

WinSPS

Programming according to IEC 61131-3 Software Manual



Edition

104

WinSPS

Programming according to IEC 61131-3 Software Manual

1070 072 305-104 (03.03) GB



© 2001 – 2003

by Bosch Rexroth AG, Erbach / Germany
All rights reserved, including applications for protective rights.
Reproduction or distribution by any means subject to our prior written permission.

Discretionary charge 18.– €

Contents

	Page
1	Safety Instructions 1–1
1.1	Intended use 1–1
1.2	Qualified personnel 1–2
1.3	Safety markings on components 1–3
1.4	Safety instructions in this manual 1–4
1.5	Safety instructions for the described product 1–5
1.6	Documentation, software release and trademarks 1–6
2	Quick start and input examples 2–1
2.1	Project Default settings 2–1
2.2	Programming variations 2–2
2.3	Edit IEC file 2–3
2.4	Check symbol file 2–6
2.5	Load program in the controller 2–7
2.6	Observe and test the program on the monitor 2–8
3	Introduction 3–1
3.1	What is IEC 61131-3? 3–1
3.2	Programming languages of the IEC 61131-3 3–1
3.2.1	The programming language IL 3–2
3.2.2	The programming language ST 3–3
3.3	Why use programming languages as per IEC 61131-3 ? 3–3
3.4	Difference from “classical” programming languages 3–3
3.5	Model of the programming as per IEC 61131-3 3–5
3.6	Compatibility and fulfillment of standard 3–8
3.7	Programming system and controller 3–8
4	Project preparations 4–1
4.1	Installation 4–1
4.2	Default settings 4–1
4.2.1	Licensing the programming languages 4–1
4.2.2	Project default settings 4–2
5	Writing programs in the WinSPS Editor 5–1
5.1	Declaration tables 5–3
5.1.1	POU type 5–3
5.1.2	Variable declaration 5–4
5.1.3	Type definition 5–8
5.2	Instructions part 5–11
5.3	Error messages 5–12
5.4	Global variable declaration – variable editor 5–13
5.5	Global type definition – type editor 5–14
5.5.1	TYPE: Data Type 5–14
5.5.2	STRUCT: Data structure 5–15
5.5.3	ENUM: Enumeration 5–16
5.6	Constant definition – DEFINE Editor 5–18

6	Program Structure	6-1
6.1	Program Organization Units– modules of the IEC	6-1
6.2	POU types	6-3
6.2.1	Main program – PROGRAM	6-4
6.2.2	Function block – FUNCTION_BLOCK	6-5
6.2.3	Function – FUNCTION	6-7
6.3	Declaration part	6-8
6.3.1	Variable types	6-9
6.3.2	Applicability and access options of the variable types	6-11
6.4	Instructions part	6-12
6.5	Calls between POUs	6-12
6.5.1	Call hierarchy	6-12
6.5.2	Recursive calls	6-13
6.5.3	Call interface – parameters during the call	6-14
6.5.4	Calling up the function blocks	6-17
6.5.5	Calling up the functions	6-19
6.6	Instance building of function blocks	6-21
6.6.1	Validity of function blocks	6-22
6.6.2	Module with “memory”	6-22
6.6.3	Instance building for combination with “classical” programming languages 6-22	
7	Data model	7-1
7.1	Language elements	7-1
7.1.1	Key words	7-2
7.1.2	Identifiers	7-3
7.1.3	Literals	7-4
7.1.4	Delimiter	7-7
7.1.5	Comments	7-8
7.2	Data types	7-9
7.2.1	Elementary data types	7-9
7.2.2	Derived data types (Type definition)	7-11
7.2.3	Generic data types	7-13
7.3	Variables	7-15
7.3.1	Declaration of variables	7-15
7.3.2	Initialization of variables and remanence	7-16
7.3.3	Access to variables	7-18
7.3.4	Physical addresses	7-19
7.3.5	String variables	7-21
7.3.6	ARRAY	7-22
7.3.7	Data structures (STRUCT)	7-28
7.3.8	Variable attributes	7-30

8	Programming language Instruction List (IL)	8–1
8.1	Instructions	8–1
8.2	Working register and status bits	8–2
8.3	Current Result (CR) – the universal accumulator	8–2
8.4	Program rules	8–4
8.4.1	IL sequences	8–4
8.4.2	Label	8–5
8.4.3	Nesting levels, Parenthesis	8–5
8.5	Instruction set	8–7
8.5.1	Load instructions – LD	8–9
8.5.2	Assignments – ST, S, R	8–10
8.5.3	Boolean operators AND, &, OR, XOR	8–12
8.5.4	Arithmetic operators ADD, SUB, MUL, DIV	8–15
8.5.5	Comparison operators– GT, GE, EQ, LE, LT, NE	8–18
8.5.6	Jump operators – JMP, JMPC, JMPCN	8–19
8.5.7	Call of function blocks – CAL, CALC, CALCN	8–20
8.5.8	Call of functions	8–22
8.5.9	Return jump – RET, RETC, RETCN	8–23
9	Programming language Structured Text (ST)	9–1
9.1	Expressions, operands and operators	9–1
9.2	Instructions	9–3
9.2.1	Assignment	9–4
9.2.2	Call of a function block	9–4
9.2.3	Return jump – RETURN	9–5
9.2.4	Conditional execution	9–5
9.2.5	Selection, – IF	9–6
9.2.6	Multi-selection– CASE	9–7
9.2.7	FOR loop	9–8
9.2.8	WHILE loop	9–10
9.2.9	REPEAT loop	9–10
9.2.10	Deflecting and non-deflecting loops	9–11
9.2.11	Premature loop end – EXIT	9–12
10	Check load and test program	10–1
10.1	Check / compile module	10–1
10.2	Link all modules – Create new project	10–2
10.3	Project specifications in the symbol file	10–4
10.4	Load program and modules	10–5
10.5	Monitor	10–6
11	Use of IEC modules in the classical programming languages	11–1
11.1	Pure IEC programs	11–1
11.2	Mixed programs	11–2
11.3	Function block call	11–4
11.3.1	Call parameter list – wizard for FB call	11–5
11.3.2	Changing the FB calls	11–8
11.3.3	Deleting the FB calls	11–9
11.3.4	Call in the Sequential Function Chart	11–10
11.4	Symbol file – interface of mixed programming	11–11
11.4.1	Physical addresses and miscellaneous data	11–11
11.4.2	Symbolic operands via the call interface	11–13
11.4.3	Symbolic operands as global variables	11–14
11.4.4	Global type definitions	11–14
11.5	Differences in case of mixed programming	11–17

12	Standardized functionality	12-1
12.1	Standard functions	12-1
12.1.1	Generic data types and "overloaded" functions	12-1
12.1.2	Extensibility of functions	12-3
12.1.3	Type conversion	12-3
12.1.4	Numeric functions	12-5
12.1.5	Arithmetic functions	12-5
12.1.6	Shift functions	12-7
12.1.7	Boolean functions – logical links	12-8
12.1.8	Selection	12-8
12.1.9	Comparison	12-9
12.1.10	Functions for strings	12-10
12.2	Standard function block	12-12
12.2.1	Bistable elements – Flipflops	12-14
12.2.2	Edge detection	12-15
12.2.3	Counter	12-16
12.2.4	Timer	12-18
13	Standard fulfilment	13-1
13.1	Common elements	13-1
13.2	Language elements	13-14
13.3	Causes of errors	13-16
A	Annex	A-1
A.1	Abbreviations	A-1
A.2	Index	A-2

1 Safety Instructions

Before you start programming the software PLC PCL or the CL550 by using programming languages according to IEC 61131-3, we recommend that you thoroughly familiarize yourself with the contents of this manual. Keep this manual in a place where it is always accessible to all users.

1.1 Intended use

This manual contains information required for the proper use of this product. The products described hereunder have been developed, manufactured, tested and documented in compliance with the safety standards. These products pose no danger to persons or property if they are used in accordance with the handling stipulations and safety notes prescribed for their configuration, mounting, and proper operation.

1.2 Qualified personnel

This instruction manual is designed for specially trained personnel. The relevant requirements are based on the job specifications as outlined by the ZVEI and VDMA professional associations in Germany. Please refer to the following German-Language publication:

Weiterbildung in der Automatisierungstechnik
Publishers: ZVEI and VDMA Maschinenbau Verlag
Postfach 71 08 64
60498 Frankfurt/Germany

This instruction manual is specifically designed for PLC technicians. Basic skills in Programmable Logic Controllers are an advantage, however, they are not mandatory.

Interventions in the hardware and software of our products not described in this instruction manual may only be performed by our skilled personnel.

Unqualified interventions in the hardware or software or non-compliance with the warnings listed in this instruction manual or indicated on the product may result in serious personal injury or damage to property.

Installation and maintenance of the products described hereunder is the exclusive domain of trained electricians as per IEC 826-09-01 (modified) who are familiar with the contents of this manual.

Trained electricians are persons of whom the following is true:

- They are capable, due to their professional training, skills and expertise, and based upon their knowledge of and familiarity with applicable technical standards, of assessing the work to be carried out, and of recognizing possible dangers.
- They possess, subsequent to several years' experience in a comparable field of endeavour, a level of knowledge and skills that may be deemed commensurate with that attainable in the course of a formal professional education.

With regard to the foregoing, please read the information about our comprehensive training program. The professional staff at our training centre will be pleased to provide detailed information. You may contact the centre by telephone at (+49) 6062 78-258.

1.3 Safety markings on components



DANGER! High voltage!



DANGER! Corrosive battery acid!



DANGER! Hazardous light emissions
(optical fibre cable emitters)!



Disconnect mains power before opening!



Lug for connecting PE conductor only!



Functional earthing or low-noise earth only!



Screened conductor only!

1.4 Safety instructions in this manual




DANGER

This symbol is used wherever insufficient or lacking observance of this instruction can result in **personal injury**.



CAUTION

This symbol is used wherever insufficient or lacking observance of instructions can result in **damage to equipment or data files**.

 This symbol is used to alert the user to an item of special interest.

1.5 Safety instructions for the described product

**DANGER**

Fatal injury hazard through ineffective Emergency-STOP devices! Emergency-STOP safety devices must remain effective and accessible during all operating modes of the system. The release of functional locks imposed by Emergency-STOP devices must never be allowed to cause an uncontrolled system restart! Before restoring power to the system, test the Emergency-STOP sequence!

**DANGER**

Danger to persons and equipment!
Test every new program before operating the system!

**DANGER**

Retrofits or modifications may interfere with the safety of the products described hereunder!
The consequences may be severe personal injury or damage to equipment or the environment. Therefore, any system retrofitting or modification utilizing equipment components from other manufacturers will require express approval by Bosch.


1.6 Documentation, software release and trademarks

Relevant documentation

The present manual provides the user with comprehensive information about programming the

- PCL (Software PLC)
- CL550

according to IEC 61131-3.

 **All informations about PCL are also valid for the integrated controllers iPCL and PCLrho4.0, even if they are not mentioned in this manual.**

Overview of available manuals:

Manuals	Language	Order no.
PCL and CL550, Programming and Operation, Software Manual	english	1070 072 189
iPCL, System Description an Programming Manual	english	1070 073 875

 **In this manual the floppy disk drive always uses drive letter A:, and the hard disk drive always uses drive letter C:.**

Special keys or key combinations are shown enclosed in pointed brackets:

- Named keys: e.g., <Enter>, <PgUp>,
- Key combinations (pressed simultaneously): e.g., <Ctrl> + <PgUp>

Release

 **The descriptive information contained in this manual applies to:**

Software:	WinSPS	Version 3.1 and later
Firmware:	PCL	Version 2.3 and later
	CL550	Version 1.4 and later
	iPCL	NC software version 7.3 and later
	PCLrho4.0	Version VO04L and later

Trademarks

All trademarks referring to software that is installed on Bosch products when shipped from the factory represent the property of their respective owners.

At the time of shipment from the factory, all installed software is protected by copyright. Software may therefore be duplicated only with the prior permission of the respective manufacturer or copyright owner.

MS-DOS® and Windows™ are registered trademarks of Microsoft Corporation.

2 Quick start and input examples

Various working steps should be explained

- Project default settings
- Edit IEC file
- Check symbol file
- Load program into the controller
- Observe and test the program on the monitor

using a simple example.

Additional detailed input examples can be found in the help of WinSPS Software, section "Introduction to WinSPS".

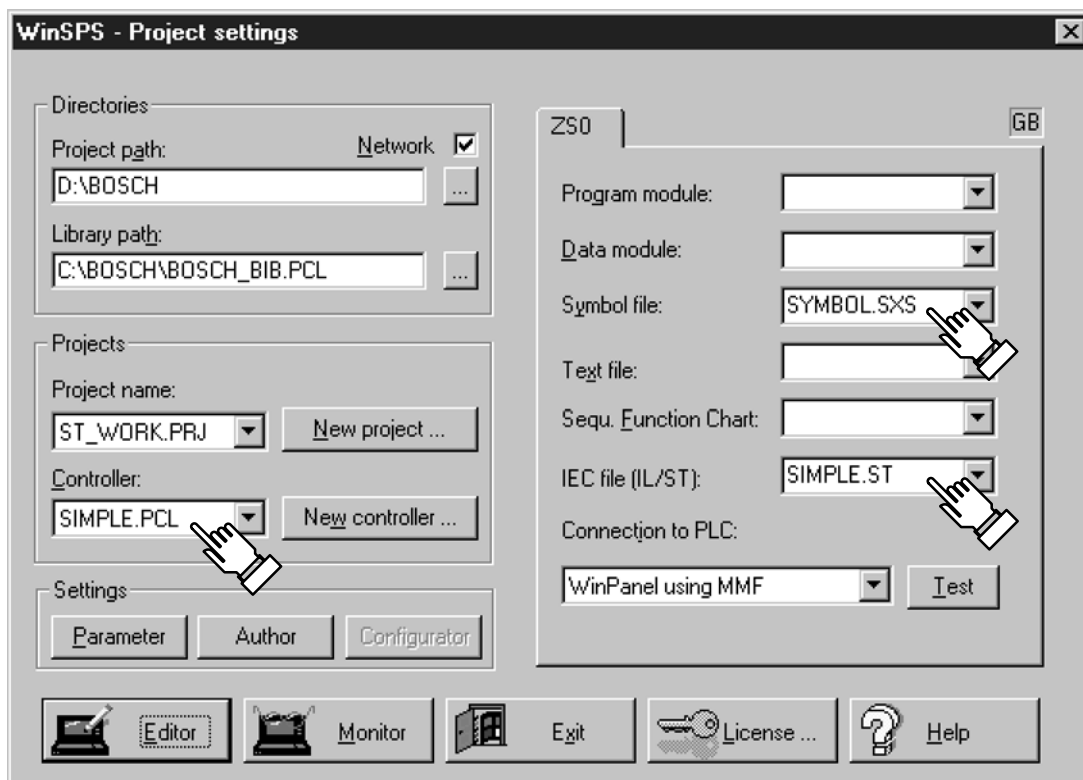
2.1 Project Default settings

In the project default settings of the WinSPS, files and access paths are specified. For the IEC programming, a separate **license** is required. The licensing is likewise called up in the default settings window, also refer to section 4.

The following illustration shows which input fields are important for the editing of IEC programs:

- **Controller:** Only controller of the type **PCL**, **iPCL**, **PCLrho4.0** or **CL550** can be used.
- **Symbol file:** In the symbol file, various inputs of WinSPS are managed automatically.
- **IEC file (IL/ST):** For the editing of IEC files, a filename must be entered in this input field.
The programming language is identified through the file extension :
Instruction List (IL): ".IL"
Structured Text (ST): ".ST"
The file extension must be entered by the user.

Start the editor for entering the program by pressing the appropriate button.



Project default settings

2.2 Programming variations

The WinSPS allows you to use two variations of the IEC programming. Detailed information and examples concerning this can be found in section 11:

- 1) Combination of IEC modules and “classical programming languages”
- 2) IEC program without classical parts

The first variation is particularly useful when you want to program specific functions using IEC modules, however, other control functions are programmed e.g. in the classical programming language Bosch-IL. In this case, the IEC modules are called up from the IL, but not vice versa.

In case of the second variation on the other hand, only IEC instructions are allowed. This is then meaningful when only the programming languages of the IEC 61131-3 are to be used in a control program.

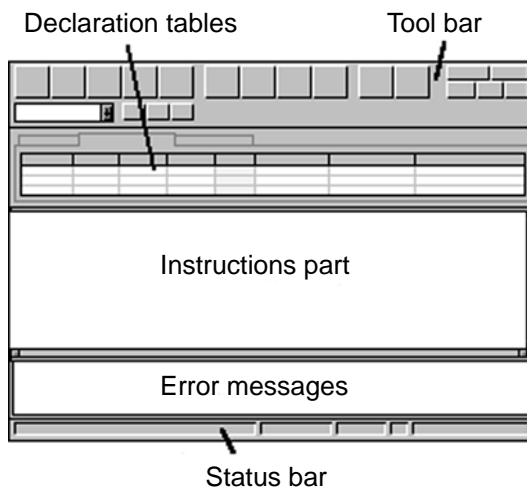
The following program example shows two variations without classical parts. The first variation is shown in section 11.

2.3 Edit IEC file

The entry in the WinSPS editor takes place in the program editor: Button *Prog.* and button *IEC*.

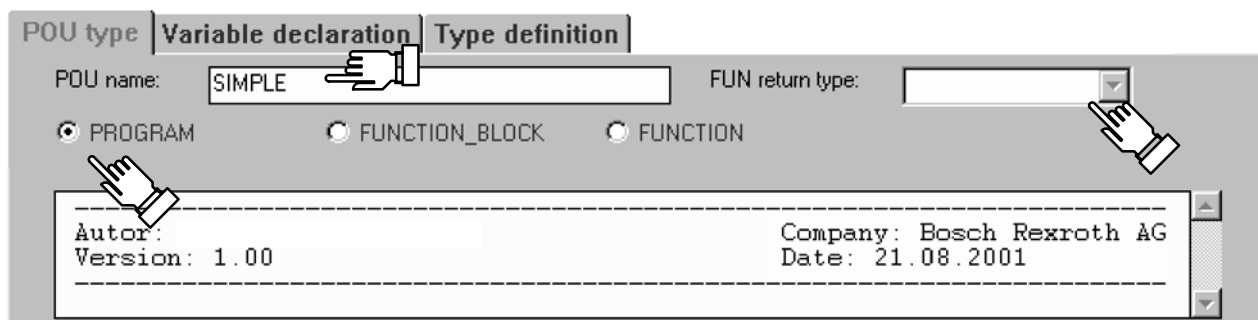


The editor is divided in different areas:



Declaration tables

In the first tab of the declaration tables the **POU name** and the **POU type** PROGRAM, FUNCTION_BLOCK or FUNCTION can be entered. Moreover, in case of a FUNCTION, the data type of the function value (FUN Return type) is set.



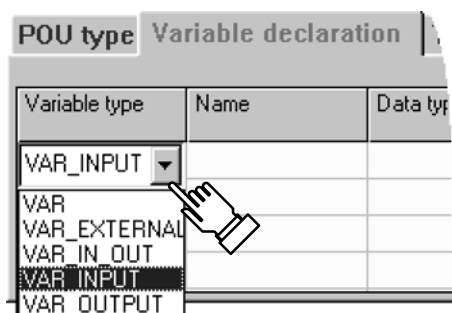
CAUTION

If you make modifications in an input field, you must confirm the input before loading the file in the controller . Changes are accepted only after the confirmation – e.g. using the <Enter> or <Tab> key!

The second tab allows the entry of **variable declarations** by the user. Every variable is entered row by row.

POU type							
Variable declaration							
Type definition							
Variable type	Name	Data type	Initial value	Address	Attribute	Monitor data	Comment
VAR	OUTPUT	BOOL		%Q1.7			
VAR	INPUT	BOOL		%I0.3			

The fields “variable type”, “data type” and “attribute” are selection fields. By clicking on this fields, a list of selection options is offered, refer to the illustration below. In the other fields, the entries are made from the keyboard.



The variable declaration can be made even by purely text input. The switch-over between the declaration tables and text input takes place from the button:



Example of the text input:

```
VAR
  OUTPUT AT %Q1.7 : BOOL;
  INPUT AT %I0.3 : BOOL;
END_VAR
```


The third tab allows the entry of local **type definitions**. Detailed information and input examples concerning this are shown in section 5.1.3.

Instructions part

Instructions of the programming language **IL** or **ST**, can be entered directly in the instructions part, refer to the examples:

POU type							
Variable declaration			Type definition				
Variable type	Name	Data type	Initial value	Address	Attribute	Monitor data	Comment
VAR	OUTPUT	BOOL		%Q1.7			
VAR	INPUT	BOOL		%I0.3			


LDN INPUT
ST OUTPUT



Instructions in the programming language IL

POU type							
Variable declaration			Type definition				
Variable type	Name	Data type	Initial value	Address	Attribute	Monitor data	Comment
VAR	OUTPUT	BOOL		%Q1.7			
VAR	INPUT	BOOL		%I0.3			

OUTPUT := NOT INPUT;



Instruction in the programming language ST

2.4 Check symbol file

On editing and completion of the IEC files,

1. every single IEC file (POU) is compiled i.e. translated into program code
2. all files of the control project are combined into an integrated program (link, Create new project)
3. the integrated program is loaded in the controller.

The working steps can be called up individually or jointly from the menu function *controller* ► *load*. Detailed information concerning this can be found in section 10.

The symbol file should be checked and in necessary, changed so that the second step „Create new project“ can be carried out. The symbol file is called up using the button



Normally, WinSPS automatically manages the entries in the symbol file. An entry must be changed manually if the need be. At the position for "OM1", the module for main program (PROGRAM POU) must be entered.

Example:

```
OM1,R SIMPLE ; Cyclic program processing
```

The example shows the entry of the module "SIMPLE" in the symbol file. In the current control folder, the file "Simple.IL" or "Simple.ST" must exist, which must be of the POU type "PROGRAM".

2.5 Load program in the controller

The menu function *controller* ► *Load* opens a dialog window. Activate the option "load integrated program" and start the loading process.

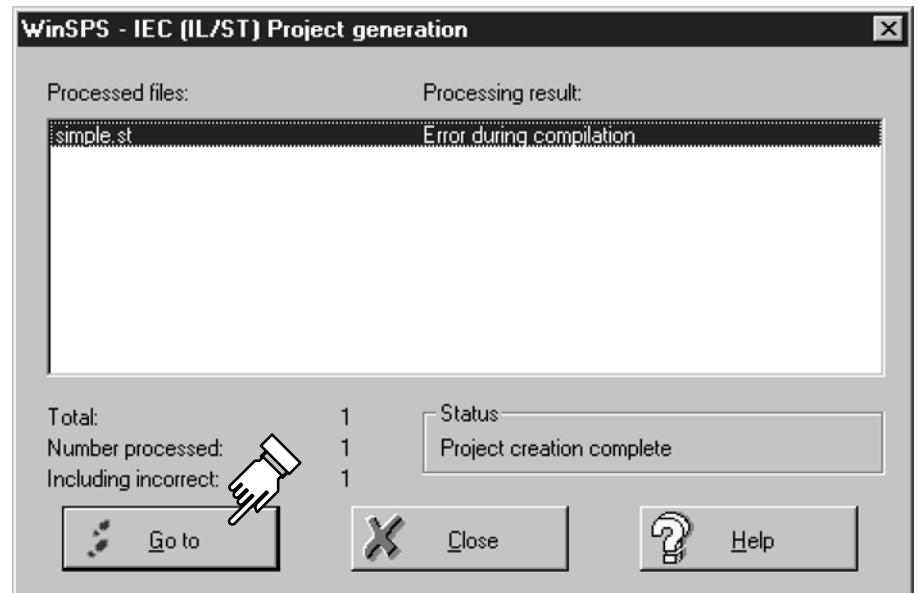


DANGER

Do not load the sample program into an active system!

Before loading, WinSPS checks the project to be loaded. Only error free compiled and linked modules are loaded. If necessary, before loading, the compiler or linker function for these modules is automatically called up.

Error messages are outputted should errors be detected during compilation or linking. Using the button "Go to", one can jump to the error position within the module.



Project generation during the loading process

2.6 Observe and test the program on the monitor

On pressing the button "IEC", the monitor for IEC modules is shown.



In the declaration tables, the current process values of a variable are shown in the column "monitor data".

In future versions of the WinSPS with the associated firmware versions for PCL or CL550, in the instructions part, the current process values shall be shown matching with the current instruction row.

Further information can be found in section 10.5.

Variable type	Name	Data type	Initial value	Address	Attribute	Monitor data	Comment
VAR	OUTPUT	BOOL		%Q1.7		FALSE	
VAR	INPUT	BOOL		%I0.3		FALSE	

```
OUTPUT := NOT INPUT;
```

Monitor detail with an example of the programming language ST

3 Introduction

3.1 What is IEC 61131-3?

At the beginning of the nineties, the International Electrotechnical Commission (IEC) established the standard IEC 1131. At the end of that decade, this standard was renamed as IEC 61131. This standard standardizes – previously manufacturer-dependent – PLC programming. Part 3 (IEC 61131-3) of this international standard contains specifications of the programming languages. As German standard DIN IEC 61131-3, it moreover replaces the standards DIN 19239, DIN 40719T6 and the VDI guidelines VDI 2880 page 4.

In order to understand the IEC 61131-3 in greater detail, we recommend the following literature:

John, Tiegelkamp: IEC 61131-3: Programming Industrial Automation Systems, Springer Verlag, Berlin,

and the information available in the Internet at PLCopen:

www.plcopen.org

3.2 Programming languages of the IEC 61131-3

The languages retained from the "classical" PLC programming

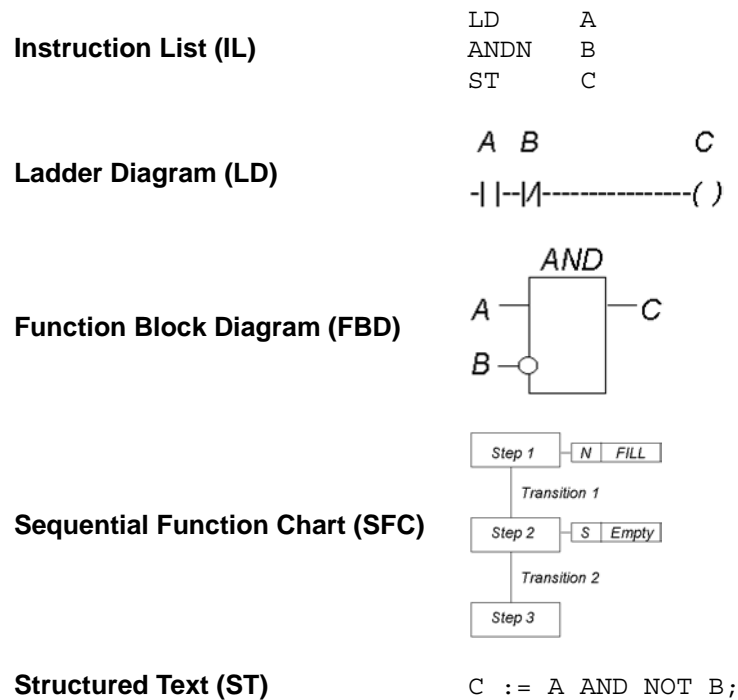
- Instruction List (IL)
- Ladder Diagram (LD)
- Function Block Diagram (FBD), as used by Bosch earlier: Function plan

were also included in the standard like the programming languages designed according to the current requirements:

- Sequential Function Chart (SFC)
- Structured Text (ST)

To some extent, the different languages allow conversion into each other. They can also be mixed virtually in any way so that within the framework of a project e.g. an ST module can be called up from a programming step of the SFC.


Examples



Programming languages of the WinSPS

The programming system WinSPS presently supports the programming languages IL and ST in conformance with the standard IEC 61131-3. The programming languages LD, FBD and the presentation method SFC are based on IEC 61131-3 in further sections.

Parallel to the IL as per IEC, programming can also be done in WinSPS in the established, "classical" IL. The classical IL is invoked in the editor and monitor using the button *IL*, the Instruction List as per IEC using the button *IEC*.

-  **In order to prevent mix-up with the established Bosch programming languages, the following language application is made:**
IEC-IL: Instruction List as per IEC 61131-3.
Bosch-IL: Classical Instruction List (based on DIN 19239).

3.2.1 The programming language IL

The Instruction List (IL) as per IEC 61131-3 is a machine-like programming language. Machine-like means that the instructions can be directly converted into the binary machine code of the PLC.

In comparison to the programming language ST, multiple program rows are required in IL in order to formulate an instruction.

The programming language IL is described in detail in section 8.

3.2.2 The programming language ST

The Structured Text (ST) is a text-like higher programming language. In comparison to machine-like IL, ST is a programming language, in which extensive language constructs allow a very compact formulation of the programming task.

An ST program consists of instructions. In an instruction, values are worked out and assigned, modules are called up and exited, and command flow is controlled.

ST offers the advantage that an open program structure can be realized. ST has a lesser efficiency – for example, in comparison IL. The programs are slower depending upon the complexity.

The programming language ST is described in detail in section 9.

3.3 Why use programming languages as per IEC 61131-3 ?

A great advantage of the programming languages as per IEC 61131-3 is evident when used for different PLC systems. Reusability and interchangeability of programs simplifies the portability between various systems.

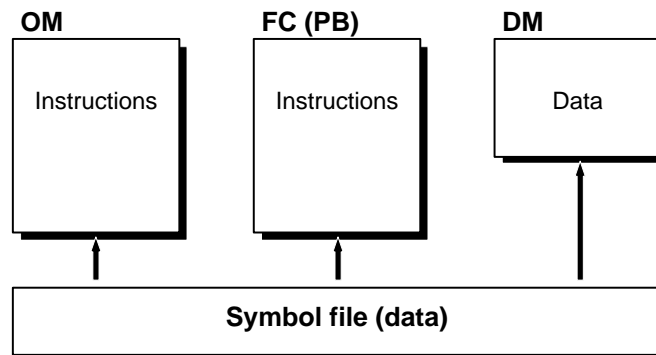
The user enjoys the advantage of a uniform language and program structure. As a result of this, there is an advantage in terms of savings in the training of the application programmers.

Due to standardization and certification, program systems can be compared and evaluated among themselves, refer to section 3.6. The programs can be easily ported between various systems.

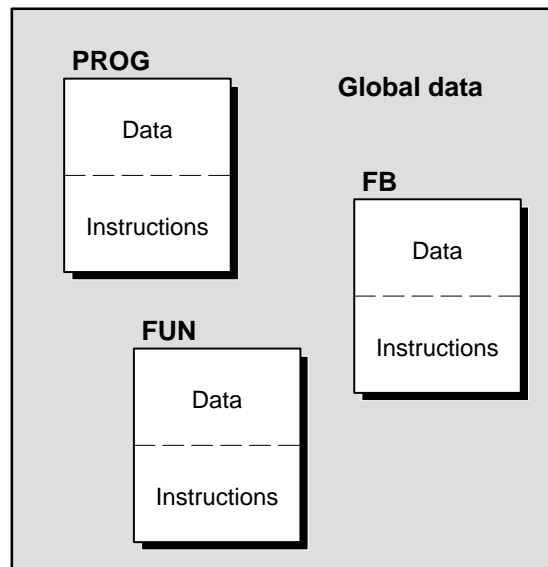
3.4 Difference from “classical” programming languages

- The IEC 61131-3 is an international standard. The syntax of the programming language is basically in the English language.
- The IEC 61131-3 specifies type, structure and contents of modules. It acts the same way with the data to be processed. The standardized data organization makes data modules, data arrays and symbol file redundant, refer to the illustration below. In this regard, also refer to the section 4.2.2.
- IL, LD and FBD are based on the above-mentioned structure and data administration. With regard to these points, they are fundamentally different from the classical procedures.
- SFC is a presentation method based on the IEC 60848 (French GRAFCET standard).
- ST is a new language which was realized only with the introduction of IEC 61131-3.

- If in a “classical” program, data is declared globally (in the symbol file), while programming as per IEC 61131-3, there exists the option of defining the data locally also and there, it can be protected against unintentional access.
- Input and output parameters also have access protection in case of module calls.
- Checking of the data formats in case of variables and direct memory addresses of the PLC such as E/A/M. As a result of this, access of data in incorrect format is ruled out.
- Variables – apart from physical addresses – do not have a fixed memory area in the PLC. This is automatically assigned by WinSPS during program set-up .
- The instance building in case of function blocks allows implementation of modules “with memory”. This principle is known from classical counter and timer functions. It can however also be used for user function blocks in case of modules as per IEC.



Modules and data of the “classical” programming



Modules and data of the programming as per IEC 61131-3

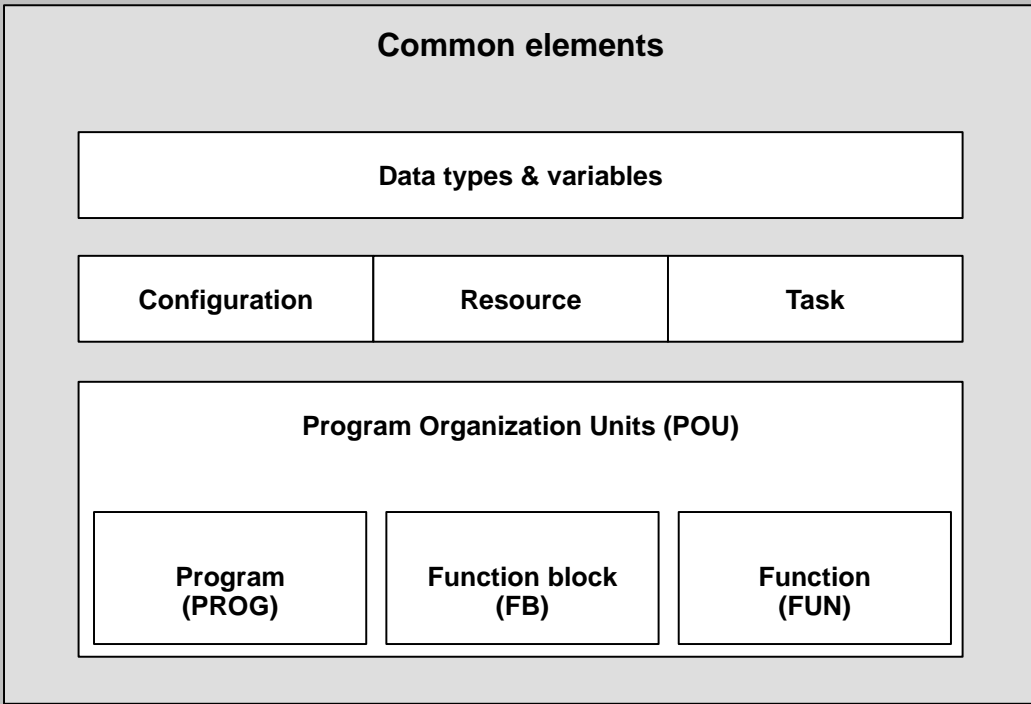
3.5 Model of the programming as per IEC 61131-3

Basis of the IEC is a model which clearly distinguishes commonly usable elements from the individual programming languages, refer to the following illustration. As a result of this, many features can be used the same way for different programming languages.

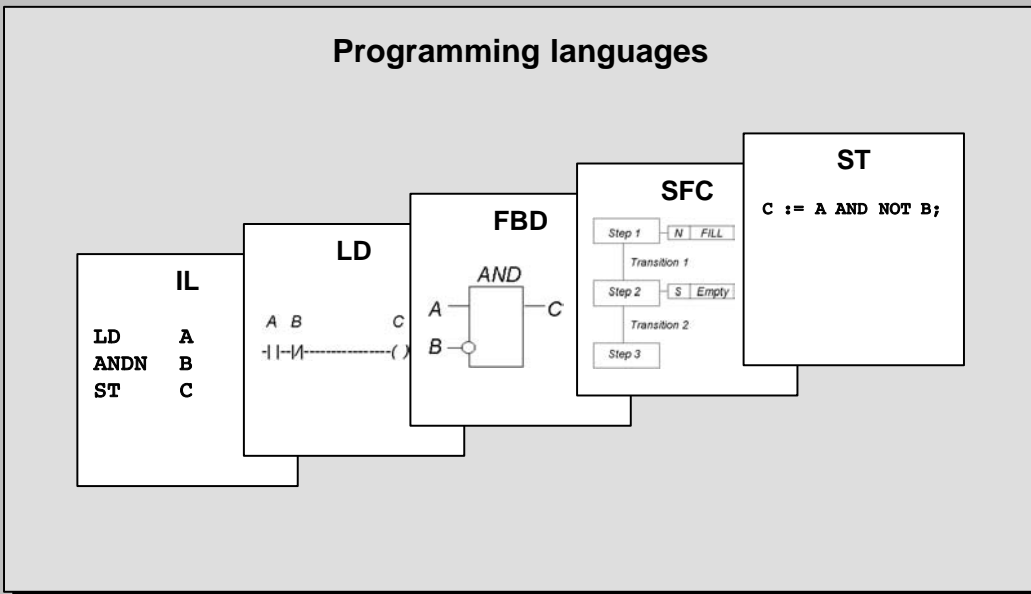
The common elements are handled in detail in sections 6, 7 and 12, the programming languages are handled in sections 8 and 9.

IEC 61131-3 standard

Common elements



Programming languages



Common and specific elements of the IEC 61131-3

Data types and variables

Format, contents and syntax of data types and variables are fixed by the IEC and to a great extent, they are identical for all IEC programming languages. The programming system WinSPS monitors the usage of variables according to their type, as a result of which program errors are virtually ruled out.

The access to hardware addresses such as inputs, outputs and labels is possible. However, hardware addresses (physical addresses) must as variables be assigned to a data type in advance. With this, even here, the usage of variables according to their type is monitored. Moreover, the declaration of hardware addresses is allowed only in a single module. This increases the portability to systems with other hardware features.

Every variable has a standard or user defined initial value. Variables can be assigned the attribute "RETAIN" in order to allow a remanent behavior.

Arrays and data structures allow the implementation of very complex data models.

Configuration, Resource, Task

The IEC plans these elements for runtime characteristics, the assignment of the PLC hardware and for communication links between controls.

These elements are presently not supported.

Program Organization Units

Program Organization Units (POU) are the modules in the IEC 61131-3. Features and interfaces of the modules are clearly defined. Every module has its own data range. As a result, even modules "with memory" can be implemented. These modules can be called up repeatedly within a program cycle, without mutually affecting each other.

An efficient parameterization of the module interface allows protection of variables against unauthorized access, passing or return of defined variables in a module, passing of pointers to variables and access to global variables.

Typical PLC functionalities such as time, counters or arithmetic functions are standardized by the IEC as standard functions and function modules. These have clearly defined interfaces and provide established results. They can be called up from all programming languages.

Programming languages

Depending on the application area, a selection can be made among the five programming languages. All above-mentioned features of the "common elements" allow their use in any of the five programming languages.

WinSPS presently supports two programming languages: IL and ST.

3.6 Compatibility and fulfillment of standard

The IEC 61131 is not a mandatory rule book but a guideline, which can be followed by the manufacturers to greater or lesser degree. The respective manufacturer must disclose the degree of implementation. For this, the IEC intends a certification, which provides the user exact information concerning the compatibility and norm fulfillment.

 **Pay attention to the norm fulfillment of the WinSPS in section 13.**

The Bosch programming system WinSPS has the “Base Level Certificate” for the programming language ST.

3.7 Programming system and controller

Bosch supports programming in compliance with IEC with the programming system WinSPS. A lot of help is available for programming and commissioning. The following programming languages conform to IEC 61131-3:

- Instruction List
- Structured Text

Other programming languages in WinSPS are strongly based on the IEC 61131-3.

The programming as per IEC is allowed with the controllers listed in section 1.6. Kindly pay attention to the instructions concerning the version number even there. Due to the constant advancement of programming system and controller, attention is to be paid to the appropriate firmware and software versions.

The listed controllers are suitable due to their modern and open hardware architecture, specially in view of the specifications of the IEC 61131-3. With firmware updates, these controllers can be adapted to the future developments. For this, also pay attention to various instructions concerning present function extensions.

4 Project preparations

The programming tool WinSPS provides an easy-to-use editor for inputting data and instructions in various IEC programming languages. The monitor allows program tracking and data observation for commissioning and error detection. Before calling up the editor or monitor, a few default settings must be made.

4.1 Installation

The WinSPS software (order no. 1070 077 925) can be installed on the PC from the CD "PLC Tools" or directly from the internet. The internet address is: "www.BoschRexroth.de". For the installation, kindly pay attention to the accompanying text files.

4.2 Default settings

The dialog window for default settings appears after the WinSPS software is invoked. Here, all project and control related settings are made. These are retained even after exiting the program.

The default settings are divided in various functions:

- License
- Directories
- Projects
- Settings
- File names and link to the control.

4.2.1 Licensing the programming languages

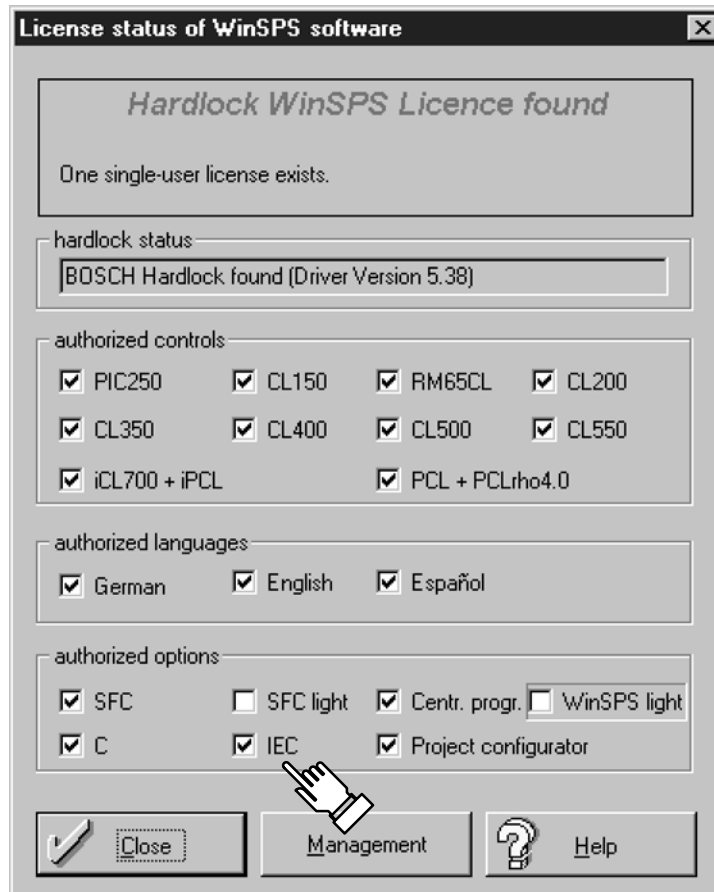
A separate license is required for programming as per IEC 61131-3. The license is called up in the default settings window. If no valid license for the IEC programming languages exists, such projects cannot be created or processed.

