PLC programming according to the IEC 61 131-3 standard in the Mosaic environment

10th edition
November 2007

History of versions

Date	Issue	Description of changes
August 2004	1	First version
October 2004	2	The description of standard libraries was added
January 2005	3	Adaptation for Mosaic Help performed.
February 2005	4	Example correction 3.6.2.5 – the word ”DO“ added
April 2005	5	Table.3.3 Special signs in strings added Correction of the SINT type range in chapter 3.2.1 in Table.3.5 Table.3.18 added function for calling above character string Added chapter 3.7.2 Library of functions above character string
November 2005	6	Description of library of conversion functions added Table 3.20 Standard functions with date and time function amended PTR_TO data type description added
February 2006	7	Description of data types and variables expanded Description of library of arithmetic functions
March 2006	8	Basic programming principles according to standard added IL language description added Library description transferred into separated document TXV 003 22
November 2006	9	Description of directives added
November 2007	10	LD and FBD graphic languages description added

1 INTRODUCTION

1.1 The IEC 61 131 standard

The IEC 61 131 standard for programmable control systems consists of 5 parts and represents an overview of requirements for advanced control systems. It is independent of any particular organization or company and has a wide international support. Constituent parts of the standard deal with both hardware and software of these systems.

In the Czech Republic, the relevant parts of this standard have been adopted under the following numbers and titles:

ČSN EN 61 131-1 Programmable controllers - Part 1: General information
ČSN EN 61 131-2 Programmable controllers - Part 2: Requirements for equipment and tests
ČSN EN 61 131-3 Programmable controllers - Part 3: Programming language
ČSN EN 61 131-4 Programmable controllers - Part 4: User support
ČSN EN 61 131-5 Programmable controllers - Part 5: Communication
ČSN EN 61 131-7 Programmable controllers - Part 7: Programming of fuzzy control

Within the European Union, these standards have been adopted under number EN IEC 61 131.

The IEC 61 131-3 standard defines programming languages and is the third part from the IEC 61 131 family of standards and represents the first real attempt to standardize programming languages for industrial automation.

The 61 131-3 standard can be viewed from different points of view, such as it is a result of a heavy work of seven international companies participating in the formulation of the standard and having a ten year experience in the field of industrial automation. Another point of view is that it contains about 200 pages with text and about 60 tables. A team belonging to the working group SC65B WG7 of the International standardization organization IEC (International Electrotechnical Commission) participated in its formulation. The result of its work is *Specification of syntax and semantics of unified family of programming languages, including general software model and structurising language*. This standard was adopted as the directive by majority of important PLC manufacturers.

1.2 Terminology

The family of standards for programmable controllers was adopted in the Czech Republic, but has not been translated into Czech yet. For this reason, the terminology in this manual is used as it has been lectured at the Czech Technical University in Prague. At the same time, English terminology is used in the text, the task of which is to assign Czech terms to the English source.

1.3 The basic idea of the IEC 61 131-3 standard

The IEC 61 131-3 standard is the third part of the IEC 61 131 family of standards. It can be divided into two basic parts:

- Mutual features
- Programming languages

1.3.1 Mutual features

Data types

Within the frame of mutual features, data types are defined. Defining data types helps prevent errors already at the beginning of the project. It is necessary to define the types of all used parameters. Usual data types are **`BOOL`**, **`BYTE`**, **`WORD`**, **`INT`** (Integer), **`REAL`**, **`DATE`**, **`TIME`**, **`STRING`** etc. It is possible to derive your own user data types from these basic ones, i.e. derived data types. This way it is possible, e.g. to define an independent analog input channel data type and repeatedly use it under a defined name.

Variables

Variables may be explicitly assigned to hardware addresses (e.g. to inputs, outputs) only in configurations, sources or programs. This way a great level of hardware independency is achieved together with the possibility of using the software repeatedly on different hardware platforms.

The sphere of action of the variables is standardly limited to the organizational unit in which they were declared (variables are local there). This means that their names may be used in different parts without limitations. Many further errors are eliminated by this precaution. Should variables have a global sphere of action, e.g. within the whole project, then they must be declared as global (VAR_GLOBAL). An initial value can be assigned to the parameters during the start or cold start of a process or machine so to be able to set the correct initial state.

Configuration, resources and tasks

The highest level of a complete software solution of a specific control problem is called a *configuration*. The configuration is dependent on the specific control system, including hardware lay-out, e.g. processor unit types, memory areas assigned to input and output channels and characteristics of the system program equipment (the operation system).

Within the frame of the configuration, we can then define one or more so called *resources*. We can view the resources as a type of device that is able to execute IEC programs.

We can define one or more so called *tasks* in the resource. The tasks control executions of program groups and/or of function block. These units may be executed either periodically or after the creation of a special launch event which can be for instants a changed variable.

Programs are created from a row of various software items which are registered in one of the languages defined by the standard. A program is often created from a network of functions and function blocks which are able to exchange data. Functions and function blocks are the foundation stones which contain data structures and algorithms.

Program organization units

Functions, function blocks and programs are within the IEC 61 131 standard referred to as *Program Organization Units*. Sometimes the abbreviation POU is used for this frequently used and important term.

Functions

The IEC 61 131-3 standard defines standard functions and user defined functions. Standard functions are e.g. ADD for summing, ABS for absolute value, SQRT for square root, SIN for sine and COS for cosine. After user functions are defined, they can be used repeatedly.

Function blocks

We can understand function blocks as integrated circuits which represent a hardware solution for a specialized control function. They contain algorithms and data, so they can keep a history record (in contrast to functions). They have a clearly defined interface and hidden internal variables, analogous to integrated circuits or black boxes. They are able to unambiguously separate two levels of programmers or service personnel. A classic example of function blocks are e.g. temperature regulation loops or PID regulators.

Once a function block is defined, it can be repeatedly used in the given program, in a different program or even in a different project. It is universal and has unlimited use. Function blocks may be programmed in a random programming language defined by the standard. So they can be fully user defined. Derived function blocks are based on standard function blocks but it is possible, within the standard rules, to create new customer function blocks.

The interface of functions and function blocks is described in the same way: Between a declaration determining block name and a declaration for block end, lies a list of declarations of input and output declarations and a unique code in the so called block body.

Programs

According to the above mentioned, it is possible to say that a program is really a network of functions and function blocks. A program may be programmed in a random programming language defined by the standard.

1.3.2 Programming languages

Four programming languages are defined by the standard. Their semantics and syntax are exactly defined and no room for inexact expressions is left. By being able to work with these languages a new door opens to the use of a wide range of control systems which are based on this standard.

Programming languages can be divided into the following basic categories:

Text languages
IL - Instruction List language
ST - Structured Text language

Graphic languages
LD - Ladder Diagram language (contact diagram language)
FBD - Function Block Diagram language

For the first overview, the same logical function is shown of Figure 1.1.; the sum of variable A and negated variable B with the result saved into variable C; expressed in all four programming languages.

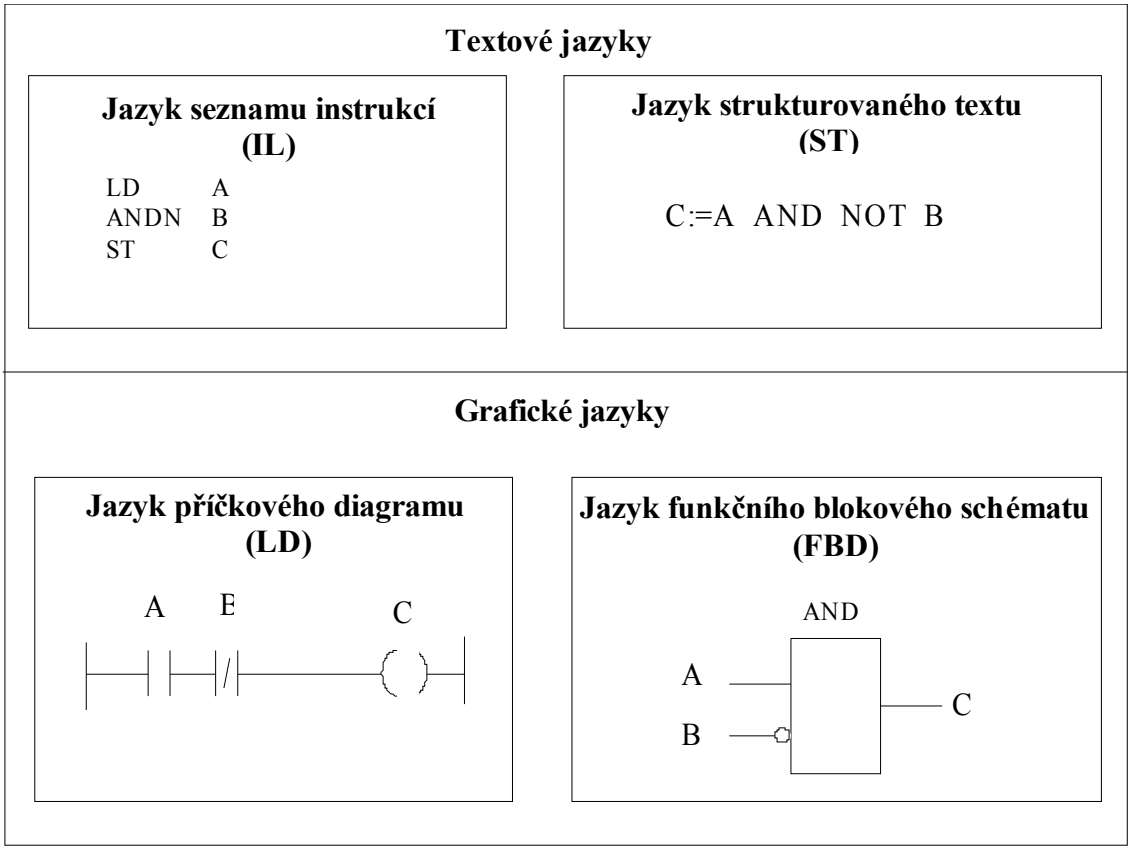


Figure 1 ANDN logical function in all four basic languages

The choice of programming languages depends on the programmer's experience, on the type of problem, on the level of problem description, on the control system structure and on many

more factors, e.g. industry type, custom practice of company implementing the control system, team co-worker experience, etc.

