the declaration blocks (VAR_GLOBAL / END_VAR) within the program source changes. • During downloading[2], the rule is: This data block is only initialized when the version code of a data block is changed[5]. • Functions for data backup and initialization take into account the version code of the data blocks.
	Non-retentive unit variables in the implementation section	

[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 300)" and "Initialization of non-retentive global variables (Page 302)".

[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 use of units depends on the programming language, refer to the relevant section (Page 281).

[5] Initialization of a changed data block also occurs during a download in RUN mode, provided that the following condition is fulfilled:
With the **SIMOTION ST** programming language
The following attribute (Page 368) has been specified within a pragma (Page 374): { BlockInit_OnChange := TRUE; }.
With the **SIMOTION MCC** or **SIMOTION LAD/FBD** programming languages:
A pragma line is inserted in the declaration table with the following check box enabled in this line: *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.

7.2.5 Declaring variables in the Variable declaration dialog box ("on-the-fly" variable declaration)

You can declare a new variable in an ST source file without placing the cursor in the desired declaration block. For example, the cursor can remain at the code position in the body of a POU at which you want to use the new variable. You declare the variable "on-the fly" in the **Variable Declaration** dialog box.

Procedure

To define variables "on-the-fly" in the **Variable Declaration** dialog box, proceed as follows:

1. Place the cursor at the place of use of the new variable.
2. Optionally, enter the name of the new variable.
3. Select the **Variable declaration** context menu.
   The **Variable Declaration** dialog box appears.
   If necessary, the **Name** field is preassigned with the name of the new variable.

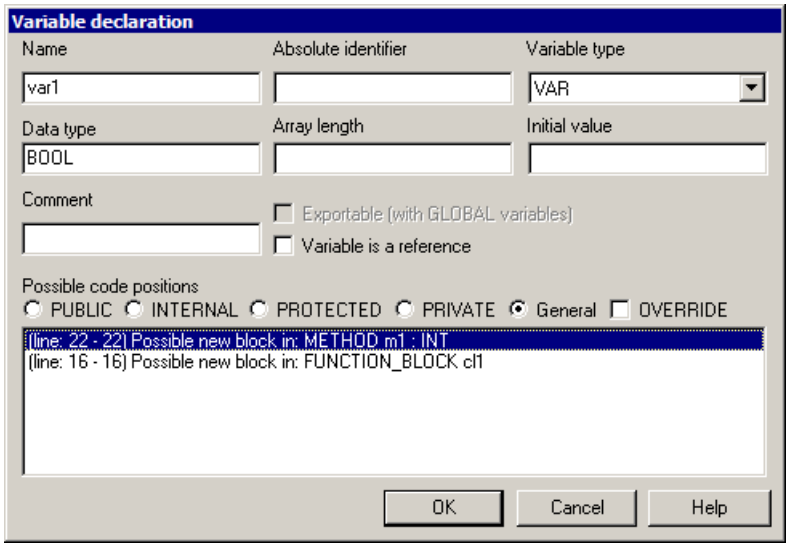


Figure 7-7 Variable Declaration dialog box

4. Make the other entries in the dialog box according to the following table.
5. Set the desired filter criteria according to the following table and select the desired code position.
6. Confirm with **OK**.

Principle

Table 7-18 Variable declaration parameter description

Field	Description
Name	Enter the name of the variable to be declared. The name must comply with the Rules for identifiers (Page 97). The field is preassigned with the word in which the cursor is positioned when the window is called.
Absolute identifier	You define the variable as access to the fixed process image of the BackgroundTask (Page 332) by entering an absolute identifier. The field is only active when a data type is selected that is contained in the general data types (Page 116) ANY_BIT or ANY_INT. Enter the absolute identifier according to the syntax (Page 339).

Field	Description
Variable type	Select the variable type. Available for selection are all permitted keywords for variable types (Page 137). Default is VAR when the cursor is within a POU when the window is called, otherwise VAR_GLOBAL.
Data type	Enter the data type. You are supported by the "Automatic completion" (Page 49) function (CTRL+space).
Array length	Not available when an absolute identifier is entered or the variable is a reference. You can define the variable as an array [0..N-1] by entering an array length N. You have the following options for entering the array length: • You can specify a constant positive integer value. • You can enter a value range with ".." separating the min. and max. values. • You can enter a constant expression of data type DINT (or of a data type which is implicitly convertible to DINT). If the array is empty, a single variable is created rather than an array.
Initial value	Not available when an absolute identifier is entered. You can specify this initialization value as a constant or as an expression. The following are permissible: • Constants • Arithmetic operations • Bit slice and data conversion functions
Comment	A comment can be entered in this column. All characters and special characters are permitted.
Exportable (for GLOBAL variables)	Only available when the keyword for a unit variable or unit constant has been selected as variable type (e.g. VAR_GLOBAL). If the checkbox is activated: • The variable declaration is inserted in the interface section of the ST source file. If the checkbox is deactivated: • The variable declaration is inserted in the implementation section of the ST source file.
Variable is a reference	Only available when a valid data type has been entered. With activated checkbox, a general reference (Page 249) is created for the entered data type. The keyword REF_TO is set as prefix for the specified data type. The checkbox is activated automatically when the entered data type starts with the keyword REF_TO.
Possible code positions	

Field		Description
	Filter criteria	You can select the following as filter criteria: • An access identifier within classes, function blocks or namespaces – **PUBLIC** – **INTERNAL** – **PROTECTED** – **PRIVATE** – **General** (default, not an access identifier) • The keyword **OVERRIDE**.
	List of the code positions	A list of the possible code positions is displayed at which the variable can be declared. A distinction is made between: • Insertion in an existing variable block • Creation of a new variable block The first entry in the list is selected. In the editor window, the code position is displayed and selected at which the variable or the new variable block will be inserted. Select from the

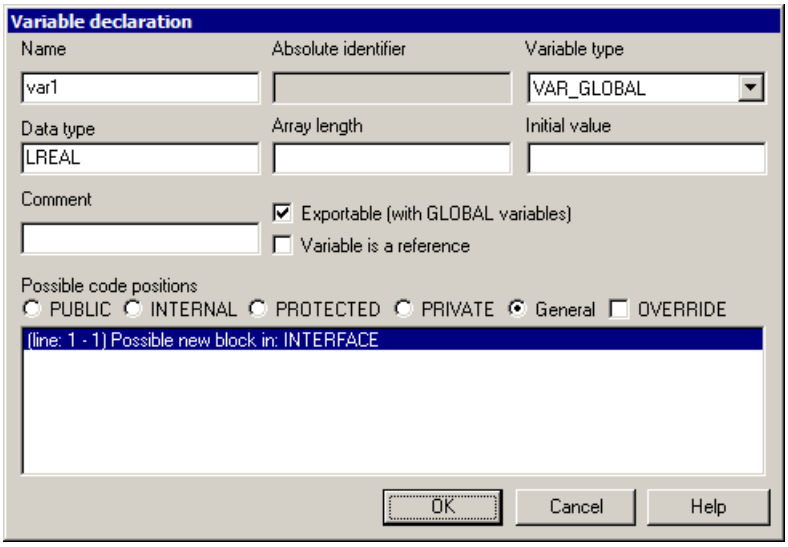


Figure 7-8 Example: Variable declaration

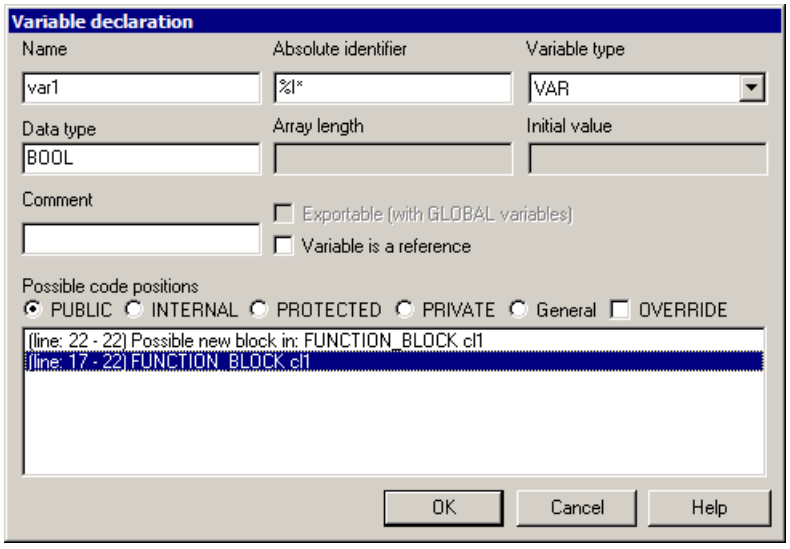


Figure 7-9 Example: Variable declaration (absolute name)

Result

The variable or the new variable block is inserted at the selected code position in the ST source file.

The cursor is at the same position before the dialog box was called.

7.2.6 Variables and HMI devices

Exported variables

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 374).
  - 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 a data block identified in this way are not exported to HMI devices. The HMI consistency check is also omitted for them during the download.

- Non-retentive static variables (VAR) of programs provided that the compiler option (Page 63) "Only create program instance data once" is activated
- Non-retentive static variables (VAR) of function blocks
  This is the default setting when the compiler option (Page 63) "Permit object-oriented programming" is enabled. 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 374).

  The variables of a data block identified in this way are not exported to HMI devices. The HMI consistency check is also omitted for them during the download.
  The pragma is not evaluated if the compiler option "Permit object-oriented programming" is **not** activated. The export behavior cannot be changed.
- Non-retentive public variables (VAR PUBLIC) of classes (default setting, as of version V4.5 of the SIMOTION Kernel)
  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 374).

  The variables of a data block identified in this way are not exported to HMI devices. The HMI consistency check is also omitted for them during the download.

Note

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.

Non-exported variables

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 374).
  - 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 a data block identified in this way are exported to HMI devices. Consequently, they undergo the HMI consistency check during downloading.
- Local variables (VAR, VAR_TEMP) of functions or methods
- Temporary variables (VAR_TEMP) of programs, function blocks or classes
- Retentive local variables (VAR RETAIN) of programs (as of version 4.5 of the SIMOTION Kernel)
- Retentive local variables (VAR RETAIN) of function blocks (as of version 4.5 of the SIMOTION Kernel)
  This is the default setting when the compiler option (Page 63) "Permit object-oriented programming" is enabled. 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 374).

  The variables of a data block identified in this way are exported to HMI devices. Consequently, they undergo the HMI consistency check during downloading.
  The pragma is not evaluated if the compiler option "Permit object-oriented programming" is **not** activated. The export behavior cannot be changed.
- Retentive local variables (VAR RETAIN) of classes (default setting, as of version 4.5 of the SIMOTION Kernel)
  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 374).

  The variables of a data block identified in this way are exported to HMI devices. Consequently, they undergo the HMI consistency check during downloading.

- Non-retentive static variables (VAR) of programs provided that the compiler option (Page 63) "Only create program instance data once" is **not** activated
- Non-retentive protected or private variables (VAR, VAR PROTECTED, VAR PRIVATE) of classes (default setting, as of version V4.5 of the SIMOTION Kernel)
  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 374).

  The variables of a data block identified in this way are exported to HMI devices. Consequently, they undergo the HMI consistency check during downloading.

Example for the SIMOTION ST programming language

Table 7-19 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
```

7.3 Access to inputs and outputs (process image, I/O variables)

7.3.1 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.
  Further information: Direct access and process image of the cyclic tasks (Page 323).
- 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.var-name*.
  Further information: Direct access and process image of the cyclic tasks (Page 323).
- 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 range 0 .. 63 can be used. Exception: I/O addresses that are accessed using the process image of the cyclic task can only be used with the setting “Common process image” (Page 334) (as of Kernel V4.2).
  Further information: Access to the fixed process image of the BackgroundTask (Page 332).

A comparison of the most important properties is contained in "Important properties of direct access and process image" (Page 320).

You can use I/O variables like any other variable, see "Access I/O variables" (Page 343).

Note

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

7.3.2 Important features of direct access and process image access

Table 7-20 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 **Exception**: I/O variables comprising more than one byte must not contain addresses 63 and 64 contiguously (example: PIW63 or PQD62 are not permitted).		Addresses 0 .. 63. **Exception**: Up to version V4.1 of the SIMOTION Kernel or the "Separate process image" setting, addresses used for the process image of the cyclic tasks are not permitted.
Address configuration	Necessary. The addresses used must be present in the I/O and appropriately configured. The "Rules for I/O addresses for direct access and the process image of the cyclic tasks" (Page 326) must be observed.		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: • SynchronousTasks, • TimerInterruptTasks, • BackgroundTask.	BackgroundTask.
Memory area for process images	-	Depends on the SIMOTON Kernel version: • Up to version V4.1: Separate memory areas in all cases • As of version V4.2: Option to select a common memory area	

	Direct access	Access to process image of cyclic tasks	Access to fixed process image of the BackgroundTask
Update	• Onboard I/O of SIMOTION devices C230-2, C240, and C240 PN: Update occurs in a cycle clock of 125 µs. • With I/O devices – via PROFIBUS DP or PROFINET on an isochronous SIMOTION device[1] – via DRIVE-CLiQ – Onboard I/O of SIMOTION D devices: The update is performed in the position control cycle clock[2]. • With I/O devices – via PROFIBUS DP or PROFINET on a **non-**isochronous SIMOTION device[1] – via P-Bus: The update is performed in the interpolation cycle[2] [3]. Inputs are read at the start of the cycle clock. Outputs are written at the end of the cycle clock.	Update occurs with the assigned task: • Inputs are read before the assigned task is started and transferred to the process input image. • Process output image is written to the outputs after the assigned task has been completed.	An update is made with the *BackgroundTask*: • Inputs are read before the *BackgroundTask* is started and is transferred to the process input image. • Process output image is written to the outputs when the *BackgroundTask* is complete.
Consistency	–	During the entire cycle of the assigned task. **Exception**: Direct access to output occurs.	During the entire cycle of the *BackgroundTask*. **Exception**: Direct access to output occurs.
	Consistency is only ensured for elementary data types. 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. Each byte of the address range may only be assigned to a single I/O variable. Syntax of I/O address: e.g. PIW1022, PQ63.3.		Possible, but not necessary: • For the entire device as I/O variable in the symbol browser • As unit variable • As local static variable in a program
Download of new or changed I/O variables	Only possible in STOP mode.		-
Use the absolute address	Not supported.		Possible, with the following syntax: E.g. %IW62, %Q63.3.

	Direct access	Access to process image of cyclic tasks	Access to fixed process image of the BackgroundTask
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: • Up to version V4.1 or the "Separate process image" setting: Always Big Endian • As of version V4.2 and the "Common process image" setting: 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: • Up to version V4.1: No • As of version V4.2: Yes	Yes
Write protection for outputs	Possible; **Read only** status can be selected.	Not supported.	Not supported.
Declaration of arrays	Possible.		Not supported.
Further information	Direct access and process image of the cyclic tasks (Page 323).		Access to the fixed process image of the BackgroundTask (Page 332).
Responses in the event of an error	Error during access from user program, alternative reactions available: • CPU Stop[4] • Substitute value • Last value See SIMOTION Basic Functions Description of Functions.	Error during generation of process image, alternative reactions available: • CPU stop[5] • Substitute value • Last value See SIMOTION Basic Functions Description of Functions.	Error during generation of process image, reaction: CPU stop[5] **Exception**: If a direct access has been created at the same address, the behavior set there applies.
Access			
• In the RUN operating state	Without any restrictions.	Without any restrictions.	Without any restrictions.
• During the StartupTask	Possible with restrictions: • Inputs can be read. • Outputs are not written until StartupTask is complete.	Possible with restrictions: • Inputs are read at the start of the StartupTask. • Outputs are not written until StartupTask is complete.	Possible with restrictions: • Inputs are read at the start of the StartupTask. • Outputs are not written until StartupTask is complete.

	Direct access	Access to process image of cyclic tasks	Access to fixed process image of the BackgroundTask
• During the ShutdownTask	Without any restrictions.	Possible with restrictions: • Inputs retain status of last update • Outputs are no longer written.	Possible with restrictions: • Inputs retain status of last update • Outputs are no longer written.

[1] A SIMOTION device is considered isochronous if at least one PROFIBUS DP or PROFINET interface is operated isochronously (constant bus cycle time). For SIMOTION D there is an exception with the DP Integrated interface to which the SINAMICS Integrated is connected.

[2]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).

3 IPO or IPO_2 adjustable, see “Setting system cycle clocks" section in the Basic Functions Function Manual.

[4] Call the ExecutionFaultTask.

[5] Call the PeripheralFaultTask.

7.3.3 Direct access and process image of cyclic tasks

Property

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 325) can be used.

A comparison of the most important properties, including in comparison to the fixed process image of the BackgroundTask (Page 332) is contained in "Important properties of direct access and process image" (Page 320).

Note

Observe the rules for I/O addresses for direct access and the process image of the cyclical tasks (Page 326).

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).

A detailed status of I/O variables (Page 330) can be read as of version V4.2 of the SIMOTION Kernel, for example, in order to check the availability of the I/O variables.

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 327) 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 327) 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 I/O variables (Page 330) can be read, for example, in order to check the availability of the I/O variables.

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 381) 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.
- **Up to version V4.1 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.

Note

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 332).

Table 7-21 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: • Setting available for selection • Standard for newly created devices	Up to version V4.1 of the SIMOTION Kernel applies: • System characteristic, not configurable. The following applies as of version V4.2 of the SIMOTION Kernel: • Setting available for selection • 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 333).	
Further information	Common process image (Page 334)	Separate process image (Page 336)

7.3.3.1 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 7-22 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	4.2	V4.3	As of V4.4
C230-2	0 .. 2047 [3]	0 .. 2047 [3]	0 .. 2047 [3]	–	–	–
C240	–	0 .. 4095 [3]	0 .. 4095 [3]	0 .. 4095 [3]	0 .. 4095 [3]	0 .. 4095 [3]
C240 PN [1]	–	–	0 .. 4095 [4]	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]	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]	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]	0 .. 16383 [3 4]
D445	0 .. 4095 [3]	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]	0 .. 16383 [3 4]
D455-2	–	–	–	0 .. 16383 [3 4]	0 .. 16383 [3 4]	0 .. 16383 [3 4]
P320 [2]	–	–	0 .. 4095 [3]	0 .. 4095 [3]	0 .. 4095 [3]	–

SIMOTION device	Address range for SIMOTION Kernel version					
	V3.2	V4.0	V4.1	4.2	V4.3	As of V4.4
P320-4	–	–	–	–	–	0 .. 4095 [3]
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.

7.3.3.2 Rules for I/O addresses for direct access and the process image of the cyclical tasks

Note

You must observe the following rules for the I/O variable addresses for direct access and the process image of the cyclic task (Page 323). 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 (e.g. WORD, ARRAY data type) must be within a continuous address range configured in HW Config, e.g. within the address range (slot or subslot) of *one* I/O module.
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 variable 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 332). These accesses are not taken into account during the consistency check of the project (e.g. during download).

7.3.3.3 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 insert 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 329).
   - Optional for outputs:
     Activate the **Read only** checkbox if you only want to have read access to the output. You can then read an output that is already being written by another process (e.g. output of an output cam, PROFIdrive telegram).
     A read-only output variable cannot be assigned to the process image of a cyclic task.
   - **Data type** of the variables in accordance with "Possible data types of the I/O variables" (Page 330).

4. Optionally, you can also enter or select the following (not for data type BOOL):
   - **Array length** (array size).
   - **Process image** or direct access:
     Can only be assigned if the **Read only** checkbox is 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
   - **Substitute 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 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.

Note

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 343).

7.3.3.4 Syntax for entering I/O addresses

Syntax

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 323), 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 7-23 Syntax for the input of the I/O addresses for direct access or process image of the cyclic tasks

Data type	Syntax for		Permissible address range					
	Input	Output	Direct access		Process image		e.g. direct access D435 V4.1	
BOOL	PIn.x	PQn.x	n: x:	0 .. *MaxAddr* 0 .. 7		-[1]	n: x:	0 .. 16383 0 .. 7
BYTE	PIBn	PQBn	n:	0 .. *MaxAddr*	n:	0 .. *MaxAddr*	n:	0 .. 16383
WORD	PIWn	PQWn	n:	0 .. 62 64 .. *MaxAddr* - 1	n:	0 .. 62 64 .. *MaxAddr* - 1	n:	0 .. 62 64 .. 16382
DWORD	PIDn	PQDn	n:	0 .. 60 64 .. *MaxAddr* - 3	n:	0 .. 60 64 .. *MaxAddr* - 3	n:	0 .. 60 64 .. 16380
n = logical address x = bit number								
MaxAddr =	Maximum I/O address of the SIMOTION device depending on the SIMOTION Kernel version, see Address range of the SIMOTION devices (Page 325).							
[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 326).

7.3.3.5 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 323). 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 7-24 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 329).

7.3.3.6 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*, for example, in order to check the availability of the I/O variables. 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 7-25 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** 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** 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.) 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.) 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.** 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. 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). Certain telegrams (axis, synchronous operation, output cam, measuring input telegrams) are synchronized by exchanging cyclic life-signs. Errors are detected when the cyclic life-sign is checked. This invalidates the data in the telegram. 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.** 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. Remedy: Call the _synchronizeDpInterfaces() function.
16#0000_0080	7	**I/O cannot be used synchronously** 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. Remedy: Call the _synchronizeDpInterfaces() function.
16#0000_0100	8	**Bus connection (sync slave) is not isochronous in relation to the sync master.** The SIMOTION control is the sync slave on a bus and has not yet synchronized its bus connection with the sync master. The isochronous I/O on this bus cannot be used yet. Remedy: Switch on/connect the sync master.
16#0000_0200	9	**DP station is deactivated.** The partner module has been deactivated. 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.** The connected module is in STOP mode and not sending any new data as a result. Remedy: Switch the connected module to RUN.
16#0000_0800	11	**PROFINET: Failure detected by submodule** (e.g. channel error) The connection to the connected device is OK. The error must be searched for in the connected device. Troubleshooting: Diagnostics buffer, device diagnostics with HW Config
16#0000_1000	12	**PROFINET: Failure detected by module** (e.g. submodule failed, removed, etc.) The connection to the connected device is OK. The error must be searched for in the connected device. 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.) The connection to the connected device is OK. The error must be searched for in the connected device. 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.) There is no connection to a partner on PROFINET. Possible cause: Partner is switched off, cable pulled out, incorrect parameter assignment for connection Troubleshooting: Best to use the PROFINET topology editor in HW Config.
16#0000_8000	15	**Slot/subslot is not connected** (disconnection alarm). The connection to the connected device is OK. The error must be searched for in the connected device (e.g. module/submodule removed). Troubleshooting: Diagnostics buffer, device diagnostics with HW Config
16#0001_0000	16	**Device is not connected** (station failure). There is no connection to a partner. Possible cause: Partner is switched off, cable pulled out.
16#0002_0000	17	**Substitute value behavior during access** 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 substitute value behavior set (substitute value, last value) takes effect during direct access to this address or during process image updates.
16#4000_0000	30	**Diagnostics address only** 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** There is no hardware configured for this logical address.

7.3.4 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 381) 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 7-26 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: • Setting available for selection • Standard for newly created devices	Version V4.1 and lower of the SIMOTION Kernel applies: • System characteristic, not configurable The following applies as of Version V4.2 of the SIMOTION Kernel: • Setting available for selection • 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. Updates use the configured cyclic tasks.	Not supported. 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 325)".	
Further information	Common process image (Page 334)	Separate process image (Page 336)

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 cyclic tasks (Page 323) is contained in "Important properties of direct access and process image" (Page 320).

Note

The rules for I/O addresses for direct access and the process image of the cyclical tasks (Page 326) 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 338): The absolute PI access identifier contains the address of the input/output and the data type.
- Using a symbolic PI access (Page 340): 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 342): In the symbol browser, you define a valid I/O variable for the entire device that references the corresponding absolute PI access.

7.3.4.1 Common process image (as of Kernel V4.2)

As of Version V4.2 of the SIMOTION Kernel, the "Common process image" setting (Page 381) 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 332) is a subset of the memory area for the process image of the cyclic tasks (Page 323).
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 7-27 "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.

7.3.4.2 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 381) 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 332) and the process image of the cyclic tasks (Page 323) 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 323) 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 7-28 "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#0008
	%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.

7.3.4.3 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 332) by directly using the identifier for the address (with implicit data type). The syntax of the identifier (Page 339) 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 339).

Note

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

7.3.4.4 Syntax for the identifier for an absolute process image access

For the absolute access to the fixed process image of the BackgroundTask (Page 338), 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 340).
- For the creation of an I/O variables for accessing the fixed process image of the BackgroundTask (Page 342).

Table 7-29 Syntax for the identifier for an absolute process image access

Data type	Syntax for		Permissible address range	
	Input	Output		
BOOL	%In.x or %IXn.x [1]	%Qn.x or %QXn.x[1]	n: x:	0 .. 63 [2] 0 .. 7
BYTE	%IBn	%QBn	n:	0 .. 63 [2]
WORD	%IWn	%QWn	n:	0 .. 63 [2]
DWORD	%IDn	%QDn	n:	0 .. 63 [2]
n = logical address x = bit number				
[1] The syntax %IXn.x or %QXn.x is not permitted when defining I/O variables. [2] For a separate process image (Page 336), the following applies: No addresses that are used in the process image of the cyclic tasks. See note below.				

Examples

Input at logic address 62, WORD data type: **%IW62**.

Output at logical address 63, bit 3, BOOL data type: **%Q63.3**.

Note

Up to Version V4.1 of the SIMOTION Kernel or the "Separate process image" (Page 336) 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" (Page 334) setting on the device.

Note

The rules for I/O addresses for direct access and the process image of the cyclical tasks (Page 326) 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 7-30 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
```

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

You can access the fixed process image of the BackgroundTask (Page 332) 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:

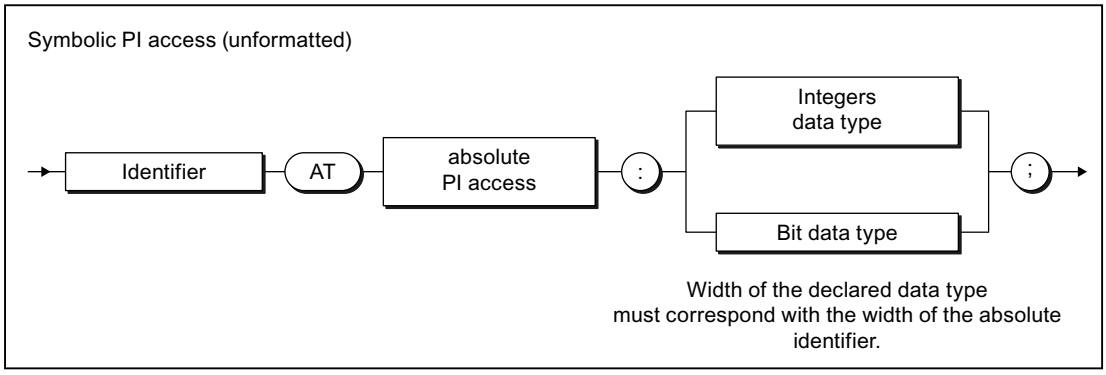


Figure 7-10 Declaration of a symbolic access to the process image

For the absolute PI access, see "Syntax for the identifier for an absolute PI access (Page 339)".

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 341)". 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 342).

7.3.4.6 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 332). 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 340).
- For the creation of an I/O variable (Page 342).

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 7-31 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 339)".

7.3.4.7 Example of symbolic PI access

If, for example, you want to access the CPU memory area (absolute PI access (Page 339)) %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.

7.3.4.8 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. 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 insert 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 detail view, select the Symbol browser tab and scroll down to the end of the variable table (empty row).
4. In the 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 339)
     (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 341).
5. Select optionally the display format used to monitor the variable in the symbol browser.

You can now access this variable using the address list 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 343).

7.3.5 Accessing I/O variables

You have created an I/O variable for:

- Direct access or process image of the cyclic tasks (Page 323).
- Access to the fixed process image of the BackgroundTask (Page 332).

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

Note

Consistency is only ensured for elementary data types.

When using arrays, the user is responsible for ensuring data consistency.

Note

If you have declared unit variables or local variables of the same name (e.g. *var-name*), specify the I/O variable using *_device.var-name* (predefined 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.

7.4 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.

Note

The same rules as for the names of program source files apply to the library names, see Insert ST source file (Page 25). 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 207). 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.

7.4.1 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 7-32 Illegal ST commands and variable access in libraries

Prohibited commands:

- *_getTaskId* function (see *SIMOTION Basic Functions* Function Manual).
- *_getAlarmId* function (see *SIMOTION Basic Functions* Function Manual).
- *_checkEqualTask* function (see *SIMOTION Basic Functions* Function Manual).
- If the library is not device-dependent (i.e. compiled without reference to a SIMOTION device or SIMOTION Kernel version):
  - System functions of SIMOTION devices (see the Parameter Manual for SIMOTION devices)
  - Version-dependent system functions

Prohibited variable accesses:

- Unit variables (retentive and non-retentive)
- Global device variables (retentive and non-retentive)
- I/O variables
- Instances of the technology objects and their system variables
- Variables of task names and configured messages (*_task* and *_alarm* namespaces, see Namespaces (Page 353), *Predefined namespaces* table)
- If the library is not device-dependent (i.e. compiled without reference to a SIMOTION device or SIMOTION Kernel version):
  - System variables of SIMOTION devices (see the Parameter Manual for SIMOTION devices)
  - Configuration data of technology objects (see Parameter Manual of configuration data for the relevant SIMOTION technology package)

Note

The **Program status** debug function is not available in libraries.

Compiling an individual library

To compile an individual library, proceed as follows:

1. Select the library in the project navigator.
2. Select the **Edit > Object Properties** menu command.
3. Select the **Technology package** 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).

Note

If the library to be compiled imports another library, note the following:

1. For the imported library, at least the same devices and SIMOTION kernel versions must be selected as for the importing library.
   Alternatively, the imported library can be compiled independently of devices if the prerequisites for this are fulfilled (refer to the *SIMOTION Basic Functions* Function Manual).
2. The imported library must already be compiled individually with reference to all configured devices, kernel versions and technology packages.
   Compilation of the library as part of a project-wide compilation is generally not sufficient.

Compiling a library as part of a project-wide compilation

When you compile the whole SIMOTION project (e.g. by choosing **Project > Save and recompile all** or by performing an XML import), the libraries used are also compiled.

Note

When performing project-wide compilation, note the following:

1. The system automatically identifies dependencies between libraries and selects the appropriate compilation sequence.
2. A library is **only** compiled with reference to the SIMOTION devices (including versions of the SIMOTION kernel) that are configured in the project and which use the library.
3. Other SIMOTION devices and kernel versions set for the library are ignored.

7.4.2 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.

7.4.3 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 353).

- 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 7-33 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
```

7.5 Use of the same identifiers and namespaces

7.5.1 Use of the same identifiers

General Information

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.

Note

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

Public 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.

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
```

7.5.2 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 348). In order to still access these hidden identifiers, you can use namespaces in certain cases.

7.5.2.1 Namespaces for libraries and technology packages

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:

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
```

Note

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.

7.5.2.2 Predefined namespaces

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 7-34 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 323). Local namespace for _device. Nesting as in _device._direct.*name* is permitted.
_project	For names of SIMOTION devices in the project; only used with technology objects on other devices. With unique project-wide names of technology objects, used also for these names and their system variables.
_task	For TaskID: The _task.*name* variable contains the TaskId of the task with the *name* identifier (see SIMOTION Basic Functions Function Manual).
_quality	As of version V4.2 of the SIMOTION Kernel: For the detailed status of I/O variables (Page 330). A value with data type DWORD is supplied. 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. 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

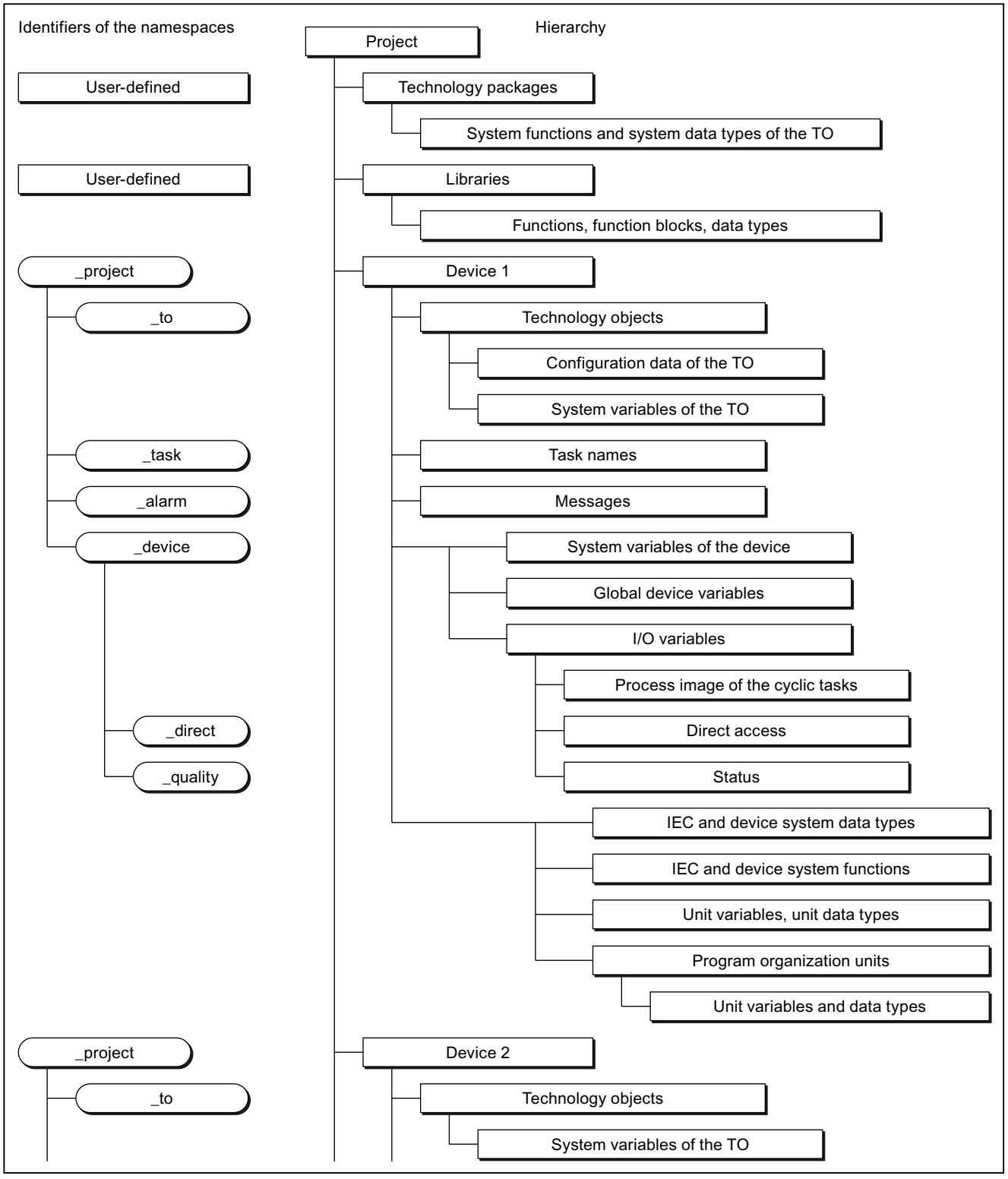


Figure 7-11 Namespaces and identifier hierarchy

7.5.2.3 Object-oriented namespace

Declaration of object-oriented namespaces

You declare object-oriented namespaces in the interface or implementation section of an ST source file or within another object-oriented namespace.

Use the following syntax:

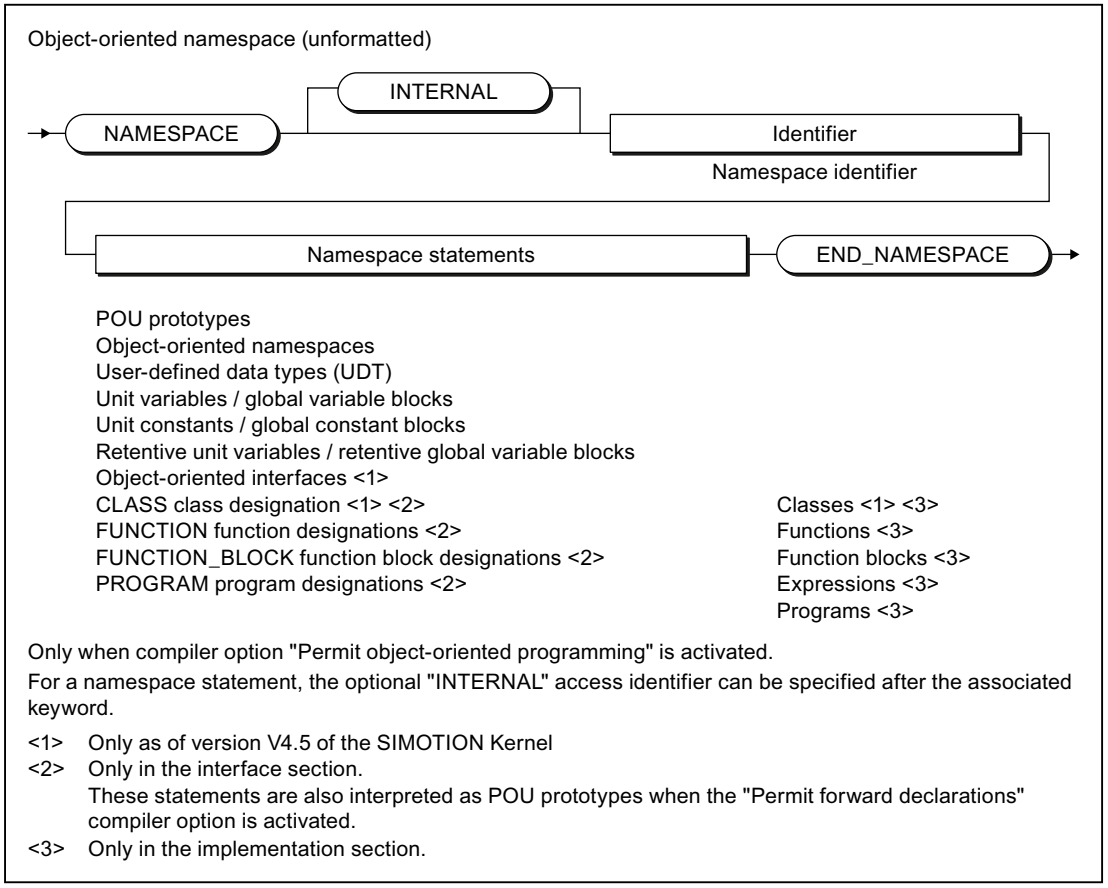


Figure 7-12 Syntax: Object-oriented namespace

Specify an identifier for the name of the namespace after the NAMESPACE and the optional INTERNAL keywords.

This is then followed by:

- The statements permitted in the namespace, e.g.:
  - Declarations of user-defined data types
  - Global variable declarations
  - Object-oriented interfaces
  - Program organization units, such as classes, functions, function blocks, programs and their prototypes
  - Other namespaces
- The END_NAMESPACE keyword

The INTERNAL keyword is possible only in namespace declarations within a namespace.

Object-oriented namespaces can be nested as required. Whereby, the nesting can be abbreviated by specifying a structured namespace path of the form *namespace_1.namespace_2*.

```
// Declaration of a nested namespace
NAMESPACE ns1
  NAMESPACE INTERNAL ns_int
    (* Elements of ns1.ns_int *)
  END_NAMESPACE
  (* Further elements of ns1 *)
END_NAMESPACE
// Declaration of a nested namespace
// by fully qualified name
// ns1 is public; INTERNAL acts only on the last name part
NAMESPACE INTERNAL ns1.ns_int
  (* Elements of ns1.ns_int *)
END_NAMESPACE
```

As for structured variables, access can be made from outside an object-oriented namespace to its elements by prefixing the namespace identifier separated with a period to the element identifier: *namespace.element*.

The INTERNAL access identifier can be specified for every element defined within a namespace (default is public). The INTERNAL access identifier immediately follows the associated keyword, e.g. VAR_GLOBAL, VAR_GLOBAL CONSTANT, TYPE, CLASS, INTERFACE, FUNCTION_BLOCK, FUNCTION, PROGRAM or NAMESPACE. Methods, type and variable declarations, as well as constants, can also be declared within INTERNAL classes and function blocks; see sections "Access identifier within classes (Page 215)" and "Access identifiers within object-oriented FBs (Page 185)". Access can be made to elements identified with INTERNAL only within the higher-level namespace.

Name conflicts for elements defined within a namespace are not permitted.

The definition of a namespace does not need to be closed. If a previously defined namespace with the same name is found, the content of the declaration is included there. This also applies when the namespace is specified with a USES statement.

Predefined namespaces (Page 353) (e.g. _task, _alarm, _device, _project) cannot be extended. Consequently, it is not permitted to use these identifiers for a namespace definition. Access per USING with these namespaces is also not possible.

If a previously defined namespace (defined in a separate source or imported from a source or library with USES or USELIB) is redeclared, the previous access identifier must be retained.

- If no access identifier is specified with a new declaration of a previously declared INTERNAL namespace, a compiler warning is issued and the namespace remains INTERNAL.
- A compiler error message will be issued if an attempt is made to declare a previously published namespace as INTERNAL subsequently.
- If the same namespace is declared in two different sources or libraries as INTERNAL and public, and these sources/libraries are used together in another source, the namespace is indicated as INTERNAL there.

This ensures that the elements of a namespace declared as INTERNAL cannot be made public subsequently.

If public program organization units (POUs) are classified in namespaces, the namespace must be specified in the interface section and in the implementation section of the source.

Access to namespaces and their elements

Within a namespace, all included elements can be used without name prefix, even when they were declared as INTERNAL. If no element was found for an identifier within the namespace, a search is made for the identifier in the enclosing scope that can also be a namespace or the global scope of the source. Access without name prefix is also possible to such found elements. This also applies to elements declared as INTERNAL in the enclosing namespace, because the lower-level namespace in which the identifier search begins always belongs to the current namespace.

In addition to the above-mentioned search in the scope, it is also possible to access elements contained in the namespaces. This requires the complete designation of the namespace written separated with a period before the element identifier: *namespace_1.namespace_11.element*. Because an external access is typically involved, no access is possible to elements declared as INTERNAL.

Table 7-35 Example: Access to elements of a class declared as INTERNAL

```
NAMESPACE ns1
  CLASS INTERNAL cl_int
    METHOD PUBLIC m_public
      // Warning: Method is INTERNAL.
      // (The complete class is INTERNAL)
      ;
    END_METHOD
    METHOD INTERNAL m_internal
      ;
    END_METHOD
  END_CLASS
  VAR_GLOBAL
    vg_clint : cl_int;
    // Instance can be created, because within ns1.
  END_VAR
END_NAMESPACE
```

```
FUNCTION_BLOCK fb_test
  VAR
    v_clint : ns1.cl_int;
    // Instance cannot be created
    // cl_int is INTERNAL in ns1.
  END_VAR
  ns1.vg_clint.m_public();
  (* Access to method m_public not possible Although the instance vg_clint from ns1 is
visible, access is not possible to its elements *)
END_FUNCTION_BLOCK
```

Table 7-36 Example: Access to elements of a namespace declared as INTERNAL

```
NAMESPACE ns1
  NAMESPACE INTERNAL ns_int
    CLASS cl
      METHOD PUBLIC   m_public   ; END_METHOD
      METHOD INTERNAL m_internal ; END_METHOD
    END_CLASS
  END_NAMESPACE
  VAR_GLOBAL
    vg_cl : ns_int.cl;
    // Instance can be created, because within ns1.
  END_VAR
END_NAMESPACE
FUNCTION_BLOCK fb_test1
  VAR
    v_cl : ns1.ns_int.cl;
    // Instance cannot be created
    // ns_int is INTERNAL in ns1.
  END_VAR
  ns1.vg_cl.m_public();
  // Access possible to method m_public
  ns1.vg_cl.m_internal();
  // Access not possible to method m_internal
END_FUNCTION_BLOCK
```

USING directive

The USING directive has the following syntax:

```
USING namespace-list;
```

The namespace list is a list of namespace identifiers each separated by a comma.

Example:

```
USING namespace_1, namespace_2;
```

The USING directive affects the scope of the source file section in which it is specified. In this scope, the public elements of the namespaces specified in the USING directive can be used without details of the namespace identifier. These elements are considered as if they were declared themselves there.

This means name conflicts between the elements from different namespaces shown with USING, and elements of the current scope are not permitted. Compiler error messages are issued.

The USING directive always acts in the scope of the source file section in which it is specified. It does not apply to external accesses in this source file section. Consequently, the USING directive is always restricted to the current source. It has no effect in other sources that the current source uses.

The namespace identifier specified on the USING directive (possibly cascaded) must reference a namespace valid at this time. A USING directive does not implicitly declare namespaces.

USING directives can be specified at the following places, possibly several times successively:

- In the interface section of the source:
  Immediately after statements for importing technology packages, libraries and other sources (USEPACKAGE, USELIB, USES).
  This allows, for example, declared elements in namespaces for libraries to be shown in the global namespace.
- For the **first** declaration of a namespace within a source:
  Immediately on the following namespace identifier itself.
- Within a program organization unit (POU), e.g. class, function block, object-oriented interface, program, method:
  Immediately before the declaration section of the POU.

Examples

Table 7-37 Example: USING directive in the interface section of a source

```
INTERFACE
  USELIB mylib;   // Link with the mylib library
  USING ns_mylib; // Show the elements of the namespace
       // ns_mylib in the global namespace of the source
  // ...
END_INTERFACE
```

Table 7-38 Example: USING directive in a namespace

```
NAMESPACE ns2
  USING ns1;
  USING ns1.ns11;
  // ...
END_NAMESPACE
```

Table 7-39 Example: USING directive in the interface section of a program organization unit

```
CLASS cl
  USING ns1;
  USING ns1.ns11;
  // ...
  METHOD m : VOID
    USING ns1.ns11.ns111;
    ;
  END_METHOD
END_CLASS
```

Note

The USING directive cannot be used on namespaces included in the namespace path of the current source file section.

Table 7-40 Example: USING directive to a higher-level namespace not possible

```
NAMESPACE ns1.ns11.ns111
  CLASS cltest
    USING ns11; // Error: ns11 is part of its own declaration path
    // ...
  END_CLASS
END_NAMESPACE
```

Note

Namespace identifiers of other USING directives are not visible within a USING directive.

Table 7-41 Example: USING directive for nested namespaces