In the license dialog window, you have the choice between various types of licenses.

- Soft-license: A license is installed on the hard disk of the programming unit, without any requirement of additional hardware.

- **Hardlock license:** Instead of a software license, you can use a hardlock for plugging into the parallel interface or as an internal ISA Bus card (IntroCard). You can get the hardlock from the Bosch Software Service, see cover page for address. A hardlock can simultaneously hold the license for WinSPS, WinCAN, WinDP and WinPanel.
- **14 Days test license:** There is an alternate option to install a test licence (= free of cost, 14 days test licence). A test licence can be installed only once on the computer.

With the help of the function *License ► Show*, which is called up in the project default settings, the current license status can be displayed, refer to the image.

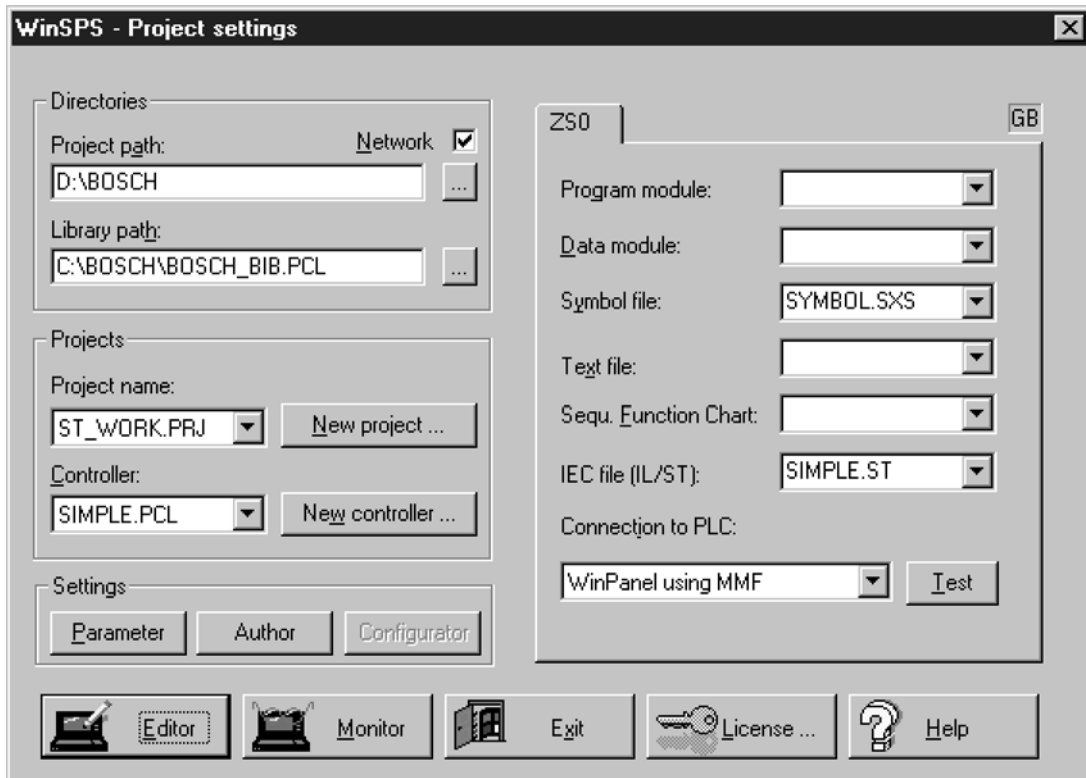


License status: License for IEC programming

Please also pay attention to the instructions concerning the license in the WinSPS help and in the file "ReadPLC.doc" in the subdirectory "DOC" of the WinSPS Installation directory.

4.2.2 Project default settings

In the project default settings, files and access paths are specified. The default settings relevant for the processing of IEC programs are shown hereinafter. The meanings of the rest of the input fields are explained in detail in the WinSPS help.



Example of a project default settings

Program file

This field can be left empty.

The data concerning the program files is required only when IEC modules are to be called from “classical programming languages” (e.g. classical IL). This option is discussed in detail in chapter 11.

In case of IEC programming projects, the WinSPS generates automatic program modules. For these modules, file names are reserved that may not be used for other purposes.

Reserved program files are the file names of the used IEC files with the extension “.IL “ and “.ST”. WinSPS generates new program files with the extension “.PXO”. Similarly, the symbol names of the program modules FC512 to FC1023 are reserved. The numbering is control dependent and can be adjusted through the symbol file depending upon the application. More information in this regard can be obtained in chapter 10.3.

Symbol file

The programming as per IEC 61131-3 does not foresee any symbol file. In order to however allow a mixing of “classical” programming languages with programming languages as per IEC, the symbol file is not dispensed with due to compatibility reasons. The processing of the symbol file is differentiated depending upon the structure of the program:

- Combination of an IEC program with classical program parts:**
 All used symbols, as well as program and data modules of the classical programming language must be entered manually in the symbol file. This is discussed in detail in chapter 11.4.


- **Purely IEC programming:**

The symbol file is managed automatically from WinSPS. No entries should be made by hand.

Data module file

The information concerning a data module is not required. The WinSPS automatically generates data modules. For these modules, file names are reserved that may not be used for other purposes.

Reserved data module files are IM0.PXD, as well as IM512.PXD to IM1023.PXD (ascending numbering). Similarly, the symbols of the associated data modules DM0, as well as DM512 to DM1023 are reserved. The numbering is control dependent and can be adjusted through the symbol file depending upon the application. More information in this regard can be obtained in chapter 10.3.

 **The automatic generation of data modules refers to WinSPS version 3.1 and lower. In future versions of the WinSPS, no data modules shall be reserved and set up.**

IEC file (IL/ST)

The editor for IEC programming languages can be activated only when a file name is entered in field "IEC file (IL/ST)". With the input of a file name in this field, a module for the programming as per IEC 61131-3 is set up. With regard to modules, the IEC speaks of "program organization units", in short: POU.

The used programming language instruction list (IL) or structured text (ST) is determined by the file extension ".IL" or ".ST". The file extension must be entered by the user.

 **The used programming language is determined by the file extension!**

POU and file names may not appear more than once in a project. The POU name can be entered in the editor.

Example:

File name: MODULE.IL	POU Name: Module_1
File name: MODULE.ST	POU Name: Module_2

Though the POU names of both the modules in this example are different, yet the filenames within a project may not be identical; not even when they are used for different programming languages as shown here.

5 Writing programs in the WinSPS Editor

The WinSPS Editor supports program writing as per IEC using an easy-to-use user interface which minimizes the input fields.

The input takes place in the program editor, which is activated using the button "Progr.". On pressing the button "IEC", the programming as per IEC 61131-3 is selected.



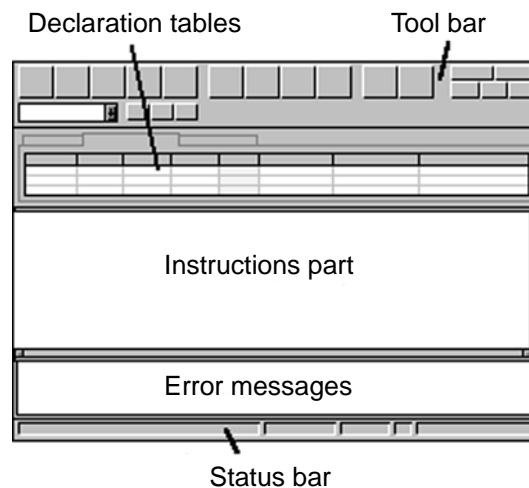
The used programming language instruction list (IEC-IL) or structured text (ST) is determined by the file extension "*.IL" or "*.ST". So long as in the project default settings, no filename is entered in the field "IEC file", the programming as per IEC cannot be activated.

All inputs are accepted in the current IEC file *.IL or *.ST. These files are converted into other files later on with the compilation or generation of the project by WinSPS. In this case, it involves reserved programs and modules. Moreover, entries are accepted in the current symbol file. In this regard, refer to the section 10.

Editor range

The editor for IEC files can be divided in three sections:

- Declaration tables
- Instructions part
- Error messages

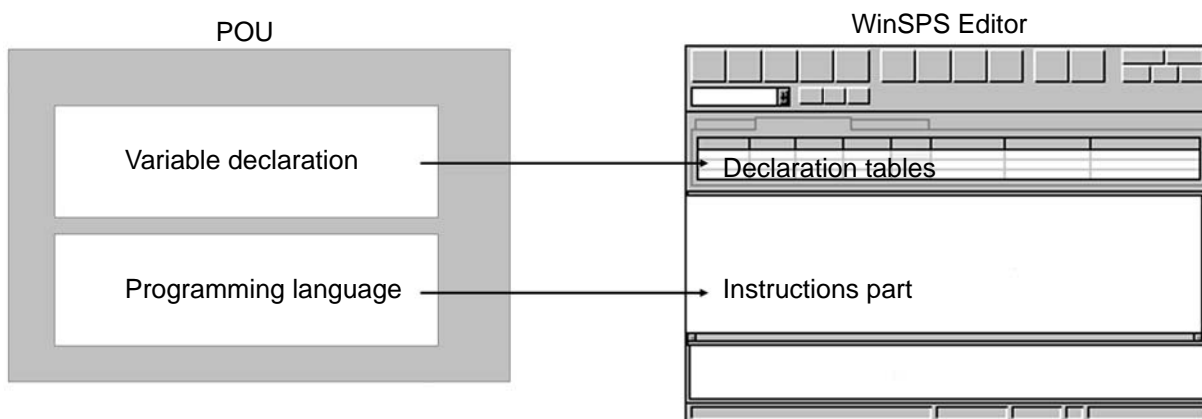


In the declaration table, POU type and name, variables and local type definitions are edited. In the instructions part, the program instructions are entered. When there are erroneous inputs, error messages are outputted in the lower window area by the subsequent compiler or linker run.

Common elements

A feature of the IEC 61131-3 is the definition of common elements which can be used in different programming languages in the same way. Within a POU (module) the variable declarations and type definitions are common elements i.e. independent of the used programming language, the variable declaration and type definition are identical.

A POU as per IEC 61131-3 is divided in the declaration part for variables and in the instructions part of the respective programming language. As a result, the commonly usable elements can be clearly demarcated by the programming language. In the WinSPS Editor, this demarcation is achieved through the parts "declaration tables" and "instructions part".



5.1 Declaration tables

With the help of the declaration tables, the input of variables and data types is simplified. All inputs are converted into the syntax of the IEC so that the errors can be minimised.

The inputs can however also be made without declaration tables in pure text programming. The switchover between the declaration tables and text input takes place from the button:



The declaration is divided in various input masks (tables) which are represented by three tabs:

- POU type (general data)
- Variable declaration
- Data type definitions (user-defined, local)

5.1.1 POU type

In the input mask, general data concerning the current module (POU = Program Organisation Unit) is specified.

Example of a POU of the type PROGRAM with the name SIMPLE



CAUTION

If you make modifications in an input field, you must confirm the input before loading the file in the controller. Changes are accepted only after the confirmation – e.g. using the <Enter> or <Tab> key!

POU name

Normally, the POU name initially corresponds to the current filename *.IL or *.ST. The name can be changed and may not be identical to the filename. The length of the POU name may not be more than 32 characters. Specifications for the identifiers of the IEC are to be followed, refer to section 7.1.2.

POU type

The characteristic of the module is determined from the POU type. The three POU types PROGRAM, FUNCTION_BLOCK and FUNCTION are specified using the selection switch.

FUN Return Type

In case of a POU of the type FUNCTION, the data type of the function value can be selected in the field FUN Return Type.

Comment

The large input field in the lower area allows the entry of any comment text. This text is placed at the top of the file. No comment markings should be entered, these are automatically added by WinSPS.

Further information concerning the POU can be found in section 6.1.

5.1.2 Variable declaration

In this table, the variables of the current POU are edited. Every used variable must be declared.

Every variable is represented by a row of the table. In order to enter a new variable, select an empty row. In order to change an existing entry, position on the row or the column to be changed. In order to delete a variable, after positioning, press the button:

**CAUTION**

If you make modifications in an input field, you must confirm the input before loading the file in the controller.

Changes are accepted only after the confirmation – e.g. using the <Enter> or <Tab> key!

Variable type

A selection window opens on clicking a field in the variable type. The selection option changes depending upon the POU type. The following table shows the possible variable types and the POU types, in which these are available:

Variable type	Explanation	PROG	FB	FUN
VAR	Local variable within the POU	Yes	Yes	Yes
VAR_INPUT	Input variable	Yes	Yes	Yes
VAR_OUTPUT	Output variable	Yes	Yes	–
VAR_IN_OUT	Input and output variable	Yes	Yes	–
VAR_EXTERNAL	Global variable, which is declared in another POU	Yes	Yes	–
VAR_GLOBAL	Global variable, which is declared in this POU.	Yes	–	–
VAR_ACCESS*	Access paths	Yes	–	–

* The variable type VAR_ACCESS is presently not supported.

Multiple declarations of a variable type are automatically put together by WinSPS. Here, the variables are at times rearranged so that the variables of the same type follow directly after one another.

Detailed information can be found in section 6.3.1.

Name

The name of the variables are entered in the second table field. This name must correspond to the identifier specifications of the IEC, refer to section 7.1.2.

Data type

A selection window opens on clicking a field in the data type. Elementary data types predefined by the IEC as well as global user-defined data types can be selected. Global user-defined (derived) data types are defined in the type editor, refer to section 5.5. If you use local user-defined data types, the type names can be entered by hand. Local, user-defined data types can be edited in the declaration table of the type definition.

The elementary data types are listed in section 7.2.1. Information concerning the use of user-defined data types can be found in this section further below, as well as in section 7.2.2.

Initial value

Here, the variable can be assigned an initial value. This value is to be entered as numeric, string or time literal depending upon the data type.

Example:

Data type TIME_OF_DAY involves a time variable. The initial value must therefore be available in the following form, e.g. **tod#14:30:00.0**

Further information concerning literals can be found in section 7.1.3.

Address

In case of POU type PROGRAM, there is an option to assign to the variables physical addresses such as inputs and outputs of the PLC. Physical addresses can be declared only in the PROGRAM POU.

Example:

The PLC input 3.0 should be assigned to a variable name: The input field "Address" contains **%IX3.0**. The data type must be BOOL.


Information concerning format and the use of physical addresses can be found in section 7.3.4.

Attribute

The IEC defines attributes, using which other features can be assigned to the variables. A selection window opens on clicking an attribute field in the data type. The following table shows all attributes and their meaning:

Attribute	Explanation
RETAIN	Battery backed-up, remanent
CONSTANT	Constant
R_EDGE	Rising edge
F_EDGE	Falling edge
READ_ONLY	Wite-protected
READ_WRITE	Reading and writing access

* The attributes related to the edge control are presently not supported. Edge control can be realised using the standard function blocks R_TRIG and F_TRIG, refer to section.

 **All variables are basically handled as with RETAIN attribute. Exceptions are the standard FBs. These are basically not remanent i.e. they are re-initialized after every STOP/RUN switchover. The remnance characteristic can moreover be configured in the organisation module OM2. Follow the instructions in the software manual "PCL and CL550 (order no. 1070 072 189) or in "iPC" (order no. 1070 073 875.**

Allowed attributes of the individual variable types are listed in the following table:

Variable type	RETAIN	CONSTANT	R_EDGE F_EDGE	READ_ONLY READ_WRITE
VAR	Yes	Yes	–	–
VAR_INPUT	–	–	Yes	–
VAR_OUTPUT	Yes	–	–	–
VAR_IN_OUT	–	–	–	–
VAR_EXTER- NAL	–	–	–	–
VAR_GLOBAL	Yes	Yes	–	–
VAR_ACCESS*	–	–	–	Yes

* The variable type VAR_ACCESS is presently not supported.

Further information concerning attributes are listed in section 7.3.8.

Monitor data

In this column, the current process data of the variables is shown in the monitor of the WinSPS. This column cannot be edited.

Comment

Here, entry of any comment is possible. The comment is given at the end of the line of the variable declaration. No comment markings should be entered, these are automatically added by WinSPS.

Example

The following example shows multiple variable declarations. Subsequently, the conversion of the example into the textual presentation shows:

POU type							
Variable declaration				Type definition			
Variable type	Name	Data type	Initial value	Address	Attribute	Monitor data	Comment
VAR	Local	BOOL		%Q1.0			any comment
VAR_INPUT	In1	INT	15				
VAR_INPUT	In2	DATE					
VAR_OUTPUT	Out	WORD			RETAIN		
VAR_IN_OUT	InOut	LREAL					
VAR_GLOBAL	Global	WORD					
VAR_EXTERNAL	Global	WORD					

Example of different variable declarations (illustrative)

```

VAR
  Local AT %Q1.0 : BOOL;    (* any comment *)
END_VAR

VAR_INPUT
  InOut : LREAL
END_VAR

VAR_INPUT
  In1 : INT := 15;
  In2 : DATE;
END_VAR

VAR_OUTPUT RETAIN
  Out : WORD
END_VAR

VAR_GLOBAL
  Global : WORD
END_VAR

VAR_EXTERNAL
  Global : WORD
END_VAR
    
```

Textual presentation of the above-mentioned example (illustrative)

5.1.3 Type definition

In this part, derived data types of the current POU can be defined. Derived data types – also called type definition – are user defined data types which are based on the “elementary data types”. With type definitions, new data types with extended or altered attributes can be generated. In addition to this, very complex data models can be realised.

The following can be defined:

- User-defined data types
- Data structures
- Enumerations

The type definitions are valid only for the current module (POU). Here, it also involves local type definitions. Global type definitions are made in the symbol editor with the help of the editor for global data types, refer to section 5.5.



CAUTION

If you make modifications in an input field, you must confirm the input before loading the file in the controller. Changes are accepted only after the confirmation – e.g. using the <Enter> or <Tab> key!

Type name

The input field for the type names is used in three different ways. Detailed examples are to be found in later part of this section.

4. Keyword TYPE (standard):

The keyword TYPE stands for the complete type definition of the current POU and is as a result, at a level higher than all other definitions such as data structures and enumerations. All table entries under this keyword are accepted in the POU as user-defined type definitions .

In the pure textual presentation, these type definitions are placed between the keywords TYPE and END_TYPE.

5. Enumeration, ENUM:

Enumerations are specified using the type names. Enter the name of the enumeration in the edit field “Type name” . Delimited by comma, the enumeration elements are entered in the table in the field “Data type” between two brackets () . **The table field “Name” must be empty.**

In the pure textual presentation, these enumerations are placed between the keywords TYPE and END_TYPE. Enumerations are a part of the complete local type definition.

6. Data structures, (STRUCT):

Complex data structures are not created through the standard type TYPE. Instead, any name, to be precise the structure name, is entered in the edit field. In the table , the type definitions of all the sub-elements are subsequently assigned to this structure .

In the pure textual presentation, these type definitions are placed between the keywords TYPE and END_TYPE. The structure name is specified right before this structure block. The structure itself is found between TYPE and END_TYPE. Structures are a part of the complete local type definition.

If within a local type definition, various types are to be put together, e.g. data structures and enumerations, these must be entered one after the other using the field "Type name". A detailed example for this can be found at the end of the type definition description.

In order to enter a new type definition, select an empty row in the table. In order to change an existing entry, position the write cursor on the row or the column to be changed. In order to delete a type definition, after positioning, press the button:



Name

Derived (user-defined) data types get their name in this field. This name must correspond to the identifier specifications of the IEC, refer to section 7.1.2. In case of enumeration types however, this field must remain empty.

The data type is assigned to the type at a higher-level. Thus, the data is assigned e.g. to a data structure when a structure name is specified (refer to illustration) in the field "Type name".

POU type Variable declaration Type definition		
Type name:	System_data	
Name	Data type	Initial value
Measured_value1	BOOL	1
Measured_value2	INT	-15
Measured_value3	UINT	100
Measured_value4	REAL	10.2

Example of a data structure in the type definition

```

TYPE
  System_data:
  STRUCT
    Measured_value1 : BOOL := 1;
    Measured_value2 : INT := -15;
    Measured_value3 : UINT := 100;
    Measured_value4 : REAL := 10.2;
  END_STRUCT;
END_TYPE

```

Textual presentation of the example

Data type

A selection window opens on clicking a field in the data type. Elementary data types predefined by the IEC as well as global user-defined data types can be selected. Global user-defined (derived) data types are defined in the type editor. If you use local user-defined data types, the type names can be entered by hand.

This field is used with another meaning for the enumeration type. Instead of a data type, the enumeration elements are entered between two brackets "()".

The elementary data types are listed in section 7.2.1. Information concerning the use of global user-defined data types can be found in section 5.5.

Initial value

Here, an initial value can be assigned. This value is to be entered as numeric, string or time literal depending upon the data type. Information concerning literals can be found in section 7.1.3.

Comment

Here, entry of any comment is possible. The comment is given at the end of the line. No comment markings should be entered, these are automatically added by WinSPS.

Example:

The following example shows the approach in case of complex type definitions which put together various types within a local type definition. The illustrations show working steps to be performed one after the other. Afterwards, the result is indicated with the help of the textual presentation:

POU type	Variable declaration	Type definition		
Type name: <input type="text" value="TYPE"/>				
Name	Data type	Initial value	Comment	
usiAge	USINT			
usiChildAge	ARRAY [1..20] OF usiAge			

1. Step: User-defined data types

POU type	Variable declaration	Type definition		
Type name: <input type="text" value="eSex"/>				
Name	Data type	Initial value	Comment	
	(M, F)			

2. Step: Definition of an enumeration

POU type		Variable declaration		Type definition	
Type name: <input type="text" value="Person"/>					
Name	Data type	Initial value	Comment		
szName	STRING (20)				
Sex	eSex	F			
Age	INT				

3. Step: Definition of a data structure

```

TYPE
  eSex : (M, F);
  usiAge : USINT;
  aChildAge : ARRAY [1..20] OF usiAge;
Person :
  STRUCT
    szName : STRING (20);
    Sex : eSex := F;
    Age : INT;
  END_STRUCT;
END_TYPE
    
```

Result: Textual presentation of the inputs made above

5.2 Instructions part

In the instructions part, the program instructions are entered. The programming language is identified on the basis of the file extension “.IL” or “.ST” in the project default settings.

The input format is different in different programming languages. The following applies for the textual languages IL and ST: The instructions are entered by the row. Format and examples concerning this are shown in the parts of the respective programming languages.

The different POU types are marked by keywords at the top as well as the end of the file:

POU type	Top Of File	End Of File
PROGRAM	PROGRAM Name	END_PROGRAM
FUNCTION_BLOCK	FUNCTION_BLOCK Name	END_FUNCTION_BLOCK
FUNCTION	FUNCTION Name : Type	END_FUNCTION

So long as the declaration tables are activated, the keywords at the top of the file and at the end of the file are not shown. The WinSPS completes these automatically.

If these declaration tables are deactivated, e.g. using the button:



the instructions part then corresponds to the area between the last variable declaration – ending with the keyword `END_VAR` – and the POU end – before the keyword `END_x` (where `x` corresponds to the POU type, refer to the table above). All program instructions must be entered within this part.

5.3 Error messages

In the IEC editor, the lower window area is used for displaying errors after checking. The checking is activated from the menu functions *File* ► *Create new project* and *File* ► *Compile module* – even using the button:



and also during the program loading, refer to section 10.

In this case, the error message display is erased. On completion of the checking, the error-free run or all errors are indicated. Unless specified otherwise, the display is based on the current file.

```
File D:\BOSCH\ST_WORK.PRJ\SIMPLE.PCL\ZS0\SIMPLE.ST successfully compiled 22.08.
simple.st - 0 error(s), 0 warning(s)
```

Example of an error-free checking

Checking/compiling

The following text appears in case of error-free checking:

```
POU name - 0 error(s), 0 warning(s)
```

Error messages of the compiler (refer to section 10.1) have the following structure:

```
(R,C) Error in the ...part : Error (ID)
```

R indicates the row number – *C* the column number, in which the error has occurred. Subsequently follows the instruction whether the *error lies in the variable part* (declaration part) or in the *program part* (instructions part). After the colon, a description of the error or the warning is displayed in the plain text. At the end of the row, an internal identification number *ID* is outputted in brackets. In this case, differentiation is made between warnings with the identification *W* and errors with the identification *F*.

Example:

```
(5,3) Error in the variable part : Syntax error!  
(E4006)
```

The first bracket shows that the error lies in row 3 and column 5 of the current module. The error was made in the declaration part of the module. It involves a syntax error, which means that the program text does not comply with the formation rules of the programming language. In the bracket at the end, the internal error identification for syntax error 4006 is indicated.

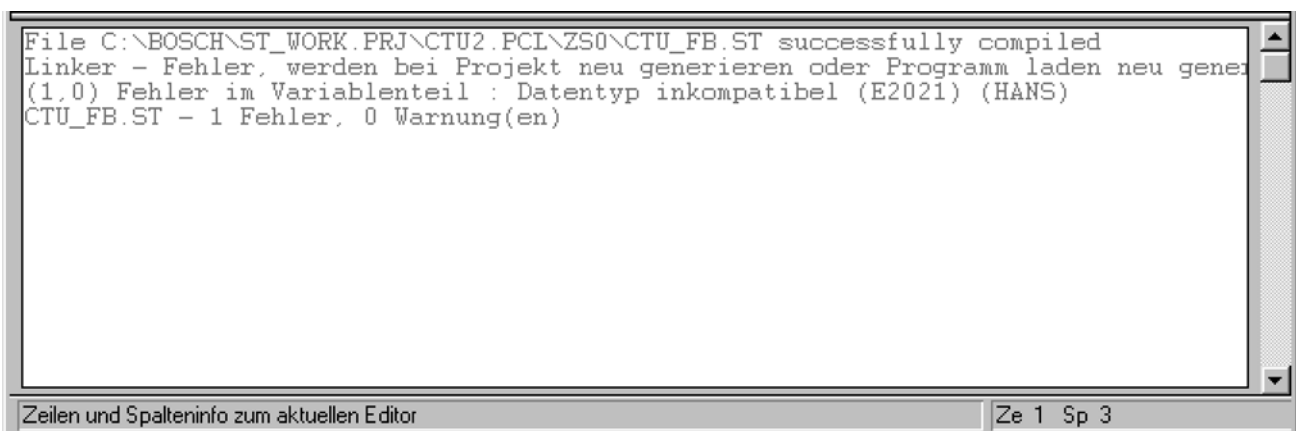
With a double-click on the error message, the write cursor is set in the row containing the error. Thus, errors can be processed directly.

Kindly keep in mind that errors could give rise to consequent errors. So under certain circumstances, the test result shows multiple errors in a program row although there may be only one error. This is a typical characteristic of compilers.

Linker – Create new project

Even the linker (refer to section 10.2) uses the window for the error output. With the linker however, no clear error localization within a module can be carried out and indicated. However, the filename of the POU, in which the error is assumed, is indicated.

The example in the illustration shows a linker error due to an incompatible data type in the module "HANS".




Error message of the linker (Create new project)

5.4 Global variable declaration – variable editor

Under preparation.

5.5 Global type definition – type editor

In the editor for global type definition, derived data types can be defined globally. Derived data types – also called type definition – are user defined data types which are based on the “elementary data types”. With type definitions, new data types with extended or altered attributes can be generated. In addition to this, very complex data models can be realised. Global means that the data types can be used in every POU.

 **All global type definitions are written and in the internal file “Bosch.Typ” and are managed there by WinSPS. If already generated global type definitions can be drawn for other PLC projects, this file can be copied to the corresponding project folder.**

The type editor can be called up within the symbol editor using the following button:



The following can be defined:

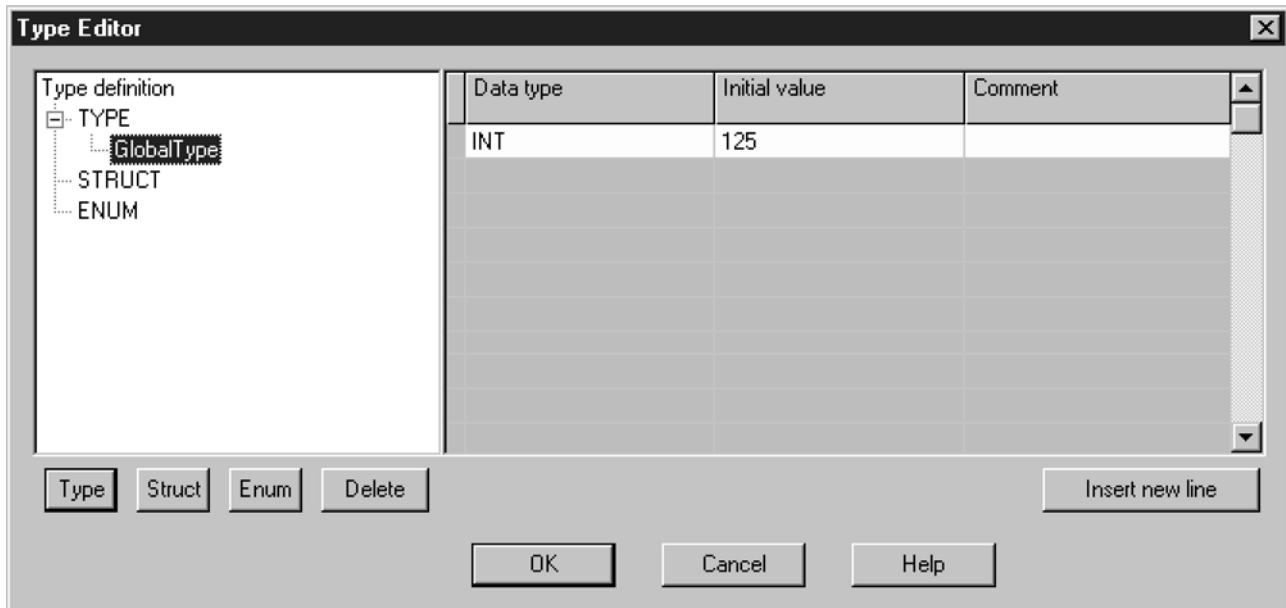
- User-defined data types
- Data structures
- Enumerations

5.5.1 TYPE: Data Type

Here, a user-defined (derived) data type can be defined. Detailed information concerning these data types can be found in section 7.2.2. By pressing the button “Type”, a new data type can be set-up within the tree structure. The new data type initially gets the type name “New type”, which can be renamed by clicking on the name. The name must correspond to the identifier specifications of the IEC, refer to section 7.1.2.

On the right side, a row is available for inputting the data type, an initial value and comments.

Elementary as well as user-defined data types can be selected. An initial value can be optionally assigned in the column. This value is to be entered as numeric, string or time literal depending upon the data type. Information concerning literals can be found in section 7.1.3.



Example of a global type definition

```
TYPE
  GlobalType : INT := 125;
END_TYPE
```

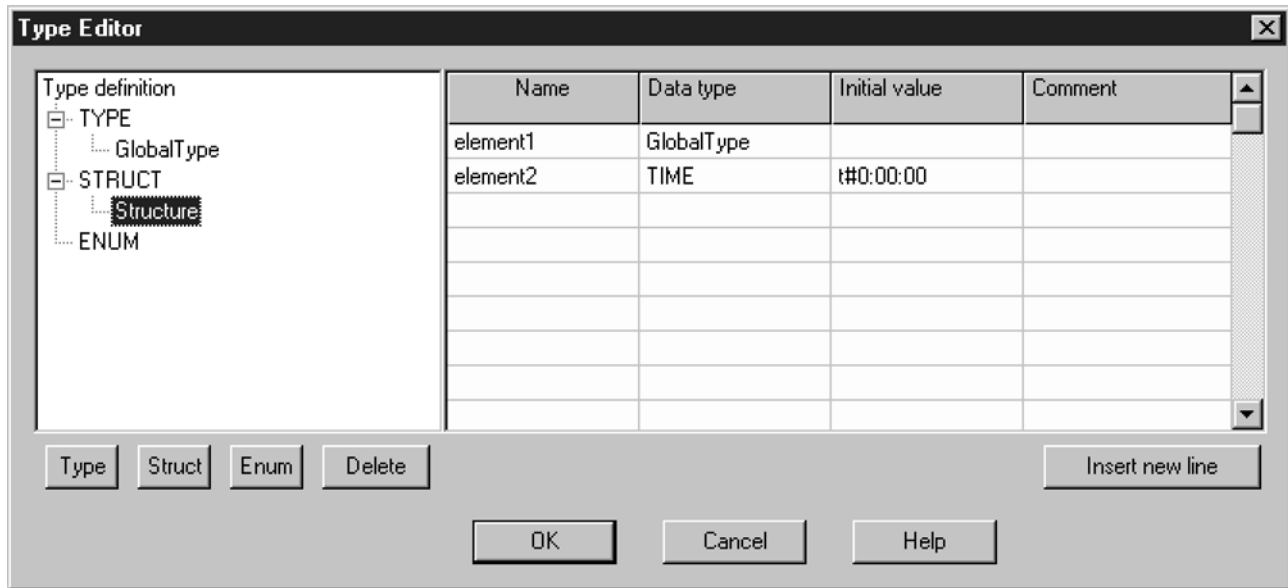
Textual presentation of the example

5.5.2 STRUCT: Data structure

By pressing the button “Struct”, a new data structure can be set-up within the tree structure. Detailed information concerning data structures can be found in section 7.3.7. The structure initially gets the name “New type”, which can be renamed by clicking on the name. The name must correspond to the identifier specifications of the IEC, refer to section 7.1.2.

On the right side, the structure elements can be entered in the table by the row. In case the rows are not sufficient, other rows can be inserted using the corresponding button.

Every structure element (sub-element) is entered in the column “Name”. For the data type of this element, elementary as well as user-defined data types can be selected. An initial value can be optionally assigned in the column. This value is to be entered as numeric, string or time literal depending upon the data type. Information concerning literals can be found in section 7.1.3.



Example of a global data structure

```

TYPE
  Structure :
  STRUCT
    element1 : GlobalType;
    element2 : TIME := t#0:00:00;
  END_STRUCT;
END_TYPE

```

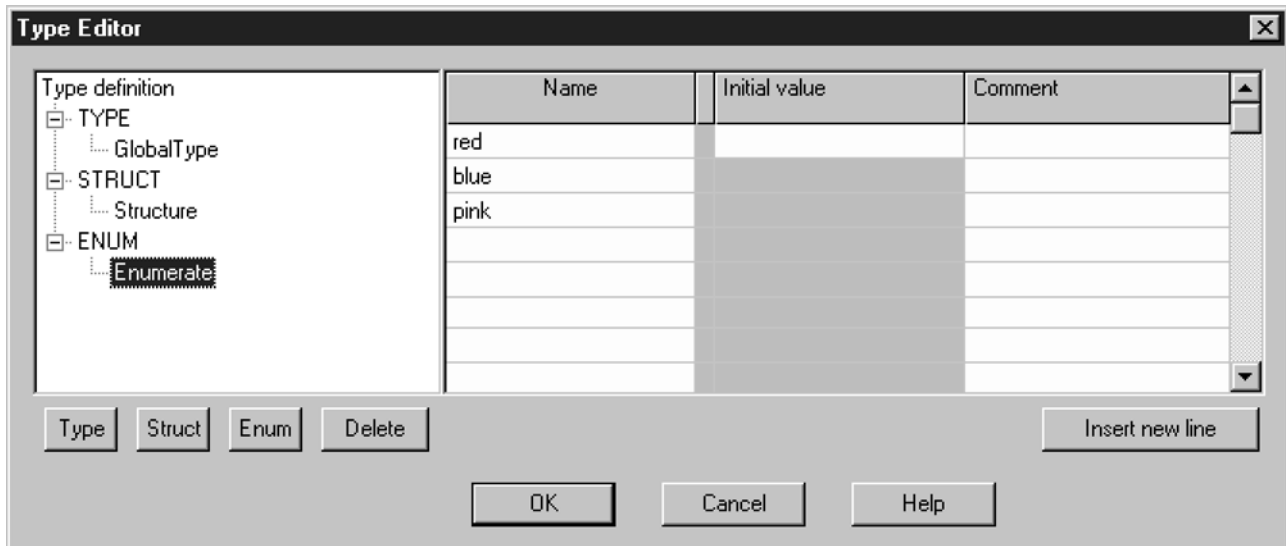
Textual presentation of the example

5.5.3 ENUM: Enumeration

By pressing the button “Enum”, a new enumeration can be set-up within the tree structure. Detailed information concerning enumerations can be found in section 7.2.2. The enumeration initially gets the name “New type”, which can be renamed by clicking on the name. The name must correspond to the identifier specifications of the IEC, refer to section 7.1.2.

The enumeration elements are assigned in the table. In case the rows are not sufficient, other rows can be inserted using the corresponding button.

Every enumeration element is entered in the column “Name”. The remaining columns have no significance.



Example of a global enumeration

```
TYPE
  Enumerate: (red, blue, pink);
END_TYPE
```

Textual presentation of the example

```
TYPE
  GlobalType : INT := 125;
  Enumerate: (red, blue, pink);
  Structure :
  STRUCT
    element1 : GlobalType;
    element2 : TIME := t#0:00:00;
  END_STRUCT;
END_TYPE
```


Summary of all the examples shown in this section

5.6 Constant definition – DEFINE Editor

Constants can be generated locally and globally with the help of the variable attribute “CONST”, also refer to section 7.3.8. In addition to this, global constants can be set-up with the “DEFINE editor”. This is called up within the symbol editor by pressing the button:



Every row in the define editor corresponds to a constant. The name must correspond to the identifier specifications of the IEC, refer to section 7.1.2. The value of the constant is assigned in the column initial value. Numeric or time literals can be entered. Information concerning literals can be found in section 7.1.3. The information concerning a data type is not required.

 **During the subsequent use of the constants in the program, attention must be paid to the data type compatibility.**

Name	Initial value	Comment
Const	15	any comment
dConst	D#2001-04-11	

OK Cancel Help

Model entries for constants in the define editor

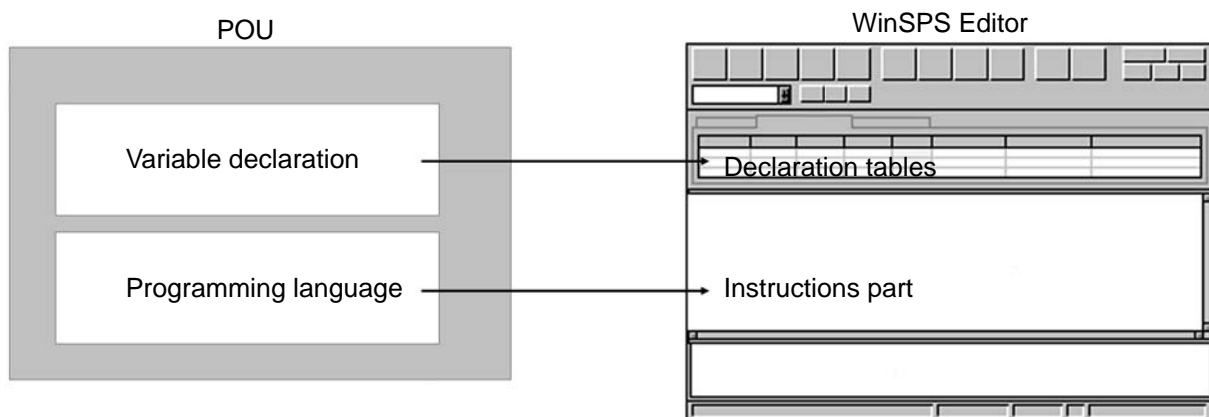
6 Program Structure

A great advantage of the IEC 61131-3 is the specification of common elements, which can be used globally for all the programming languages. The program structure, which is standardized with the design, contents and interfaces of various module types. The IEC terms the modules “Program Organization Units”, in short: POU.

6.1 Program Organization Units– modules of the IEC

In the IEC 61131-3, modules are termed as Program Organization Units (POU). It involves closed program units, which receive values from outside, process these values in the instructions part and supply results outwards.

A POU is divided in the declaration part for variables and in the instructions part of the respective programming language. In the editor of the WinSPS, this corresponds to the input areas “Declaration tables” and “Instructions part”, refer to the illustration. Further information concerning the use of a POU in the WinSPS Editor can be found in section 5.

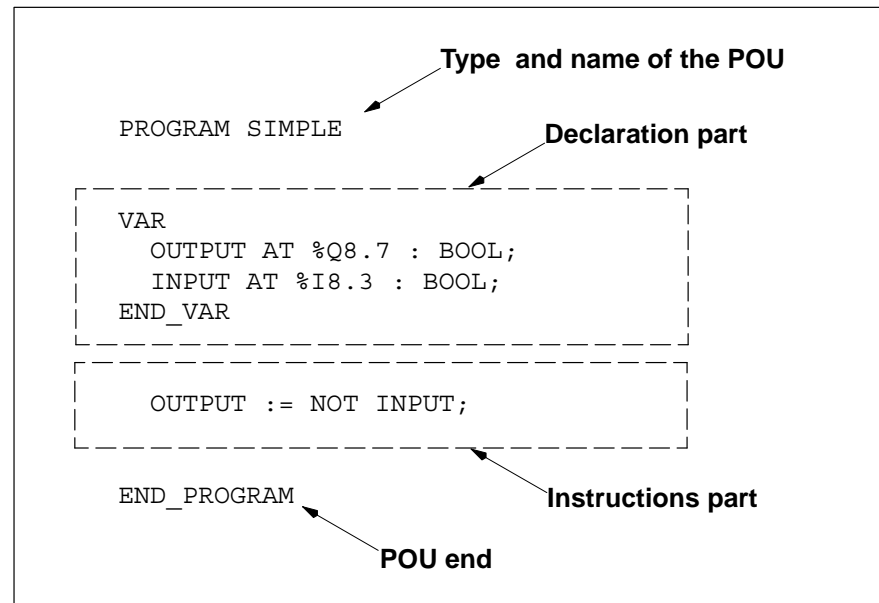


In the pure textual illustration without declaration tables, all elements of a POU are visible. The following elements form the structure of a POU:

- Data concerning the POU type and the POU name (in case of functions, additionally the data type of the function value)
- Declaration part
- Instructions part
- POU end

Example

The following example shows the structure of a POU. The programming language is basically not relevant for the structure of the POU, however, it is relevant for the input form of the program statements. In this example, the programming language ST was used:



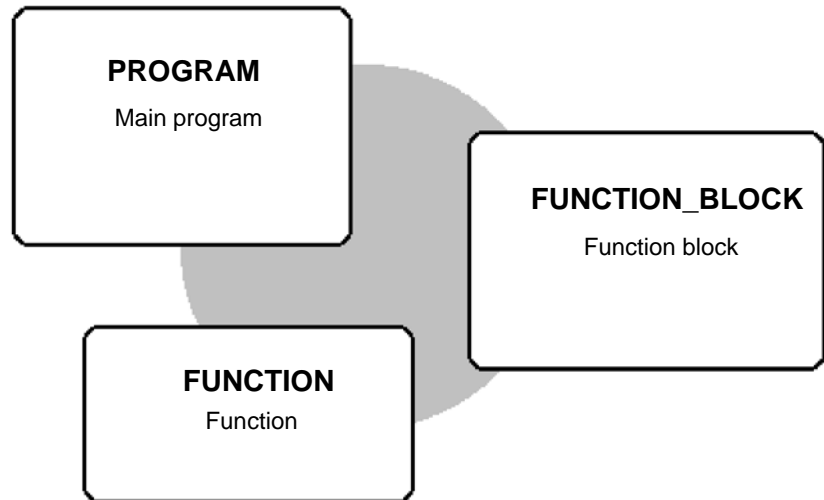
Structure of a POU

The **declaration part** begins after the specification of POU type and POU name and ends with the last declaration block.