All of the four basic languages (IL, ST, LD and FBD) are interconnected. Applications programmed using them, create a specific basic set of information to which a great volume of technical know-how is connected to. They create a basic communication tool for the cooperation between various industries and fields.

LD - Ladder Diagram language
- originates in the USA. Based on graphic representation of relay logic.

IL - Instruction List language
- European version of the LD. A text language similar to an assembler.

FBD - Function Block Diagram language
- very close to the processing industry. Expresses behavior of functions, function blocks and programs as a set of interconnected graphical blocks, similar to electronic circuit diagrams. It is a specific system of items which processes signals.

ST - Structured Text language
- a very powerful programming language which is based on the well known languages Ada, Pascal and C. It contains all important components of a modern programming language, including branching (IF-THEN-ELSE and CASE OF) and iterative loops (FOR, WHILE and REPEAT). These may be immersed. This language is an excellent tool for defining complex function blocks which then may be used in whichever programming language.

It is known that two approaches exist for systematic programming: top-down or bottom-up.

The mentioned standard supports both approaches. We can either specify the whole application and divide it into parts (subsystems), declare variables, etc. or we can start programming the application bottom-up, e.g. through derived (user) functions and function blocks. Whatever method we choose, the Mosaic development environment, which complies with IEC 11 131-3 standard, will support and help creating whole applications.

2 BASIC TERMS

This chapter briefly explains the meaning and use of basic terms when programming in accordance to standard IEC 61 131-3. These terms will be explained using simple examples. A detailed description of the terms being explained can be found in further chapters.

2.1 Basic program blocks

The basic term when programming according to the IEC 61 131 standard is the term ***Program Organisation Unit (POU)***. As it can be seen from the name, the Program Organisation Unit is the smallest independent part of a user program. POUs can be delivered by the manufacturers of control systems or they can be created by the user himself. Each POU can call another POU and during the call operation, it can pass optionally one or more parameters onto the POU being called.

There are three basic types of POUs :

- ***Function*** *(FUN)*
- ***Function block*** *(FB)*
- ***Program*** *(PROG)*

The elementary POU is ***function***, the main feature of which is that if it is called with the same input parameters, it must produce the same result (function value). This function can return one result only.

Another type of POU is ***function block***, which, when compared with the function, can remember some values from previous calls (status information, for example). These then can affect the result. The main difference between the function and the function block is the capability of the function block to have memory to store values of some variables. Functions do not have this properties and their result is unequivocally determined by input parameters when calling the function. The function block can (unlike the function) return more than one result.

The last type of POU is ***program***, representing the highest programming unit in the user program. The PLC central unit can process more programs and the ST language has means for the definitions of program initialisations (at what period of time, with what priority, etc. the program is executed).

Each POU consists of two basic parts: ***declaration*** and ***executive*** as it can be seen on Figure 2.1. In the declaration past of the POU, variables necessary for POU operation are defined. The executive part contains statements for the execution of the algorithm in question.

The definition of POU on Figure begins with the key word *PROGRAM* and is ended by the key word *END_PROGRAM*. These key word define the range of the POU. Behind the key word *PROGRAM* is the name of the POU, followed by the POU declaration part. The declaration part contains definitions of variables given between the key words *VAR_INPUT* and *END_VAR* or *VAR*

and *END_VAR* as the case may be. The executive part of the POU contains statements of the ST language for variables processing. The texts between characters *(** and ***)* are comments.