```
NAMESPACE ns1.ns11
// ...
END_NAMESPACE
NAMESPACE ns_test
  USING ns1;
  USING ns11;     // Error:
    // Namespace ns11 not visible for USING directive
  USING ns1.ns11; // Specification must be made as such
END_NAMESPACE
```

7.6 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 361)).
- on function calls and their nesting (Program structure (Page 365))
- on the memory requirement for various data areas of the program sources (Code attributes (Page 366))

7.6.1 Cross-reference list

The cross-reference list shows all identifiers in program sources (e.g. ST source files, MCC units):

- 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.

7.6.1.1 Generating and updating a cross-reference list

Initially, the cross-reference list is generated automatically when opening. Open a cross-reference list, e.g. after selecting an ST source file or library via the **Edit > Display reference data > Cross-references** menu. After changes, the update is partly performed automatically and can always be triggered manually in the detail view via the  button. When updating via the button, a check is performed as to whether a compilation is required. If a compilation is required, this is indicated by a yellow triangle next to the button.

Update of the cross-reference list when selecting a program

The list is updated automatically when opening the selected program. The local defined identifiers are therefore up-to-date.

External identifiers in the program are not updated when opening.

The opened cross-reference list is also updated automatically when compiling the program.

Update of the cross-reference list when selecting a tree (CPU, project, library, program folder)

The cross-reference list is generated automatically when opening the first time. There is no automatic update when opened again.

Note

An error-free compilation is required for a correct, consistent display of the reference data. If required, compile the project, the CPU, the program or the library first.

7.6.1.2 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 364)".

Table 7-42 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) • 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: • Technology package *name* • std_fct = IEC library • 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
Use		Use of identifier
	CALL	Call as subprogram (static binding)
	CALL VTAB	Call of a method of the dynamic binding
	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

Column	Entry in column	Meaning
Path specification		Path specification for the SIMOTION device or program source
	Name	SIMOTION device *name*
	Name1/*Name2*	• Program source *name2* on SIMOTION device *name1* • 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
	INTERFACE	Interface section of the program source
	POE type name (i.e. CLASS *name*, FUNCTION *name*, FUNCTION_BLOCK *name* INTERFACE *name* PROGRAM *name*)	Program Organization Unit (POU) *Name* within the program source. • In an MCC chart: Additional serial numbers for the command (block numbers) • In a LAD/FBD program: Additional serial numbers of the network
	I/O address	I/O variable
	METHOD *Name_1*::*Name_2*	Method *Name_2* within the class or the function block *Name_1*
	TASK *name*	Assignment for the task *name*
	UNIT	Implementation section of the program source, additional specification as POU to be made public in the interface section of the program source
	UNIT - IMPLEMENTATION	Implementation section of the program source
	_device	Global device variable
Language		Programming language of the program source
Line/Block		• In an ST source: Line number within the program source • In an MCC or LAD/FBD source: Relative line number within the command (block) or network. **Note** The absolute line number within the program source, which you need, for example, for the trace function "Trigger to code point", is obtained as follows: – Press the **Determine program line** button. – In the dialog box, press the button **Copy program line to the clipboard**.

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.

7.6.1.3 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 364) 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.

7.6.1.4 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. Confirm with **OK**.

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.

7.6.2 Program structure

The program structure contains all the function calls and their nesting within a selected element.

You can display the program structure selectively for:

- An individual program source (e.g. ST source file, MCC unit, 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)

Proceed as follows:

1. In the project navigator, select the element for which you want to display the program structure.
2. Select the **Edit > Display reference data > Program structure** menu command.
   The cross-reference tab is replaced by the program structure tab in the detail view.

Note

The display data is updated every time the program structure is opened.

You can update the detail view of an opened program structure with the F5 key.

7.6.2.1 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 7-43 Elements of the display for the program structure

Element	Description
Base (declared POU or task used))	List separated by a comma • Identifier of the program organization unit (POU) or task The specification *Name_1::Name_2* means: Method *Name_2* within the class or the function block *Name_1*. • Identifier of the program source in which the POU or task was declared, with add-on [UNIT] • Minimum and maximum stack requirement (memory requirement of the POU or task on the local data stack), in bytes [Min, Max] • Minimum and maximum overall stack requirement (memory requirement of the POU or task on the local data stack including all called POUs), in bytes [Min, Max]
Referenced POU	List separated by a comma: • Identifier of called POU The specification *Name_1::Name_2* means: Method *Name_2* within the class or the function block *Name_1*. • Optionally: Identifier of the program source / technology package in which the POU was declared: Add-on (UNIT): User-defined program source Add-on (LIB): Library Add-on (TP): System function from technology package • with function blocks or methods only: Identifier of the instance of the function block or the higher-level class • with function blocks or methods only: Identifier of program source in which the instance of the function block or of the higher-level class was declared: Add-on (UNIT): User-defined program source Add-on (LIB): Library • Number of the line in the (compiled) source in which the POE is called; several line numbers are separated by "/".

7.6.3 Code attributes

You can find information on or the memory requirement of various data areas of the program sources under code attribute.

You can display the code attributes selectively for:

- An individual program source (e.g. ST source file, MCC unit, 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)

Proceed as follows:

1. In the project navigator, select the element for which you want to display the code attributes.
2. Select the **Edit > Display reference data > Code attributes** menu command.
   The **Cross-references** tab is now replaced by the **Code attributes** tab in the detail view.

Note

The display data is updated every time the code attributes are opened.

You can update the detail view of the opened code attributes with the F5 key.

7.6.3.1 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.

7.6.4 Reference to variables

If you have selected the identifier of a variable in the open Editor window for each programming language, you can use the other places of use over the context menu to list or to skip these variables.

You select the identifier of a variable:

- In SIMOTION ST: In the Editor window of an ST source.
- In SIMOTION MCC: In the input field on the parameter screen of an open MCC command within an MCC chart
- In SIMOTION LAD/FBD: In the Editor window of a LAD/FBD program

An identifier is recognized as a variable under the following conditions:

1. The identifier is declared as a variable. The scope of the variable includes the respective window (ST source, MCC chart, LAD/FBD program).
2. The program source is compiled.
3. The variable is selected as follows (in an open parameter screen within an MCC chart):
   – The identifier is fully marked
   or
   – The cursor is within the identifier.

Note

In arrays and structures, only the variable can be selected, not a single element.

Using the **Go to** context menu, you have the following options:

- To jump to the next local place of use:
  Select the context menu **Go to > Local use >>**.
  The next place of use of the variables within the same Editor window (ST source, MCC chart, LAD/FBD program) is selected. In an MCC chart, the corresponding MCC command opens.
- Jump to the previous local place of use:
  Select the context menu **Go to > Local use <<**.
  The previous place of use of the variables within the same Editor window (ST source, MCC chart, LAD/FBD program) is selected. In an MCC chart, the corresponding MCC command opens.
- Jump to the declaration position:
  Select the context menu **Go to > Declaration position**.
  The declaration position of the variables is selected. The corresponding program source is opened, if necessary.
- List all places of use
  Select the context menu **Go to > Places of use ...**.
  In the detailed view, all places of use of the variables within their scope (including the declaration position) are listed The structure of this list is similar to the List of cross references (Page 362).
  This is how you jump to a preferred place of use:
  – Double-click on the corresponding line.
  or
  – Place the cursor in the corresponding line and click the "Go to application" button.

7.7 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 } brackets and can contain (see figure):

- Preprocessor statements for controlling the preprocessor, see Controlling the preprocessor (Page 369).
  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 374).
  The pragmas with attributes for compiler options contained in an ST source file are evaluated by the compiler and interpreted as control statements.
- Non-assigned compiler messages, see Issuing non-assigned compiler messages (Page 378).
  Non-assigned compiler messages are issued in the "Compile/check output" tab in the detailed view.

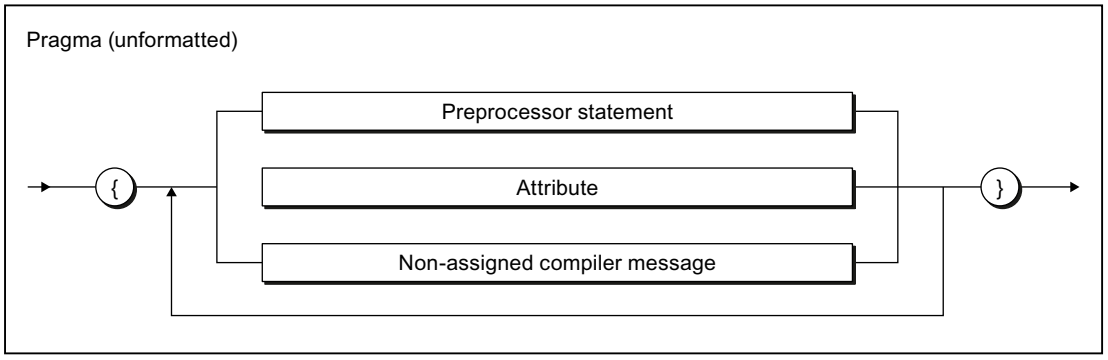


Figure 7-13 Syntax: Pragma

Note

Be sure to use the correct pragma syntax (e.g. upper- and lower-case notation of attributes).

Unrecognized pragmas are ignored with no warning message.

7.7.1 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 63)".
- Local for a program source file, see "Local compiler settings (Page 67)".

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 73). 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 374)".

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.

7.7.1.1 Preprocessor statement

You can control the preprocessor by means of statements in pragmas. The statements specified in the following syntax diagram can be used. These statements act on all subsequent lines of the ST source file.

They can be used in ST source files of a SIMOTION device or a library.

You can make definitions for the preprocessor (see Making preprocessor definitions (Page 75)) in the property dialog box of the ST source file. This enables you also to control the preprocessor with ST source files with know-how protection (see Know-how protection for ST source files (Page 75)).

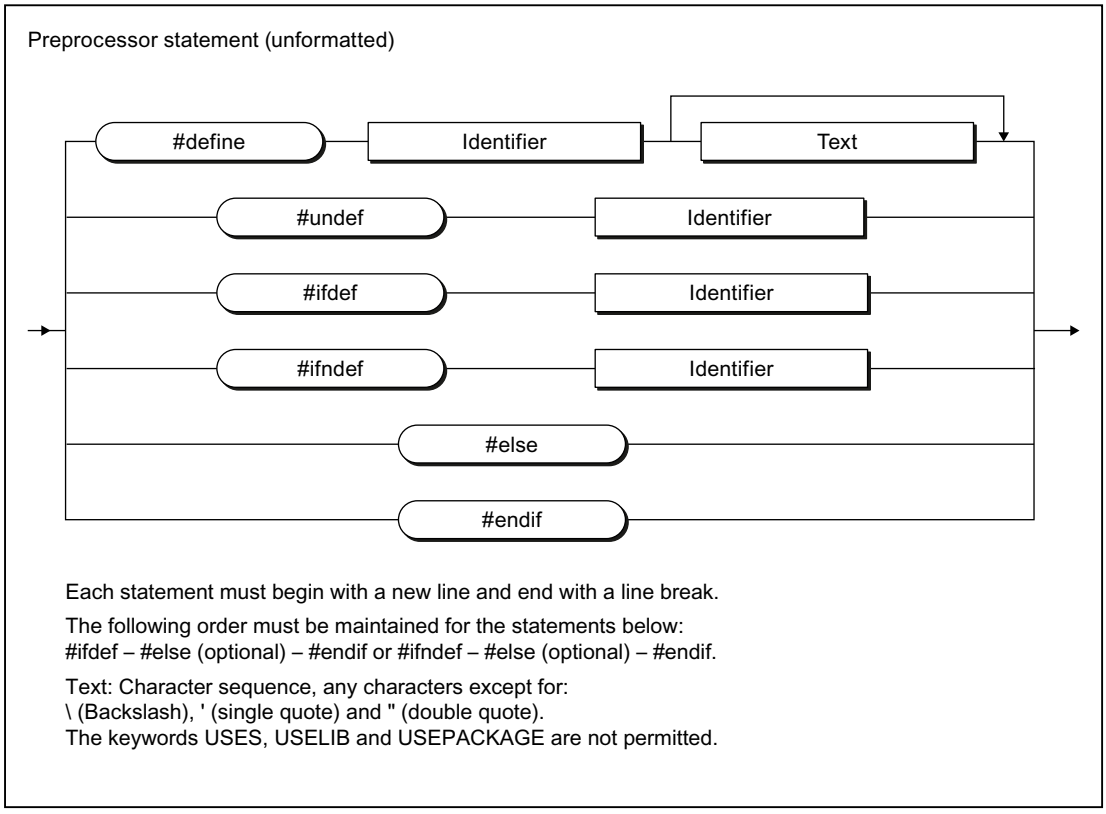


Figure 7-14 Syntax: Preprocessor statement

Table 7-44 Preprocessor statements

Statement	Meaning
#define	The specified identifier will be replaced below by the specified text. Permissible characters: See table footnote.
#undef	The replacement rule for the identifier is cancelled.
#ifdef	For variant formation (conditional compilation) If the specified identifier is defined, the following program lines (until the next pragma that contains #else or #endif) are compiled by the compiler.
#ifndef	For variant formation (conditional compilation) If the specified identifier is **not** defined, the following program lines (until the next pragma that contains #else or #endif) are compiled by the compiler.
#else	For variant formation (conditional compilation) Alternative branch to #ifdef or #ifndef. The following program lines (until the next pragma containing #endif) are compiled by the compiler, if the preceding query with #ifdef or #ifndef was not fulfilled.
#endif	Concludes variant formation with #ifdef or #ifndef.
Permissible characters: • For identifiers: In accordance with the rules for identifiers (Page 97). • 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 declared public. The defined identifiers can be used 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.

7.7.1.2 Example of preprocessor statements

Table 7-45 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
```

7.7.2 Controlling compiler with attributes

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

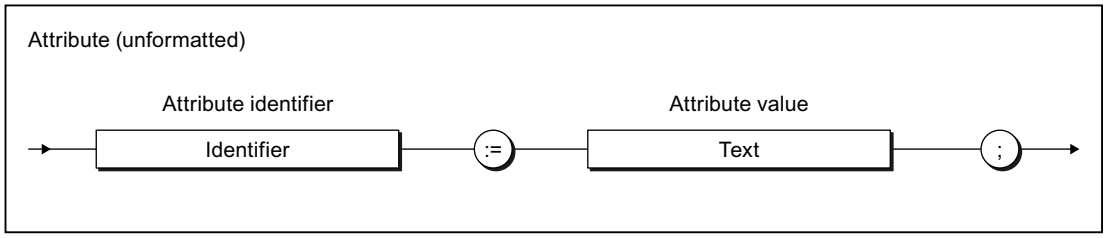


Figure 7-15 Syntax: Attributes for compiler options

Table 7-46 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*: *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*: *n* = 0 to 7: Warning class, see Meanings of the warning classes (Page 73). *n* = 16000 and higher: Number for a warning or information:
	warning:*n*:on	Warnings specified by the number *n* are displayed Permissible values for *n*: *n* = 0 to 7: Warning class, see Meanings of the warning classes (Page 73). *n* = 16000 and higher: Number for a warning or information:

Attribute identifier	Attribute value	Meaning
HMI_Export	The attribute changes the variables available on HMI devices by default. It must be placed directly after the associated keyword of the following declaration blocks: • In the interface or implementation section of an ST source file: – VAR_GLOBAL – VAR_GLOBAL RETAIN • In the declaration section of a function block provided that the compiler option (Page 63) "Permit object-oriented programming" is activated: – VAR, VAR PUBLIC, VAR PRIVATE – VAR RETAIN, VAR PRIVATE RETAIN • In the declaration section of a class (as of version V4.5 of the SIMOTION Kernel): – VAR PUBLIC – VAR, VAR PROTECTED, VAR PRIVATE – VAR RETAIN, VAR PROTECTED RETAIN, VAR PRIVATE RETAIN It affects only the unit variables declared in the associated declaration block. Detailed description of the HMI export, in particular the effect of the attribute depending on the version of the SIMOTION kernel: see Variables and HMI devices (Page 315).	
	FALSE	This attribute value is permissible: • In the interface section of an ST source file • In the following declaration blocks of a function block provided that the compiler option (Page 63) “Permit object-oriented programming" is activated: – VAR, VAR PUBLIC, VAR PRIVATE • In the following declaration blocks of a class (as of version V4.5 of the SIMOTION Kernel): – VAR PUBLIC The variables declared in the associated declaration block are **not** available on HMI devices.
	TRUE	This attribute value is permissible: • In the implementation section of an ST source file. • In the following declaration blocks of a function block provided that the compiler option (Page 63) "Permit object-oriented programming" is activated: – VAR RETAIN, VAR PRIVATE RETAIN • In the following declaration blocks of a class (as of version V4.5 of the SIMOTION Kernel): – VAR, VAR PROTECTED, VAR PRIVATE – VAR RETAIN, VAR PROTECTED RETAIN, VAR PRIVATE RETAIN The 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: • VAR_GLOBAL (in the interface and implementation section) • VAR_GLOBAL RETAIN (in the interface and implementation section) • VAR (only for programs in a unit if the "Create program instance data only once" compiler option is active). It affects only the variables declared in the associated declaration block. See also Version ID of global variables and their initialization during download (Page 309).	
	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. The attribute changes the standard definition whether the variables of the associated declaration block will be initialized for the transition to the RUN mode. It must be placed directly after the associated keyword of the following declaration blocks: • VAR_GLOBAL (in the interface and implementation section) • VAR (only for programs in a unit if the "Create program instance data only once" compiler option is active). It affects only the variables declared in the associated declaration block. See also Memory ranges of the variable types (Page 295).	
	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: Up to version V4.1 of the SIMOTION Kernel: DISABLE Version V4.2 and higher of the SIMOTION Kernel: In accordance with the setting on the device	

Note

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 7-47 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
```

7.7.3 Issue non-assigned compiler message

Within a Pragma (Page 368) you are able to prompt the computer to write non-assigned compiler messages to the "Compile/check output" tab of the detailed view.

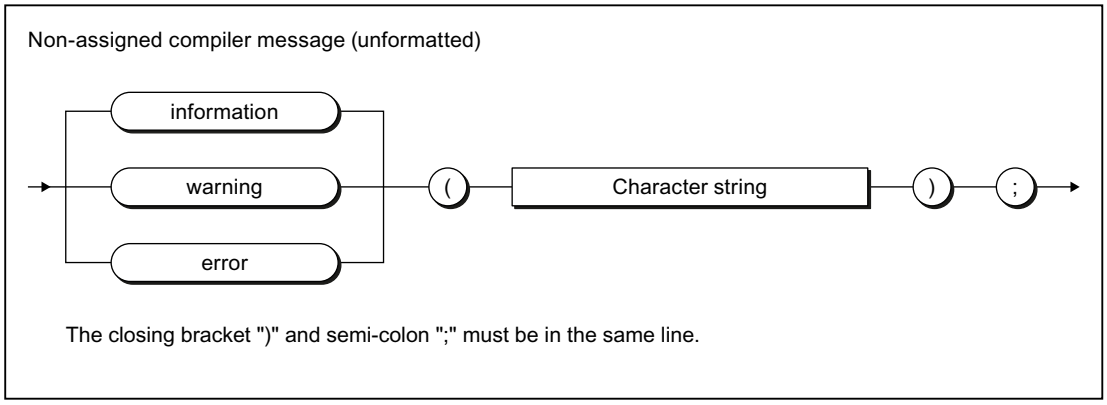


Figure 7-16 Syntax: Non-assigned compiler message

The keywords have the following meaning:

Keyword	Message number	Description
information	22002	Non-assigned information
warning	22001	Non-assigned warning
error	22000	Non-assigned error message

The text of the non-assigned message is stated after the keyword in brackets as a character string (Page 109). A semi-colon completes the statement within the pragma.

Note

These pragma statements are not included when checking the backward compatibility of the project (Menu **Project > Old project format > Check the project for backward compatibility**).

The non-assigned compiler messages can e.g. be controlled with version formation by using preprocessor statements (Page 370), see example below.

```
{ #ifdef test }
    ; // Statements
{ #else
    warning ('This message appears when "test" has not been defined');
#endif }
```

Pragmas for message output which are in hidden program sequences with the version formation are not taken into account.

7.8 SIMOTION devices

7.8.1 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 97).

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).

Note

Projects that contain device identifiers, which do not comply with the general Rules for identifiers (Page 97), 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 97) **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

7.8.2 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 336)
  - Common process image (Page 334)
- Initialization of non-retentive global variables and program data during STOP-RUN transition
- Perform time synchronization with SINAMICS drive units
- Permit OPC-UA/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. Click **OK** to confirm.

7.9 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)

The compiler option (Page 63) "Permit forward declarations" has the following effect:

- Declarations (e.g. global variable declarations) may be freely programmed between program organization units in the implementation section of an ST source file.
- The following statements within the TYPE / END_TYPE construct in the interface section (Page 258) or the implementation section (Page 260) of an ST source file are interpreted as prototypes of the relevant POU:
  - FUNCTION_BLOCK*fb-name*;
  - FUNCTION*fc-name*;
  - PROGRAM*prog-name*;
  - CLASS *class-name*; (as of version V4.5 of the SIMOTION Kernel and with compiler option (Page 63) "Permit object-oriented programming" activated)

  See below for syntax diagram.
- After the corresponding POU prototype has been declared for a class or function block, the following declarations can be programmed before the relevant POU is defined:
  - Instance declaration of the function block (Page 199) as a global variable or local variable in a program, in another function block or in a class
  - Instance declaration of the class (Page 226) as a global variable or local variable in a program, in a function block or in another class
  - Use of the function block or class as a parameter in the definition of a function block, function or method

  Instance-specific initialization is not possible in this case.

  Note

  With instance-specific initialization (only when compiler option (Page 63) "Permit object-oriented programming" is activated), the following applies:

  It is imperative that the function block or class is defined before the instance is declared or used. Declaration of the relevant POU prototype is not sufficient and therefore superfluous.

- All of the statement sections of program organization units are compiled in a second compiler run. For this reason, the following calls are possible regardless of the declaration position:
  - Call of a function block instance
  - Call of a function (Page 198)
  - Call of a method
  - Call of a program within a program (Page 207), provided that the other relevant requirements have been met.

  If a function or a program is called before it is 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.

POU prototypes

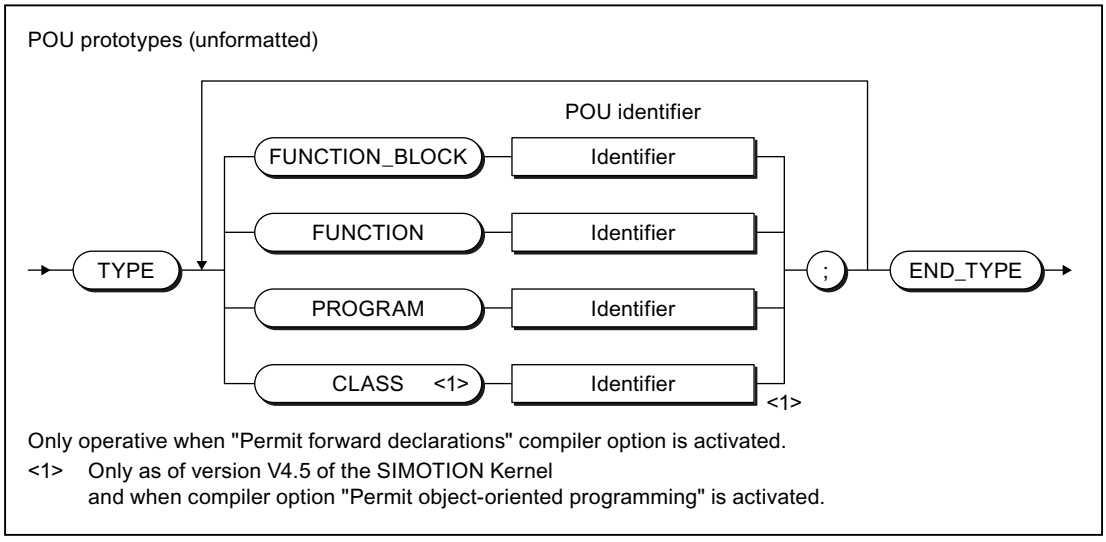


Figure 7-17 Syntax: POU prototypes

Note

If the POU prototypes are declared within the interface section, the corresponding POUs are declared public.

DECLARE PUBLIC 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 63) is not activated, the POU prototypes are ignored. Only the prototypes in the interface section are interpreted as DECLARE PUBLIC statements for the corresponding POU.

Example

Table 7-48 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
```

7.10 Jump statement and label

As additional control statement (Page 162), a jump statement is also available.

You program a jump statement (Page 175) 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:

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.

8 Error Sources and Program Debugging

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 493). Possible reactions and remedies are described for each error.

8.1 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

8.2 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 83) is recommended.

8.2.1 Operating modes for program testing

8.2.1.1 Modes of the SIMOTION devices

Various SIMOTION device operating modes are available for program testing.

Table 8-1 Operating modes of a SIMOTION device

Operating mode	Meaning
Process mode	Program execution on the SIMOTION device is optimized for maximum system performance. The following diagnostic functions are available, although they may have only restricted functionality because of the optimization for maximum system performance: • Monitor variables in the symbol browser or a watch table • Program status (only restricted): – Restricted monitoring of variables (e.g. variables in loops, return values for system functions). – Maximum of 1 program source (e.g. ST source file, MCC unit, LAD/FBD unit)[1] can be monitored. • Trace tool (only restricted) with measuring functions for drives and function generator, see online help: – Maximum of **1** trace on each SIMOTION device.
Test mode	The diagnostic functions of the process mode are available to the full extent: • Monitor variables in the symbol browser or a watch table • Program status: – Monitoring of all variables possible. – As of version V4.0 of the SIMOTION Kernel: Several program sources (e.g. ST source files, MCC units, LAD/FBD units)[1] can be monitored per task. – For version V3.2 of the SIMOTION Kernel: Maximum of 1 program source (e.g. ST source file, MCC unit, LAD/FBD unit)[1] can be monitored per task. • Trace tool with measuring functions for drives and function generator, see online help: – Maximum of **4** traces on each SIMOTION device. In addition, the following diagnostics function is available: • 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). **Note** 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: • Breakpoints Within a program source, you can set breakpoints (Page 407). When an activated breakpoint is reached, selected tasks will be stopped. • Controlling MotionTasks On the "Task Manager" tab of the device diagnostics, you can use task control commands for MotionTasks; see the SIMOTION Basic Functions Function Manual. No more than 1 SIMOTION device of the project can be switched to debug mode. SIMOTION SCOUT is in online mode, i.e. connected to the target system. Observe the following section: Important information about the life-sign monitoring (Page 390).

[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 390).
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.

8.2.1.2 Important information about the life-sign monitoring.

⚠ WARNING

Dangerous plant states possible

If problems occur in the communication link between the PC and the SIMOTION device, this may result in dangerous plant states (e.g. the axis may start moving in an uncontrollable manner).

Therefore, use the debug mode or a control panel only with the life-sign monitoring function activated with a suitably short monitoring time!

You must observe the appropriate safety regulations.

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.

Note

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

Dangerous plant states possible

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.

8.2.1.3 Life-sign monitoring parameters

Table 8-2 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: • 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. The following parameterizations are possible: • Checkbox **active**: If the checkbox is activated, life-sign monitoring is active. The deactivation of the life-sign monitoring is not always possible. • Monitoring time: Enter the timeout. **Prudence** Do not make any changes to the defaults for life-sign monitoring, if possible. Changes should only be made in special circumstances and in observance of all danger warnings.
Safety information	**Please observe the warning!** Click the button to obtain further safety information. See: Important information about the life-sign monitoring (Page 390)

8.2.2 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.

Note

If breakpoints have been activated and the SIMOTION device is in debug mode:

Entering a space switches the SIMOTION device to STOP operating mode and deactivates all outputs (ODIS).

8.2.3 Symbol Browser

8.2.3.1 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.

8.2.3.2 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 90)".
- 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

Proceed as follows:

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 8-3 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 295): • 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 function blocks whose instances are declared as unit variables. • In addition, if the program source (unit) has been compiled with the "Only create program instance data once" compiler option (Page 63): – Static variables of the programs. – 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 295): If the program source (unit) was compiled without the "Only create program instance data once" (default) compiler option (Page 63), the user memory of the task to which the program was assigned contains the following variables: • Static variables of the programs. • Static variables of the function blocks whose instances are declared as static variables of programs.	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). 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 402)).

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.

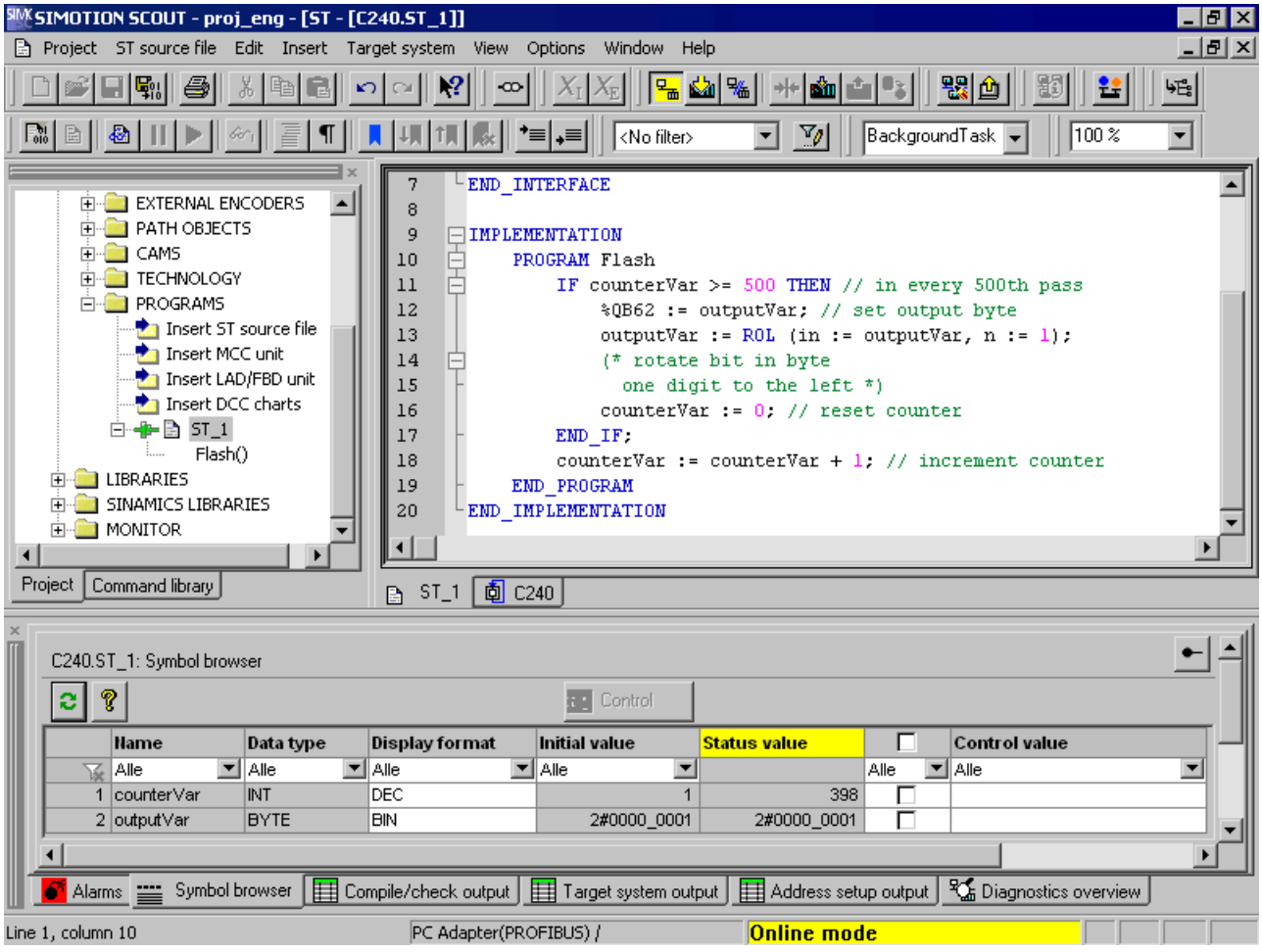


Figure 8-1 Viewing variables in the symbol browser

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
Dangerous plant states possible You assign the entered values to the variables during control. This can result in dangerous plant states, e.g. unexpected axis motion.

Note

Note when you change the values of several variables:

The values are written sequentially to the variables. It can take several milliseconds until the next value is written. The variables are changed from top to bottom in the symbol browser. There is therefore no guarantee of consistency.

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 8-4 Display invalid floating-point numbers

Display	Meaning
1.#QNAN -1.#QNAN	Invalid bit pattern in accordance with IEEE 754 (NaN Not a Number). There is no distinction between signaling NaN (NaNs) and quiet NaN (NaNq).
1.#INF -1.#INF	Bit pattern for + infinity in accordance with IEEE 754 Bit pattern for – infinity in accordance with IEEE 754
-1.#IND	Bit pattern for indeterminate

8.2.4 Monitoring variables in watch table

8.2.4.1 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.

8.2.4.2 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**
Dangerous plant states possible You assign the entered values to the variables during control. This can result in dangerous plant states, e.g. unexpected axis motion.

Note

Note when you change the values of several variables:

The values are written sequentially to the variables. It can take several milliseconds until the next value is written. The variables are changed from top to bottom in the watch table. There is therefore no guarantee of consistency.

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 8-5 Display invalid floating-point numbers

Display	Meaning
1.#QNAN -1.#QNAN	Invalid bit pattern in accordance with IEEE 754 (NaN - Not a Number). There is no distinction between signaling NaN (NaNs) and quiet NaN (NaNq).
1.#INF -1.#INF	Bit pattern for + infinity in accordance with IEEE 754 Bit pattern for – infinity in accordance with IEEE 754
-1.#IND	Bit pattern for indeterminate

8.2.5 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 90)".
- 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 394) 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

8.2.6 Program run

8.2.6.1 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 401)" 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 8-6 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, e.g. cam.pck for the library of the CAM technology package

8.2.6.2 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 8-7 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. 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 400).

8.2.6.3 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 8-8 Program run toolbar

Symbol	Meaning
	Display program run 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. See: Program run: Display code position and call path (Page 400)

8.2.7 Program status

8.2.7.1 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

Return values of the system function _trcVal are also displayed. This way interim results can be displayed in expressions. For the _trcVal function please refer to the “SIMOTION Basic Functions Function Manual”.

Note

The values of constants are not displayed.

Due to the restricted buffer capacity and the requirement for minimum runtime corruption, 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.

Method of operation of program status

On the SIMOTION device:

- While the selected monitoring range is running in the ST source file, the buffer for the variables to be monitored is filled with the corresponding values.
  The recording of the values depends on the operating mode (Page 388) (process mode or test mode) of the SIMOTION device, see following table.
- Only after leaving the selected monitoring range or abort of the recording is the buffer displayed in 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 32) 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.

Table 8-9 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 units, LAD/FBD units)	Maximum of 1 program source[1]	• As of version V4.0 of the SIMOTION Kernel: Multiple program sources[1] per task • For version V3.2 of the SIMOTION Kernel: Maximum of 1 program source[1] per task
Loops (e.g. WHILE, REPEAT, FOR)	The recording is interrupted on the first repeat loop. The values are provided. Therefore, the following applies for completely selected loops: • The appropriate values are displayed after the first run of the loop. • Changes or the values are not displayed during further runs.	If there are repeats, the recording continues correctly. Therefore, the following applies for completely selected loops: • As long as the loop is being run through, no values are displayed. • Only after leaving the monitoring range are the values 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. Values are not correctly displayed in some cases.	The recording is performed correctly during the call of the system function. Values are displayed correctly.

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

Note

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 388)).
- In version V3.2 of the SIMOTION Kernel, the programs must be assigned to various tasks.

8.2.7.2 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 67).
   - 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 63).
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 54) 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.

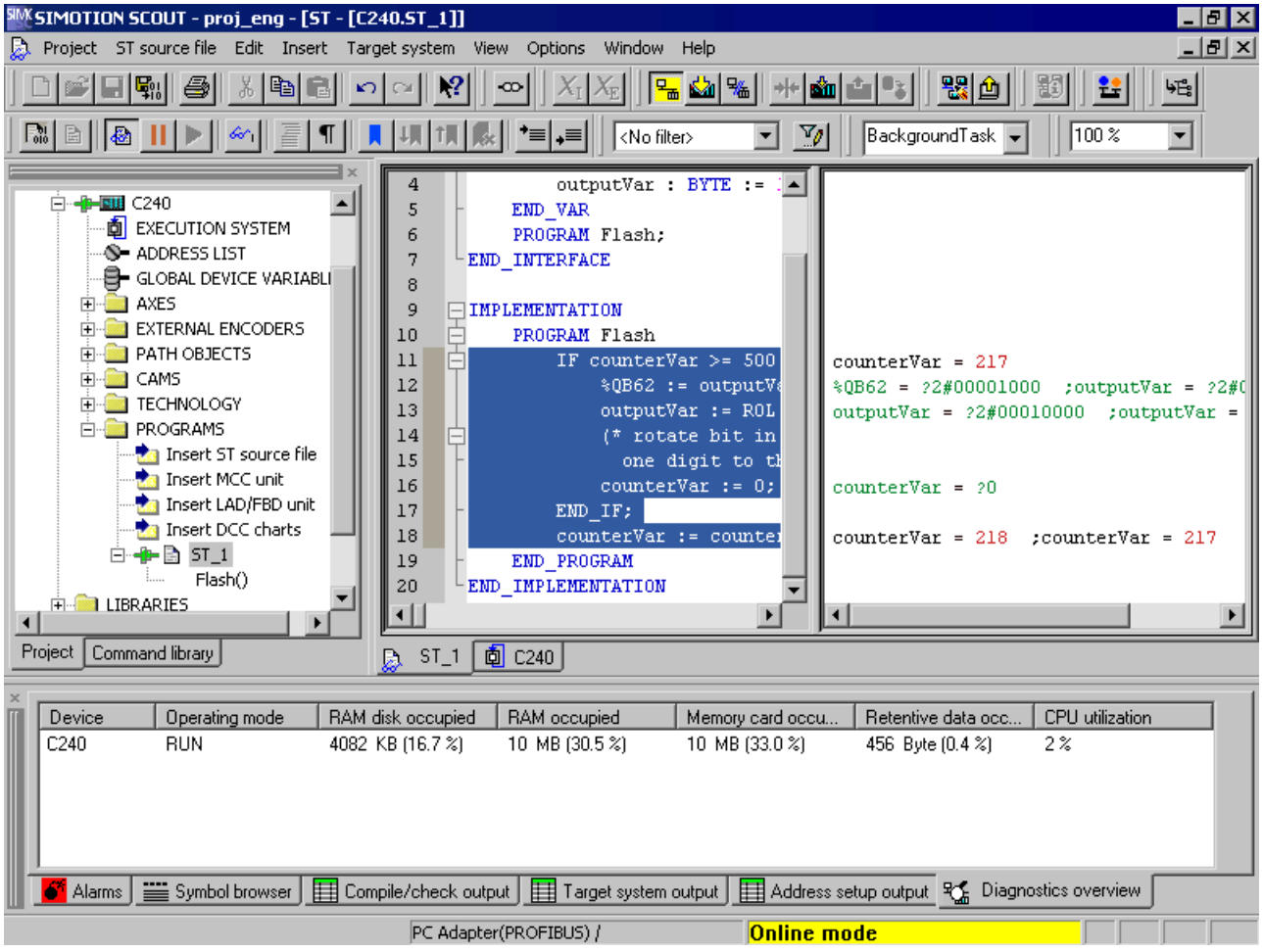


Figure 8-2 Part of an ST program in program status test mode

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 406).

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 32) 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 54) to stop the display.
- Click the button for **Continue monitoring of program variables** in the ST editor toolbar (Page 54) to continue the display.

You can force the update of the displayed values:

- Click the button for **Update** on the ST editor toolbar (Page 54).
  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.

8.2.7.3 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.

8.2.7.4 Parameter call path status program

Table 8-10 Program status call path parameter description

Field	Description
Calling task	Select the task. All tasks in which the selected code position is called are available for selection.
Current code position	The selected section of the POU (code position) is shown (with the name of the ST source file, line number, name of the POU)
is called by	Select the calling code position. The following are available: • The code positions to be called within the selected task (with the name of the program source, line number, name of the POU). If the selected calling code position is in turn called by several code positions, further lines are displayed in which you proceed similarly. • **All:** All displayed code positions are selected. Moreover, all code positions (up to the top level of the hierarchy) are selected from which the displayed code positions are called.

8.2.8 Breakpoints

8.2.8.1 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 408).
2. Specify the tasks to be stopped, see Specifying the debug task group (Page 409).
3. Set breakpoints, see Setting breakpoints (Page 412).

4. Define the call path, see Defining a call path for a single breakpoint (Page 415).
5. Activate the breakpoints, see Activating breakpoints (Page 420).

8.2.8.2 Setting the debug mode

⚠ WARNING

Dangerous plant states possible

If problems occur in the communication link between the PC and the SIMOTION device, this may result in dangerous plant states (e.g. the axis may start moving in an uncontrollable manner).

Therefore, use the debug mode only with activated life-sign monitoring (Page 390) with a suitably short monitoring time!

You must observe the appropriate safety regulations.

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 388).
4. Accept the safety information
5. Parameterize the sign-of-life monitoring.
   See also section: Important information about the life-sign monitoring (Page 390).
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 414) is displayed.

As long as no breakpoints are activated, you can edit program sources in debug mode (Page 392).

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.

Note

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 the STOP operating state.
- The outputs are deactivated (ODIS).

⚠ WARNING

Dangerous plant states possible

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.

8.2.8.3 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 408).

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.

Note

Note the different behavior when an activated breakpoint is reached, see the following table.

Table 8-11 Behavior at the breakpoint depending on the tasks to be stopped in the debug task group.

Property		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. • Active: ODIS will be deactivated. All outputs and technology objects are released. • 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 412)).
2. Define the call path (see Defining a call path for a single breakpoint (Page 415)).
3. Activate the breakpoints (see Activating breakpoints (Page 420)).

8.2.8.4 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 388).

Table 8-12 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. Assign all tasks to be stopped on reaching a breakpoint to the **Tasks to be stopped** list.
All tasks	Select this selection option if you only want to stop all user tasks. The SIMOTION device remains in STOP mode after an activated breakpoint is reached, all outputs and technology objects will be deactivated (ODIS activated). In this case, also select whether the outputs and technology objects are to be released again after resumption of program execution.
'Resume' activates the outputs (ODIS deactivated).	Only if **All tasks** is selected. Activate the checkbox, to release again the outputs and technology objects after program execution has been resumed. All outputs and technology objects can only be released after a download of the project with deactivated checkbox.

Note

Note the different behavior at the activated breakpoint depending on the tasks to be stopped, see table in Define the debug task group (Page 409).

You can only make changes to the debug task group if no breakpoints are active.

8.2.8.5 Debug table parameter

The debug table shows all breakpoints in the program sources of a SIMOTION device.

Table 8-13 Debug table parameter description

Field		Description
Debug points (table)		
	Active	The activation state of the corresponding breakpoint is displayed and can be modified: **Active**: The breakpoint is activated. **Inactive**: The breakpoint is deactivated. See: Activating breakpoints (Page 420).
	Source, line (POU)	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
	Call path	Click the button to define the call path for the breakpoint. See: Defining the call path for a single breakpoint (Page 415).
All breakpoints ...		
	Activate	Click the button to activate all breakpoints (in all program sources) of the SIMOTION device. See: Activating breakpoints (Page 420).
	Deactivate	Click the button to deactivate all breakpoints (in all program sources) of the SIMOTION device. See: Activating breakpoints (Page 420).
	Delete	Click the button to clear all breakpoints (in all program sources) of the SIMOTION device. See: Setting breakpoints (Page 412).

8.2.8.6 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 408).
4. The tasks to be stopped are specified, see Specifying the debug task group (Page 409).

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 404) 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 415).
2. Activate the breakpoints, see Activating breakpoints (Page 420).

8.2.8.7 Breakpoints toolbar

This toolbar contains important operator actions for setting and activating breakpoints:

Table 8-14 Breakpoints toolbar

Symbol	Meaning
	Set/remove breakpoint Click this icon to set at breakpoint for the selected code position or to remove an existing breakpoint. See: Setting breakpoints (Page 412).
	Activate/deactivate breakpoint Click this icon to activate or deactivate the breakpoint at the selected code position. See: Activating breakpoints (Page 420).
	Edit the call path Click this icon to define the call path for the breakpoints: • If a code position **with** breakpoint is selected: The call path for this breakpoint. • If a code position **without** breakpoint is selected: The call path for all breakpoints of the POU. See: Defining the call path for a single breakpoint (Page 415), Defining the call path for all breakpoints (Page 418).
	Activate all breakpoints of the active POU Click this symbol to activate all breakpoints in the active program source or POU (e.g. ST source file, MCC chart, LAD/FBD program). See: Activating breakpoints (Page 420).
	Deactivate all breakpoints of the active POU Click this symbol to deactivate all breakpoints in the active program source or POU (e.g. ST source file, MCC chart, LAD/FBD program). See: Activating breakpoints (Page 420).
	Remove all breakpoints of the active POU Click this symbol to remove all breakpoints from the active program source or POU (e.g. ST source file, MCC chart, LAD/FBD program). See: Setting breakpoints (Page 412).
	Debug table Click this icon to display the debug table. See: Debug table parameters (Page 411).
	Display call stack Click this icon after reaching an activated breakpoint to: • View the call path at the current breakpoint. • View the code positions at which the other tasks of the debug task group have been stopped together with their call path. See: Displaying the call stack (Page 423).
	Resume Click this icon to continue the program execution after reaching an activated breakpoint. See: Resuming program execution (Page 425), Displaying the call stack (Page 423).

Symbol	Meaning
	Next step (SIMOTION Kernel as of version V4.4) Only available for the MCC and LAD/FBD programming languages: Click this icon to resume the program execution until the next MCC command or LAD/FBD network is reached. See: Resume program execution in single steps.
	Step through the subprogram (SIMOTION Kernel as of version V4.4) Only available for the MCC programming language. Click this icon to jump to the called subprogram and stop at the first command. The subprogram must be created in the MCC or LAD/FBD programming language. See: Resume program execution in single steps.

8.2.8.8 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 408).
4. The tasks to be stopped are specified, see Specifying the debug task group (Page 409).
5. Breakpoint is set, see Setting breakpoints (Page 412).

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 420).

Note

You can use the "Display call stack (Page 423)" 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 418)

8.2.8.9 Breakpoint call path / task selection parameters

Table 8-15 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. The following are available: • **All calling locations starting at this call level** The user program will always be started when the activated breakpoint in any task of the debug task group is reached. • The individual tasks from which the POU with the selected breakpoint can be reached. The user program will be stopped only when the breakpoint in the selected task is reached. The task must be in the debug task group. The specification of a call path is possible.
Current code position	The code position is shown with the set breakpoint (with the name of the program source file, line number, name of the POU).
Is called by	Only for functions and function blocks: Select the call path, i.e. the code position to be called (in the calling POU). The following are available: • **All calling locations starting at this call level** **No** call path is specified. The user program will always be stopped at the activated breakpoint when the POU in the tasks is reached. • Only when a single task is selected: The code positions to be called within the selected task (with the name of the program source, line number, name of the POU). The call path is specified. The user program will be stopped at the activated breakpoint only when the POU is called from the selected code position. If the POU of the selected calling code position is also called from other code positions, further lines are displayed successively in which you proceed similarly.
The breakpoint will be activated at each nth pass.	If you do not want the breakpoint to be activated until the code position has been reached a certain number of times, set this number.

Note

You can only make changes to the debug task group if no breakpoints are active.

8.2.8.10 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 408).
4. The tasks to be stopped are specified, see Specifying the debug task group (Page 409).

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 420).

Note

You can use the "Display call stack (Page 423)" 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 415)

8.2.8.11 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 8-16 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. The following are available: • **All calling locations starting at this call level** The user program will always be started when an activated breakpoint of the POU in any task of the debug task group is reached. • The individual tasks from which the selected breakpoint can be reached. The user program will be stopped only when an activated breakpoint in the selected task is reached. The task must be in the debug task group. The specification of a call path is possible.
Current POU	The POU in which the cursor is located is displayed (with the name of the program source file, name of the POU).
Is called by	Only for functions and function blocks: Select the call path, i.e. the code position to be called (in the calling POU). The following are available: • **All calling locations starting at this call level** **No** call path is specified. The user program will always be stopped at an activated breakpoint when the POU in the selected tasks is called. • Only when a single task is selected: The code positions to be called within the selected task (with the name of the program source, line number, name of the POU). The call path is specified. The user program will be stopped at an activated breakpoint only when the POU is called from the selected code position. If the POU of the selected calling code position is also called from other code positions, further lines are displayed successively in which you proceed similarly.
The breakpoint will be activated at each nth pass.	If you do not want the breakpoint to be activated until the code position has been reached a certain number of times, set this number.
Apply this call path to all previous breakpoints of this POU	Click the **Apply** button, if you want to apply the call path to all previously set breakpoints of the current POU. Any existing settings will be overwritten.

8.2.8.12 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 408).
4. The tasks to be stopped are specified, see Specifying the debug task group (Page 409).
5. Breakpoints are set, see Setting breakpoints (Page 412).
6. Call paths are defined, see Defining a call path for a single breakpoint (Page 415).

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 425) 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.

The following applies up to version V4.3 of the SIMOTION Kernel:

- In the case of activated breakpoints, the "Single step" test function of the SIMOTION MCC programming language cannot be used.

The following applies as of version V4.4 of the SIMOTION Kernel:

- The "Single step" test function of the SIMOTION MCC programming language is not available in the Debug mode.

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 415)), 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 409)". The breakpoint is highlighted.

In the Task status function bar, (Page 425) the task in which the breakpoint was reached is highlighted in red ( ).

The following applies to the programming languages MCC or LAD/FBD: If the debug task group is stopped by a breakpoint, then the user has the option to change to another task, belonging to the debug task group, in the combo box. Always the breakpoint of the currently selected task is visualized.

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 394).

You can use the "Display call stack (Page 423)" 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

You can resume the execution of the stopped tasks, see "Resuming program execution" (Page 425).

As of version V4.4 of the SIMOTION Kernel, you can resume the task in single steps that has been stopped at the activated breakpoint in the MCC and LAD/FBD programming languages, see Resuming program execution in single steps.

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.

8.2.8.13 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 409) 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 400)".

8.2.8.14 Breakpoints call stack parameter

When an activated breakpoint (Page 420) is reached, you can display the following for each task in the debug task group (Page 409):

- 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 8-17 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. 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 400)".

8.2.8.15 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 (Page 414) to "Continue".

The stopped task is continued until the next active breakpoint is reached.

8.2.9 Task status function bar

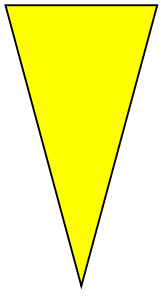
In a combo box, the Task status function bar displays all tasks of the active SIMOTION device, to which a program is assigned.

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 8-18 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 420).	
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 420) 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

A selection of a task in the combo box is only possible:

- For the following test functions of the SIMOTION MCC programming language:
  - Monitoring
  - Single step
  - Trace
- at activated breakpoints (Page 420) in the MCC or LAD/FBD programming languages.

8.2.10 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.

8.2.11 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.

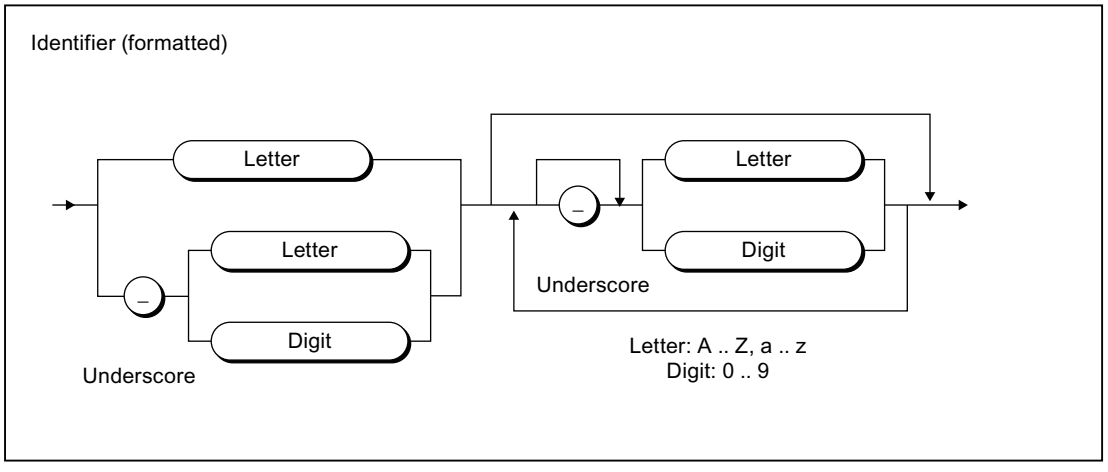


Figure A-1 Example of a lexical rule

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.