The **instructions part** begins after the last declaration block – ending with the keyword **END_VAR**. The instructions part ends before the keyword **END_x**, where x corresponds to the POU type.

6.2 POU types

For the structuring of a PLC program, three POU types (module types) are available:



Program structuring


The structuring of the application program is achieved by dividing the task to be processed into sub-tasks. Each of these sub-tasks is placed in a POU. The POU's can be called from each other so that the calling POU is interrupted and the program processing continues in the called POU. After processing this POU, the program control returns to the calling POU and the processing is continued. Further information in this regard can be found in section 6.5.

Identification of a POU

The different POU types are marked by keywords at the beginning and end of the file. At the top of the file, the name of the POU is specified after the keyword. In case of a function, the data type of the function value is specified additionally:

POU type	Top Of File	End Of File
PROGRAM	PROGRAM Name	END_PROGRAM
FUNCTION_BLOCK	FUNCTION_BLOCK Name	END_FUNCTION_BLOCK
FUNCTION	FUNCTION Name : Type	END_FUNCTION

In the WinSPS Editor, these keywords are not visible so long as the “declaration tables” are activated, refer to section 5.1.

 **The short names PROG, FB and FUN mentioned below were selected only for this document and may not be used in place of the keywords while programming.**

6.2.1 Main program – PROGRAM

Key word: PROGRAM, short name: PROG

This module has exactly one occurrence for each central processing unit. Exception: Multitasking (not currently supported by Bosch). The main program runs through the cyclic processing in the PLC.

PLC peripherals and communication

In the main program, the assignment of the PLC peripherals is made. Global variables and access paths can be declared here. In the other POU types, the same is not possible.

The use of the variable type VAR_ACCESS for the global PLC communication is presently not possible.

Interface

The program may call up function blocks and functions.

Mixed programming

In case of mixed programming with classical programming languages, there may not be any PROGRAM POU. The main program is instead realized with the help of the organization module OM1, refer to section 11.2.

Entry in the symbol file

The symbol file is automatically managed by WinSPS during the generation of IEC programs. In case of mixed programming mentioned above, a few entries must however be made in the symbol file by hand, refer to section 11.4.

Example of a PROGRAM POU (programming language ST):

```
PROGRAM CTU_PROG (* POU type and name *)

VAR                                (* Declaration part *)
  Counter : CTU_FB;
  AT %Q4.0 : BOOL;
  AT %Q4.1 : BOOL;
  Set1 AT %I4.0 : BOOL;
  Set2 AT %I4.1 : BOOL;
  Res AT %I4.2 : BOOL;
END_VAR

                                (* Instructions part *)

  Counter (Set_1:=Set1, Set_2:=Set2, Res:=Res);
  %Q4.0 := Counter.C1_Max;
  %Q4.1 := NOT Counter.C1_Max;

END_PROGRAM (* POU end *)
```

POU type	Variable declaration	Type definition
POU name: <input type="text" value="SIMPLE"/>	FUN return type: <input type="text"/>	
<input checked="" type="radio"/> PROGRAM	<input type="radio"/> FUNCTION_BLOCK	<input type="radio"/> FUNCTION
<pre> Autor: Version: 1.00 Company: Bosch Rexroth AG Date: 21.08.2001 </pre>		

```

Counter (Set_1:=Set1, Set_2:=Set2, Res:=Res);
%Q4.0 := Counter.C1_Max;
%Q4.1 := NOT Counter.C1_Max;

```

Declaration and instructions of a PROGRAM POU in the WinSPS Editor, also refer to section 5.1.

6.2.2 Function block – FUNCTION_BLOCK

Key word: *FUNCTION_BLOCK*, short name: FB

Function blocks form an important structuring element for PLC programs due to numerous useful characteristics.

Interface

A function block may call up other function blocks and functions. Recursive calls are not allowed, refer to section 6.5.2.

A function block may not have any input parameter, or it may have one or more input parameters. Likewise, a function block may not have any output parameter, or it may have one or more output parameters. The access to this data is discussed in section 6.5.3.

While calling up a function block, not all input parameters need to be specified. The missing input parameters retain their values from the previous call or are initialized with initial values. The initial values are specified in the declaration part of the FB. If this data is missing, the standard values of the data type are used, refer to section 7.2.1.

Validity

Function blocks are locally valid. They must be declared in the calling POU. The declaration of FBs is also termed as instance building or instancing.

Instance building

From the function block, multiple instances can be built; these occupy independent memory locations for inputs, output and intermediate results. The function block itself is not called up, instead, always an instance of the function block is called up. The principle of the instance building is illustrated in detail in section 6.6.

Module with memory

Function blocks have a “memory”, i.e. the local variables of the module retain their (instance specific) value across various calls. This is an independent characteristic when for example, counter functionalities are to be built. Thus, through this instance building, multiple counters of the same type can be built. Every counter has its own memory. They do not influence each other.

Standard function blocks

In addition to the syntax of the programming languages, the IEC 61131-3 also standardizes important PLC functionalities. These are predefined in the norm as “Standard functions” or “Standard function blocks”. All manufacturers of programming systems or module libraries must follow these directions at the time of implementation.

The standard function blocks are listed in section 12.2.

In addition to this, manufacturers can offer other function blocks. Even the user can plan his own FBs, which can be used globally across the project (FB library).

Example of a function block (programming language ST):

```
FUNCTION_BLOCK CTU_FB (* POU type and name *)

VAR_INPUT                (* Declaration part *)
    Set_1 : BOOL;
    Set_2 : BOOL;
    Res : BOOL;
END_VAR

VAR_OUTPUT
    Cl_Max : BOOL;
END_VAR

VAR
    Counter_1 : CTU;
    Counter_2 : CTU;
    CV1 : INT;
END_VAR

(* Instructions part *)

Counter_1 (CU:=Set_1, RESET:=Res, PV:=20);
Counter_2 (CU:=Set_2, RESET:=Res, PV:=20);
Cl_Max := CTU_FUN (Counter_1.CV, Counter_2.CV);
CV1 := Counter_1.CV;

END_FUNCTION_BLOCK (* POU end *)
```


POU type	Variable declaration	Type definition
POU name:	<input type="text" value="CTU_FB"/>	FUN return type: <input type="text"/>
<input type="radio"/> PROGRAM	<input checked="" type="radio"/> FUNCTION_BLOCK	<input type="radio"/> FUNCTION
<pre> Autor: Version: 1.00 Company: Bosch Rexroth AG Date: 21.08.2001 </pre>		

```

Counter_1 (CU:=Set_1, RESET:=Res, PV:=20);
Counter_2 (CU:=Set_2, RESET:=Res, PV:=20);
Cl_Max := CTU_FUNCTION (Counter_1.CV, Counter_2.CV);
CV1 := Counter_1.CV;

```

Declaration and instructions of a FB POU in the WinSPS Editor, also refer to section 5.1.

6.2.3 Function – FUNCTION

Keyword: *FUNCTION*, short name: FUN

Functions are used to deliver an unambiguous result through by processing the input parameters. As a result, functions are used for recurring tasks such as mathematical functions.

Interface

A function may call up other functions. Recursive calls are not allowed, refer to section 6.5.2.

Functions may not have any input parameter, or they may have one or more input parameters and may return exactly one function value. The access to this data is discussed in section 6.5.3.

With every call to the function, all input parameters are to be specified.

Validity

Functions are globally valid. They are as a result, available for all POU's and therefore, they must not be declared in the calling POU.

Module without memory

A function may use local variables for intermediate results, they however do not retain their value across various calls. A function therefore has no 'memory', i.e. for the same inputs, it always supplies the same result.

Standard functions

In addition to the syntax of the programming languages, the IEC 61131-3 also standardizes important PLC functionalities. These are predefined in the norm as "Standard functions" or "Standard function blocks". All manufacturers of programming systems or module libraries must follow these directions at the time of implementation.

The standard functions are listed in section 12.1 . In this regard, also pay attention to the characteristics of “common data types” in section 7.2.3.

In addition to this, manufacturers can offer other functions. Users can plan their own functions, which can be used globally across the project (Functions library).

Example of a function (programming language ST):

```

FUNCTION CTU_FUN : BOOL (* POU type, name and
                        function value *)

VAR_INPUT              (* Declaration part *)
  CV_1 : INT := 0;
  CV_2 : INT := 0;
END_VAR

                        (* Instructions part *)

  CTU_FUN := CV_1 > CV_2;

END_FUNCTION          (* POU end *)

```

The screenshot shows the WinSPS Editor interface for declaring a function POU. The 'POU type' tab is selected, and the 'FUNCTION' radio button is chosen. The POU name is 'CTU_FUNCTION' and the return type is 'BOOL'. The declaration part contains the following text:

```

-----
Autor:                               Company: Bosch Rexroth AG
Version: 1.00                         Date: 21.08.2001
-----

```

Declaration of a function POU in the WinSPS Editor, also refer to section 5.1.

6.3 Declaration part

In the declaration part of a POU, variables are assigned to a specific data type (declared). Here, assignments related to physical addresses can be made and other characteristics can be fixed, also refer to section 5.1.

Moreover, in the declaration part of a POU, local, user-defined data types are fixed, refer to section 7.2.2.

The declaration part is located at the beginning of the POU. It is independent of the programming language used.

6.3.1 Variable types

Different types of variables characterize different application purposes of the variables. A differentiation is made between local variables and variables which are visible from outside (call interfaces, also refer to section 6.5.3).

Depending upon the variable type, variables are integrated into a block. Block beginning is formed by the keyword "VAR...". Every block ends with "END_VAR". The frequency of same variable types and their sequence can be anything. The WinSPS integrates scattered variables of the same variable type automatically into one block during inputs into the "declaration tables". The formation rules and names for variables can be found in section 7.3.

With the help of a few examples, the following list shows all possible types of variables:

Local variable

The local variable is valid only inside the POU in which it was declared. In this example, a physical address is assigned.

```
VAR
  Local AT %Q1.0 : BOOL;
END_VAR
```

Input variable

An input variable is declared when the variable is only to be read within a POU or when it is to be used for the parameter transfer to a function or a function block. As a result, the variable may not be changed in this POU. In the calling POU, the variable name is used as "formal parameter", refer to section 6.5.3.

In the example, there are two input variables. The formal parameters are "In1" and "In2". The variable "In1" moreover was assigned the "Initial value" 15, refer to section 6.5.3.

Also pay attention to the remark "Use of input and output parameters" in section 6.5.3.

```
VAR_INPUT
  In1 : INT := 15;
  In2 : DATE;
END_VAR
```

Output variable

In contrast to the input variables, the output variables for a function block may be changed in the POU. In the calling POU, it may only be read.

The example moreover shows the use of the variable attribute "RETAIN", refer to section 7.3.8.

Also pay attention to the remark "Use of input and output parameters" in section 6.5.3.

```
VAR_OUTPUT RETAIN
  Out : WORD
END_VAR
```

Input and output variable

An IN_OUT variable is read by the FB, processed and outputted under the same name. With it, the characteristics of the VAR_INPUT and the VAR_OUTPUT variables can be combined.

Also pay attention to the remark "Use of input and output parameters" in section 6.5.3.

```
VAR_IN_OUT
  InOut : LREAL
END_VAR
```



DANGER

VAR_IN_OUT cannot be assigned an initial value. The initialization can take place with the specification of the corresponding formal parameters instead of in the FB calls.

In case initial values are specified within VAR_IN_OUT, WinSPS does not generate any error message during compilation or linking. The initial values are however not set in the controller!

Global variable

A variable is declared as "global variable" if it is valid for a program and in all function blocks which are called up from this program. A global variable declared in the program is known inside the program as well as inside the FBs, which can be called up from this program. In all the called up FBs, in which this global variable is to be used, it must be declared as VAR_EXTERNAL with the same identifier (name).

```
VAR_GLOBAL
  Global : WORD
END_VAR
```

External variable

If a global variable is to be used inside a function block, it must be declared as "VAR_EXTERNAL" with the same identifier (name) and data type as under VAR_GLOBAL.

In functions, global variables cannot be used.

```
VAR_EXTERNAL
  Global : WORD
END_VAR
```

Access paths

Access paths across the project.

 **The variable type VAR_ACCESS is presently not supported by Bosch controller!**

```
VAR_ACCESS
  Path : CPU_1.Simple.Param_1 : WORD
END_VAR
```

POU type		Variable declaration		Type definition			
Variable type	Name	Data type	Initial value	Address	Attribute	Monitor data	Comment
VAR	Local	BOOL		%Q1.0			
VAR_INPUT	In1	INT	15				
VAR_INPUT	In2	DATE					
VAR_OUTPUT	Out	WORD			RETAIN		
VAR_IN_OUT	InOut	LREAL					
VAR_GLOBAL	Global	WORD					
VAR_EXTERNAL	Global	WORD					

All above-mentioned examples are summarized in the declaration tables of the WinSPS Editor (illustrative), also refer to section 5.1.

6.3.2 Applicability and access options of the variable types

Depending upon the POU type, different types of variables are allowed:

Variable type	Explanation	PROG	FB	FUN
VAR	Local variable within the POU	Yes	Yes	Yes
VAR_INPUT	Input variable	Yes	Yes	Yes
VAR_OUTPUT	Output variable	Yes	Yes	–
VAR_IN_OUT	Input and output variable	Yes	Yes	–
VAR_EXTERNAL	Global variable, which is declared in another POU	Yes	Yes	–
VAR_GLOBAL	Global variable, which is declared in this POU.	Yes	–	–
VAR_ACCESS*	Access paths	Yes	–	–

* The variable type VAR_ACCESS is presently not supported

The following examples should illustrate the access options of variables. An external access implies processing inside the calling POU. The internal access characterizes the behavior inside the POU, in which the variable was declared.

Variable type	External access	Internal access
VAR	–	writing / reading
VAR_INPUT	writing	reading
VAR_OUTPUT	reading	writing / reading
VAR_IN_OUT	writing / reading	writing / reading
VAR_EXTERNAL	writing / reading	writing / reading
VAR_GLOBAL	writing / reading	writing / reading
VAR_ACCESS*	writing / reading	writing / reading

* The variable type VAR_ACCESS is presently not supported!

It is evident from the table that only the variable type VAR does not offer any option for the external access. All other types of variables are suitable for the data exchange between POU's.

The variable types VAR_INPUT and VAR_OUTPUT provide a mechanism that prevents undesired manipulation.

 **While using input and output parameters, pay attention to the instructions under section 6.5.3.**

6.4 Instructions part

The declaration part is followed by the instructions part (body) of a POU. In the instructions part, the PLC instructions to be executed are entered in the respective programming language.

The description of the programming in the instructions part follows in the chapters on the programming languages:

- Instruction list (IL), chapter 8
- Structured Text (ST), chapter 9

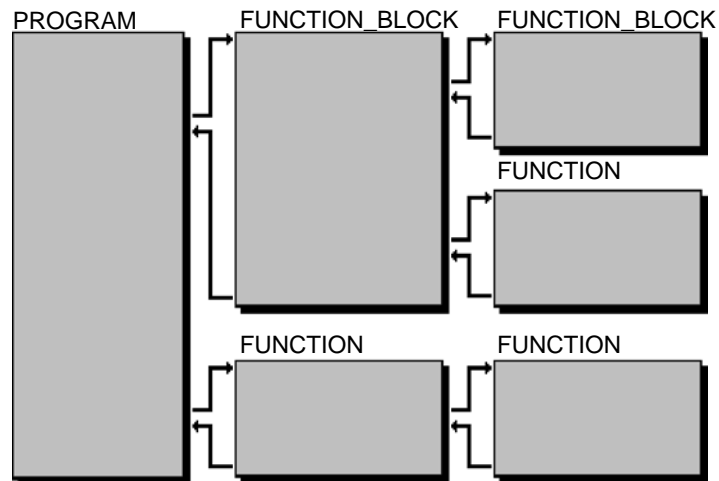
6.5 Calls between POU's

An application program can be structured using the call option of POU's. The syntax of the calls is different among the programming languages. Detailed information in this regard can be obtained in the related sections of the respective programming language.

6.5.1 Call hierarchy

The following rules apply for the mutual calls to the program organization units (POU), also refer to the illustration below:

- PROGRAM may call up FUNCTION_BLOCK and FUNCTION.
- FUNCTION_BLOCK may call up FUNCTION or FUNCTION_BLOCK.
- FUNCTION may call up only FUNCTION.



Calling up options of the individual POU types

6.5.2 Recursive calls



DANGER

Recursive calls (calling up self) are not allowed. Even indirect recursive calls are not allowed! Recursive calls can be detected through the checking by WinSPS only in a limited way.

Through recursive calls, the maximum PLC cycle time is exceeded and the controller are stopped!

Example of a recursive call (ST):

```

FUNCTION recursive : INT

VAR_INPUT
    Par01, Par02 : INT;
END_VAR

    Recursive := Par01 + Recursive (Par01, Par02);

END_FUNCTION

```

The example shows the function "Recursive". Inside this function, the same function is again called up. This leads to an endless loop of function calls. The program does not respond any more and cannot be interrupted. It leads to controller stoppage since the program cycle time is exceeded.

Example of an indirect recursive call (ST):

```

FUNCTION Fun_1 : INT

VAR_INPUT
  Par01, Par02 : INT;
END_VAR

  Fun_1 := Par01 + Fun_2 (Par01, Par02);

END_FUNCTION

FUNCTION Fun_2 : INT

VAR_INPUT
  Par01, Par02 : INT;
END_VAR

  Fun_2 := Par01 + Fun_1 (Par01, Par02);

END_FUNCTION

```

The example shows two functions, which when considered individually – do not contain any recursion. The functions however call each other. “Fun_1” calls up “Fun_2”, “Fun_2” again calls up “Fun_1”. This results in a recursive chain of calls – an indirect recursion.

6.5.3 Call interface – parameters during the call

Input parameter

Parameters can be transferred to functions and function blocks at the time of calling up. These parameters can be used for internal processing inside the calling function or the function block. There, they are termed as input variables. In the calling POU, they are termed as input parameters or actual parameters.

Output parameter

In case of function blocks, there is an option of transferring the return values to the calling POU through output parameters. It is fixed in the declaration part of the FB, which input and output variables should the function block have.

In case of functions, only input variables can be declared. Functions supply only one output parameter, the so-called **function value** or **return value**.

Formal parameter

At the time of calling up a function or a function block, the variable names of the respective input parameter may/must be specified. This name is also referred to as formal parameter. Rules for the use of formal parameters are discussed in section 6.5.4 and 6.5.5.

Initial values

Initial values can be specified in the declaration part of the POU to be called up. If this data is missing, the standard values of the data type are used, refer to section 7.2.1. Further information concerning the initialisation can be found in sections 6.5.4 and 6.5.5.

Variable types of the parameters

The following table shows, for which purpose can various variable types be used. A detailed enumeration of all variable types can be found in section 6.3.1.

	Variable type	Comment
Call interface	VAR_INPUT VAR_IN_OUT	Input parameter
Return values	VAR_OUTPUT VAR_IN_OUT Function value in case of FUNCTION	Output parameter
Global interface	VAR_GLOBAL VAR_EXTERNAL VAR_ACCESS	Global data and access paths
Local data	VAR	Only internal POU: no interchange possible

Use of input and output parameters:

For the variable types of the call interface and the return values, there are different access methods. Pay attention to the following characteristics:

VAR_INPUT

Transfer of not the input variable itself, instead, transfer of a copy (memory) to the POU. This principle is termed as "call by value". This results in the protection from undesired manipulation within the called POU.

VAR_OUTPUT

In the calling POU, the data can only be read as the output parameters are generated only in the called POU (return by value). Output parameters are protected from being changed in the calling POU.

VAR_IN_OUT

In case of this variable type, there is no access protection, as here only one pointer is transferred to a memory location (call by reference). This principle certainly does not provide any protection, but it can be ideally used during the transfer of complex data structures. The program processing is very efficient as the variables do not have to be copied during runtime.



DANGER

VAR_IN_OUT can not be preset to an initial value. The initialization can instead take place in the FB call with specification of the corresponding formal parameter.

In case initial values are specified within the declaration of VAR_IN_OUT, WinSPS does not generate any error message during compilation or linking. The initial values are however not set in the controller!

Example

The following example shows a function block and subsequently a call option in the programming language ST:

```

FUNCTION_BLOCK FB1
VAR_IN
  Par_1 : BOOL;
  Par_2 : BOOL;
END_VAR

VAR_OUT
  Out : INT := 0;
END_VAR

  IF Par_1 = Par_2 THEN
    Out := Out + 1;
  END_IF;

END_FUNCTION_BLOCK

```

The function block “FB1” has two **input parameters** with the names “Par_1” and “Par_2” of the data type “BOOL”, and the **output parameter** “Out” of data type “INT”.

```

PROGRAM ABC
VAR
  AT %Q2.0      : BOOL;
  X1 AT %I2.0   : BOOL;
  Counter       : INT := 0;
  FB1_Inst      : FB1;
END_VAR

  FB1_Inst (Par_1:=%Q2.0, Par_2:=X1 | Counter := Out);

END_PROGRAM

```

The program “ABC” shows a call option of the function block “FB1”. The local variables “%Q2.0” and “X1” are used as input parameters, the output parameter is assigned to the variable “Counter”.

The assignment takes place in the instance call with the specification of the formal parameters “Par_1”, “Par_2” and “Out”.

The data types of the local variables and the FB internal variables must match at the transfer interface.

6.5.4 Calling up the function blocks

Call

The call to a function block takes place not through the FB name itself, instead through the instance name. This is determined by the "Instance building", refer to section 6.6.

Parameter transfer


The transfer of input data and the evaluation of output data differ depending upon the programming language. In the programming language IL and ST, input data is "given" with the call, in brackets and delimited by commas. The **actual parameters** are assigned to the **formal parameters** using ":=". Output parameters can similarly be assigned to the formal parameters in the call. Input and output parameters are then separated from each other with the character "|", refer to the example below.

The second method for the parameter transfer is the initialization before the call. In IL, the parameters are initialized through a combination of loading (LD) and assignment operators (ST). In ST, the assignment operator is used, refer to the examples below.

The different methods are illustrated in detail in the corresponding sections of the respective programming language.


Sequence of parameters

Since for every assignment of the input parameters, the specification of the formal parameters is required, the sequence of the parameters in case of FB calls is not relevant.

 **The names of the formal parameters for standard function blocks are specified in section 12.2 and in the WinSPS help, section "Programming in compliance with IEC 61131-3, Standard Function Blocks".**

Leaving out and initializing the parameters

While calling up a function block, not necessary parameters may be left out. The missing input parameters retain their values from the previous call or are preset with initial values. The initialization takes place only at the time of first call of an FB instance. Afterwards, always the values of the previous call are used. This characteristic is termed as "Module with memory".

 **VAR_IN_OUT parameters may not be left out. During compilation, WinSPS generates an error message in case VAR_IN_OUT parameters are left out.**

The initial values are specified in the declaration part of the FB. If this data is missing, the standard values of the data type are used, refer to section 7.2.1.

**DANGER**

VAR_IN_OUT can not be preset to an initial value. The initialization can instead take place in the FB call with specification of the corresponding formal parameter.

In case initial values are specified within the declaration of VAR_IN_OUT, WinSPS does not generate any error message during compilation or linking. The initial values are however not set in the controller!

Write input variables and read output variables

The access to the input and output data takes place in case of a function block with the FB instance name, followed by a dot "." as delimiter for the variable following it. If the output variables within the FBs are not assigned values, they receive the default values of their data type, refer to section 7.2.1.

Examples

FB Call in the programming language IL, method 1:

```
CAL CTU_Instance (RESET:=%IX3.6, PV:=Limit, CU:=_1S2)
```

Method 2:

```
LD %IX3.6
ST CTU_Instance.RESET
LD Limit
ST CTU_Instance.PV
LD _1S2
ST CTU_Instance.CU
CAL CTU_Instance
```

Access to output data in IL:

```
LD CTU_Instance.CV
ST Result
```

FB Call method 1, with assigning of the output parameter:

```
CAL CTU_Instance (RESET:=X1, PV:=100, CU:=X5 |
                Result:=CV)
```

FB Calls in the programming language ST, method 1:

```
CTU_Instance1 (RESET:=%IX3.6, CU:=_1S2, PV:=Limit);
CTU_Instance2 (CU:=_1S5, PV:=Limit, RESET:=%IX7.4 );
```

Method 2:

```
CTU_Instance1.RESET := %IX3.6;
CTU_Instance1.CU := _1S2;
CTU_Instance1.PV := Limit;
CTU_Instance1 ();
```

Access to output data in ST:

```
Result := CTU_Instance.CV;
```

6.5.5 Calling up the functions

Call


A function is called directly using the function name. Functions are always known and can be called up across the project without declaration or instance building.

Parameter Transfer

In the programming language ST, the **formal parameters** can be specified or left out during the call. The **actual parameters** are assigned to the formal parameters using “:=”, refer to the example below.

In the programming language IL, the formal parameters are basically not specified during the call. Here, the sequence of the input parameters must definitely be followed.

As the methods of parameter transfer are quite different in different programming languages, they are explained in detail in the related sections of the respective programming languages.

 **In case of standard functions in the context of the common data types, the description of the input variables is dispensed with. While calling such functions, no formal parameters can be specified. The standard functions are listed in section 12.1 . Information concerning common data types can be found in section 7.2.3.**

Leaving out and initializing the parameters

While calling a function, input parameters may be left out. The missing parameters are preset to the initial values. The initial values are specified in the declaration part of the function. If this data is missing, the standard values of the data type are used, refer to section 7.2.1.

Sequence of parameters

When formal parameters (variable names) are left out – in the programming language IL and in case of standard function, they must in fact be left out – the sequence of the parameter is to be followed. It results from the sequence of the declaration in the module to be called up. Refer to the example below.

If formal parameters are used for all the input parameters (programming language ST), the sequence of the parameters is not relevant.

If input parameters are left out, the initial values are used.

Write input variables and read function value

The input parameters are transferred directly with the call. In case of programming language IL, there is an option of transferring the initial input parameters from the **Current Result** (CR) of the earlier executed instruction without further specifications. Further information concerning this procedure can be found in sections 8.3 and 8.5.8.

Functions supply a function value (return value) as output parameter. This can be assigned directly, or can be used for further processing. If the function value within the function is not described with a value, it gets the default value of its data type, refer to section 7.2.1.

Examples

Function call and assignment of the function value in the programming language IL:

```
SHL  2#10010110, 2
ST   Shift_Left
```

Use of the AR for the first parameter in IL:

```
LD   2#10010110
SHL  2
ST   Shift_Left
```

Function call in ST with formal parameters:

```
Shift_Left := SHL (N:=2, IN:=2#10010110);
```

Without formal parameter. The sequence must be followed:

```
Shift_Right := SHR (2#01101100, 4);
```

Function call “CONCAT” and further processing through second function call “LEN”:

```
iLength := LEN ( CONCAT ('To', 'gether'));
```

6.6 Instance building of function blocks

Due to the principle of instance building (instancing), function blocks are considerably different from the other POU types.

FBs may not be called up directly using their (type)names. Instead, an instance of the FB is called up. The instance is built in the declaration part of the calling POU or as a global instance in another POU (refer to section 6.6.1). The instance building can be compared to the assignment of a variable to a data type, also refer to section 7.3.1. The declaration of a variable is the same as the instance building of a data type.

Example:

```

VAR
  Valve_1 : BOOL;           (* Boolean variable=Instance of the data type BOOL *)
  Valve_2 : BOOL;           (* Other instance of the data type BOOL *)
  Variable_name : Data_type; (* common form *)

  CTU_Instance1 : CTU;      (* Instance of the function block CTU *)
  CTU_Instance2 : CTU;      (* Other instance of the FB CTU *)
  FB_Instancename : FB_type; (* common form *)
END_VAR
    
```

The example illustrates the principle of instance building. The data type BOOL can (obviously) be used multiple times by building the instances of the data type through the declaration of variables – in this case, “Valve_1” and “Valve_2”.

In the lower part of the example, similar procedure is followed. With the data type BOOL, the function block “CTU” can be used multiple times. CTU is a standard function block, refer to section 12.2, using which a counter function can be built. With the declaration (building) of two instance of this module, two counter functions “CTU_Instance1” and “CTU_Instance2”, which are independent of each other, can be realized.


In the instructions part, the function block is called up using the instance name, also refer to section 6.5.4.

POU type		Variable declaration		Type definition			
Variable type	Name	Data type	Initial value	Address	Attribute	Monitor data	Comment
VAR	Counter_1	CTU					
VAR	Counter_2	CTU					

For instance building of an FB in the declaration tables of the WinSPS Editor, also refer to section 5.1.

6.6.1 Validity of function blocks

With the instance building, the validity of the function block is determined. If the instance building takes place inside the variable type VAR, the instance can be used only inside this POU. With the combination VAR_GLOBAL and VAR_EXTERNAL, an FB instance can be used in all modules.


 **In contrast to this, functions are always known and can be called up across the project without declaration or instance building.**

6.6.2 Module with “memory”

For each instance of a function block, an independent copy (memory) of the data is created. With this mechanism, data can be stored over multiple calls (memory). This is necessary in order to e.g. realize timers, counters or Flip Flops.

The used variable type in the declaration part of the FB decides, which data is copied in the memory of the FB:

- Local variables (VAR) and input / output variables (VAR_INPUT, VAR_OUTPUT) are copied fully.
- Pointer to variables in VAR_IN_OUT are stored, however, not the values themselves. When for example, a data structure is transferred via VAR_IN_OUT to the FB, the data elements of the structure are not stored.
- Global variables are basically not copied (VAR_EXTERNAL).

 **Since for every instance, a data copy is assigned in the memory of the PLC, care must be taken to keep this data area as small as possible. With the use of VAR_IN_OUT for example, the pointer can be transferred to a complex structure in order to save space.**

6.6.3 Instance building for combination with “classical” programming languages

The instance building is also used to allow the combination of modules as per IEC 61131-3 (POU) with modules of the “classical” programming languages such as Bosch-IL. For this, the WinSPS provides special mechanisms. Detailed information concerning this can be found in chapter 11.

7 Data model

Independent of the programming language used, the data model describes all common language elements. As a result, it presents the common elements of the IEC 61131-3.

7.1 Language elements

In the IEC, **no** distinction is made between UPPERCASE and lowercase letters. However, in order to make better distinction, the style shown in the example given below must be followed.

As the IEC has an international application, country-specific umlauts such as ä, é, ñ, etc. are not allowed. However in future, umlauts should also be allowed in the supplements to the standard. Umlauts are fundamentally allowed in comments, refer to section 7.1.5.

As language elements, the IEC uses so-called

- Key words
- Identifiers
- Literals and
- Delimiters.

A POU is made up of these language elements.

Language element	Explanation	Examples
Key word	Terms specified by IEC 61131-3	VAR_IN_OUT, CAL, FUNCTION_BLOCK
Identifiers	User defined terms for names of variables, POUs, etc.	Radius, Counter_1, _1S15, CTU_FUN
Literal	Representation of the values of various data types	3.1416, 8.7266E-3, 'Text'
Delimiter	Different meaning	(.),+,-,*,:=,;,#,\$,comma, space, carriage return (CR)

Example using a function in the programming language ST:

```

FUNCTION Circumference : REAL
VAR_INPUT
  Radius : REAL;
END_VAR
  Circumference := Radius * 2 * 3.1416; (* comment *)
END_FUNCTION

```

The example shows

key words: **printed in bold**

Identifiers: normal

Literals: *in italics*

Delimiter: Colon, star, equal, semicolon, parenthesis and space.

7.1.1 Key words

Key words are specified and reserved by the IEC. They cannot be used for other purposes e.g. for variables. If by mistake a key word is used for user-defined names, the compiler gives out an error message. For better understanding, keywords should be written in uppercase.

The following list shows all reserved key words in alphabetical order:

ABS, ACOS, ACTION, ADD, AND, ANDN, ANY, ANY_BIT, ANY_DATE, ANY_INT, ANY_NUM, ANY_REAL, ARRAY, ASIN, AT, ATAIN,

BOOL, BY, BYTE,

CAL, CALC, CALCN, CASE, CD, CDT, CLK, CONCAT, CONFIGURATION, CONSTANT, COS, CTD, CTU, CTUD, CU, CV,

D, DATE, DATE_AND_TIME, DELETE, DINT, DIV, DO, DS, DT, DWORD,

ELSE, ELSIF, END_ACTION, END_CASE, END_CONFIGURATION, END_FOR, END_FUNCTION, END_FUNCTION_BLOCK, END_IF, END_PROGRAM, END_REPEAT, END_RESOURCE, END_STEP, END_STRUCT, END_TRANSITION, END_TYPE, END_VAR, END_WHILE, EN, ENO, EQ, ET, EXIT, EXP, EXPT,

FALSE, F_EDGE, F_TRIG, FIND, FOR, FROM, FUNCTION, FUNCTION_BLOCK,

GE, GT,

IF, IN, INITIAL_STEP, INSERT, INT, INTERVAL,

JMP, JMPC, JMPCN,

L, LD, LDN, LE, LEFT, LEN, LIMIT, LINT, LN, *LOAD**, LOG, LREAL, LT, LWORD,

MAX, MID, MIN, MOD, MOVE, MUL, MUX,

N, NE, NEG, NOT,

OF, ON, OR, ORN,

P, PRIORITY, PROGRAM, PT, PV,

Q, Q1, QU, QD,

R, RI, R_TRIG, READ_ONLY, READ_WRITE, REAL, RELEASE, REPEAT, REPLACE, *RESET**, RESOURCE, RET, RETAIN, RETC, RETCN, RETURN, RIGHT, ROL, ROR, RS, RTC, R_EDGE,

S, S1, SD, SEL, SEMA, *SET**, SHL, SHR, SIN, SINGLE, SINT, SL, SQRT, SR, ST, STEP, STN, STRING, STRUCT, SUB,


TAN, TASK, THEN, TIME, TIME_OF_DAY, TO, TOD, TOF, TON, TP, TRANSITION, TRUE, TYPE,

UDINT, UINT, ULINT, UNTIL, USINT,

VAR, VAR_ACCESS, VAR_EXTERNAL, VAR_GLOBAL, VAR_IN_OUT, VAR_INPUT, VAR_OUTPUT,

WHILE, WITH, WORD,

XOR, XORN.

 *** The key words R, S and LD of the input parameters are used in the programming language instruction list (IL) with another meaning. Due to this conflict, there are difficulties during translation by a compiler. This problem was taken up in the working group for further development of the IEC 61131-3. In a revised version of the standard, the parameter names should be changed to SET, RESET and LOAD. WinSPS already uses this modified form.**

7.1.2 Identifiers

In comparison to keywords, identifiers are alphanumeric strings which are used by the user for variable names, POU names, constants etc. . The format of an identifier is fixed by the IEC.

An identifier is a series of letters, numbers and understroke character (_) which must start either with a letter or an understroke character.

The understroke is significant in the identifiers i.e. AB_C is different from A_BC.

Multiple understrokes following directly one after the other are not allowed.

 **With the activation of WinSPS function switch /O2, the checking is turned off so that multiple understrokes may be allowed to appear in the symbolic identifiers.**

As there is no difference between uppercase and lowercase letters, the identifier

“Hopper_Empty” is identical to “HOPPER_EMPTY” or “hopper_empty”.

An identifier may be maximum 32 characters long.

Examples:

Right:	Wrong:
VOLTAGE_REGULATOR	VOLTAGE__REGULATOR
_Turningdirection_plus	_Turningdirection_+
Marker_2	2_Marker
_1S15	1S15
Input_0_7	Input 0 7
Output_W4	Ouput_%W4

7.1.3 Literals

Depending upon the data type, three different types of literals are available:

- Numeric literals
- String literals
- Time literals

Numeric literals

Numeric literals can be used for numbers as bit sequence as well as for integers and floating point numbers. The following table shows the classification of the numeric literals.

Characteristic	Examples
Boolean data	TRUE (not equal to 0) FALSE (0)
Integer literals	-12 0 123_456 +986
Binary literals	2#11111111 (255 decimal) 2#11100000 (240 decimal) 2#1111_1111_1111_1111 (65535 decimal)
Octal literals	8#377 (255 decimal) 8#340 (240 decimal)
Hexadecimal literals	16#FF or 16#ff (255 decimal) 16#1000 (4096 decimal)
Floating point	+523.31 -0.08 398E+4 (Exponent) 4e2 -34E-15

Boolean data is represented by the keywords *FALSE* and *TRUE*. Likewise, in case of physical addresses, one is allowed use of *0* in place of *FALSE* and *1* in place of *TRUE*.

Decimal literals are represented in traditional style. They can have a leading sign (+ or –). Base-specific numbers may not have any sign.

The letters A to F for the hexadecimal numbers 10 to 15 can be written in uppercase or lowercase.

Understroke characters introduced for better readability are allowed and are not significant. Leading understroke characters are not allowed.

Floating point numbers can be written with a decimal "." as punctuation mark, or can be written in exponential form. Even in this case, leading signs are allowed.

String literal

A string literal (earlier ASCII constant) is a sequence of zero or more characters which are enclosed by single quotation marks ('').

The maximum length is manufacturer-dependent and for the Bosch controller, it's value is presently 63 characters.

The string terminator '\0' is automatically added.

Displayable characters out of the standard ASCII character set:
Hexadecimal 20 to 7F.

Characters outside this range can be represented with a leading \$ sign (e.g. \$80).

In addition, the IEC provides for special characters which begin with the dollar sign. With this, characters can be formatted for outputting on display units or printers.

Examples:

Character combination	Explanation
\$\$	Single dollar sign
\$'	Single quotation mark
\$R or \$r	Carriage return (CR = \$0D)
\$L or \$l	Linefeed (LF = \$0A)
\$P or \$p	Formfeed (FF = \$0C)
\$N or \$n	New row
\$T or \$t	Tab

 In addition to IEC 61131-3, Bosch allows direct use of umlauts.

Examples for string literals:

```
'ABC_123'          (* Output: ABC_123 *)
''                (* Empty string *)
' '              (* String of length 1,
                with space *)
'$''            (* Output: ' *)
'$R$L$0D$0A'    (* String of length 4,
                each with 2 CR and LF *)
'$ $1.00'       (* Output: $1.00 *)
'$9Aberd$84mpfung' (* Output: Überdämpfung *)
'Smörgåsbord'   (* allowed by Bosch ! *)
```

Time literals

With the time literals, values for time duration, time and date are formed.

The IEC allows a short form for time literals.

Time type	Examples	Short form
Time duration	TIME#1h_15m_30s_60ms TIME#14ms	T#
Time of day	TIME_OF_DAY#11:36:15.20	TOD#
Calendar date	DATE#2001-04-09	D#
Date and time	DATE_AND_TIME#2001-04-09-11:36:15.20	DT#

The type time duration is used for processing a relative time. The other types in comparison to this are used for absolute day of time and date.

Understroke characters introduced for better readability are allowed and are not significant.

In case of time duration, positions, which are not relevant, may be left out. A time duration may “overflow”.

Example of non-relevant positions and an overflow:

T#80m_15s corresponds to T#1h_20m_15s (overflow) and may be written both ways. As a granulation in milliseconds is not necessary in this example, the corresponding positions may be left out.

7.1.4 Delimiter

Delimiters are special characters that are used for different purposes. Thus, colon is for example used not only for the time within the literal "time of day" but also for defining a data type name.

The following table contains all delimited characters.

Name / Function	Character
Space	
End Of Line (CR+LF, EOL : end of line)	
Beginning of a comment	(*
End of a comment	*)
Plus	+
Minus	-
Multiplication	*
Division	/
Exponent as operator	**
Exponent in case of floating point literal	e
Hash sign	#
Dollar sign	\$
Parenthesis	()
Square brackets	[]
Percentage sign	%
Ampersand sign	&
Quotation marks	'
Comma	,
Semicolon	;
Decimal sign	.
Dot dot	..
Colon	:
Assignment symbol	:=
Assignment symbol for PROGRAM call	=>
Prefix time literals	T#, D#, TOD#, DT# TIME#, DATE#, TIME_OF_DAY#, DATE_AND_TIME#, D, H, M, S, MS

7.1.5 Comments

In addition to the said language elements, there is an option to add comments which are excluded from the program processing.

Comments must be enclosed at the beginning and end with special character combination (* or *).

Nested comments are not allowed.

Comments may be added at those positions where spaces are allowed.

In case of programming language instruction list (IL), comments may be used within an instruction row only at the end of the line.

Within comments, all characters out of the ANSI character set, including umlauts, are allowed.

 **The comment marking semicolon “;” used in the classical programming is not allowed.**

Examples:


```
(* IEC compatible comment marking *)
LD %IB4      (* IL comment *)
```

Incorrect examples:

```
(* (* nested comments are not allowed *) *)
LD (* not allowed in case of IL! *) %IB4
```

Input of comments through WinSPS

In the declaration tables of the WinSPS editor, the comments can be entered in a column meant for the purpose. The comment is given at the end of the respective line end. No comment markings “(* *)” must be entered, these are automatically added by WinSPS, refer to the illustration.