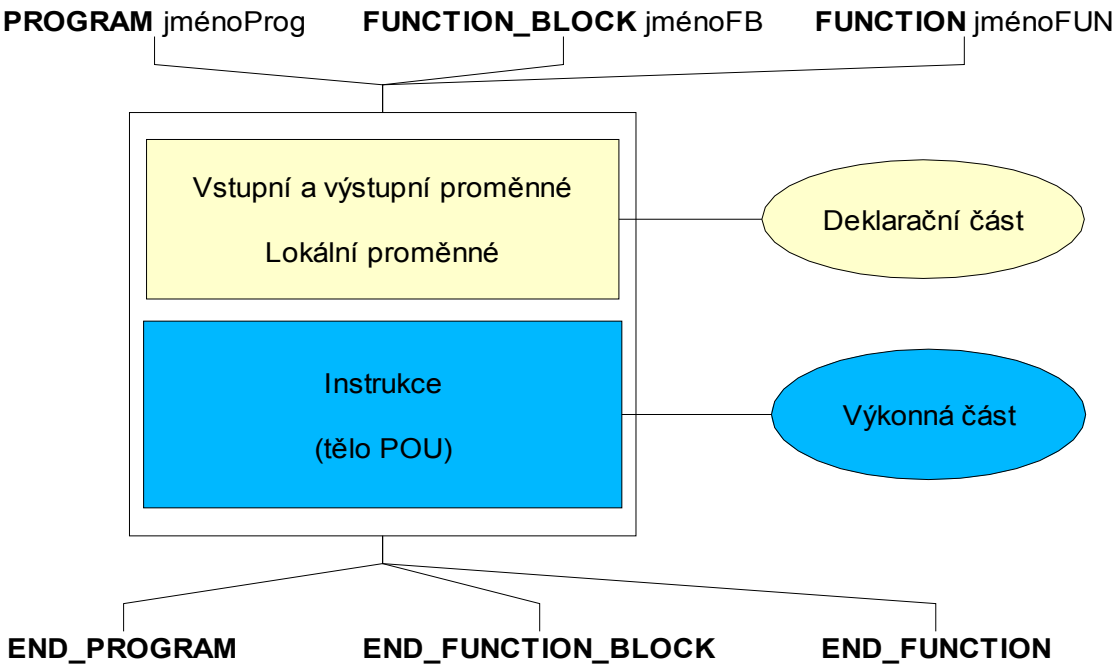


Figure 2 Basic POU structure

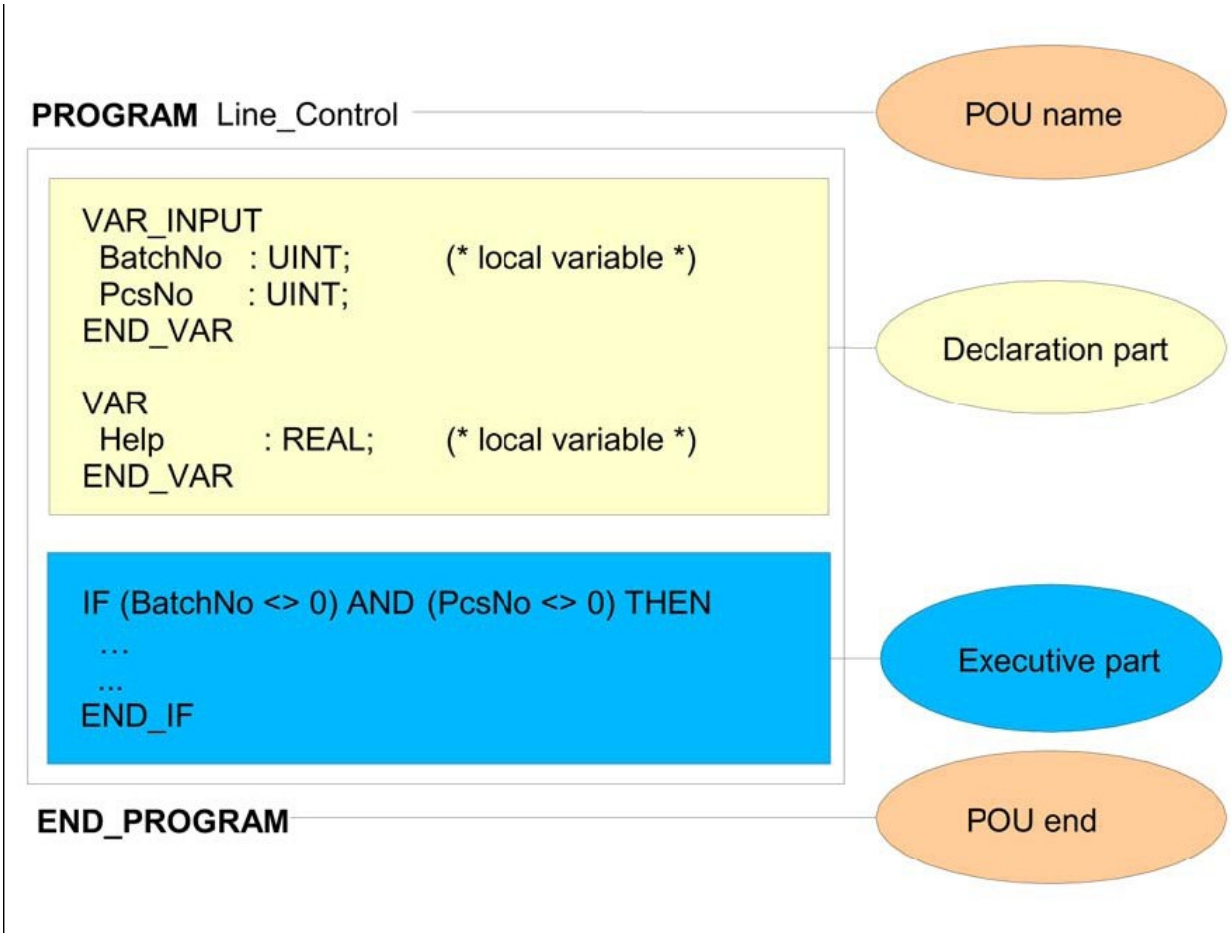


Figure3 Basic POU PROGRAM structure

2.2 POU variables declaration

Variables are used to store and process information. Each ***variable*** is defined by ***the name*** and ***data type***. The data type specifies the size of the variable in the memory and at the same time, it defines to a great extent the way of variable processing. For the definitions of variables, standard data types are available (Bool, Byte, Integer, ...). The use of these types depends on what kind of information will be saved at the variable (e.g. Boolean type for information YES - NO, **INT** type for saving integers with sign, etc.). The user has a possibility to define his/her own data types. The position of the variables in the memory is ensured automatically by the programming environment. If necessary, the position of the variables in the memory can be defined by the user, too.

According to the purpose of use, variables can be divided into ***global*** and ***local*** ones. Global variables are defined outside the POU and can be used at any POU (they are "visible" from any POU). Local variables are defined inside the POU and they can be used within this POU (they are not "visible" from other POUs).

Finally, variables can be used to pass parameters during POU calling. In such cases, we talk about ***input*** or ***output*** variables, as the case may be.

Example 1 POU variables declaration

```
FUNCTION_BLOCK PromExampleDeclaration

  VAR_INPUT                   (* input variables *)
    logCondition  : BOOL;     (* binary value *)
  END_VAR
  VAR_OUTPUT                  (* output variables *)
    Result        : INT;      (* integer value with sign *)
  END_VAR
  VAR                         (* local variables *)
    CheckSum            : UINT;     (* integer value *)
    PartResult    : REAL;     (* real value *)
  END_VAR

END_FUNCTION_BLOCK
```

In example 2.1., a POU input variable is defined, the name of it is *logCondition* and is of the BOOL type, which means it can contain the TRUE value (logic "1") or the FALSE value (logic "0"). This variable serves as the input parameter passed during POU calling.

Another defined variable is the output one, the name of which is ***result*** and is of **INT** type (integer) so it can contain integer values in the range from –32 768 to +32 767. At this variables, the value is passed onto a superordinate POU.

The variables defined between the key words ***VAR*** and ***END_VAR*** are local ones and they can therefore be used within the POU in question only. The variable **C*heckSum*** is of **UINT** type (unsigned integer) and can store integers in the range from 0 to 65535. The variable ***PartResult*** is of **REAL** type and is used for work with real numbers.

2.3 POU executive part

The POU executive part follows the declaration one and contains statements and instructions, which are processed by the PLC central unit. In exceptional cases, the POU definition does not need to contain any declaration part and, in this case, the executive part is written immediately behind the POU start definition. An example can be a POU working only with global variables, which is not an ideal solution, but it can exist.

The POU executive part can contain call instructions of further POUs. During the execution of the call instructions, there can be passed parameters for the functions being called or for function blocks, as the case may be.

2.4 Program example

Example 2 Program example

```
VAR_GLOBAL
  // inputs
  sb1 AT %X0.0,
  sb2 AT %X0.1,
  sb3 AT %X0.2,
  sb4 AT %X0.3   : BOOL;

  // outputs
  km1 AT %Y0.0,
  km2 AT %Y0.1,
  km3 AT %Y0.2,
  km4 AT %Y0.3   : BOOL;
END_VAR


FUNCTION_BLOCK fbStartStop
//-------------------------------------------------------------------------
  VAR_INPUT
    start        : BOOL R_EDGE;
    stop         : BOOL R_EDGE;
  END_VAR
  VAR_OUTPUT
    vystup       : BOOL;
  END_VAR

  output := ( output OR start) AND NOT stop;
END_FUNCTION_BLOCK



FUNCTION_BLOCK fbMotor
//-------------------------------------------------------------------------
  VAR_INPUT
    motorStart   : BOOL;
    motorStop    : BOOL;
  END_VAR
  VAR
    startStop    : fbStartStop;
    motorIsRun   : BOOL;
    startingTime : TON;
  END_VAR
  VAR_OUTPUT
    star         : BOOL;
    triangle     : BOOL;
  END_VAR

  startStop( start := motorStart, stop := motorStop,
             output => motorIsRun);
  startingTime( IN := motorIsRun, PT := TIME#12s, Q => triangle);
END_FUNCTION_BLOCK
```

```
PROGRAM Test
//------------------------------------------------------------------------------
  VAR
    motor1       : fbMotor;
    motor2       : fbMotor;
  END_VAR

  motor1( motorStart := sb1, motorStop := sb2,
          star => km1, triangle => km2);
  motor2( motorStart := sb3, motorStop := sb4,
          star => km3, triangle => km4);
END_PROGRAM


CONFIGURATION exampleProgramST
  RESOURCE CPM
    TASK FreeWheeling(Number := 0);
    PROGRAM prg WITH FreeWheeling : Test ();
  END_RESOURCE
END_CONFIGURATION
```

3 COMMON ELEMENTS

This chapter describes the syntax and semantics of the basic elements of programming languages for the PLC systems according to the IEC 61 131-3 standard.

Syntax describes the elements that are available for programming PLCs and the ways, how they can be combined.

Semantics then formulates their meaning.

3.1 Basic elements

Each program PLC program consists of basic ***simple elements***, certain smallest units, from which declarations and statements are created. These simple elements can be divided into:

- Delimiters
- Identifiers
- Literals
- Keywords
- Comments

For a better transparency, bolt type face is used for the keywords, so that the structure of declarations and statements can be expressed better. Additionaly, they have different colours in the Mosaic environment.

Delimiters are special characters (such as (,), =, :, space, etc.) with different meanings.

Identifiers are alphanumeric character strings used for the expression of user functions, labels or POUs (such as Temp_N1, Switch_On, **Step4**, Move_right, etc.).

Literals are used for direct representation of variable values (such as *0,1*; *84*; *3,79*; *TRUE* ; *green* etc.).

Keywords are standard identifiers (such as FUNCTION, REAL, VAR_OUTPUT, etc.). Their exact formulation and meaning corresponds to the standard IEC 61 131-3. The keywords must not be used for creation of any user names. For typing of keywords, both lower-case and upper-case letters can be used including any of their combinations. Among the reserved keywords belong:

- *names of elementary data types*
- *names of standard functions*
- *names of standard function blocks*
- *names of input parameters of standard functions*
- *names of input and output parameters of standard function blocks*
- *IL and ST language elements*

All reserved keywords are shown in annex H of the standard IEC 61 131-3.

Comments do not have any syntactic or semantic meaning, but they are an important part of program documentation. A comment can be written anywhere, where the space character can be

typed. During compilation, these strings are ignored, and so they can contain also the characters of national alphabets. The language compiler can recognize two types of comments:

- *General comments*
- *Line comments*

General comments are character strings beginning with **(*** and terminated with ***)**. This allows writing all the necessary types of comments as you can see from the example below. ***Line comments*** are character strings beginning with // and terminated by the line end. The advantage of line comments is the possibility to be nested into general comments (see lines with definition of variables *Help1* a *Help2* in the following example, which will be considered as a comment and will not be compiled by the compiler).

Example 3 Comments

```
(************************************************
this is an example
of a multi-line comment
************************************************)
VAR_GLOBAL
  Start,              (* general comment, e.g. : button START *)
  Stop      : BOOL;    (*STOP button*)
  Help     : INT;     // line comment

(*
  Help1    : INT;     // nested line comment
  Help2    : INT;
*)

END_VAR
```

3.1.1 Identifiers

An identifier is a string of letters (lower-case or upper-case letters), numbers underline characters and is used to name the following elements of language ST:

- *constant name*
- *names of variables*
- *names of derived data types*
- *names of functions, function blocks and programs*
- *names of tasks*

An identifier has to begin with a letter or underline character and must not contain space characters. The national alphabet characters (letters with breves and acutes) are not allowed to be used in the identifiers. The location of the underline character is of importance, for example „BF_LM“ and „BFL_M“ are two different identifiers. There are not allowed more underline characters following each other. The size of the letter in an identifier does not play any role. For example motor_off equals to MOTOR_OFF or Motor_Off. If motor_off is a name of a variable, then all the representations will mean the same variables.

The maximum length of an identifier is 64 characters.