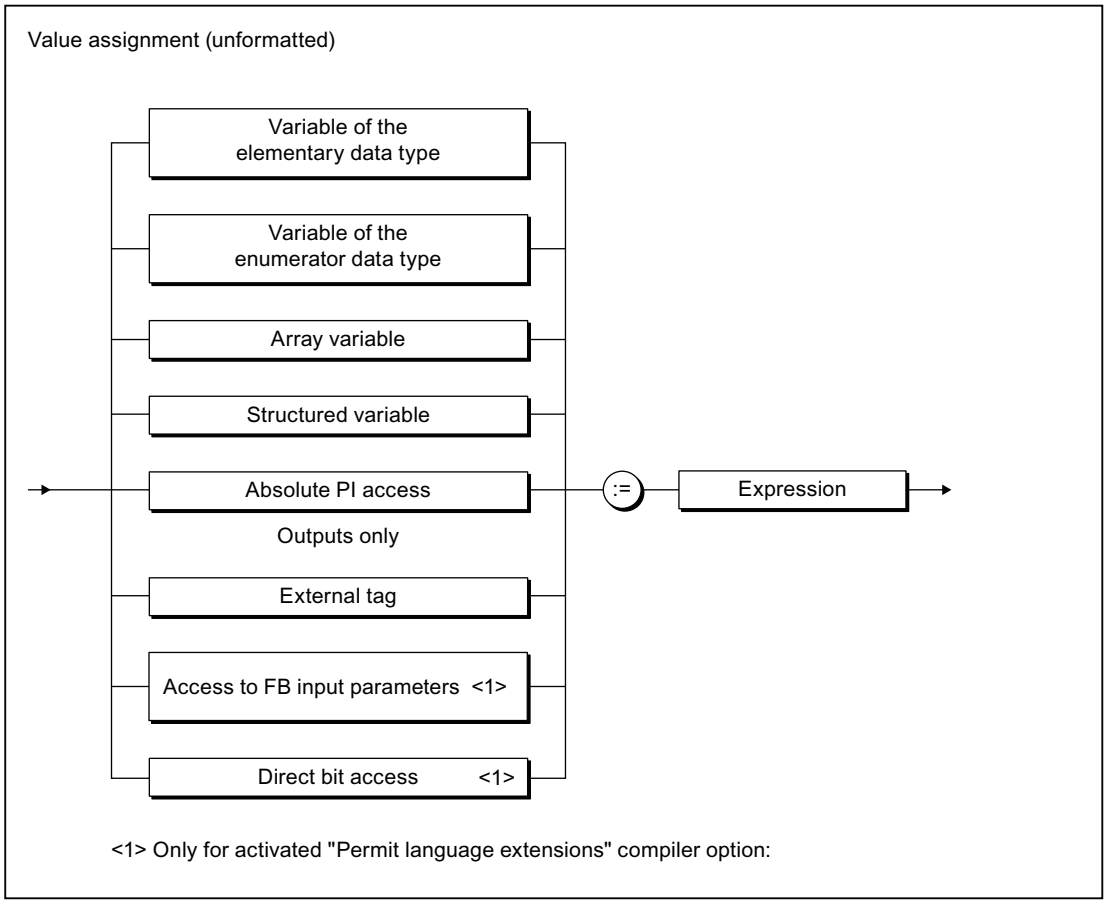


Figure A-2 Example of a syntactic rule

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 427) 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 441)).

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

Characters	Description	Lexical rule
//	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 95)

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

Characters	Code for	Lexical rule
TIME#	Time information	Duration
TIME_OF_DAY#	Time information	Time of day
TOD#	Time information	Time of day
d	Separator for time interval (day)	Days (rule: Sequence representation)
h	Separator for time interval (hours)	Hours (rule: Sequence representation)
m	Separator for time interval (minutes)	Minutes (rule: Sequence representation)
ms	Separator for time interval (milliseconds)	Milliseconds (rule: Sequence representation)
s	Separator for time interval (seconds)	Seconds (rule: Sequence representation)

A.1.2.4 Predefined identifiers for process image access

Below is a list of all predefined variables in ST that you can use to access CPU memory areas (absolute identifiers). Note that you can read and write outputs but you can only read inputs.

Table A-5 Absolute identifier

Identifier	Description	Lexical rule
%In.x or **%IXn.x**	CPU input range with byte and bit address	Absolute PI access
%IBn	CPU input range with byte address	Absolute PI access
%IWn	CPU input range with word address	Absolute PI access
%IDn	CPU input range with double word address	Absolute PI access
%Qn.x or **%QXn.x**	CPU output range with byte and bit address	Absolute PI access
%QBn	CPU output range with byte address	Absolute PI access
%QWn	CPU output range with word address	Absolute PI access
%QDn	CPU output range with double word address	Absolute PI access

A.1.2.5 Operators

Below is a list of all ST operators and the syntactic rules in which they are used.

Table A-6 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, instance declaration
+, –	Arithmetic operators: Unary operators, sign	Expression, exponent
+, –, *, / **MOD**	Basic arithmetic operators	Expression, basic arithmetic operator
**	Arithmetic operators: Power operator	Expression
NOT	Logic operators: Negation	Expression, operand
AND, &, OR, XOR	Basic logic operators	Basic logic operator
<, >, <=, >=, =, <>	Relational operator	Relational operator
=>	Assignment operator	Output assignment
?= [1]	Dynamic type conversion	Dynamic type conversion
^	Dereferencing operator	Dereferencing of general references

[1] Only with compiler option "Permit object-oriented programming".

A.1.2.6 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-7 ST keywords and predefined identifiers in the basic ST system

Keyword/identifier	Description	Rule
ABS	Standard numeric function	Function call
ABSTRACT [2]	Keyword	Class (Page 455), Method in class (Page 456)
ACOS	Standard numeric function	Function call
AND	Logic operator	Basic logic operator
ANYOBJECT	General data type for technology objects	TO data type

Keyword/identifier	Description	Rule
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	Keyword for address specification	Component declaration with relative address, symbolic PI access
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 (Page 489)
BYTE	Elementary data type	Bit data type
BYTE_TO_BOOL	Standard function for type conversion	Function call
BYTE_TO_DINT	Standard function for type conversion	Function call
BYTE_TO_DWORD	Standard function for type conversion	Function call
BYTE_TO_INT	Standard function for type conversion	Function call
BYTE_TO_SINT	Standard function for type conversion	Function call
BYTE_TO_UDINT	Standard function for type conversion	Function call
BYTE_TO_UINT	Standard function for type conversion	Function call
BYTE_TO_USINT	Standard function for type conversion	Function call
BYTE_TO_WORD	Standard function for type conversion	Function call
BYTE_VALUE_TO_LREAL	Standard function for type conversion	Function call
BYTE_VALUE_TO_REAL	Standard function for type conversion	Function call
CASE	Introduces a control statement for selection	CASE statement
CLASS [2]	Introduces the class	Class (Page 455)
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
CONTINUE [1]	Jump directly to the beginning of a loop	CONTINUE statement

Keyword/identifier	Description	Rule
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 data type
DATE_AND_TIME	Elementary data type for date and time	Time data 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 .. 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
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 (Page 489), WHILE statement (Page 489), WAITFORCONDITION statement (Page 491)
DT	Shorthand notation for DATE_AND_TIME	Time data 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

Keyword/identifier	Description	Rule
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_CLASS [2]	Ends class	Class (Page 455)
END_EXPRESSION	Ends the EXPRESSION statement	Expression (Page 454)
END_FOR	Ends the FOR statement	FOR statement (Page 489)
END_FUNCTION	Ends the function	Function (Page 454)
END_FUNCTION_BLOCK	Ends the function block	Function block (Page 454)
END_IF	Ends the IF statement	IF statement
END_IMPLEMENTATION	Ends implementation section	Implementation section
END_INTERFACE	Ends interface section, ends object-oriented interface [2]	Interface section, object-oriented interface (Page 457)
END_LABEL	Ends the LABEL statement	Jump label declaration (Page 468)
END_METHOD [2]	Ends method	Method in class (Page 456), Method in FB (Page 458), Method prototype (Page 457)
END_NAMESPACE [2]	Ends the namespace	Namespace
END_PROGRAM	Ends the program	Program (Page 455)
END_REPEAT	Ends the REPEAT statement	REPEAT statement (Page 490)
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 (Page 491)
END_WHILE	Ends the WHILE statement	WHILE statement (Page 490)
ENUM_TO_DINT	Standard function for type conversion	Function call
EXIT	Direct exit from loop execution	EXIT statement (Page 490)
EXP	Standard numeric function	Function call
EXPD	Standard numeric function	Function call
EXPRESSION	Programmable event for waiting task	Expression (Page 454)
EXPT	Standard numeric function	Function call
EXTENDS [2]	Derivation of a class	Class (Page 455)
F_TRIG	Detects falling edge	Function block call
FALSE	Predefined Boolean constant: Logical condition false, value equal to 0	–

Keyword/identifier	Description	Rule
FINAL [2]	Keyword	Class (Page 455), Method in class (Page 456)
FIND	Standard function for string editing	Function call
FOR	Introduces the control statement for loop execution	FOR statement (Page 489)
FROM_BIG_ENDIAN [1]	Standard function for endian conversion	Function call
FROM_LITTLE_ENDIAN [1]	Standard function for endian conversion	Function call
FUNCTION	Introduces the function	Function (Page 454)
FUNCTION_BLOCK	Introduces the function block	Function block (Page 454)
GOTO	Jump	GOTO statement (Page 491)
IF	Introduces a control statement for selection	IF statement
IMPLEMENTATION	Introduces the implementation section	Implementation section
IMPLEMENTS [2]	Implements object-oriented interfaces	Class (Page 455)
INSERT	Standard function for string editing	Function call
INT	Elementary data type for single precision integer with value range -2**15.. 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 interface section, introduces object-oriented interface [2]	Interface section, object-oriented interface (Page 457)
IS_VALID [1]	Standard validity check function	Function call
LABEL	Definition of jump labels	Jump label declaration (Page 468)
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
LOWER_BOUND [1]	Standard function for lower array boundary	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

Keyword/identifier	Description	Rule
LREAL_TO_INT	Standard function for type conversion	Function call
LREAL_TO_REAL	Standard function for type conversion	Function call
LREAL_TO_SINT	Standard function for type conversion	Function call
LREAL_TO_STRING	Standard function for type conversion	Function call
LREAL_TO_UDINT	Standard function for type conversion	Function call
LREAL_TO_UINT	Standard function for type conversion	Function call
LREAL_TO_USINT	Standard function for type conversion	Function call
LREAL_VALUE_TO_BOOL	Standard function for type conversion	Function call
LREAL_VALUE_TO_BYTE	Standard function for type conversion	Function call
LREAL_VALUE_TO_DWORD	Standard function for type conversion	Function call
LREAL_VALUE_TO_WORD	Standard function for type conversion	Function call
MAX	Standard function for selection	Function call
METHOD [2]	Introduces the method	Method in class (Page 456), Method in FB (Page 458), Method prototype (Page 457)
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
NAMESPACE [2]	Introduces the namespace	Namespace
NULL	Invalid reference	-
OF	Keyword	ARRAY data type specification, CASE statement
OR	Logic operator	Basic logic operator
OVERLAP	Introduction for a structure with overlapping address ranges	STRUCT data type specification
OVERRIDE [2]	Keyword	Method in class (Page 456), Static variable block, Retentive local variable block
PRIVATE [2]	Access identifier	Class access identifier (Page 456), FB access identifier (Page 459)
PROGRAM	Introduces the program	Program (Page 455)
PROTECTED [2]	Access identifier	Class access identifier (Page 456)
PUBLIC [2]	Access identifier	Class access identifier (Page 456), FB access identifier (Page 459)
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

Keyword/identifier	Description	Rule
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
REF [2]	Formation of general references	General reference
REF_TO [2]	Declaration of general references	General reference
REPEAT	Introduces the control statement for loop execution	REPEAT statement (Page 490)
REPLACE	Standard function for string editing	Function call
RETAIN	Declaration of buffered variables	Retentive variable block
RETURN	Control statement for returning from subprogram	RETURN statement (Page 490)
RIGHT	Standard function for string editing	Function call
ROL	Bit string standard functions	Function call
ROR	Bit string standard functions	Function call
RS	Bistable function block (priority reset)	Function block call
RTC	Real-time clock	Function block call
SEL	Standard function for selection	Function call
SHL	Bit string standard functions	Function call
SHR	Bit string standard functions	Function call
SIN	Standard numeric function	Function call
SINT	Elementary data type for short integer with value range -128 .. 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

Keyword/identifier	Description	Rule
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	–
SUPER [2]	Keyword for calling the method of the base class	Class method call
TAN	Standard numeric function	Function call
THEN	Introduces subsequent actions if condition true	IF statement
THIS [2]	Keyword for calling the currently valid method	Class method call, FB method call
TIME	Elementary data type for time information	Time data type
TIME_OF_DAY	Elementary data type for time of day	Time data 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 (Page 489)
TOD	Shorthand notation for TIME_OF_DAY	Time data type
TOF	OFF delay	Function block call
TON	ON delay	Function block call
TO_BIG_ENDIAN [1]	Standard function for endian conversion	Function call
TO_LITTLE_ENDIAN [1]	Standard function for endian conversion	Function 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 .. 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

Keyword/identifier	Description	Rule
UINT	Elementary data type for unsigned single precision integer with value range 0 .. 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 (Page 490)
UPPER_BOUND [1]	Standard function for upper array boundary	Function call
USELIB	Introduces the library name	–
USEPACKAGE	Introduces the package name	–
USES	Introduces a reference to other units	–
USING [2]	Use of a namespace	Namespace
USINT	Elementary data type for unsigned short integer with value range 0 .. 255	Numeric data type
USINT_TO_BYTE	Standard function for type conversion	Function call
USINT_TO_DINT	Standard function for type conversion	Function call
USINT_TO_DWORD	Standard function for type conversion	Function call
USINT_TO_INT	Standard function for type conversion	Function call
USINT_TO_LREAL	Standard function for type conversion	Function call
USINT_TO_REAL	Standard function for type conversion	Function call
USINT_TO_SINT	Standard function for type conversion	Function call
USINT_TO_UDINT	Standard function for type conversion	Function call
USINT_TO_UINT	Standard function for type conversion	Function call
USINT_TO_WORD	Standard function for type conversion	Function call
USINT_VALUE_TO_BOOL	Standard function for type conversion	Function call
VAR	Introduces a declaration block for local variables 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 (Page 454)

Keyword/identifier	Description	Rule
WAITFORCONDITION	Introduces the control statement for a task waiting for a programmable event	WAITFORCONDITION statement (Page 491)
WHILE	Introduces the control statement for loop execution	WHILE statement (Page 490)
WITH	Use in conjunction with WAITFORCONDITION	WAITFORCONDITION statement (Page 491)
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

1 Only with active compiler option "Permit language extensions, IEC61131 3rd edition".

2 Only with active compiler option "Permit object-oriented programming".

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

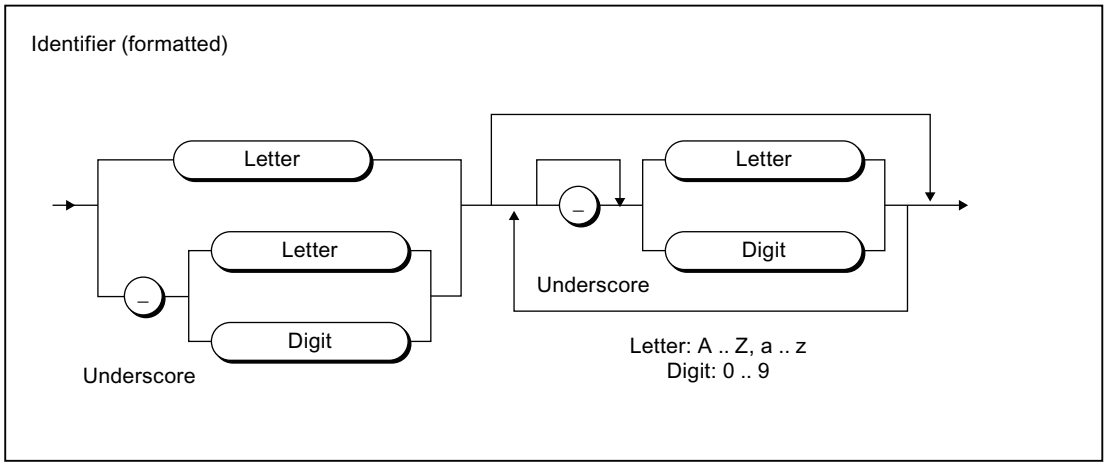


Figure A-3 Identifier

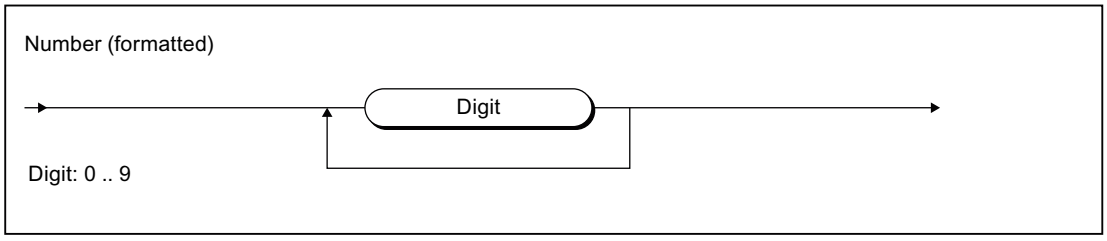


Figure A-4 Number

A.1.3.2 Notation for constants (literals)

Literals

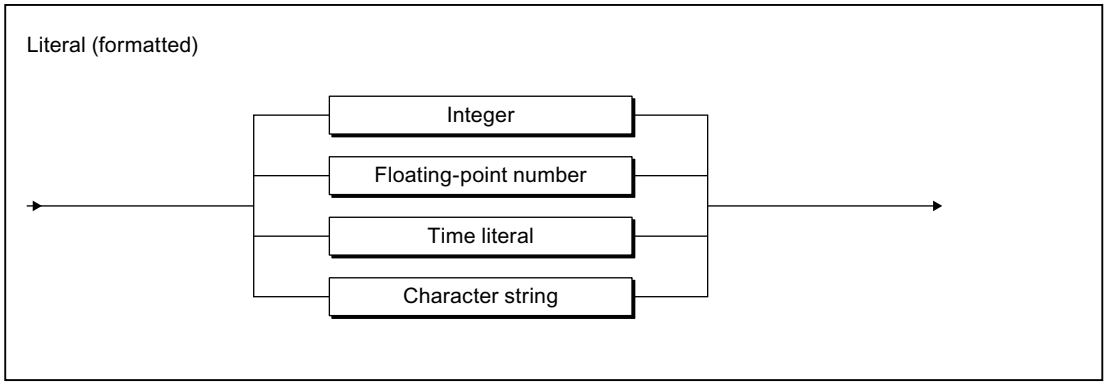


Figure A-5 Literal

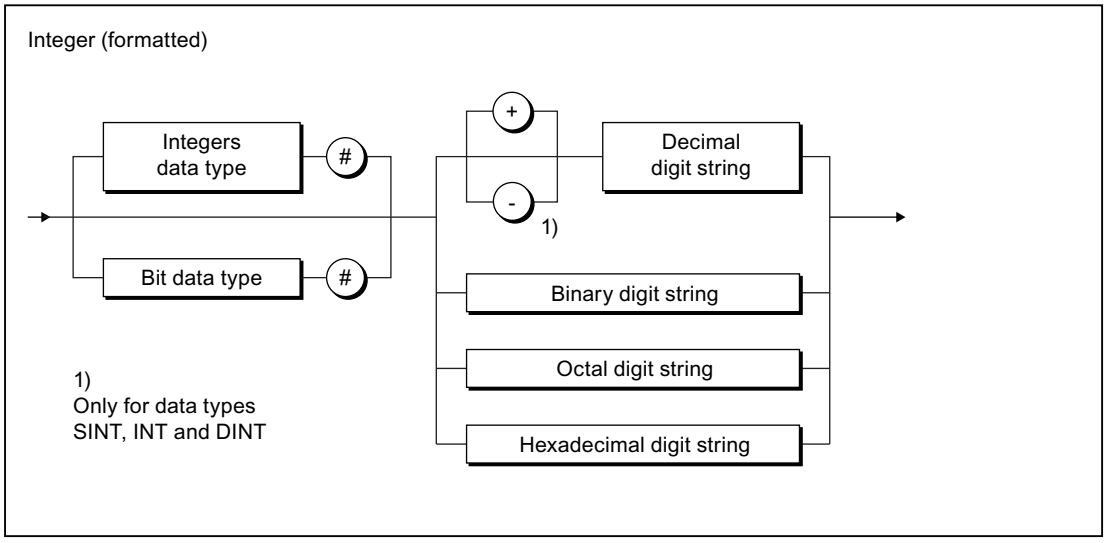


Figure A-6 Integer

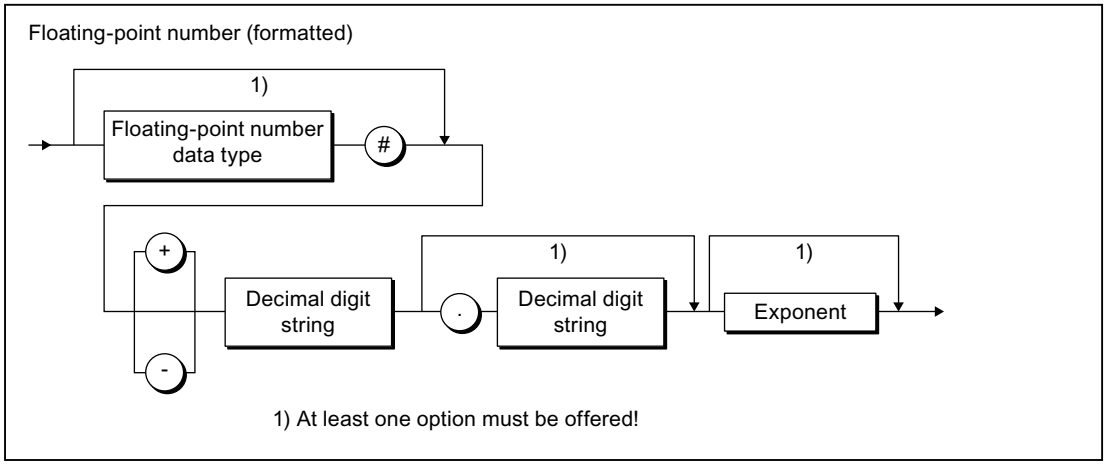


Figure A-7 Floating-point number

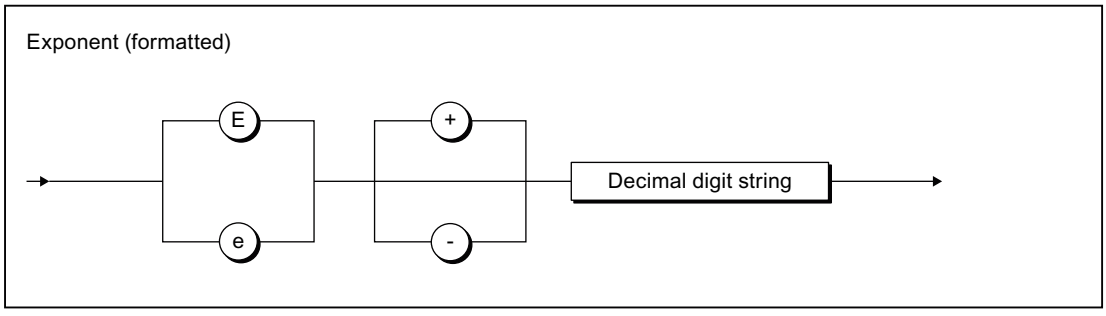


Figure A-8 Exponent

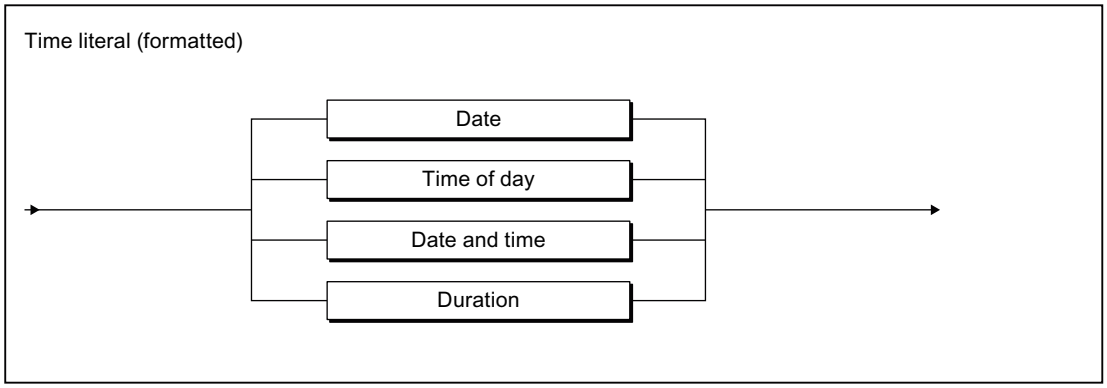


Figure A-9 Time literal

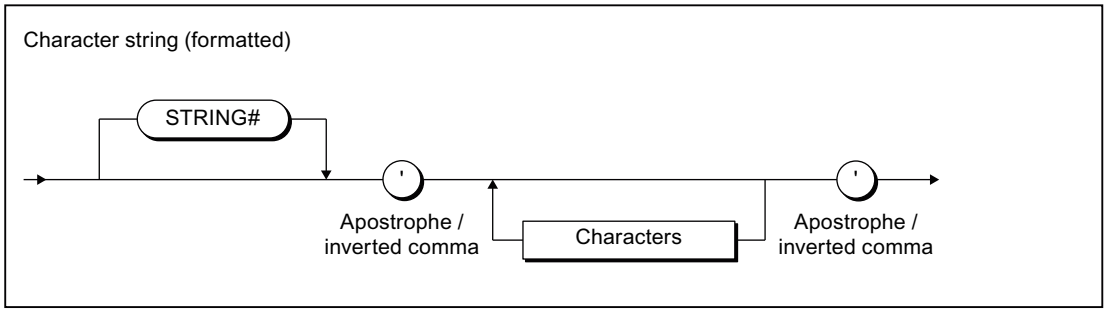


Figure A-10 Character string

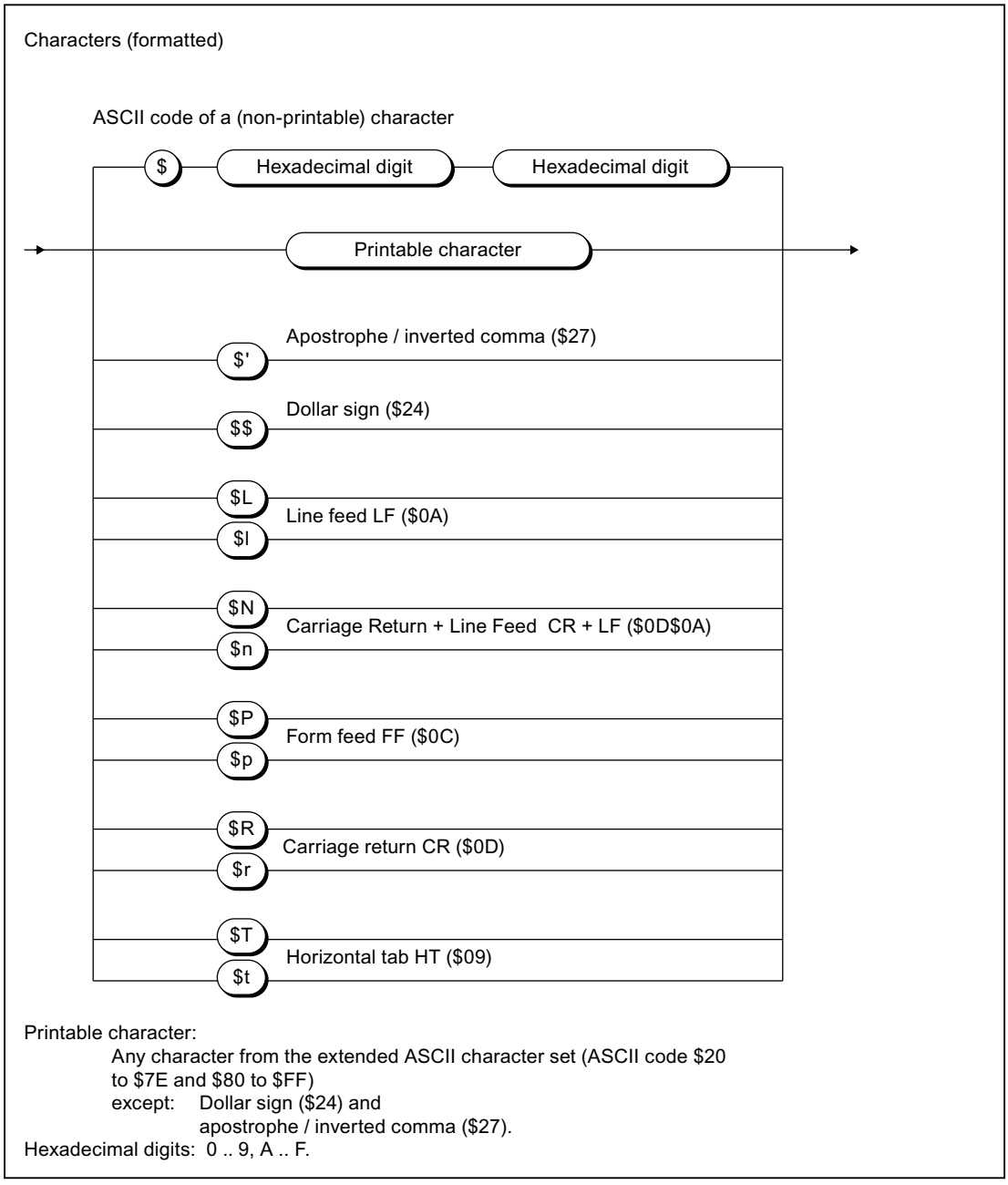


Figure A-11 Character

Digit string

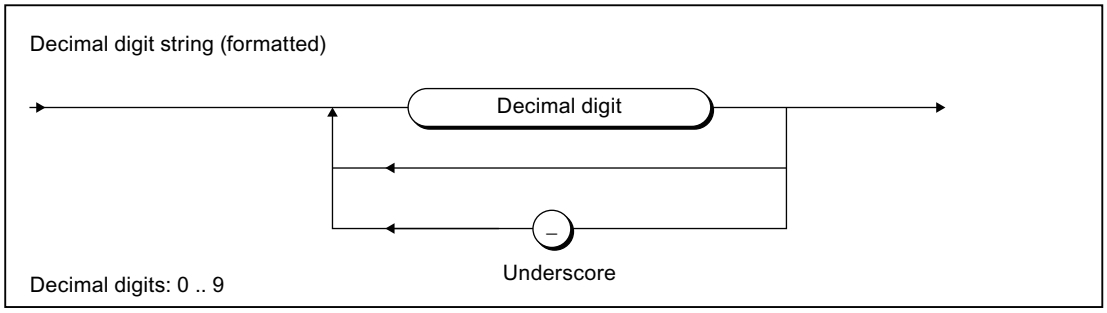


Figure A-12 Decimal digit string

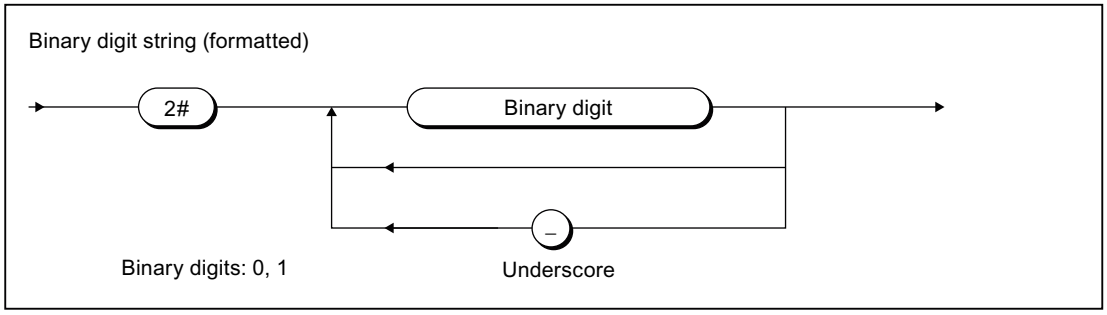


Figure A-13 Binary digit string

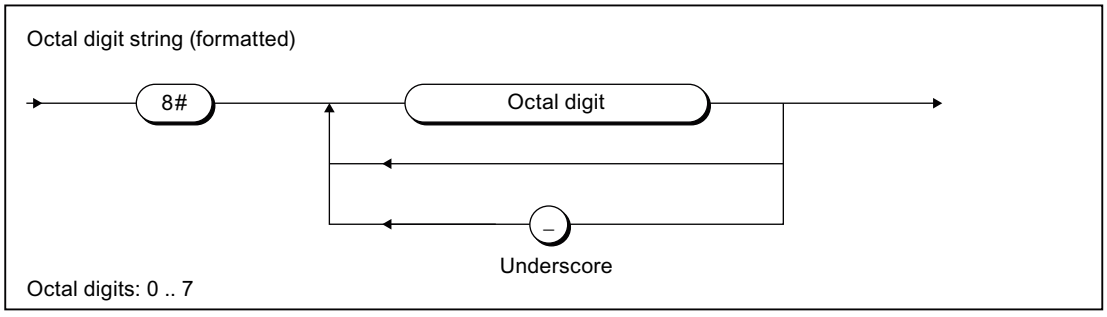


Figure A-14 Octal digit string

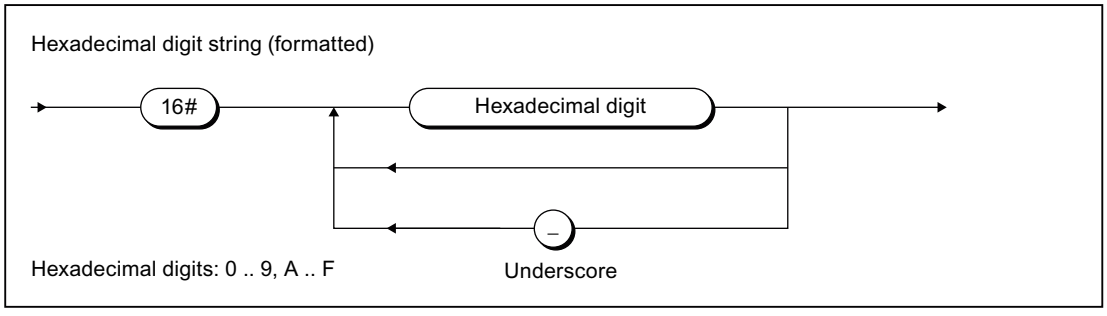


Figure A-15 Hexadecimal digit string

Date and time

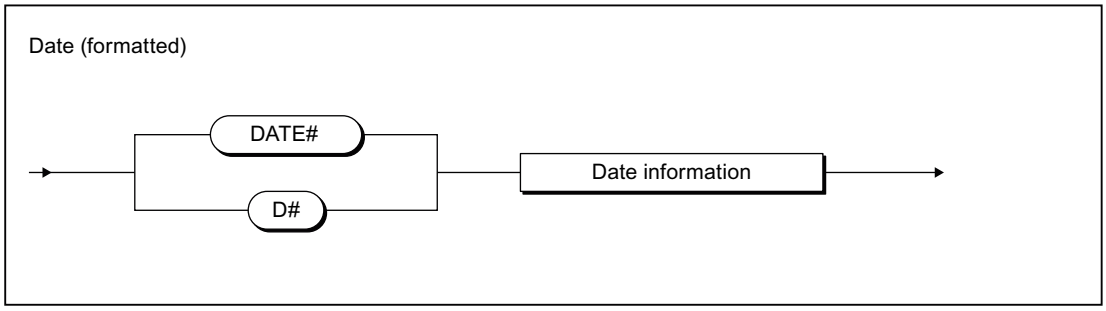


Figure A-16 Date

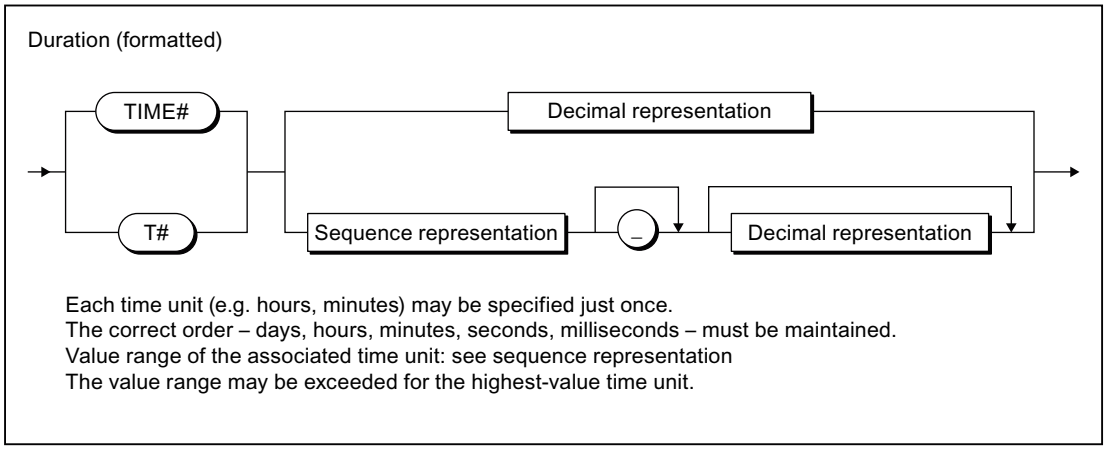


Figure A-17 Time

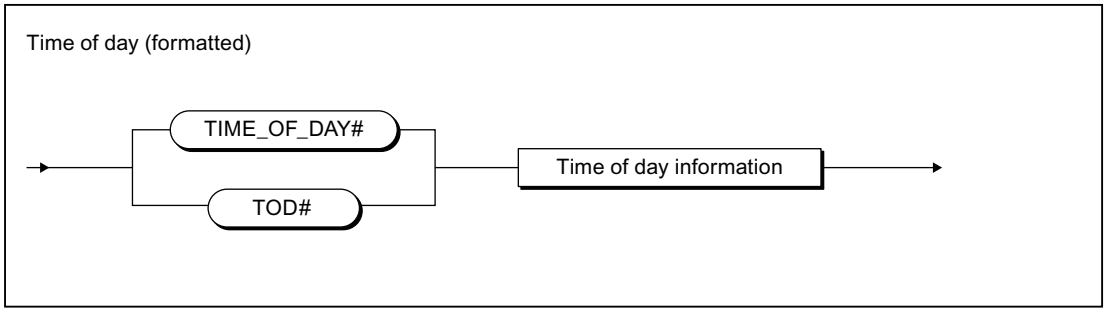


Figure A-18 Time

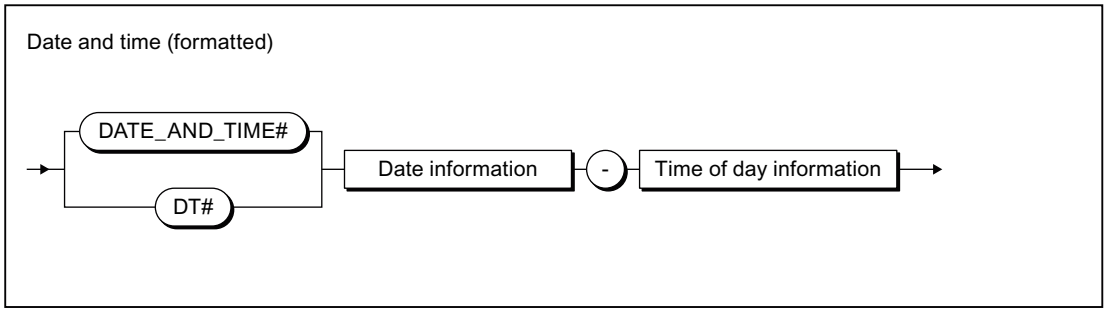


Figure A-19 Date and time

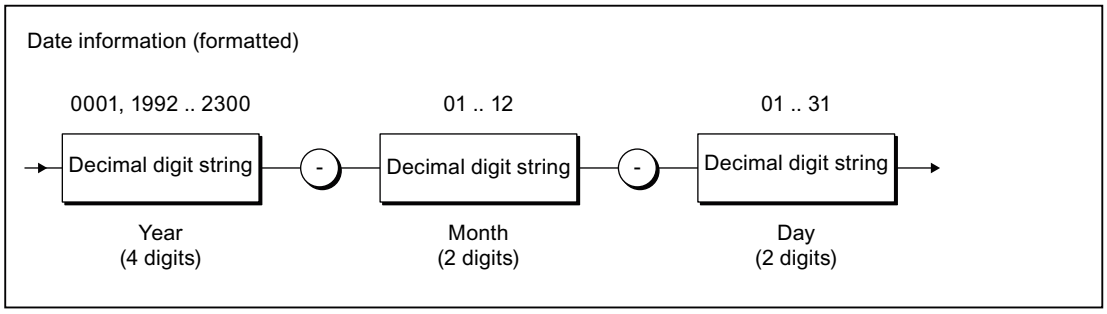


Figure A-20 Date information

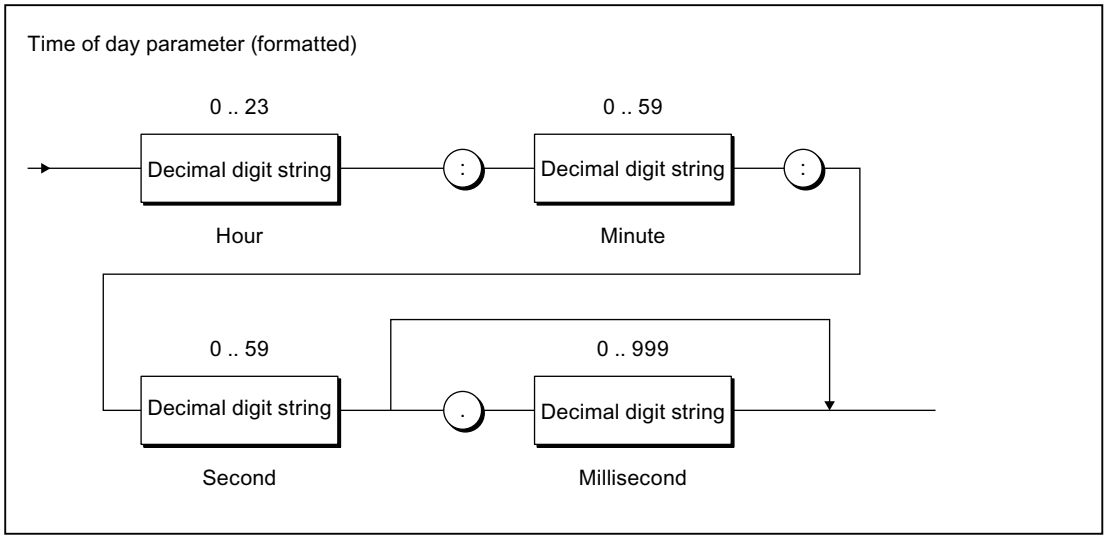


Figure A-21 Time of day information

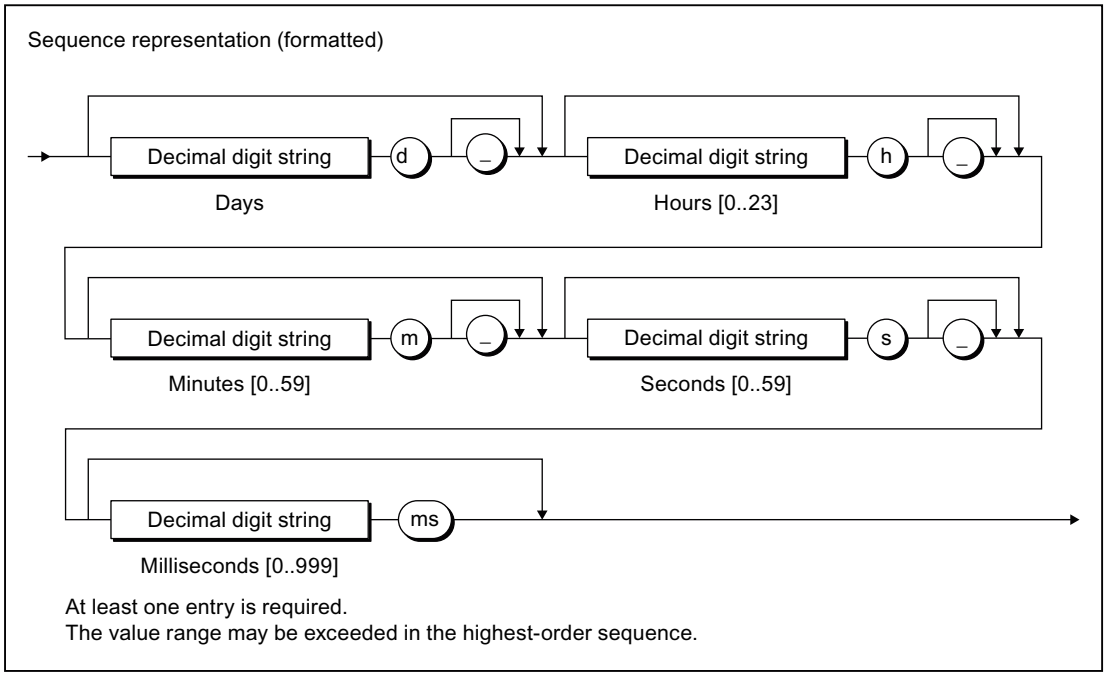


Figure A-22 Sequence representation

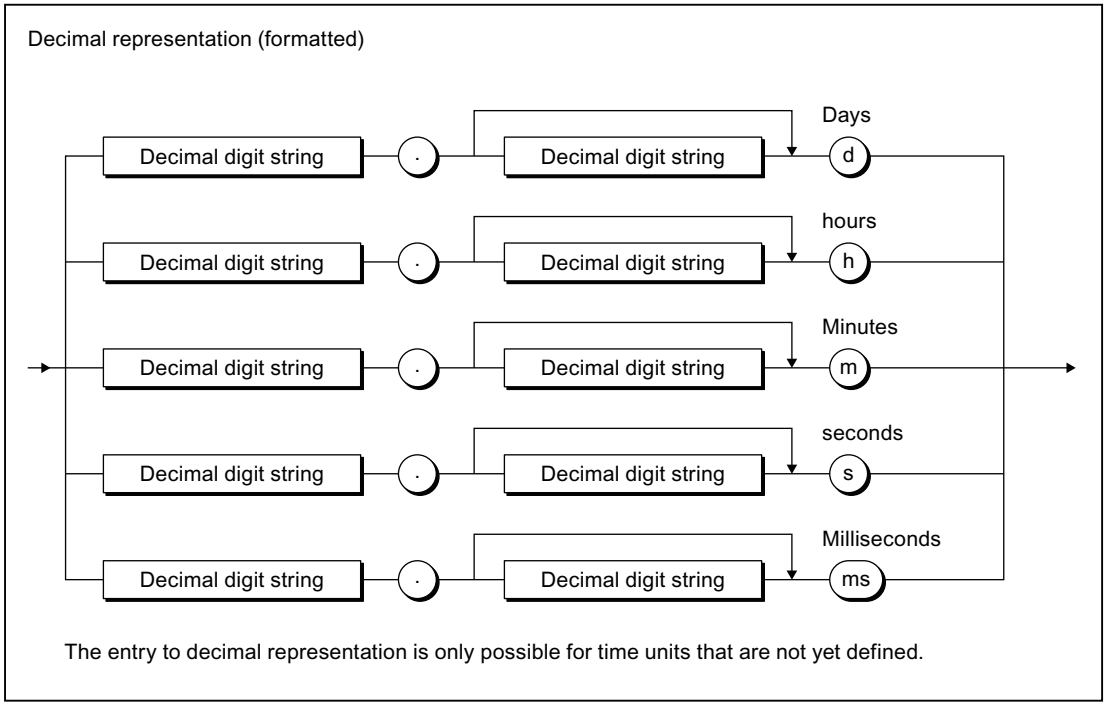


Figure A-23 Decimal representation

A.1.3.3 Comments

Note the following when inserting comments:

- The character pairs (* and *) are ignored within the line comment.
- Nesting of block comments is not allowed as standard. However, you can nest line comments in block comments.
  With activated compiler option "Permit language extensions IEC61131 3rd edition" only: It is possible to nest block comments.
- You can use the complete extended ASCII character set in comments.
- Comments are not allowed in formatted (lexical) rules.