POU type							
Variable declaration		Type definition					
Variable type	Name	Data type	Initial value	Address	Attribute	Monitor data	Comment
VAR	Local	BOOL		%Q1.0			any comment
VAR_INPUT	In1	INT	15				
VAR_INPUT	In2	DATE					

Comments input: The comment markings are automatically set by WinSPS.

7.2 Data types

The characteristic of a variable is determined from the data type. For example, for time operations other data types are required in comparison to the data types for floating point arithmetic.

In the IEC, the important data types have been standardized. They are referred to as “elementary data types” .

In addition to this, there is an option of defining the user-specific data type. These are called “derived data types” or also “type definitions”.

Another data type of the IEC is used for standard functions. These are “generic data types”.

7.2.1 Elementary data types

Elementary data types are standardized in the IEC 61131-3 and are predefined by keywords (e.g. BOOL, SINT, etc.).

Data types are used for the declaration of the variables, refer to section 7.3.

The initialization value of a data type is fixed in the declaration part using the assignment operator “:=”. If no initialization value is assigned, the standard value defined by the IEC is assumed to be the default value, refer to the following table: Default value.

The following table shows all usable data types. In addition to this, the IEC specifies the data types LINT, ULINT and LWORD, which are not supported by the Bosch controller.

Key word	Data width (Bit)	Default value	Explanation
BOOL	1	FALSE	Logical value (not equal to 0 => TRUE, equal to 0 => FALSE). The values 0 = FALSE and 1 = TRUE may be used (manufacturer-specific).
BYTE	8	0	Bit sequence of length 8 (0 to 255).
WORD	16	0	Bit sequence of length 16 (0 to 65535).
DWORD	32	0	Bit sequence of length 32 (0 to 4,294,967,295).
SINT	8	0	Short integer, figure with a sign (-128 to +127).
INT	16	0	Integer, figure with a sign (-32768 to +32767).
DINT	32	0	Double integer, figure with a sign (-2,147,483,648 to +2,147,483,647).
USINT	8	0	Unsigned short integer, figure without a sign with an adjustable figure base in the value range (0 to 255).
UINT	16	0	Unsigned short integer, figure without a sign with an adjustable figure base in the value range (0 to 65,535).
UDINT	32	0	Unsigned double integer, figure without a sign with an adjustable number base in the value range (0 to 4,294,967,295).
REAL	32	0.0	Floating point number = Floating Point (Double word) (1,175494351e-38 to 3,402823466e+38)
LREAL	64	0.0	Floating point number = Floating Point (Quadruple word) (2,2250738585072014e-308 to 1,7976931348623158e+308)
TIME	32*	T#0s	Time duration Input in hours (h), minutes (m), seconds (s), milliseconds (ms)
DATE	32*	D#1900-01-01	Calendar date. Format: Year-Month-Day
TIME_OF_DAY	32*	TOD#00:00:00	Time. Format: Hours:Minutes:Seconds (24 Hours time format)
DATE_AND_TIME	64*	DT#1900-01-01-00:00:00	Date and time
STRING	Variable, max. 63 Character *	" (Empty string = '\$00')	ASCII string. Representation: Hexadecimal 20 to 7F; characters outside this range with leading \$ sign (e.g. \$80). The character string is enclosed on the left and the right side by a quotation mark (e.g. 'Test'). Special control characters can also be entered.

* manufacturer or system dependent

7.2.2 Derived data types (Type definition)

Derived data types – also called type definition – are user defined data types which are based on the elementary data types. With type definitions, new data types with extended or altered attributes can be generated. In addition to this, very complex data models can be realized.

Type definitions can be made locally for the current POU, or globally for the entire project, also refer to sections 5.1 and 5.5.

For local use, derived data types are defined in the declaration part of the POU within the keywords `TYPE ... END_TYPE`.

Example of type definitions:

```
TYPE
  Unsigned : UINT;
  Signed : INT := -1;
  ChildAge : USINT (0..17);
  Children : ARRAY [1..10] OF ChildAge;
END_TYPE
```

The example shows type definitions “Unsigned” and “Signed”, which are derived from the elementary data types UINT or INT, resp. . The user defined data type “Signed” moreover gets the initial value “-1”.

The definition of the derived data type “ChildAge” is based on the data type USINT with a limited range (not supported as yet). The example of the array definition definition “Children” shows that derived data types can be used again for the new derivations.

Other characteristics

Other characteristics can be assigned to the data types in order to generate individual and complex data types:

- Initial value
- Enumeration
- Range
- Array
- Data structure

Initial value

An initial value can be assigned to the derived data type using the assignment operator “:=”, also refer to section 7.3.2. If the assignment is missing, the standard default value of the elementary data type is used, refer to section 7.2.1.

Examples:

```
TYPE
  Signed : INT := -1;
  InitDate : DATE := d#2000-01-01;
  InitStr : STRING (63) := '< input please >';
END_TYPE
```

Enumeration (ENUM)

An enumeration is a list with names (text constants). These names can be used for the processing of variables instead of values (numeric literals). All names are “enumerated” while defining in a name list. The enumeration is marked by parenthesis “()” .

Examples:

```
TYPE
  WeekDays : (Mo, Tu, We, Th, Fr);
  WeekendDays : (Sa, Su);
END_TYPE
```

In the example, two enumerations are defined. The enumeration “Week-Days” contains five names (text constants), the list “WeekendDays” contains two names.

WinSPS converts (internally) the text constants into an enumeration of integers with the help of the compiler. Every enumeration begins with the value “0”. All the following elements are numbered in the increasing order.

In the example given above, WinSPS uses the following **internal** numerical constants:

Mo	Tu	We	Th	Fr	Sa	Su
0	1	2	3	4	0	1

With the variable declaration of an enumeration , the names can be used in the program. The internal numbering is not visible at any point of time.

Example (programming language ST):

```
VAR
  AtWork : WeekDays ;
  AtHome : WeekendDays;
END_VAR

  AtWork := Mo;
  AtHome := Su;
```

RANGE

 **Ranges are not supported as yet!**

ARRAY

See Section 7.3.6.

Data structure (STRUCT)

See Section 7.3.7.

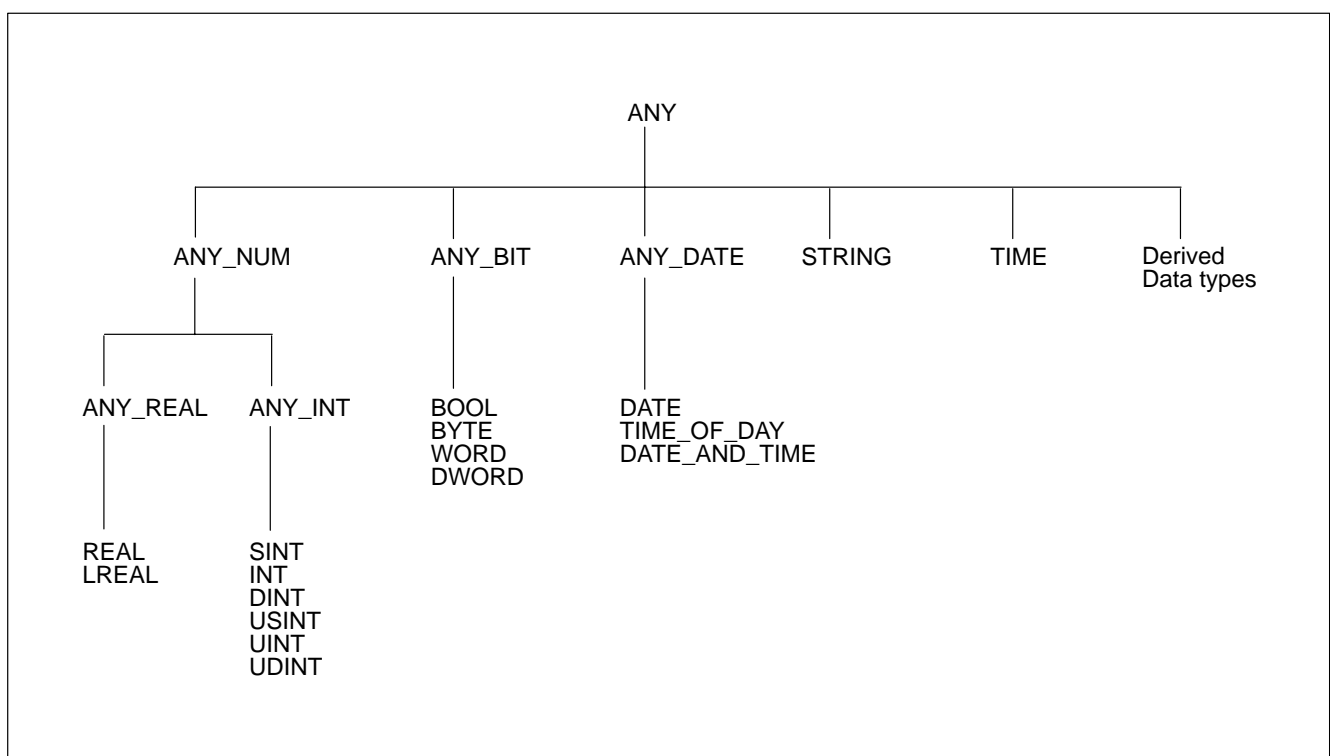
7.2.3 Generic data types

With the help of the generic data types, the elementary data types can be integrated into groups. They are started with the prefix "ANY". The following data types are integrated into groups.

- Signed/unsigned integer (ANY_INT)
- Floating point number (ANY_REAL)
- Numbers (ANY_NUM)
- Bit string (ANY_BIT)
- Date, time (ANY_DATE)
- All above-mentioned types as well as strings, time duration and derived data types (ANY)

With the grouping, a hierarchical order is also established. The following illustration shows the hierarchical levels, where the data type ANY is placed at the highest level. The data types LINT, ULINT and LWORD are not supported by Bosch and are not specified in the illustration.


☞ Generic data types (ANY...) are used only for illustrative group formation of elementary data types. They can not be used for the declaration of variables or for program processing. The declaration takes place instead through the elementary data types associated with the group, e.g. "BYTE" for the group "ANY_BIT".



Hierarchical order of the generic data types.

Overloaded functions

The characteristics of the generic data types are used in many standard functions. As a result of this, input variables of a function can not only be applied for one but multiple data types. With this characteristic, they are termed as overloaded or overloadable functions, refer to section 12.1.1.

 **Generic data types can be used only for standard and manufacturer functions. The programming of overloaded functions (user functions) is not possible.**

7.3 Variables

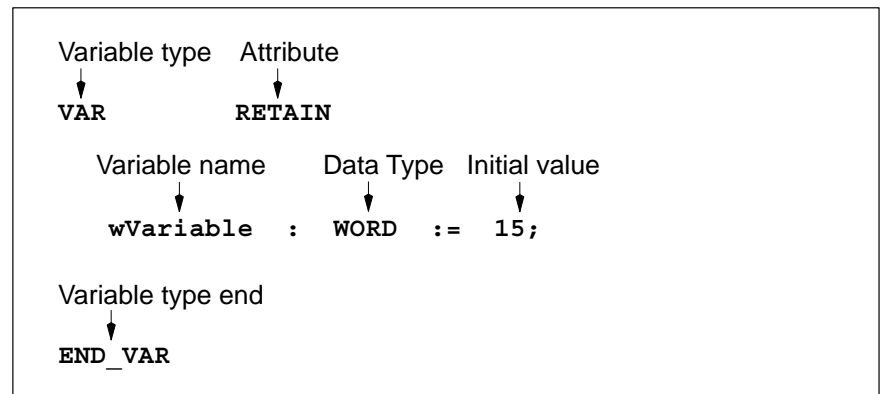
Variables have the following characteristics:

- Variables represent user related data ranges.
- In comparison to the constants, the contents of a variable can be changed.
- Variables are declared in the declaration part of a POU, or in the editor for global variables. Here, a data type is assigned to an identifier (variable name).
- The use of variables according to their types is checked so that access in incorrect data format can be prevented.
- With the declaration of a variable, initial values for the program start can be specified.
- Variable attributes allow additional features such as remanence.
- The memory area of a variable is automatically assigned. Physical addresses form an exception.

7.3.1 Declaration of variables

All variables are declared in the declaration part of a POU. In the instructions part, no variables may be declared. WinSPS provides a special editor for global variables, refer to section 5.4.

The declaration of a variable should get illustrated with the following example:



Elements of a variable declaration (example)

The use of variables can be influenced through the **variable type**, refer to section 6.3.1. Depending upon the variable type, variables are integrated into a block.

Attributes mark a special characteristic of the variables or a variable block, refer to section 7.3.8.

Rules applicable for identifiers also apply for **variable names**, refer to section 7.1.2.

The **data type** determines the characteristic of the variables, refer to section 7.2.

Initial values are the starting value for the program start or for the first call to the function or function blocks, refer to section 7.3.2.

Every declaration row is terminated with a semicolon “;”. Every variable block is ended with “END_VAR”.

In the declaration part, any number of variable blocks may be specified. The frequency of same types of variables and the sequence can be anything. In case of entries in the declaration tables, WinSPS integrates scattered variables of the same variable type into one block.

While declaring an instance for function blocks, the FB name is given in place of the data type, for structure of the structure name, also refer to section 6.6 or 7.3.7, resp. .

Examples of different variable declarations:

```
VAR
  Local AT %Q1.0 : BOOL;          (* Physical address *)
  In1 : INT := 15;              (* Variable with initial value *)
  In2 : DATE;                   (* Data type: Date *)
  Counter : CTU;                (* Instance of a FB *)
  Person : Person_Str;         (* Decl. of a structure *)
  V1, V2 : WORD;               (* two decl. in a row *)
END_VAR

VAR_OUTPUT RETAIN
  Out : WORD;                  (* Output variable with attribute *)
END_VAR
```

7.3.2 Initialization of variables and remanence

With the declaration of a variable using the assignment operator “:=”, an initial value (starting value) can be assigned. If the assignment is missing, the standard default value of the elementary data type is used, refer to section 7.2.1. Thus, variables always have specific initial values.

Examples:

```
VAR
  Logical1 : BOOL := TRUE;
  Negative : INT := -1;
  InitDate : DATE := d#2000-01-01;
  InitStr : STRING (63) := '< input please >';
  NoInit : BOOL;                (* Standard default value for
                                data type BOOL = FALSE *)
END_VAR
```

Variable types

Initial values may be specified in all variable types except in VAR_IN_OUT and VAR_EXTERNAL. Global variables are initialized within VAR_GLOBAL.

**DANGER**

VAR_IN_OUT cannot be assigned an initial value. The initialization can instead take place in the FB call with the specification of the corresponding formal parameter.

In case initial values are specified within the declaration of VAR_IN_OUT, WinSPS does not generate any error message during compilation or linking. The initial values are however not set in the controller!


One-time initialization

The initial value is assigned to the variables only once. In case of declarations within the PROGRAM POU, the assignment takes place at the start and also during the first run of the program. Initial values, which are declared within a FUNCTION_BLOCK, are assigned during the first call. As a FUNCTION does not have a "memory", initial values are more or less assigned with every call.

Remanence characteristics (RETAIN)

A remanence characteristic is ensured by the variable attribute RETAIN (refer to section 7.3.8). Remanence means that the values of a variable are retained after a power failure.

Moreover, remanence ranges are configured in the organization module OM2. Pay attention to the instructions in the software manual "PCL and CL550, Programming and Operation" (order no. 1070 072 189) or in "iPCL, System Description and Programming Manual" (order no. 1070 073 875).

 **All variables are basically handled as with RETAIN attribute. Exceptions are the standard FBs. These are basically not remanent i.e. they are initialized after every STOP/RUN switchover.**

Cold reset

Cold reset means : Initialization of all variables.

Cold reset is the type of start, in case of which entire variables and memory areas are reinitialized. All variables are overwritten by the initial or default values of the data type. This process, which is also termed as restart, can take place automatically under certain circumstances (e.g. after power failure) or can even be carried out manually by the user (e.g. by pressing the reset button).

Warm start

Warm start means : Initialization of the non remanent variables.

During a warm start of the controller (e.g. after a switchover STOP/RUN), the values from the last PLC cycle are restored even if initial values were specified. The remanence characteristic thus has priority over the initial values.

Restart of the PLC system and its application program at the interruption point (Hot Restart) explained in the IEC 61131-3 is not supported by the Bosch controller.

Startup characteristics

Here, the different starting options and the characteristics are shown:

Case 1: Program loading

Program loading results in the initialization of **all the** variables (cold start).

Case 2: Stopping after error, as well as pressing the reset key (CL550)

This asynchronous program break results in the initialization of **all the** variables (cold reset).

Case 3: Switchover between STOP/RUN mode

While changing over from STOP to RUN without changing the PLC program, the **non-remanent** variables are initialized (restart, warm start). The **remanent** variables retain their runtime values.

Case 4: Restart after power failure or through regular starting of the controller from a mains switch or “booting up” (PCL) resp.

After starting the controller, the **non-remanent** variables are initialized (restart, warm start). The **remanent** variables retain their runtime values.

Further detailed information concerning the remanence and starting characteristics can be found in the software manual “PCL and CL550, Programming and Operation” (order no. 1070 072 189) or in “iPCL, System Description and Programing Manual” (order no. 1070 073 875).


7.3.3 Access to variables

The access to variables within the instructions part is dependent upon the programming language used. Further information can be found in the respective section of the programming language.

7.3.4 Physical addresses

Physical PLC addresses are

- Inputs
- Outputs
- Marker

 **Physical addresses may be declared only within the POU of the type “PROGRAM” in the variable types “VAR” or “VAR_GLOBAL”. In the function block, the import through “VAR_EXTERNAL” is possible, however, not in the functions.**

Format

Physical addresses are differentiated from the other variables through prefixed “%” character. Subsequently follows a prefix which is specified by the IEC:

Prefix	Explanation
I	Input
Q	Output
M	Marker

With the flag for the data format following this, the data width is encrypted. The check, whether the data matches the data type, is not performed. The address data following this is manufacturer-specific. In the WinSPS, two positions – separated by a decimal – are realized.

1st Position: Byte number
2nd Position: Bit number

In case of bit addresses, the bit number is specified. In case of all other data formats, the bit addresses and the decimal are left out.

Flag	Data format	Width	Example
X or none	Bit	1 bits	%IX0.0 or %I0.0
B	Byte	8 bits	%IB0
W	Word	16 bits	%QW16
D	Double word	32 bits	%MD4

Declaration and access

 **Physical addresses must be declared.**

The keyword “AT” is specified in the declaration block before the physical address and separated by a space. With this, the assignment to the physical address takes place. The assignment of symbolic names is not compulsory but recommended. As a result of this, physical addresses can be directly shown or accessed through a symbolic name.

Examples (programming language ST):

```

VAR
  AT %Q0.0 : BOOL;      (* directly shown variable *)
  bIn AT %I0.1 : BOOL;  (* with symbolic names *)
  wM4 AT %MW4 : WORD;  (* marker as word operator *)
END_VAR

```

```
%Q0.0 := bIn;  (* access *)
```

The example contains the declaration of the marker word with the data type WORD. This variable cannot be accessed by mistake with another data type such as DWORD or INT.

**CAUTION**

WinSPS manages physical addresses with the help of the symbol file. Entries for inputs, outputs and markers are automatically taken up and the same may not be changed!

Access outside the PROGRAM POU

Outside the PROGRAM POU, physical addresses can be accessed in different ways:

- 1) Through the call interface of functions and function blocks
- 2) As global data with VAR_GLOBAL and VAR_EXTERNAL (not allowed in case of functions)

Example concerning 1):

```

PROGRAM X
VAR
  AT %Q2.0 : BOOL;
  X1 AT %I2.0 : BOOL;
  ...
END_VAR

  FB1_Inst (Par_1:=%Q2.0, Par_2:=X1);
END_PROGRAM

```

```

FUNCTION_BLOCK FB1
VAR_IN_OUT
  Par_1 : BOOL;
  Par_2 : BOOL;
END_VAR
  ...
END_FUNCTION_BLOCK

```

The physical addresses are declared in the declaration part of the PROGRAM POU as **local** variables. During the call to the "FB1", these addresses are assigned to the formal parameters and thus are transferred to the function block. Within the "FB1", these physical addresses are accessed using the names of the formal parameters "Par_1" and "Par_2".

Example concerning 2):

```

PROGRAM Y
VAR_GLOBAL
  AT %Q2.0 : BOOL;
  X1 AT %I2.0 : BOOL;
END_VAR
...

  FB2_Inst ();
END_PROGRAM

```

```

FUNCTION_BLOCK FB2
VAR_EXTERNAL
  %Q2.0 : BOOL;
  X1 : BOOL;
END_VAR
...
END_FUNCTION_BLOCK


```

Here, the physical addresses are declared as global variables. During call to the “FB2”, they are not transferred to the function block. In the declaration part of the “FB2”, the addresses are imported via “VAR_EXTERNAL”. As a result of this, the variable names of the addresses do not change.

7.3.5 String variables

The data type STRING for strings allows a flexible length up to 63 characters. With the declaration, a string length can be reserved. The length is given after the keyword “STRING” within the parenthesis “()”. If there is no length data, it is automatically set to 32. The string terminator ‘\0’ is automatically appended and must not be included in the data concerning the string length.

For string variables, the rules for string literals are applicable, refer to section 7.1.3.

 **Please note that the string length must not be exceeded when strings are assigned to a variable (see example below). The compiler will not detect an exceeding. The consequence will be that other data areas of the SPS are overwritten during runtime.**

Examples:

```

VAR
  sz_8_Char : STRING(8);
  sz_8_Initialized : STRING(8) := '8 characters';
  sz_32_Char : STRING; (* Standard length = 32 *)

  (* faulty strings because of exceeded length*)
  sz_Error_1 : STRING(8) := '9 characters';
  sz_Error_2 : STRING := 'This string consists of
  more than 32 characters';
END_VAR

```

7.3.6 ARRAY

Single element and multielement variables

A single element variable represents a single element of a data type. These can be already mentioned physical addresses, or any simple variable.

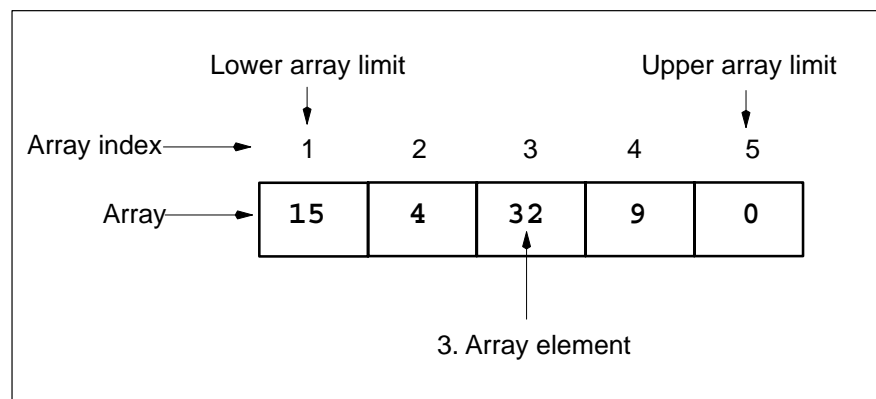
Examples:

```
bIn AT %I0.1 : BOOL;           (* Physical address *)
lrFloat : LREAL;              (* long real number *)
szStr : STRING := 'Hello';    (* String *)
```

In contrast to the single element variables, with multi-element variables, arrays and structures (refer to section 7.3.7) can be formed.

Array

An array consists of a series of variables of the same data type. The variables contained in an array are termed as array elements. The individual array elements can be accessed through an array index. The initial value and the end value of the array index are determined from the array limits.



Example for an array with 5 array elements


Array definition

```
Array_1: ARRAY[1..5] OF INT := [15, 4, 32, 9, 0];
```

The example shows an array with the name "Array_1". It contains five array elements of data type INT starting with the array index 1.

The array limits are specified in the square brackets. In this case, the upper limit is separated from the lower limit by two dots: For the array limits integer numeric literals are used.

The specification of initial values is similarly possible for arrays. Here, a special syntax is to be followed, see below: Initialization of arrays.

 **The maximum array size is presently limited to 502 Bytes. If the allowed array size is exceeded, an error message is given out during compilation.**



DANGER

While accessing array elements, the array limits may not be exceeded.

Access above array limits is presently not checked and leads to uncontrolled behavior!

Access to array elements

The access in the instructions part takes place using variable names followed by square brackets, in which the array index is specified.

Examples

Example for the declaration of an array and access in the programming language IL:

```

VAR
  aValue : ARRAY[1..5] OF SINT;
          (* Array declaration *)
  siElement : SINT;          (* Var. for test purposes *)
END_VAR

LD      15          (* Array initialization *)
ST      aValue[1]
LD      4
ST      aValue[2]
LD      32
ST      aValue[3]
LD      9
ST      aValue[4]
LD      aValue[5]  (* contents of the 5th array
                   element *)
    
```

Example for access in the programming language ST:

```

aValue[1] := 15;      (* Array initialization*)
aValue[2] := 4;
aValue[3] := 32;
aValue[4] := 9;
siElement := aValue[5]; (* Contents of the 5th
                        array element*)
    
```

Input of arrays

POU type		Variable declaration		Type definition			
Variable type	Name	Data type	Initial value	Address	Attribute	Monitor data	Comment
VAR	aValue	ARRAY[1..5] OF SINT					
VAR	aExp3	ARRAY[0..5] OF INT	[3(7), 2(13), 115]				

Example for the entry of arrays in the declaration table .

In combination with structures, arrays are declared not inside a variable block but in the type definition. This case is explained in detail in section 7.3.7.

Mutidimensional arrays

Mutidimensional arrays can be created by specification of multiple comma separated array limits.

The array elements are accessed using the comma separated array index.

Example:

```
VAR
  aValue : ARRAY[1..5,0..15] OF SINT;
END_VAR

  aValue[1,0] := 15; (* First array element*)
  aValue[5,15] := 0; (* Last array element*)
```

The example shows a two-dimensional array. The array size can be determined by multiplying the number of array elements of every dimension:

1st Dimension: [1..5] = 5 Elements

2nd Dimension: [0..15] = 16 Elements

In the example shown above, $5 * 16 = 80$ measured values can be realized.

Initialization of arrays

For the initialization, the statements concerning the "Initialization of variables" from section 7.3.2 are applicable.

The initialization of arrays can take place using two different methods:

- Initialization during the declaration
- Initialization in the instructions part at the runtime

Depending upon the application, both methods have advantages and disadvantages:

	Advantages	Disadvantages
Initialization in the declaration part	For the initialization, there is no processing time in the PLC.	Individual array elements can be initialized with great difficulty in case of complex arrays. The clarity suffers. Array elements can be assigned only simple literals.
Initialization in the instructions part	Complex arrays can be clearly initialized using appropriate algorithms. Array elements can be assigned extensive part expressions e.g. with complex calculations.	Depending on the array size, the initialization at the runtime requires processing time in the PLC. In order to avoid repeated initialization operations, suitable mechanisms must be programmed.

Array initialization in the declaration part

For the array initialization in the declaration part, special syntax is used. This must be illustrated with the help of a few examples:

Single dimensional array:

Beginning with the first array element, the initial values are entered in an ascending order and delimited by comma. The complete expression is enclosed by square brackets.

```
aExp11 : ARRAY[1..3] OF INT := [5,10,15];
```

This initialization is equivalent to the following assignment (programming language ST):

```
aExp11[1] := 5;  
aExp11[2] := 10;  
aExp11[3] := 15;
```

Leaving out of initial values:

With the assignment, not all array elements should be initialized. The array elements lying at the back may be left out during initialization.

```
aExp12 : ARRAY[1..3] OF INT := [5,10];
```

This initialization is equivalent to the following assignment:

```
aExp12[1] := 5;  
aExp12[2] := 10;
```

The last element is not initialized and as a result contains the standard default value "0" of the elementary data type "INT".

Multiplier:

With the multiplier, multiple elements following after one another can be initialized with the same value. The multiplier is set before the parenthesis. The initial value is given inside the parenthesis.

```
aExp13 : ARRAY[0..5] OF INT := [3(7),2(13),115];
```

In the example, the array elements are initialized as shown below:

```
aExp13[0] := 7;  
aExp13[1] := 7;  
aExp13[2] := 7;  
aExp13[3] := 13;  
aExp13[4] := 13;  
aExp13[5] := 115;
```

Moreover, a multiplier can also be used on multiple values. The values are delimited by comma.

```
aExp14 : ARRAY[0..5] OF INT := [2(7,13),2(115)];
```

This initialization with multiplier is equivalent to the following assignment:

```
aExp14[2] := 7;
aExp14[1] := 13;
aExp14[2] := 7;
aExp14[3] := 13;
aExp14[4] := 115;
aExp14[5] := 115;
```

Multidimensional arrays:

In case of multidimensional arrays, there is a nesting of square brackets. A combination [] is required for each dimension. Thus for example, a three dimensional array is initialized with three nested brackets levels [[[]]]. The bracket sequence is illustrated through the following example.

```
aExp15 : ARRAY[1..3,1..2] OF INT :=
[[1,2],[3,4],[5,6]];
```

In the example, the array elements are initialized as shown below:

```
aExp15[1,1] := 1;
aExp15[1,2] := 2;
aExp15[2,1] := 3;
aExp15[2,2] := 4;
aExp15[3,1] := 5;
aExp15[3,2] := 6;
```

The multiplication factor can also be used for multidimensional arrays. In this case, the multiplication factor must be adjusted to the dimension range:

```
aExp16 : ARRAY[1..2,1..3] OF INT := [2([3,6,9])];
```

This initialization is equivalent to the following assignment:

```
aExp16[1,1] := 3;
aExp16[1,2] := 6;
aExp16[1,3] := 9;
aExp16[2,1] := 3;
aExp16[2,2] := 6;
aExp16[2,3] := 9;
```

The following example shows the complete initialization of a three dimensional array. Each array element contains the value "1":

```
aExp17 : ARRAY[1..4,1..2,1..3] OF INT :=
[4([2([3(1))]))];
```

Array initialization at the runtime

In case of initialization at the runtime and the evaluation of array elements, multidimensional arrays can be processed very clearly with the help of recursive statements. The following example shows the array access with the help of FOR loop of the programming language ST (also refer to section 9.2.7):


```

VAR
  aExpl8 : ARRAY[1..4,1..2,1..3] OF INT;
  i, j, k : INT;
  bInit : BOOL := TRUE;
END_VAR

IF (bInit = TRUE) THEN
  FOR i := 1 TO 4 DO
    FOR j := 1 TO 2 DO
      FOR k := 1 TO 3 DO
        aExpl8[i,j,k] := i * 2 + (k * 5);
      END_FOR;
    END_FOR;
  END_FOR;
  bInit := FALSE;
END_IF;

```

Using the IF selection statement of the boolean variable “bInit”, the initialization is controlled so that the array is initialized only once. The initialization is as a result actuated only at the start of the program or during the first call to the POU resp. .


 **Array initializations at the runtime require processing time in the cyclic PLC program. This time must be taken into consideration in the calculation of the maximum PLC cycle time.**

Complex data model

With combinations of arrays and data structures, very complex data objects can be formed:

- Array in structure
- Arrays of structures

The options are illustrated in section 7.3.7.

 **These complex data models are in preparation and allowed in the WinSPS higher version 3.10.**

Definition or declaration of arrays?

Arrays may be specified within a variable declaration VAR ... END_VAR as well as within a type definition TYPE ... END_TYPE. Both variants are allowed and are different from each other in the sense that within the type definition only the “outline” of the array is defined. Within the variable declaration in comparison, a concrete data object is set up.

Inside the type definitions, arrays should be set up when they must be referred in this definition block.

Example of array in structure

```

TYPE
  aMeasValues : ARRAY [1..20] OF INT;
  Measurement :
  STRUCT
    szMeasObject : STRING (20);
    dtMeasDate : DATE;
    MeasValue : aMeasValues; (* Array in structure *)
  END_STRUCT;
END_TYPE

```

Only with the declaration of an instance of this structure, a data object for accessing the structure elements and array elements is set up.

7.3.7 Data structures (STRUCT)

Arrays consist of elements of the same data type. A data structure (in short: structure) in comparison allows integration of various data types related to a data object.

Definition

Structures are user-defined data types. These are defined within the keywords TYPE and END_TYPE, refer to section 7.2.2. The structure elements are specified between the keywords STRUCT and END_STRUCT. The structure elements are entered like rows of a variable declaration, refer to section 7.3.1. Directly before the keyword STRUCT, the structure name is specified followed by a colon ":". Rules applicable for identifiers also apply for names, refer to section 7.1.2. END_STRUCT is terminated with a semicolon ";".

Common syntax for the definition of structures:


```

TYPE
  Structure_name :
  STRUCT
    Structure_element1 : Data type := Initial value;
    Structure_element2 : Data type := Initial value;
    Structure_elementn : Data type := Initial value;
  END_STRUCT;
END_TYPE

```

Declaration and access

With the definition, first of all the "outline" of the structure is specified. Only with the declaration of an instance of this structure a data object for accessing the structure elements is set up. Due to instance building, a structure can be used multiple times. Each instance here has its own range.

 **The maximum size of each structure is presently limited to 502 Bytes. If the allowed size is exceeded, an error message is given out during compilation.**

The structure elements are accessed via the dot-operator.

Example of definition and declaration of a structure and access to structure elements in the programming language IL:

```

TYPE
  Person_Str:
  STRUCT          (* Definition of a structure *)
    szName : STRING (20);
    szWeb  : STRING (63);
    udiZIP : UDINT
  END_STRUCT;
END_TYPE

VAR
  Person1 : Person_Str; (* Declaration of structure *)
  Person2 : Person_Str; (* 2nd instance of structure*)
  ZIP := UDINT;
END_VAR

(* Access to structure *)
LD   'Katzenmeier'
ST   Person1.szName
LD   'www.buero-fuer-informatik.de'
ST   Person1.szWeb
LD   Person2.udiplz
ST   PLZ

```

Example for access in the programming language ST:

```

(* Access to structure *)
Person1.szName := 'Katzenmeier';
Person1.szWeb  := 'www.buero-fuer-informatik.de';
PLZ := Person2.udiplz;

```

Input of structures

For the entry and processing of structures, WinSPS provides input masks for local and global type definitions, refer to section 5.1.3 and 5.5.2.

POU type	Variable declaration	Type definition		
Type name: <input type="text" value="Person_Str"/>				
Name	Data type	Initial value	Comment	
szName	STRING (20)			
szWeb	STRING (63)			
udZIP	UDINT			

Example for definition of a local structure in the declaration table.

Complex data model

With combinations of arrays and data structures, very complex data objects can be formed:

- Structure in structure
- Array in structure
- Arrays of structures

☞ **These complex data models are in preparation and allowed in the WinSPS higher version 3.10.**

7.3.8 Variable attributes

In addition to the data type and initial value, attributes can be assigned to the variables. Attributes mark a special characteristic of the variables. The following attributes are defined in IEC 61131-3 :

Attribute	Explanation
RETAIN	Battery backed-up, remanent
CONSTANT	Constant
R_EDGE *	Rising edge
F_EDGE *	Falling edge
READ_ONLY	Write-protected
READ_WRITE	Read and write access

* The attributes related to the edge control are presently not supported. Edge control can be realised through the standard function blocks "R_TRIG" and "F_TRIG", refer to section 12.2.

☞ **Up to and including the firmware versions 2.3 of PCL and 1.4 of CL550, all variables are basically RETAIN handled. Exceptions are the standard FBs. These are basically not remanent i.e. they are initialized after every STOP/RUN switchover.**

"RETAIN" and "CONSTANT" refer to the complete part of a variable type and are set directly behind the keyword of the variable type. The other attributes can be set individually, however, not in combination with the definitive marks!

Simultaneous use of "CONSTANT" und "RETAIN" does not make sense and is not allowed!

Examples:

```
VAR RETAIN (* Variables with RETAIN characteristic *)
  wRemanent : WORD
  bRemanent AT %Q3.7 : BOOL;
END_VAR
```

```
VAR_ACCESS
  bState : CPU_3.%I5.3 : BOOL READ_ONLY;
                                     (* access path only read access *)
END_VAR
```

Attribute CONST and DEFINE Editor

By assigning the attribute "CONSTANT", a constant is generated instead of a variable. This constant occupies in the PLC the memory location underlying its data type.

Alternately, in the so-called "DEFINE Editor" global constants are set up, refer to section 5.6. Constants defined there do **not** occupy any memory location in the PLC. It is recommended that the constants be always defined using this editor and not using the variable attribute "CONSTANT".

Application area of attributes

Attributes are not allowed the same way for all variable types:

Variable type	RETAIN	CONSTANT	R_EDGE F_EDGE	READ_ONLY READ_WRITE
VAR	Yes	Yes	–	–
VAR_INPUT	–	–	Yes	–
VAR_OUTPUT	Yes	–	–	–
VAR_IN_OUT	–	–	–	–
VAR_EXTER- NAL	–	–	–	–
VAR_GLOBAL	Yes	Yes	–	–
VAR_ACCESS*	–	–	–	Yes

* The variable type VAR_ACCESS is presently not supported.

Input of attributes

In the declaration table for the variable declaration, attributes can be selected from a column meant for the purpose. WinSPS automatically sets this individually or to a variable block in case of a definitive mark.

POU type		Variable declaration		Type definition			
Variable type	Name	Data type	Initial value	Address	Attribute	Monitor data	Comment
VAR	byBuffered	BYTE			RETAIN		
VAR	byConstant	BYTE	16#0D		CONSTANT		
VAR	byNone	BYTE	16#0A				

Example for the specification of attributes in the declaration table .

```
VAR
  byNone : BYTE := 16#0A;
END_VAR
```

```
VAR RETAIN
  byBuffered : BYTE;
END_VAR
```

```
VAR CONSTANT
  byConstant : BYTE := 16#0A;
END_VAR
```


Illustration of the above-mentioned example in the text form

8 Programming language Instruction List (IL)

The instruction list (IL) as per IEC 61131-3 involves a machine-like language. Machine-like means that the instructions can be directly converted into the binary machine code of the PLC.

WinSPS and IL

Parallel to the IL as per IEC, programming can also be done in WinSPS in the tried and tested, "classical" IL. The classical IL is called up in the editor and monitor using the button *IL*, the instruction list as per IEC using the button *IEC*.

 **In order to prevent mixup with the established Bosch programming languages, the following practice is used:**
IEC-IL: Instruction list as per IEC 61131-3.
Bosch-IL: Classical instruction list (based on DIN 19239).

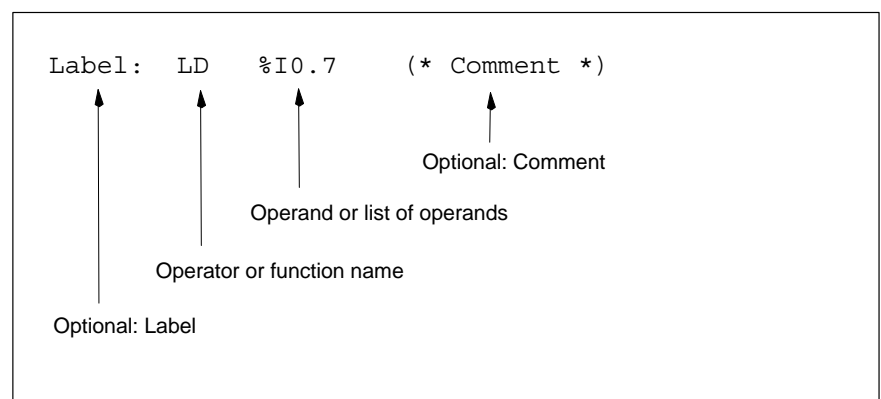
In this section, the IEC-IL is explained. For the programming in Bosch-IL, the manual "PCL and CL550, Programming and Operation" (order no. 1070 072 189) or "iPCL, System Description and Programming Manual" (order no. 1070 073 875), is available.

Please pay attention to the programming examples concerning the IEC-IL in the WinSPS help, chapter "Introduction to WinSPS".

8.1 Instructions

In IL, execution instructions are specified for the PLC in a row. A complete sequence of instructions (IL sequence) can stretch over multiple instruction rows.

An IL instruction row is specified in the following format:



Instructions in IL

The **Label** is optional. It is specified when a jump is made from another instructions row into this row, refer to section 8.4.2.

At the position **operator** , either an IL operator or a function name is specified. This describes the operation to be carried out. All possible operators are listed in section 8.5.

There must be at least one **delimiter** between the operator and operand. The space or the tab can be used as delimiter.

Operands are variables or constants, using which an operation (instruction) is carried out using the operator. At the time of a function call, the operand is an input parameter. Depending upon the type of the operation or the function call, multiple operands can be specified as **operand list** delimited by comma. For a few operators, no operand is necessary, e.g. "RET".

In order to realise nesting, parenthesis are allowed, refer to section 8.4.3.

The **label** is optional. Comments may be given within an instruction row at the end of the line. Also refer to section 7.1.5.

 **The comment marking semicolon “;” used in the classical programming is not allowed.**

8.2 Working register and status bits

Working registers (accumulators), as you find in the classical Bosch-IL application, do not exist in case of IL as per IEC 61131-3. However, there is a “virtual” working register with the name “CR ”, refer to section 8.3.

Likewise, there are no **status bits**, which in the Bosch-IL, indicate the conditions Interrupt, Carry, Overflow, Zero and Negation.

8.3 Current Result (CR) – the universal accumulator

Similar to the register for the “Result of Logic Operations” (RLO) out of Bosch-IL, the IEC-IL recognizes an accumulator with the name “**Current Result**” (CR). It is used to accept and process operands and results of the PLC instructions.

Example of an evaluation of the CR:

```
M1: LD      iOperand1
      ADD    iOperand2
      ST     iResult      (* calculation result *)
      EQ     100          (* comparison CR = 100 ? *)
      JMPC   M2           (* ...then jump to M2 *)
```

In the example, the instruction EQ is used, which accesses the current result (CR). The CR contains the result of the addition generated by the instruction Store (ST). It can be used until it is reset by an operation such as LD. A list of IL instructions which influence the CR, is shown in the following table and in the individual instruction descriptions in section 8.5.

Behaviour in case of different operators

Depending upon the operator (refer to section 8.5), the CR is influenced differently:

Influence	Explanation
create	The operators LD and LDN create a CR i.e. the CR is initialised with a new value.
process	The instruction uses the CR of the preceding result and modifies the same. Examples: &, OR, ADD, MUL, GT, EQ, LE.
leave unchanged	The instruction uses the CR of the preceding result, however, it does not modify the same. Examples: ST, S, R, RETC
set to undefined	The following instructions of a CR, which is set as undefined, may not use the same for linking. Example: JMP, CAL

In case of certain operators, the IEC leaves open, whether the CR is processed further, or must remain unchanged or must be set as undefined. A description of the CR behavior of the WinSPS is given in the sub-sections of section 8.5.