Table.1 Examples of valid and invalid identifiers

Valid identifiers	Invalid identifiers
XH2	2XH
MOTOR3F, Motor3F	3FMOTOR
Motor3F_Off, Motor3F_OFF	MOTOR3F__Off
SQ12	SQ$12
Delay_12_5	Delay_12.5
Rek	Řek
_3KL22	__3KL22
KM10a	KM 10a

Example 4 Identifier

```
TYPE
  _Phase          : ( star, triangle);
END_TYPE

VAR_GLOBAL CONSTANT
  _3KL22          : REAL := 3.22;
END_VAR

VAR_GLOBAL
  SQ12  AT %X0.0  : BOOL;
  KM10a AT %Y0.0  : BOOL;
  XH2             : INT;
END_VAR

FUNCTION_BLOCK MOTOR3F
  VAR_INPUT
    Start         : BOOL;
  END_VAR
  VAR
    Delay_12_5    : TIME;
    Status        : _Phase;
  END_VAR
  VAR_OUTPUT
    Motor3F_Off   : BOOL;
  END_VAR

END_FUNCTION_BLOCK
```

3.1.2 Literals

Literals are used for the direct representation of variable values.
Literals can be divided into three groups:

- *numeric literals*
- *character strings*
- *time literals*

If we want to emphasize a data type of a recorded literal, it is possible to record the literal starting with a data type name followed by the sign # (e.g. **`REAL#12.5`**). In case of time literals, the stating of the type is necessary (e.g. **`TIME#12h20m33s`**).

3.1.2.1 Numeric literals

A numeric literal is defined as a number (constant) in the decimal system or in a systém with another base than ten (e.g. binary system, octal or hexadecimal system). Numeric literals can be divided into integer and real literals. A simple underline character located between numbers of a numeric literal does not influence its value, it is allow for improving readability. Some examples of numeric literals are shown in Table 3.2 .

Table.2 Examples of numeric literals

Description	Numeric literal – example	Note
Integer literal	14	
	INT#–9	-9
	12_548_756	12 548 756
Real literal	–18.0	
	REAL#8.0	
	0.123_4	0,1234
Real literal s exponentem	4.47E6	4 470 000
	652E–2	6,52
Literal with base 2	2#10110111	183 decimally
Literal with base 8	USINT#8#127	87 decimally
Literal with base 16	16#FF	255 decimally
Bool literal		
FALSE	FALSE	
	BOOL#0	0
TRUE	TRUE	
	BOOL#1	1

Example 5 Numeric literals

```
VAR_GLOBAL CONSTANT
  Const1          : REAL  := 4.47E6;
  Const2          : LREAL := 652E-2;
END_VAR

VAR_GLOBAL
  MagicNum        : DINT  := 12_548_756;
  Amplitude       : REAL  := 0.123_4;
  BinaryNum       : BYTE  := 2#10110111;
  OctalNum        : USINT := 8#127;
  HexaNum         : USINT := 16#FF;
  LogicNum        : BOOL  := TRUE;
END_VAR

FUNCTION Parabola : REAL
  VAR_INPUT
    x,a,b,c   : REAL;
  END_VAR

  IF a <> 0.0 THEN
    Parabola := a*x*x + b*x + c;
  ELSE
    Parabola := 0.0;
  END_IF;
END_FUNCTION


PROGRAM ExampleLiterals
  VAR
    x,y    : REAL;
  END_VAR

  y := Parabola(x := x, a := REAL#2.0, b := Const1, c := 0.0 );
END_PROGRAM
```

3.1.2.2 Character string literals

A character string is a sequence of no string (empty string) or of more characters, starting and ending with (‘). Examples: ‘‘ (empty string), ‘temperature‘ (not empty string of the length eleven, containing word temperature).

The dollar characters, $ is used as a prefix allowing introduction of special characters in a string. Special characters not being printed, are used for example for text formatting for a printer or on a display. If the dollar character is before two hexadecimal number, the string is interpreted as hexadecimal representation of an eight-bit code of a character. For example, string ‘$0D$0A‘ is understood as representation of two codes, 00001101 and 00001010. The first code represents the Enter character at the ASCII Table, (CR, decimally 13) and the second code represents LineFeed character (LF, decimally 10).

Literals of a character string, so called strings, are used for example for text exchange among various PLCs or among a PLCs and another components of an automation system, or for programming texts that are displayed on control units or operator panels.

Table.3 Special characters in strings

Used as:	Meaning
$$	Dolar character
$'	Single quote mark character
$L or $l	Line feed (16#0A) character
$N or $n	New line character
$P or $p	New page character
$R or $r	Carriage return (16#0D) character
$T or $t	Tab (16#09) character

Table.4 Examples of character string literals

Example	Note
''	Empty string, 0 length
'temperature'	Not empty string, 11 character length
'Character $'A$''	String containing quotation mark (Character 'A')
' End of text $0D$0A'	String terminated by CR and LF characters
' Price is 12$$'	String containing a $ character
'$01$02$10'	String containing three characters: 1,2 and 16

Example 6 Character sting

```
PROGRAM ExampleStrings
  VAR
    message     : STRING := '';  // empty string
    value       : INT;
    valid       : BOOL;
  END_VAR

  IF valid THEN
    message := 'Temperature is ';
    message := CONCAT(IN1 := message, IN2 := INT_TO_STRING(value));
    message := message + ' [C]';
  ELSE
    message := 'Temperature is not available !';
  END_IF;
  message := message + '$0D$0A';
END_PROGRAM
```

3.1.2.3 Time literals

When performing control, we need two various data types that are related to the time in some way. Firstly, it is ***duration data***, which means a period of time elapsed or should elapse in connection with an event. Secondly, it is "absolute time" data consisting of *date* (according to the calender) and *time data within one day*, called ***time of day***. The time data can be used for synchronization of the start or end of an event being controlled in relation to the absolute time frame. Examples of time literals are shown in Table 3.6.

Duration. A time literal for duration begins with some of the keywords T#, t#, TIME#, time#. The time data itself is expressed in time units: hours, minutes, seconds and milliseconds. The abbreviations for individual parts of the time data are shown in Table 3.5. For their notification, lower-case as well as upper-case letters can be used.

Table.5 Abbreviations for time data

Abbreviation	Meaning
ms, MS	Miliseconds
s, S	Seconds
m, M	Minutes
h, H	Hours
d, D	Days

Day time and date. Date and time data representation within a day is the same as at ISO 8601. The prefix can be long or short. The keywords for a date are D# or DATE#. For time data within a day, keywords TOD# or TIME_OF_DAY# are used. For summary data on "absolute time", keywords DT# or DATE_AND_TIME# are used. The size of letters is again not important.

Table.6 Examples of various time literals

Description	Examples
Duration	T#24ms, t#6m1s, t#8.3s t#7h_24m_5s, TIME#416ms
Date	D#2003-06-21 DATE#2003-06-21
Day time	TOD#06:32:15.08 TIME_OF_DAY#11:38:52.35
Date and day time	DT#2003-06-21-11:38:52.35 DATE_AND_TIME#2003-06-21-11:38:52.35

Example 7 Time literals

```
VAR_GLOBAL
  myBirthday          : DATE := D#1982-06-30;
  firsthManOnTheMoon  : DT   := DT#1969-07-21-03:56:00;
END_VAR

PROGRAM ExampleDateTime
  VAR
    coffeeBreak  : TIME_OF_DAY := TOD#10:30:00.0;
    dailyTime    : TOD;
    timer        : TON;
    startOfBreak : BOOL;
    endOfBreak   : BOOL;
  END_VAR

  dailyTime    := TIME_TO_TOD( GetTime());
  startOfBreak := dailyTime > coffeeBreak AND dailyTime < TOD#12:00:00;
  timer(IN := startOfBreak, PT := TIME#15m, Q => endOfBreak);
END_PROGRAM
```

3.2 Date type

In accordance with the standard IEC 61 131-3, for different language are defined by so called **elementary**, predefined data types, **generic** data types for related data types groups. A mechanism is available, by which the user can create his/her own **user** data types (derived data types, type definition).

3.2.1 Elementary data types

Elementary data types are characterized by their data width (number of bits) or also by their value range. An overview of supported data types is stated in Table.3.7.

Table.7 Elementary data types

Keyword	English	Data type	Bits	Value range
BOOL	Boolean	Boolean number	1	0,1
SINT	Short integer	Short integer	8	–128 to 127
INT	Integer	Integer	16	–32 768 to +32 767
DINT	Double integer	Integer, double length	32	–2 147 483 648 to +2 147 483 647
USINT	Unsigned short integer	Unsigned integer, short	8	0 to 255
UINT	Unsigned integer	Integer, unsinged	16	0 to 65 535
UDINT	Unsigned double integer	Integer unsigned, double length	32	0 to +4 294 967 295
REAL	Real (simple precision)	Number in floating point (simple precision)	32	±2.9E-39 to ±3.4E+38 Acc. IEC 559
LREAL	Long real (double precision)	Number in floating point (double precision)	64	Acc.IEC 559
TIME	Duration	Duration	24d 20:31:23.647	
DATE	Date (only)	Date	From 1.1.1970 00:00:00	
TIME_OF_DAY or TOD	Time of day (only)	Time of day	24d 20:31:23.647	
DATE_AND_TIME or DT	Date and time of day	„Absolute time“	From 1.1.1970 00:00:00	
STRING	String	String	Max.255 characters	
BYTE	Byte(bit string of 8 bits)	8 bit sequence	8	No range declared
WORD	Word (bit string of 16bits)	16 bit sequence	16	No range declared
DWORD	Double word (bit string of 32 bits)	32 bit sequence	32	No range declared

Initialization of elementary data types

An important principal when programming according to the IEC 61 131-3 is that all program variables have the same initial (start) value. If the user does not state differently, the variable will be initialized with an implicit (preset, default) value, according to the used data type. Predefined initial values for elementary data types are usually nulls, by data it is D#1970-01-01. An overview of predefined initial values is stated in Table.3.8.