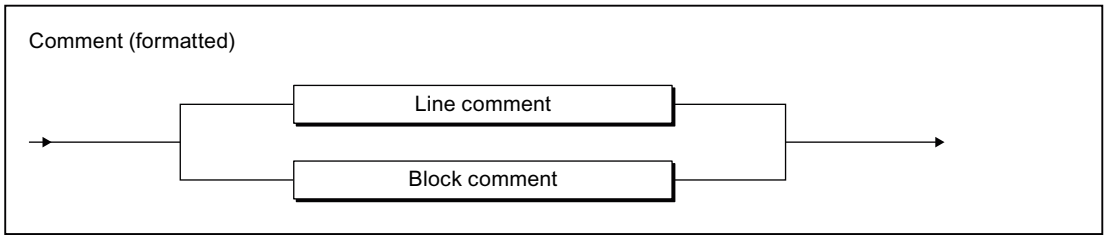


Figure A-24 Comment

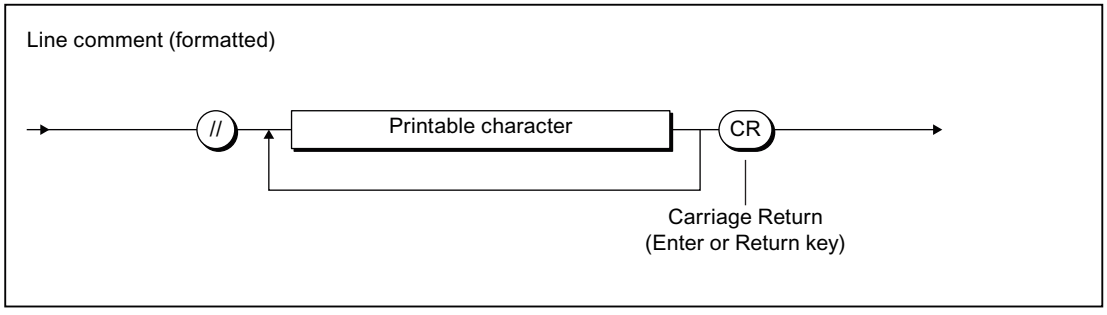


Figure A-25 Line comment

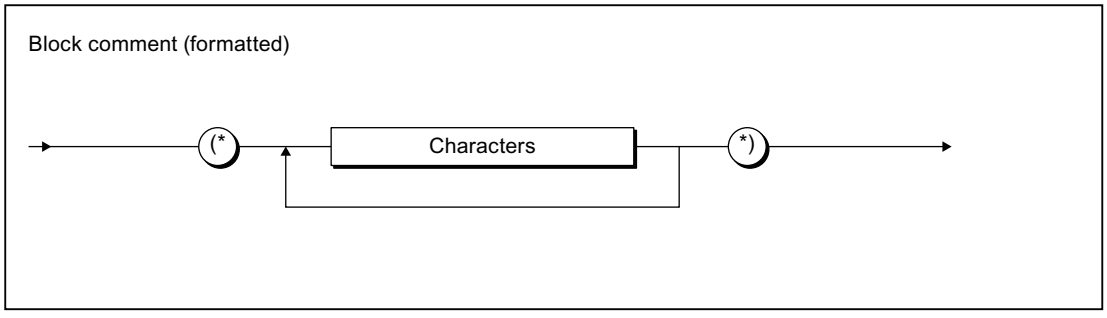
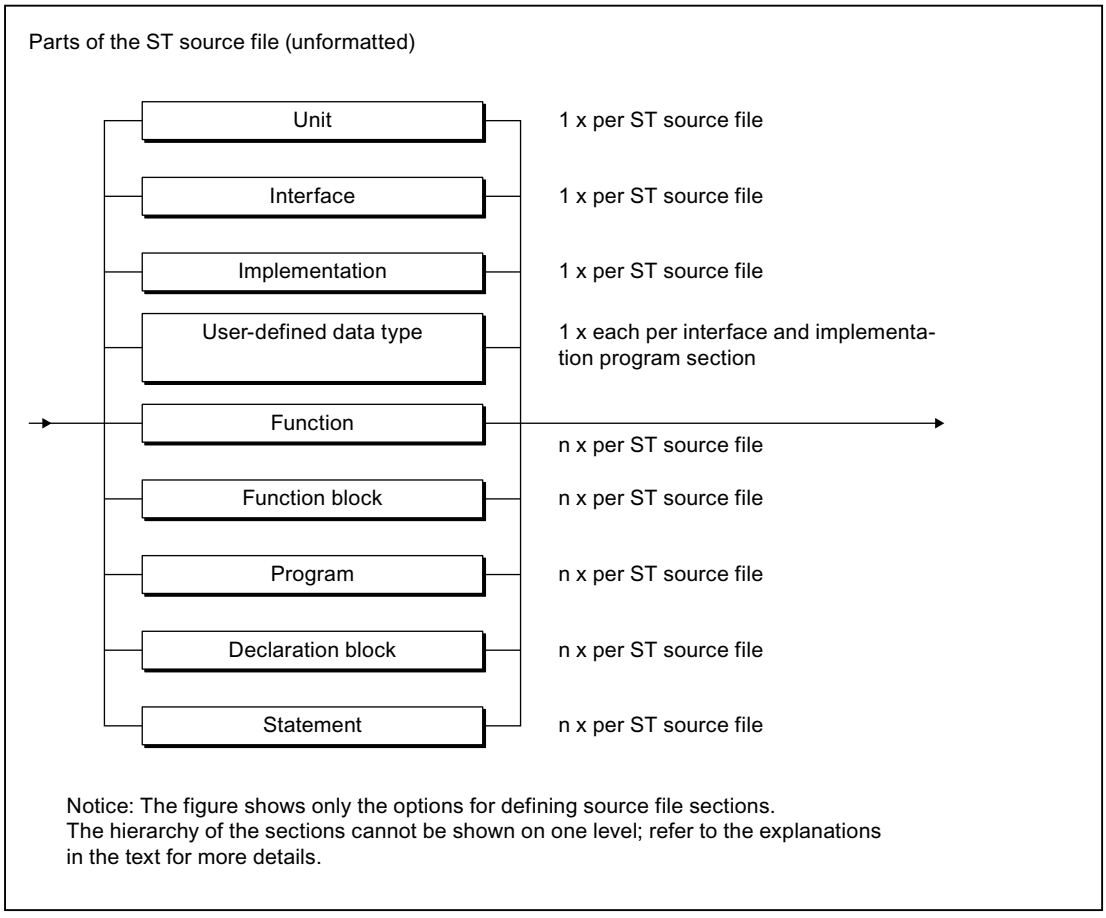


Figure A-26 Block comment

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

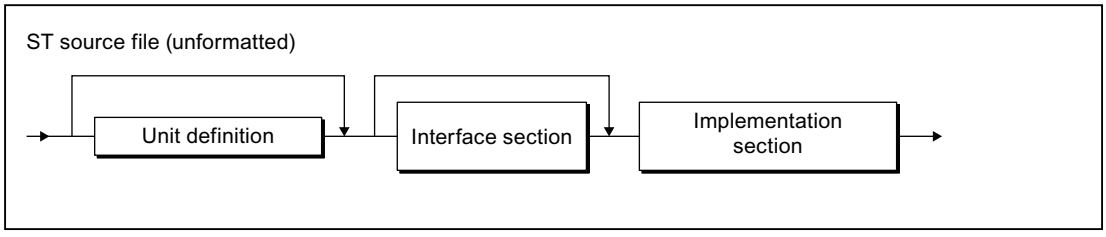


Figure A-27 ST source file

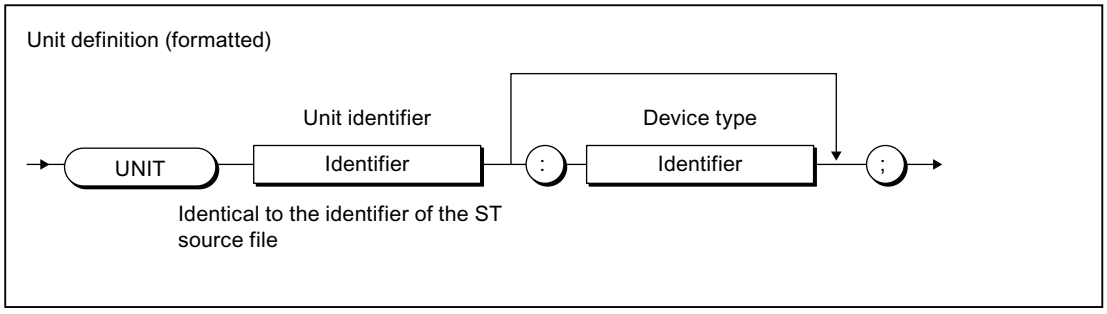


Figure A-28 Unit definition

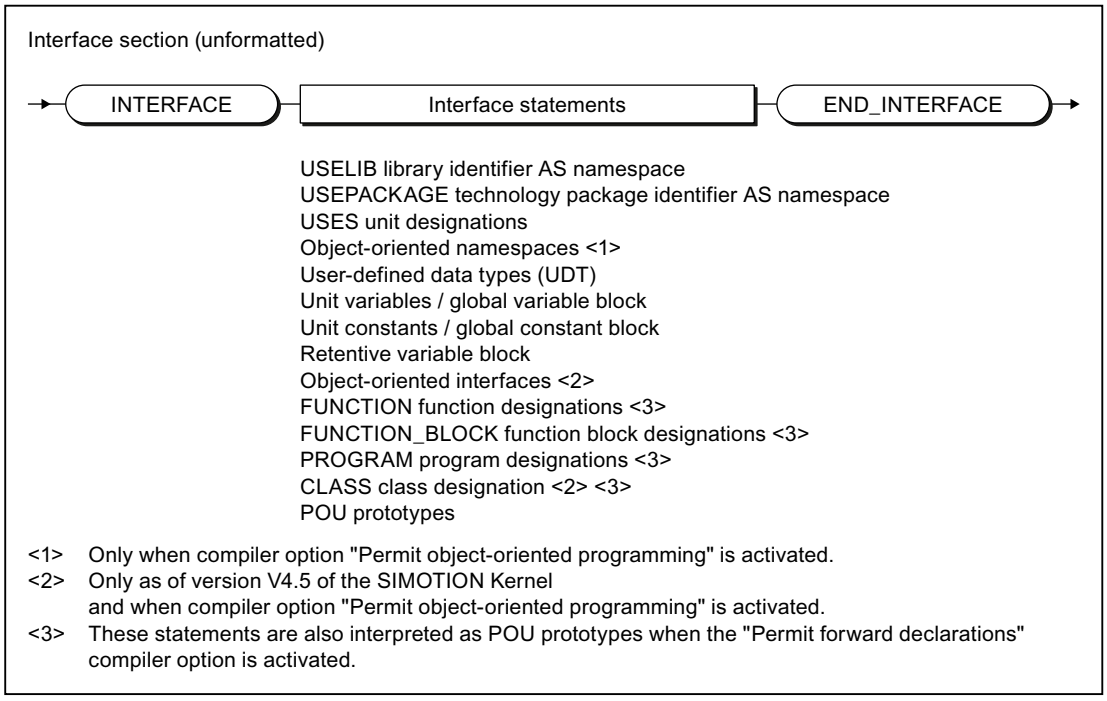


Figure A-29 Interface section

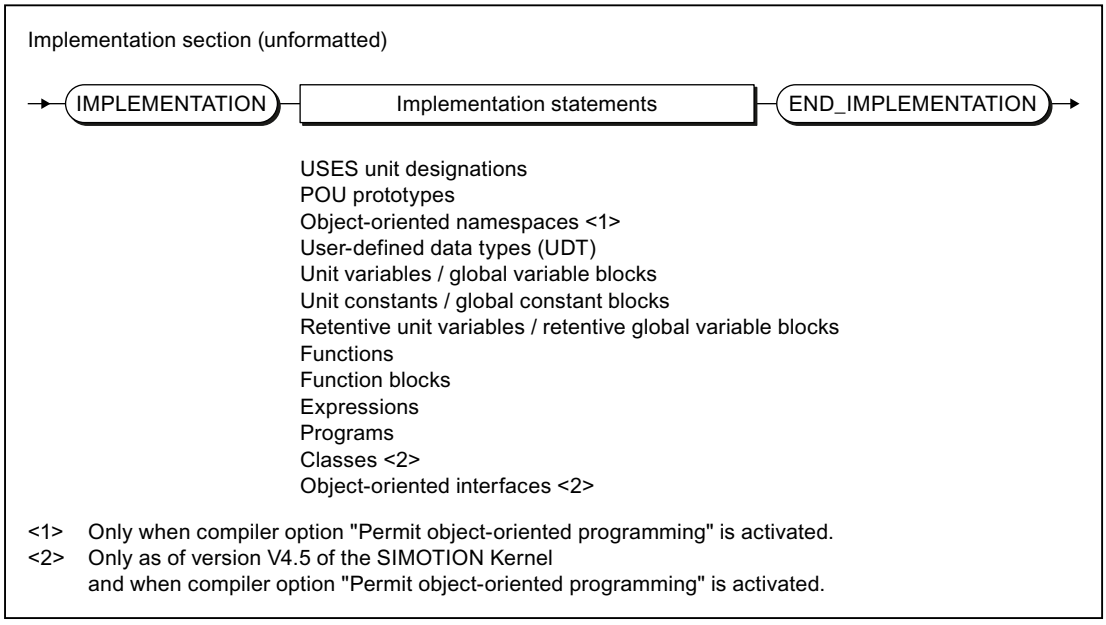


Figure A-30 Implementation section

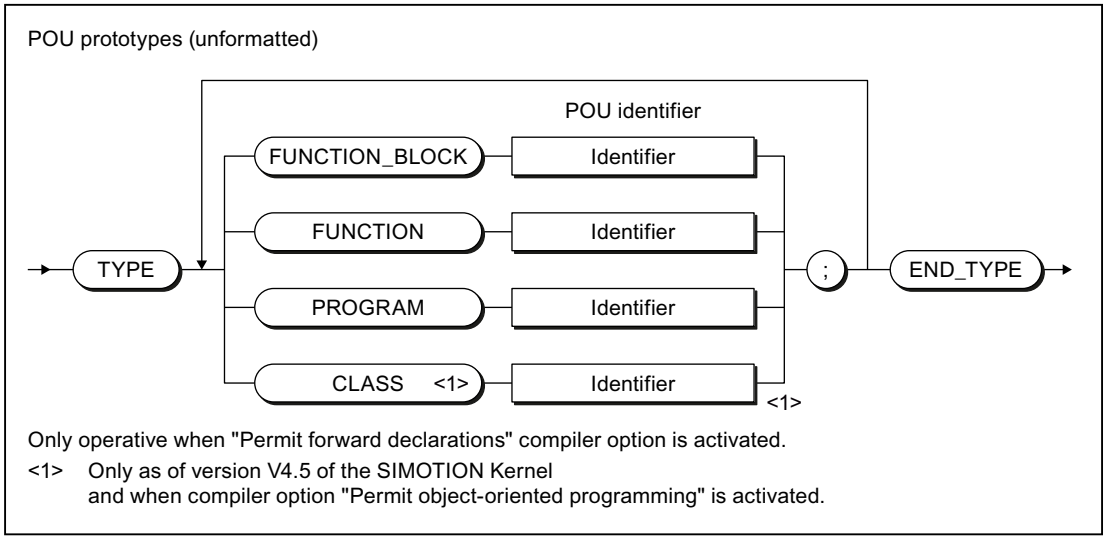


Figure A-31 POU prototypes

A.1.3.6 Program organization units (POU)

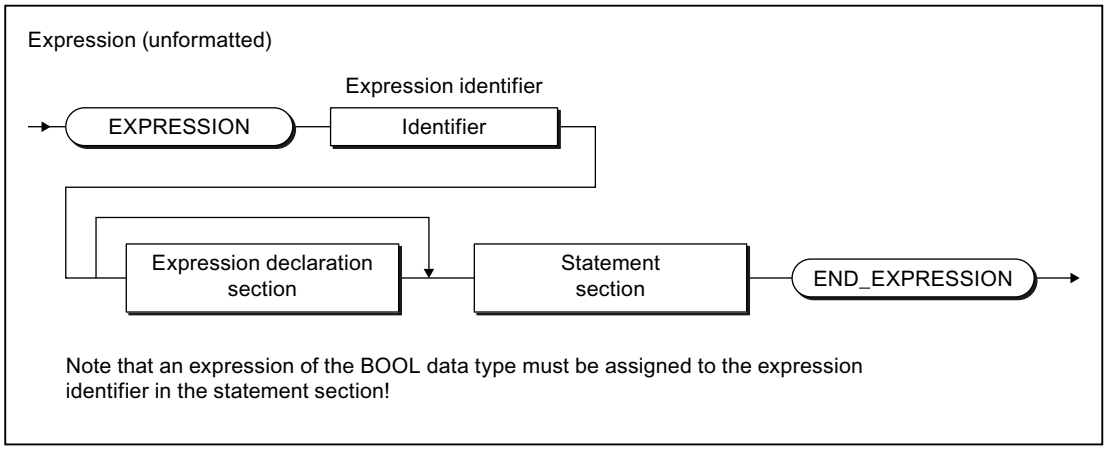


Figure A-32 Expression

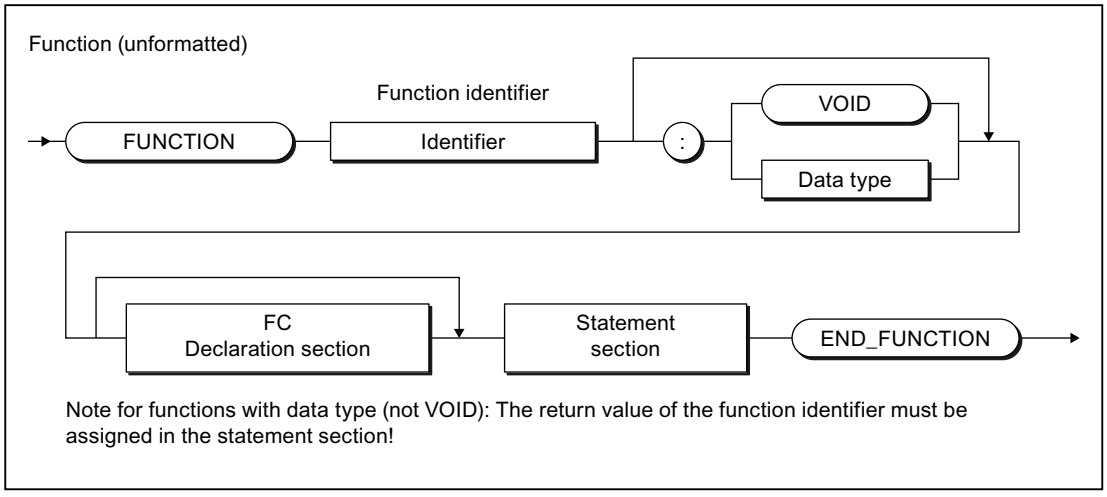


Figure A-33 Function (FC)

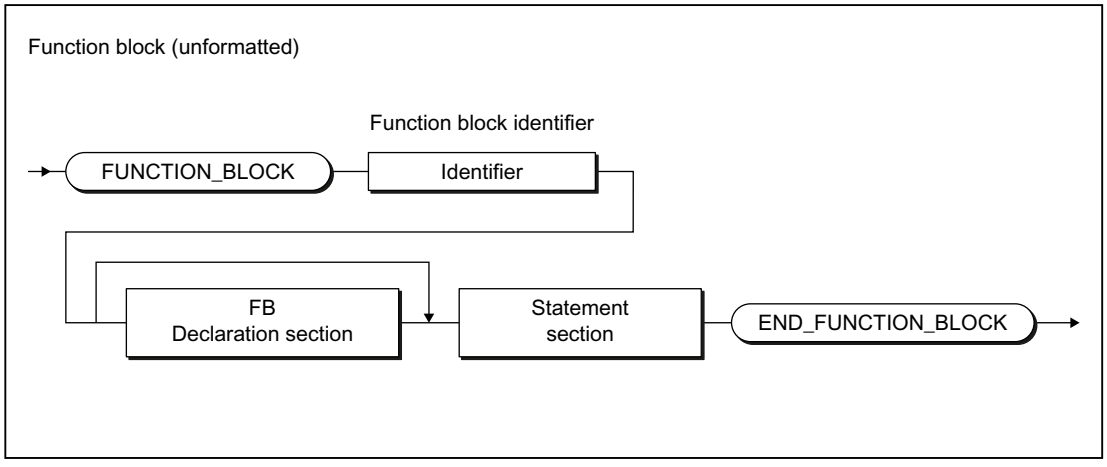


Figure A-34 Function block (FB)