Function calls

The CR allows itself to be used not only on a result of a instruction sequence but also before that. In case of a function call for example, there is the option of transferring the initial input parameters from the CR of the earlier executed instruction without further specifications, refer to section 8.5.8.

Size and data type

In the real sense, the accumulator is a working register in the processor of the PLC . However, in case of 61131-3, this accumulator is not assigned to a fixed PLC memory. The CR is rather a "virtual" working register. As a result of this, the CR lets itself be used in a flexible manner for various data types and data widths. The universal accumulator CR can accept the following data types:

- Generic data type, refer to section 7.2.3
- Instance of a function block

 **Attention must be paid to the compatibility of the data types for operations following one after the other.**

8.4 Program rules

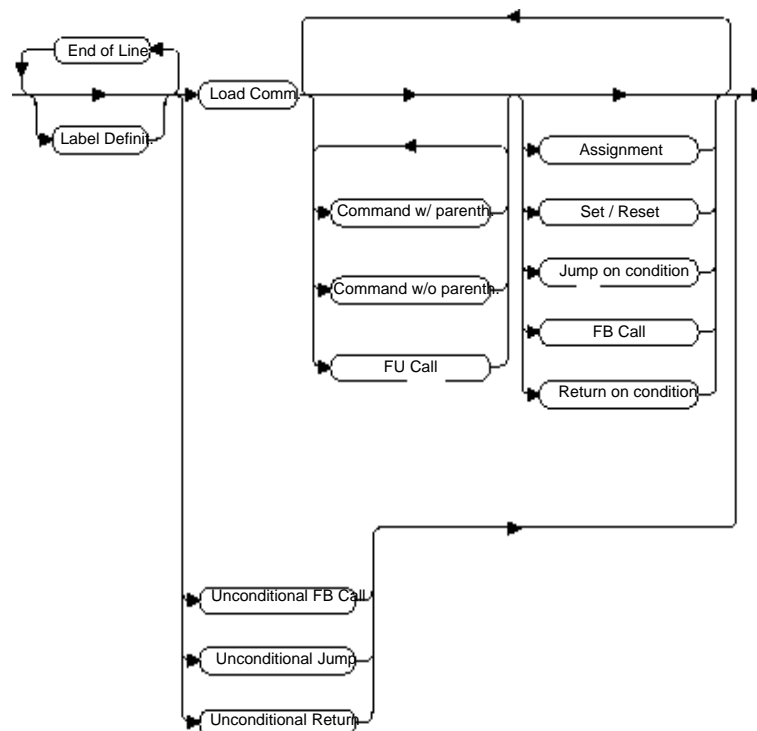
8.4.1 IL sequences

An IL sequence starts with a load instruction (LD or LDN) and is not terminated until a new load instruction is used. Absolute function block calls as well as jump and reverse jump operations form an exception. These are terminated on their own.

With the load instruction, the data type of the specified operand establishes the allowed data type for the following operand in this sequence. Two operations following one after the other must be type-compatible. With the help of the standard functions for type conversion, incompatible data types can be converted, refer to section 12.1.3.

With the parenthesis, multiple nesting levels can be realized, refer to section 8.4.3.

The following syntax graph gives an overview of the IL sequences. The order is from the left to the right. The usage of the individual command groups is described in detail in section 8.5.



Syntax graph for IL sequences

8.4.2 Label

The label is specified when a jump is made from another instructions row into this row. The label may also be used alone in a row i.e. without an operator or operand. If no label is specified in an instruction row, even the colon must be left out.

Rules applicable for the identifiers also apply for the names of the labels, refer to section 7.1.2.

 **Labels may exist only at the beginning of a sequence i.e. before a load instruction.**



Syntax graph of the label

Example for use of labels:

```

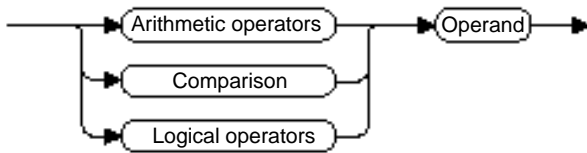
M1: LD    iResult
M2: ADD   iOperand1    (* M2 is at this position
                       not allowed! *)
      ST   iResult      (* calculation result *)
      LT   100          (* Comparison CR < 100 ?... *)
      JMPC M1          (* ...then jump to M1 *)
      JMP  M3          (* ...otherwise jump to M3 *)
...
M3: LD    iOperand2
M4: ST    iOperand3    (* Label not allowed here:
                       generates compiler error *)
...

```

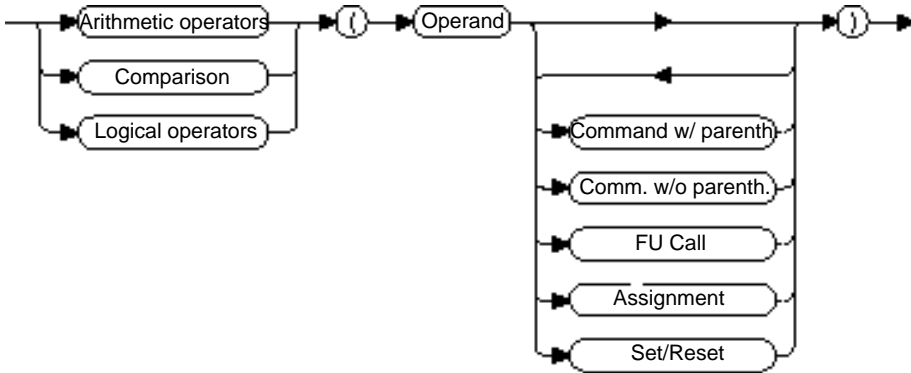
8.4.3 Nesting levels, Parenthesis

IL sequences can be combined with the help of the parenthesis modifier, in order to link the result with the CR. Such parenthesis modifier can be nested in multiple levels.

For introducing a parenthesis level, an opening parenthesis "(" is placed directly after the operator. Every level is ended with a separate instruction row with the closing parenthesis ")".



Syntax graph for commands without parenthesis



Syntax graph for command with parenthesis

The following table shows, for which operators a nesting level may be opened, also refer to section 8.5:

Logical operators	Arithmetic operators	Comparison
AND(&(ANDN(&N(OR(ORN(XOR(XORN(ADD(SUB(MUL(DIV(GT(GE(EQ(NE(LE(LT(
ADD(SUB(MUL(DIV(GT(GE(EQ(NE(LE(LT(
GT(GE(EQ(NE(LE(LT(

In comparison to the programming language ST, there is no ranking of arithmetic operators in the IL. Thus for example, an addition has the same priority during the processing as a multiplication. The processing can be specified through the sequence of the instruction rows and through parenthesis levels.

Example for multiple nesting levels:

```
LD      5          (* 5 *)
ADD (   2          (* 2 *)
ADD    4          (* 6 *)
MUL (   3          (* 3 *)
SUB    1          (* 2 *)
)      )          (* 12 *)
)      )          (* 17 *)
ST     iResult    (* 17 *)
```

The example works out the following mathematical expression:
 $5 + (2 + 4 * (3 - 1))$

In the comment, the current result CR is given according to the respective instruction. Thus, the variable "iResult" contains the value 17.

8.5 Instruction set

The following table shows a list of all IL instructions.

Modify operators

A few operators listed in the table can be modified in order to thus widen the meaning:

N Negation of the operand:

Reversing the binary signal state (from TRUE to FALSE, or from FALSE to TRUE),

or complement of one for a bit pattern (every single bit is reversed).

Examples: **&N**, **LDN**, **ORN**

C Conditional execution of the instruction:

Only when the CR contains the (boolean) state TRUE is the instruction executed

Examples: **JMPC**, **CALC**

The negation and the conditional execution can be combined in case of some operators.

Examples: **JMPCN**, **CALCN**

(Introduction of a nesting level:

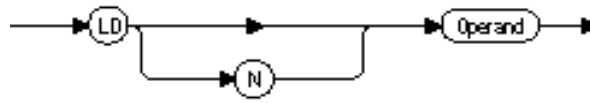
Example: **ADD(**, **XOR(**, **GT(**

Options for the nesting are listed in the table in the column “(”. Further information can be found in section 8.4.3.

Operator	Operand	(Explanation	Refer to section
LD	ANY		Copies the operand value in the working register CR	8.5.1
LDN	ANY_BIT		Copies the negated operand value in the working register CR	
ST	ANY		Sets the operand value equal to the current result (CR)	8.5.2
STN	ANY_BIT		Sets the operand value equal to the negated value of the current result CR	
S	BOOL		Sets the boolean operand value to 1 when the CR is 1	
R	BOOL		Sets the boolean operand value to 0 when the CR is 1	
AND	ANY_BIT	(Boolean AND	8.5.3
&	ANY_BIT	(Boolean AND	
ANDN	ANY_BIT	(Boolean AND, negated	
&N	ANY_BIT	(Boolean AND, negated	
OR	ANY_BIT	(Boolean OR	
ORN	ANY_BIT	(Boolean OR, negated	
XOR	ANY_BIT	(Boolean exclusive OR	
XORN	ANY_BIT	(Boolean exclusive OR, negated	
ADD	ANY_NUM	(Addition	
SUB	ANY_NUM	(Subtraction	
MUL	ANY_NUM	(Multiplication	
DIV	ANY_NUM	(Division	
GT	*	(Greater than	8.5.5
GE	*	(Greater than equal to	
EQ	*	(Equal to	
NE	*	(Not equal to	
LE	*	(Less than equal to	
LT	*	(Less than	
JMP	Label		Unconditional jump to the label	8.5.6
JMPC	Label		Jump to the label when CR = 1	
JMPCN	Label		Jump to the label when CR = 0	
CAL	Instance name		Unconditional FB call	8.5.7
CALC	Instance name		FB call when CR = 1	
CALCN	Instance name		FB call when CR = 0	
RET	–		Unconditional reverse jump	8.5.9
RETC	–		Reverse jump when CR = 1	
RETCN	–		Reverse jump when CR = 0	
)	–		Closing parenthesis in case of nesting	8.4.3

* ANY_INT, ANY_BIT, ANY_DATE, STRING, TIME

8.5.1 Load instructions – LD



Syntax graph of load instructions

Loading: LD

The value of the specified operand is loaded in the working register CR. The original contents of the working register are overwritten. The operand is not changed. The data type of the specified operand establishes the allowed data type for the following operands in this sequence.

Allowed data types: ANY, refer to section 7.2.3.

Influencing the CR: Create, refer to section 8.3.

Examples:

```

LD      5          (* Value 5 is loaded *)
LD      tod#8:27:00 (* Day time *)
LD      VAR_1      (* Contents of a variable *)
LD      %I5.7      (* directly shown
                   physical address *)
  
```

Negated loading: LDN

The negated value (also refer to the beginning of the section 8.5) of the specified operand is loaded in the working register CR. The original contents of the working register are overwritten. The operand is not changed. The data type of the specified operand specifies the allowed data type for the following operands in this sequence.

Allowed data types: ANY_BIT, refer to section 7.2.3.

Influencing the CR: Create, refer to section 8.3.

Example:

```

LDN     2#01101010 (* !!! Data type ANY_BIT !!!
*)
  
```

In the example, the negated value of the operand is loaded. Thus, the CR contains the value 2#10010101 after the execution of the instruction.

8.5.2 Assignments – ST, S, R

Assigning: ST

The contents of the working register CR are assigned to the specified operand. The original value of the operand is overwritten. The data type of the specified operand must match with the data type of the data element in the CR. The data type of the CR is established by the data type of the variables which has a value assigned to it. Further assignments can then be made only with variables of the same data type. An assignment can be followed by another.

Allowed data types: ANY, refer to section 7.2.3.

Influencing the CR: Leave unchanged, refer to section 8.3.

Examples:

```

ST    VAR_1      (* Assign CR to a variable *)
ST    %Q5.7     (* directly shown
                physical address *)

LD    tod#8:27:00 (* Load time of day... *)
ST    todVar1   (* ...assign 2 variables *)
ST    todVar2

```

Negated assigning: STN

The negated contents (also refer to the beginning of the section 8.5) of the working register CR is assigned to the specified operand. The original value of the operand is overwritten. The data type of the specified operand must match with the data type of the data element in the CR. The data type of the CR is established by the data type of the variables which has a value assigned to it. Further assignments can then be carried out only with variables of the same data type. Another assignment “ST” or “STN” can follow an assignment “STN” .

Allowed data types: ANY_BIT, refer to section 7.2.3.

Influencing the CR: Leave unchanged, refer to section 8.3.

Example:

```

LD    2#00111100 (* !! Data type ANY_BIT !! *)
STN   VAR_1

```

In the example, the specified value is assigned to the CR using “LD”. With “STN”, the negated value 2#11000011 is assigned to the variables “VAR_1”. The CR however contains the unchanged value 2#00111100.

Set: S

The specified operand is set (TRUE) when the contents of the working register CR is equal to “1” (TRUE). The operand remains set until a reset instruction reverses this state.

When this setting condition is not fulfilled, there is no change in the operands.

Allowed data type: BOOL, refer to section 7.2.3.

Influencing the CR: Leave unchanged, refer to section 8.3.

Examples:

```
LD    %M15.3      (* Load marker bit *)
S     %Q7.2       (* Set output *)
```

In the example, the output 7.2 is set depending upon the state of the marker 15.3.

Resetting: R

The specified operand is reset (FALSE) when the contents of the working register CR is equal to "1" (TRUE).

When this resetting condition is not fulfilled, there is no change in the operands.

Allowed data type: BOOL, refer to section 7.2.3.

Influencing the CR: Leave unchanged, refer to section 8.3.

Examples:

```
LDN   %M15.3      (* Load negated marker bit *)
R     %Q7.2       (* Reset output *)
```

In the example, the output 7.2 is set depending upon the negated state of the marker 15.3.

Example: Bistable elements

With a combination of setting and resetting instructions, the function of a bistable element (flipflop) can be replicated.

The output of a bistable element can be switched by a fulfilled setting condition to "1" (TRUE) or by a fulfilled resetting condition to "0" (FALSE). The state remains until the condition for the opposite state is fulfilled. If both conditions are simultaneously fulfilled, the data element assumes the state, for which the condition was processed last. So, it is "set dominant" or "reset dominant", refer to the examples:

Set dominant flipflop (SR):

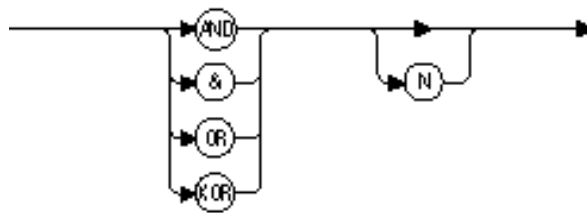
```
LD    _ReSet      (* Reset variable *)
R     _Q1         (* Reset output*)
LD    _Set1       (* Set variable *)
S     _Q1         (* Set output *)
```

Reset dominant flipflop (RS):

```
LD    _Set        (* Set variable *)
S     _Q1         (* Set output *)
LD    _ReSet1     (* Reset variable *)
R     _Q1         (* Reset output*)
```

For bistable elements, the standard function blocks "SR" and "RS" are also available, refer to section 12.2.1.

8.5.3 Boolean operators AND, &, OR, XOR



Syntax graph of the logical operators

Boolean AND: AND, &

Boolean (logical) AND of the specified operand value with the contents of the working register CR. The result is stored in the CR. The operand value is not changed.

In case of bit patterns (byte or word operands), the individual bits of the specified operand are linked with the corresponding bits of the CR.

Either the keyword "AND", or the short form "&" may be used.

Allowed data types: ANY_BIT, refer to section 7.2.3.

Influencing the CR: Further processing, refer to section 8.3.

Examples:

```
LD    %I8.2      (* phys. Binary address *)
&     %I8.5
ST    %Q4.0
```

Only when at the input 8.2 **and** 8.5, the state "1" (TRUE) exists, the output 4.0 is similarly set to "1".

```
LD    2#10100110 (* Bit pattern *)
AND   2#11110000
ST    %QB0
```

After the AND link of both the bit patterns, the output byte contains the bit pattern 10100000.

Negated AND: ANDN, &N

In case of binary operation, the contents of the specified operand are negated (also refer to the beginning of the section 8.5) and linked with the contents of the working register CR with the AND function. The result is stored in the CR. The operand value is not changed.

In case of bit patterns (byte or word operands), every single bit of the operand is negated (complement of one).

Either the keyword "ANDN", or the short form "&N" may be used.

Allowed data types: ANY_BIT, refer to section 7.2.3.

Influencing the CR: Further processing, refer to section 8.3.

Examples:

```
LD      %I8.2      (* phys. binary address *)
&N     %I8.5      (* negated state *)
ST      %Q4.2
```

The state at the input 8.5 is negated and subsequently follows the AND operation with input 8.2. At the input 8.2, the state “1” (TRUE) **and** at the input 8.5, the state “0” (FALSE) must exist so that the output 4.2 is set to “1”.

```
LD      2#10100110  (* Bit pattern *)
ANDN   2#11110000  (* Bit pattern is negated *)
ST      %QB4
```

Negation of the second bit pattern gives rise to 2#00001111. This bit pattern is linked to the first one in a binary form. The output byte contains the bit pattern 00000110.

Boolean OR: OR

Boolean (logical) OR of the specified operand value with the contents of the working register CR. The result is stored in the CR. The operand value is not changed.

In case of bit patterns (byte or word operands), the corresponding bits of every related operand are linked.

Allowed data types: ANY_BIT, refer to section 7.2.3.

Influencing the CR: Further processing, refer to section 8.3.

Examples:

```
LD      %I8.2      (* phys. binary address *)
OR      %I8.5
ST      %Q4.1
```

When the state at the input 8.2 **or** 8.5 is set to “1” (TRUE) – even at both the inputs simultaneously, the output 4.1 is set to “1”.

```
LD      2#10100110  (* Bit pattern *)
OR      2#11110000
ST      %QB8
```

After the OR link of both the bit patterns, the output byte contains the bit pattern 11110110.

Negated OR: ORN

In case of binary operation, the contents of the specified operand are negated (also refer to the beginning of the section 8.5) and linked with the contents of the working register CR with the OR function. The result is stored in the CR. The operand value is not changed.

In case of bit patterns (byte or word operands), every single bit of the operand is negated (complement of one).

Allowed data types: ANY_BIT, refer to section 7.2.3.

Influencing the CR: Further processing, refer to section 8.3.

Examples:

```
LDN   %I8.2      (* phys. address negated *)
ORN   %I8.5      (* likewise negated *)
ST    %Q4.3
```

With the instruction LDN, the input state at 8.2 is set in the CR after negation. The state at the input 8.5, is similarly negated by the operator ORN. Subsequently follows the OR link.

```
LD    2#10100110 (* Bit pattern *)
ORN   2#11110000 (* Bit pattern is negated *)
ST    %QB12
```

Negation of the second bit pattern gives rise to 2#00001111. This bit pattern is linked to the first one in a binary form. The output byte contains the bit pattern 10101111.

Exclusive OR: XOR

Boolean (logical) exclusive OR of the contents of the specified operand with the contents of the working register CR. Exclusive OR means that both the operands, which are to be linked, must be different so that the result assumes the state "1" (TRUE). The result is stored in the working register. The operand value is not changed.

In case of bit patterns (byte or word operands), the corresponding bits of every related operand are linked.

Allowed data types: ANY_BIT, refer to section 7.2.3.

Influencing the CR: Further processing, refer to section 8.3.

Examples:

```
LD    %I8.2      (* phys. binary address *)
XOR   %I8.5
ST    %Q4.4
```

The following table shows the result at the output 4.4 dependent upon the inputs:

%I8.2	FALSE	TRUE	FALSE	TRUE
%I8.5	FALSE	FALSE	TRUE	TRUE
%Q4.4	FALSE	TRUE	TRUE	FALSE

```
LD    2#10100110 (* Bit pattern *)
XOR   2#11110000
ST    %QB16
```

After the XOR link of both the bit patterns, the output byte contains the bit pattern 01010110.

Negated XOR: XORN

In case of binary links, the contents of the specified operand are negated (also refer to the beginning of the section 8.5) and linked with the contents of the working register CR with the XOR function. The result is stored in the CR. The operand value is not changed.

In case of bit patterns (byte or word operands), every single bit of the operand is negated (complement to one).

Allowed data types: ANY_BIT, refer to section 7.2.3.

Influencing the CR: Further processing, refer to section 8.3.

Examples:

```
LD    %I8.2      (* phys. binary address *)
XORN  %I8.5
ST    %Q4.5
```

The state at the input 8.5 is negated and subsequently follows the XOR link with input 8.2. The following table shows the result at the output 4.5 dependent upon the inputs (before the negation):

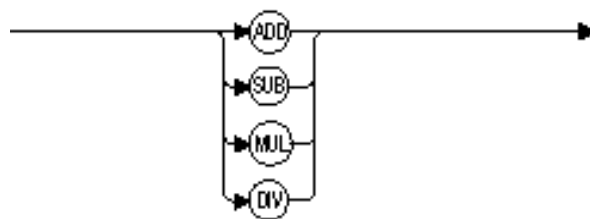
%I8.2	FALSE	TRUE	FALSE	TRUE
%I8.5	FALSE	FALSE	TRUE	TRUE
%Q4.5	TRUE	FALSE	FALSE	TRUE

```
LD    2#10100110 (* Bit pattern *)
XORN  2#111110000
ST    %QB20
```

Negation of the second bit pattern gives rise to 2#00001111. This bit pattern is linked to the first one in a binary form. The output byte contains the bit pattern 10101001.

8.5.4 Arithmetic operators ADD, SUB, MUL, DIV

For arithmetical (mathematical) calculations, there are four types of basic calculations available in IL i.e. addition, subtraction, multiplication and division.




Syntax graph of the arithmetical operators

Addition: ADD

The specified operand value is added to the contents of the working register CR and the result is stored in the CR. The operand value is not changed.

On the operator ADD, the opening parenthesis for the introduction of a nesting level can be used, refer to section 8.4.3.

 **The function ADD for multiple addition of numeric operands and for the addition of time values and addresses is described in section 12.1.**

Allowed data types: ANY_NUM, refer to section 7.2.3.

Influencing the CR: Further processing, refer to section 8.3.

Examples:

```
LD    24          (* Unsigned *)
ADD   10
ST    uiVar
```

“uiVar” and the CR contains the value “34” after addition.


```
LD    78.25      (* Floating point number *)
ADD   -1.5       (* Unsigned *)
ADD   6.75
ST    rVar
```

“rVar” and the CR contains the value “83.5” after addition. If “rVar” is an integer variable (ANY_INT), it would contain the value “83”. The after decimal positions are thus cut off.

Subtraction: SUB

The specified operand value (subtrahend) is subtracted from the contents of the working register CR (minuend) and the result is stored in the CR. The operand values are not changed.

On the operator SUB, the opening parenthesis for the introduction of a nesting level can be used, refer to section 8.4.3.

 **The function SUB for multiple subtraction of numeric operands and for the subtraction of time values and address is described in section 12.1.**

Allowed data types: ANY_NUM, refer to section 7.2.3.

Influencing the CR: Further processing, refer to section 8.3.

Example:

```
LD    24
SUB   10
ST    uiVar
```

“uiVar” and the CR contains the value “14” after subtraction.

Multiplication: MUL

The specified operand value is multiplied with the contents of the working register CR and the result is stored in the CR. The operand value is not changed.

On the operator MUL, the opening parenthesis for the introduction of a nesting level can be used, refer to section 8.4.3.

- ☞ **The function MUL for multiple multiplication of numeric operands and for the multiplication of time values and address is described in section 12.1.**

Allowed data types: ANY_NUM, refer to section 7.2.3.

Influencing the CR: Further processing, refer to section 8.3.

Example:

```
LD    24          (* Unsigned *)
MUL   20
ST    uiVar
```

“uiVar” and the CR contains the value “480” after multiplication . The variable “uiVar” may not be of data type “SINT” or “USINT” as this cannot store the result. Instead, a data type with larger data width must be selected, e.g. “UINT”.

Division: DIV

The contents of the working register CR (dividend) are divided by the value of the specified operands (divisor) and the result (quotient) is stored in the CR. The operand value is not changed.

On the operator DIV, the opening parenthesis for the introduction of a nesting level can be used, refer to section 8.4.3.

- ☞ **The function DIV for multiple division of numeric operands and for the division of time values and address is described in section 12.1.**

Allowed data types: ANY_NUM, refer to section 7.2.3.

Influencing the CR: Further processing, refer to section 8.3.

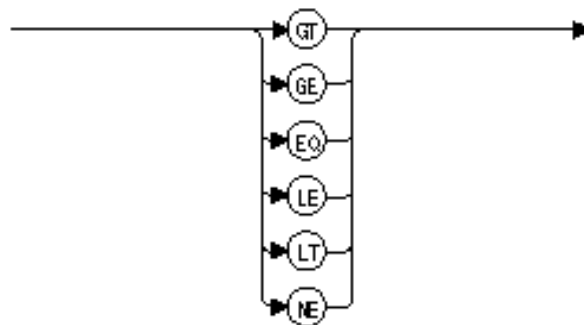
Example:

```
LD    24          (* Unsigned *)
DIV   10
ST    iVar
```

If “iVar” is a real variable (ANY_REAL), “iVar” and the CR contain after the division a value of “2.4”. In case of an interger variable (ANY_INT), the value would be “2”. The after decimal positions are thus cut off.

8.5.5 Comparison operators– GT, GE, EQ, LE, LT, NE

With the help of boolean checks of operators, i.e. checking whether a condition is met (TRUE) or not (FALSE), the program execution can be controlled. The comparison operators are e.g. combined with jump or call operators, refer to sections 8.5.6 and 8.5.7.



Syntax graph of the comparison operators

The following comparison operators are available:

GT	greater than
GE	greater than or equal
EQ	equal to
LE	less than or equal
LT	less than
NE	not equal

The specified operand value is compared with the contents of the working register CR. The operand value was subtracted from the value of the working register. The satisfied condition is indicated through the boolean state TRUE in the CR; FALSE signals that the condition was not fulfilled. The original contents of the CR are overwritten. The operand value is not changed.

During the comparison, a type conversion of the working register CR takes place: After the comparison, the CR is of the data type BOOL.

The boolean result of a comparison operation can be used as condition for a function block-call, a jump to a label, other logical linking operations or for a reverse jump out of a POU to the higher-level structure level.

Allowed data types: ANY_INT, ANY_BIT, ANY_DATE, STRING, TIME, refer to section 7.2.3.

Influencing the CR: Further processing, refer to section 8.3.

Example:

```

M1:  LD    iResult
      ADD  iOperand1
      ST   iResult      (* calculation result *)
      LE   100          (* Comparison CR <= 100... *)
      JMPC M1           (* ...then jump to M1 *)
      JMP  M3           (* ...otherwise jump to M3 *)

```

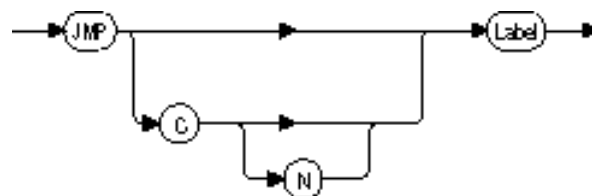
The result of the addition is saved in the variables “iResult” and in the CR. If the operands of the addition and of result are e.g. of the data type “INT”, the CR is also assigned the data type “INT”. The comparison operation following this, saves the boolean comparison result “TRUE” or “FALSE” in the CR. The data type of the CR changes to “BOOL”. Due to this, the subsequent jump instructions “JMPC” and “JMP” are in the position to evaluate the boolean value in the CR.

8.5.6 Jump operators – JMP, JMPC, JMPCN

With the help of jump statements, one can branch to a jump destination. The jump destination must always have a sequence beginning marked by the label (refer to section 8.4.2).

After a jump, the state of the working register CR is undefined and must be redefined with a load instruction.

A jump is possible only within a POU.



Syntax graph of program jumps

Unconditional jump: JMP

The program is set to the position which is specified as the jump location.

Allowed data types: Non-relevant.

Influencing the CR: Undefined, refer to section 8.3.

Example:

```

M1: LD    iANY
    ...
    JMP   M3          (* unconditional jump *)

M2: ...

M3: LD    xANY
    ...
    JMP   M3          (* unconditional jump *)

```

The unconditional jump to the label “M3” has the effect that to start with, the instruction rows between the jump statement “JMP” and the label “M3” are not executed. With another jump statement to label “M2”, the above-mentioned instruction rows are executed at a later point of time.

Conditional jumps: JMPC, JMPCN

Conditional jump depending upon the boolean contents of the working register CR.

JMPC: Jump when TRUE

If the CR contains the value TRUE, the jump is executed and the program is continued further from the jump destination.

If the CR contains the value FALSE, no jump is executed. The program is continued further with the statement following the jump instruction.

JMPCN: Jump when FALSE

If the CR contains the value FALSE, the jump is executed and the program proceeds further from the jump destination.

If the CR contains the value TRUE, no jump is executed. The program is continued further with the statement following the jump instruction.

Allowed data types: Non-relevant.

Influencing the CR: Undefined in case the jump is executed. Unchanged in case the processing continues with the next statement. Also refer to section 8.3.

Example:

```
M1 : LD    %Q3.5
      JMPC M1
```

The example evaluates the state at the (physical) output 3.5. If there is a signal available at the output, a jump to "M1" takes place. Only when no signal is available at "Q3.5", the program processing is continued after the jump statement.

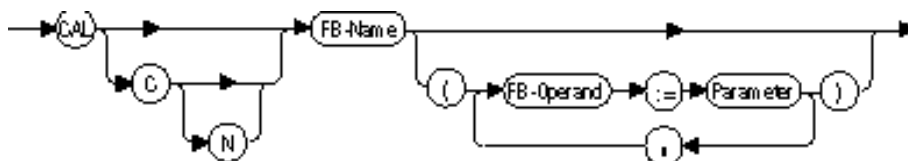
CAUTION! The program processing is not continued through the loop. If the signal lies for a period longer than the maximum program cycle time at "Q3.5", the controller switches into the STOP mode.

8.5.7 Call of function blocks – CAL, CALC, CALCN

As in the case of jump statements, one can branch to a jump destination. The jump destination is here the name of the FB instance.

After a FB call, the state of the working register CR is undefined and must be redefined with a load instruction.

Further information concerning FB calls can be found in the sections 6.5.3 and 6.5.4.



Syntax graph of FB calls

Unconditional calls: CAL

The program is continued further in the function block which is specified as operand. The unconditional call may be programmed only after completion of an IL sequence and is not allowed within the parenthesis modifiers.

Allowed data types: Non-relevant.

Influencing the CR: Undefined, refer to section 8.3.

The input parameters of the function block can be “given” with the call in the parenthesis – by separating the parameters with commas. The actual parameters are assigned using “:=” to the formal parameters.

The second method for the parameter transfer is the initialization before the call by a combination of load (LD) and assignment statements (ST). In case of these methods, the FB call takes place without parenthesis.

Example, method 1:

```
CAL    CTU_1 (RESET:=%IX3.6, PV:=Limit, CU:=_1S2)
```

Example, method 2:

```
LD     %IX3.6
ST     CTU_1.RESET
LDN    Limit
ST     CTU_1.PV
LD     _1S2
ST     CTU_1.CU
CAL    CTU_1
```

Conditional call: CALC, CALCN

Conditional FB call depending upon the boolean contents of the working register CR.

CALC: Call when TRUE

If the CR contains the value TRUE, the as operand specified FB (instance) is called up, in which the program processing will be continued.

If the CR contains the value FALSE, no call is executed. The program is continued further with the statement following the call instruction.

CALCN: Call when FALSE

If the CR contains the value FALSE, the as operand specified FB (instance) is called up, in which the program processing will be continued.

If the CR contains the value TRUE, no call is executed. The program is continued further with the statement following the call instruction.

Allowed data types: Non-relevant.

Influencing the CR: Undefined in case the call is executed. Unchanged in case the processing continues with the next statement. Also refer to section 8.3.

Example:

```
LD      bCounterReset
CALCN  CTU_2 (RESET:=%IX3.2, PV:=Limit, CU:=_5S1)
```

In the example, the FB instance “CTU_2” may only be called up when the boolean variable “bCounterReset” contains the value “FALSE”.

8.5.8 Call of functions

Allowed data types: Non-relevant.

Influencing the CR: Further processing, refer to section 8.3. The CR is occupied with the function value in the function.

Unconditional call

Functions are called up without specify an operator. The input parameters are transferred directly. Due to this, no conditional calls can be implemented.

An option allows to transfer the initial input parameters from the CR of the earlier executed instruction without further specifications.

Example for the use of the CR for the first parameter in case of a function call:

```
LD      2#10010110  (* 1st Param. is loaded
                    into the CR *)
SHL     2            (* 2nd Param. in the call *)
ST      Links       (* Function value in the CR *)
```

In the example, the function SHL is called up. The first parameter is transferred without assignment but only with the load instruction (LD) – with the help of the CR. The second parameter is specified directly in the function call. The function value (return value of the function) is written by the function in the CR. Thus, the function value can be used directly after the instruction row of the function call.

Function value

The return value of the function (function value) is written after the function call in the CR for further processing.

Within the called function, the return value is generated by writing the CR to the function name.

Example:

```
ST      FUN_Name
```

Formal parameter

In contrast to the programming language ST, formal parameters may basically not be specified in the call. Here, the order of the input parameters must definitely followed.

Further information concerning function calls can be found in the sections 6.5.3 and 6.5.5.

Examples

Example 1:

```

LD    szString    (* String variable *)
MID   3, 2        (* Function call +
                  Parameters *)
ST    szNew       (* CR = Function value *)

```

In the example, the standard function “MID” (refer to section 12.1) is called up. This function requires three input parameters. The first parameter is written with the load instruction “LD” in the CR. The other two parameters are specified – delimited by comma – after the function call. After the function call, the function value from the CR is assigned to the variable “szNew” using the instruction “ST”.

Example 2:

```

MID   szString, 3, 2
ST    szNew

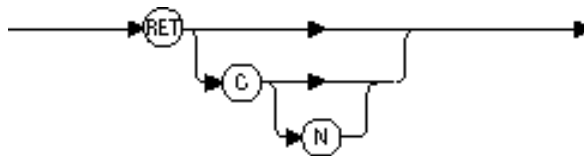
```

This variation shows the same function call as in example 1, with the difference that here, all three parameters are transferred in the call.

8.5.9 Return jump – RET, RETC, RETCN

The return jump instructions result in a return to the calling POU. With the return jump to a POU, the calling POU is continued at the interrupted position.

After a return jump, the state of the working register CR remains unchanged.



Syntax graph of return jumps

Unconditional return jump: RET

The instruction “RET” results in an unconditional return jump to the calling POU. In case of functions, the current function value is entered in the CR.

Allowed data types: No operand available.

Influencing the CR:

Return jump out of function block: Leave unchanged.

Return jump out of function: The current function value is stored in the CR. Also refer to section 8.3.

Example:

```

LD    bError
JMPCN M4
RET                                     (* unconditional return *)
M4 :

```

The example contains an extract from a function or a function block. If the variable "bError" contains the boolean value "FALSE", the program processing is continued from the label "M4". In case "bError" contains the value "TRUE", the return jump to the calling POU follows. The same example can also be realized differently with the help of a conditional return jump, refer to the example below.

Conditional return jump: RETC, RETCN

Conditional return jump depending upon the boolean contents of the working register CR.

RETC: Return when TRUE

If the CR contains the value TRUE, the return jump to the higher-level POU follows.

If the CR contains the value FALSE, no return jump is executed. The program is continued further with the statement following the return jump instruction.

RETCN: Return when FALSE

If the CR contains the value FALSE, the return jump to the higher-level POU follows.

If the CR contains the value TRUE, no return jump is executed. The program is continued further with the statement following the reverse jump instruction.

Allowed data types: No operand available.

Influencing the CR: Leave unchanged, refer to section 8.3.

In comparison to the unconditional return jump "RET", in case of a conditional returns of a function, the CR is **not** overwritten by the function value!

Example:

```
LD    bError
RETC
```

In the example, the boolean variable "bError" is evaluated. In case "bError" contains the value "TRUE", the return to the calling POU follows. Should the variable "bError" contain the boolean value "FALSE", the program processing is continued after the statement "RETC".

9 Programming language Structured Text (ST)

The Structured Text (ST) is a textual higher programming language. In comparison to machine-like IL, ST is a programming language, in which extensive language constructs allow a very compact formulation of the programming task.

An ST program consists of instructions. In an instruction values are computed and assigned, modules are called up and leaved, and command flow is controlled.

ST offers the advantage of implementing an open program structure. The disadvantage of this language lies in its lower efficiency. The programs are slower and longer.

Please pay attention to the programming examples concerning the ST in the WinSPS help, chapter "Introducing WinSPS".

9.1 Expressions, operands and operators

As already mentioned, an ST program consists of instructions. An instruction contains an expression (partial instruction), using which a result is built. Here, complex links can be generated with the help of operands and operators.

Operands

All possible operands are listed in the following table:

Operand	Examples
Literals refer to section 7.1.3	15 'ABC' t#5m_15s
Variables refer to section 7.3	Valve_1 aMeasValue[1,0] Person.szName
Function calls refer to section 6.5.5	LEN (CONCAT ('to', ' gether'))

Operators

The operands can be linked to each other through operators. The following table shows all operators of the programming language ST. The precedence during the processing is shown in the decreasing order of precedence; i.e. the "Parenthesis" have the highest priority. If an expression has more than one operator, the order of precedence is to be taken into account. Operators with higher priority are processed before the operators with lower priority. In case of equal priority, the processing takes place from left to right.

Operator	Explanation	Example
()	Parenthesis	(2+3)*(4+4) Result: 40 Without parenthesis: 18
	Function call	CONCAT ('TO','GETHER') Result: 'TOGETHER'
**	Exponentiation	2**3 Result: 8
-	Negation	-12 Result: -12
NOT	Complement	NOT FALSE Result: TRUE
*	Multiplication	3 * 4 Result: 12
/	Division	12 / 6 Result: 2
MOD	Modulo (Remainder of a division)	23 MOD 6 Result: 5
+	Addition	3 + 5 Result: 8
-	Subtraction	5 - 7 Result: -2
< > <= >=	Comparison: Less than Greater than Less than or equal Greater than or equal	45 > 66 Result: FALSE
=	Equality	T#34h = T#6d7h Result: FALSE
<>	Inequality	10 <> 16#A Result: FALSE
& AND	Boolean AND	TRUE AND FALSE Result: FALSE
XOR	Boolean Exclusive OR	TRUE XOR FALSE Result: TRUE
OR	Boolean OR	TRUE OR FALSE Result: TRUE

Function calls as operators

Calls of function blocks represent a closed instruction, refer to section 9.2. In contrast to that, functions (FUN) are called up within a partial instruction and are counted among the expressions. Further information and examples concerning function calls can be found in section 6.5.5.

9.2 Instructions

An ST program consists of instructions. Instructions are terminated with the semicolon. As a result, even multiple instructions may be given in a row. In comparison to IL, the end of line (line break) does not separate the instructions. It is interpreted as a space character.

All ST instructions are listed in the following table:

Explanation	Key word	Examples	Refer to section
Assignment	:=	a := 5;	9.2.1
Calling of an function block		FBName (Para1:=5, Para_n := 10);	9.2.2
Return jump	RETURN	RETURN;	9.2.3
Selection	IF	IF a<b THEN c:=1; ELSIF a=b THEN c:=2; ELSE c:=3; END_IF;	9.2.5
Multi-selection	CASE	CASE n OF 2: p:=4; 3: p:=p+3; 5..10: p:=100-p(p-q); ELSE p:= MAX(a,b); END_CASE;	9.2.6
Iterations	FOR	FOR a:=1 TO 100 BY 2 DO b[a/2]:=a; END_FOR;	9.2.7
	WHILE	WHILE y > 1 DO y:= y-2; END_WHILE;	9.2.8
	REPEAT	REPEAT a:=a+b; UNTIL a<100 END_REPEAT;	9.2.9
End of loop	EXIT	EXIT;	9.2.11
Empty statement	;	::	

Structured programming

In ST, there is no GOTO statement. As a result, structured programming gets enforced. Jumps can be reproduced through appropriate IF or CASE statements.

The option of allowing multiple statements in a row leads to confusing program structure. Such programs, which do not comply with structured programming, are also termed as "Spaghetti Code". Instead e.g. in case of nesting, it is possible to use indenting so that the processing structure can as a result be quickly recognized. An example for this indenting is quite distinctly illustrated in table above in the example of the IF statement.

9.2.1 Assignment

An assignment transfers the result of the evaluation of an expression to a variable. The assignment copies the value existing on the right side of the equivalence sign ':' into the variable on the left side.


In the assignments, single element as well as multiple element variables can be used.


Examples:

```
(* Assignment: Variable a is assigned the value 5 *)  
a := 5;
```

```
(* Two assignments in one row *)  
b[1] := a**2; d := b[2];
```

```
(* Assignment with a function call *)  
d := REAL_TO_INT (c);
```


 **The variable on the left side must be of the same type as the currently assigned value.**

 **In case of an assignment, do not forget to enter a colon before the equivalence sign ":=". A standalone equivalence sign "=" is used for comparison operations.**

9.2.2 Call of an function block

In ST, a function block (FB) is called up using its name and its formal parameters existing in the parenthesis. In this case, the sequence is not significant. The value of the actual parameters is assigned to the formal parameters. Output parameters can similarly be assigned to the formal parameters in the call. Input and output parameters are then separated from each other with the character "|". The assignment can take place directly in the call through corresponding statements.

While calling up a function block, not necessary parameters may be left out.

 **VAR_IN_OUT parameters may not be left out. During compilation, WinSPS generates an error message in case VAR_IN_OUT parameters are left out.**

Detailed information concerning the call up interface of FBs is discussed in section 6.5.4.

Example for the assignment of input parameters in the call:

```
FB_InstanceName (Para1 := 27, Para2 := uiVar);
```

Example for the assignment of input parameters before the call:

```
FB_InstanceName.Para1 := 27;  
FB_InstanceName.Para2 := uiVar;  
FB_InstanceName ();
```

Example for the evaluation of an output parameter:

```
Erg := FB_InstanceName.ParaOut;
```

Example for the assignment of input and output parameters in the call:

```
FB_InstanceName (Para1 := 27, Para2 := uiVar |  
                Erg := ParaOut);
```

9.2.3 Return jump – RETURN

A function or a function block can be left with the RETURN statement, even before completion. In case of functions, the function value must already be assigned at this point of time. If the function value is not assigned, it gets the default value of its data type, refer to section 7.2.1.