Table.8 Predefined initial values for elementary data

Data types	Initial Value
BOOL, SINT, INT, DINT	0
USINT, UINT, UDINT	0
BYTE, WORD, DWORD	0
REAL, LREAL	0.0
TIME	T#0s
DATE	D#1970-01-01
TIME_OF_DAY	TOD#00:00:00
DATE_AND_TIME	DT#1970-01-01-00:00:00
STRING	’ ’ (empty string)

3.2.2 Generic data types

Generic data types always express the whole group (genus) of data types. They start with prefix **ANY**. For example, by notation of **ANY_BIT**, all data types shown further are understood: **DWORD, WORD, BYTE, BOOL**. An overview of generic data types is shown in Table 3.9. The names of generic data types begining with **ANY_** are not according to the IEC keywords. The are intended only for marking type groups with the same features.

Table.9 Overview of generic data types

<table>
<tr><th colspan="5">ANY</th></tr>
<tr><th>ANY_BIT</th><th colspan="3">ANY_NUM</th><th>ANY_DATE</th><th rowspan="3">TIME
STRING</th></tr>
<tr><td rowspan="2">BOOL
BYTE
WORD
DWORD</td><th colspan="2">ANY_INT</th><th>ANY_REAL</th><td rowspan="2">DATE
DATE_AND_TIME
TIME_OF_DAY</td></tr>
<tr><td>INT
SINT
DINT</td><td>UINT
USINT
UDINT</td><td>REAL
LREAL</td></tr>
</table>

3.2.3 Derived data types

Derived types, i.e. types specified by manufacturer or by user, can be declared by means of textual structure **TYPE…END_TYPE**. The names of new types, their data types, possible with their initial values, are given within this textual structure. These derived data types can be further used together with the elementary data types in declarations of variables. The definition of the de-

rived data type is global, i.e. can be used in any PLC program part. The derived data type takes adopts the type features from which it was derived from.

3.2.3.1 Simple derived data types

Simple derived data types originate directly from elementary data types. The most common reason for creating a new data type is its different initialization value, which can be assigned directly in the type declaration using a assignment operator ":=". If its initialization value is not declared in the declaration of the new type then it accepts the initialization value from the type it was derived from.

The enumerated data type also belongs to simple derived data types. It is usually used for naming features or versions instead of using a number code to each version which makes the program easier to read. The initialization value of the enumerated data type is always the value of the first element stated in the enumeration.

Example 8 Example of simple derived data types

```
TYPE
  TMyINT       : INT;                 // simple derived data types
  TRoomTemp    : REAL := 20.0;        // new data type with initialization
  THomeTemp    : TRoomTemp;
  TPumpMode    : ( off, run, fault); // new data type declared via
                                        enumerated values
END_TYPE

PROGRAM SingleDerivedType
  VAR
    pump1Mode    : TPumpMode;
    display      : STRING;
    temperature  : THomeTemp;
  END_VAR

  CASE pump1Mode OF
    off   : display := 'Pump no.1 is off';
    run   : display := 'Pump no.1 is running';
    fault : display := 'Pump no.1 has a problem';
  END_CASE;
END_PROGRAM
```

Single element variables having user type declared can be used anywhere, where a variable of "**parent**" type can be used. For example, variable "temperature" from example 3.6 can be used anywhere, where variables of type **REAL** can be used. This rule can be applied recursively.

Array or **structure** can also be a new data type.

3.2.3.2 Derived array data type

One-dimensional arrays

An array is an aligned row of elements of the same data type. Every element of the array has an index assigned to it, through which it is possible to access the element, i.e. the value of the index determines with which element the array will work with. The index may only be within the value range defined by the array. If the index value exceeds the declared array size, then a run-time error (error executed during system running) will be executed. A one-dimensional array is an array which has only one index, as seen in Figure 3.1.

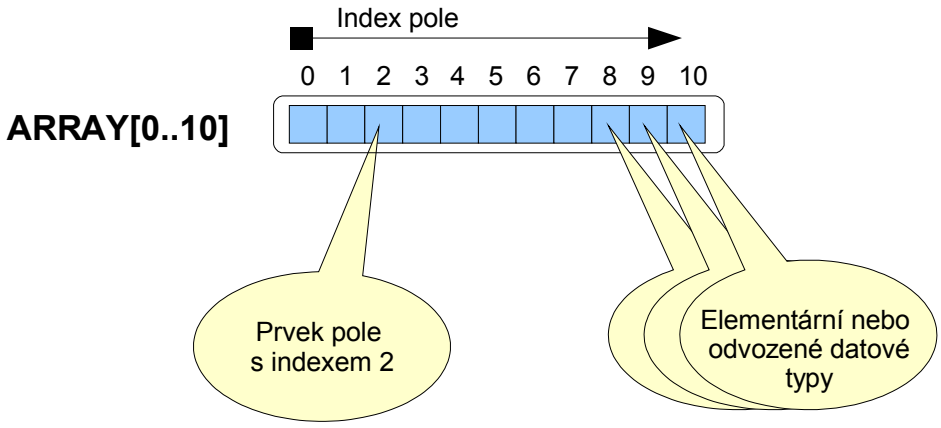


Figure 4 A one-dimensional array

An array element may be an elementary of a derived data type. POU array instances are not yet supported. Example 3.7 shows a declaration of a derived array data type. Declaration is done via the keyword **ARRAY** followed by array dimension in square brackets. The array size determines the range of acceptable indexes. The array size is then followed by the keyword **OF** with a data type specification for array elements. The index of the first array element must be a positive number or zero. Negative indexes are not acceptable. The maximum size of an array is limited by the memory range of variables in the control system.

Array type declaration may also contain the initialization of individual elements (see types **TbyteArray** and **TRealArray**). Initialization values are stated in the array type declaration behind the assignment operator ":=" in square brackets. If less initialization values are defined than needed for the array dimension, then elements without defined initialization values have their initial values preset according to the value of the used data type. For initializing a large number of array elements with a same value a so called repeater can be used. In such a case the number of repeating of the initialization value is stated in round brackets on the place of the initialization value. For example `25( -1)` will initialize 25 array elements with a value of -1.

Example 9 Derived data type one-dimensional array

```
TYPE
  TVector     : ARRAY[0..10]  OF INT;
  TByteArray  : ARRAY[1..10]  OF BYTE := [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
  TRealArray  : ARRAY[5..9]   OF REAL := [ 11.2, 12.5, 13.1];
  TBigArray   : ARRAY[1..999] OF SINT := [ 499( -1), 0, 499( 1)];
END_TYPE

PROGRAM Example1DimArray
  VAR
    index      : INT;
    samples    : TVector;
    buffer     : TByteArray;
    intervals  : TRealArray;
    result     : BOOL;
  END_VAR

  FOR index := 0 TO 10 DO
    samples[index] := 0;            // clear all samples
  END_FOR;
  result := intervals[5] = 11.2;    // TRUE
  result := intervals[8] = 0.0;     // TRUE
END_PROGRAM
```

Multi-dimensional arrays

Multi-dimensional arrays are arrays where we need more than one index to access one element. The array has then one or more dimensions which can be different for each index. Two-dimensional arrays can be described as a matrix of elements as seen on figure 3.2. Elements of multi-dimensional arrays may be of the elementary of derived data type, similar to one-dimensional arrays.

The Mosaic compiler supports a maximum of four-dimensional arrays.

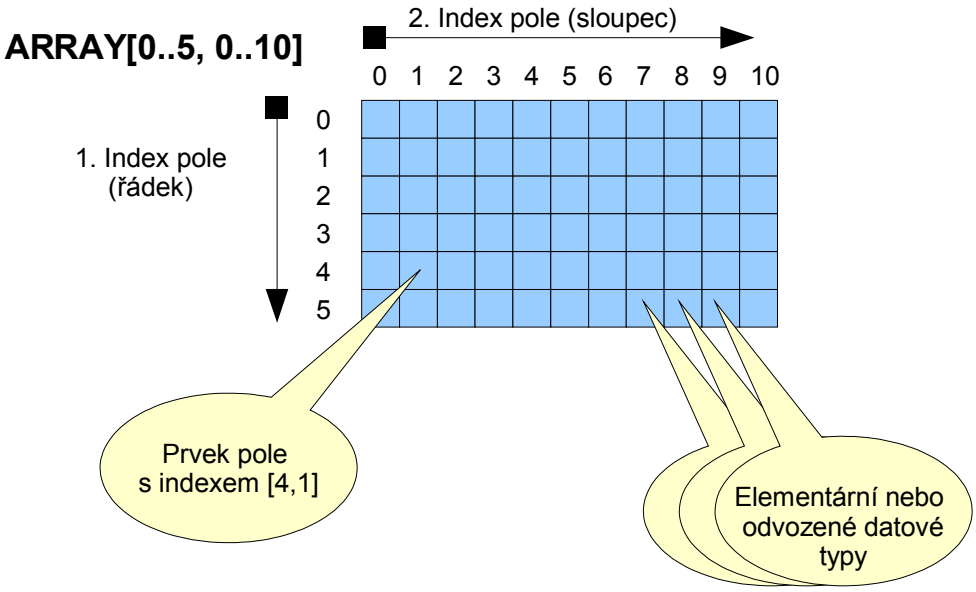


Figure 5 A two-dimensional array

Initialization of multi-dimensional arrays is done in the same manner as by one-dimensional arrays; first all elements for the first dimension are initialized (i.e. for example array[0,0], array[0,1], array[0,2] to array[0,n]) and then the procedure is repeated for the other values of the first index. The last elements to be initialized are for array[m,0], array[m,1], array[m,2] and finally array[m,n]. When initializing multi-dimensional arrays, it is possible to use a repeater for initializing more elements at once as is shown in example 3.8 by the type **`TThreeDimArray1`**. The same declaration is stated in the notes without the use of repeaters.