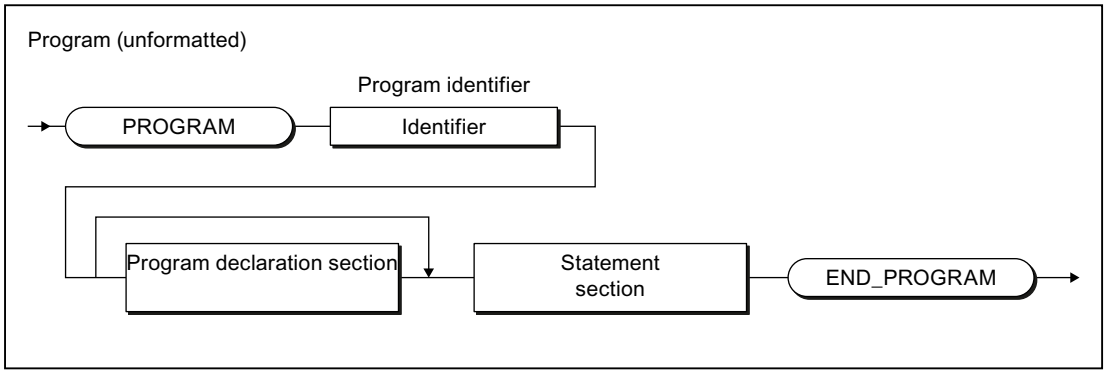


Figure A-35 Program

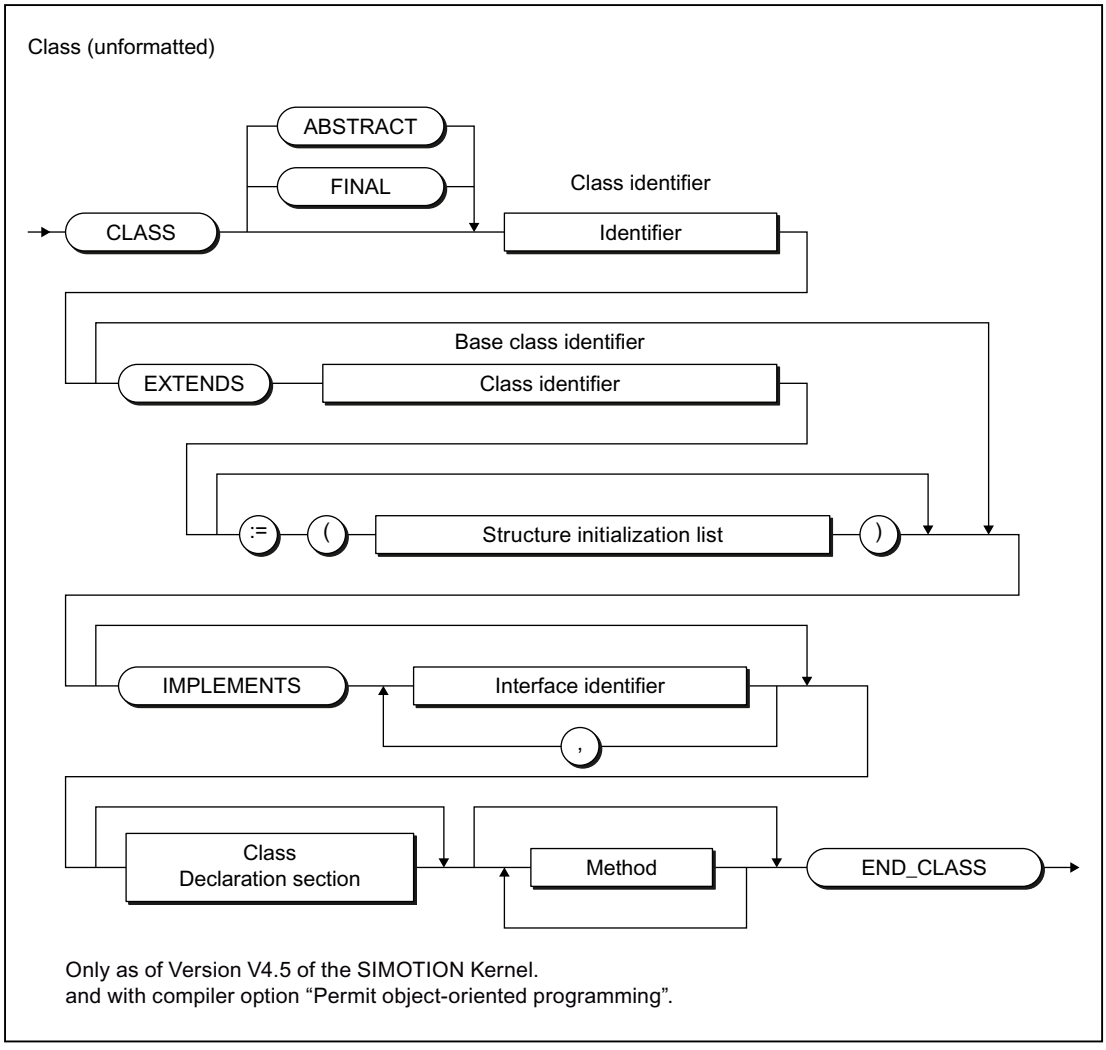


Figure A-36 Syntax: Class

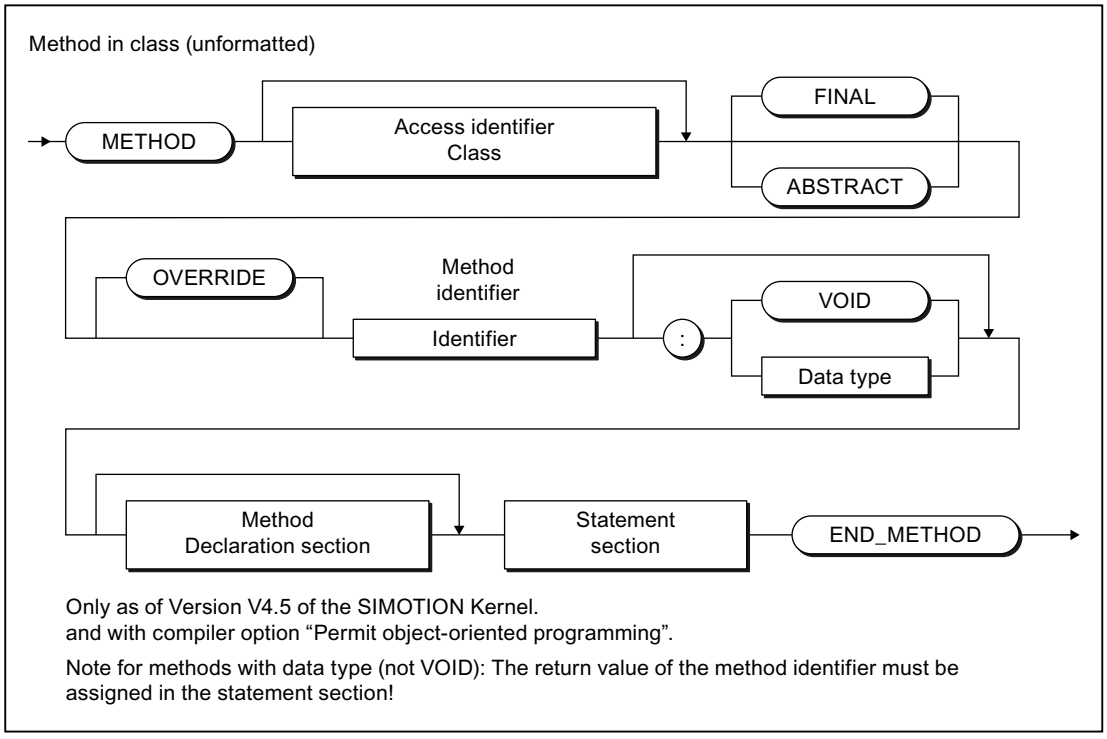


Figure A-37 Syntax: Method

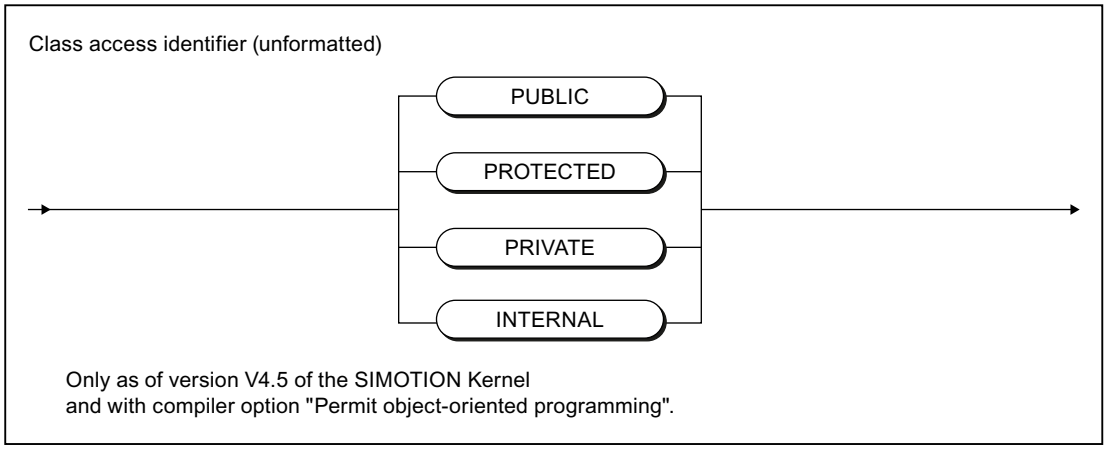


Figure A-38 Syntax: Class access identifier

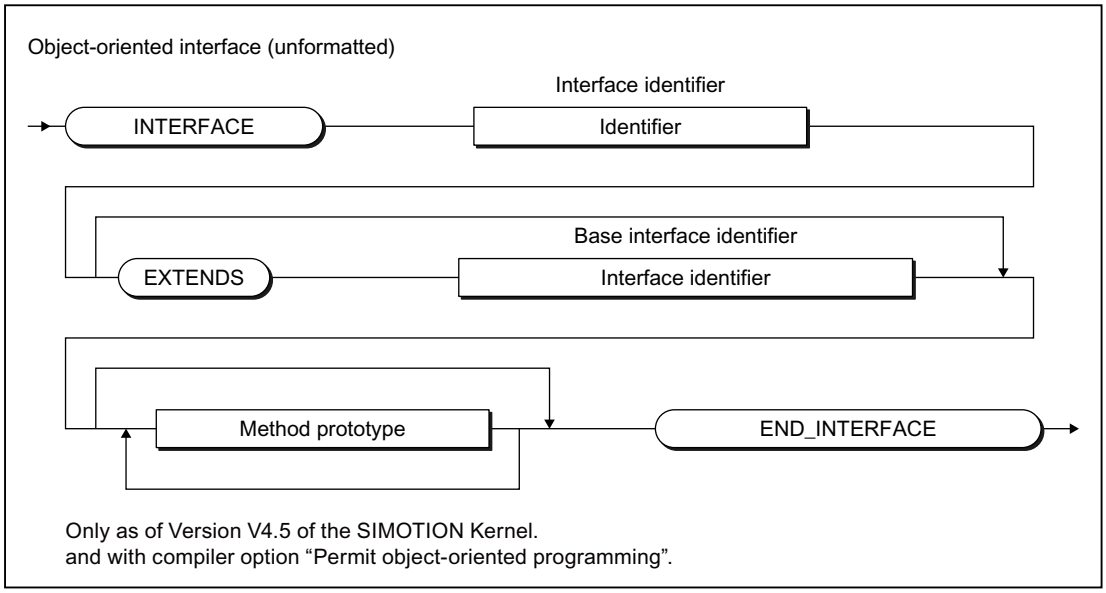


Figure A-39 Syntax: Object-oriented interface

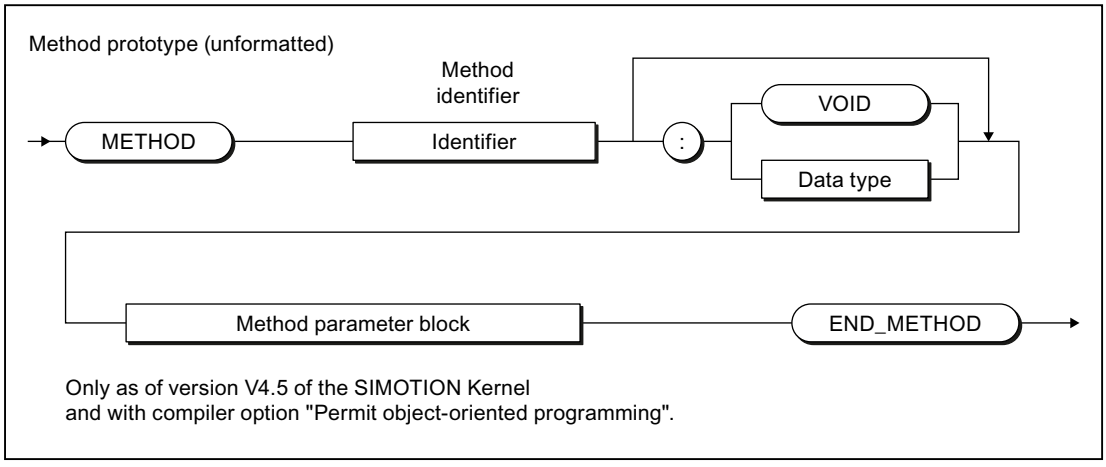


Figure A-40 Syntax: Method prototype

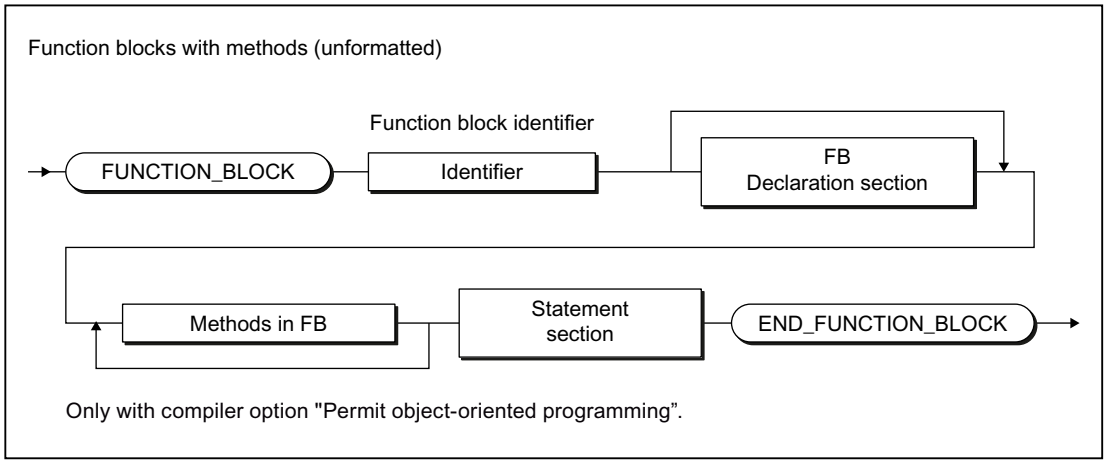


Figure A-41 Syntax: Function block (FB) with methods

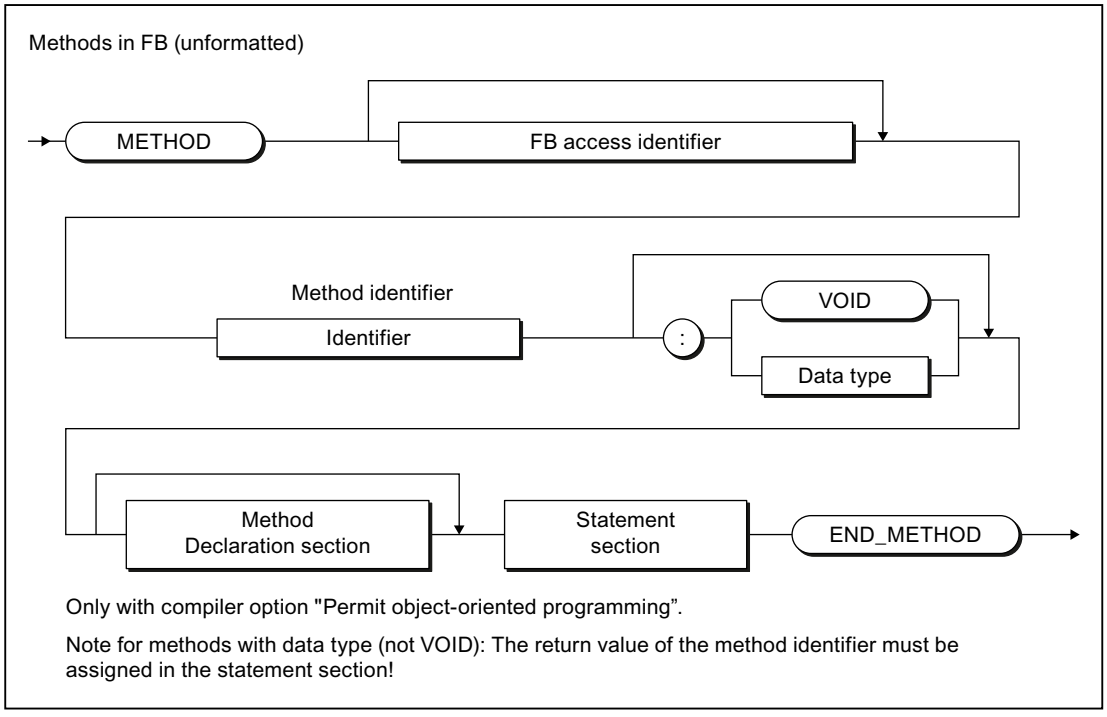


Figure A-42 Syntax: Methods in the function block

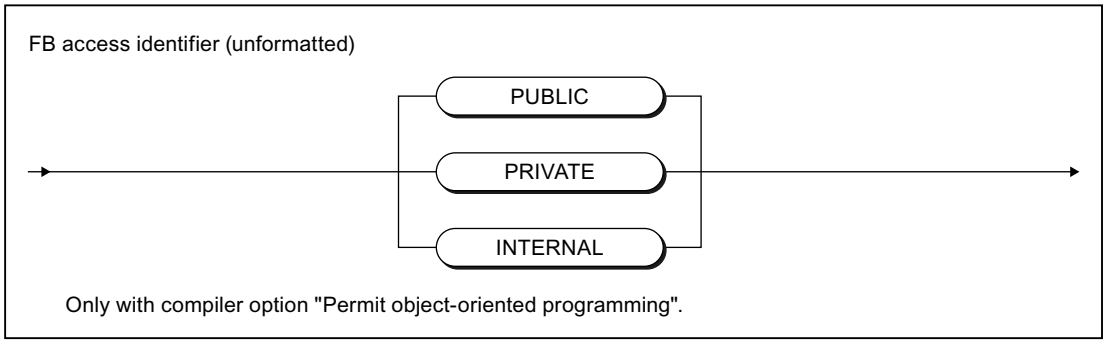


Figure A-43 Syntax: FB access identifier

A.1.3.7 Declaration sections

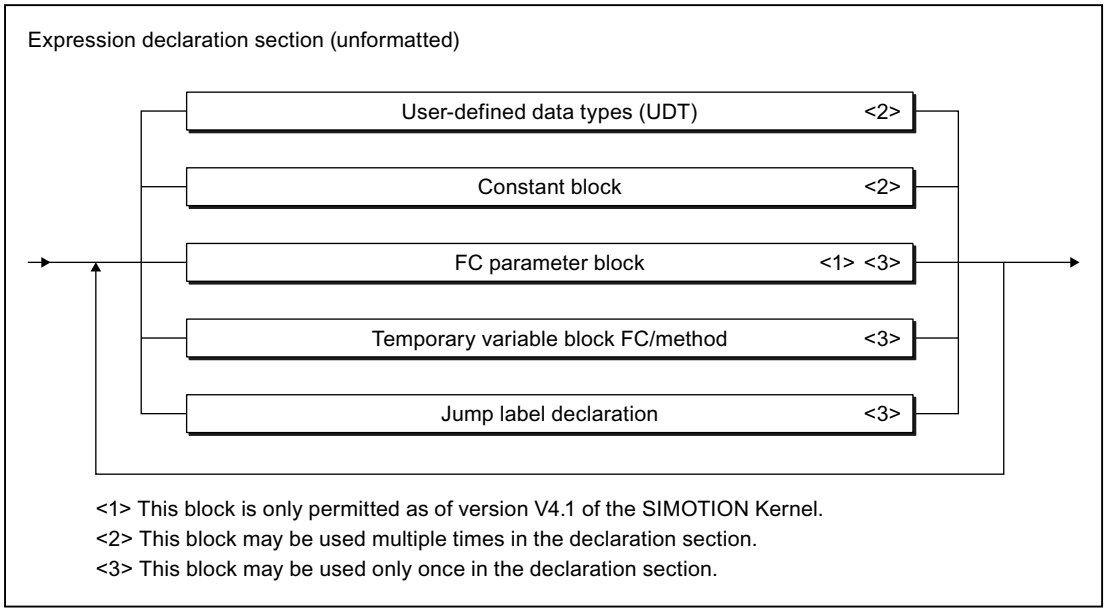


Figure A-44 Expression declaration section

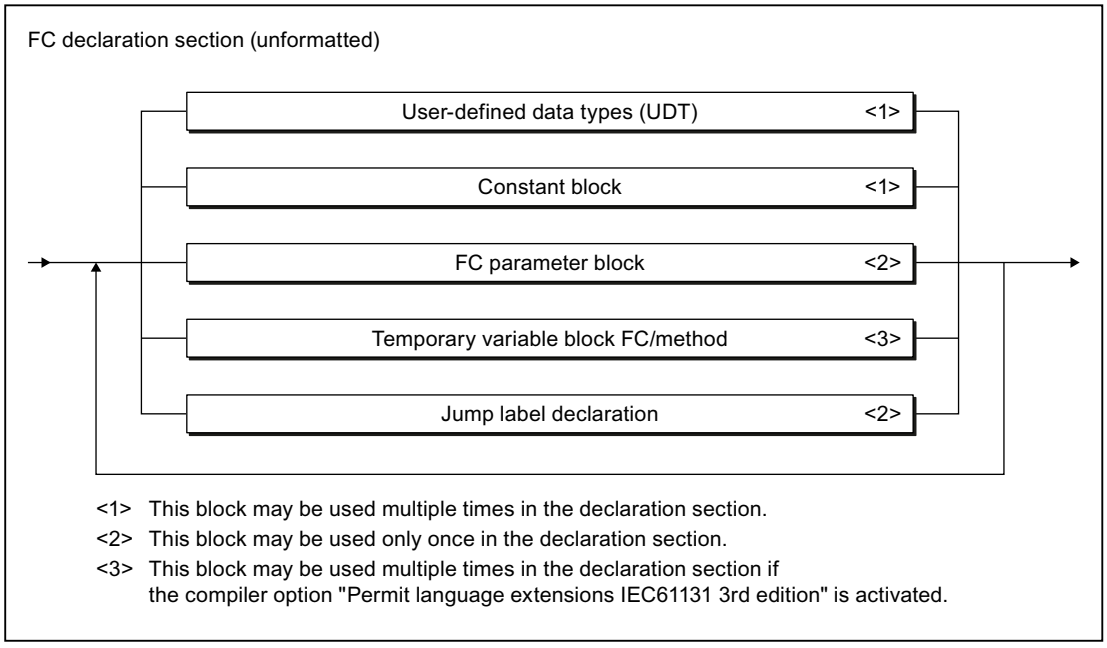


Figure A-45 FC declaration section

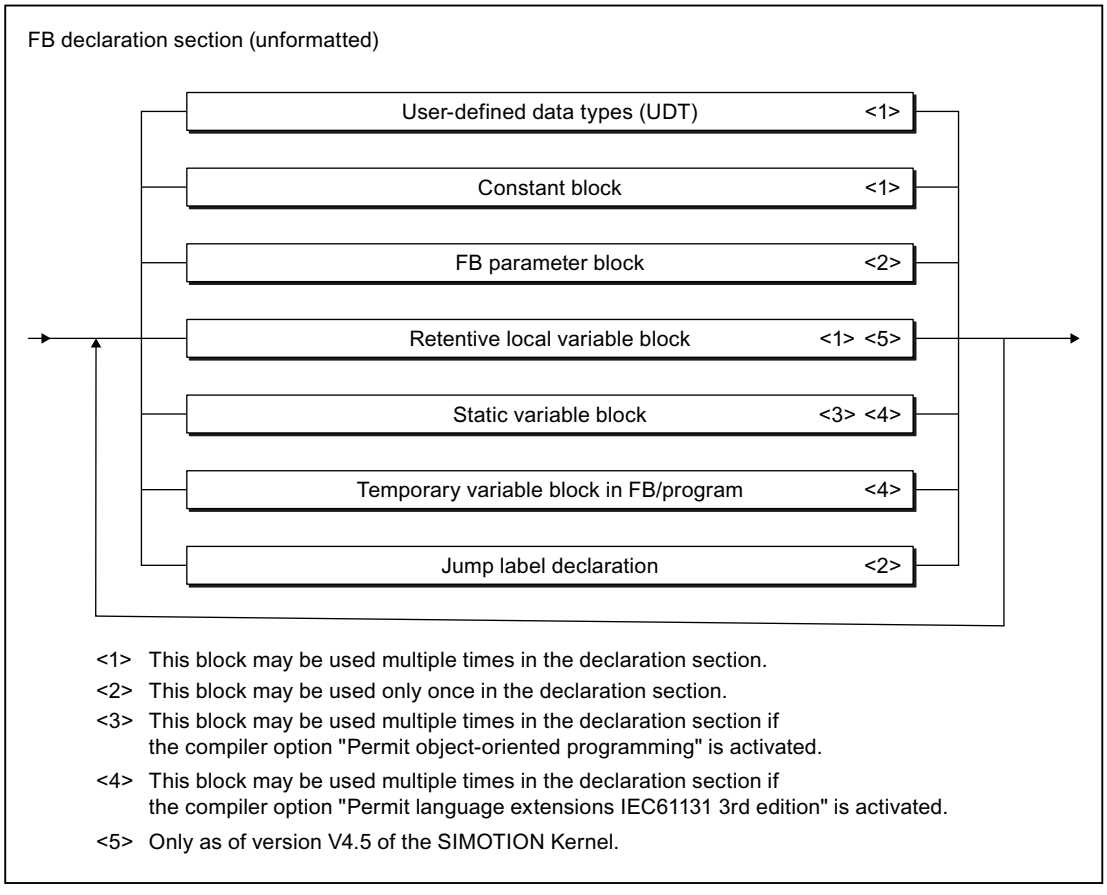


Figure A-46 FB declaration section

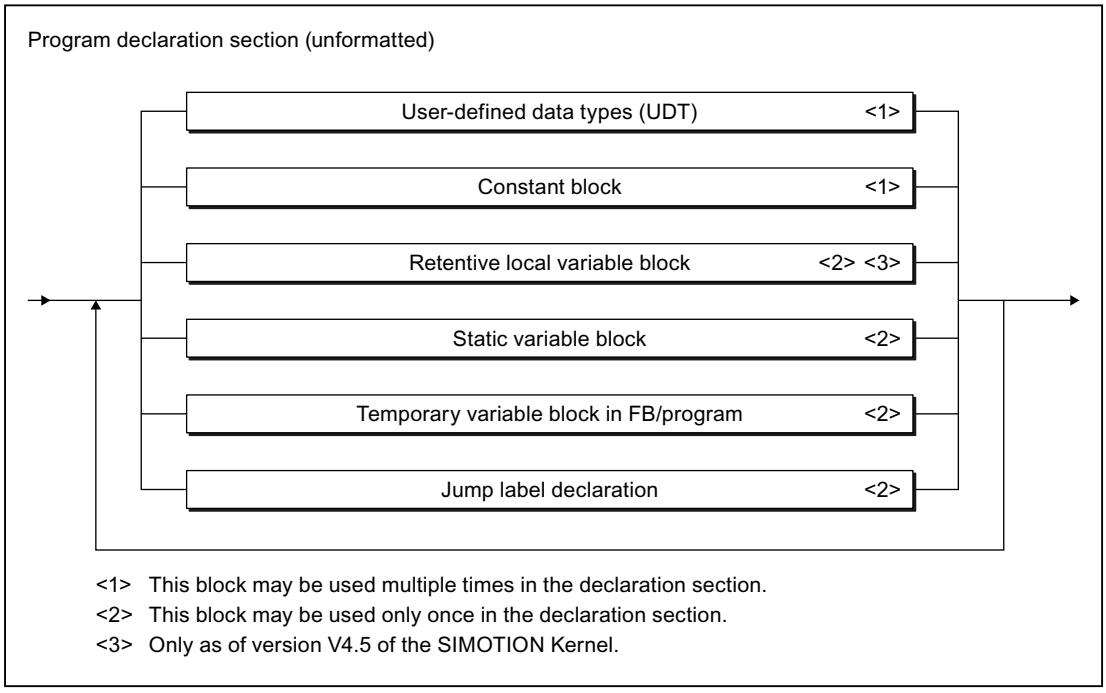


Figure A-47 Program declaration section

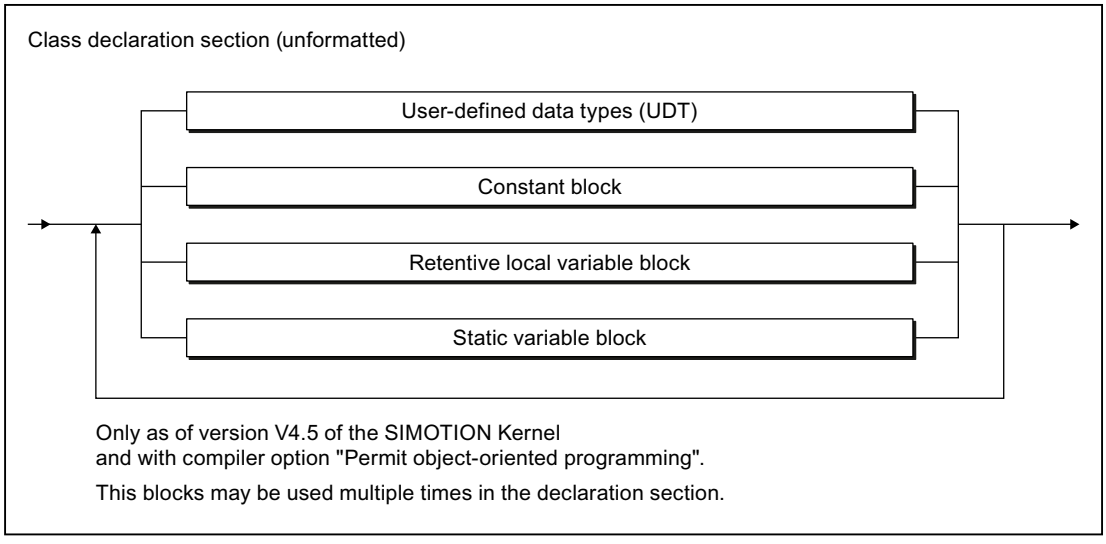


Figure A-48 Class declaration section

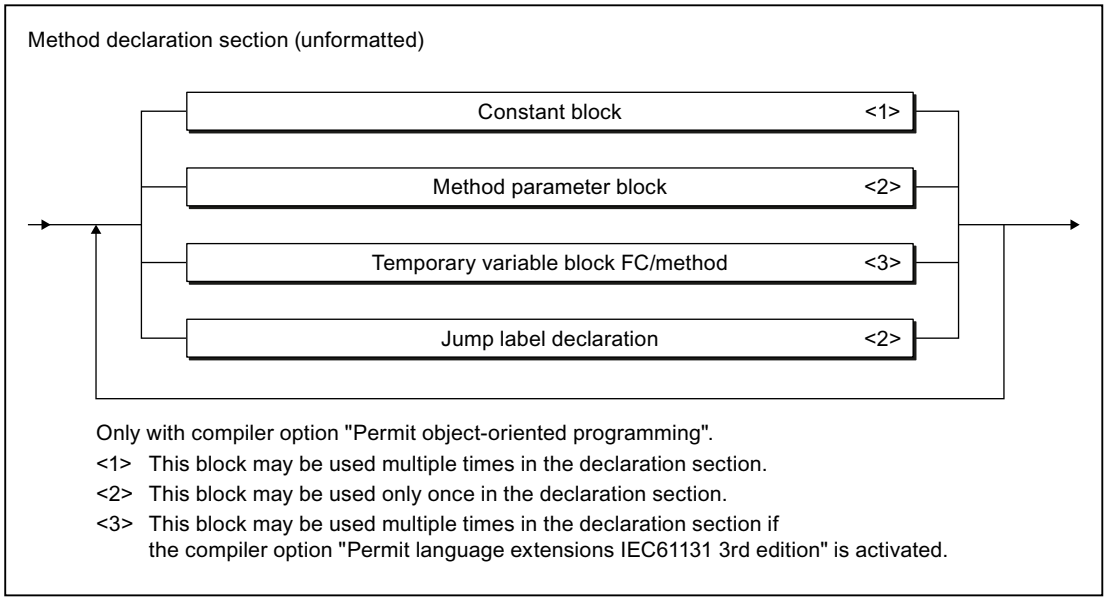


Figure A-49 Method declaration section

A.1.3.8 Structure of the declaration blocks

Constant blocks

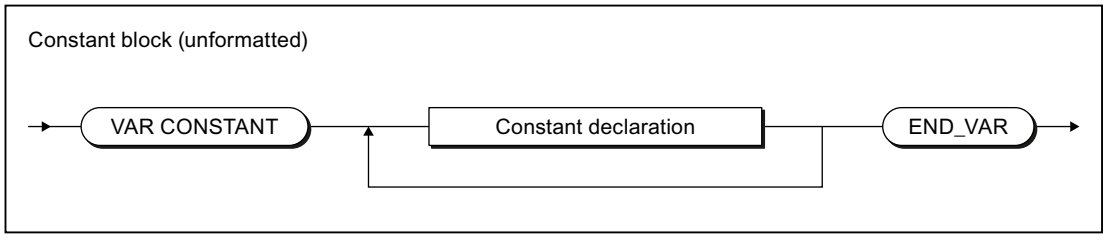


Figure A-50 Constant block

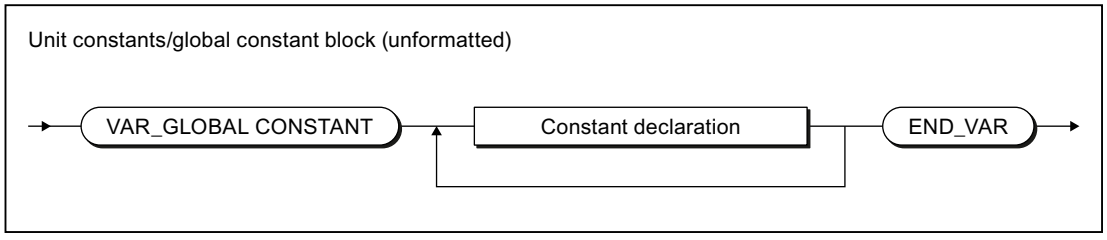


Figure A-51 Unit constants / global constant block

Variable blocks

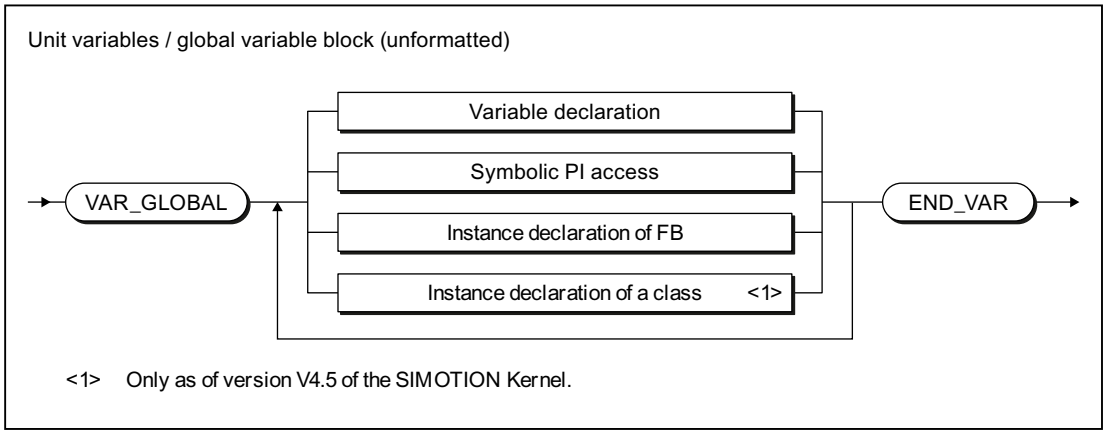


Figure A-52 Unit variables / global variable block

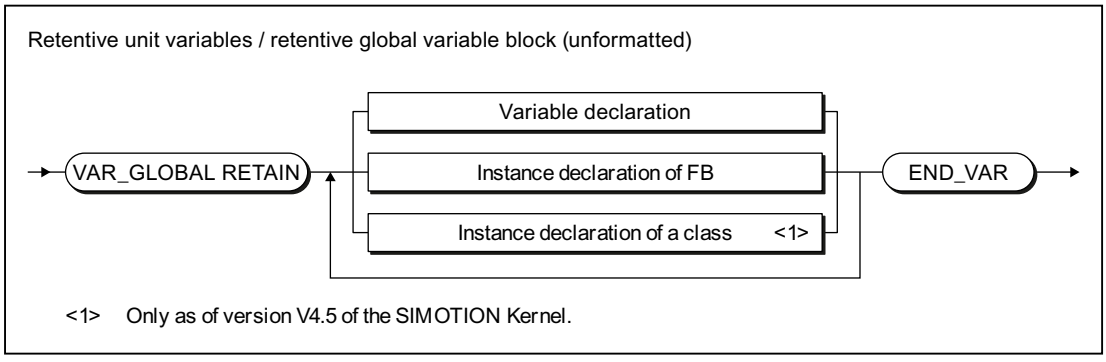


Figure A-53 Retentive global variable block

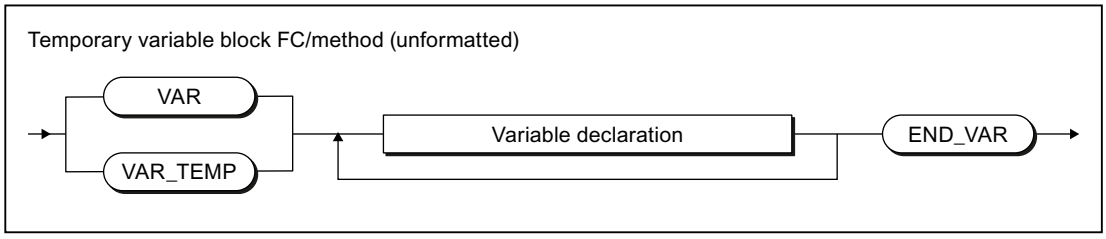


Figure A-54 Temporary variable block FC/method

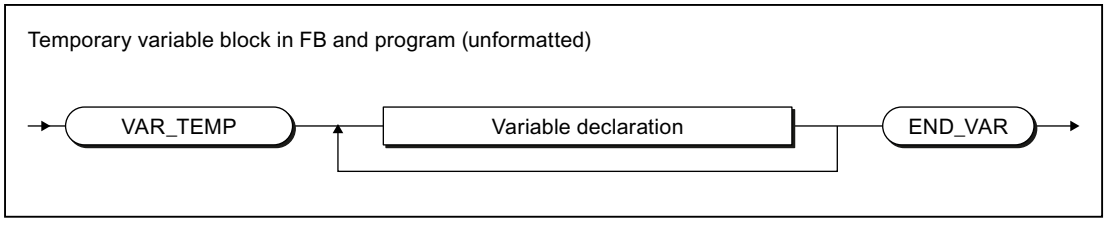


Figure A-55 Temporary variable block in the FB/program

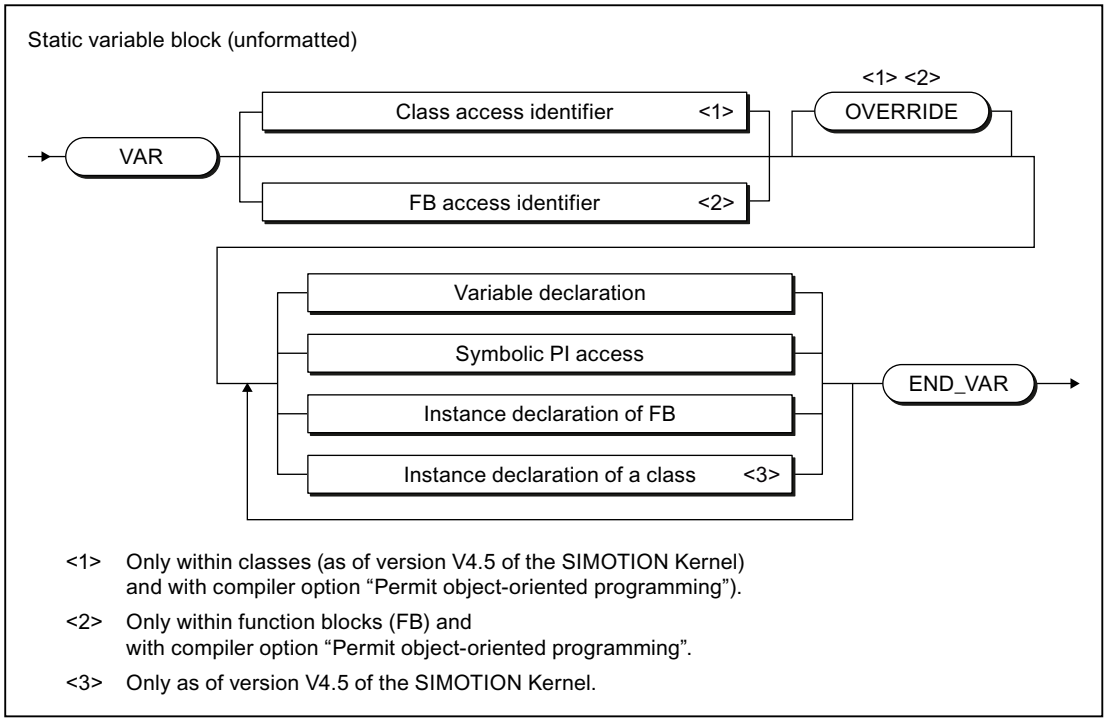


Figure A-56 Static variable block

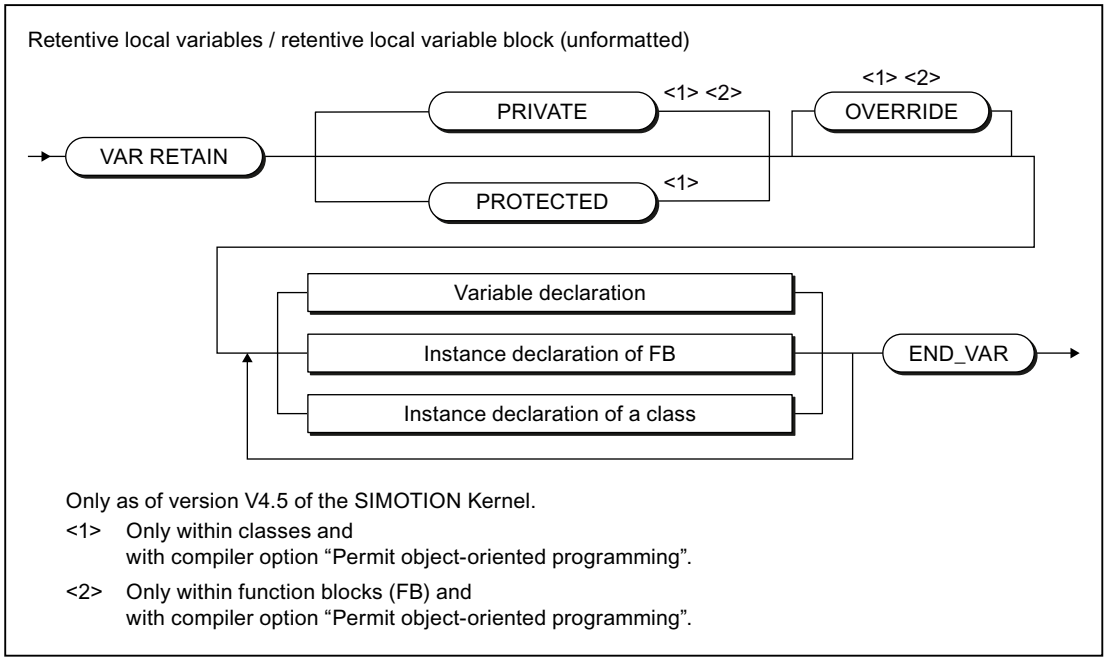


Figure A-57 Syntax: Retentive local variable block

Parameter fields

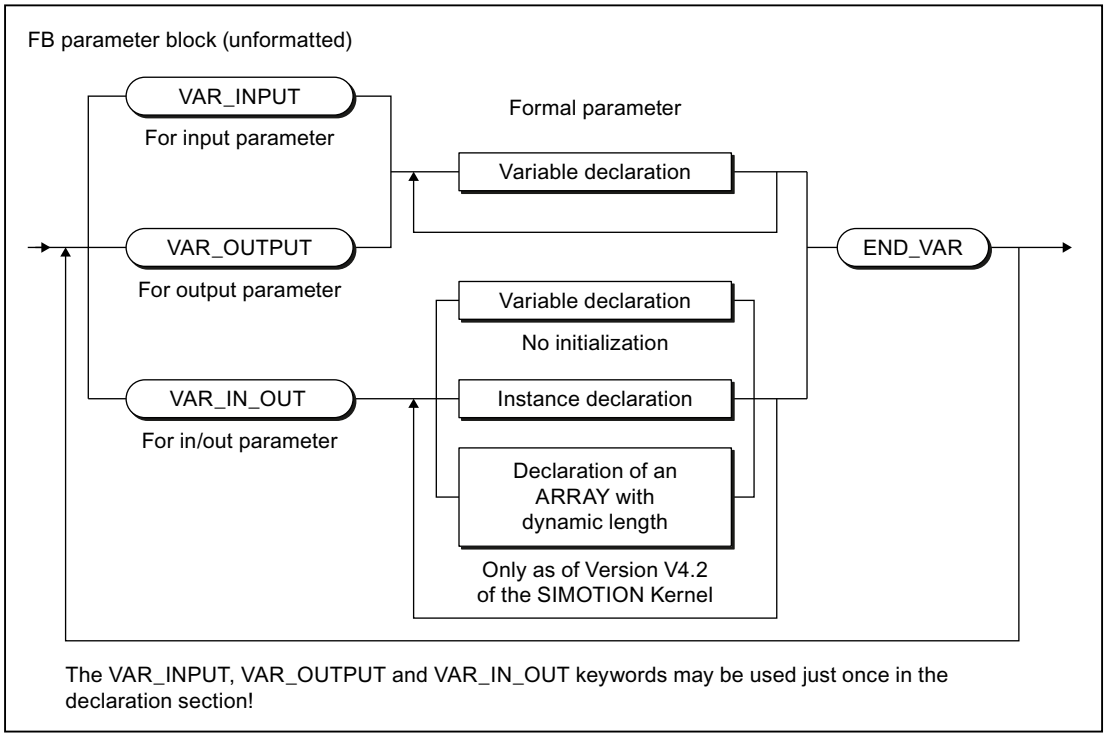


Figure A-58 FB parameter block

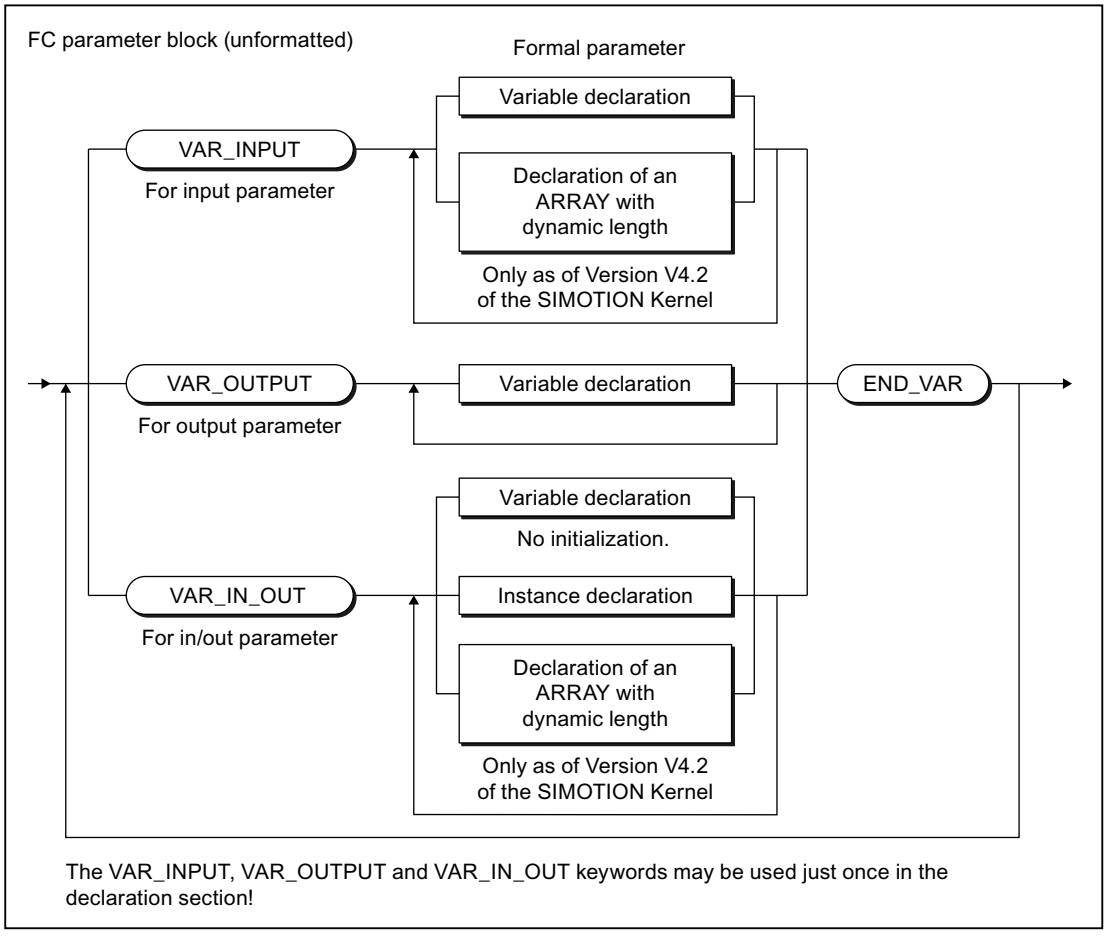


Figure A-59 FC parameter block

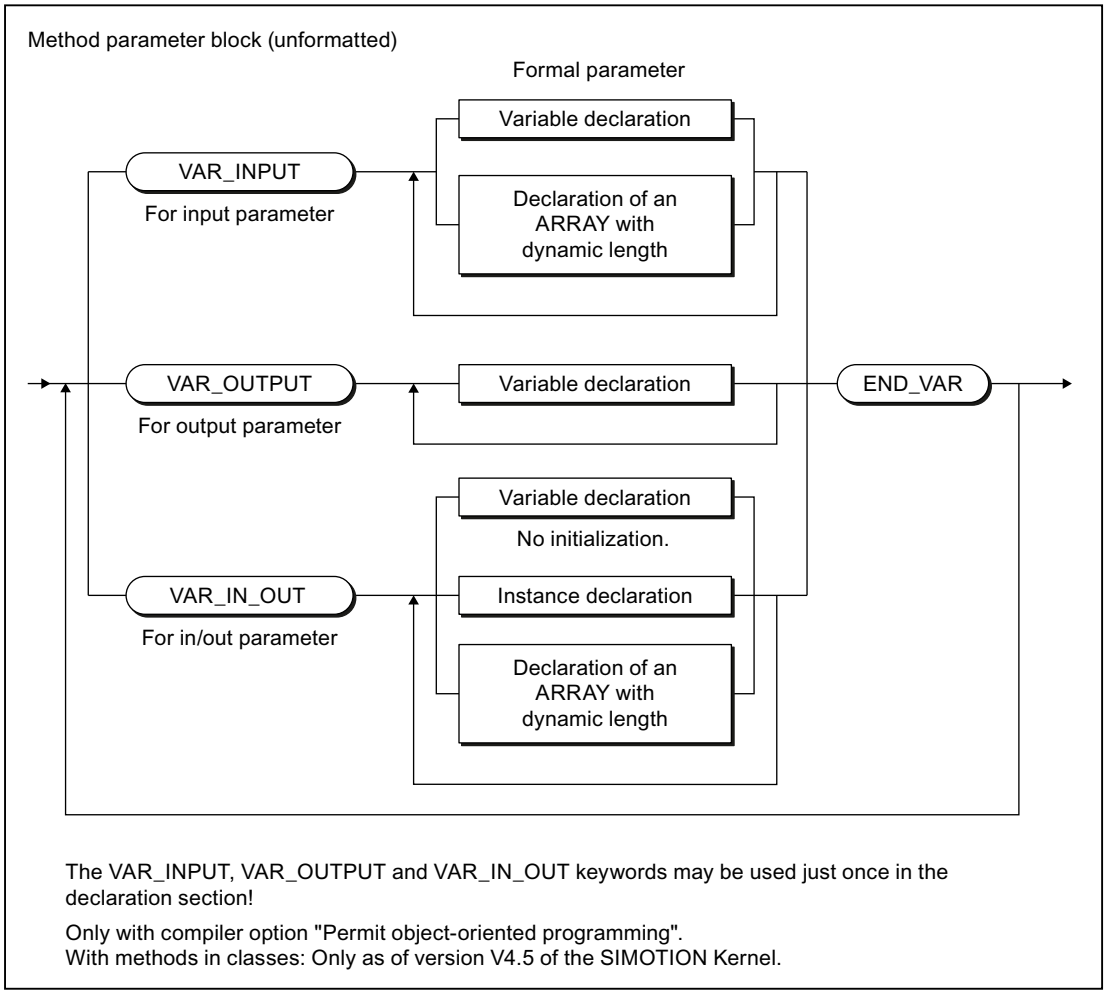


Figure A-60 Method parameter block

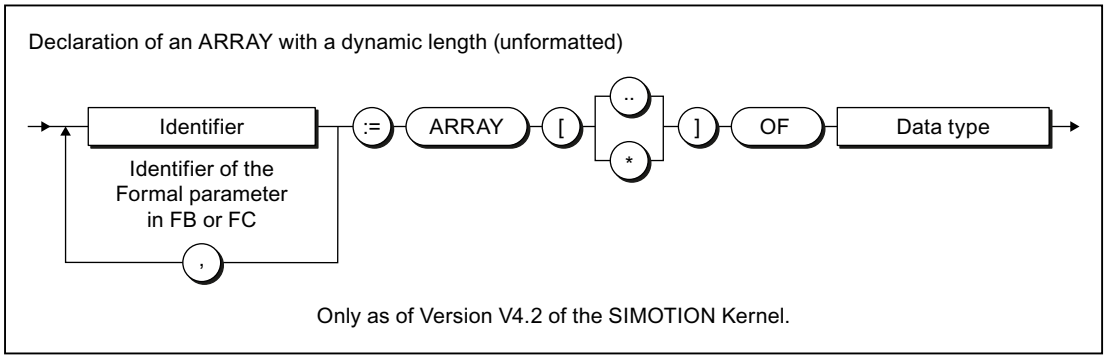


Figure A-61 Declaration of an ARRAY with a dynamic length

Jump labels

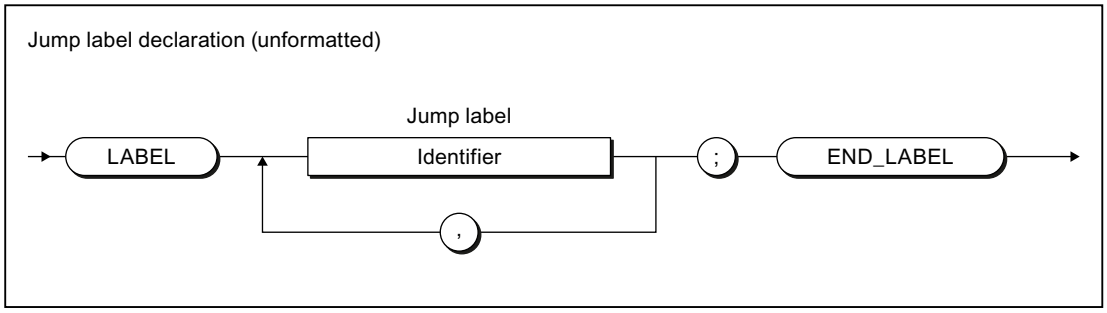


Figure A-62 Jump label declaration

Declarations

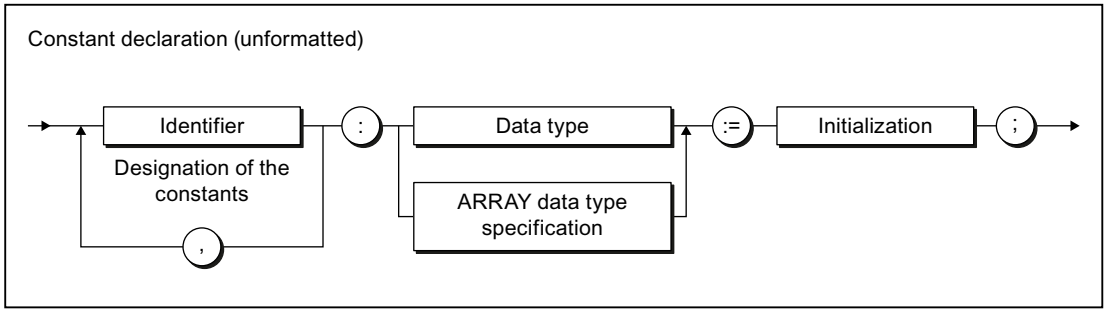


Figure A-63 Constant declaration

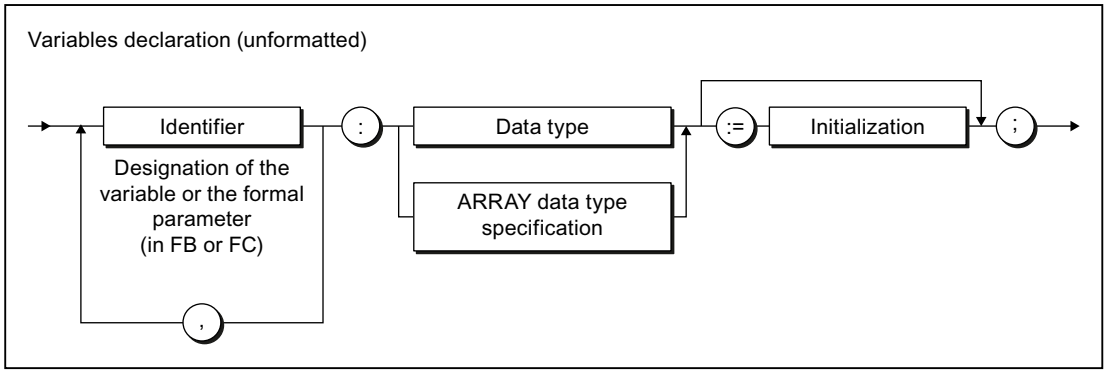


Figure A-64 Variable declaration

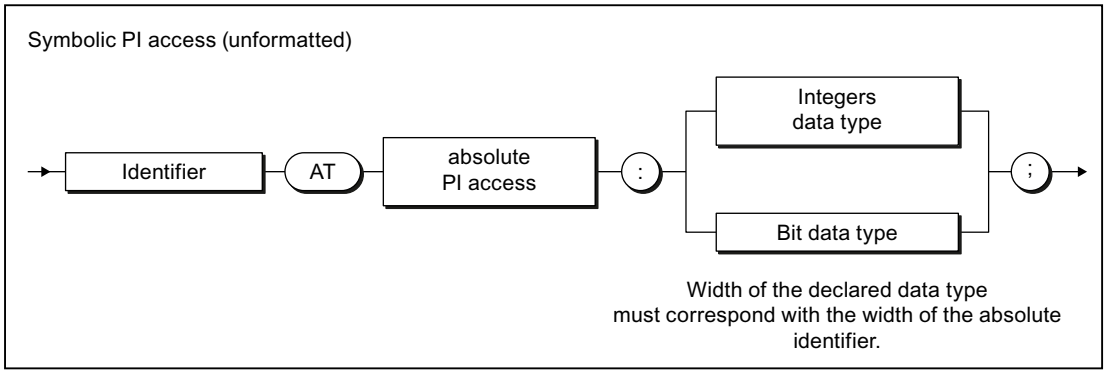


Figure A-65 Symbolic PI access

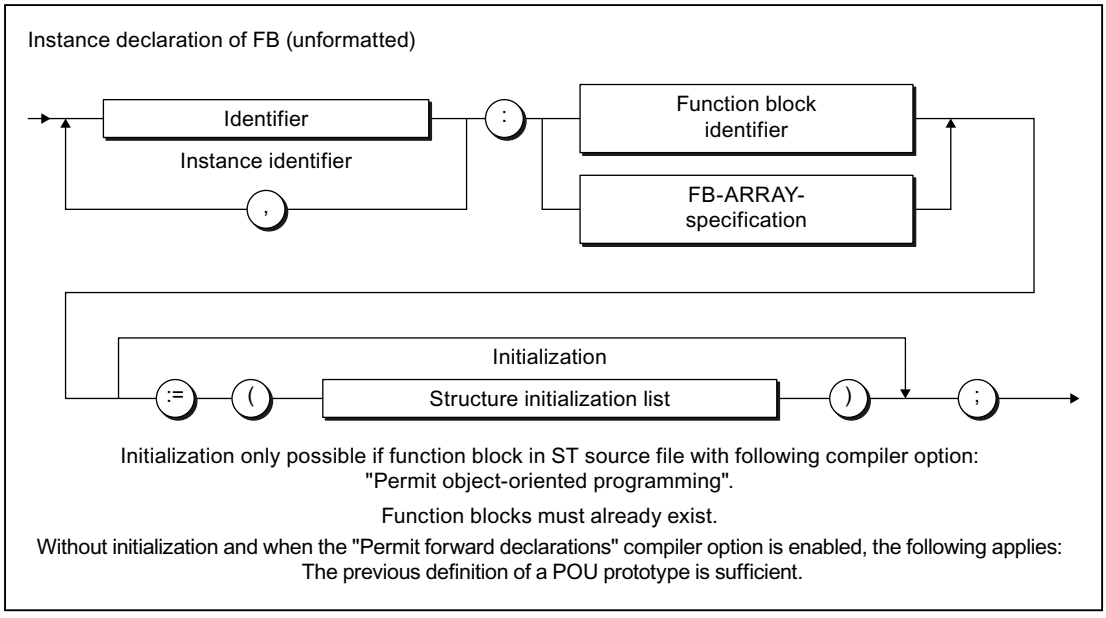


Figure A-66 Instance declaration of FB

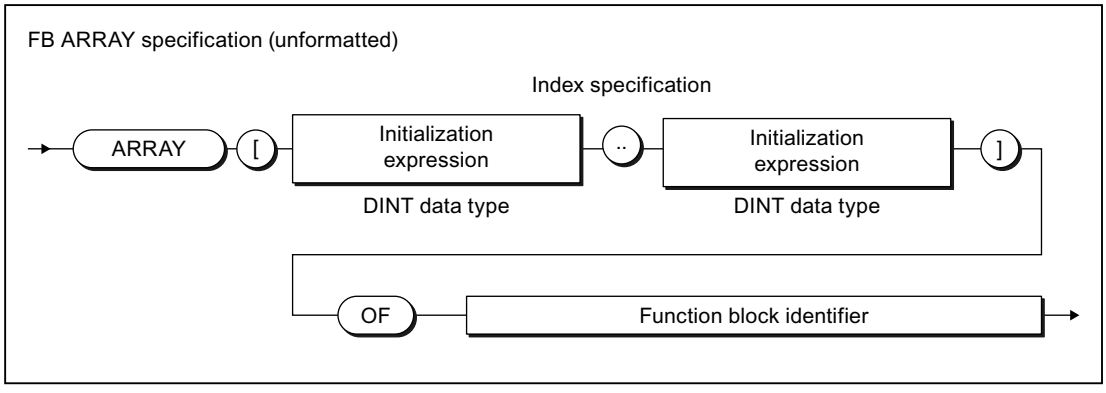


Figure A-67 FB ARRAY specification

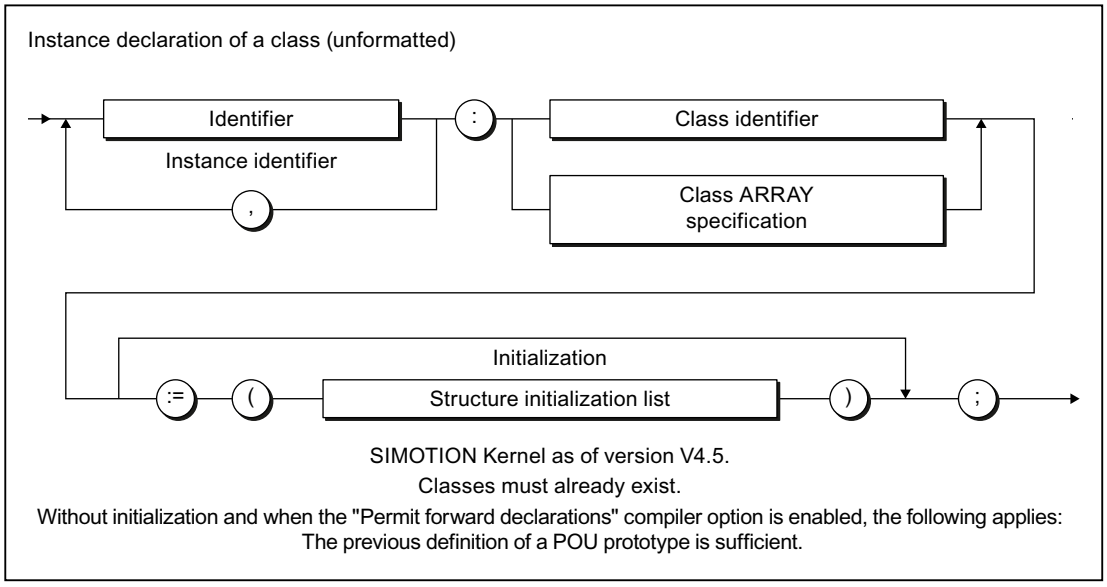


Figure A-68 Syntax: Instance declaration of a class

Initialization

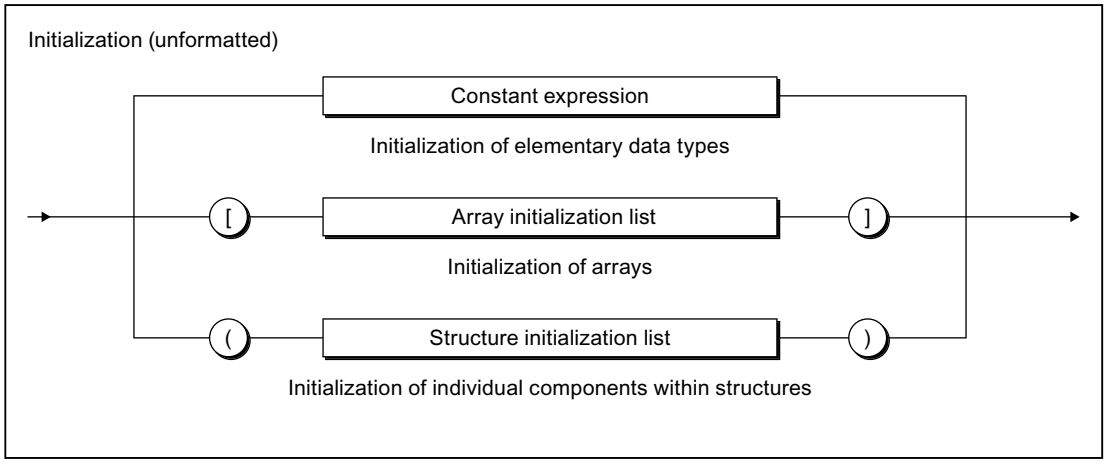


Figure A-69 Initialization

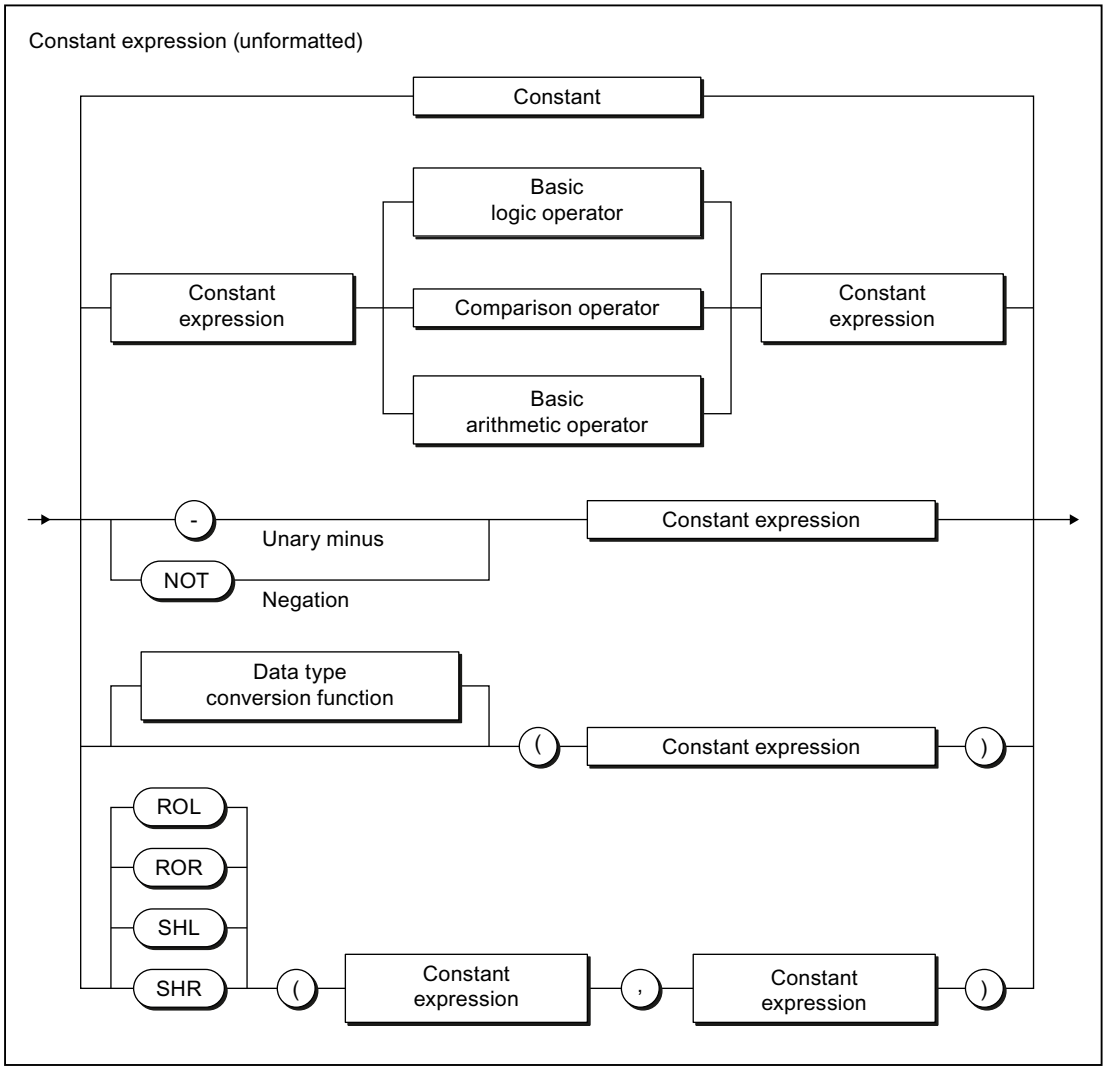


Figure A-70 Constant expression

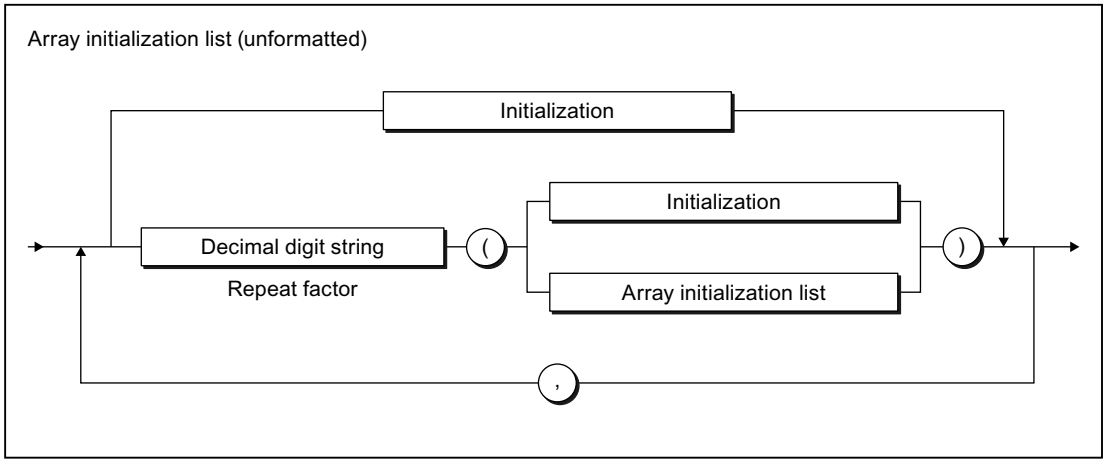


Figure A-71 Array initialization list

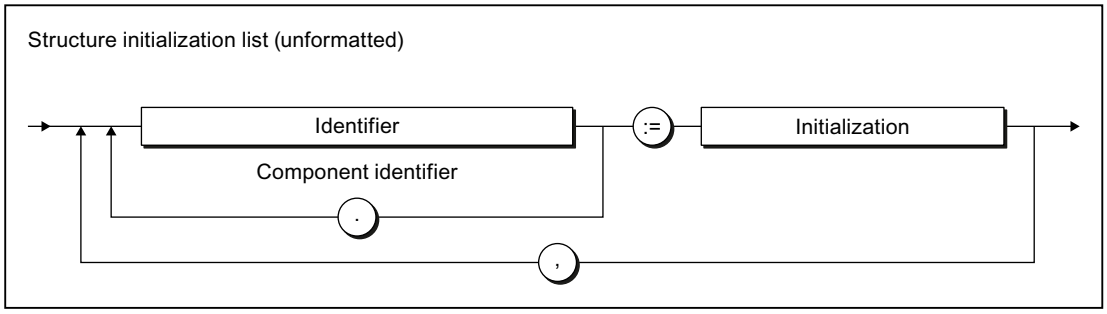


Figure A-72 Structure initialization list

A.1.3.9 Data types

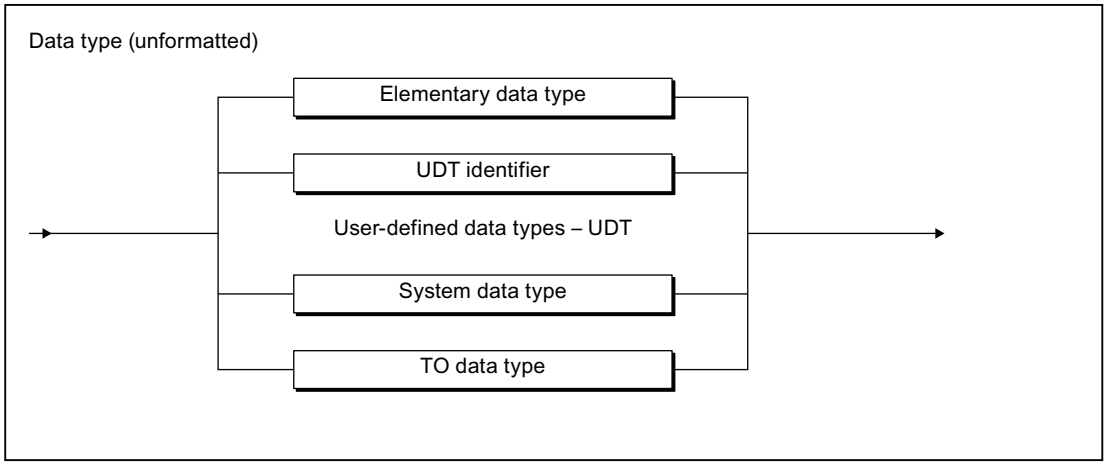


Figure A-73 Data type

Elementary data types

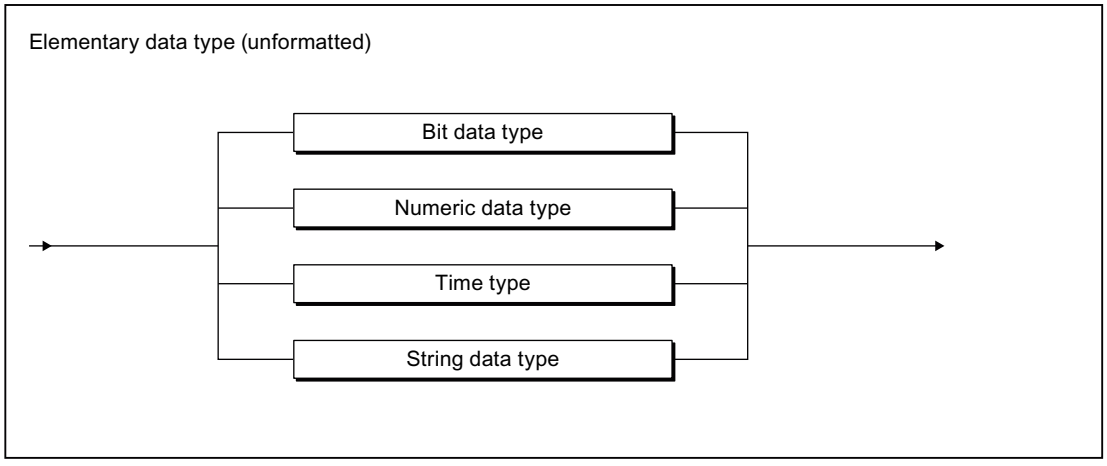


Figure A-74 Elementary data type

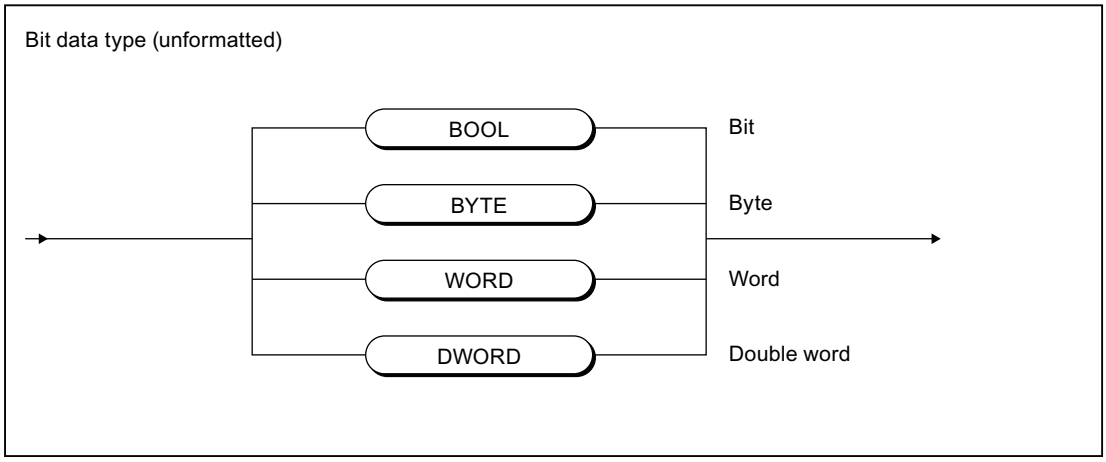


Figure A-75 Bit data type

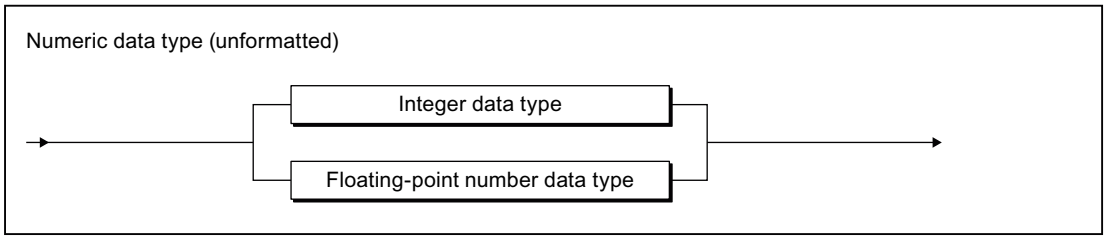


Figure A-76 Numeric data type

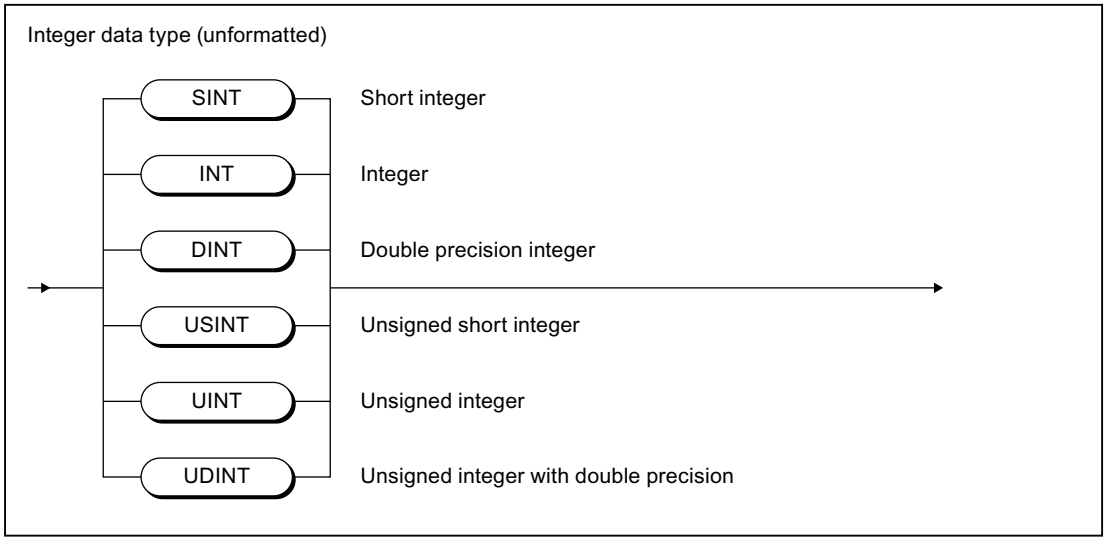


Figure A-77 Integer data type

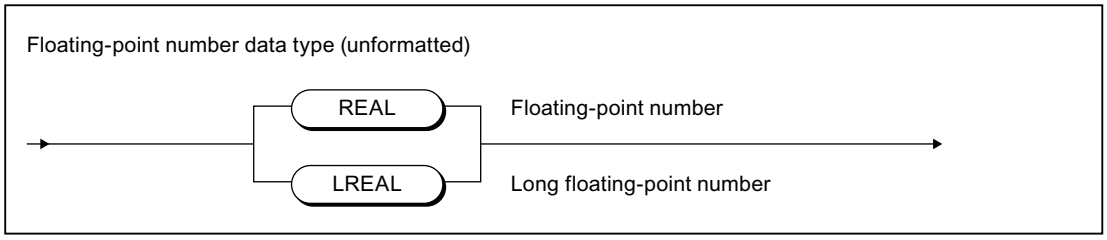


Figure A-78 Floating-point number data type

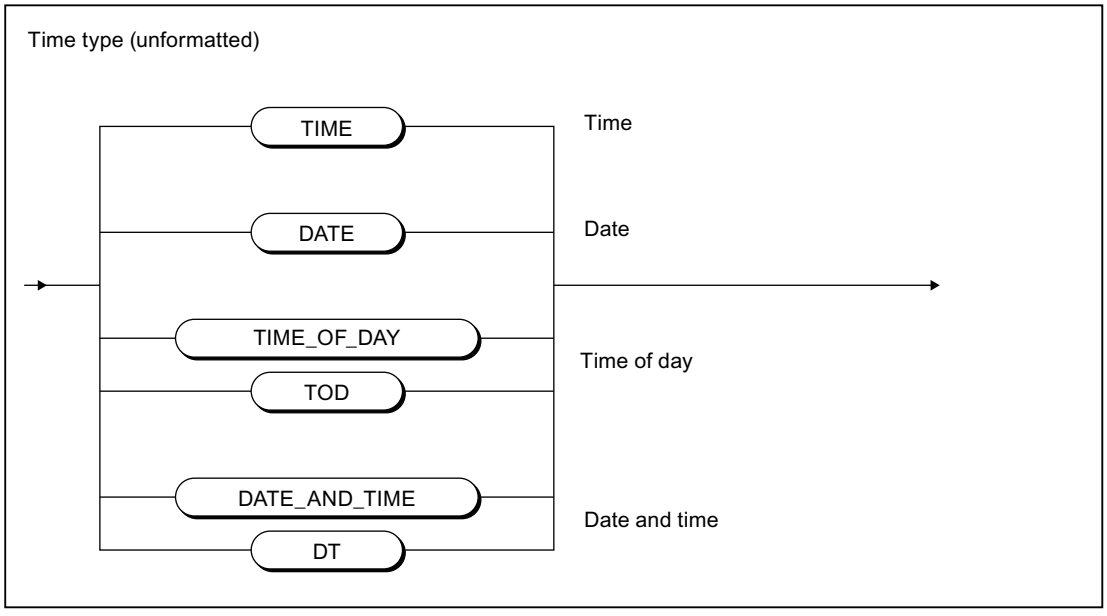


Figure A-79 Time data type

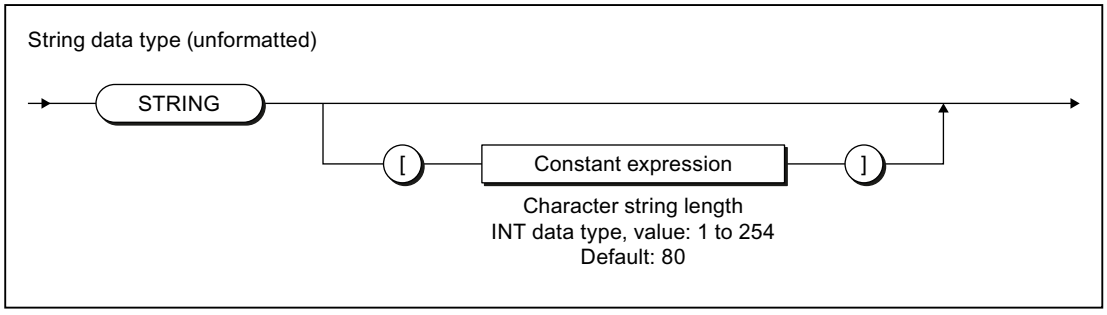


Figure A-80 String data type

User-defined data types

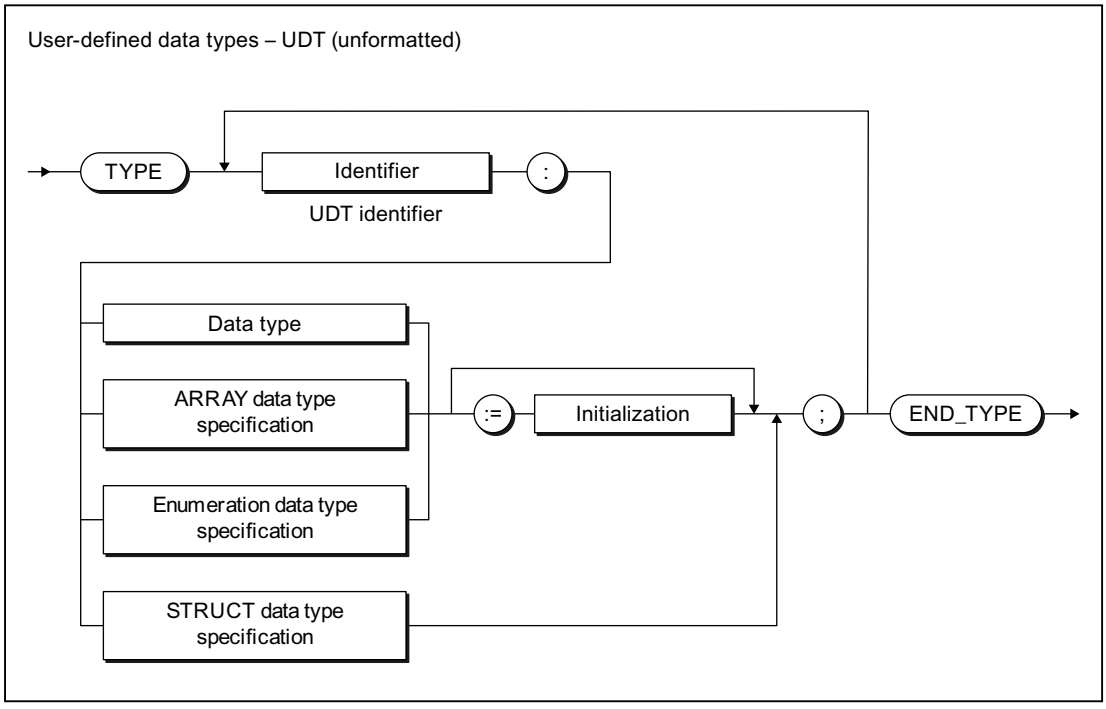


Figure A-81 User-defined data type

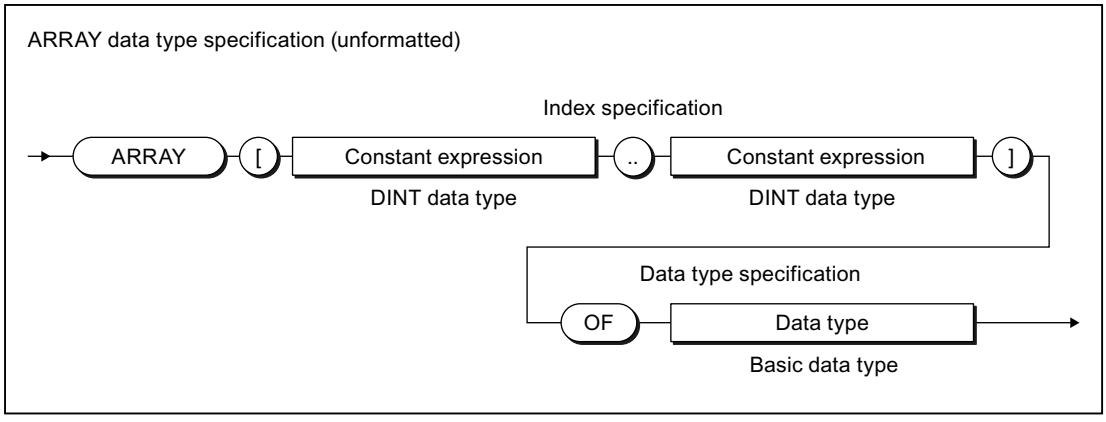


Figure A-82 ARRAY data type specification

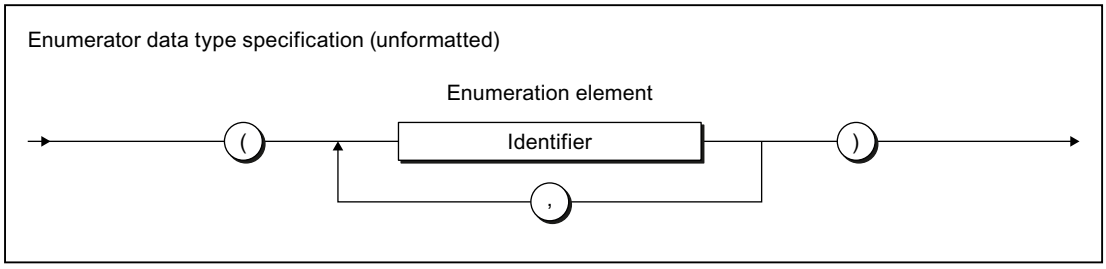


Figure A-83 Enumeration data type specification

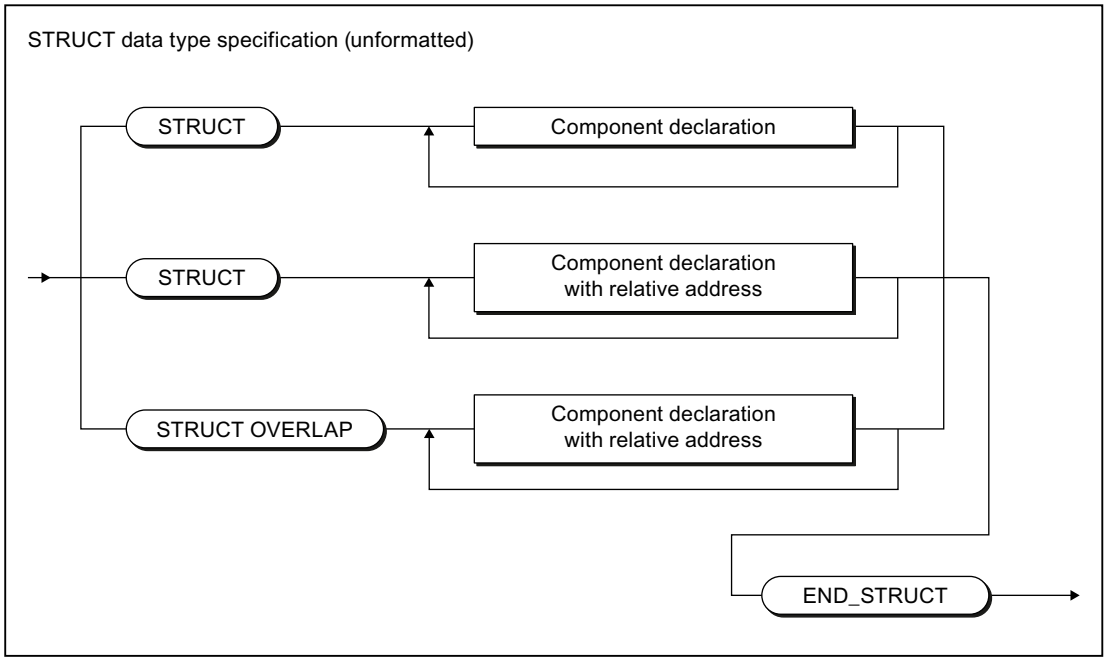


Figure A-84 STRUCT data type specification

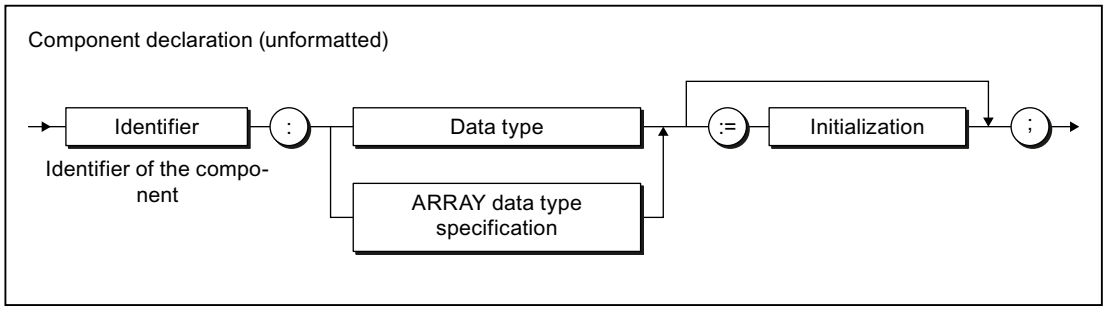


Figure A-85 Component declaration

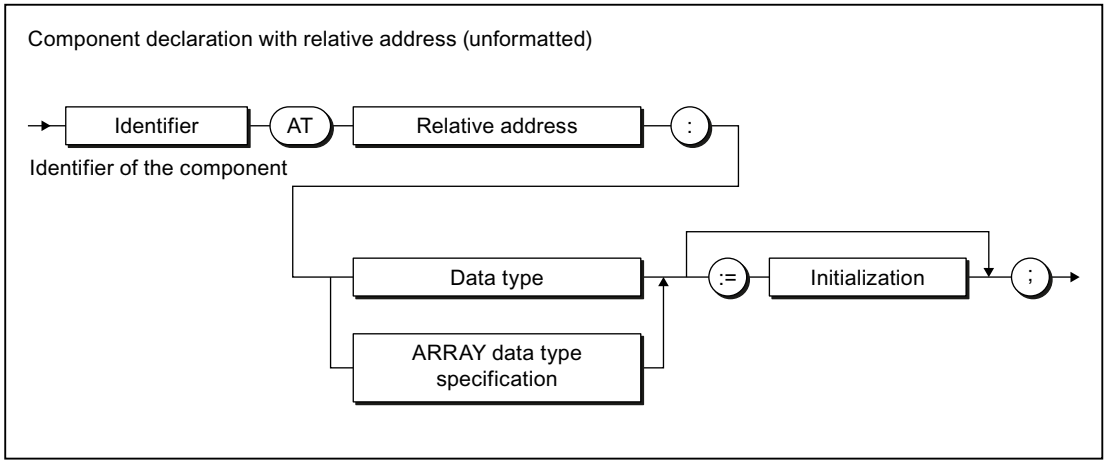


Figure A-86 Component declaration with relative address

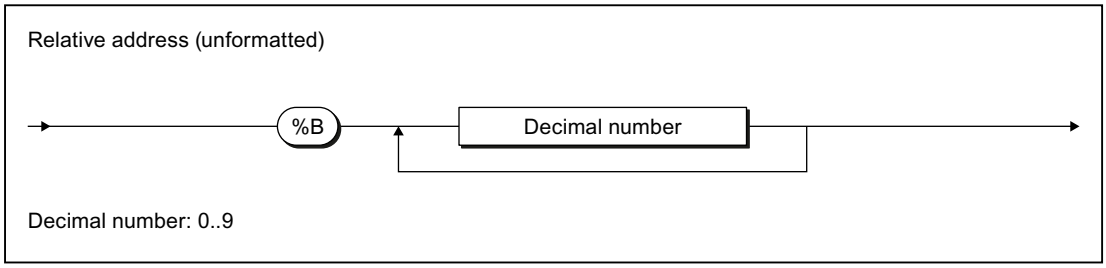


Figure A-87 Relative address

A.1.3.10 Statement section

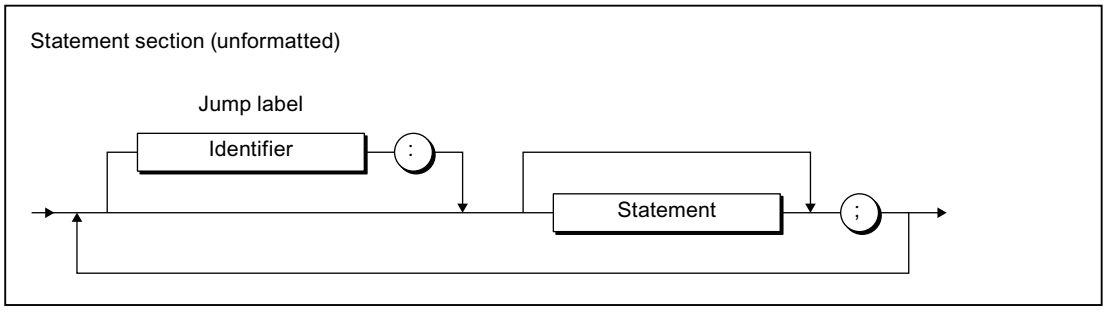


Figure A-88 Statement section

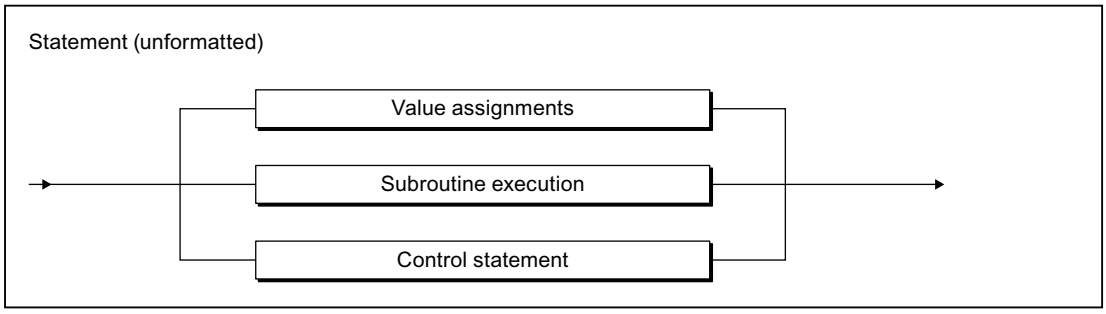


Figure A-89 Statement

A.1.3.11 Value assignments and operations

Value assignment and expression

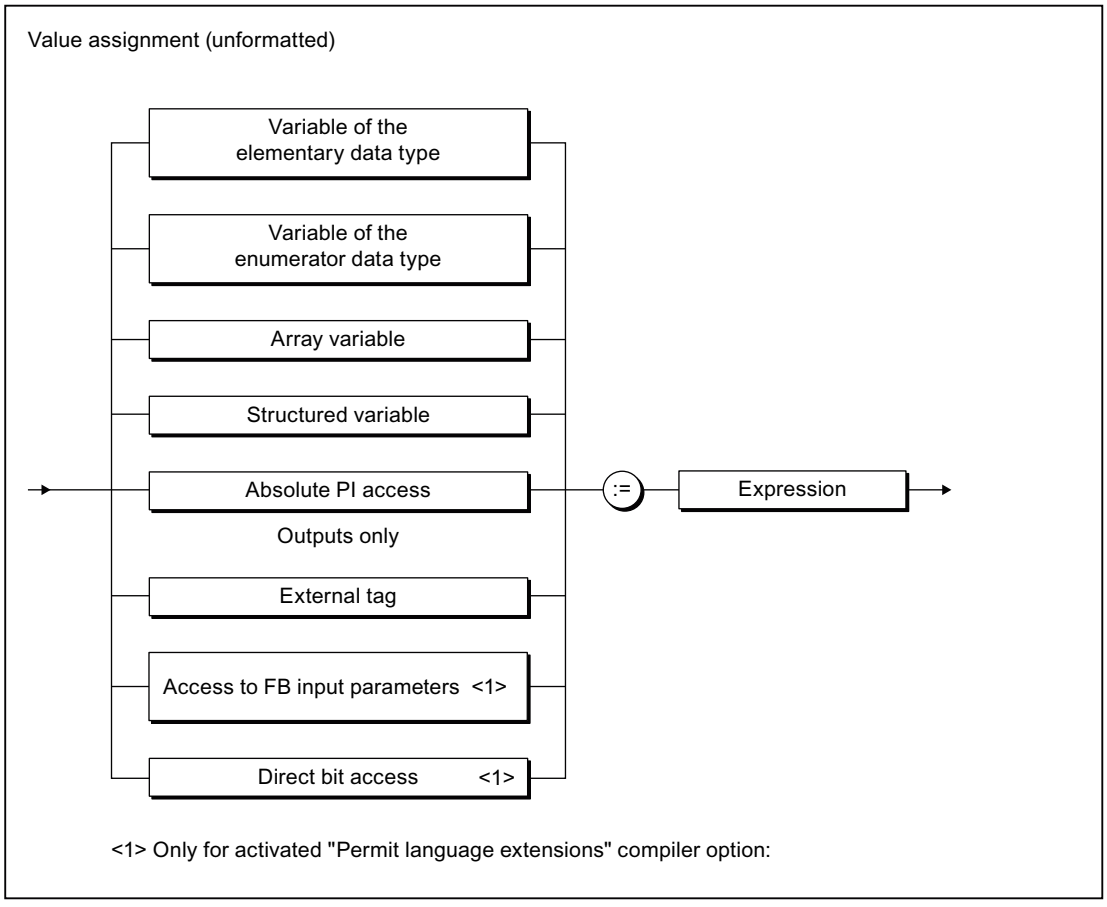


Figure A-90 Value assignments

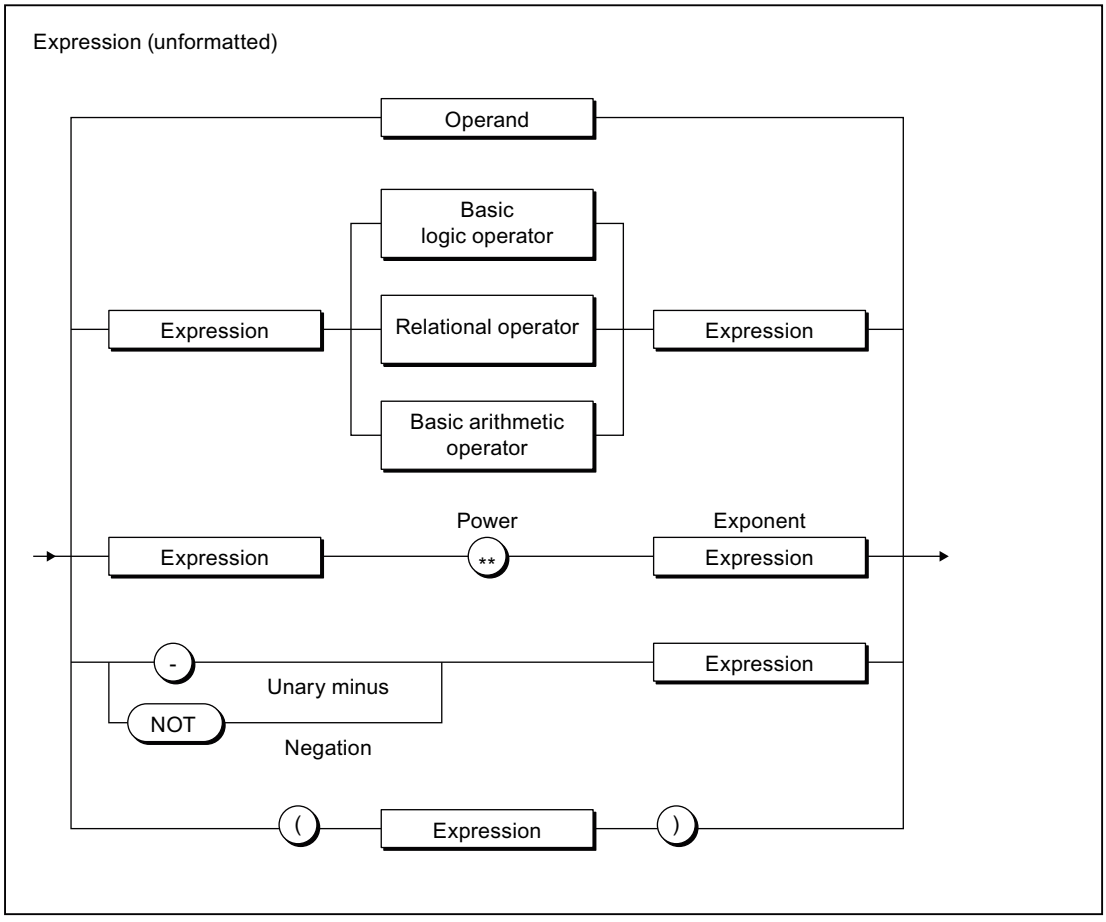


Figure A-91 Expression

Operands

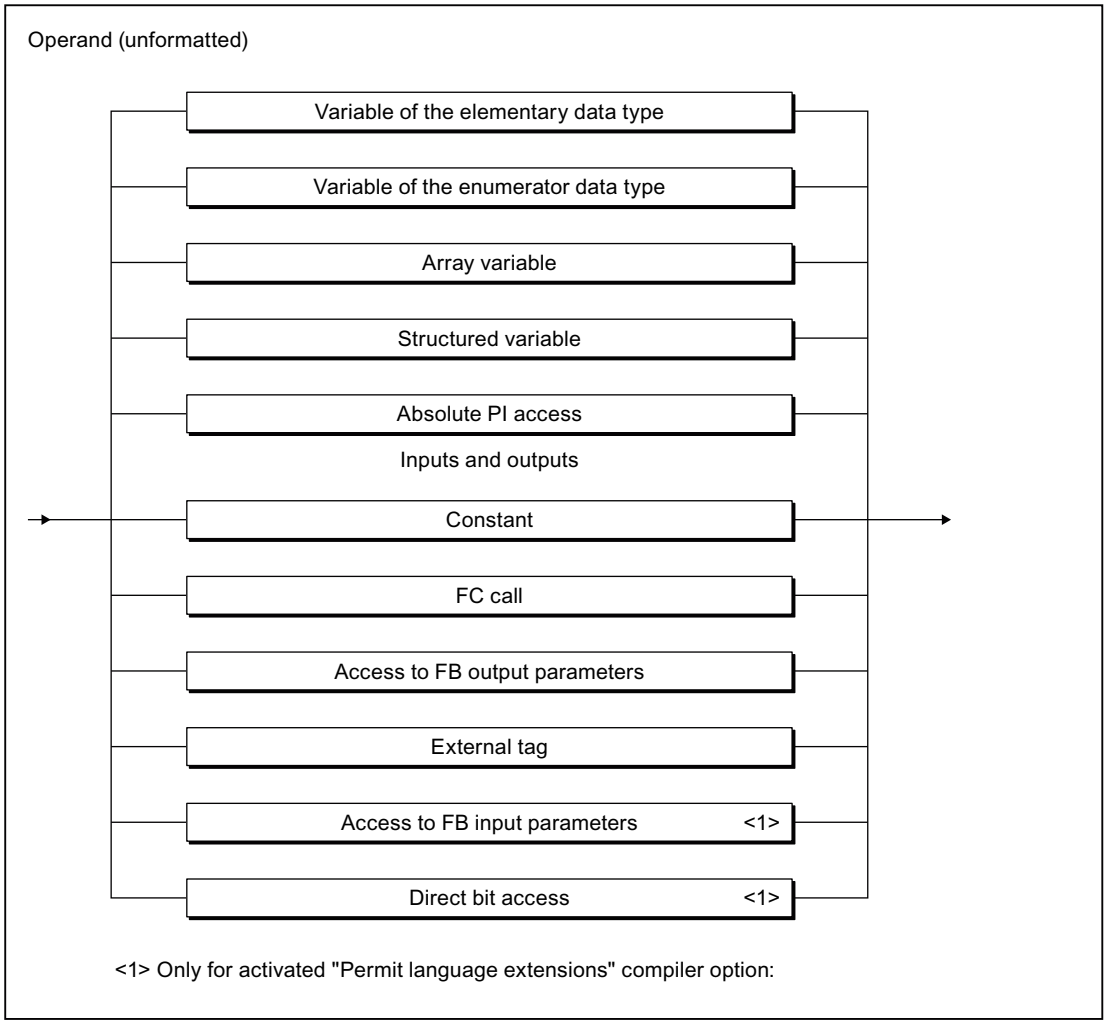


Figure A-92 Operand

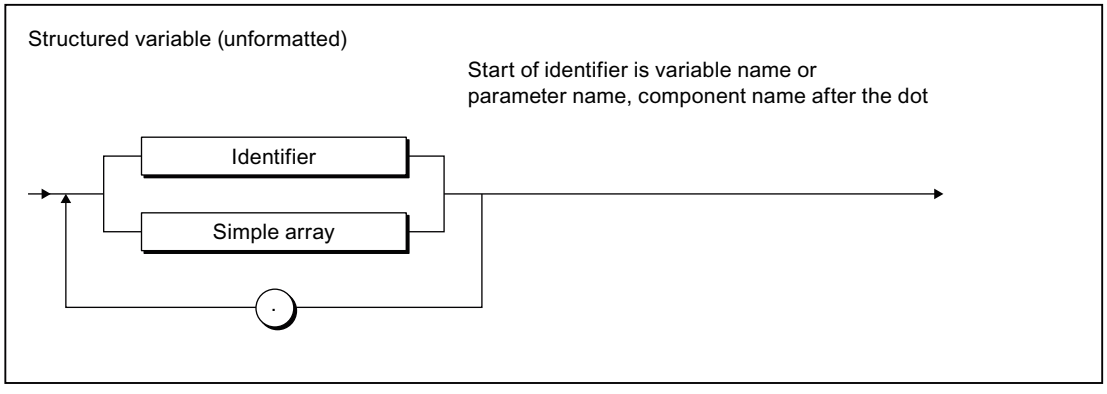


Figure A-93 Structured variable

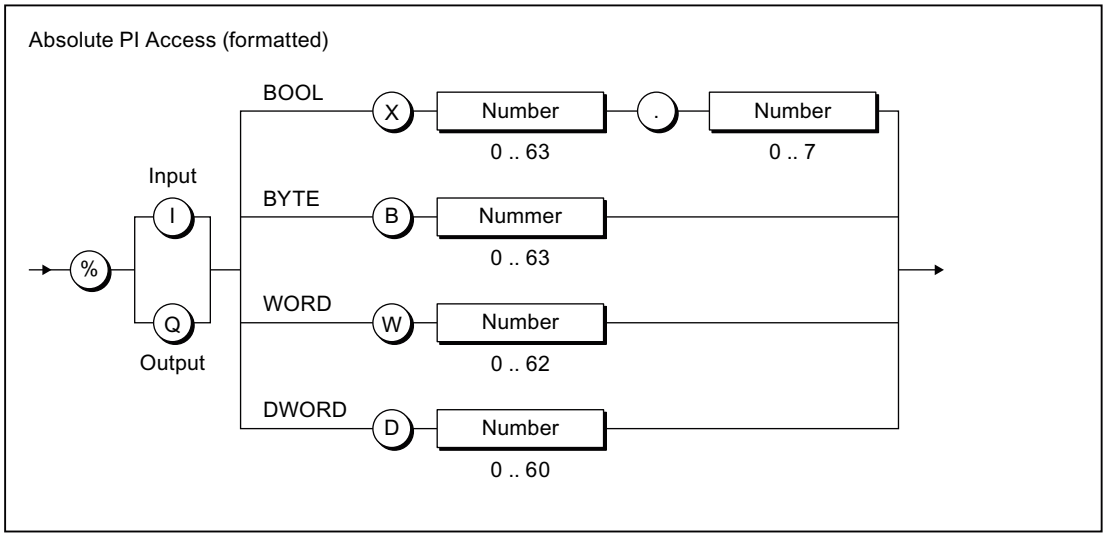


Figure A-94 Absolute PI access

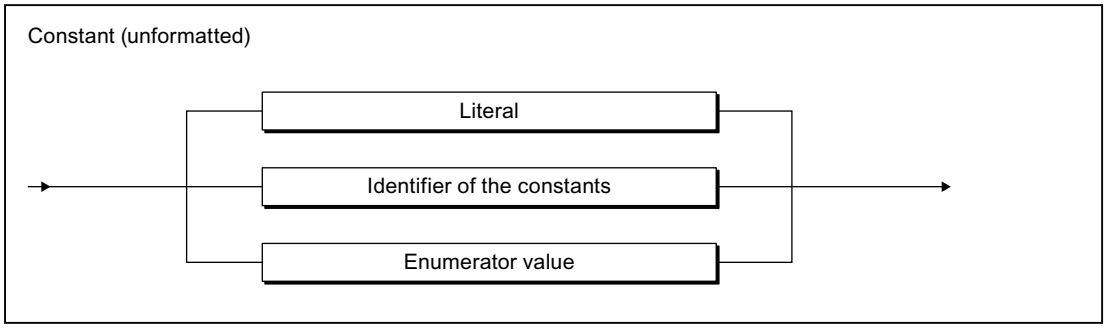


Figure A-95 Constant

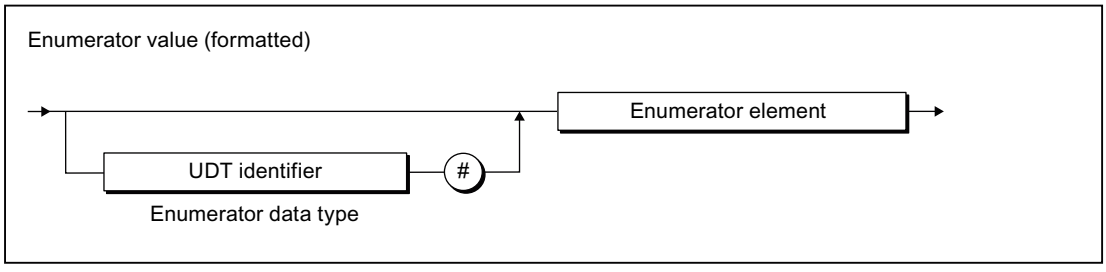


Figure A-96 Enumerator value

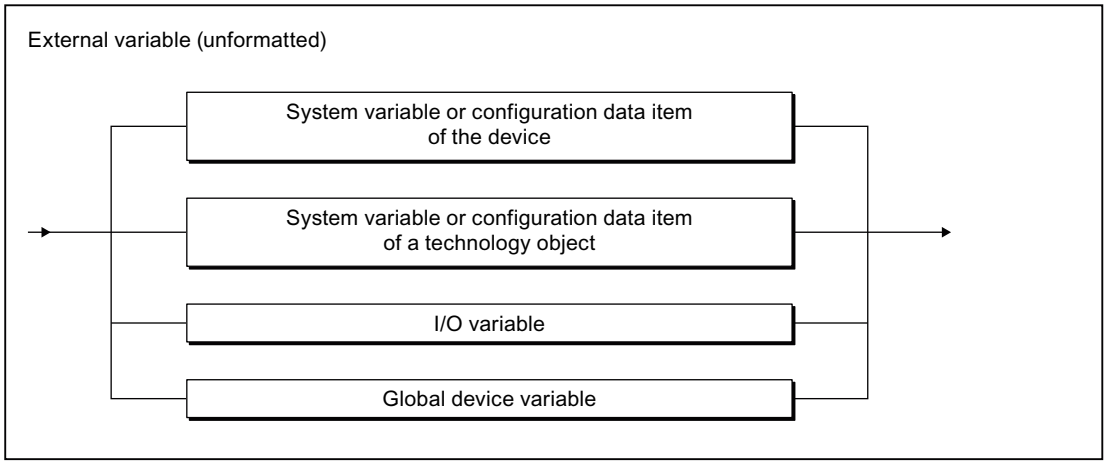


Figure A-97 External tag

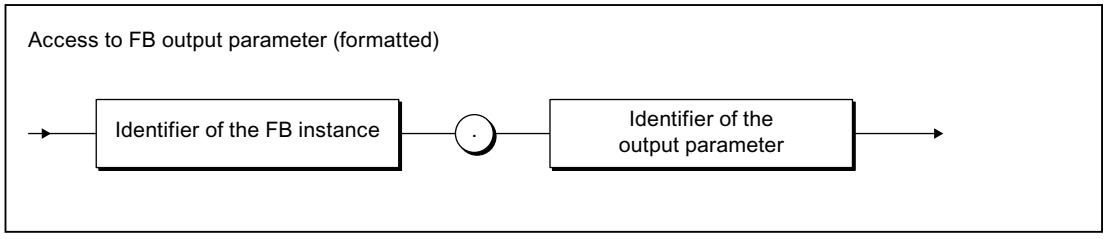


Figure A-98 Access to FB output parameters

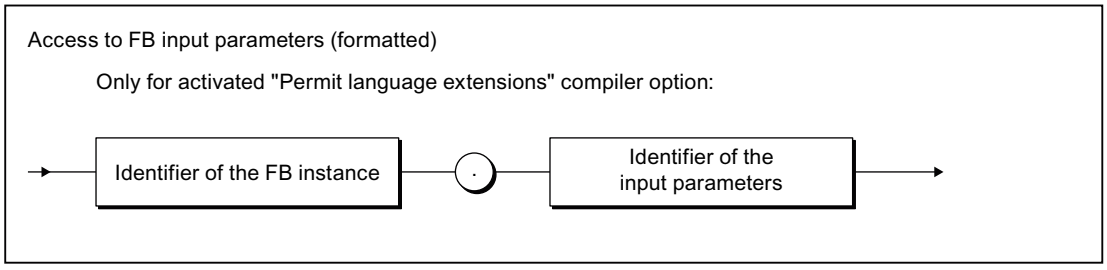


Figure A-99 Access to FB input parameters

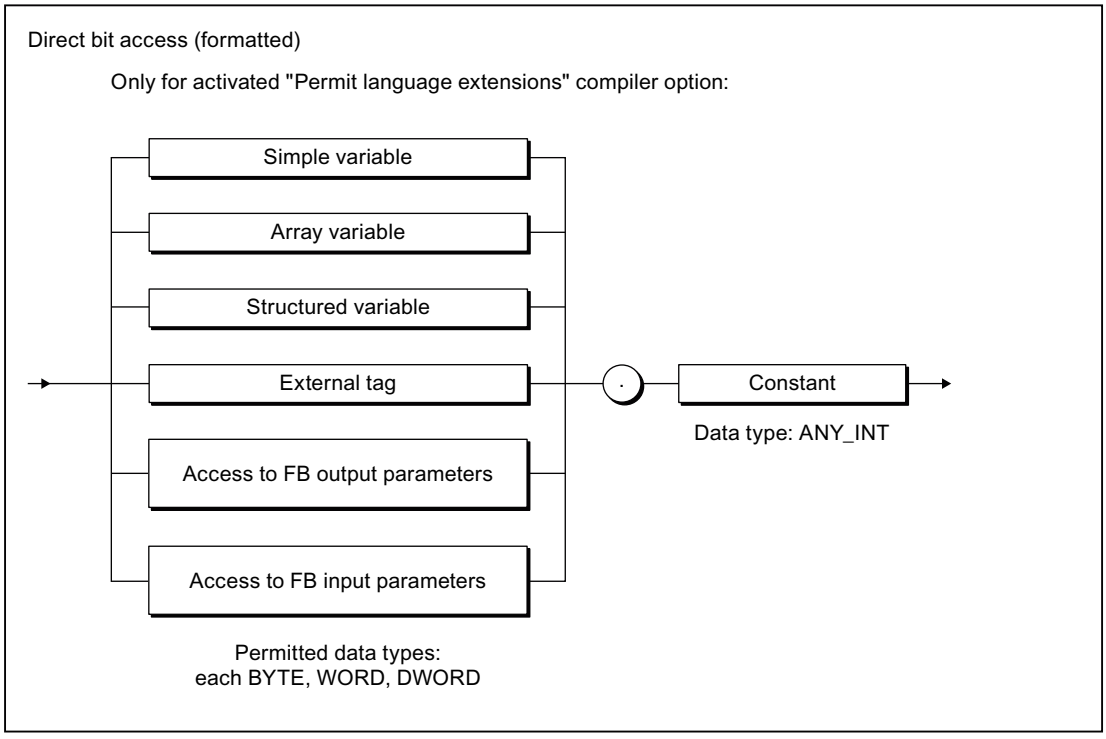


Figure A-100 Bit access

Operators

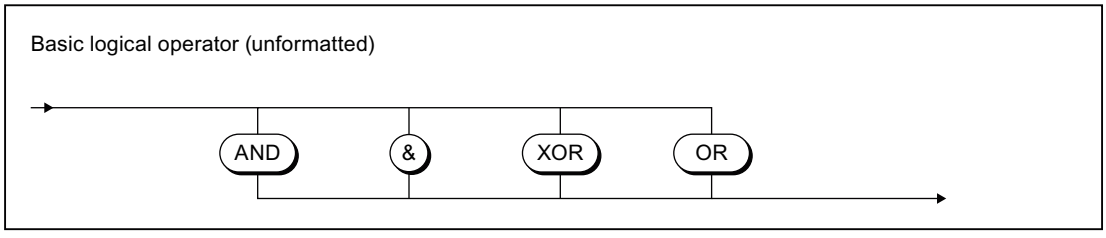


Figure A-101 Basic logic operator

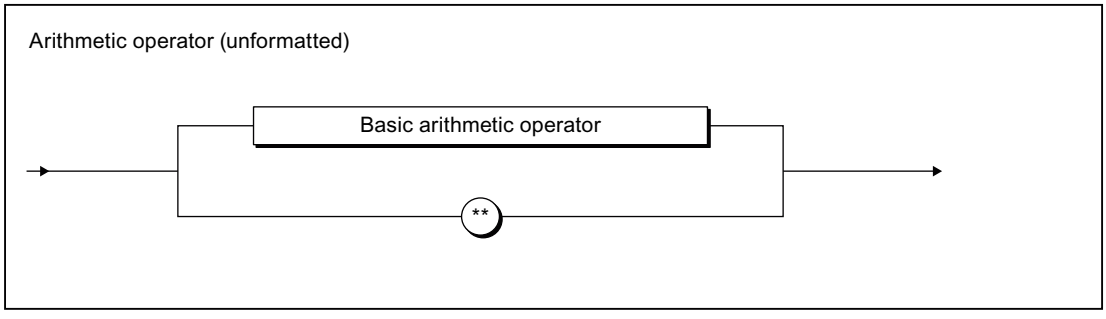


Figure A-102 Arithmetic operator

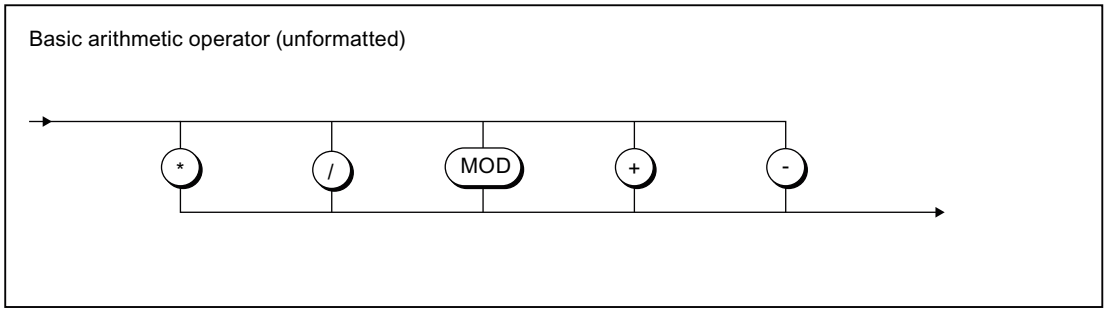


Figure A-103 Basic arithmetic operator

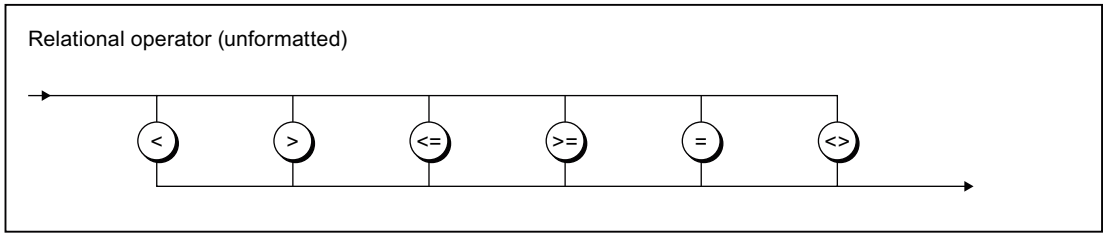


Figure A-104 Relational operators

A.1.3.12 Call of functions, function blocks and methods

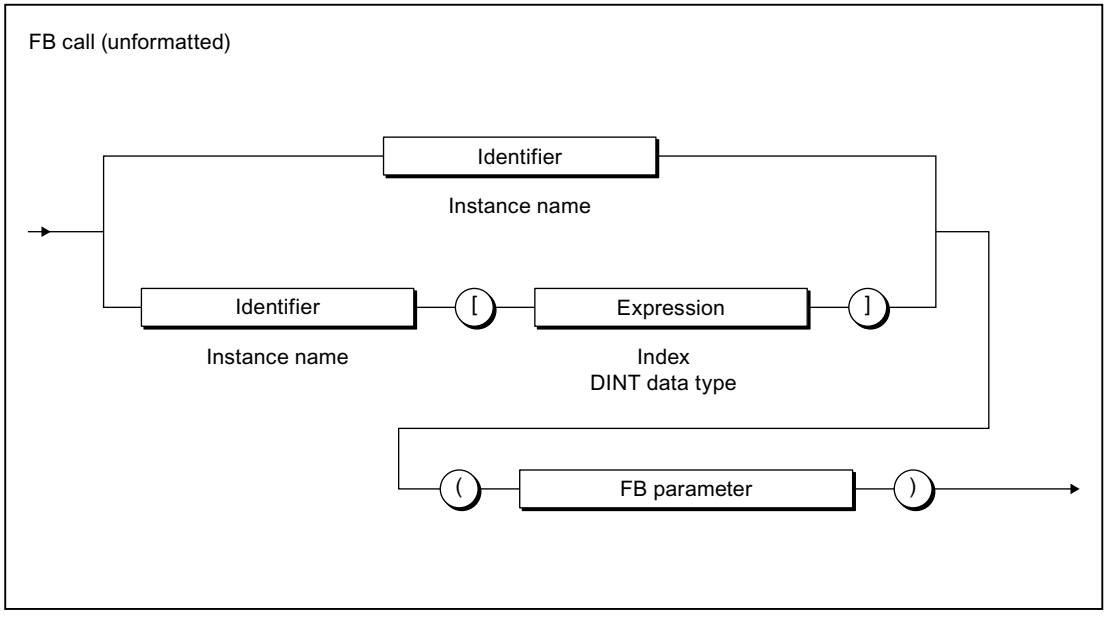


Figure A-105 FB call

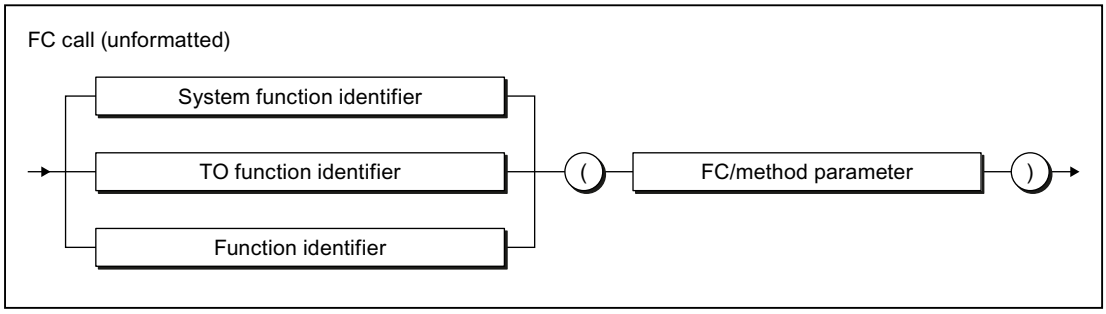


Figure A-106 FC call

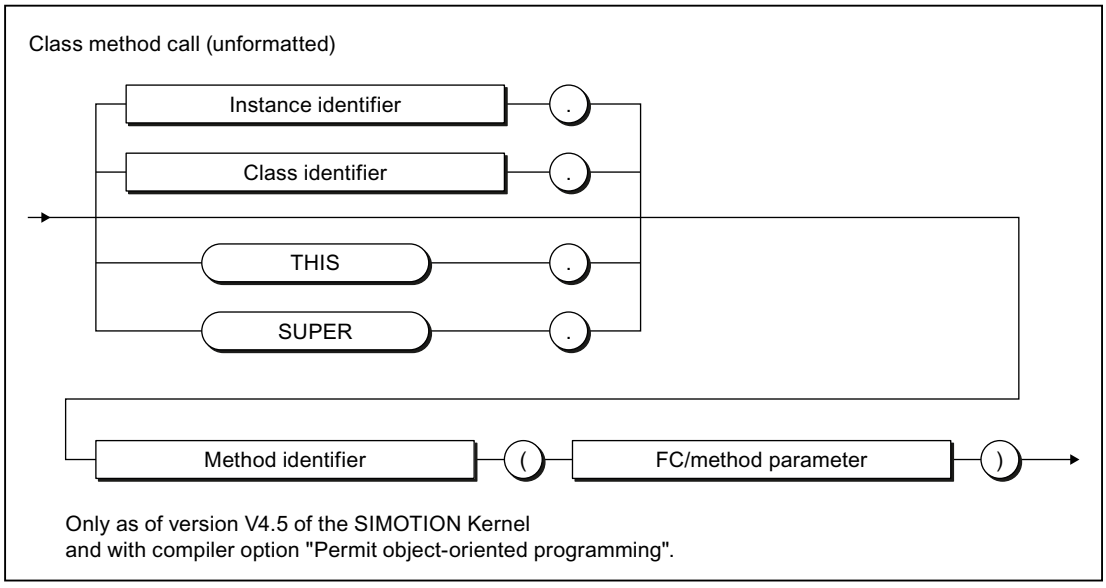


Figure A-107 Class method call

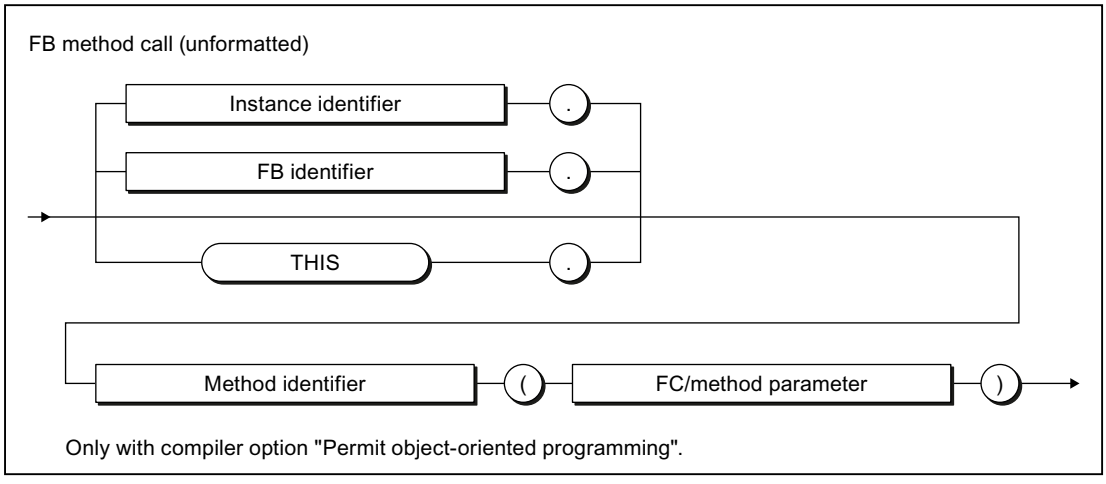


Figure A-108 FB method call

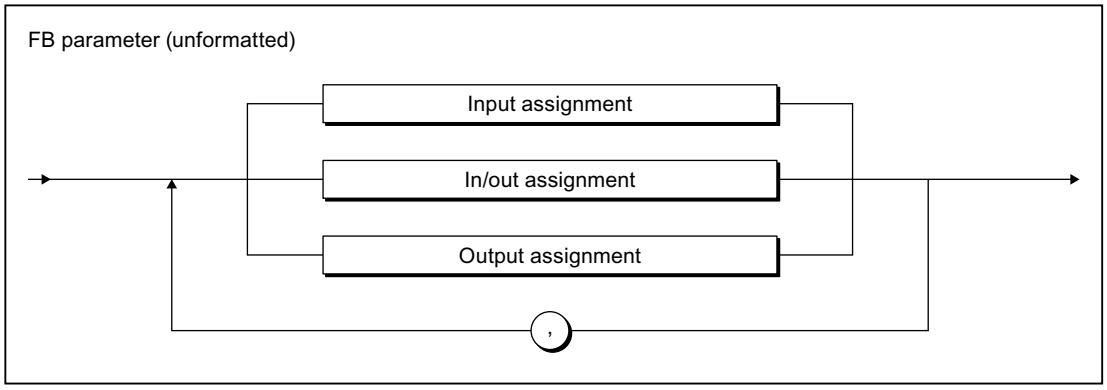


Figure A-109 FB parameter

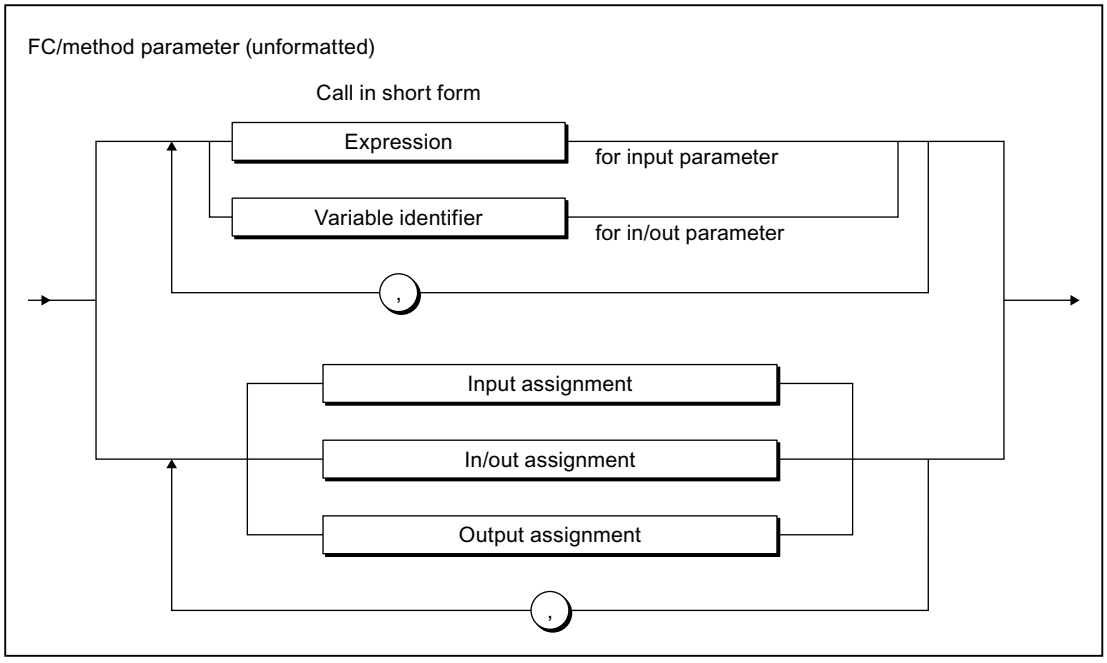


Figure A-110 FC parameter

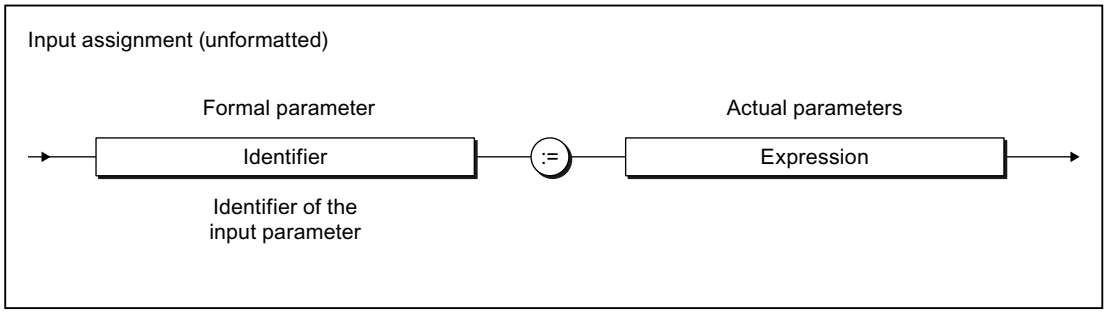


Figure A-111 Input assignment

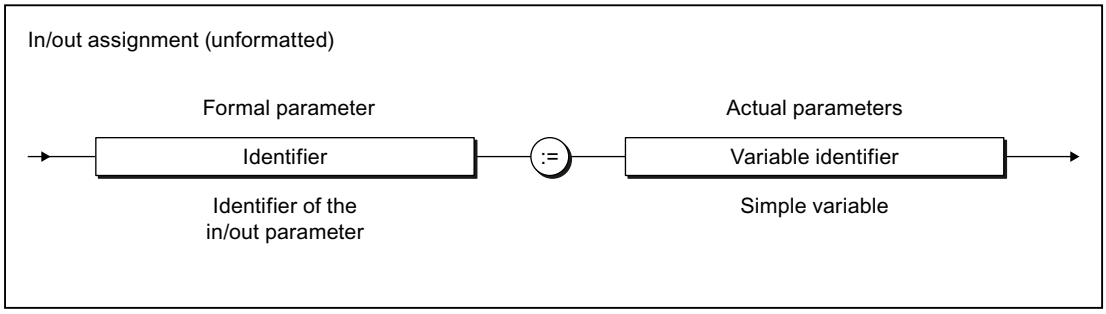


Figure A-112 In/out assignment

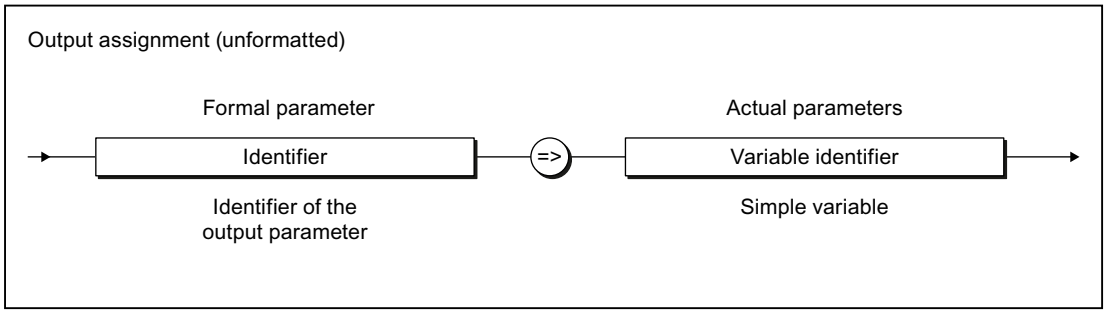


Figure A-113 Output assignment

A.1.3.13 Control statements

Branches

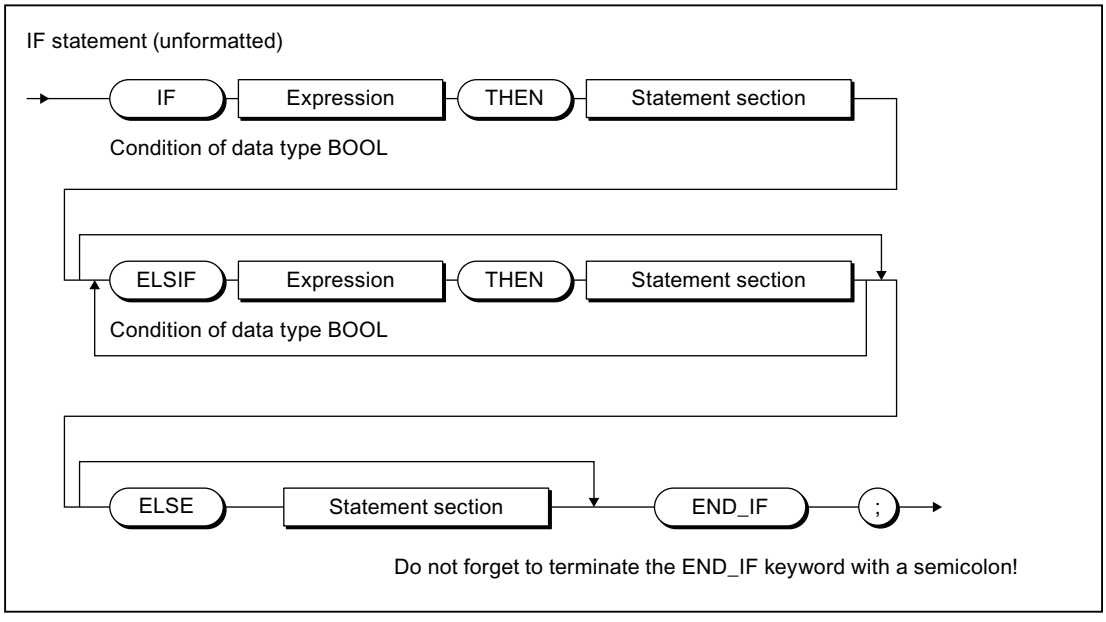


Figure A-114 IF statement

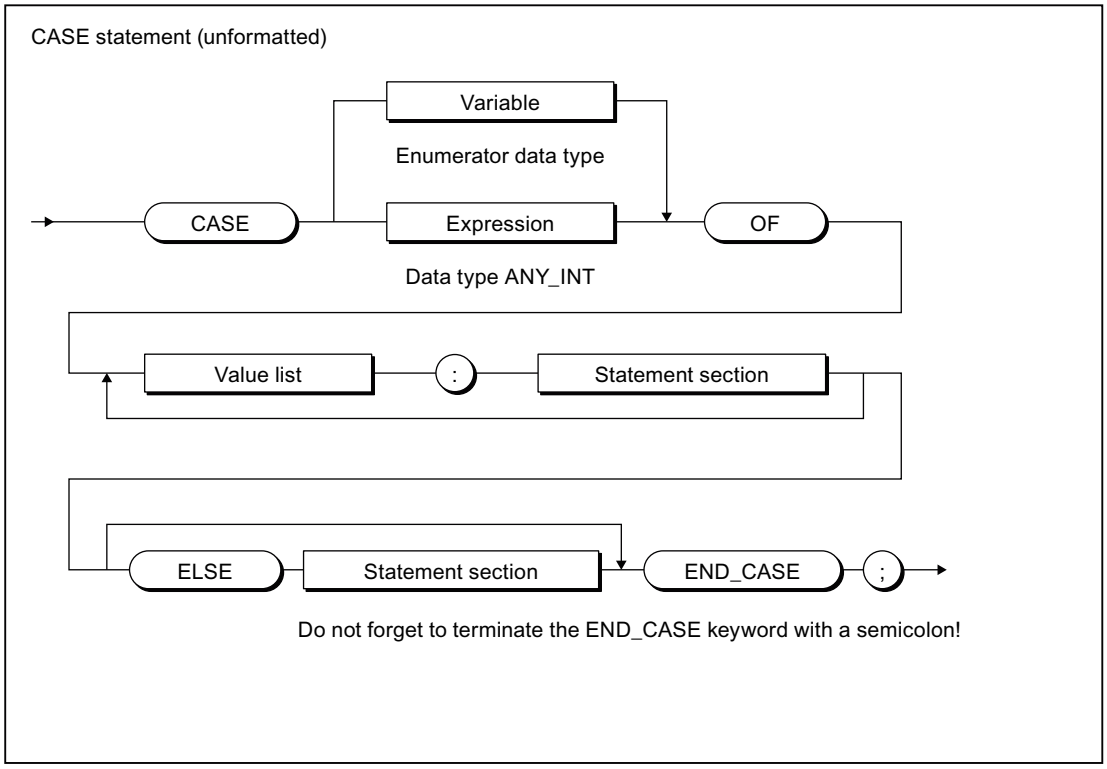


Figure A-115 CASE statement

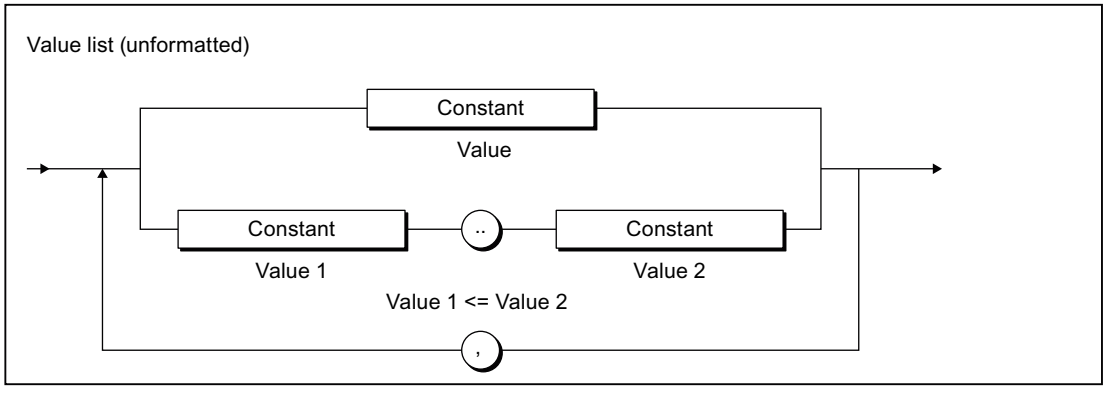


Figure A-116 Value list

Repetition statements and jump statements

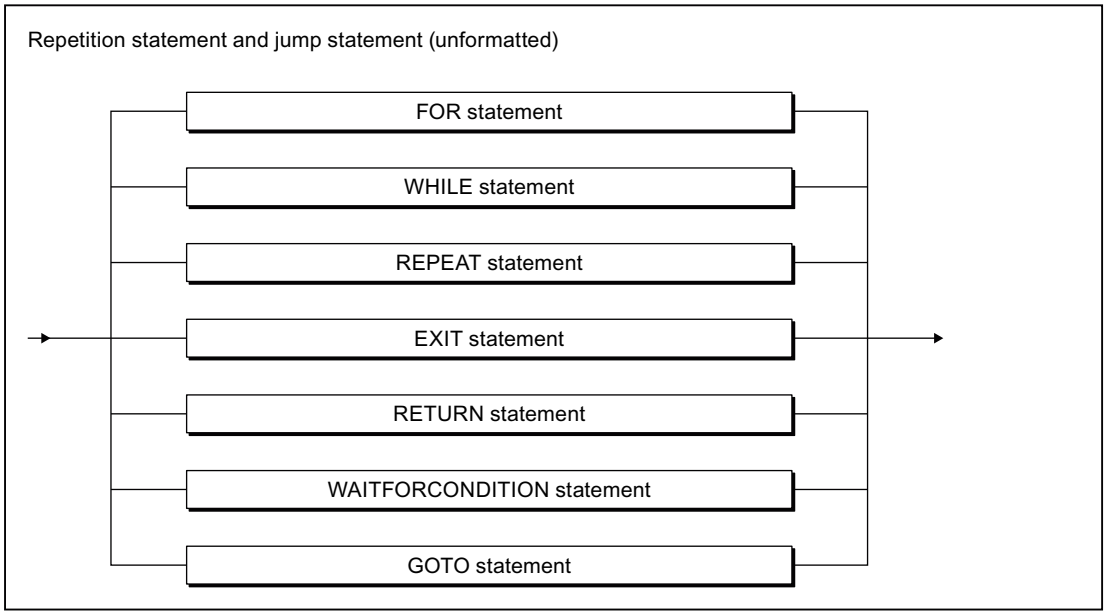


Figure A-117 Repetition statement and jump statements

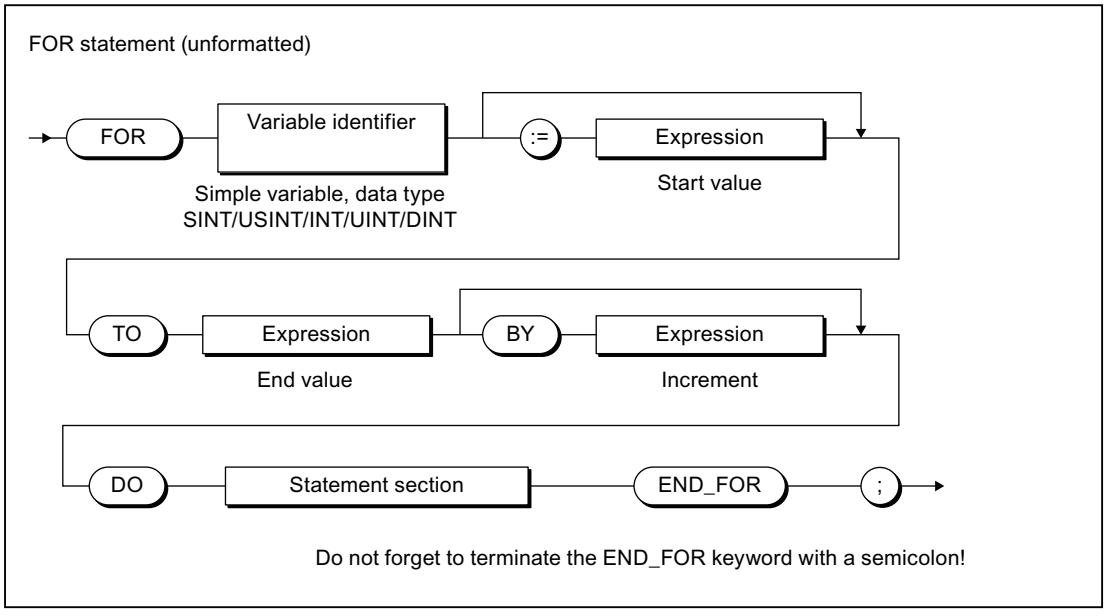


Figure A-118 FOR statement

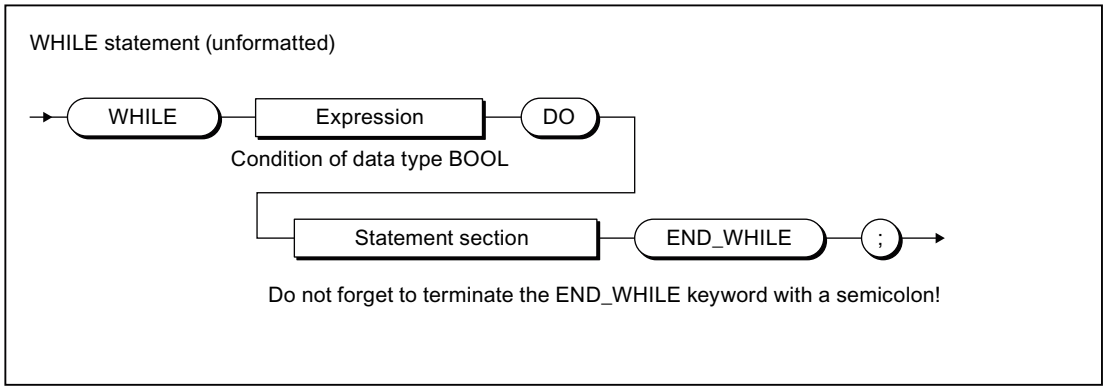


Figure A-119 WHILE statement

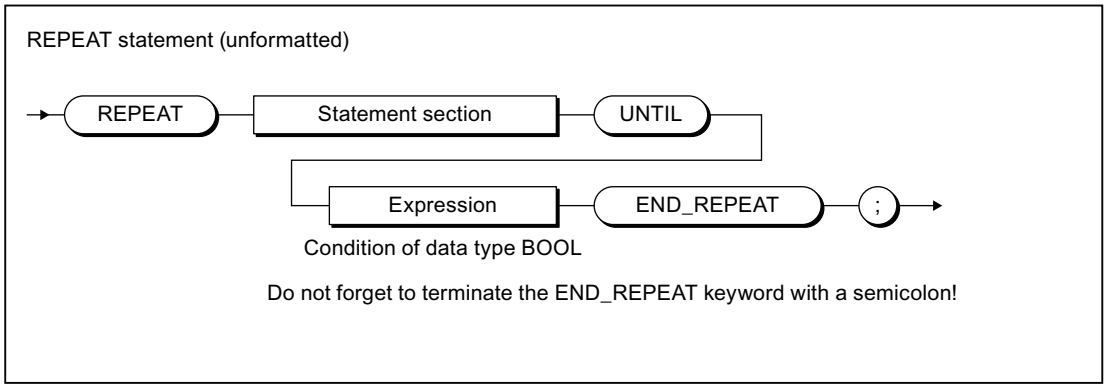


Figure A-120 REPEAT statement

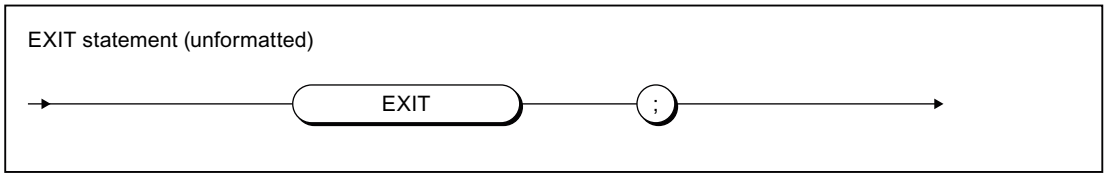


Figure A-121 EXIT statement

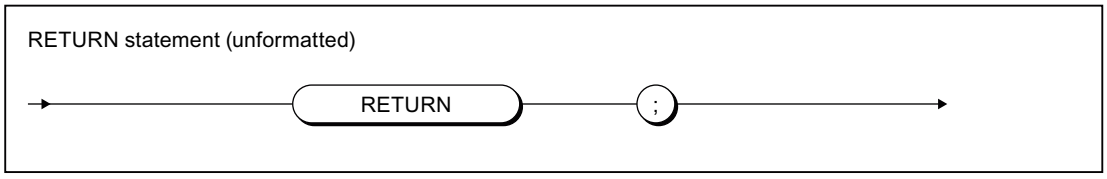


Figure A-122 RETURN statement

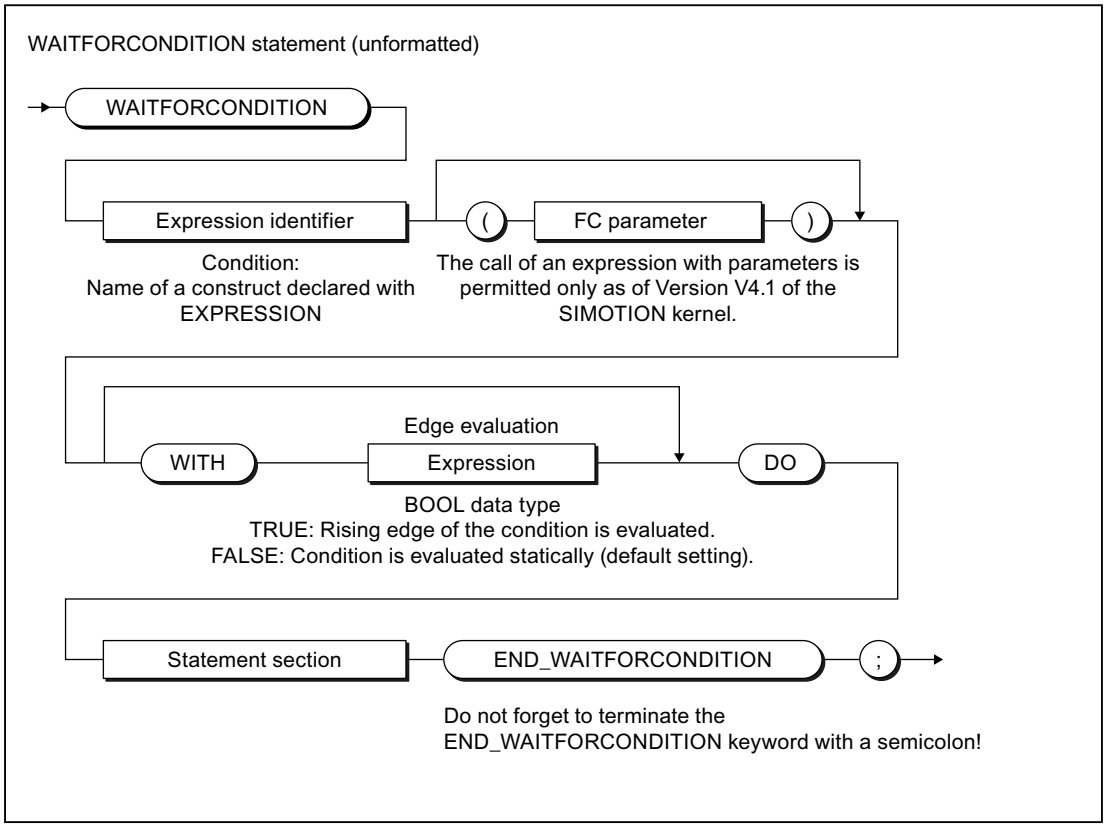


Figure A-123 WAITFORCONDITION statement

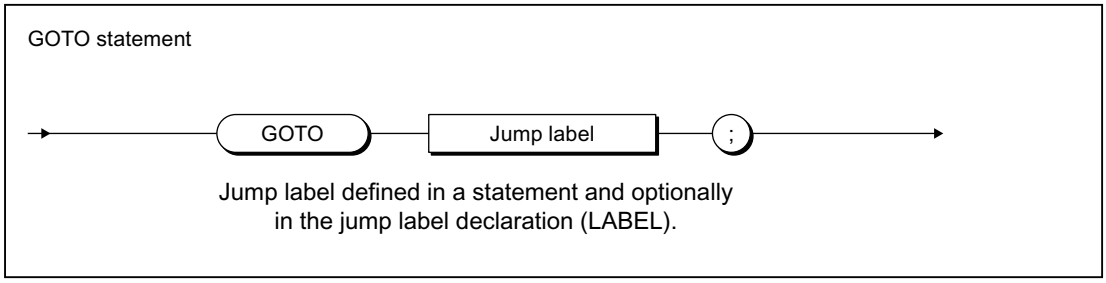


Figure A-124 GOTO statement

A.1.3.14 Pragmas

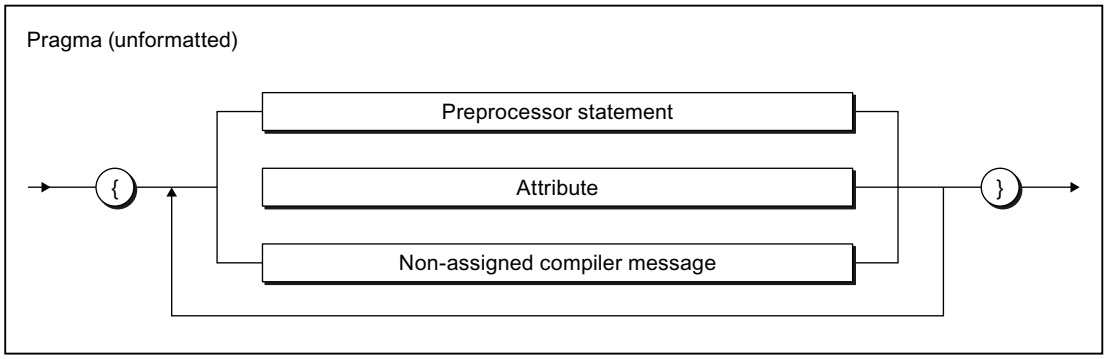


Figure A-125 Syntax: Pragma

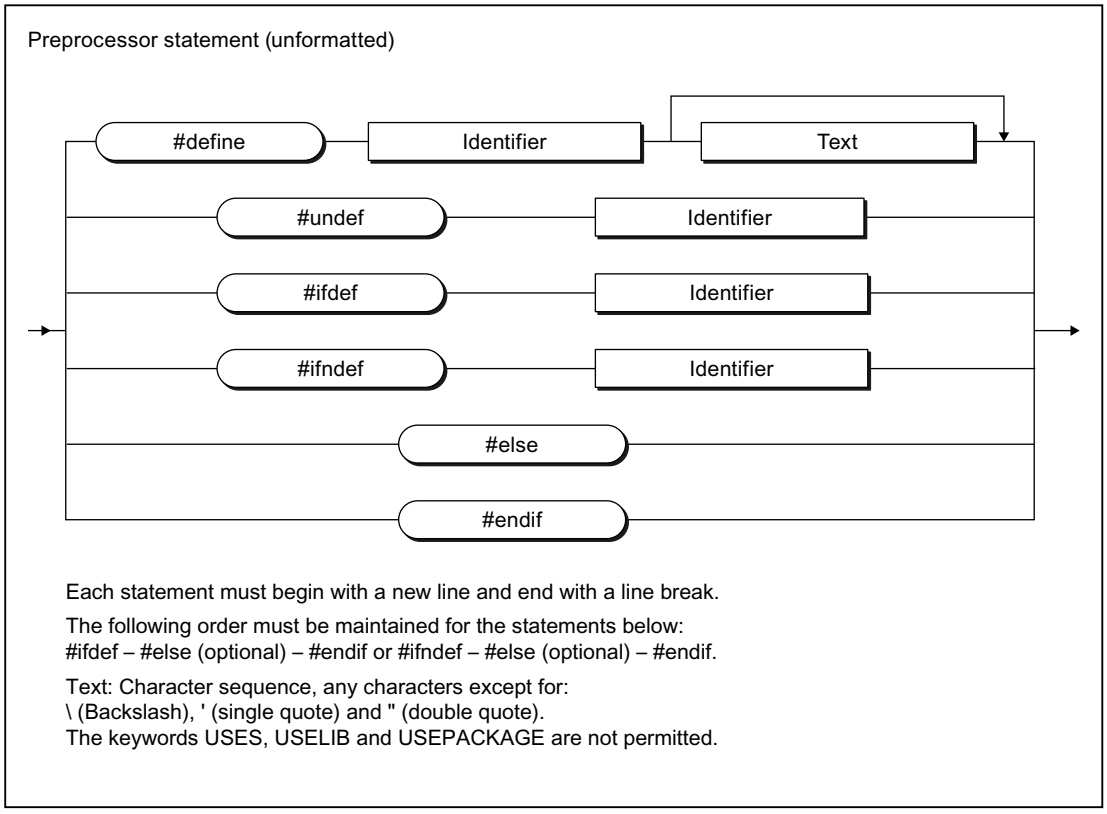


Figure A-126 Syntax: Preprocessor statement

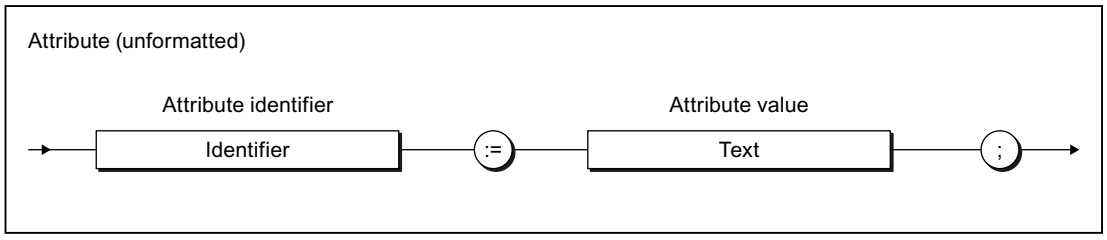


Figure A-127 Syntax: Attributes for compiler outputs

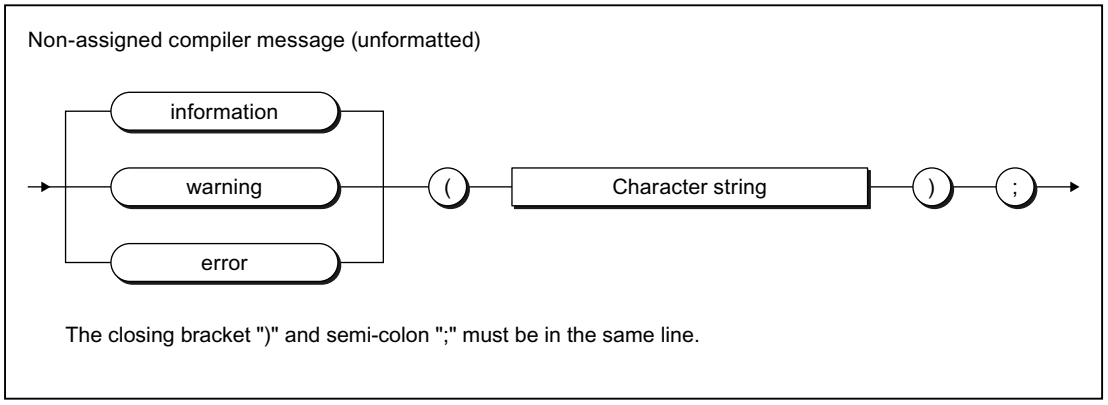


Figure A-128 Syntax: Non-assigned compiler message