Example:

```
IF a > b THEN RETURN;  
END_IF;
```

9.2.4 Conditional execution

Often there is a need to make the execution of specific statements dependent on a condition. With a conditional expression the program flow can be controlled. There are two groups of control structures:

- **Selections:**
Selection (IF) and Multi-selection (CASE)
- **Iterations:**
FOR, WHILE and REPEAT loop, refer to section 9.2.10.

Selection statements are used when the statements cannot be executed in all events, instead, they are executed dependent upon one or more conditions.

Iterations or loops allow computing of specific program steps multiple times. In comparison to jumps (GOTO), loops are useful for the structured programming.

9.2.5 Selection, – IF


With the IF statement block, the program flow can be controlled depending upon an expression that can be evaluated. The program flow is dependent upon the boolean result of this expression.

- Conditional expression as well as further nesting with **ELSE** and **ELSIF** are indicated within the keywords **IF** and **END_IF**.
- If the expression given after the **IF** is TRUE, the statements in the **THEN** part are executed.
- If the expression is not true (FALSE), the statements in the **ELSE** part are executed or the expression after the associated **ELSIF** is checked.
- If no ELSE or ELSIF part is available, or in case, no condition is fulfilled, the processing is continued after **END_IF**.

Syntax

The following table shows the syntax and the nesting option of the branching:

IF Expression THEN Statement_block;	Execution of statement blocks only when the condition is fulfilled.
ELSIF Expression THEN Statement_block;	Execution only when the expressions are not fulfilled and the ELSIF expression is fulfilled. This partial statement can be omitted or can be repeated as many times as may be necessary.
ELSE Statement_block;	Execution only when all previous expressions are not fulfilled. The ELSE part is optional.
END_IF ;	Closing the IF statement.

 **Kindly keep in mind that every statement and the keyword END_IF are to be followed by a semicolon as shown in the table above.**

Example:

```
IF a < b THEN
  c := 1;
ELSIF a = b THEN
  c := 2;
ELSE
  c := 3;
END_IF;
```

In the example, it is checked whether “a” is less than “b”. If this is the case (condition = TRUE), the statement “c := 1” is then executed. Afterwards, the program programming is continued after END_IF.

However, if “a” is not less than “b” (condition = FALSE), the next condition after ELSIF is checked. If “a” is equal to “b”, the statement “c := 2” is executed. Afterwards, the program programming is continued after END_IF.

Only when “a < b” and “a = b” are not fulfilled, the statement after ELSE “c := 3” is executed. This is the case when “a > b”.

9.2.6 Multi-selection– CASE


With the nesting of an IF ELSE statement, the number of options can practically be increased to any extent. The disadvantage in that case is that the clarity decreases with increasing nesting depth.

The CASE statement is a multi-selection.

- Condition and statement blocks are specified within the keywords **CASE** and **END_CASE**.
- For condition, an integer (signed) variable, which is compared in the CASE statement blocks, refer to syntax: "case_value :".
- The evaluation of the expression is started with the keyword **OF**.
- Depending upon the condition expression, the assignments associated with this value are executed.
- Subsequently, the program control switches to the first statement after **END_CASE**.
- Optionally before the end, an additional statement block with the keyword **ELSE** can be realized. This is processed only when no branching condition is fulfilled.
- A CASE statement block can be processed also for multiple values of the condition variables. The individual values are separated from each other by comma, refer to the example below: "5, 9 :".
- Also an integer range can be specified. In this case, the upper limit is separated from the lower limit by two dots, refer to the example below: "8..10 :".

Syntax

```
CASE Expression OF  
    case_value : Statement block;  
    ELSE statement block;  
END_CASE;
```

 **Kindly keep in mind that every statement and the keyword END_CASE are to be followed by a semicolon. Every CASE value or range of values is separated by a colon ":" from the statement block.**

Example:

```

CASE a OF
  1 :      b := 1;
  2 :      b := 2;
  5,9 :    b := 3;
  8..10 :  b := 4;
  ELSE     b := 0;
END_CASE;

```

In the example, the variable “a” is checked as a conditional expression. Depending upon the contents of these variables, “b” is assigned the following value:

a	1	2	5	9	8	10	else
b	1	2	3	3	4	4	0

In the shown example, the CASE value “9” appears in two statement blocks “5,9” and “8..10”. A CASE selection is processed from top to bottom. As soon as a statement block is processed due to matching of the condition, the following conditions are **not** checked any more, and the program control jumps to the END_CASE statement. In case “a” has the value 9, the statement “b := 3” is executed. The statement “b := 4” is not executed, although even here the condition could be fulfilled.


 **Therefore, take care that the same CASE values do not appear in different statement blocks.**

9.2.7 FOR loop

Iteration statements or loops allow execution of specific program steps multiple times. The programming language ST has three types of loops FOR, WHILE and REPEAT.

The FOR loop is also termed as counting loop.

- The FOR loop is delimited by the keywords **FOR** and **END_FOR**.
- The FOR loop allows a specific number of repetitions by using an integer **control variable** (running index).
- The control variable is set at an **initial value** and after completion of every loop, it is increased by a specific **increment value** or decreased in case of a negative value.
- The break condition is specified through an **end value**, which the control variable must run over by increasing or decreasing.
- The specification of the increment value “BY 1” is optional. The standard value is 1.
- Loops can be prematurely terminated with the EXIT statement, refer to 9.2.11.

 **The values for initial value, end value, running index and increment value must not be changed within the FOR loop (e.g. by assignment). Otherwise, errors will occur.**

Syntax

```
FOR Control_variable := Initial_value TO End_value BY Increment DO
    Statement_block;
END_FOR;
```

Example 1:

```
FOR i := 0 TO 100 BY 2 DO (* increment value = 2 *)
    Field[i] := 10 * i;
END_FOR;
```

In the first example, the control variable “i” is set to the initial value 0. End value is 100, the incremental is 2. With this, after each completion of the loop “i” is increased by 2. Moreover, “i” is used within the loop in order to initialize the array “Field”. The following enumeration shows the contents of the array element for the first 4 and the last run:

Passage: 1	2	3	4	51
i := 0	i := 2	i := 4	i := 6	i := 100
Field[0] := 0	Field[2] := 20	Field[4] := 40	Field[6] := 60	Field[98] := 1000

When the control variable “i” reaches the value 100, the loop is run through for the last time. However, the header of the loop is subsequently evaluated once more and “i” is set to the value 102, but with this, the end value is crossed over and subsequently the program control jumps to the loop end.

Example 2:

```
FOR k := -20 TO 0 DO (* increment value = 1 *)
    x := 20 + k;
    Field[x] := k;
END_FOR;
```

In this FOR loop, the initial value is a negative integer. Since no increment value is specified, the standard value 1 is used. The loop is executed 21 times.

Example 3:

```
FOR j := 50 TO 1 BY -1 DO (* decremental *)
    Field[j] := j MOD 5;
END_FOR;
```

In this example, a negative increment value is selected (decrement value). The control variable is as a result counted backwards. In case of decrement values, the initial value must be greater than the end value.

An example for the initialization of a multidimensional array is shown in section 7.3.6 with the help of a FOR loop.

9.2.8 WHILE loop

In case of the WHILE loop, a boolean expression is evaluated.

- The loop is delimited by the keywords **WHILE** and **END_WHILE**.
- So long as the evaluation of the expression produces TRUE, the statements in the loop are executed.
- As soon as the expression produces FALSE, the program processing continues after the loop.
- The statement block begins after the keyword **DO** and ends with **END_WHILE**.
- Since the expression in the header of the loop is checked, a **deflecting** loop can be formed i.e. by appropriate formulation of the condition, it can be achieved that the statements in the loop not executed even a single time, refer to section 9.2.10.
- Loops can be prematurely terminated with the EXIT statement, refer to section 9.2.11.

Syntax

```
WHILE Expression DO
    Statement_block;
END_WHILE;
```

On translation, the syntax means: So long as the expression is satisfied, repeat the statement block.

Example:

```
i := 1;
WHILE i < 10 DO (* so long as i is less than 10,
                repeat *)
    Field[i] := 10 * i;
    i := i + 2;
END_WHILE;
```

So long as the contents of the variable "i" are less than 10, the loop is executed. Take care that the value of "i" is increased within the loop so that no endless loop exists. The loop in the example is executed 5 times.

9.2.9 REPEAT loop

Even in case of the REPEAT loop, a boolean expression is evaluated.

- The loop is enclosed by the keywords **REPEAT** and **END_REPEAT**.
- The evaluation of the break condition takes place at the end of the statement block so that the loop is executed at least once (**non-deflecting**, refer to section 9.2.10).
- The expression for the break condition is set directly before **END_REPEAT** and started by the keyword **UNTIL**.
- So long as the evaluation of the expression produces FALSE, the statements in the loop are executed.
- As soon as the expression produces TRUE, the program processing continues after the loop.

- The statement block begins after the keyword **REPEAT** and ends with **UNTIL**.
- Loops can be prematurely terminated with the EXIT statement, refer to section 9.2.11.

Syntax

```
REPEAT  
    Statement_block;  
UNTIL Expression  
END_REPEAT;
```

On translation, the syntax means: Repeat the statement block till the expression is true.

Example:

```
i := 1;  
REPEAT  
    Field[i] := 10 * i;  
    i := i + 2;  
UNTIL i > 10      (* repeat till i is larger than 10 *)  
END_REPEAT;
```

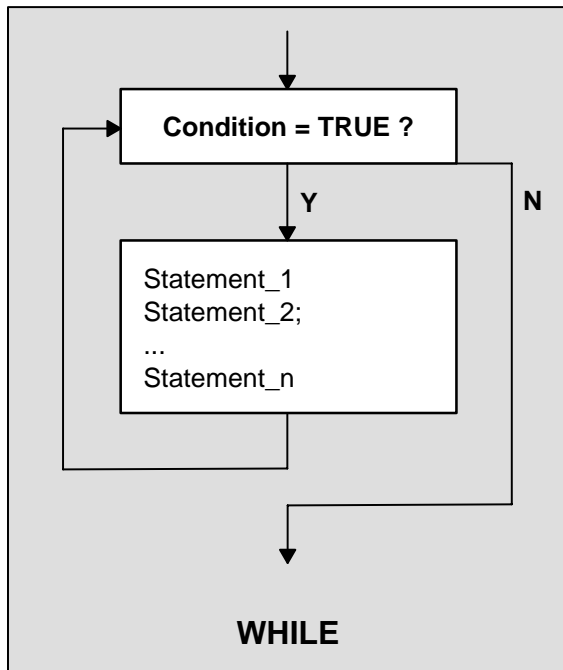
The loop is executed till the contents of the variable “i” become larger than 10. Take care that the value of “i” is increased within the loop so that no endless loop exists. The loop in the example is executed 5 times.

9.2.10 Deflecting and non-deflecting loops

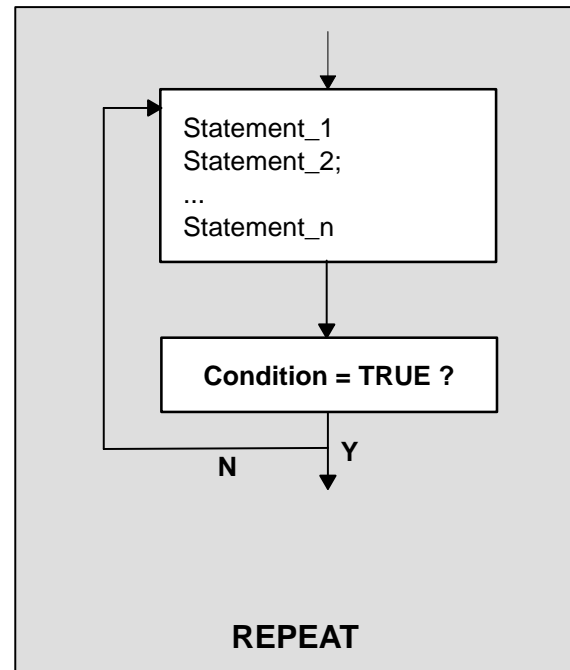
Using the WHILE or FOR loops – through appropriate formulation of the loop condition – the processing of the statements within the loop can be deflected. i.e. the loop can be skipped.

The statements of the REPEAT loop however are in any case executed at least once (non-deflecting), independent of whether the condition is fulfilled or not.

The following image shows the different behavior with the example of the WHILE and REPEAT loops.



Control flow of the deflecting loop



Control flow of the non deflecting loop

9.2.11 Premature loop end – EXIT

With the keyword EXIT, a loop can be exited prematurely. The program processing is continued after the loop. In case of nested loops, only the innermost repetitive statement is exited.

Example:

```
FOR i := 2 TO 20 BY 2
  FOR j := 0 TO 9
    IF bError THEN EXIT; END_IF; (* condition for exiting the inner loop *)
    Array_2[i,j] := i / 2 + j;
  END_FOR;
  Array_1[i] := i * 5; (* continuation after EXIT break *)
END_FOR;
```

10 Check load and test program

What distinguishes the modules (POU) of the IEC 61131-3 from most of the classical programming languages is that they cannot be loaded directly. After the entry of a program, the following processing steps must be followed:

- Compile module = Translate into program code
- Link all modules = Link into an integrated program
- Afterwards, the program is loaded in the controller

The processing steps can be carried out independent of each other. In order to accelerate the loading process, after each program change, WinSPS identifies which modules are to be compiled and which are not to be compiled.

Program tracking and data observation in the WinSPS Monitor are useful for commissioning and error detection, refer to section 10.5.

10.1 Check / compile module

This function allows checking of an individual module and must be executed after entering all the rows of a POU and after every change.

Since a POU forms a closed unit in itself, the compiler can translate this independent of other program parts into a program code that can be run (Compilation). Thus, all modules can be converted into the program code gradually.

This allows a step by step program development. Partial programs can be developed and tested independently.

In the POU, entire call interface data is known with the declaration in VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT and VAR_EXTERNAL. As a result, all modules can be subsequently linked quickly into an integrated program.

WinSPS call

Menu *File* ► *Check / Compile Module*

Keyboard shortcut <Ctrl> + <Alt> + <C>

Tool bar



Compiler messages

In the IEC editor, the lower window area is used for displaying errors after testing, refer to section 5.3.

 **Only modules, which are compiled error-free, can be loaded in the controller.**

10.2 Link all modules – Create new project

All modules are linked into an entire program. The “linker” identifies from the call structure all those modules (POU), which belong to a project. Therefore, POU and file names may not appear more than once in a project.

Example:

File name: **MODULE.IL** POU Name: **Module_1**
 File name: **MODULE.ST** POU Name: **Module_2**


Though the POU names of both the modules in this example are different, yet the filenames within a project may not be identical; not even when they are used for different programming languages as shown here.

WinSPS call

Menu *File* ► *Create new project*
 Keyboard shortcut <Ctrl> + <Alt> + <G>

Executable program files

Before executing the linker function, all modules are compiled automatically. Subsequently, executable program files (FC) are generated from the modules (POU) in the ASCII format. For the data of a POU, modules are generated. The names of this program and data modules are independent of the project specifications in the symbol file, refer to section 10.3.

 **In case calls of IEC modules out of the classical programming languages are to be realized, the calls must be made before the generation, also refer to section 11.2.**

Identification of the main program

In the symbol file, the module for the main program must be entered at the position for “OM1”. Here, differentiation is made between the pure programming as per IEC 61131-3 and the mixed programming with the classical program parts:

- **Pure IEC programming:**

The file name of the PROGRAM POU is entered by WinSPS automatically at the position of the OM1. If this entry is missing, the entry can be made manually, e.g..

```
OM1,R IEC_PROG ; Cyclic program processing
```

- **Mixed programming**

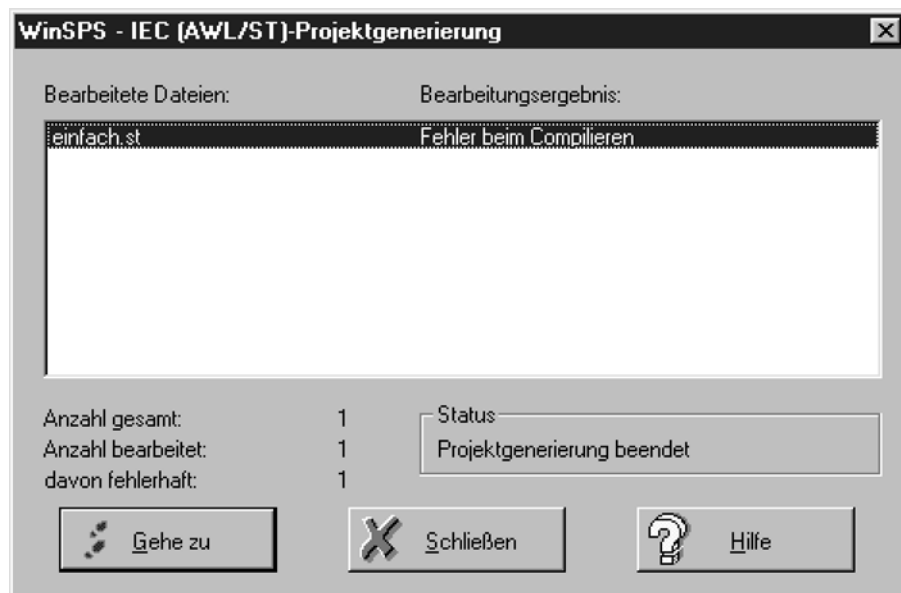
The file name for the organization module OM1 must in any event be entered by hand in the symbol file, e.g.

```
OM1,R OM1 ; Cyclic program processing
```

Linker messages

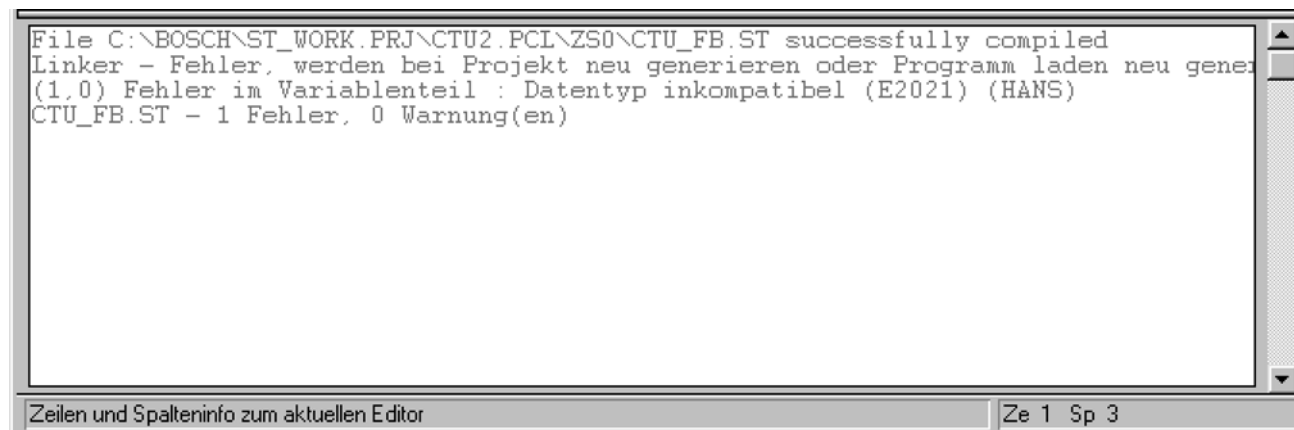
Error messages are mostly attributed to the inconsistencies of the interfaces, global and external data or to the crossing of the physical address area.

Error messages are outputted should errors be detected during compilation or linking. Using the button "Go to", one can jump to the error position within the module.



Compiler error during "Create new project"

For a module, an error message appears in the lower editor window. Rows and character position can however be uniquely identified and indicated by the linker only in exceptional cases.




Error message of the linker (Create new project)

10.3 Project specifications in the symbol file

With the creation of a project are the ASCII rows of the IEC files converted into executable control instructions and set into the program modules. Data modules are generated, in which all variables are stored. Various necessary symbols are entered in the symbol file. The following modules are generated by WinSPS automatically:

- Program files **Name.PXO**, where the name corresponds to the file name of the IL or ST file
- Data module **IM512.PXD** to **IM1023.PXD**, as well as **IM0.PXD**

IM0.PXD is generated only in case of pure IEC programming – without classical program parts – for the data of the PROGRAM POU.


 **The automatic generation of data modules refers to WinSPS version 3.1 and lower. In future versions of the WinSPS, no data modules are reserved and set up.**

Moreover, the following symbolic identifiers are given in the symbol file:

- Symbols FC512 to FC1023
- Symbols DM512 to DM1023, as well as DM0

The numbering is control dependent and can be adjusted in the symbol file depending upon the application. Two areas are reserved there for the automatically generated symbols, refer to the example given below. The number starts with the number after “Start section” and ends with “End section”. In the symbol file, the limits can be adjusted by changing these numbers when needed. As far as possible, this must take place before the first compilation or linking of the project.

In the data module area, not only symbols but the names for automatically generated data modules IMx.PXD are adjusted. The numbering for DM0 (IM0.PXD) cannot be changed.

 **WinSPS generates symbol names and the associated files only for free numbers within the specified area. If a module generated by the user already exists in this area, WinSPS does not change the entry and the module, instead, continues with the automatic generation with the next free number.**

Example

```
; *** BEGIN ST program module ***
; In this area only the following values are allowed
...
; Start section = 512
; End section = 1023

; *** BEGIN ST data module ***
; In this area only the following values are allowed
...
; Start section = 512
; End section = 1023
```


Identification of the main program

Moreover, in the symbol file the module for the main program must be entered at the position for "OM1", refer to section 10.2.

10.4 Load program and modules

All modules are loaded in the controller. The loading of partial programs or the post-loading of individual modules is not possible. Always the complete program is loaded.

In case of mixed programming with classical program parts, individual non-IEC modules can also be post-loaded.

Only error free compiled and linked modules are loaded. In case modules are not compiled and linked free of errors (regenerate project), the compiler or linker function for this module is called up automatically before loading.

WinSPS call

Menu *Controller* ► *Load*
Keyboard shortcut: <Ctrl> + <Alt> + <L>

10.5 Monitor

On pressing the button "IEC", the monitor for IEC modules is shown.



Since in case of function blocks (FB), every instance has an independent data and program code area, the different instances can be indicated via a selection field, refer to the illustration given below: Selection fields "simple (0)".

In the declaration tables, the current process values of a variable are shown in the column "monitor data".

In future versions of the WinSPS with the accompanying firmware versions for PCL or CL550, in the instructions section, the current process values shall be shown fitting in the current instruction row.

The dividing line between the instructions and the actual values can be shifted towards left and right in order to adjust the indication field individually.

Variable type	Name	Data type	Initial value	Address	Attribute	Monitor data	Comment
VAR	OUTPUT	BOOL		%Q1.7		FALSE	
VAR	INPUT	BOOL		%I0.3		FALSE	

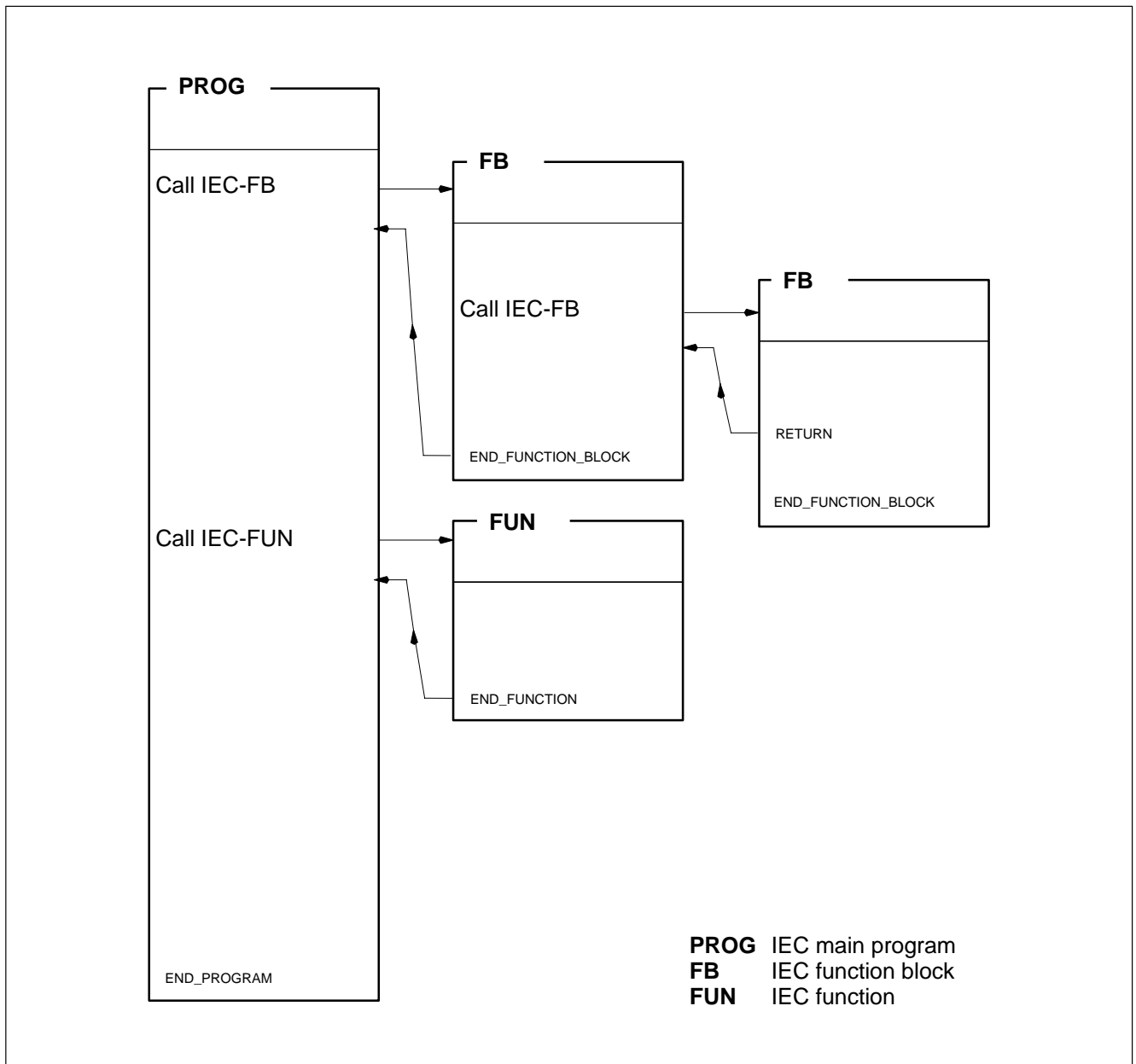
OUTPUT := NOT INPUT;

Monitor detail with the example of the programming language ST

11 Use of IEC modules in the classical programming languages

11.1 Pure IEC programs

The following illustration demonstrates the typical call hierarchy in case of a pure IEC program.



Pure IEC 61131-3 programming without "classical" modules

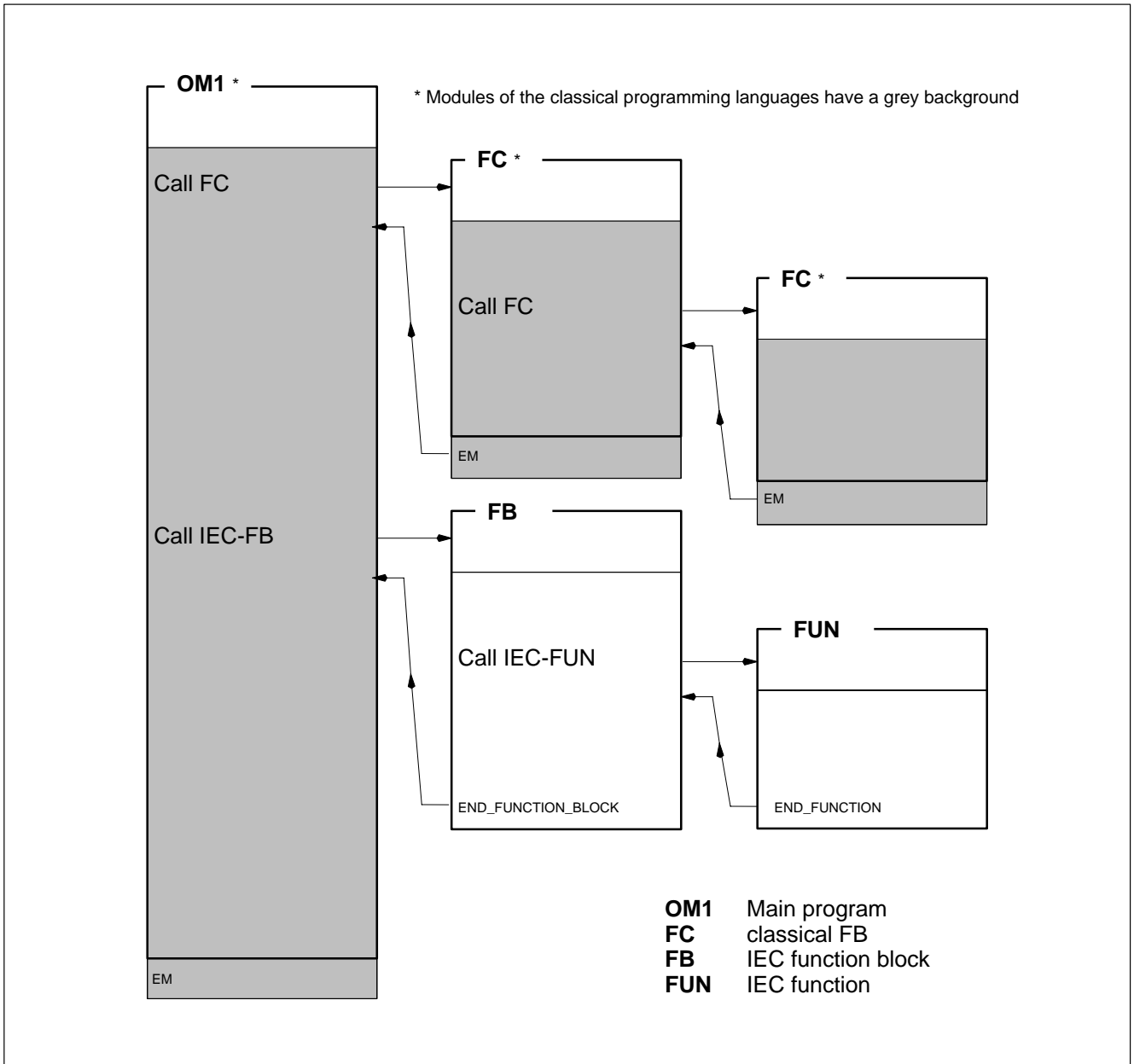
11.2 Mixed programs

In addition to the “pure” IEC programming with the POU types *PROG*, *FB* and *FUN*, Bosch allows the calling of IEC modules from the “classical” programming languages Instruction List (Bosch-IL) and Sequential Function Chart (SFC).

A few conditions and options exist if the classical programming languages are mixed with the IEC programming:

- IEC modules (POU) may be called up from a classical OM (Organisation Module) or FB (Functions Block, Program Module).
- Moreover, IEC modules can be called up in a step action in the programming language SFC or within the PLC instructions for a step, refer to section 11.3.4.
- Presently, only IEC modules of the type `FUNCTION_BLOCK` (FB) can be called up.
- While calling an FB, an instance of the FB is built even in the classical programming environment. The instance building is managed by the user with the help of a WinSPS wizard, refer to section 11.3.
- IEC modules cannot call up any “classical” modules.
- IEC modules can call up other IEC modules (FB and FUN).
- In place of the PROGRAM POU, the organization module `OM1` is used as the main program.
- The module `OM1` must be entered by hand in the symbol file, e.g.
`OM1,R OM1 ; Cyclic program processing`
- Operands from the symbol file can also be used in the IEC modules, or for the call interface from a classical to an IEC module.
- Type definitions, which were entered in the global type editor (refer to 5.5), can also be used in the mixed programs, refer to section 11.4.4.
- In order to allow use of physical addresses (inputs, outputs and markers) in the mixed programming, WinSPS provides a suitable mechanism using the symbol file, refer to section 11.4.1.

The following illustration demonstrates the possible call hierarchy in a mixed program. The example illustrates the call to an IEC function block (FB).



Calling up IEC modules from an "classical" programming language.

11.3 Function block call

Bosch allows the call to IEC POU's of the type FUNCTION_BLOCK (FB) from the classical programming languages Instructions List (Bosch-IL) or within the Sequential Function Chart (SFC) in a step action or the PLC instructions for a step, refer to the example in section 11.3.4.

Entries in the symbol file

In place of the PROGRAM POU, the organization module OM1 is used as main program. This module must be entered by hand in the symbol file, e.g.

```
OM1,R OM1 ; Cyclic program processing
```

All other program modules (FC), which are used in the program, must likewise be entered by hand in the symbol file. IEC modules (function blocks and functions) are basically managed by WinSPS automatically and do not have to be entered by hand in the symbol file.

Instance calls

As in the case of pure IEC programming, here also instances are called up and not the FB itself. Information concerning the instance of a function block can be found in section 6.6.

Multiple working steps following one after another are required for the call:

1. The FB to be called up must be created and compiled free of any errors.
2. In the symbol file, all operands must be declared, which are to be used as actual parameters of the FB call interface, also refer to section 11.4 and 6.5.3.
3. The write cursor must be positioned in a free row of the Bosch-IL module – better in an empty network. At this position, the FB call is inserted.
4. Using the menu function *Edit* ► *Call up parameter list*, the wizard for inserting a POU call is started. Multiple dialog windows appear one after another, refer to section 11.3.1.

With the menu function *File* ► *Create new project*, all calls are checked and converted into an executable program (linked).

11.3.1 Call parameter list – wizard for FB call

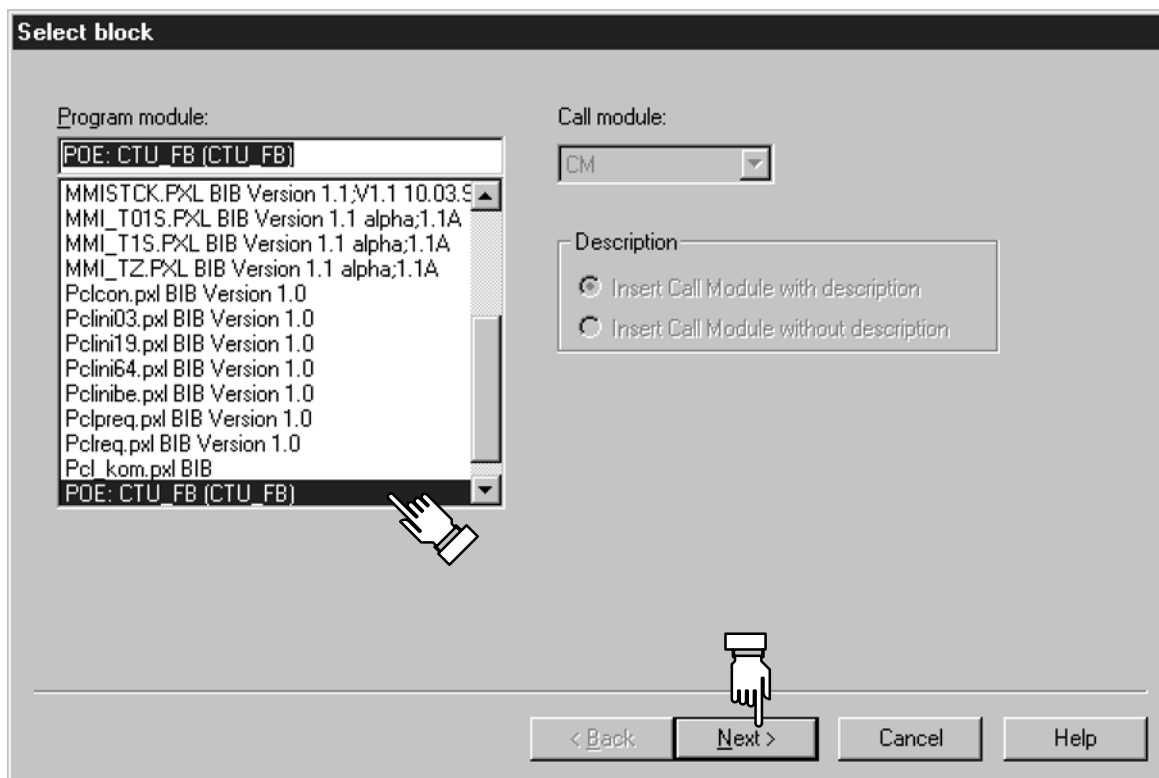
The menu function *Edit* ► *Call up parameter list* or the key combination <Ctrl> + <P> opens a dialog window for the selection of library and IEC modules.

Select block

All classical Bosch function blocks with parameter header as well as the IEC modules are listed. The modules available for selection are sorted in the alphabetical order. The IEC function blocks can be found under the letter “P” for POU. If no FBs are shown, either none exist or are not compiled free of any errors.

The module call always takes place unconditionally (CM).

Select the desired FB from the list and press “Next”.



First dialog window of the wizard for function block calls

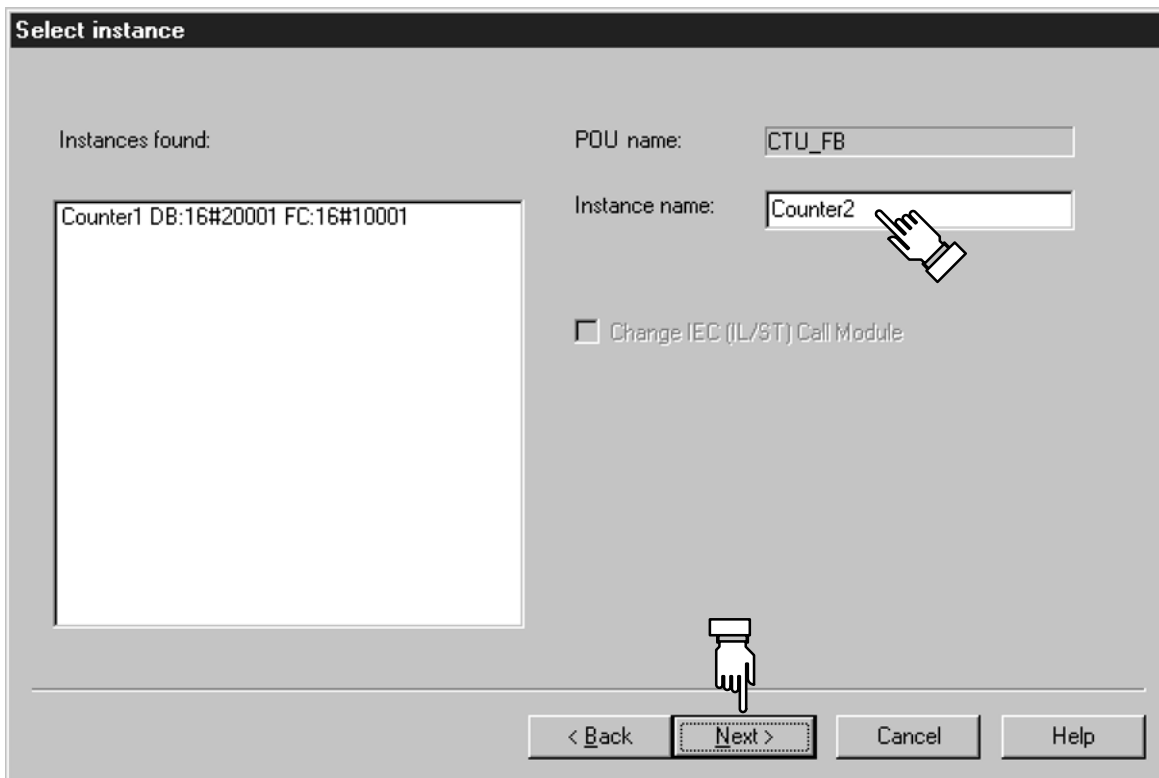
Select instance

Here, already existing or new instances can be selected for the FB call.

Since every instance has its independent program code and data range, the associated program (FC) and data modules (DM) are shown here, also refer to section 10.3.

If an existing instance is to be called up again, this can be selected from the list. A new instance is entered in the input field "Instance name".

Afterwards, press "Next".



Second dialog window for specifying an instance of the function block

Configure input and output parameters

This dialog window is independent of the call interface of the function block. All parameters of the FB call interface are indicated. The names of the variables (formal parameter) and the associated data type of the variable types VAR_IN, VAR_OUT and VAR_IN_OUT are listed.

In the input field "Actual parameter", either absolute values or symbolic operands can be entered.

In case of symbolic operands, these must have already been entered in the symbol file. In contrast to the pure IEC programming, attention must be paid to the uppercase/lowercase letters in case of mixed programming. If you for example enter the symbolic operand "MARKERWORD_8" in the symbol file, you must use this style "MARKERWORD_8" also inside the IEC modules and the call interfaces.

Absolute operands are entered in the classical style, e.g. Q2.0.

Entries in the symbol file for example in the illustration below:

```
I2.0, BOOL      SET1 ;  
I2.1, BOOL      SET2 ;  
I2.2, BOOL      RES ;
```

Configure input and output parameters

Variable type	Name	Data type	Attribute	Actual parameter
VAR_INPUT	Set_1	BOOL		·SET1
VAR_INPUT	Set_2	BOOL		·SET2
VAR_INPUT	Res	BOOL		·RES
VAR_OUTPUT	C1_Max	BOOL		Q2.0

< Back Finish Cancel Help

Last dialog window for the specification of the parameters of the call interface

Afterwards, press "Finish". With this, a module call is generated by WinSPS automatically in the Bosch-IL. Moreover, entries are accepted in the symbol file.



CAUTION

Do not change the automatic entries in the Bosch-IL and the symbol file!

Especially, the comment rows must remain unchanged so that the WinSPS can manage the calls of the IEC modules!