Example 10 Derived data type multi-dimensional array

```
TYPE
  TTwoDimArray   : ARRAY [1..2,1..4] OF SINT := [ 11, 12, 13, 14,
                                                  21, 22, 23, 24 ];

  TThreeDimArray : ARRAY [1..2, 1..3, 1..4] OF BYTE :=
                                                [ 111, 112, 113, 114,
                                                  121, 122, 123, 124,
                                                  131, 132, 133, 134,
                                                  211, 212, 213, 214,
                                                  221, 222, 223, 224,
                                                  231, 232, 233, 234 ];

  TThreeDimArray1 : ARRAY [1..2, 1..3, 1..4] OF BYTE :=
                                                [ 4(11), 4(12), 4(13),
                                                  4(21), 4(22), 4(23) ];
(*
  TThreeDimArray1 : ARRAY [1..2, 1..3, 1..4] OF BYTE :=
                                                [ 11, 11, 11, 11,
                                                  12, 12, 12, 12,
                                                  13, 13, 13, 13,
                                                  21, 21, 21, 21,
                                                  22, 22, 22, 22,
                                                  23, 23, 23, 23 ];
*)
END_TYPE
```

```
PROGRAM ExampleMultiDimArray
  VAR
    twoDimArray   : TTwoDimArray;
    threeDimArray : TThreeDimArray;
    element       : BYTE;
    result        : BOOL;
  END_VAR

   result  := twoDimArray[1, 4] = 14;     // TRUE
   element := threeDimArray[ 2, 1, 3];    // element = 213
END_PROGRAM
```

Similar to the derived data type array, it is possible to directly declare an array variable type as shown in chapter 3.

3.2.3.3 Derived data type Structure

Structures are data types which contain, similar to arrays, more elements (items). However on contrary to arrays, all elements in a structure do not have be of the same data type. A structure can be derived from elementary as well as from derived data types. A structure can be created hierarchy style which means that an already defined structure can be an element of a structure The situation is described in figure 3.3.

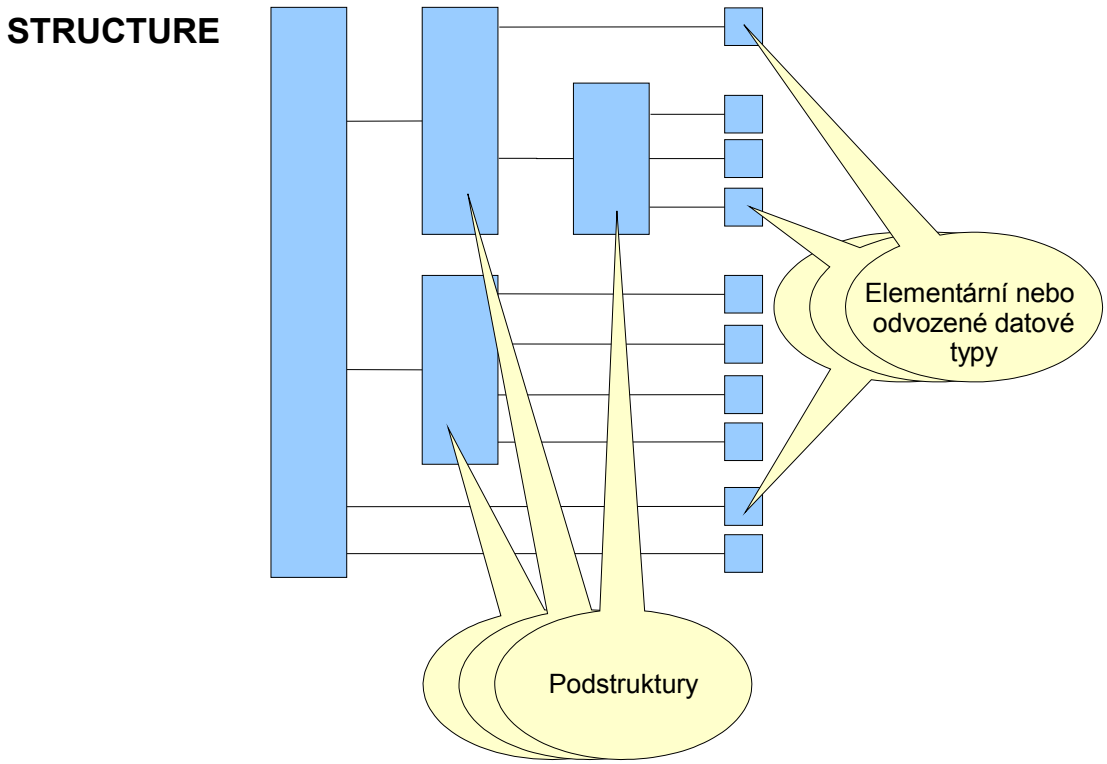


Figure 6 Structure

The definition of a new structure data type is done using keywords **STRUCT** and **END_STRUCT** within the construction **TYPE** … **END_TYPE**. Data types of individual elements of a structure and their names are stated inside **STRUCT** … **END_STRUCT**. It is possible to initialize structures by stating element values behind the sign ":=" in the same way as by the previous derived data types.

If we create a structure type variable, then access to individual structure elements will be "**variableName.elementName**" as seen on example 3.9.

Example 11 Derived data type Structure

```
TYPE
  TProduct :
    STRUCT
      name         : STRING := 'Engine M11';
      code         : UINT;
      serie        : DINT;
      serialNum    : UDINT;
      expedition   : DATE;
    END_STRUCT;
END_TYPE

PROGRAM ExampleStruct
  VAR
    product     : TProduct;
    product1    : TProduct;
  END_VAR

  product.code        := 700;
  product.serie       := 0852;
  product.serialNum   := 12345;
  product.expedition  := DATE#2002-02-13;
END_PROGRAM
```

The initialization of the structure type variables is done using structure element names when declaring variables. The difference between initializing a structure data type and initializing a structure type variable is shown in examples 3.9 and 3.10. The functional difference is obvious. While in example 3.9, every **TProduct** type variable will have a **NAME** element automatically initialized to the value "**Engine M11**", then in example 3.10 the implicit **NAME** element initialization is an empty string that will be exchanged by the "**Engine M11**" string only if the variable is **PRODUCT**.

Example 12 Initialization of derived type Structure

```
TYPE
  TProduct :
    STRUCT
      name         : STRING;
      code         : UINT;
      serie        : DINT;
      serialNum    : UDINT;
      expedition   : DATE;
    END_STRUCT;
END_TYPE
```

```
PROGRAM ExampleStruct
  VAR
    product     : Tproduct := ( name := 'Engine M11');
    product1    : TProduct;
  END_VAR

  product.code       := 700;
  product.serie      := 0852;
  product.serialNum  := 12345;
  product.expedition := DATE#2002-02-13;
END_PROGRAM
```

3.2.3.4 Combining structures and arrays in derived data types

Arrays and structures can be randomly combined in definitions of derived data types. An array may be an element of a structure and a structure may be an element of an array as shown example 3.11.

Example 13 A structure as an element in an array

```
VAR_GLOBAL CONSTANT
  NUM_SENSORS   : INT := 12;
END_VAR

TYPE
  TLimit :
    STRUCT
      low          : REAL := 12.5;
      high         : REAL := 120.0;
    END_STRUCT;

  TSensor :
    STRUCT
      status       : BOOL;
      pressure     : REAL;
      calibration  : DATE;
      limits       : TLimit;
    END_STRUCT;

   TSenzorsArray  : ARRAY[1..NUM_SENSORS] OF TSensor;
END_TYPE

PROGRAM ExampleArrayOfStruct
  VAR
    sensors : TSenzorsArray;
    i       : INT;
  END_VAR

  FOR i := 1 TO NUM_SENSORS DO
    IF (sensors[i].pressure >= sensors[i].limits.low)  AND
       (sensors[i].pressure <= sensors[i].limits.high)
    THEN
       sensors[i].status := TRUE;
    ELSE
       sensors[i].status := FALSE;
    END_IF;
  END_FOR;
END_PROGRAM
```

3.2.4 Data type Pointer

The pointer data type is an addition to the IEC 61 131 standard. In other words, the pointer is not defined by the mentioned standard and programs which will be using this data type cannot be used for PLCs programmed in a different environment than Mosaic.