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-8 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]] Examples: • -D myident // Definition of myident; this can be queried using #ifdef. • -D myident= // myident is defined as empty character string • -D "myident=This is a text" // myident is defined as character string 'This is a text'. The quotation marks only have to be used if the replacement text contains a blank.

A.2.2 Scanner errors (2001, 2002)

Table A-9 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 ... 3161)

Table A-10 Declaration errors in POU (3002 ... 3100)

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, VAR RETAIN variable declaration blocks are permitted only once for each POU.
3005	TASK statement: The task link has already been made in the source file for the specified task. Further task linking not possible.
3006	Incorrect stack size for task specified. Only positive integers are permitted.
3007	The specified identifier must be a task identifier; see task configuration.
3008	The specified identifier must be a program identifier. The declaration is made in the statement PROGRAM xx ... END_PROGRAM.
3009	The EXPRESSION keyword must be followed by an identifier. The declaration is made in the statement EXPRESSION xx ... END_EXPRESSION.
3010	The specified identifier is not an EXPRESSION identifier. Check whether the declaration was made using the statement EXPRESSION xx ... END_EXPRESSION.
3011	The TASK statement is not permitted in the unit. Use the task configuration in the Workbench.
3012	The specified identifier has already been declared at another position. It cannot be used again as a function identifier.
3013	The specified identifier has already been declared at another position. It cannot be used again as a function block identifier.
3014	The UNIT statement is expected. The following forms are permissible: • UNIT myunit; • UNIT myunit : dvtype; The UNIT statement is only required when compiling at the ASCII file level. It is optional when the compiler is called from the Workbench.
3015	The source file is not ended with END_IMPLEMENTATION. Observe the structure for a source file!
3016	No further statements may be specified after keyword END_IMPLEMENTATION.
3017	The task declaration is not ended with END_TASK. Observe the structure for a source file!
3018	The POU declaration is not ended with END_FUNCTION, END_FUNCTION_BLOCK, or END_PROGRAM. Observe the structure for a source file!
3019	A POU starting with keywords FUNCTION, FUNCTION_BLOCK, or PROGRAM is expected.

Error	Description
3020	The task linking statement is expected. Structure: TASK tname ... END_TASK;
3021	Methods can only be declared within classes, function blocks or interfaces. In this case, no further statements may be inserted between the variable or data type declaration and the methods.
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.
3028	The specified identifier has already been declared at another position. It cannot be used again as a class identifier.
3029	The specified identifier has already been declared at another position. It cannot be used again as an interface identifier.
3040	The specified identifier does not reference a namespace. It cannot be used in the USING statement.
3041	The specified identifier references a namespace that cannot be accessed. The namespace is declared INTERNAL.
3042	The specified identifier references a namespace whose symbols could not be added to the current context.
3043	The specified identifier references a namespace used system-internal. The identifier cannot be used as user namespace.
3050	A class definition starting with the keyword CLASS is expected.
3051	The class definition does not end with END_CLASS. Observe the structure for a source file.
3052	A method definition starting with the keyword METHOD is expected.
3053	A method definition ends with END_METHOD. The method definition contains a syntax error. Observe the structure for a source file.
3060	It is not permissible to use the specified access identifier at this position. For example, the access identifier PROTECTED cannot be used within a FUNCTION_BLOCK.
3061	The specified method property cannot be accepted. For example, OVERRIDE, ABSTRACT or FINAL cannot be declared for a method within a FUNCTION_BLOCK. OVERRIDE can be used for methods in CLASS only if a base class exists in which the method has already been declared. In this case, OVERRIDE cannot be declared for PRIVATE methods.
3062	If a method is overridden in a derived class, the access identifiers must be identical. Accordingly, a PROTECTED method can only be overridden with PROTECTED and a PUBLIC method only with PUBLIC.
3063	If a method is to be overridden in a derived class, it is absolutely essential to specify the keyword OVERRIDE in the method declaration.
3064	A method that is already defined in the base class cannot be overridden with "ABSTRACT".
3065	The keyword "OVERRIDE" is used in the declaration of the specified method. Since the method is not defined in the base class, it is not permissible to specify this keyword.
3066	The method signatures are different in the specified classes. This is not compatible with override mechanisms. For example, not all the parameters have the same name or they are defined in a different sequence or they have different data types. Even the return value must have the same data type.

Error	Description
3067	If a method is declared with the PUBLIC identifier in a base class and a method of the same name is required by an object-oriented interface to be implemented, this method must already be declared PUBLIC in the base class. Methods in object-oriented interfaces are always PUBLIC. It is not possible to change the access level for derivations.
3068	A method that is already defined in the base class and identified as "FINAL" cannot be overridden.
3069	A method cannot be declared ABSTRACT if the class is identified as FINAL. This combination cannot be used because it is not possible to create any instances of this class.
3070	The specified data type cannot be used as a base class or a base interface. A CLASS can only ever be derived from another CLASS. An INTERFACE can only ever be derived from another INTERFACE. It is not possible to use mixtures of the two or to create derivations of other data types.
3071	The specified CLASS is identified with the keyword FINAL. A CLASS identified as FINAL cannot be used as a base class.
3072	The specified CLASS is not exported in the interface section of the unit or stems from a unit imported in the implementation section and cannot therefore be used as a base class of an exported CLASS. • If the CLASS mentioned in the error message is declared in the current unit, it must be exported in the interface section. CLASS <classname>; must be added there for this purpose. • If it is a CLASS from another unit, the USES statement for this unit must be moved from the implementation section to the interface section.
3073	The specified CLASS is identified with the keyword FINAL. However, the relevant class does not implement all of the methods (in base classes) identified as ABSTRACT and therefore cannot be instantiated. All ABSTRACT methods must either be implemented in the class, or the class cannot be identified as FINAL.
3075	The specified identifier does not refer to an INTERFACE. Only INTERFACE identifiers can be used for IMPLEMENTS.
3076	The specified INTERFACE is already implemented by the CLASS itself or by the base class(es). An INTERFACE can only ever be implemented once by a CLASS.
3077	The specified INTERFACE is not exported in the interface section of the unit or stems from a unit imported in the implementation section and cannot therefore be used at an exported CLASS (IMPLEMENTS). • If the INTERFACE mentioned in the error message is declared in the current unit, this declaration must be included in the interface section. For this purpose, the entire declaration must be moved to the interface section. • If it is an INTERFACE from another unit, the USES statement for this unit must be moved from the implementation section to the interface section.