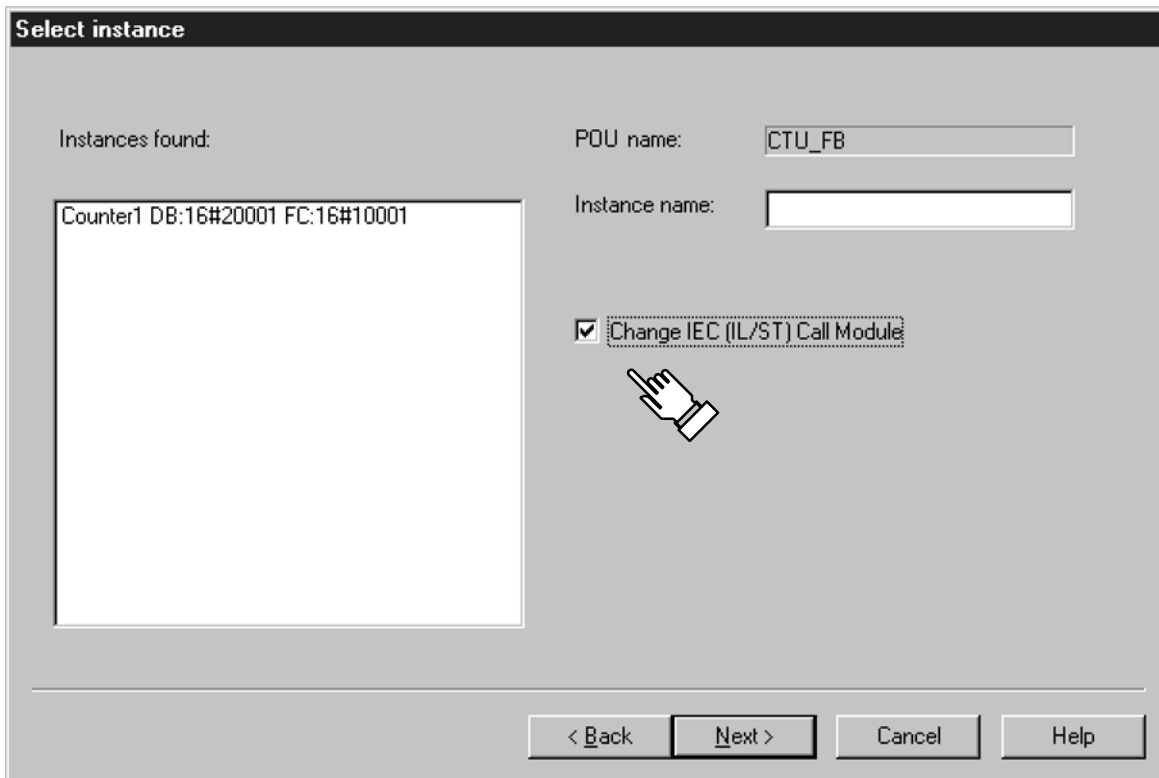
11.3.2 Changing the FB calls

An FB call generated through the wizard can be changed subsequently from the menu function *Edit ► Call up parameter list*.

1. The FB should have been compiled free of any error.
2. In the symbol file, all operands, which are to be used as actual parameters of the FB call interface, must be declared, also refer to section 11.4 and 6.5.3.
3. The write cursor must be positioned within the call to be changed. In the Bosch-IL, an FB call can be identified from the preliminary comment rows:


```
; IEC_FUNCTION_BLOCK: ...
```

 Position the write cursor after the preliminary comment rows, e.g. on a "DEF" statement.
4. Using the menu function *Edit ► Call parameter list* the wizard for calling a POU call is started. Multiple dialog windows appears one after another.
5. In the first dialog window, select the desired function block, as shown in section 11.3.1.
6. In the second dialog window, select the desired instance, or enter a new instance, also refer to section 11.3.1. You must activate the option "Change IEC (IL/ST) call module", otherwise, the call parameters are not changed, instead, a new FB call is inserted. The following illustration shows this dialog window.



In this dialog window, the option "Change IEC (IL/ST) Call module" must be activated

7. In the third dialog window, the parameters of the call interface are edited, also refer to section 11.3.1.

With the menu function *File* ► *Create new project*, all calls are checked and converted (linked) into an executable program.

11.3.3 Deleting the FB calls

In order to delete an FB call from a Bosch-IL, the automatically generated entries must be deleted by hand:

FB and instance

Within the Bosch-IL module, an FB call can be identified from the preliminary comment rows:

```
; IEC_FUNCTION_BLOCK: ...
```

You can find the instance names further below:

```
; IEC_INSTANCE: ...
```

Delete FB call

If for the call, you have created a separate network in the Bosch IL, the network can thus be simply deleted.

In case, other statements in addition to the FB call exist in the network, delete the program rows between the comment markings

```
; IEC_FUNCTION_BLOCK: ...
```

and

```
; END_IEC_FUNCTION_BLOCK
```

(including these rows).

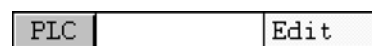
Cleaning up

With the menu function *File* ► *Create new project*, the symbols, which have become redundant, are removed from the symbol file, and with this, the project is updated to the current status.

11.3.4 Call in the Sequential Function Chart

The call to a function block (FUNCTION_BLOCK) is possible in two ways inside the Sequential Function Chart (SFC):

- In a step action
- Within the PLC instructions, in a single step




PLC instructions in a step action



PLC instructions in a single step

PLC instructions are entered in the network "User IL". Position the write cursor in a free IL row, in which the FB call can be inserted. The call takes place as described in section 11.3.

 **Kindly take care that the PLC instructions are executed in a step action or in a single step in each PLC cycle. If the processing should take place dependent upon the step, this can be achieved by the jump statement "JPCI", refer to the example.**

Example:

```
JPCI End4n5 (* Jump when the step is not active *)
```

```
End4n5:
```

The example shows the PLC instructions of a step function for the step "4N5". With the instruction "JPCI End4n5", the PLC instructions are executed only when the step "4N5" is active.

The FB call can be inserted e.g. between the instructions "JPCI End4n5" and "End4n5:".

After the label "End4n5:", instructions can be inserted, which are to be executed in every PLC cycle.

11.4 Symbol file – interface of mixed programming

In order to be able to access common data in a mixed programming, these must be entered in the symbol file. While doing so, take care that reserved areas or symbols are not edited. These areas and symbols are characterised within the symbol file by remarks in the comment rows. In this regard, also refer to the section 10.3.

In contrast to the pure IEC programming, attention must be paid to the uppercase/lowercase letters in case of mixed programming. If you for example enter the symbolic operand "MARKERWORD_8" in the symbol file, you must use this style "MARKERWORD_8" also inside the IEC modules and the call interfaces.

11.4.1 Physical addresses and miscellaneous data

In the IEC 61131-3, physical PLC are standardized through the address operator %, the prefix **I**, **Q**, **M** and the marking of the data width. The address data following this is however manufacturer specific. Bosch has implemented two positions separated by a dot.

In the classical programming, PLC addresses have a different format. The following table contains a few examples of physical addresses in various types of programming:

IEC 61131-3	Classical programming
%Q3.1	Q3.1
%MW8	M8
%IX5.7	I5.7

Access

In two examples in section 7.3.4, it was shown how physical addresses outside the PROGRAM POU are accessed:

- 1) Through the call interface of functions and function blocks
- 2) As global data (not possible in case of functions)

These options can be realized even in case of mixed programming, refer to section 11.4.2 and 11.4.3.

Since in the mixed programming, there is no PROGRAM POU, the symbol file is used as declaration source.

Data declaration

In the symbol file, no variable types such as "VAR", "VAR_GLOBAL" or "VAR_EXTERNAL" can be specified. Declarations in the symbol file is always global (VAR_GLOBAL).

In addition to the physical addresses, other data areas of the PLC can be used for mixed programs:

Operand	Explanation
I	Physical input
Q	Physical output
M	Marker
SM	Special marker
DB	Data buffer
DF	Data field

Format

Declaration format of commonly usable operands in the symbol file:

`Operand, data type Variable name`

 **The data type is specified directly after the (absolute) operand – separated by comma. No space or delimiter may be set in between.**

The specification of the data type is optional. In the IEC modules however a checking of the data type is carried out. It is recommended that always the suitable data be specified so that on project generation (linking), no warnings are outputted.

The variable name must comply with the rules for the identifiers, refer to section 7.1.2.

Examples

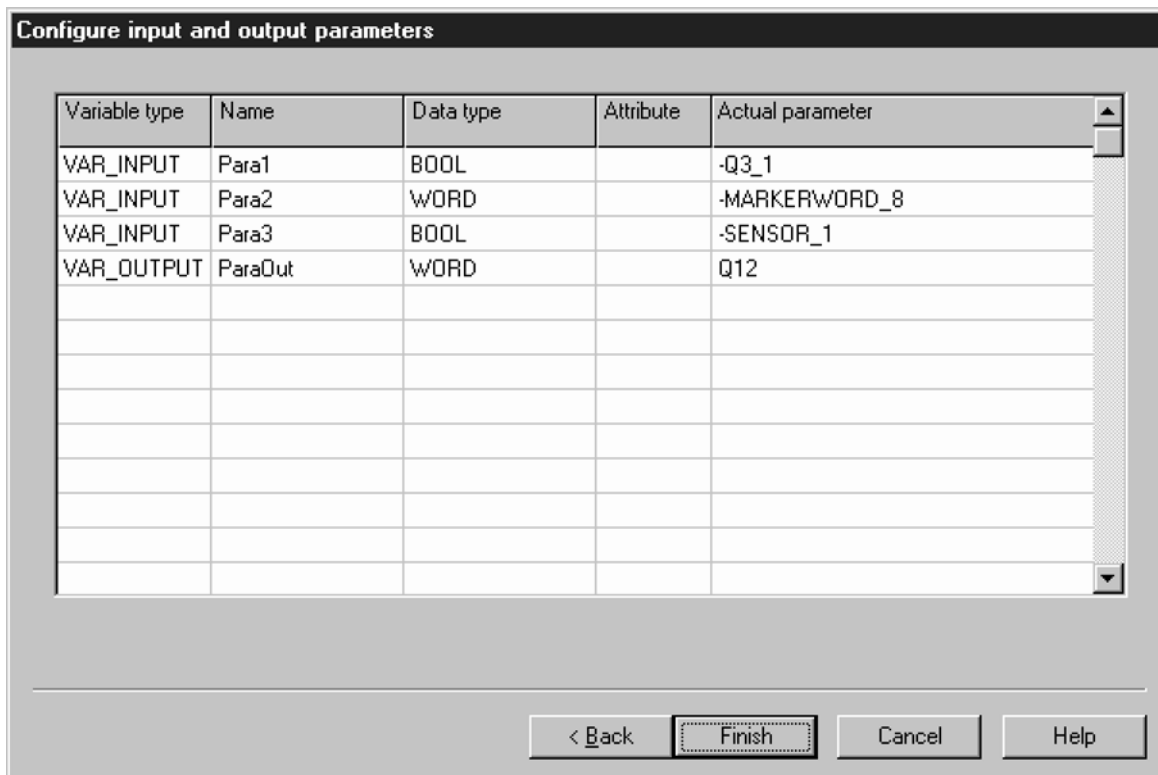
```
Q3.1, BOOL      Q3_1
MW8, WORD      MARKERWORD_8
I5.7, BOOL      SENSOR_1
```

11.4.2 Symbolic operands via the call interface

FUNCTION_BLOCKS can be called up with the help of the WinSPS wizard “Call up parameter list” from the Bosch-IL or from a SFC step, refer to section 11.3.

In the dialog window of the actual parameters, the symbolic operands of the input or output parameters are assigned. Here, every symbol must be prefixed with a hyphen (–), refer to the illustration below.

First, the operands (variables) must be entered in the symbol file. Absolute physical operands can be similarly specified. The uppercase /lowercase letters of the parameters must match with the operands in the symbol file.



Dialog window of the WinSPS function “Call up parameter list”

Entries in the symbol file related to the above-mentioned example:

```
Q3 . 1 , BOOL      Q3_1
MW8 , WORD        MARKERWORD_8
I5 . 7 , BOOL     SENSOR_1
```

The actual parameter for “ParaOut” must not be entered in the symbol file as in this case, directly the absolute operand “Q12” is transferred instead of a symbolic operand.

11.4.3 Symbolic operands as global variables

There is an option for importing the operands declared in the symbol file as global variables in FUNCTION_BLOCKS. In this regard, three things must be kept in mind:

- Absolute addresses cannot be accessed from IEC modules. All addressed must as a result be available in the symbol file as symbolic operands.
- The data type of the symbolic operands must be specified in the symbol file. If the data type is left out, a warning is given out in the called module at the time of linking (create new project).

Model entries in the symbol file:

```
Q3.1, BOOL      Q3_1
MW8, WORD       MRKERWORD_8
I5.7, BOOL      SENSOR_1
```

Example for access in an IEC module:

```
FUNCTION_BLOCK xy
VAR_EXTERNAL
  Q3_1 : BOOL;
  MARKERWORD_8 : WORD
  SENSOR_1 : BOOL;
END_VAR
...
```

In the FB, the global variables with the variable type VAR_EXTERNAL are imported. The uppercase /lowercase letters of the variable names must match with the operands in the symbol file.

11.4.4 Global type definitions


Type definitions, which are entered in the global type editor (refer to section 5.5), can also be used in the modules of the classical programming languages in the call interface to the IEC modules.

For this, the name of the derived data types is assigned to an operand and variable names in the symbol file.

Format

Declaration format for global type definitions in the symbol file:

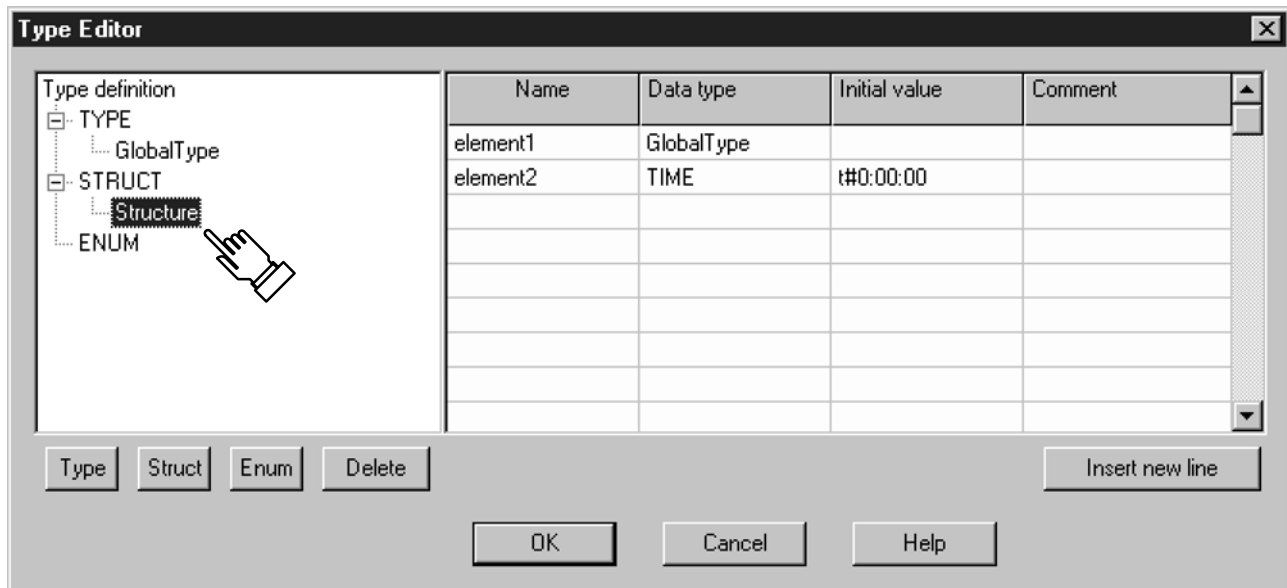
```
Operand, data type      Variable name
```

 **The data type is specified directly after the (absolute) operand – separated by comma. No space or delimiter may be set in between, refer to the example below.**

The variable name must comply with the rules for the identifiers, refer to section 7.1.2.

Example

The following illustration shows an extract from the type editor with various derived data types (the elements of the data structure “Structure” are visible).



Global derived data types in the type editor

These data types are declared in the symbol file for the mixed programming environment:

```
M20,GlobalType    INTEGER_125
DF0,Structure     STRUCTURENAME
DB4,Enumerate    COLORS
```

Working out the data size

To start with, with the declaration of a derived data type in the symbol file, an “initial address” is created. All subsequent elements of the data type occupy the memory addresses following this address. Thus for example, a data structure occupies a continuous memory area depending upon the number and data size of the structure elements.

BOOL types occupy a complete Byte in the memory.

Moreover, attention must be paid to the maintenance of “base byte addresses”. Information concerning this can be found in the software manual of the respective controller.



DANGER

With the declaration of a derived data type in the symbol file, only an “initial address” is created! The subsequent addresses must be worked out from the data size of the derived data types and the displacements due to the base byte addresses. Subsequent addresses may not be used for other access!

If subsequent addresses are used repeatedly, data area in the PLC could get overwritten by mistake!

Access to global type definitions

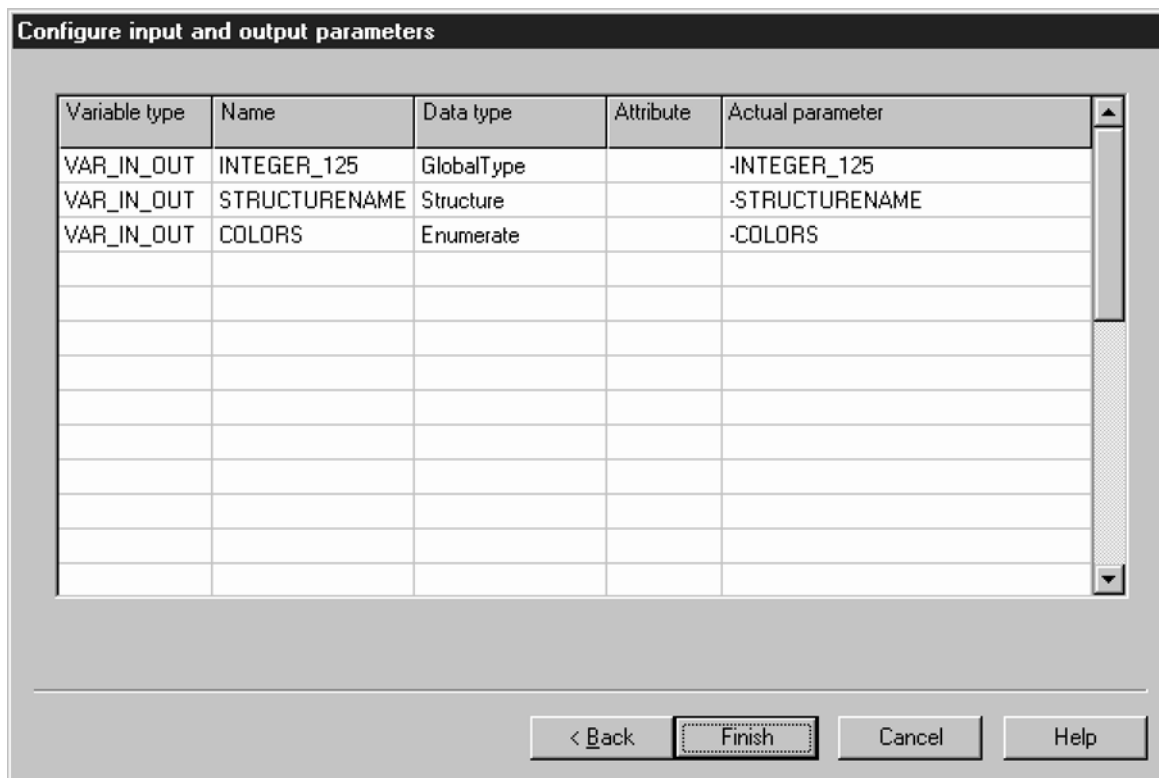
In the classical programming language (Bosch-IL), variable declared in the symbol file can be used to access the derived data type. However, neither elements of structures nor the names (text constants) of enumerations can be accessed.

The declaration in the symbol file should rather allow transfer of a pointer to a memory address in an IEC module. For the transfer of pointer variables, the variable type "VAR_IN_OUT" in the function blocks is particularly suitable.

Similarly, the access via the variable type "VAR_EXTERNAL" is possible within a function block.

Example for access via call interface

The following illustration shows a typical application example for the transfer of global data types via the call interface. Using the menu function "Call up parameter list", an FB call is generated inside the classical module, also refer to section 11.3. In the dialog window of the actual parameters, the symbolic operands of the input or output parameters are assigned. In this example, pointer variables (VAR_IN_OUT) are implemented as parameters:



Dialog window of the WinSPS function "Call up parameter list"

It is important during parameter setting that the derived data types inside the FB declaration match the definition in the type editor, refer to the illustration above: Column data type. The uppercase /lowercase letters of the variable names must match with the entries in the symbol file.

Example for access via VAR_EXTERNAL

The access as global variable is allowed in the function block using the variable type "VAR_EXTERNAL":

```
VAR_EXTERNAL
  INTEGER_125 : GlobalType;
  STRUCTURENAME : Structure;
  COLORS : Enumerate
END_VAR
```

Even here, make sure that the derived data types inside the FB declaration match the definition in the type editor. The uppercase/lowercase letters of the variable names must match with the entries in the symbol file.

11.5 Differences in case of mixed programming

In case of mixed programming, keep in mind that a few differences exist between the classical programming and the programming as per IEC 61131-3. The following table shows a comparison of important elements:

	IEC 61131-3	Classical programming
Physical addresses	Physical addresses are assigned a prefix and data format. Example: %IX3.7	Physical addresses have another format. Example: I3.7 In case of mixed programming, the classical format is used throughout.
Strings (strings)	The string terminator \0 is automatically appended.	\0 must be appended manually if the string is to be processed free of any error in the IEC environment.
Arrays and structures	The maximum size of arrays and structures is equal to the data module length minus header data. Example: Array1 : ARRAY[1.. 502] OF BYTE. 502 bytes are allowed, otherwise, an error message is given out at the time of compilation.	No restriction.
Identifiers	Uppercase and lowercase letters do not play any role.	In case of mixed programming, attention must be paid to the uppercase / lowercase letters. If for example the symbolic operand "MARKERWORD_8" is entered in the symbol file, you must use this style "MARKERWORD_8" also inside the IEC modules and the call interfaces.

Notes:

12 Standardized functionality


In addition to the program and data structure, the IEC 61131-3 also standardizes important PLC functionalities. These are predefined in the norm as standard functions or function blocks. During implementation, all manufacturers of programming system or module libraries must follow these directions.

12.1 Standard functions

The standard functions are divided in various groups:

- Functions for data conversion
- Numeric functions
- Arithmetic functions
(in case of Bosch only through symbolic style, refer to 12.1.5)
- Shift functions
- Boolean functions
- Functions for selection and comparison
(Option not currently supported by WinSPS)
- Functions for strings (string functions)
- Special functions – data type – time
- Special functions – data types – enumeration
(not currently supported by WinSPS)

The following sections include instructions for parameterization and detailed explanations concerning the standard functions.

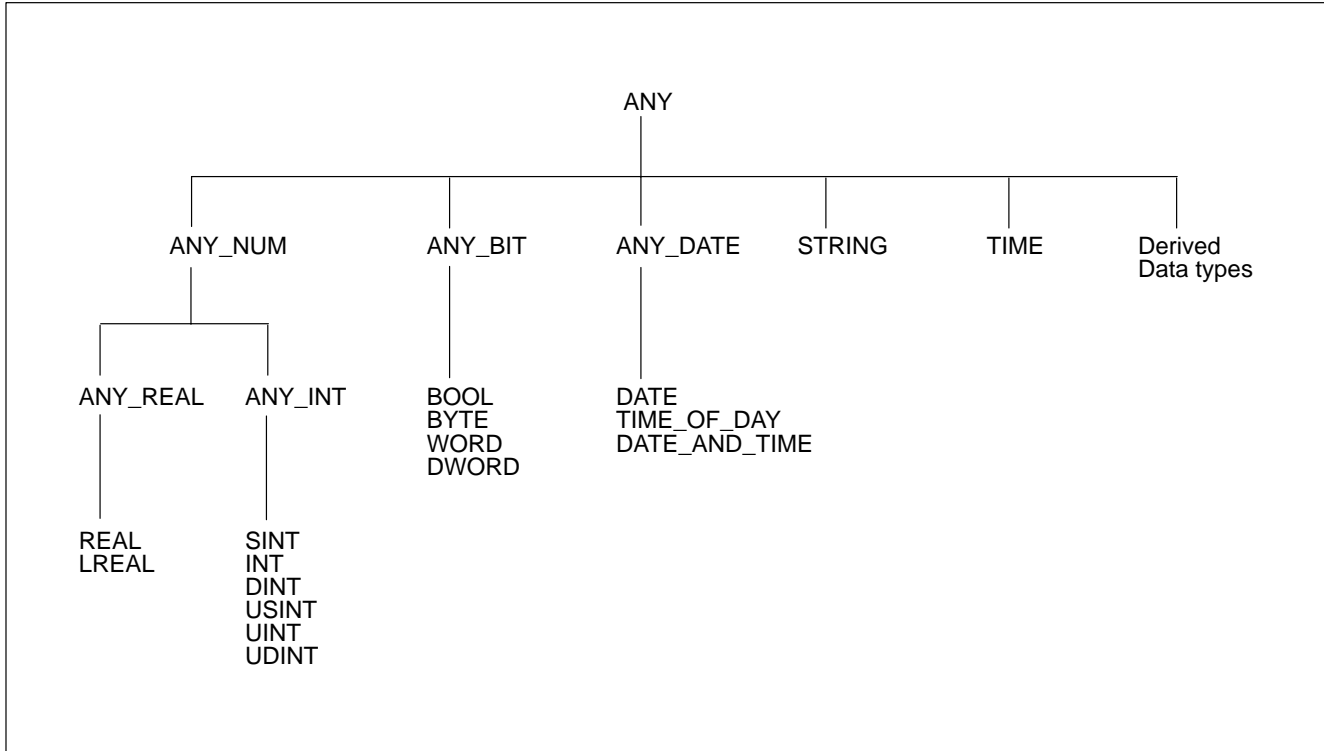
 **Pay attention to the instructions concerning the norm conformity in section 13.**
The supported standard functions are listed there.

12.1.1 Generic data types and “overloaded” functions

In many standard functions, the characteristics of **generic data types** are used, refer to 7.2.3 and the illustration below. As a result of this, input variables of a function can not only be applied for one but multiple data types. With this characteristic, they are termed as overload or overloadable functions.

In the following explanation of the standard functions, it is specified that which functions are overloadable and which are not overloadable.

- ☞ **Generic data types (ANY...) are used only for illustrative group formation of elementary data types. They can not be used for the declaration of variables or for program processing. The declaration takes place instead through the elementary data types associated with the group , z.B. "BYTE" for the group "ANY_BIT".**



Hierarchical order of the generic data types.

Example

The input parameter IN of the standard function SHL (shift bit by bit towards left) is of the common data type ANY_BIT. As a result, the data types BOOL, BYTE, WORD and DWORD are allowed.

While calling up, the entire overloaded inputted parameters and possibly, also the function value should have the same data type or the same data width. If for example the comparison function LT is called up, both the parameters to be compared must be of the same data type.

- ☞ **Generic data types can be used only for standard and manufacturer functions. The programming of overloaded functions (user functions) is not possible.**

12.1.2 Extensibility of functions

If a standard function allows a variable number of input parameters, it is termed as extensible. Fundamentally, these are a few arithmetic (ADD, MUL), bit string (boolean links), comparison and string functions.

Example

```
CONCAT ("To", "gether")      (* 2 input parameters *)
CONCAT ("To", "get", "her") (* 3 input parameters *)
```

12.1.3 Type conversion

The standard functions for the type conversion convert the input variable in the data type of the function value (return value).

Function	Data Type Input	Data Type Function value	Ü	E	Explanation
..._TO_...	ANY	ANY	Y	-	Convert data type
TRUNC	ANY_REAL	ANY_INT	Y	-	Make integer
BCD_TO_...	ANY_BIT	ANY	Y	-	from BCD
..._TO_BCD	ANY_BIT	ANY_BIT	Y	-	To BCD
TIME_AND_DATE_ TO_TIME_OF_DAY	DT	TOD	-	-	To time of day
DATE_AND_TIME_ TO_DATE	DT	DATE	-	-	To date


O: overloadable, E: extensible, Y: Yes, -: No

* The **BCD** coded conversion functions are not supported.

The general description of the function **..._TO_...** allows the conversion of all possible elementary data types, refer to the list below. The source data type is specified on the left side, the destination data type on the right side.

During the type conversion of **ANY_REAL** to **ANY_INT** data types after decimal positions are rounded or rounded off to the next integer.

The function **TRUNC** truncates all after decimal positions in order to get an integer value.

 **During the type conversion of numeric data types, take care that the allowed value of the destination data type is not exceeded. If the value of the input operand is too large, the result is reduced to the available bit length of the output operand and thus supplies an incorrect value.**

Examples (programming language ST):

```

Var := INT_TO_UDINT (-200);      (* Var := 200 *)
Var := INT_TO_SINT (-200);      (* does not create any
                                useful value *)

Var := REAL_TO_DINT (-825.66); (* VAR := -826 *)
Var := TRUNC (-825.66);        (* VAR := -825 *)
Var := BYTE_TO_UINT (16#96);   (* Var := 150 *)
Var := INT_TO_BOOL (100);      (* Var := TRUE *)
Var := INT_TO_BOOL (0);        (* Var := FALSE *)

```

**DANGER**

The functions for the type conversion **...TO_STRING** and **STRING_TO_...** are under preparation. However, no error message is generated during compilation.

A run-time error would occur in the controller and the controller would **STOP!**

The following list shows all the possible type conversion functions “...TO...”:

BOOL_TO_BYTE	BOOL_TO_DINT	BOOL_TO_DWORD	BOOL_TO_INT	BOOL_TO_SINT
BOOL_TO_TIME	BOOL_TO_UDINT	BOOL_TO_UINT	BOOL_TO_USINT	BOOL_TO_WORD
BYTE_TO_BOOL	BYTE_TO_DINT	BYTE_TO_DWORD	BYTE_TO_INT	BYTE_TO_SINT
BYTE_TO_TIME	BYTE_TO_UDINT	BYTE_TO_UINT	BYTE_TO_USINT	BYTE_TO_WORD
DINT_TO_BOOL	DINT_TO_BYTE	DINT_TO_DWORD	DINT_TO_INT	DINT_TO_SINT
DINT_TO_TIME	DINT_TO_UDINT	DINT_TO_UINT	DINT_TO_USINT	DINT_TO_WORD
DWORD_TO_BOOL	DWORD_TO_BYTE	DWORD_TO_DINT	DWORD_TO_INT	DWORD_TO_SINT
DWORD_TO_TIME	DWORD_TO_UDINT	DWORD_TO_UINT	DWORD_TO_USINT	DWORD_TO_WORD
INT_TO_BOOL	INT_TO_BYTE	INT_TO_DINT	INT_TO_DWORD	INT_TO_SINT
INT_TO_TIME	INT_TO_UDINT	INT_TO_UINT	INT_TO_USINT	INT_TO_WORD
SINT_TO_BOOL	SINT_TO_BYTE	SINT_TO_DINT	SINT_TO_DWORD	SINT_TO_INT
SINT_TO_TIME	SINT_TO_UDINT	SINT_TO_UINT	SINT_TO_USINT	SINT_TO_WORD
TIME_TO_BOOL	TIME_TO_BYTE	TIME_TO_DINT	TIME_TO_DWORD	TIME_TO_INT
TIME_TO_SINT	TIME_TO_UDINT	TIME_TO_UINT	TIME_TO_USINT	TIME_TO_WORD
UDINT_TO_BOOL	UDINT_TO_BYTE	UDINT_TO_DINT	UDINT_TO_DWORD	UDINT_TO_INT
UDINT_TO_SINT	UDINT_TO_TIME	UDINT_TO_UINT	UDINT_TO_USINT	UDINT_TO_WORD
UINT_TO_BOOL	UINT_TO_BYTE	UINT_TO_DINT	UINT_TO_DWORD	UINT_TO_INT
UINT_TO_SINT	UINT_TO_TIME	UINT_TO_UDINT	UINT_TO_USINT	UINT_TO_WORD
USINT_TO_BOOL	USINT_TO_BYTE	USINT_TO_DINT	USINT_TO_DWORD	USINT_TO_INT
USINT_TO_SINT	USINT_TO_TIME	USINT_TO_UDINT	USINT_TO_UINT	USINT_TO_WORD
WORD_TO_BOOL	WORD_TO_BYTE	WORD_TO_DINT	WORD_TO_DWORD	WORD_TO_INT
WORD_TO_SINT	WORD_TO_TIME	WORD_TO_UDINT	WORD_TO_UINT	WORD_TO_USINT

12.1.4 Numeric functions

The data type of the input parameter must match with the data type of the function value.

Function	Data Type Input	Data Type Function value	Ü	E	Explanation
ABS	ANY_NUM	ANY_NUM	Y	-	Absolute value
SQRT	ANY_REAL	ANY_REAL	Y	-	Square root
LN	ANY_REAL	ANY_REAL	Y	-	Natural algorithm
LOG	ANY_REAL	ANY_REAL	Y	-	Algorithm base 10
EXP	ANY_REAL	ANY_REAL	Y	-	Exponentiation
SIN	ANY_REAL	ANY_REAL	Y	-	Sine function
COS	ANY_REAL	ANY_REAL	Y	-	Cosine function
TAN	ANY_REAL	ANY_REAL	Y	-	Tangent function
ASIN	ANY_REAL	ANY_REAL	Y	-	Inverse sine function
ACOS	ANY_REAL	ANY_REAL	Y	-	Inverse cosine function
ATAN	ANY_REAL	ANY_REAL	Y	-	Inverse tangent function

O: overloadable, E: extensible, Y: Yes, -: No

Example (IL):

```
LD      Alpha
COS
ST      cosAlpha      (* Cosine *)
```

Example (ST):

```
cosAlpha := COS (Alpha);      (* Cosine *)
```

12.1.5 Arithmetic functions

 In the programming language ST, while processing the data types "ANY_NUM", only the symbolic style of the arithmetic standard functions can be entered.

Example:

```
Var1 := Var2 + Var3 + Var4;      (* error free *)
Var1 := ADD (Var2, Var3, Var4);  (* is reported as
error *)
```

The modifier for the symbolic style is specified in the table in column "M".

Function	M	Data Type Input	Data Type Function value	Ü	E	Explanation
ADD	+	ANY_NUM ANY_NUM	ANY_NUM	Y	Y	Addition
ADD	+	TIME TIME	TIME	Y	-	Time addition
ADD	+	TOD TIME	TOD	Y	-	Time addition
ADD	+	DT TIME	DT	Y	-	Time addition
MUL	*	ANY_NUM ANY_NUM	ANY_NUM	Y	Y	Multiplication
MUL	*	TIME ANY_NUM	TIME	Y	-	Time multiplication
SUB	-	ANY_NUM ANY_NUM	ANY_NUM	Y	-	Subtraction
SUB	-	TIME TIME	TIME	Y	-	Time subtraction
SUB	-	DATE DATE	TIME	Y	-	Time subtraction
SUB	-	TOD TIME	TOD	Y	-	Time subtraction
SUB	-	TOD TOD	TIME	Y	-	Time subtraction
SUB	-	DT TIME	DT	Y	-	Time subtraction
SUB	-	DT DT	TIME	Y	-	Time subtraction
DIV	/	ANY_NUM ANY_NUM	ANY_NUM	Y	-	Division
DIV	/	TIME ANY_NUM	TIME	Y	-	Time division
MOD		ANY_NUM ANY_NUM	ANY_NUM	Y	-	Modulo, residue bilding
EXPT		ANY_NUM ANY_NUM	ANY_NUM	Y	-	Exponent
MOVE		ANY_NUM ANY_NUM	ANY_NUM	Y	-	Assignment

S: Symbolic style, O: overloadable, E: extensible, Y: Yes, -: No

If the division of an integer does not result in an integer, the after decimal positions are truncated.

 **The division by zero is not allowed!**

Example (ST):

```
todMESZ:= ADD (todMEZ, T#1h); (* Addition with the
                                type time of day *)
```

12.1.6 Shift functions

The data type of the first input parameter must match with the data type of the function value.

The second input parameter "N" must contain an integer value. The parameter contains the number of the positions to be shifted or rotated.

Function	Data Type Input	Data Type Function value	O	E	Explanation
SHL	ANY_BIT N	ANY_BIT	Y	-	Shifting towards left (fill with zeros from left)
SHR	ANY_BIT N	ANY_BIT	Y	-	Shifting towards right (fill with zeros from left)
ROR	ANY_BIT N	ANY_BIT	Y	-	Rotate towards right (ring shape)
ROL	ANY_BIT N	ANY_BIT	Y	-	Rotate towards left (ring shape)

O: overloadable, E: extensible, Y: Yes, -: No

Example (ST):

```
%QB8 := SHR (2#01101100, 4); (* Shift 4 positions
                             towards right *)
                             (* %QB := 2#00000110 *)

Rot1 := ROL (2#11000011, 2); (* Rotate 2 positions
                             towards left *)
                             (* Rot1 := 2#00001111 *)
```


12.1.7 Boolean functions – logical links

The data type of the input parameter must match with the data type of the function value.

The symbol for the symbolic style is specified in the table in the second column.

Function	S	Data Type Input	Data Type Function value	O	E	Explanation
AND	&	ANY_BIT ANY_BIT	ANY_BIT	Y	Y	Logical AND
OR		ANY_BIT ANY_BIT	ANY_BIT	Y	Y	Logical OR
XOR		ANY_BIT ANY_BIT	ANY_BIT	Y	Y	Logical exclusive OR
NOT		ANY_BIT	ANY_BIT	Y	–	Logical negation

O: overloadable, E: extensible, Y: Yes, –: No

 **In the programming language ST, only the symbolic style of the arithmetic standard functions can be entered.**

Examples (IL):

```
LDN   B
AND   A
ST    C

LD    2#10100110  (* Bit pattern *)
&    2#11110000
ST    %QB0
```

Example (ST):

```
C := A AND NOT B;
```

12.1.8 Selection

Functions for selection are presently not supported.

12.1.9 Comparison

The data type **ENUM** stands for the “derived data type” enumeration, refer to 7.2.2.

Function	Data Type Input	Data Type Function value	O	E	Explanation
GT	ANY ANY	BOOL	Y	Y	Greater than
GE	ANY ANY	BOOL	Y	Y	Greater than or equal
EQ	ANY ANY	BOOL	Y	Y	Equal
EQ	ENUM ENUM	BOOL	-	-	Equal
LT	ANY ANY	BOOL	Y	Y	Less than
LE	ANY ANY	BOOL	Y	Y	Less than or equal
NE	ANY ANY	BOOL	Y	-	Unequal
NE	ENUM ENUM	BOOL	-	-	Unequal

O: overloadable, E: extensible, Y: Yes, -: No

Example (IL):

```

M1: LD    iOperand1
    ADD   iOperand2
    ST    iResult      (* calculation result *)
    EQ    100          (* comparison CR = 100 ?... *)
    JMPC  M2          (* ...then jump to M2 *)

```

Example (ST):

```

bLT := LT (Var_1, Var_2);
(* Comparison, whether Var_1 is less than Var_2 *)


```

12.1.10 Functions for strings

Function	Data Type Input	Data Type Function value	O	E	Explanation
LEN	STRING	INT	-	-	Determining the length of a string
LEFT	STRING L	STRING	Y	-	Cutting off the beginning of a string, L characters from left
RIGHT	STRING R	STRING	Y	-	Cutting off the end of a string, R characters from right
MID	STRING L R	STRING	Y	-	Cutting of the beginning and end of a string, L characters from left and R characters from right
CONCAT	STRING STRING	STRING	-	Y	Concatenating multiple strings
CONCAT	DATE TOD	DT	-	-	Date and time of day to put together date + time of day
INSERT	STRING STRING P	STRING	Y	Y	Inserting a string (2nd STRING) in another (1st STRING), from position P onwards
DELETE	STRING L P	STRING	Y	Y	Deleting a part of a string, L characters from position P onwards
REPLACE	STRING STRING L P	STRING	Y	Y	Replacing a part (1st STRING) in a string by another part (2nd STRING), L characters from position P onwards
FIND	STRING STRING	INT	Y	Y	Searching a string (2nd STRING) within another string (1st STRING). Function value = Found position, otherwise 0

O: overloadable, E: extensible, Y: Yes, -: No

* The function **CONCAT** with date and time is not supported.

 **Positions within strings are counted starting from "1".**

Example (ST):

```
iLength := LEN ('String'); (* Determining the
                           string length:
                           iLength := 6 *)

szMid := MID ('ABCDEF', 2, 1 );
          (* Cutting off: szMid := 'BCDE' *)

szCon := CONCAT ('To', 'get', 'her');
          (* szCon := 'Together' *)
```

```
szNew := INSERT ('ABCDEF', '12345', 3, 2 );  
          (* Inserting: szNew := 'A123BCDEF' *)  
  
iPos := FIND ('ABCDEF', 'DE' );(* Search: iPos := 4 *)
```

**DANGER**

If the size of the string is increased by a function, the related string variable must be declared large enough, for example, see below. It can otherwise have the effect that the address range is crossed, which can lead to uncontrolled behavior!

Erroneous example:

```
VAR  
  szVar : STRING (13);  
END_VAR  
  
szVar := CONCAT ('more ', 'characters');
```

The string variable “szVar” can take 13 characters. In the function call “CONCAT” however it must be able to accept at least 14 characters. It has the effect that the address range is exceeded.

12.2 Standard function block

The IEC 61131-3 defines the following standard function blocks which can be divided in groups:

Name	Inputs	Outputs	Explanation
Bistable elements			
SR	Set1*,ReSet*	Q1	Setting on priority
RS	Set*,ReSet1*	Q1	Resetting on priority
Edges			
R_TRIG	CLK	Q	Detection of rising edge
F_TRIG	CLK	Q	Detection of falling edge
Counter			
CTU	CU,ReSet*,PV	Q,CV	Forward counter
CTD	CD,Load*,PV	Q,CV	Backward counter
CTUD	CU,CD,R*,Load*,PV	QU,QD,CV	forward/backward counter
Time			
TP	IN,PT	Q,ET	Pulse generator
TON	IN,PT	Q,ET	Switch on delay
TOF	IN,PT	Q,ET	Switch off delay
RTC	EN,PDT	Q,CDT	Real-time clock

* Refer to instructions

☞ The key words “R”, “S” und “LD” of the input parameters are used in the programming language instructions list (IL) with another meaning. Due to this conflict, there are difficulties during translation by a compiler. This problem was taken up in the working group for further development of the IEC 61131-3. In a revised version of the standard, the parameter names should be changed to “Set”, “ReSet” und “Load”. Bosch already uses this modified form.

☞ All variables are basically handled as with RETAIN attribute. Exceptions are the standard FBs. These are basically non-remanent i.e. they are re-initialized after every STOP/RUN switchover.