The reason why this data type is missing among the standardized data types is mainly programming safety. An incorrectly used pointer can lead to a program crash, which is unacceptable when controlling technologies. It is not possible to discover the error during program assembly or during program operation. Experience from the C programming language, where pointers are often used, show that a great part of incorrect program operations is caused by incorrect pointer handling. On the other side, only a very few programs exist that are programmed in C and without pointers. What does this mean? Pointers can be very good servants but very bad masters. The responsibility for the correctness of a program using pointers lies only on the programmer, because methods helping him discover errors (compiler, type control, run-time checks, etc.) are useless regarding pointers. An advantage of pointers is their higher effectivity of programming. In many cases pointers enable shorter and quicker programs, mainly if structures, arrays and their combinations are used. And the last reason for using pointers is the fact that some problems can be solved only by using pointers.

A pointer is really a pointer to a variable that can be of the elementary or derived type. The declaration of pointers is done using the keyword **PTR_TO** followed by the data type name to which the pointer is pointing to. The pointer data type can be used everywhere, where an elementary data type can be used. POUs do not support pointers.

The pointer variable type contains an address to another variable. A pointer can be worked with in two ways. First it is possible to change its value (increase, decrease, etc.) and so change the variable to which the pointer will point to. Second it is possible to work with the variable value to which the pointer is pointing to. The first mentioned operation is called pointer arithmetics, the second is called pointer dereference.

Pointer arithmetics

The first operation that a program has to do with a pointer is to fill the variable's address to which the pointer will be pointing to. The implicit initialization of the pointer data type is -1 which means that the pointer does not point to any variable. This is also the only case that can be discovered by the run-time check and identified as an error.

The initialization of a pointer, i.e. its filling with the address of the variable it will be showing to, is done using the system function **ADR()**. The parameter of this function is the name of the variable, we want the pointer to be filled with. For example `myPtr := ADR( myVar)` fills the `myPtr` pointer with the `myVar` variable's address; i.e. the `myPtr` pointer will point to the variable `myVar`.

The pointer data type can be used for arithmetic operations with the purpose of changing the address of a variable. The `PTR_TO` type can be combined with data types `ANY_INT`. If the `myVar` variable is placed in the memory on the address %MB100 and the `yourVar` variable will be on the address %MB101, then the expression `myPtr := myPtr + 1` will increase the value of the pointer by 1, so the pointer will be pointing to the `yourVar` variable(instead of the original `myVar` variable). Off course only under the condition that both variables are of a data type which uses a single byte in the memory. In case of the `PTR_TO` type the arithmetics function only byte-wise, which means that after adding the value 15, the pointer will be pointing to a variable 15 bytes further up the memory.

Pointer dereference

Pointer dereference is an operation that enables to work with a variable to which a pointer is pointing to. The sing ^ is used for dereference. The expression **value := myPtr^** fills the **value** variable with the value of the **myVar** variable (off course under the condition that **myPtr** points to **myVar** and the **value** variable is of the same data type as the **myVar** variable).

Example 14 Pointers

```
VAR_GLOBAL
  arrayINT    : ARRAY[0..10] OF INT;
END_VAR

PROGRAM ExamplePtr
  VAR
    intPTR    : PTR_TO INT;
    varINT    : INT;
  END_VAR

  intPTR  := ADR( arrayINT[0]);          // init ptr
  intPTR^ := 11;                         // arrayINT[0] := 11;
  intPTR  := intPTR + sizeof( INT);      // ptr to next item
  intPTR^ := 22;                         // arrayINT[1] := 22;
  intPTR  := intPTR + sizeof( INT);      // ptr to next item
  varINT  := intPTR^;                    // varINT := arrayINT[2];
END_PROGRAM
```

Example 3.12 uses the function **sizeof()** for increasing the address to which the **intPTR** is pointing to. This function returns the number of bytes of the given data type or variable.

Another example shows how easy it is to make a mistake when working with pointers. The program is the same as in example 3.12 only with a different **intPTR** pointer initialization. While the initialization in the first case is executed in every cycle by the statement **intPTR := ADR(arrayINT[0])**, the pointer initialization in the second example is executed already in the declaration of the variable **intPTR : PTR_TO INT := ADR(arrayINT[0])**. That causes that the first program cycle, after system restart, will be correct but the pointer in the second cycle will start with an element address of **arrayINT[2]** instead of **arrayINT[0]**. During a cyclic execution of the program this means that the program in example 3.13 will rewrite the whole variable memory in a very short time with **INT#11** and **INT#22** values, which is something we surely do not want. Please remember that it is necessary to take extra care when working with pointers.

Example 15 Incorrect initialization of pointer

```
VAR_GLOBAL
  arrayINT    : ARRAY[0..10] OF INT;
END_VAR

PROGRAM ExamplePtrErr
  VAR
    intPTR    : PTR_TO INT := ADR( arrayINT[0]);
    varINT    : INT;
  END_VAR
```

```
                                        // for 1st cycle only !!!
  intPTR^ := 11;                        // arrayINT[0] := 11;
  intPTR  := intPTR + sizeof( INT);     // ptr to next item
  intPTR^ := 22;                        // arrayINT[1] := 22;
  intPTR  := intPTR + sizeof( INT);     // ptr to next item
  varINT  := intPTR^;                   // varINT := arrayINT[2];
END_PROGRAM
```

3.3 Variables

According to IEC 61 131-3 ***variables*** are strictly speaking are means for identification of data objects, the content of which can change, i.e. data associated with inputs, output or PLC memory. A variable can be declared by one of the elementary data types or by some of the user (derived) data types.

In this way programming according to IEC 61 131-3 cam closer to standardly used solutions. Instead of hardware addresses or symbols, the variables are defined in such a way as they are used in higher programming languages. Variables are identifiers (names) assigned by the programmer, which are used, strictly speaking, to reserve a location in memory and they contain program data values.

3.3.1 Variables declaration

Each programmable controller program organization unit (POU) type declaration (i.e., each declaration of a program, function, or function block) shall contain at its beginning at least one declaration part which specifies the types of the variables used in the POU. This declaration part shall have the textual form of one of the keywords VAR, VAR_TEMP, VAR INPUT or VAR_OUTPUT, followed in the case of VAR by the qualifier CONSTANT. Behind the keywords follows one or more declarations of variables separated by semicolons and terminated by the keyword END_VAR. The declaration of their initial values can be part of variable declaration.

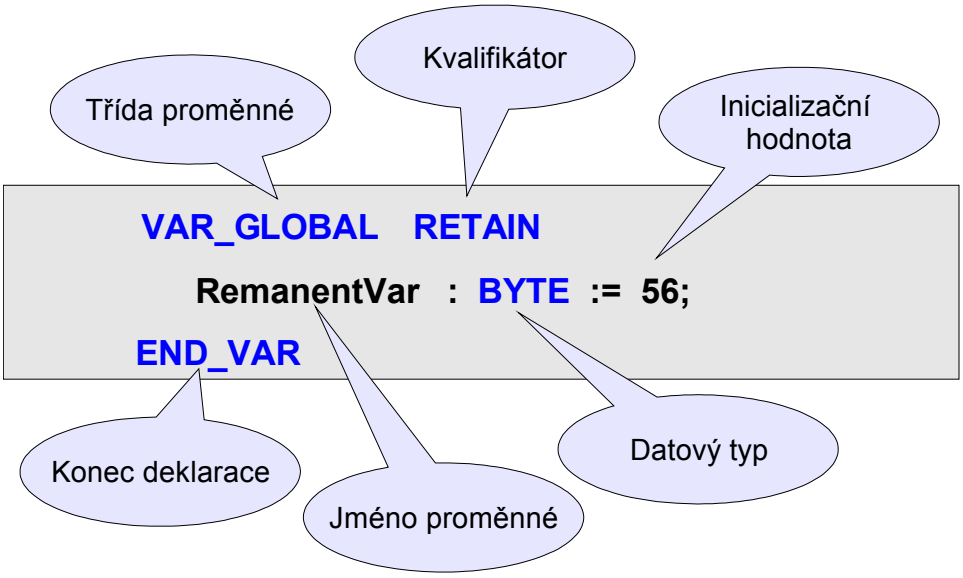


Figure 7 Variables declaration according to IEC

The *scope* (range of validity) of the declarations contained in the declaration part is ***local*** to the program organization unit in which the declaration part is contained. That is, the declared variables shall not be accessible to other POUs except by explicit argument passing via variables which have been declared as ***input variables*** (VAR_UNIT) or ***output variables*** (VAR_OUTPUT) of this units. The one exception to this rule is the case of variables which have been declared to be ***global***. Such variables are defined outside the declarations of all POUs and begin with the keyword VAR_GLOBAL. Behind the keyword VAR_GLOBAL can be optionally put the RETAIN or CONSTANT qualifier.