3078	The specified INTERFACE is identified as a prototype. This is a programming error. Object-oriented interfaces must always be defined in full.
3080	A method that is identified as PRIVATE cannot be declared as either ABSTRACT or FINAL. It must always be implemented and no override mechanisms of any kind may be used.
3081	A method identifier is concealing a global identifier. This is only permissible if you activate the compiler option 'Permit forward declarations'. This ensures that calls of this method within the current CLASS or the current FUNCTION_BLOCK without a preceding name refer to this method in all other methods. The global identifier can no longer be accessed in the class because it is concealed by the method identifier.
3100	The specified identifier is already defined and identified as public in the base class. It cannot be declared again in the derived class.

Error	Description
3150	The specified identifier has already been identified as public without INTERNAL being specified. It is not possible to declare it INTERNAL again differently. • If this situation occurs for a POU, this POU has already been identified as public without INTERNAL in the INTERFACE section of the source. Add the INTERNAL identifier here to use the POU only within the namespace. • If this situation occurs for a namespace, an identically named namespace declaration from a source or library imported via USES/USELIB may also be present. In this case, all declarations must be either INTERNAL or not INTERNAL.
3151	The specified identifier has already been declared as INTERNAL in the current source or in an imported source or library. The current declaration location, however, does not contain the INTERNAL identifier. Add INTERNAL for the declaration of the affected element.
3160	The specification of USING within a namespace is possible only for the first declaration within the source and then applies to all blocks of the affected namespace. Move the USING statement to this position.
3161	The specification of USING within a namespace or a POU is permissible only for namespaces not contained in the declaration path. For example, a namespace NS1 or class, function, etc., declared in NS1, may not itself contain any USING statement for NS1.

A.2.4 Declaration errors in data type declarations (4001 ... 4105)

Table A-11 Declaration errors in data type declarations (4001 ... 4105)

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;

Error	Description
4012	The specified data type cannot be used at the current position. • When compiler option "Permit object-oriented programming" is not activated, type identifiers declared locally in the POU may not be used with VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT because they must also be known outside the POU for parameter transfer purposes. • When compiler option "Permit object-oriented programming" is activated, the local data types used for variables, parameters and other type declarations must have at least the same access definition as that applicable at the point of use. For example, a local data type used at the transfer parameters of a PROTECTED method must itself be declared as at least PROTECTED. PRIVATE data types may only be used at local variables.
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. 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.
4100	The definition of the structure component requires the specification of a user-defined offsets. If offsets are explicitly specified in a structure definition, the offset specification is necessary for all structure components. Furthermore, the keyword OVERLAP requires the explicit specification of the offsets.
4101	The offset specified in the definition of the structure component is not allowed because a component without an explicitly specified offset has already been defined. If offsets are explicitly specified in a structure definition, the offset specification is necessary for all structure components.
4102	The offset value specified in the definition of the structure component is not allowed. The value must be a multiple of the number specified in the error message so that the data elements are in the correct alignment.
4103	The offset values declared in the structure definition result in overlaps in the memory layout. This is only allowed for identification of the structure as OVERLAP.
4105	The overlaps in the memory space are not allowed for the following data types: STRING, ANYOBJECT and all TO-references derived thereof!

A.2.5 Declaration errors in variable declarations (5001 ... 5509)

Table A-12 Declaration errors in variable 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.

Error	Description
5006	Check the number of initialization values for array initialization.
5007	Syntax error in the specification of the time and date literals.
5008	An instance of a function block or a class cannot be created at the specified position. For example, instances of function blocks or classes cannot be created in functions or methods. The output parameters (VAR_OUTPUT) of function blocks cannot be FB instances. Furthermore, classes and function blocks that have been compiled under the compiler option "Permit object-oriented extensions" cannot be used as input parameters (VAR_INPUT). It is not possible to copy classes and function blocks of the kind that would be created if they were to be used in this way.
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.
5011	The data type stated in the declaration cannot be accessed. It is not identified as public. The data type must be declared as PUBLIC to allow external access and PROTECTED to allow access from a derived class.
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.
5030	If it is necessary for variable declarations and POU implementations to be compiled in an optional sequence in the implementation section of the source file, the compiler option "Permit forward declarations" must be activated.
5040	The specified data type cannot be used within a declaration of retentive variables (VAR RETAIN, VAR_GLOBAL RETAIN) because it already contains retentive elements (VAR RETAIN).
5041	The specified data type cannot be used within a declaration of retentive variables (VAR RETAIN, VAR_GLOBAL RETAIN). The data type does not contain any information that can be backed up or restored in the retentive memory. This message is issued, for example, if a class or a function block does not contain instance data of any kind. This message is also output for variables of type INTERFACE or with the data type of a technology object.
5042	The specified data type cannot be used within a declaration of retentive variables (VAR RETAIN, VAR_GLOBAL RETAIN). The data type contains at least one system function block in the instance data or is one itself. Since its information cannot be restored, this data type cannot be used in retentive variable declarations.
5050	The class definition is identified as ABSTRACT. As a result, it is not possible to create any instances of the relevant class.
5051	The specified method is identified as ABSTRACT and has not yet been implemented in a method override mechanism. As a result, it is not possible to create any instances of the relevant class.
5052	The class in which a method is defined cannot be used as a variable or parameter at methods within a class declaration. The same applies to function blocks and interfaces. In this case as well, the data type of the function block or interface cannot be used within the declaration section as a data type for variables or transfer parameters.
5053	A variable with the specified data type cannot be directly declared as an in/out parameter (VAR_IN_OUT). Since dynamic type conversion is not supported for in/out parameter calls, it is not possible, for example, to return an INTERFACE value or a reference to VAR_IN_OUT. Declare the variable as either VAR_INPUT or VAR_OUTPUT.
5054	The specified class does not possess any instance data. As a result, it is not possible to create any instances of the relevant class in the form of an ARRAY.