Description of all input and output parameters:

Parameter	Explanation	Data Type
CD	Input count downwards (Count Down)	R_EDGE
CDT	Current date and time (Current Date and Time)	DT
CLK	Cycle input (CLock)	BOOL
CU	Input count upwards (Count UP)	R_EDGE
CV	Current Value	INT
ET	Current time (End Time)	TIME
IN	Time input (INput)	BOOL
LOAD (LD)	Load counter value (LoaD)	INT
PDT	Date/time (Preset Date and Time)	DT
PT	Time (Preset Time)	TIME
PV	Counter value	INT
Q	Output	BOOL
QD	Output downwards (Down)	BOOL
QU	Output upwards (Up)	BOOL
RESET (R)	Reset input (Reset)	BOOL
R1	R on priority	BOOL
SET (S)	Set input (Set)	BOOL
S1	S on priority	BOOL

12.2.1 Bistable elements – Flipflops

The norm recognizes two types of Flip Flops :

- **SR** Flipflop Set dominant
- **RS** Flipflop Reset dominant

SR Flipflop



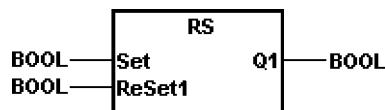
SR Flipflop: Set dominant

Set1 set condition
ReSet reset condition
Q1 Output condition of the bistable element

The function block "SR" has the characteristic of switching a data element – the output "Q1" – statically to the binary state "1" or "0".

The switchover between both the states takes place depending upon the value of the binary operands "Set1" and "ReSet". At the beginning, the output "Q1" is initialized with the value "0". The first processing of the function block with the value "1" of the operand "Set1" has the effect that the output "Q1" contains the value "1" – it is set. After this, the change in the value of "Set1" has no influence on the output "Q1". The value "1" of the input operand "ReSet" switches the output "Q" to the state "0" – the output is reset. If both the input operands take the value "1", the fulfilled set condition dominates i.e. "Q1" is set on priority.

RS Flipflop



RS Flipflop: Reset dominant

Set set condition
ReSet1 reset condition
Q1 Output condition of the bistable element

The function block "RS" has the characteristic of switching a data element – the output "Q1" – statically to the binary state "1" or "0".

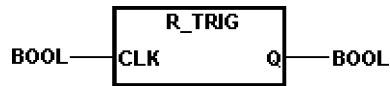
The switchover between both the states takes place depending upon the value of the binary operands "Set" and "ReSet1". At the beginning, the output "Q1" is initialized with the value "0". The first processing of the function block with the value "1" of the operand "Set" has the effect that the output "Q1" contains the value "1" – it is set. After this, the change in the value of "Set" has no influence on the output "Q1". The value "1" of the input operand "ReSet1" switches the output "Q1" to the state "0" – the output is reset. If both the input operands take the value "1", the fulfilled reset condition dominates i.e. "Q1" is reset on priority.

12.2.2 Edge detection

The norm recognizes two different modules for edge detection:

- **R_TRIG** rising edge
- **F_TRIG** falling edge

Rising edge R_TRIG

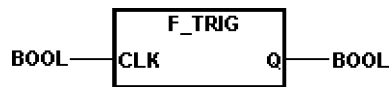


R_TRIG: Rising edge

- CLK** Input operand, whose rising edge is detected
Q Output operand reports the rising edge of "CLK"

The function block "R_TRIG" evaluates the condition of the input operand "CLK". The condition change from "0" in a processing cycle to "1" in the next processing cycle is detected and is indicated through the output "Q" with the binary value "1". The state "1" exists at the output only in the processing cycle, in which the condition change of "CLK" is detected and a rising edge is reported.

Falling edge F_TRIG



F_TRIG: Falling edge

- CLK** Input operand, whose falling edge is detected
Q Output operand reports the falling edge of "CLK"

The function block "F_TRIG" evaluates the condition of the input operand "CLK". The condition change from "1" in a processing cycle to "0" in the next processing cycle is detected and is indicated through the output "Q" with the binary value "1". The state "1" exists at the output only in the processing cycle, in which the condition change of "CLK" is detected and a falling edge is reported.

12.2.3 Counter

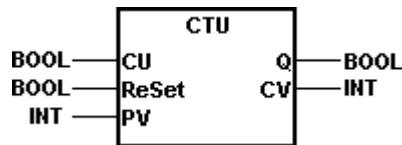
Counters can follow in different directions. The norm makes three counters available:

- **CTU** Upward counter
- **CTD** Downward counter
- **CTUD** Upward / downward counter

☞ **The inputs of the counter modules are edge triggered (R_EDGE) i.e. a rising edge must exist so that the counter value changes.**

☞ **Counters must be initialized with count limit or an initial value (PV), otherwise, they do not run!**

Upward counter CTU

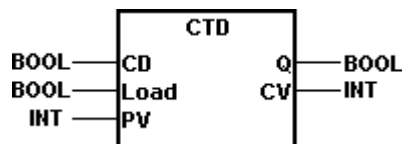


CTU: Upward counter

CU	Count pulse, rising edge
ReSet	reset condition
PV	count limit
Q	Message: Counter value greater than equal to PV
CV	Counter value

The function block "CTU" is used for upward counting of pulses which are supplied by the input operand "CU". During the initialization, the counter gets the value "0". Every rising edge at the input "CU" increments the counter, i.e. increases its value by one. The counter value can be erased with the value "1" of the operand "ReSet". The output operand "CV" supplies the current counter value. If the counter value lies below the limit "PV", the output operand "Q" takes the binary value "0". The reaching or exceeding the limit sets the output "Q" to "1".

Downward counter CTD

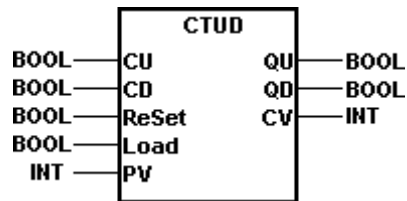


CTD Downward counter

CD	Count pulse, rising edge
Load	set condition
PV	initial value
Q	Message: Counter value smaller than equal to zero
CV	Counter value

The function block "CTD" is used for downward counting of pulses which are supplied by the input operand "CD". During the initialization, the counter gets the value "0". With the value "1" of the operand "Load", the value specified by the operand "PV" is accepted as the initial value in the counter. Every rising edge at the input "CD" decrements the counter, i.e. decreases its value by one. The output operand "CV" supplies the current counter value. If the counter value lies above the value "0", the output operand "Q" takes the binary value "0". When the counter value becomes equal to "0" or less than "0", the output "Q" is set to "1".

Upward and downward counter CTUD



CTUD: Upward / downward counter

CU	Count pulse, rising edge
CD	Count pulse, rising edge
ReSet	reset condition
Load	load condition
PV	load value
QU	Message: Counter value greater than equal to PV
QD	Message: Counter value smaller than equal to zero
CV	Counter value

The function block "CTUD" is used for upward and downward counting of pulses. During the initialization, the counter gets the value "0". Every rising edge at the input "CU" increments the counter, i.e. increases its value by one, a rising edge at the input "CD" decrements the counter, e.e. decreases its value by one. With the value "1" of the operand "Load", the value specified by the operand "PV" is accepted in the counter.

The counter value can be erased with the value "1" of the operand "ReSet". While the static state "1", the operand "ReSet" continues, the fulfilled count conditions or the fulfilled load conditions have no influence on the counter value.

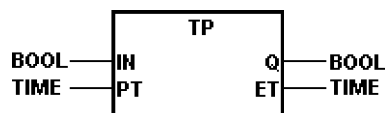
The output operand "CV" supplies the current counter value. If the counter value lies below the load value "PV", the output operand "QU" takes the boolean value "0". On reaching or exceeding the load value, the output "QU" is set to "1". While the counter value lies above the value "0", the output operand "QD" takes the binary value "0". When the counter value becomes equal to "0" or less than "0", the output "QD" is set to "1".

12.2.4 Timer

The norm recognizes three time elements and the processing of the PLC internal real-time clock:

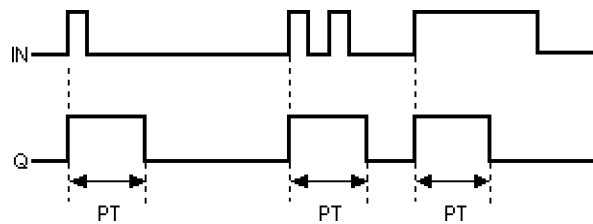
- **TP** Pulse
- **TON** On-delay
- **TOF** Off-delay
- **RTC** Real-time clock

Pulse TP



PT Time as pulse

IN	Start condition
PT	time specification
Q	condition of the timer
ET	current time



Time diagram

The rising edge of the input operand "IN" starts the time function of the timer "TP" for the time duration specified through the operand "PT". During this period, the output operand "Q" is in the state "1". The condition change at the input "IN" then does not have any influence on the flow.

If the "PT" value changes after the start, it becomes effective only with the next rising edge of the operand "IN".

The output operand "ET" shows the current time value. If the operand "IN" is in the state "1" on the expiry of the starting period, the operand "ET" retains the value. When the condition of the operand "IN" switches to "0", the value of "ET" changes to "0".

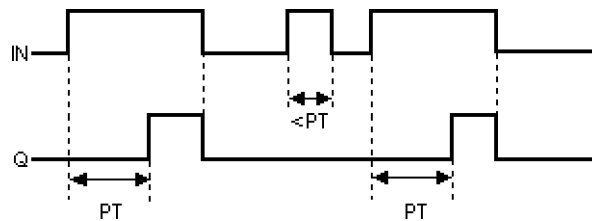
During the non-running period, every edge triggers a pulse of specific duration.

On-delay TON



TON: On-delay

- IN** Start condition
- PT** time specification
- Q** condition of the timer
- ET** current time



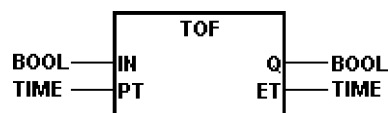
Time diagram

The rising edge of the input operand "IN" starts the time function of the timer "TON" for the time duration specified through the operand "PT". During the running period, the output operand "Q" is in the state "0". After the expiry of the starting period, the condition changes to "1" and the same is maintained until the operand "IN" changes to "0".

If the "PT" value changes after the start, it becomes effective only with the next rising edge of the operand "IN".

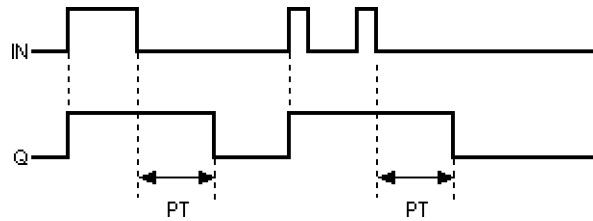
The output operand "ET" shows the current time value. If the started time is expired, the operand "ET" retains the value so long as the operand "IN" is in the state "1". When the condition of the operand "IN" switches to "0", the value of "ET" changes to "0". If the condition of the operand "IN" changes while the time runs to "0", the process is interrupted and the operand "ET" again assumes the value "0". The switching on at the input "IN" switches on the output "Q" delayed by the specified time duration.

Off-delay TOF



TOF: Off-delay

- IN** Start condition
- PT** time specification
- Q** condition of the timer
- ET** Expiring time



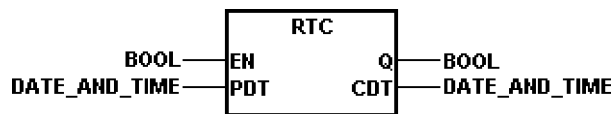
Time diagram

The state "1" of the input operand "IN" is transferred without delay to the output operand "Q". The falling edge of the "IN" starts the time function for the time duration specified through the operand "PT". The condition change at the input "IN" to "0" then does not have any influence on the flow. After the time expires, the operand "Q" changes to the state "0".

If the "PT" value changes after the start, it becomes effective only with the next rising edge of the operand "IN".

The output operand "ET" shows the current time value. If the started time is expired, the operand "ET" retains the value so long as the operand "IN" is in the state "1". When the condition of the operand "IN" switches to "0", the value of "ET" changes to "0". The switching off at the input "IN" switches off the output "Q" delayed by the specified time duration.


Real-time clock RTC



RTC: Real-time clock

EN	Set condition (not in function)
PDT	Set value for the real-time clock (not in function)
Q	Display of the current value of EN
CDT	Current date and time

The output operand "Q" indicates the condition of "EN". The current time status is outputted through the operand "CDT".

 **The IEC allows the setting of the PLC internal real-time clock with the function block RTC. In case of Bosch, RTC can be used only for reading the real-time clock. The setting must take place using the corresponding system command in the WinSPS editor or monitor. RTC does not work with milliseconds.**

13 Standard fulfilment

The following conformity tables as per IEC 61131-3 are used as checklists for evaluating the standard fulfillment. They demonstrate the characteristics of the WinSPS programming system. For the classical programming languages Bosch-IL, LD, FBD and SFC, the statements do not apply or apply only to a limited extent. The numbering in the tables are identical to the numbering in the IEC 61131-3.

The Bosch programming system WinSPS has the "Base Level Certificate" for the programming language ST.

13.1 Common elements

Character set characteristics (table 1)

No.	Explanation	Yes	No
1	Required character set	x	
2	Lowercase letters	x	
3a	Hash sign (#)	x	
	or		
3b	Pound sign		x
4a	Dollar sign (\$)	x	
	or		
4b	Currency symbol		x
5a	Vertical bar ()		x
	or		
5b	Exclamation sign (!)	x	
6a	Indexing delimiter: Square brackets []	x	
	or		
6b	Parenthesis ()		x

Identifier characteristics (table 2)

No.	Explanation	Yes	No
1	Uppercase letters and figures	x	
2	Uppercase and lowercase letters, figures, embedded underlines	x	
3	Uppercase and lowercase letters, figures, leading and embedded underlines	x	

Comment characteristics (table 3)

No.	Explanation	Yes	No
1	Comments	x	

Numeric literals (table 4)

No.	Explanation	Yes	No
1	Integer literals	x	
2	Real literals	x	
3	Real literals with exponents	x	
4	Binary literals	x	
5	Octal literals	x	
6	Hexadecimal literals	x	
7	Boolean zero and one	x	
8	Boolean FALSE and TRUE	x	

Characteristics of string literals (Table 5)

No.	Explanation	Yes	No
1	Empty string (zero length)	x	
	String of length 1 with character A	x	
	String of length 1 with space	x	
	String of length one with a quotation mark	x	
	String of length 2 with CR and LF character	x	
	String of length 5, printed as "\$1.00"	x	

2 Character combinations in strings (table 6)

No.	Explanation	Yes	No
2	Dollar sign (\$\$)	x	
3	Single quotation mark (\$')	x	
4	Linefeed (\$L or \$l)	x	
5	New line (\$N or \$n)	x	
6	New page (\$P or \$p)	x	
7	Carriage return (\$R or \$r)	x	
8	Tab (\$T or \$t)	x	

Time duration literal characteristics (table 7)

No.	Explanation	Yes	No
	Time duration literals without understroke:		
1a	Short prefix	x	
1b	Long prefix	x	
	Time duration literals with understroke:		
2a	Short prefix	x	
2b	Long prefix	x	

Literals for date and time (table 8)

No.	Explanation	Yes	No
1	Literals for date (long prefix: DATE#)	x	
2	Literals for date (short prefix: D#)	x	
3	Literals for time of day (long prefix: TIME_OF_DAY#)	x	
4	Literals for time of day (short prefix: TOD#)	x	
5	Literals for date and time (Long prefix: DATE_AND_TIME#)	x	
6	Literals for date and time (short prefix: DT#)	x	

Elementary data types (table 10)

No.	Keyword	Data Type	Yes	No
1	BOOL	Boolean	x	
2	SINT	Short integer	x	
3	INT	Integer	x	
4	DINT	Double integer	x	
5	LINT	Long integer		x
6	USINT	Unsigned short integer	x	
7	UINT	Unsigned integer	x	
8	UDINT	Unsigned double integer	x	
9	ULINT	Unsigned long integer		x
10	REAL	Real number	x	
11	LREAL	Long real number	x	
12	TIME	Time duration	x	
13	DATE	(only) date	x	
14	TIME_OF_DAY or TOD	(only) time	x	
15	DATE_AND_ TIME or TD	Date and time	x	
16	STRING	Variable length string	x	
17	BYTE	Bit string of length 8	x	
18	WORD	Bit string of length 16	x	
19	DWORD	Bit string of length 32	x	
20	LWORD	Bit string of length 64		x

Characteristics of data type declaration (table 12)

No.	Characteristics of data type declarations	Yes	No
1	Direct derivation of elementary data types	x	
2	Data types for enumeration types	x	
3	Data type for range		x
4	Data type for array	x	
5	Data type for structures	x	

Preset initial value (table 13)

Explanation	Initialization value	Yes	No
BOOL, SINT, INT DINT, LINT,	0	x	
USINT, UINT, UDINT, ULINT	0	x	
BYTE, WORD, DWORD, LWORD	0	x	
REAL, LREAL	0.0	x	
TIME	T#0s	x	
DATE	D#0001-01-01	x	
TIME_OF_DAY	TOD#00:00:00	x	
DATE_AND_TIME	DT#0001-01-01-00:00:00	x	
STRING	”(the empty string)	x	

**Characteristics of the initial values of data types
(Table 14)**

No.	Explanation	Yes	No
1	Initialization of directly derived types	x	
2	Initialization of data types for enumeration	x	
3	Initialization of data types for sub-range		x
4	Initialization of data types for array	x	
5	Initialization of data types for structure	x	
6	Initialization of data types for derived structure		x

**Characteristics of prefix for memory area and size for directly displayed variables
(Table 15)**

No.	Explanation	Yes	No
1	I: Memory area input	x	
2	Q: Memory area output	x	
3	M Memory area label	x	
4	X: Single bit size	x	
5	None: Single bit size	x	
6	B: Byte size	x	
7	W Word size	x	
8	D Double word size	x	
9	L: Long word size		x

Keywords for variable declaration (table 16)

Key word	Yes	No
VAR	x	
VAR_INPUT	x	
VAR_OUTPUT	x	
VAR_IN_OUT	x	
VAR_EXTERNAL	x	
VAR_GLOBAL	x	
VAR_ACCESS		x
RETAIN	x	
CONSTANT	x	
AT	x	

Characteristics of type assignment for variable (table 17)

No.	Explanation	Yes	No
1	Declaration of directly displayed non-buffered variables	x	
2	Declaration of directly displayed buffered variables		x
3	Declaration of memory area in case of symbolic variables	x	
4	Assignment of memory area in case of array	x	
5	Automatic memory allocation for symbolic variable	x	
6	Declaration for array	x	
7	Declaration for buffered array		x
8	Declaration for structured variables	x	

Characteristics of assignment of initial values for variable (Table 18)

No.	Explanation	Yes	No
1	Initialization of directly displayed non-buffered variables		x
2	Initialization of directly displayed buffered variables		x
3	Allocation of memory area and initial value for symbolic variables		x
4	Allocation of memory area and initialization for array		x
5	Initialization of symbolic variables	x	
6	Initialization for array	x	
7	Declaration and initialization for buffered array		x
8	Initialization of structured variables	x	
9	Initialization of constants	x	

Type data and overloaded functions (table 21)

No.	Explanation	Yes	No
1	Overload functions (type independent)	x	
2	Functions with type data		x

Characteristics of functions for type conversion (table 22)

No.	Explanation	Yes	No
1	*_TO_**	x	
2	TRUNC	x	
3	BCD_TO_**		x
4	*_TO_BCD		x

Standard functions with numeric variables (table 23)

No.	Explanation	Yes	No
1	ABS	x	
2	SQRT	x	
3	LN	x	
4	LOG	x	
5	EXP	x	
6	SIN	x	
7	COS	x	
8	TAN	x	
9	ASIN	x	
10	ACOS	x	
11	ATAN	x	

Arithmetic standard functions (table 24)

No.	Name	Symbol	Yes	No
12	ADD	+	x	
13	MUL	*	x	
14	SUB	-	x	
15	DIV	/	x	
16	MOD		x	
17	EXPT	**	x	
18	MOVE	:=	x	

Standard bit shift functions (table 25)

No.	Name	Yes	No
1	SHL	x	
2	SHR	x	
3	ROR	x	
4	ROL	x	

Bit boolean standard functions (table 26)

No.	Name	Yes	No
5	AND	x	
6	OR	x	
7	XOR	x	
8	NOT	x	

Standard functions for selection (table 27)

No.	Name	Yes	No
1	SEL		x
2a	MAX		x
2b	MIN		x
3	LIMIT		x
4	MUX		x

Standard functions for comparison (table 28)

No.	Name	Yes	No
5	GT	x	
6	GE	x	
7	EQ	x	
8	LE	x	
9	LT	x	
10	NE	x	

Standard functions for strings (table 29)

No.	Name	Yes	No
1	LEN	x	
2	LEFT	x	
3	RIGHT	x	
4	MID	x	
5	CONCAT	x	
6	INSERT	x	
7	DELETE	x	
8	REPLACE	x	
9	FIND	x	

Functions for data types of time (table 30)

No.	Name	Operation	Yes	No
1	ADD	TIME + TIME = TIME	x	
2		TOD + TIME = TOD	x	
3		DAT + TIME = DAT	x	
4	SUB	TIME - TIME = TIME	x	
5		DATE - DATE = TIME	x	
6		TOD - TIME = TOD	x	
7		TOD - TOD = TIME	x	
8		DAT - TIME = DAT	x	
9		DAT - DAT = TIME	x	
10	MUL	TIME * ANY_NUM = TIME	x	
11	DIV	TIME / ANY_NUM = TIME	x	
12	CONCAT	DATE TOD = DAT	x	
		Functions for type conversion		
13		DATE_AND_TIME_TO_TIME_OF_DAY	x	
14		DATE_AND_TIME_TO_DATE	x	

Functions for data types of enumeration (table 31)

No.	Name	Yes	No
1	SEL		x
2	MUX		x
3	EQ	x	
4	NE	x	

Characteristics of function block declaration (table 33)

No.	Explanation	Yes	No
1	RETAIN for internal variables	x	
2	RETAIN for output variables	x	
3	RETAIN for internal function blocks		x
4a	Input/ output declaration (text form)	x	
4b	Input/ output declaration (graphical)		x
5a	Function block instance name as input (text form)		x
5b	Function block instance name as input (graphical)		x
6a	Function block instance name as input/ output (text form)		x
6b	Function block instance name as input/ output (graphical)		x
7a	Function block instance name as external variable (text form)	x	
7b	Function block instance name as external variable (graphical)		x
	Declaration in text form of inputs		
8a	With rising edge	x	
8b	With falling edge	x	
	Graphical declaration of inputs		
9a	With rising edge		x
9b	With falling edge		x

Bistable standard functions (table 34)

No.	Name	Yes	No
1	SR	x	
2	RS	x	
3	SEMA		x

Standard function blocks edge detection (table 35)

No.	Name	Yes	No
1	R_TRIG	x	
2	F_TRIG	x	

Standard function blocks counters (table 36)

No.	Name	Yes	No
1	CTU	x	
2	CTD	x	
3	CTUD	x	

Standard function blocks timer (table 37)

No.	Name	Yes	No
1	TP (Puls)	x	
2a	TON (on delay)	x	
2b	T---0 (on delay)		x
3a	TOF (off delay)	x	
3b	0---T (off-delay)		x
4	RTC (real time clock)	x	

Characteristics of program declaration (table 39)

No.	Explanation	Yes	No
1	RETAIN for internal variables	x	
2	RETAIN for output variables		x
3	RETAIN for internal function blocks		x
4a	Input/ output declaration (Textform)		x
4b	Input/ output declaration (graphical)		x
5a	Function block instance name as input (text form)		x
5b	Function block instance name as input (graphical)		x
6a	Function block instance name as input/ output (text form)		x
6b	Function block instance name as input/ output (graphical)		x
7a	Function block instance name as external variable (text form)		x
7b	Function block instance name as external variable (graphical)		x
	Declaration in text form of inputs		
8a	With rising edge		x
8b	With falling edge		x
	Graphical declaration of inputs		
9a	With rising edge		x
9b	With falling edge		x
10	Formal input and output parameters		x
11	Declaration of directly displayed non-buffered variables	x	
12	Declaration of directly displayed buffered variables	x	
13	Declaration of memory area in case of symbolic variables	x	
14	Assignment of memory area in case of field		x
15	Initialization of directly displayed non-buffered variables		x
16	Initialization of directly displayed buffered variables		x
17	Allocation of memory area and initial value for symbolic variables		x
18	Allocation of memory area and initialization for field		x
19	Use of directly displayed variables	x	
20	VAR_GLOBAL .. END_VAR Declaration with a PROGRAM	x	
21	VAR_ACCESS .. END_VAR Declaration with a PROGRAM		x

13.2 Language elements

Operators of instructions list, IL (table 52)

No.	Operator	Modifier	Yes	No
1	LD	N	x	
2	.ST	N	x	
3	S		x	
	R		x	
4	AND	N,(x	
5	&	N,(x	
6	OR	N,(x	
7	XOR	N,(x	
8	ADD	(x	
9	SUB	(x	
10	MUL	(x	
11	DIV	(x	
12	GT	(x	
13	GE	(x	
14	EQ	(x	
15	NE	(x	
16	LE	(x	
17	LT	(x	
18	JMP	C, N	x	
19	CAL	C, N	x	
20	RET	C, N	x	
21)		x	

Characteristics of function block call in the language IL
(Table 53)

No.	Explanation	Yes	No
1	CAL with list of input parameters	x	
2	CAL with loading/ saving of input parameters	x	
3	Use of input operators		x

Operators of language ST (table 55)

No.	Explanation	Yes	No
1	Bracketing	x	
2	Function evaluation	x	
3	Exponentiation	x	
4	Negation	x	
5	Complement	x	
6	Multiplication	x	
7	Division	x	
8	Modulo	x	
9	Addition	x	
10	Subtraction	x	
11	Comparison	x	
12	Equality	x	
13	Inequality	x	
14	Boolean AND	x	
15	Boolean AND	x	
16	Boolean exclusive OR	x	
17	Boolean OR	x	

Statements of language ST (table 56)

No.	Explanation	Yes	No
2	Function block call and use of FB output	x	
3	RETURN	x	
4	IF	x	
5	CASE	x	
6	FOR	x	
7	WHILE	x	
8	REPEAT	x	
9	EXIT	x	
10	Empty statement	x	
1	RETAIN for internal variables	x	

13.3 Causes of errors

Causes of errors (table E.1)

Causes of errors	System reaction
Value of a variable differentiates the identified range	–
Length of the initialization list does not match the number of array entries	Syntax error
Type conversion error	Syntax error
Numeric result exceeds the range for data types	–
Division by zero	–
Mixed input data types in case of a selection function	Syntax error
Selector outside the range	–
Invalid character position	–
Result exceeds range for data type	–
Zero or more as a starting step in SFC network	–
Simultaneously fulfilled non-prioritized transitions	–
Side effects in the evaluation of a transition condition	–
Action control error	–
Unsafe or unachievable sequence	–
Data type conflict in VAR_ACCESS	–
Task requires too many processor resources; execution end not reached; Other task time plan conflicts	–
Numeric result exceeds the range for data type	–
Division by zero; Invalid data type for operand	–
Reverse jump from the function without assigned value	–
Infinite loop	Cycle time error
Same identifier as connector and element name	–
Non-initialized reverse link variable	–
Numeric result exceeds the range for data types	–
Division by zero	–

A Annex

A.1 Abbreviations

Abbreviation	Description		
DM	Data Module	LD	Ladder Diagram
ESD	Electro Static Discharge Abbreviation for all terms relating to electro-static discharge, e.g. ESD protection, ESD hazards, etc.	OM	Organisation Module
FB	FUNCTION_BLOCK acc. to IEC 61131-3	PCL	Bosch Software PLC. The name is composed from PC (Personal Computer) and CL (Control Logic).
FBD	Function Block Diagram	PLC	Programmable Logic Controller
FC	Function Call of the classical program languages	POU	Program Organisation Unit
FUN	FUNCTION acc. to IEC 61131-3	PROG	PROGRAM, main program acc. to IEC61131-3
IEC	International Electrotechnical Commission	SFC	Sequential Function Chart
IL	Instruction List	ST	Structured Text
		WinSPS	Bosch software for programming PLCs

A.2 Index

A

Access paths, 6–10
 Accumulator, 8–2
 Actual parameter, 11–6
 Actual Result, 8–2
 Addition, 8–15
 Address, 5–5
 AND, 8–12
 ANDN, 8–12
 ANY, 7–13
 Arithmetic functions, 12–5
 Arithmetic operators, 8–15
 ARRAY, 7–22
 Array, 7–22
 ASCII constants, 7–5
 ASCII string, 7–10
 Assignment, 8–10, 9–4
 AT, 7–19
 Attributes, 5–6, 7–30

B

Base byte address, 11–15
 Base Level Certificate, 3–8
 Base-specific number, 7–5
 Binary links, 12–8
 Binary numbers, 7–4
 Binary operations, 8–12
 Bistable elements, 8–11, 12–14
 Bit sequence, 7–10
 Body, 6–12
 BOOL, 7–10
 Boolean data, 7–4
 Boolean functions, 12–8
 Boolean operators, 8–12
 Bosch IL, 3–2, 11–4
 BYTE, 7–10

C

CAL, 8–21
 CALC, 8–21
 CALCN, 8–21
 Calendar date, 7–6, 7–10
 Call interface, 6–14
 Call parameter list, 11–5
 call-by-reference, 6–15
 call-by-value, 6–15
 Carry, 8–2
 CASE, 9–7
 Causes of errors, 13–16
 Certificate, 3–8
 Check, 10–1
 Check module, 10–1
 Checklists, 13–1
 Classical programming language, 3–2, 11–1
 Cleaning up, 11–9
 Cold reset, 7–17
 Comment, 7–8
 Common elements, 3–5, 5–2, 13–1

Comparison, 12–9
 Comparison operators, 8–18
 Compile module, 10–1
 Compliance Tables, 13–1
 Conditional execution, 8–18, 9–5
 Conditional FB call, 8–21
 Conditional jump, 8–20
 Conditional return jump, 8–24
 Configuration, 3–7
 Conformity tables, 13–1
 CONSTANT, 7–30
 Constant definition, 5–18, 7–31
 Counter, 12–16
 Counting loop, 9–8
 CR, 8–2
 Create new project, 10–2
 CTD, 12–16
 CTU, 12–16
 CTUD, 12–17
 Current Result, 8–2

D

Data buffer, 11–12
 Data field, 11–12
 Data model, 7–1
 Data module, 4–4, 10–4
 Data structure, 7–28
 Data Type, 5–5, 5–9, 7–9
 Data type conversion, 12–3
 Data width, 7–19
 DATE, 7–10
 Date, 7–6
 DATE_AND_TIME, 7–10
 Declaration, 7–15
 Declaration part, 6–8
 Declaration tables, 5–3
 Default settings, 4–1
 Default value, 7–9
 DEFINE Editor, 5–18
 Delimiter, 7–7
 Derived data types, 7–11
 DINT, 7–10
 Directly shown address, 7–19
 Division, 8–17
 DM, 10–4, 11–6
 Documentation, 1–6
 Downward counter, 12–16
 DWORD, 7–10

E

Edge detection, 12–15
 Elementary data types, 7–9
 ELSE, 9–6, 9–7
 ELSEIF, 9–6
 Emergency–STOP–devices, 1–5
 End Of Line, 7–7
 End section, 10–4
 Entire program, 10–2

ENUM, 5-16, 7-12
 Enumeration, 7-12
 Error messages, 5-12
 Exclusive OR:, 8-14
 EXIT, 9-12
 Exponential numbers, 7-5
 Extensibility , 12-3
 External variable, 6-10

F

F_EDGE, 7-30
 F_TRIG, 12-15
 Falling edge, 12-15
 FALSE, 7-10
 FB, refer to: Function block, 6-5
 FBD, 3-1
 FC, 10-4
 Figure with a sign, 7-10
 Figure without a sign, 7-10
 Firmware version, 3-8
 Flipflop, 8-11, 12-14
 Floating point numbers, 7-4, 7-10
 Floppy disk drive, 1-6
 FOR, 9-8
 Formal parameter, 6-14
 FUN, refer to: Function, 6-7
 FUN ReturnType, 5-4
 FUNCTION, 6-7
 Function, 6-7
 Call, 6-19, 8-22
 Extensibility , 12-3
 Overloaded, 12-1
 Standard functions, 12-1
 Function block, 6-5
 Call, 6-17, 8-20, 11-5
 Changing the call, 11-8
 Deleting the call, 11-9
 Instance, 6-21
 Memory, 6-22
 Standard function block, 12-12
 Validity, 6-22
 Function Block Diagram, 3-1
 Function value, 6-14
 FUNCTION_BLOCK, 6-5

G

Gate, 8-12
 Generate project, 10-2
 Generic data types, 7-13, 12-1
 Global type definition, 5-14, 11-14
 Global variable, 6-10
 GRAFCET, 3-3

H

Hard disk drive, 1-6
 Hexadecimal numbers, 7-4
 Hot Restart, 7-17

I

Identifiers, 7-3, 11-17
 IEC 61131-3, 3-1

IEC file, 4-4
 IEC IL, 3-2
 IEC_FUNCTION_BLOCK, 11-9
 IF, 9-6
 IL, 3-1, 8-1
 Current Result, 8-2
 Instruction set, 8-7
 Instructions, 8-1
 Label, 8-5
 Nesting, 8-5
 Sequence, 8-4
 IM, 10-4
 Initial value, 5-5, 5-10, 6-15, 7-11, 7-16
 Initializing the PCL, 7-9, 7-16
 Input and output variable, 6-10
 Input parameter, 6-14
 Input variable, 6-9
 Inputs, 7-19
 Installation, 4-1
 Instance, 6-21, 6-22, 10-6, 11-4
 Instruction list, refer to: IL, 8-1
 Instruction set, 8-7
 Instructions
 IL, 8-1
 ST, 9-3
 Instructions part, 5-11, 6-12
 INT, 7-10
 Interrupt, 8-2
 Iterations, 9-5

J

JMP, 8-19
 JMPC, 8-20
 JMPCN, 8-20
 Jump statements, 8-19, 9-3

K

Key words, 7-2

L

Label, 8-1, 8-5
 Ladder Diagram, 3-1
 Language elements, 7-1, 13-14
 LD, 3-1, 8-9
 LDN, 8-9
 Library, 6-6, 6-8
 License, 4-1
 Link, 8-12
 Link modules, 10-2
 Linker, 10-2
 Links, 12-8
 LINT, 7-9
 Literals, 7-4
 Load instructions, 8-9
 Load modules, 10-5
 Load program, 10-5
 Local variable, 6-9
 Logical links, 12-8
 Logical operations, 8-12
 Logical value, 7-10
 LREAL, 7-10

LWORD, 7–9

M

Main program, 6–4, 10–2
 Marker, 7–19
 Mathematical operators, 8–15
 Memory, 6–22
 Memory address, 11–15
 Mixed programs, 11–2
 Module Calls, 6–12
 Modules, 6–1
 Monitor, 10–6
 Monitor data, 5–7
 Multi-selection, 9–7
 Multielement variables, 7–22
 Multiplication, 8–16
 Multitasking, 6–4

N

Names, 7–2
 Negation, 8–2
 Nesting levels, 8–5
 Number base, 7–5
 Numbers, 7–4
 Numeric functions, 12–5
 Numeric literals, 7–4

O

Octal numbers, 7–4
 Off-delay, 12–19
 OM1, 11–2
 On-delay, 12–19
 Operand, 8–2
 Operator, 8–2
 Operators
 Arithmetic, 8–15
 Assignment, 8–10
 Boolean, 8–12
 Call of FBs, 8–20
 Comparison, 8–18
 IL (Programming language), 8–7
 Jump, 8–19
 Load instructions, 8–9
 Return jump, 8–23
 ST (Programming language), 9–1
 OR, 8–13
 Order of precedence, 9–1
 Organisation module, 11–2
 ORN, 8–13
 Output parameter, 6–14
 Output variable, 6–9
 Outputs, 7–19
 Overflow, 8–2
 Overloaded function, 7–14, 12–1

P

Parameter, 6–14
 Parenthesis, 8–5
 Physical addresses, 5–5, 7–19, 11–11, 11–17
 PLC addresses, 7–19
 PM, 11–6

Pointer, 6–15, 11–16
 Post-loading, 10–5
 POU, 6–1
 Call parameter, 6–14
 Calls, 6–12
 POU name, 5–3
 POU type, 5–3, 6–3
 Power failure, 7–17
 Priority, 9–1
 PROG, 6–4
 PROGRAM, 6–4
 Program file, 4–3, 10–4
 Program module, 4–3, 11–2
 Program Organization Units, refer to: POU, 6–1
 Programming languages, 3–1
 Project specifications, 10–4
 Pulse, 12–18

Q

Qualified personnel, 1–2
 Quotation marks, 7–5

R

R, 8–11
 R_EDGE, 7–30
 R_TRIG, 12–15
 RANGE, 7–12
 Range, 7–12
 READ_ONLY, 7–30
 READ_WRITE, 7–30
 REAL, 7–10
 Real time clock, 12–20
 Recursion, 6–13
 Register, 8–2
 Release, 1–6
 Remanence characteristic, 7–17
 REPEAT, 9–10
 Reset dominant, 12–14
 Resetting command, 12–14
 Resetting instruction, 8–11
 Resource, 3–7
 Restart, 7–17
 Result of logic operations, 8–2
 RET, 8–23
 RETAIN, 7–17, 7–30
 RETC, 8–24
 RETCN, 8–24
 RETURN, 9–5
 Return jump, 8–23, 9–5
 Return value, 6–14
 return-by-value, 6–15
 Rising edge, 12–15
 RLO, 8–2
 Rotate, 12–7
 RS, 12–14
 RTC, 12–20

S

S, 8–10
 Safety instructions, 1–4
 Safety markings, 1–3

- Selection, 9-5, 9-6, 12-8
- Sequential Function Chart, 3-1, 11-10
- Set dominant, 12-14
- Set instruction, 8-10
- Setting command, 12-14
- SFC, 3-1
- Shift, 12-7
- Sign, 7-5
- Single element variable, 7-22
- SINT, 7-10
- Software version, 3-8
- Space, 7-7
- Special characters, 7-5, 7-7
- Special marker, 11-12
- SR, 12-14
- ST, 8-10, 9-1
 - Assignment, 9-4
 - Expressions, 9-1
 - Function block call, 9-4
 - Function call, 9-2
 - Instructions, 9-3
 - Loop end, 9-12
 - Loops, 9-8
 - Operands, 9-1
 - Operators, 9-1
 - Priority, 9-1
 - Return jump, 9-5
 - Selection, 9-6, 9-7
- Standard fulfillment, 13-1
- Standard function block, 12-12
 - Bistable elements, 12-14
 - Counter, 12-16
 - Edge detection, 12-15
 - Timer, 12-18
- Standard functions, 12-1
 - Arithmetic, 12-5
 - Shift, 12-7
 - Boolean, 12-8
 - Comparison, 12-9
 - Numeric, 12-5
 - Selection, 12-8
 - Strings, 12-10
 - Type conversion, 12-3
- Standard operation, 1-1
- Standard value, 7-9
- Start section, 10-4
- Status bit, 8-2
- Step action, 11-10
- STN, 8-10
- STRING, 7-10
- String literal, 7-5
- String terminator, 7-5
- String variables, 7-21
- Strings, 11-17, 12-10
- STRUCT, 5-15, 7-28
- Structure, 7-28
- Structured Text, refer to: ST, 9-1
- Subtraction, 8-16
- Symbol file, 4-3, 10-4, 11-11
- Symbolic operands, 11-13

T

- Task, 3-7
- Terminator, 7-5
- Text constants, 7-12
- THEN, 9-6
- TIME, 7-10
- Time, 7-6, 7-10
- Time duration, 7-6, 7-10
- Time literals, 7-6
- Time of day, 7-6
- TIME_OF_DAY, 7-10
- Timer, 12-18
- TOF, 12-19
- TON, 12-19
- TP, 12-18
- Trademarks, 1-6
- Translate module, 10-1
- TRUE, 7-10
- TYPE, 5-14, 7-11
- Type conversion, 12-3
- Type definition, 5-8, 7-11
- Type editor, 5-14
- Type name, 5-8

U

- UDINT, 7-10
- UINT, 7-10
- ULINT, 7-9
- Umlauts, 7-1, 7-6
- Unconditional FB call, 8-21
- Unconditional jump, 8-19
- Unconditional return jump, 8-23
- Understroke, 7-3, 7-5, 7-6
- UNTIL, 9-10
- Upward and downward counter, 12-17
- Upward counter, 12-16
- USINT, 7-10

V

- Validity, 6-22
- VAR, 6-9
- VAR_ACCESS, 6-10
- VAR_EXTERNAL, 6-10
- VAR_GLOBAL, 6-10
- VAR_IN_OUT, 6-10
- VAR_INPUT, 6-9
- VAR_OUTPUT, 6-9
- Variable, 7-15
 - Access, 7-18
 - ARRAY, 7-22
 - Attributes, 7-30
 - Data structure (STRUCT), 7-28
 - Initialization, 7-16
 - Physical address, 7-19
 - String, 7-21
- Variable attributes, 7-30
- Variable declaration, 5-4
- Variable type, 5-4, 6-9, 6-15

W

- Warm start, 7-17

WHILE, 9–10
WORD, 7–10
Working register, 8–2

X
XOR, 8–14

Z
Zero, 8–2

Bosch Rexroth AG

Electric Drives and Controls
Postfach 11 62
64701 Erbach
Berliner Straße 25
64711 Erbach
Deutschland
Tel.: +49 (0) 60 62/78-0
Fax: +49 (0) 60 62/78-4 28
www.boschrexroth.com

Australia

Bosch Rexroth Pty. Ltd.
3 Valediction Road
Kings Park NSW 2148
Phone: +61 (0) 2 98 31 77 88
Fax: +61 (0) 2 98 31 55 53

USA

Bosch Rexroth Corporation
5150 Prairie Stone Parkway
Hoffmann Estates, Illinois 60192
Phone: +1 (0) 847 6 45-36 00
Fax: +1 (0) 847 6 45-08 04

United Kingdom

Bosch Rexroth Ltd.
Broadway Lane, South Cerney
Cirencester GL7 5UH
Phone: +44 (0) 1285-86 30 00
Fax: +44 (0) 1285-86 30 03

Canada

Bosch Rexroth Canada Corp.
490 Prince Charles Drive South
Welland, Ontario L3B 5X7
Phone: +1 (0) 905 7 35-05 10
Fax: +1 (0) 905 7 35-56 46