Error	Description
5055	Initialization is only possible with the stated value at the specified position. For example, INTERFACE variables that are defined within structures, methods or functions, or as VAR_TEMP or VAR_IN_OUT, cannot be initialized with class instances. In this case, only NULL can be specified as a value. References to technology objects that are defined within structures, methods or functions, or as VAR_TEMP or VAR_IN_OUT, may only be initialized with the value TO#NIL.
5056	An initialization value may not contain a variable ARRAY index. An initialization value is not a constant.
5057	The instance variables of classes or function blocks of type INTERFACE may only be initialized with exported global class instances. They can also be initialized with local class instances defined with the class or the base classes. They cannot be initialized with class instances from the IMPLEMENTATION area.
5058	Variables of type INTERFACE can only be initialized with the value NULL or with specified class instances. They cannot be initialized with the value of other variables of type INTERFACE.
5059	Method arguments of type INTERFACE cannot be initialized with class instances that are declared as retentive local variables (VAR RETAIN) within the CLASS or the FUNCTION_BLOCK.
5070	The specified variable must be initialized. Please enter a valid initialization value. To do this, use the valid syntax for instance initialization, i.e.: <var_name> : <var_type> := (<element_name1> := <value1>, <element_name2> := <value2>);
5071	The specified variable must be initialized. The variable must be declared before an initialization value can be specified: • either in a declaration section made accessible for initialization with OVERRIDE • or in a declaration section identified as PUBLIC.
5072	The specified data type has been declared by "forward declaration". It is not possible to specify instance-specific initialization values at the stated position. If it is a variable in the implementation section of the source file, check whether you can place it after the declaration of the relevant CLASS or FUNCTION_BLOCK. It is not generally possible to initialize variables created by "forward declaration" in the interface section of the source file.
5080	It is not permissible to specify the initialization value " * " for the specified data type. Instance-specific initialization can be forced using " * "' only for variables of data type INTERFACE or for the references of technology objects.
5081	The initialization value " * " may only be specified for static variables of the CLASS and FUNCTION_BLOCK. Furthermore, the declaration block that contains the variable(s) must permit initialization by OVERRIDE or PUBLIC.
5090	An I/O reference can be declared only within VAR declaration blocks for classes and function blocks. An attempt was made to declare such a variable in a different declaration section. This is not permitted.
5091	The specified data type is not permitted. Only the following data types can be used for I/O references: • Integer data types, e.g. SINT, UINT • Bit data types, e.g. BOOL, WORD • Arrays of these elementary data types, other than arrays of the BOOL data type Other data types cannot be used, because they cannot be specified in the address list.
5092	The I/O reference cannot be linked with the specified I/O variable. The I/O reference requires that the affected variable must be read/write. • I/O references declared with %I* can be linked with all variables of an appropriate data type from the address list. • I/O references declared with %Q* require a variable from the address list for an output. This variable may not be identified with the attribute "read only".
5100	The specified variables cannot be preassigned an initialization value.

Error	Description
5110	The following specifications containing the special character $... are permitted: $$, $', $L, $N, $P, $R, $T. Moreover, the numeric value of a character can be stated using $xx, where xx stands for the two-digit hexadecimal specification of the character code.
5111	The special character can only be specified via $... . This applies to: $L, $N, $P, $R, $T
5112	Multi-line character string constants are not permitted. To produce a new line in the output, use the appropriate special character in the character string, e.g. $N, RL.
5120	No general reference can be generated to the specified data type. General references can be generated for the following data types: • Elementary data types • User-defined data types • Classes • Function blocks with at least one static variable. References to the following data types are not permitted: • General references • Object-oriented interfaces • Technology object data types • Function blocks without static variables.
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.
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 ... 6302)

Table A-13 Errors in the expression (6001 ... 6302)

Error	Description
6001	Syntax error: A statement terminated with a semicolon is expected, e.g. a := b*c;
6002	Syntax error: An expression is expected, e.g. x < y .
6003	The specified identifier is no variable identifier. You must specify a variable identifier. Check whether the indicated identifier is covered. Up to and including V4.0, access to global device identifiers was possible within a program or function block of the same name despite warning 16021.
6004	The index for array access must be the DINT data type. Perform a suitable type conversion or use another expression.
6005	Type conflict in 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: • Variable cannot be read. • Attempt to access a local variable of a function or function block from outside.
6007	Cannot write specified variable. A value assignment is not possible.
6008	The specified function does not supply a return value. An application in the expression is therefore not possible (function declared with a return value of VOID).
6009	The specified identifier does not refer to a function or a function block instance. Therefore it cannot be used as function identifier. When calling a program, the compiler option "Allow language extensions" should have been set (-C lang_ext).
6010	The specified identifier is not included as an input parameter (VAR_INPUT) or in/out parameter (VAR_IN_OUT) in the declaration of the POU (function or function block). It cannot be used in the POU call.
6011	The number of function arguments in the call differs from the declaration, or the call parameters required are missing in the call.
6012	RETURN is not permitted syntactically at this position. RETURN may only be used in functions.
6013	EXIT is not permitted syntactically at this position. EXIT can only be used within FOR, WHILE, and REPEAT.
6014	The specified index value is outside the array limits. Only index values that match the array declaration are permissible.
6015	The specified task control command cannot be applied to the task. It is not permitted 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. Further valid address declarations are %IBx or %QBx for byte access operations, %IWx or %QWx for word access operations and %IDx or %QDx for double word access operations.
6021	The specified byte offset of the directly displayed variables lies outside the permissible address space.

Error	Description
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 or method has not been 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.
6036	An operation is applied to a constant. The value of the constant is outside the definition range for the function. Examples are: • Application of SQRT to a negative number. • Use of logarithmic functions on a number <= 0. • Use of ASIN or ACOS on a number outside the interval [0..1]
6037	An attempt was made to divide a constant by zero. This operation is not permitted.
6038	The specified function parameter occurs more than once in the argument list.
6039	The specified POU (function or function block) cannot be used. Possible causes: • The definition of the POU in the implementation section is missing. Only the prototype was specified in the interface section. • The POU is fully defined only after its use (e.g. call, instance declaration). If necessary, move this POU in the program source before the POU in which it is used. • An instance of the function block cannot be declared as unit variable in the same program source in which this function block is defined.
6040	Only simple variables may be used as semaphores; indexing is not possible.

Error	Description
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.
6043	Only simple variables that belong to the static instance data of the class can be used as semaphores.
6050	There is a type conflict in the expression for the specified operation / variables. 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.
6055	The expression contains a type conflict for the specified operation. Dynamic data type conversion is not permitted for the operand.
6060	At the function call, there is a mixture of assignments of function arguments and setting parameters. Use one form of the function call. Example: • f (x, y); or • f (in1 := x, in2 := y);
6061	The specified parameter of the function or the function block is an in/out parameter. Consequently, a variable must be assigned when the POU is called.
6062	The specified identifier cannot be used as a function argument. Only variables from the declaration blocks VAR_INPUT and VAR_IN_OUT are permitted.
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.
6064	The specified POU is a prototype, for which there is no implementation. Therefore, calling is not possible.
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.
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 associated device; it must have the syntax PI* or PQ*.
6085	The specified variable has been declared within the source file with a data type declared by forward declaration. It is not possible in this instance to use public subcomponents of this variable in other declaration sections as initialization values for other interface variables or references. The use of subcomponents by means of ARRAY access is also not possible.
6100	The specified construct can only be compiled if the device type to be used is stated. To do this, set the required device variants at the library. It is not possible to compile the construct as a device-neutral library.
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 permitted in libraries.

Error	Description
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.
6141	An attempt has been made to access a value of an enumeration data type (ENUM) that cannot be assigned in the current scope. The specified value must be preceded by the declaration scope of the enumeration data type. This is the name of the CLASS, the FUNCTION_BLOCK or the NAMESPACE within which the relevant enumeration data type has been declared. Furthermore, you should enter the identifier of the enumeration data type before you specify the value. Use the following notation: <scope_name>.<enum_type_name>#<enum_value>.
6150	The specified bit offset lies outside the valid range for the specified data type.
6160	The stated array type without length specification is only permitted: • when VAR_IN_OUT parameters are declared in functions, function blocks or methods, • when VAR_INPUT parameters are declared in functions and methods. These kinds of variables cannot be declared in any other declaration sections.
6161	The direct assignment between arrays without length specification is not possible. You need to iterate over the elements for the assignment.
6170	The specified identifier is no variable identifier. You must specify a variable identifier. It should be noted that local variables of a POU are not usable in an independent EXPRESSION (condition).
6180	The specified function is not usable for variables that contain structures with overlapping memory spaces (OVERLAP).
6200	Only with compiler option "Permit language extensions" (-C lang_ext): The called PROGRAM contains instance data (VAR ... END_VAR declaration block) stored in the user memory of the assigned task. This means a call of the PROGRAM from another POU is not possible. Compile the source file with the "Create program instance data only once" compiler option (-C prog_once) or remove the instance data.
6201	Only with compiler option "Permit language extensions" (-C lang_ext): The call of a PROGRAM is not supported in functions. Such calls can be made only in function blocks or another PROGRAM.
6250	The specified method is not declared public. It has not been declared PUBLIC and cannot therefore be called.
6251	SUPER cannot be used in the current context. SUPER can be used in class methods only if the relevant class has a base class (EXTENDS required).
6252	The specified method is implemented in the base class. It cannot be called directly. To call the method, use either SUPER (call the method of the base class with static binding) or THIS (call with dynamic binding). A call that is not preceded by these names is supported only for methods implemented directly in the current class. The call syntax without a preceding name cannot be used for methods of the base class. An error message is also issued if a method without a preceding name is called that is programmed to be overridden after the current call point. In this case, the source file can be successfully compiled after the compiler option "Permit forward declarations" has been set.
6253	The specified method is not identified as public in the base class. It has not been declared as PUBLIC or PROTECTED and cannot therefore be called.

Error	Description
6254	The specified method is implemented in the base class. To be able to recognize any method override mechanism programmed in the current class, this method must either be specified before it is called for the first time or the compiler option "Permit forward declarations" must be activated. The method can be successfully compiled with this option even if the method is not to be overridden.
6260	A reference cannot be generated on the specified variable. References can only be generated on elements that are in the global memory of the controller (VAR_GLOBAL). It cannot be ensured that the specified variable is in the global memory of the controller. It is therefore not possible to generate references to the following variables: • Input parameters (VAR_INPUT) and output parameters (VAR_OUTPUT) for functions or methods • In/out parameters (VAR_IN_OUT) for functions, function blocks or methods • Temporary variables (VAR_TEMP as well as VAR in functions and methods) • Retentive local variables (VAR RETAIN) • Variables of technological objects • Variables of the address list
6261	A reference cannot be generated on the specified variable. Variables to which references are generated must be read/write. It is therefore not possible to generate references to constants.
6262	A reference cannot be generated on the specified variable. Type conversion and dereferencing is not possible within REF().
6300	Only when pragma "ToHookApplicable = YES" is specified: The call to the specified POU is not possible since this POU is not marked as ToHookApplicable. Only the other functions suitable for this can be used in hook functions.
6301	Only when pragma "ToHookApplicable = YES" is specified: The access to the specified variable is not possible while compiling for use in TO hooks. In TO hooks, the following variables cannot be accessed: • Variables of technology objects • I/O variables • Global device variables
6302	Only when pragma "ToHookApplicable = YES" is specified: It is not possible to call the specified method because this method requires a dynamic call (i.e. via the VTable). Only static method calls may be used in hook functions.

A.2.7 Syntax errors, errors in the expression (7000 ... 7014)

Table A-14 Syntax errors, errors in the expression (7000 ... 7014)

Error	Description
7000	A syntax error has occurred. Possible causes: • Incorrectly ended control structures (e.g. END_IF missing) • Statements not terminated with ; • Missing 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: • Directly per value • Symbolically via a preceding constant declaration • 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: • Incorrectly terminated control structures (e.g. END_IF missing) • Statements not terminated with ; • Missing brackets
7011	A syntax error has occurred. Possible causes: • Incorrectly terminated control structures (e.g. END_IF missing) • Statements not terminated with ; • Missing brackets
7012	A syntax error in the statement, that starts at the specified line, has occurred. Possible causes: • Incorrectly terminated control structures (e.g. END_IF missing) • Statements not terminated with ; • Missing brackets
7013	A syntax error has occurred. An illegal construct is being used.
7014	A syntax error has occurred. Possible causes: • Incorrectly terminated control structures (e.g. END_IF missing) • Statements not terminated with ; • Missing brackets
7068	A source is being used by another source with incorrect RT interface stamp. Compile the used sources individually. In case of doubt, select "Project > Save and compile all".

A.2.8 Error when linking a source file (8001 ... 8010)

Table A-15 Error when linking a source file (8001 ... 8010)

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.
8002	The data structure of the specified POU cannot be determined. The cause of the problem is that the POU contains other POUs that are resulting in recursion in the data structure, or that POUs are used that have not been implemented. The message is issued in conjunction with the compiler option 'Permit forward declarations'. Note further error messages indicating the exact error location at which recursion or lack of implementation has been detected.
8010	The interface method is not implemented by the specified class. Since the interface is referenced at the class by IMPLEMENTS, it is absolutely essential that the method is implemented. This also applies if a base class already contains a method implementation of the same name. When an interface is linked, this implementation must be explicitly overridden in the class in which the interface link is programmed.

A.2.9 Errors while loading the interface of another UNIT or a technology package (10000 ... 10101)

Table A-16 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: • Up to version V4.0 of the SIMOTION Kernel: 8 characters. • As of version V4.1 of the SIMOTION Kernel: 128 characters.
10002	Error while loading the interface of another UNIT, a library or technology package. The specified identifier is contained in two different imported units, libraries or technology packages. • Remove a unit, library or technology package from the import list or • Establish uniqueness between the identifiers in imported units, libraries or technology packages. Change the exporting units in the interface section or specify a namespace for a library or a technology package (USELIB ... AS namespace; USEPACKAGE ... AS namespace;).
10003	The specified data type has an invalid memory layout. Probably, the unit was created using an older version of the compiler or compiled using incompatible options. If a unit is involved, it should compiled first. Then repeat the current compilation. You can also perform "Save and recompile everything". If a package is involved, a newer version should be installed. If the error persists, inform the support department.
10004	The exported symbols of the stated unit could not be loaded. Compile the specified unit and try again. Alternatively, perform "Save and compile changes".
10005	A recursion was detected on loading packages. The specified package has already been loaded with USEPACKAGE and cannot be specified a second time.

Error	Description
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. This number includes units that are imported directly with USES as well as units containing used variables or POUs that are indirectly imported by these units.
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.
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 allocated to a device, use the statement UNIT <unitname> : <typname>;
10017	The device type has not been specified uniquely. The statement UNIT <unitname> : <typname>; has been used to specify a device type in the unit that is different from the device type detected from the assignment of the unit to the device or library.
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. If this message is issued after the CPU version has been upgraded, select "Save and recompile all".
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. If this technology package is an OEM package, the error message can be eliminated by installing this package. Otherwise remove the identifier from the statement.
10026	The technology package is used in the specified source file; however, it is not selected for the current device type on the library. Correct the USEPACKAGE statement in the source file or select the technology package for the device type.

Error	Description
10027	The specified source file is imported directly or indirectly into several units in different compilation versions. Recompile all units that use the specified unit. Do this using "Save and recompile all".
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 compiled for the required device type. Possible causes: • The device type is not selected at the library. • The specified library has not yet been compiled. The message might also be displayed if an error occurred in the last compilation. To remedy the situation, check the device settings at the library and compile the library.
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).
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.
10039	Compiling a library requires access to the sources. Access to the specified source is not possible due to the selected protection level of the know-how protection. You have to log in for compiling.
10050	All POUs exported by the specified libraries are identical. Simultaneous use of these libraries is not possible. Modify at least one exported element in one of the relevant libraries. Note: Typically, this error occurs in conjunction with know-how protected libraries when they are imported and used jointly in the project used to create the library.
10051	The specified POUs are identical. Simultaneous use of these classes and therefore these libraries is not possible. Note: Typically, this error occurs in conjunction with know-how protected libraries when they are imported and used jointly in the project used to create the library or when copies of such libraries are used jointly.
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-17 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.
15008	The specified source uses the library or the technology package with different namespace specification between the interface and implementation section. For successful compilation, the library or the technology package must have already been homed to the interface section with USELIB or USEPACKAGE.
15010	The current device setting does not support the specified function. Select a different device version if you want to use this function.
15060	The currently selected device does not support the specified function. Select a different device version if you want to use this function. To do this, exchange the CPU in the hardware catalog and, if necessary, update the firmware. For example, use of the keyword CLASS or INTERFACE requires a CPU version as of V4.5.
15061	The current device setting does not support the specified function. Select a different device version for the library if you want to use this functionality. For example, use of the keyword CLASS or INTERFACE requires a CPU version as of V4.5.
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.
15200	The specification of a bit offset for a bitstring variable requires activation of 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
- In the local compiler settings
- In an ST source file by specifying the following attribute within a pragma:
  { _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 within a pragma:
  { _U7_PoeBld_CompilerOption := warning:*n*:err }, where *n* is the number of the warning or information.

Table A-18 Warnings (16001 .. 16700)

Error	Description
16001	(Warning class: 0) Only in conjunction with the "Selective Linking" compiler option. The specified function, the function block, or the program are neither exported nor called in the current unit. No code is generated.
16002	(Warning class: 0) Only in conjunction with the "Selective Linking" compiler option. The specified unit does not contain any exported PROGRAM nor any task link. No code is generated for the unit.
16003	(Warning class: 2) The operands of the comparison operation do not contain any explicit type definition. The data type listed in the comparison can be seen in the warning message issued. Specify the data type of the used constants explicitly with <type>#<value>.
16004	(Warning class: 2) The specified type conversion may cause the variable value to change due to the reduced display width or inadequate accuracy of the target data type.
16005	(Warning class: 2) During type conversion, the dependency of the variable value can cause the sign to change.
16006	(Warning class: 2) The specified value will be rounded to the next displayable value due to insufficient display width. This is generally the value 0.0.
16007	(Warning class: 2) A loss of accuracy occurred during type conversion. Not all decimal places are considered.
16008	(Warning class: 2) A loss of accuracy occurred during initialization of the specified variables. The constant will be converted to the specified data type. Not all decimal places are considered.
16009	(Warning class: 0) Not in conjunction with compiler option "Selective linking". The specified unit does not contain any exported PROGRAMs nor a task link. Unable to access unit code. Unable to call relevant POU.
16010	(Warning class: 0) Specified program not exported to unit; therefore unable to use it in configuration of the execution level.

Error	Description
16011	(Warning class: 0) The source file does not contain any exported global variables. No data is loaded to the target system.
16012	(Warning class: 0) The specified source file name was taken over from the PROGRAMS container of the selected device. The designation of the source file in the UNIT statement was ignored.
16013	(Warning class: 2) Because of the marshalling function, the specified data type is not portably convertible. Only use SIMOTION devices in conjunction with this data type, or perform an explicit conversion of the data type.
16014	(Warning class: 2) With the specified operation, a data type conversion is performed between signed and unsigned. Because the bit string is adopted in this case, the resulting numerical value can differ from the specified value.
16015	(Warning class: 2) For the assignment of the character string constants to the variables, only part of the character string constants is transferred, because the length of the variable is insufficient to accept all characters.
16016	(Warning class: 2) The operands in the expression do not contain any explicit type definition. The data type of the operation is determined by specifying the values. The resulting data type in which the expression is calculated can be seen in the issued warning message. To define the data type: • Specify the data type of the used constants explicitly with <type>#<value> or • Use an explicit data type conversion.
16017	(Warning class: 2) The operands in the expression contain only constants. The data type of the operation can be determined by specifying the data type (in the form <type>#<value>) or explicit data type conversion. This output is used for troubleshooting, in particular, for the use of symbolic constants, because the data type of the operation cannot normally be determined easily.
16018	(Warning class: 2) The data type of the comparison operation is defined using the value of a constant that has a larger value range than the contained variable. The comparison is performed with the data type of the constant.
16019	(Warning class: 1) The declaration conceals the identifier declared in the scope of the class or the function block. This identifier can no longer be accessed from the POU in which the identifier is declared.
16020	(Warning class: 1) The declaration hides the specified identifier which has been globally defined in its own source file or an imported source file. Access to the global identifier is no longer possible from the POU where this identifier is declared locally.
16021	(Warning class: 1) The declaration hides the specified identifier which is defined on the device. You can access the global device identifier with _device.<name>.
16022	(Warning class: 1) The declaration conceals the specified identifier which is defined in the project (e.g. technology object or device). You can access the global project identifier with _project.<name>.
16023	(Warning class: 1) The declaration hides the specified identifier for the data type of a technology object. Access to the data type identifier is no longer possible.

Error	Description
16024	(Warning class: 1) The declaration hides the access to the technology object on the device. You can access this TO with _to.<name>.
16025	(Warning class: 1) The declaration hides the IEC standard function with the identical name. Access to this function is no longer possible in the current context.
16026	(Warning class: 1) The specified identifier is reserved by SIEMENS for potential extensions. The use of this identifier can cause compiler errors in later versions. If you want to avoid this, change this identifier.
16027	(Warning class: 1) 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) A label has been specified several times in a CASE statement. Only the first label is ever evaluated. Other specifications have no effect.

16040	(Warning class: 1)
	Within a CASE statement, the stated identifier is interpreted as a jump label and not as a CASE selector. This may occur if a LABEL is defined in a control structure within a CASE statement:
	CASE sel OF
	first_val: IF TRUE THEN
	second_val: // -> warning 16040
	result := 2; END_IF;
	third_val: result := 3;
	END_CASE;
	This warning will not be issued if the jump label has been explicitly declared between LABEL and END_LABEL.

16102	(Warning class: 3) The option for output of code for the program status diagnosis function is ignored because no debug information was generated. Output of debug information was deactivated via compiler options.
16103	(Warning class: 3) The option for outputting code at the library for the program status diagnosis function is ignored. The code for program status is generated as defined in the option in the individual source files.
16110	(Warning class: 3) The specified pragma statement is not supported in the current version.
16111	(Warning class: 3) The specified pragma statement is not supported in the used context. The position of the pragma in the source text is not correct.
16112	(Warning class: 3) A valid pragma has been used. However, the value specified for the pragma is not valid. Use a valid value.
16113	(Warning class: 3) The pragma contains a syntax error in an attribute declaration. The valid syntax is as follows: <Attribute identifier> := <Attribute value>; .

Error	Description
16150	(Warning class: 7) A new definition has been made for the specified identifier. Consequently, the previous definition is invalid. This warning enables the work of the preprocessor to be tracked.
16151	(Warning class: 7) An attempt has been made to delete the definition of the specified identifier with #undef. However, the identifier is not defined or the definition is already deleted. This warning enables the work of the preprocessor to be tracked.
16152	(Warning class: 7) The specified definition is not considered during code generation. This may be caused by the preprocessor being deactivated for the compiled source.
16153	(Warning class: 7) 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: 10) 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: 10) The definitions from sources imported using USES are not considered during code generation.
16171	(Warning class: 10) The definition from the specified source imported using USES could not be loaded. Compile the specified source file beforehand.
16200	(Warning class: 4) The use of a semaphore requires a global variable to enable access to it from a different task. Local task operations do not have to be blocked via semaphores.
16210	(Warning class: 4) The basis of the exponential function (EXPT standard function or ** operator) is negative. The operation can be executed at run time only under the following conditions: 1. It can be used on a device with a version of the SIMOTION Kernel as of V4.1. 2. The exponent is an integer. The ExecutionFaultTask will be initiated for non-integer exponents or for use on a device with a version of the SIMOTION Kernel up to V4.0. The program will be aborted at this position.
16220	(Warning class: 4) The condition of an IF statement, WHILE statement or REPEAT statement is a constant expression.
16230	(Warning class: 4) The expression with the specified values does not cause any change to the result; optimized code will be created.
16240	(Warning class: 4) The expression with the specified values exceeds the definition range of the operation. The result may be incorrect.
16250	(Warning class: 4) A modification of the control variable of a FOR loop occurs inside this loop. This modification is either not effective or may cause an unexpected result when editing this loop. If the modification of the variables inside the loop is necessary, then use WHILE or REPEAT at this point instead of FOR.

Error	Description
16260	(Warning class: 4) With the values assigned to the specified function, it has no effect. Optimized code is generated.
16261	(Warning class: 4) The return value of the function or method has not been assigned. The result of the function is thus lost.
16270	(Warning class: 4) The operation for the dynamic type conversion is executed as assignment operation. The associated variables enable an evaluation at the time of the compilation. As a result of this message, it may be determined that a type conversion is not possible. An error message is output as for an assignment of the variables.
16280	(Warning class: 4) The INTERNAL access identifier was specified for an element not contained in any NAMESPACE. The identifier is ignored and the element is PUBLIC.
16281	(Warning class: 4) The specified identifier has already been declared as INTERNAL in the current source or in an imported source or library. The current declaration location, however, does not contain the INTERNAL identifier. Add INTERNAL for the declaration of the affected element.
16282	(Warning class: 4) The specified NAMESPACE was identified as public in one source and declared as INTERNAL in another source. This conflict is resolved by declaring the NAMESPACE as INTERNAL. Consequently, the contained elements can be accessed only in accordance with the rules that apply to INTERNAL.
16283	(Warning class: 4) Rather than the PUBLIC access identifier specified for the data type declaration, INTERNAL is used, because the complete POU was declared as INTERNAL. This identifies contained elements as INTERNAL, because they may not be accessed outside the NAMESPACE.
16300	(Warning class: 5) The auxiliary value has a data type that cannot be converted to the data type configured for the message.
16301	(Warning class: 5) The specified auxiliary value is not evaluated during output of the message.
16302	(Warning class: 5) The data type of the auxiliary value cannot be determined from the message configuration. The specified data type is used.
16303	(Warning class: 5) No auxiliary value has been specified for the function although the message configuration requires such a value. A default value of the corresponding data type was added.
16304	(Warning class: 5) An alarm accompanying value is specified using a constant or a constant expression. The resulting data type of the alarm accompanying value can be seen in the issued warning message. To define the data type: • Specify the data type of the used constants explicitly with <type>#<value> or • Use an explicit data type conversion.
16400	(Warning class: 6) A global variable has been declared in a library. This may mean that the library cannot be used more than once.

Error	Description
16401	(Warning class: 6) A PROGRAM has been declared in a library. This cannot be used in the process system. Set the compiler option "Only create program instance data once" (-C prog_once) or declare a FUNCTION_BLOCK.
16420	(Warning class: 6) The return value has not been assigned within the function or a method. If such a function or method is called, it returns a random value.
16421	(Warning class: 6) A variable that has neither been assigned nor read in the code has been declared. The output of this warning can only be suppressed for a POU by means of a pragma.
16430	(Warning class: 6) An attempt is being made to use a function block, which is either a system function block itself or includes one, as a temporary variable. This can result in runtime problems when using the block in a cyclic level.
16440	(Warning class: 10) A global variable that does not contain any user data that can be backed up has been created in the retentive memory area. These kinds of variables should not be declared in RETAIN areas. It is not meaningful to specify RETAIN in this case and this specification is supported only for compatibility reasons TO references are an example of a data type for which this message is output.
16441	(Warning class: 6) The specified data type contains some variables that cannot be restored in addition to those that can be backed up in RETAIN areas. Data types of this kind include, for example, TO references and INTERFACE variables.
16450	(Warning class: 10) A global variable has been created in the retentive memory range. This declaration is not permissible at the specified position.
16451	(Warning class: 10) 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: 10) 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: 10) 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) The specified variable is not contained in the initialization list. The default initialization value is used.
16601	(Warning class: 6) The specified variable is not contained in the initialization list. The default initialization value is used.
16602	(Warning class: 6) The specified variable is not contained in the initialization list. The default initialization value is used.
16603	(Warning class: 6) 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.

Error	Description
16604	(Warning class: 6) The data type used in the declaration has no unique initialization value. The initial value known last is used. In order to achieve a unique behavior at this point, the initial value for the variable or data type declaration must be specified explicitly.
16605	(Warning class: 6) The memory space of the specified variable overlaps with another variable. Thus, the initialization value cannot be included. The remaining memory space is assigned the default value 0.
16606	(Warning class: 6) The memory space of the specified variable overlaps with another variable. Thus, the data type cannot be transferred to the OPC XMP information. The component is symbolically unavailable and also not included in backups using _exportUnitDataSet.
16607	(Warning class: 6) The specified function block does not contain any instance data and, as a result, has a size of 0 bytes. The instance declaration in the form of an ARRAY is not appropriate.
16700	(Warning class: 3) 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.

See also

Global compiler settings (Page 63)

Local compiler settings (Page 67)

Controlling the preprocessor and compiler with pragmas (Page 368)

Controlling compiler with attributes (Page 374)

A.2.12 Non-assigned compiler messages (22000 ... 22002)

You create non-assigned compiler messages within a pragma (Page 368). Enter the desired message text as a character string (Page 109) enclosed in brackets after the keyword "information", "warning" or "error". End the statement within the pragma with a semi-colon. See "Issuing non-assigned compiler message" (Page 378).

Table A-19 Non-assigned compiler messages (22000 ... 22002)

Error	*Description*
22000	This message is a user-defined error message. See error text.
22001	(Warning class: 0) This message is a user-defined warning. See warning text.
22002	(Warning class: 0) This message is a user-defined information message. See information text.

A.2.13 Information (32010 ... 32653)

You control this output of information together with the warnings (Page 512):

Table A-20 Information (32010 ... 32653)

Error	Description
32010	(Warning class: 6) The specified jump label identifier has been declared but not used.
32020	(Warning class: 10) The specified variable was declared globally in this source file or in another source file with the indicated data type. This may be a variable, a constant, or another identifier. This information helps when searching for the cause of compilation errors. It is issued together with error messages.
32021	(Warning class: 10) The specified variable was declared on the device as an I/O variable, a global device variable, or a system variable. This information helps when searching for the cause of compilation errors. It is issued together with error messages.
32022	(Warning class: 10) The specified variable was declared in the project as a global identifier. This may be a device identifier, a technology object or another identifier from the project. This information helps when searching for the cause of compilation errors. It is issued together with error messages.
32023	(Warning class: 10) Until now, no valid declaration has been found for the specified identifier. This information is issued together with error messages.
32024	(Warning class: 0) The specified variable has been declared as a global identifier in the current unit or in an importing unit. This information helps when searching for the cause of compilation errors. It is issued together with error messages.
32025	(Warning class: 10) The specified identifier was declared in this source or in another source in the specified namespace. In order to be able to use this identifier, it must be prefixed with the identifier of the namespace. This may be a variable, a constant or another identifier. This information is helpful when searching for the cause of compilation errors. It is issued together with error messages.
32026	(Warning class: 0) The specified identifier is a standard function that can be activated via a compiler option. In order to use the function, activate the required compiler options for language extensions.
32030	(Warning class: 0) The stated array initialization does not conform to IEC 61131-3. For portable programs, you should enclose the array initialization values in square brackets. Example of field initialization in compliance with the standard: x : ARRAY [0 to 1] OF INT := [1, 2];

Error	Description
32050	(Warning class: 0) The maximum size that can be reached via an HMI is 65536 bytes. This limit has been exceeded with the specified variable. All subsequent variables cannot be reached either.
32060	(Warning class: 0) The specified identifier is a keyword that can be activated via a compiler option. In order to use the keyword, activate the compiler option "Language extensions IEC61131-3rd Edition". This message is output as a guide for debugging syntax errors.
32061	(Warning class: 0) The specified identifier is a keyword that can be activated via a compiler option. In order to use the keyword, activate the compiler option "Permit object-oriented programming". This message is output as a guide for debugging syntax errors.
32300	(Warning class: 1) A label has been specified several times in a CASE statement. Only the first label is ever evaluated. Other specifications have no effect.
32650	(Warning class: 7) The specified identifier will be replaced thereafter by the output text. This information enables the work of the preprocessor to be tracked.
32651	(Warning class: 7) The definition of the specified identifier has been deleted with #undef. This information enables the work of the preprocessor to be tracked.
32652	(Warning class: 7) The identifier will be used with the specified replacement text in the source file. Compilation takes place with the replacement text. This information enables the work of the preprocessor to be tracked.
32653	(Warning class: 7) The specified identifier will be replaced thereafter by the output text. This information appears if additional replacements are loaded with a USES statement. This information enables the work of the preprocessor to be tracked.

A.3 Template for Example Unit

A.3.1 Preliminary information

This chapter presents a comprehensively 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_l_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 array
    // "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
    // RETAIN stored in the 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
// EXPRESSION TRUE.
```

A.3.5 Function

```
// ******************************************************
// * FUNCTION                                           *
// ******************************************************

// The declaration of an FB or FC must be placed in the source file
// before the actual use (the call), so that the code of the
// block is already known to the calling point.

FUNCTION 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_GLOBAL, VAR_GLOBAL CONSTANT,
    // VAR_GLOBAL RETAIN and VAR_RETAIN 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

-

-, 156
-1.#IND, 397, 399
-1.#INF, 397, 399
-1.#QNAN, 397, 399

"

"
    Dereferencing, 252

#

#define, 371
#else, 371
#endif, 371
#ifdef, 371
#ifndef, 371
#undef, 371

*****

*, 156
**, 156

/

/, 156

:

:, 118, 136
:=, 146, 195, 196, 222, 223, 251

?

?=, 252

_

_AdditionObjectType, 133
_alarm, 353
_CamTrackType, 133
_ControllerObjectType, 133
_device, 343, 353
_direct, 319, 324, 343, 353
_FixedGearType, 133
_FormulaObjectType, 133
_getSafeValue
    Application, 343
_PathAxis, 133
_PathObjectType, 133
_project, 353
_quality, 330, 353
_SensorType, 133
_setSafeValue
    Application, 343
_task, 353
_to, 353
_U7_PoeBld_CompilerOption, 374

+

+, 156

<

<, 158
<=, 158
<>, 158

=

=, 158
=>, 197, 224

>

>, 158
>=, 158

1

1.#INF, 397, 399
1.#QNAN, 397, 399

A

Absolute identifier
Overview, 431
Access times
Parameter, 197
ANY, 116
ANY_BIT, 116
ANY_DATE, 116
ANY_ELEMENTARY, 116
ANY_INT, 116
ANY_NUM, 116
ANY_REAL, 116
ANYOBJECT, 133
Arithmetic operators, 155
ARRAY
Data type, 120
With a defined length, 120
With a dynamic length, 193
Arrays
Data type, 120
Value assignments, 150, 151
With a defined length, 120
With a dynamic length, 193
AT, 127
Attribute
Compiler option, 374
Availability
I/O variable, 330

B

Base class
definition, 214
Structure, 214
Syntax, 214
Basic elements
of ST, 96
Basic functions, 155
Bit constants, 109
Bit data types, 114
BlockInit_OnChange, 376
BlockInit_OnDeviceRun, 376
Blocks, 96
Bookmarks, 47
BOOL, 114
Boolean data, 109
Branches
Syntax, 487
Breakpoint, 407
Activating, 420
Call path, 415, 418
Call stack, 423
Deactivating, 423
remove, 412
Setting, 412
Toolbar, 414
BYTE, 114

C

Call path
Breakpoint, 415, 418
Call stack, 423
Program run, 400
Program status, 406
CamType, 133
CASE statement
Description, 164
Character set, 97, 429
Class
Instances, 226
Names, 226
Source file section, 268
Code attributes, 366
Commands
Overview of the basic system, 432
ST programming language overview, 105
Comments, 112
Source file section, 112
Syntax, 450
Compiler, 88
Attribute, 374
Correcting errors, 63, 89
Declaration errors in data type declarations, 497
Declaration errors in POU, 494
Declaration errors in variable declarations, 498
Error when linking a source file, 508
Errors in expression, 502
Errors while loading the interface of another UNIT or technology package, 508
File access errors, 493
Implementation restrictions, 511
Information, 519
Scanner errors, 494
setting, 63
Start, 88
starting, 62
Syntax errors, errors in expression, 506
Warnings, 512
Compiler option, 63, 74

Compiling
Library, 345
Compound data types, 120, 124
CONSTANT, 138, 144
Constant block
Syntax, 462
Constants
Bit, 109
Data types for constants, 113
Date and time, syntax, 447
Digit strings, syntax, 446
Floating-point number, 108
Formatting characters and separators, 430
Globally valid, 286
Integer, 107
Literals, syntax, 442
Symbolic names, 143
Time specifications, 114
Unit constants, 286
CONTINUE statement
Description, 171
Control statements, 162
CPU memory access
Identifiers for process image access, 431
Variable model, 283
Cross-reference list, 361
Displayed data, 362
Filtering, 364
Generating, 361
Single-step monitoring (MCC), 362
Sorting, 364
Trace (MCC), 362
TSI#currentTaskId, 362
TSI#dwuser_1, 362
TSI#dwuser_2, 362
Cyclic program execution
Effect on I/O access, 319, 324, 332
Effect on variable initialization, 300

D

Data model, 283
Data type specification
ARRAY, 120
elementary, 119
Enumeration, 123
STRUCT, 124
Data types
ARRAY, 120
Bit data type, 114
Conversions, 176
Derivation of simple types, 119
elementary, 114
Elements, syntax, 472
Enumeration, 123
Enumerators, 123
Explicit conversions, 178
Implicit conversions, 176
Inheritance, 134
Initialization, 139
Numeric, 114
STRING, 115
STRUCT, 124
Structure, 124
Syntax, 472
Technology object, 132
Time, 114
TYPE, 118
User-defined, 118
User-defined, syntax, 475
DATE, 115
DATE_AND_TIME, 115
Debug mode, 389, 408
Declaration
Parameter, 137
Variable ("on-the-fly"), 312
Variables, 137
Declaration section
Syntax, 459
Declarations
Syntax, 468
Derivation of simple data types, 119
Derived data type
Array, 120
ARRAY, 120
Enumeration, 123
Enumerator, 123
STRUCT, 124
Structure, 124
Device
Rules for identifiers, 379
Settings, 381
DINT, 114
DINT#MAX, 116
DINT#MIN, 116
Direct access, 319, 323
Properties, 320, 321, 322, 323
Rules for I/O variables, 326
Update, 321
Variable model, 283
Download
Effect on variable initialization, 300
DriveAxis, 133
DT, 115

DWORD, 114

E

Editor
  Example for program, 87
  External, 79
  Internal, 29
  Operation, 87
  ST editor, 29
  Toolbar, 54
Elementary data types
  Overview, 114
Enumeration
  Defining, 123
  Example, 124
Enumeration data types, 123
Enumerators, 123
Error
  FB or FC call, 203
Error messages
  Declaration errors in data type declarations, 497
  Declaration errors in POU, 494
  Declaration errors in variable declarations, 498
  Error when linking a source file, 508
  Errors in expression, 502
  Errors while loading the interface of another UNIT or technology package, 508
  File access errors, 493
  Implementation restrictions, 511
  Information, 519
  Scanner errors, 494
  Syntax errors, errors in expression, 506
  Warnings, 512
Example, complete
  FBs and FCs, 203
  Rotate bit in output byte, 83
  ST source file (template), 521
  User-defined data types, 126
  Using data types of TOs, 134
EXIT statement
  Description, 171
Explicit data type conversions, 178
Exponent
  Description, 108
Exponentiation, 156
Export
  ST source file, 77
EXPRESSION
  Description, 267
  Syntax, 209
Expressions
  Arithmetic, 155
  Logic, 162
  Relational expressions, 158, 162
  Rules for formulation, 153, 162
ExternalEncoderType, 133

F

FB, 181
FB/FC variables
  Definition, 288
  Variable model, 283
FC, 181
File
  See Source file, 110
Floating-point number
  Data types, 114
  Description, 108
  Notation, 108
FollowingAxis, 133
FollowingObjectType, 133
FOR statement
  Description, 167
Formatting characters, 429
Forward declaration, 382
Function, 181
  Call, 198
  Call path, 406
  definition, 181
  Error sources during a call, 203
  Example, 203
  In/out parameter, 191
  Input parameters, 191
  Local variables, 192
  Output parameters, 192
  Source file section, 263, 270
  Structure, 181
  Syntax, 181
Function block, 181
  Call path, 406
  Call, syntax, 200
  Calling, 199
  definition, 182, 183
  Difference to the FC, 203
  Error sources during a call, 203
  Example, 203
  In/out parameter, 191
  Input parameters, 191
  Instances, 199
  Local variables, 192
  Names, 199

object-oriented, syntax, 183
Output parameters, 192
Source file section, 264
Structure, 182, 183
Syntax, 182
Function block with methods
Syntax, 183

G

General reference
define, 249
form, 250
Operations, 251
Global device user variables
Defining, 294
Variable model, 283
GOTO statement
Description, 175
Use, 385

H

Hardware
Setting up, 85
Hiding validity ranges, 348
HMI_Export, 375

I

I/O variable
Availability, 330
Creating, 327, 342
Direct access, 319, 323
Process image, 319, 324
Process image of the BackgroundTask, 334
Rules, 326
Status, 330
Update, 321
Variable model, 283
Identifier
Predefined, 431
Reserved for ST, 105, 432
Rules for formulating, 97
Rules for SIMOTION devices, 379
Syntax, 97
Identifiers
Syntax, 442
IF statement
Description, 162
Implementation
Source file section, 260
Implicit data type conversions, 176
Import
ST source file, 78
In/out assignment
Syntax, 197, 223, 224
In/out parameter
Function, 191
Function block, 191
Method, 220
Transfer, 196, 223
Inheritance
For technology objects, 134
when declaring public/using, 281
Initialization
Data types, 139
Syntax, 470
Time of the variable initialization, 300
Variables, 139
Input assignment
Syntax, 195, 222
Input parameters
Access in the function block, 201
Function, 191
Function block, 191
Method, 220
Transfer, 195, 222
Instance declaration of a class
Syntax, 226, 227
Instance declaration of FB
Syntax, 199
INT, 114
INT#MAX, 116
INT#MIN, 116
Integer
Data types, 114
Description, 107
Notation, 107
Integer number
See Integer, 107
Interface
Source file section, 258
Interface, object-oriented
Source file section, 271
Syntax, 239
INTERNAL, 186, 216, 355

J

Jump label, 385

Jump labels
Syntax, 468
Jump statement, 385

K

Key combination
Script editor, 59
ST editor, 59
Keyboard shortcuts, 59
Know-how protection
Libraries, 346
Source files, 75

L

LABEL declaration, 385
Language description
Resources, 95, 427, 429
Library, 343
Compiling, 345
Using, 347
Line numbering, 53
Local data stack, 295, 299
Local variables
Variable model, 283
Logic expression; bit-serial expression; expressions
logic; expressions: bit-serial; operators:logic, 160
Lower case, 54
LREAL, 114

M

MeasuringInputType, 133
Memory requirement, 295, 299
Method
define, 184
definition, 217
In/out parameter, 220
Input parameters, 220
Local variables, 220
Output parameters, 220
Structure, 184, 217
Syntax, 184, 217
MOD, 156
Monitoring variables
Variable status, 399
Multi-element variables, 150, 151

N

Names, 97
Namespace
for libraries, 352
for technology packages, 352
Predefined, 353
New
I/O variable, 327, 342
Number systems
Notation, 108
Numbers
Data types for numbers, 113
Description, 107
Notation, 107
Numeric data types, 114

O

Opening
ST source file, 27
Operands
Syntax, 480
Operating mode
Debug mode, 389, 408
Process mode, 388
Test mode, 388, 404
Operators, 432
Priority, 162
Relational operators, 158
Syntax, 483
Output parameters
Access in the function block, 201
Function, 192
Function block, 192
Method, 220
Transfer, 197, 224
OutputCamType, 133
OVERLAP, 129

P

Parameter
Access times, 197
Block (syntax), 188, 219
Declaration, 186, 218
Declaration, general, 137
Function and function block, 186, 218
Transfer (in/out parameter), 196
Transfer (input parameter), 195

Transfer (output parameter), 197
Transfer (principle), 195
Parameter blocks
Syntax, 465
Parameters
Transfer (in/out parameter), 223
Transfer (input parameter), 222
Transfer (output parameter), 224
PosAxis, 133
Pragma
Attribute, 374
Preprocessor statement, 370
Preprocessor, (See preprocessor)
Activating, 64, 68
Controlling, 369
Preprocessor statement, 370
Using, 64, 68
Warning class, 73
Preprocessor statement
Example, 373
Printing
ST source file, 79
PRIVATE, 186, 216
Process image
Cyclic tasks, 324
principle and use, 319, 332
Properties, 320, 321, 322, 323
Rules for I/O variables, 326
Symbolic access, 340
Update, 321
Process image of the BackgroundTask, 319
Process image of the cyclic tasks, 319
Process mode, 388
Program
Assigning tasks, 90
Call path, 406
Compiling, 88
Connecting to target system, 91
Creating (example), 87
Download, 92
Executing, 90, 93
Locating errors, 387
Source file section, 266
starting, 90, 93
Status (test tool), 402
Testing, 387
Program organization units
Source file section, 262
Syntax, 454
Program run, 400
Toolbar, 402
Program section
See Source file section, 257
Program structure, 365
Program structuring, 162
program variables
Definition, 288
In the data model, 287
Variable model, 283
Programming environment, 21
Project
Opening, 84
Project comparison
Overview, 426
PROTECTED, 216
Prototype
Program organization unit, 382
Prototypes, 279
PUBLIC, 186, 216

R

REAL, 114
Real number
See Floating-point number, 108
REF, 250
REF_TO, 249
Reference, 132
define, 249
form, 250
general, 249
Operations, 251
Reference data, 361
References, 5
Relational expressions, 158
REPEAT statement
Description, 170
Repetition statements and jump statements
Syntax, 489
Reserved identifiers, 99, 432
RETAIN, 138, 287
Retentive variables
Definition, 287
Variable model, 283
RETURN statement
Description, 172
Rules
Formatted, 427, 441
Semantics, 96
Unformatted, 428, 441
RUN
Effect on variable initialization, 300

S

SCOUT Workbench > See Workbench, 21
sections
  Syntax, 451
Separators, 429
Sequential program execution
  Effect on I/O access, 319, 323
  Effect on variable initialization, 300
setting
  Compiler, 63
Shortcuts, 59
SIMOTION device
  Rules for identifiers, 379
  Settings, 381
Simple data types
  Derivation, 119
Single-element variables, 147
SINT, 114
SINT#MAX, 116
SINT#MIN, 116
Source file
  Structure, 110
Source file section, 257
  Class, 268
  Data type declaration, 274
  Declaration section; declaration section:source file section, 272
  Function, 263, 270
  Function block, 264
  Implementation, 260
  Interface, 258
  Object-oriented interface, 271
  Program, 266
  Program organization unit, 262
  Statement, 112
  Statement section, 273
  Unit statement, 278
  Variable declaration, 275
ST compiler. See Compiler, 62
ST editor, 29
  On-the-fly variables declaration, 312
  Pairs of brackets, 53
ST source file
  exporting, 77
  Importing, 78
  Opening, 27
  Printing, 79
  See Source file, 110
  Template (example), 521
ST source file section
  See Source file section, 257
Standard functions, 155
Statement
  Source file section, 112, 273
Statement section
  Syntax, 477
Status
  I/O variable, 330
  Program (test tool), 402
STOP to RUN
  Effect on variable initialization, 300
STRING, 115
  Assignment, 147
  Edit, 148
  Element, 147
  Syntax diagram, 115
STRUCT, 125
STRUCT OVERLAP, 129
StructAlarmId, 117
STRUCTALARMID#NIL, 117
StructTaskId, 117
STRUCTTASKID#NIL, 117
Structure
  Defining, 124
  Example, 126
Structured variables, 150, 151
Structures
  Syntax, 451
Symbol Browser, 393
Symbolic access to I/O address space
  Process image, 340
Syntax diagram, 95
System functions
  Inheritance, 134
System variables
  Inheritance, 134
  Variable model, 283

T

T#MAX, 116
T#MIN, 116
Target variable, 145
Task
  Assigning programs, 90
  Effect on variable initialization, 300
Technology object
  Data type, 132
  Inheritance, 134
TemperatureControllerType, 133

Template
ST source file, 521
Terminals, 96
Test mode, 388, 404
Testing a program, 387
TIME, 115
Time types
Conversions, 176
Functions, 155
Overview, 114
TIME#MAX, 116
TIME#MIN, 116
TIME_OF_DAY, 115
TIME_OF_DAY#MAX, 116
TIME_OF_DAY#MIN, 116
TO#NIL, 133
TOD, 115
TOD#MAX, 116
TOD#MIN, 116
Tool tip, 54
Trace tool, 426
TSI#currentTaskId
Cross-reference list, 362
TSI#dwuser_1
Cross-reference list, 362
TSI#dwuser_2
Cross-reference list, 362
TYPE, 118
Type conversion functions, 176
Type declaration, 118

U

UDINT, 114
UDINT#MAX, 116
UDINT#MIN, 116
UDT
See User-defined data type, 117
UINT, 114
UINT#MAX, 116
UINT#MIN, 116
UNION, 129
Unit
Source file section, 278
Template (example), 521
UNIT, 278
Unit constants
Definition;, 286
Unit variables, 286
Definition, 285
Non-retentive, 286
Variable model, 283
Upper case, 54
USELIB, 258, 352
USEPACKAGE, 258, 352
User-defined data type
Syntax, 118
USES, 259, 261, 281
USING, 358
USINT, 114
USINT#MAX, 116
USINT#MIN, 116

V

Value assignments
Description, 145
Syntax, 478
VAR, 137, 291, 292
VAR CONSTANT, 138, 144
VAR RETAIN, 138, 293
VAR_GLOBAL, 137, 286
VAR_GLOBAL CONSTANT, 138, 144
VAR_GLOBAL RETAIN, 138, 287
VAR_IN_OUT, 138, 188, 189, 190, 219
VAR_INPUT, 138, 188, 189, 190, 219
VAR_OUTPUT, 138, 188, 189, 190, 219
VAR_TEMP, 138, 292
Variable blocks
Syntax, 463
Variable status
Monitoring variables, 399
Variables, 136
ARRAY, 150, 151
Battery-backed, 287
Declaration, 137
Declaration ("on-the-fly"), 312
Declaration (source file section), 275
elementary, 147
Enumerator data type, 150
Function, 192
Function block, 192
Hiding validity ranges, 348
Identical names, 348
Initialization, 139
Instance declaration of a class, 226
Instance declaration of FB, 199
Local, 288
Method, 220
Parameter declaration, 186, 218
Process image, 319, 332
Retentive, 287
Static, 288
structured, 151

Temporary, 288
timing of initialization, 300
Unit variable, 286
Validity, 283
Watch tables, 397

W

WAITFORCONDITION statement
Description, 173
Example, 175
Warning class, 73, 370
Watch tables, 397
WHILE statement
Description, 169
WORD, 114
Workbench
Elements, 23
Programming environment, 21