3.3.1.1 Variable classes

Variable classes determine the use and scope of variables. It is possible to divide variables accordingly:

- **global variables**
  - **`VAR_GLOBAL`** - not backed up variables
  - **`VAR_GLOBAL RETAIN`** - backed up variables
  - **`VAR_GLOBAL CONSTANT`** - constants
  - **`VAR_EXTERNAL`** - external variables
- **local variables**
  - **`VAR`** - local variables
  - **`VAR_TEMP`** - temporary variables
- **variables for handing over parameters**
  - **`VAR_INPUT`** - input variables
  - **`VAR_OUTPUT`** - output variables
  - **`VAR_IN_OUT`** - input-output

Table 10 Variable type

Variable type	Meaning	Designation
VAR_INPUT	**input**	**For passing input parameters into POU** **These variables are visible from other POUs and they are set from them, too.**
VAR_OUTPUT	**output**	**For passing of output variables from POU** **These variables are visible from other POUs, where only their reading can be performed.** **The change of the value of these variables can be performed within the POU only, in which the variables were declared.**
VAR_IN_OUT	**input / output**	**For indirect access to variables outside the POU** **Variables can be read and their value can be changed inside as well as outside the POU.**
VAR_EXTERNAL	**global**	**Variables defined in PLC mnemocode**
VAR_GLOBAL	**global**	**Variables available from all POUs.**
VAR	**local**	**Auxiliary variables used within POU** **They are not "visible" from another POUs, which means they can be read or their value can be changed within the POU only, in which they are declared.** **These variables can store a value also between individual calls of the POU in question.**
VAR_TEMP	**local**	**Auxiliary variables used within POU** **They are not "visible" from another POUs.** **These variables are created during the input into the POU and disappear after the POU is ended - thus they cannot store any value between two calls of the POU.**

Table.11 Use of variables for particular POUs

Variable type	PROGRAM	FUNCTION_BLOCK	FUNCTION	Outside the POU
VAR_INPUT	yes	yes	yes	no
VAR_OUTPUT	yes	yes	no	no
VAR_IN_OUT	yes	yes	yes	no
VAR_EXTERNAL	yes	yes	yes	no
VAR_GLOBAL	no	no	no	yes
VAR	yes	yes	yes	no
VAR_TEMP	yes	yes	yes	no

3.3.1.2 Qualifiers in variables declaration

Qualifiers allow defining additional features of declared variables. The keyword for a qualifier begins with VAR. In variables declarations, the following qualifiers can be used:

- ***RETAIN*** – retained variables (variables retaining their value also after the PLC power supply is OFF);
- ***CONSTANT*** – constant value (the value of an variable cannot be changed)
- ***R_EDGE*** – variable rising edge
- ***F_EDGE*** – variable falling edge

Table.12 Usage of qualifiers in variables declaration

Variable type	Meaning	RETAIN	CONSTANT	R_EDGE F_EDGE
VAR	local	no	yes	no
VAR_INPUT	input	no	no	yes
VAR_OUTPUT	output	no	no	no
VAR_IN_OUT	input / output	no	no	no
VAR_EXTERNAL	global	no	no	no
VAR_GLOBAL	global	yes	yes	no
VAR_TEMP	local	no	no	no

3.3.2 Global variables

From the point of view of availability, variables can be divided into *global and local*.

Global variables are such variables that are available to all POUs. Their definition begins with the keyword **VAR_GLOBAL** and it is not mentioned inside any POU which is shown by example 3.14. Global variables can be placed to a specific address within the PLC memory using the keyword **AT** in the variable declaration. If the **AT** keyword is missing, the compiler assigns the needed space automatically.

If the qualifier **CONSTANT** is stated in the declaration, then the variable definition has a fixed value by the declaration and it cannot be changed by the program. So they are not variables to all intents and purposes but rather constants. And if they are also of an elementary data type, the compiler will not assign any location in the memory to them, it will only use the corresponding constant in the expressions.

Variables from the **VAR_EXTERNAL** class can be global and local. If the variable declaration of this class is stated inside the POU, then it is a local variable, if not it is a global variable.

Example 16 Declaration of global variables

program in mnemocode:

```
#reg word mask                      ; variable declaration in mnemocode
P 0
        ld      $1111
        wr      mask
E 0
```

Program in ST language:

```
VAR_EXTERNAL
  mask            : WORD;           // link to variable in mnemocode
END_VAR

VAR_GLOBAL RETAIN
  maxTemp         : REAL;           // backed up variable
END_VAR

VAR_GLOBAL CONSTANT
  PI        : REAL := 3.14159;      // constant
END_VAR

VAR_GLOBAL
  globalFlag      : BOOL;
  suma            : DINT := 0;
  temp    AT %XF10 : REAL;          // temperature
  minute AT %S7   : USINT;
END_VAR

PROGRAM ExampleGlobal

  globalFlag := mask = 16#1111;                 // true
  maxTemp := MAX(IN1 := temp, IN2 := maxTemp);
END_PROGRAM
```

3.3.3 Local variables

Local variables are declared within the POU and their validity and visibility is limited to the POU in which they are declared in. It is not possible to use them from the other POUs. The declaration of local variables begins with the keywords **VAR** or **VAR_TEMP**.

Variables declared in the **VAR** class are so called static variables. These variables are assigned a fixed place in the variables memory by the compiler; this place does not move during program execution. That means that the more variables in the **VAR** class are defined, the more memory will be occupied. Another important feature of the **VAR** class variables is that their value is kept in memory during two POU callings in which they are declared in.

Variables declared in the **VAR_TEMP** class are variables which are dynamically created, when the POU starts calculating with the affected declaration. When the POU finished the calculation, the dynamically assigned memory is freed and the **VAR_TEMP** class variables are deleted. This means that declarations of **VAR_TEMP** class variables do not affect memory consumption. Such variables cannot keep values between tow POU calls, because after the POU finishes, they stop to exist.

The difference between the **VAR** and **VAR_TEMP** class variables is also in their initialization. The **VAR** class variables are initialized only when a system is restarted while the **VAR_TEMP** class variables are initialized every time they are assigned a memory location (i.e. after each calculation start of the POU). The following features can be seen in the following example.

Example 17 Declaration of local variables

```
PROGRAM ExampleLocal
  VAR
    staticCounter    : UINT;
    staticVector     : ARRAY[1..100] OF BYTE;
  END_VAR
  VAR_TEMP
    tempCounter      : UINT;
    tempVector       : ARRAY[1..100] OF BYTE;
  END_VAR

  staticCounter := staticCounter + 1;
  tempCounter   := tempCounter   + 1;
END_PROGRAM
```

The value of the local variable **staticCounter** will increase itself by repeated callings of the **ExampleLocal** program, because every calling starts a calculation with the value **staticCounter** from the last calling. In contrast to this the value of the **tempCounter** variable will be at the end of the **ExampleLocal** program always 1, independently to the number of callings of the program, because this variable is created and initialized with the value 0 on every calling made by the **ExampleLocal** program.

On the example 3.15, it is possible to show the differences in memory usage. The **staticVector** variables occupies 100 bytes in the variables memory while the **tempVector** variable does not affect the usage of the memory.

3.3.4 Input and output variables

Input and output variables are used for handing over parameters between POUs. Using such variables, we can define input and output interfaces of the POUs.

For exchanging parameters towards the POUs the **VAR_INPUT** class variables are used; they are the **input** variables. For exchanging parameters from the POUs the **VAR_OUTPUT** class variables are used; they are the **output** variables. If we imagine an e.g. **function block** as an integrated circuit, then the **VAR_INPUT** variables will represent the input signals of the circuit and the **VAR_OUTPUT** variables will represent the output signals of the circuit.

The definition of **BOOL** type variables in the **VAR_INPUT** class can be widened by using **R_EDGE** and **F_EDGE** qualifiers which enable detecting the rising or falling edge of a variable. Variables defined by a **R_EDGE** qualifier have **true** type values only if the value of a variable changes from **false** to **true**. Such a variable is also the **in** variable in example 3.16. The **FB_EdgeCounter** function block in this example will be counting rising edges (changes from a false value to true value) of the **in** input variable.

Example 18 Detection of rising edge of a input variable

```
FUNCTION_BLOCK FB_EdgeCounter
  VAR_INPUT
    in          : BOOL R_EDGE;
  END_VAR
  VAR_OUTPUT
    count       : UDINT;
  END_VAR

  IF in THEN count := count + 1; END_IF;
END_FUNCTION_BLOCK


PROGRAM ExampleInputEdge
  VAR_EXTERNAL
    AT  %X0.0   : BOOL;
  END_VAR
  VAR
    edgeCounter : FB_EdgeCounter;
    howMany     : UDINT;
  END_VAR

  edgeCounter(in := %X0.0, count => howMany);
END_PROGRAM
```

Parameters sent via input or output variables are handed over via values. In other words, this means that when the POU is calling, it is necessary to hand over the values of the input variables. After returning from the POU, it is necessary to hand over the values of the output variables.

VAR_IN_OUT class variables can be used simultaneously as input and output variables. Parameters handed over to the POU through **VAR_IN_OUT** class variables are not handed over via val-

ues, but via references. That means that when the POU is calling, the variable address hands over the place of the value, which enables to use the value as needed as an input or output variable.

The difference between handing over parameters via value and reference can be seen in example 3.17.

Example 19 Difference between using VAR_INPUT and VAR_IN_OUT variables

```
TYPE
  TMyUsintArray : ARRAY[1..100] OF USINT;
END_TYPE

FUNCTION Suma1  : USINT
  VAR_INPUT
    vector      : TMyUsintArray;
    length      : INT;
  END_VAR
  VAR
    i           : INT;
    tmp         : USINT := 0;
  END_VAR

  FOR i := 1 TO length DO tmp := tmp + vector[i]; END_FOR;
  Suma1 := tmp;
END_FUNCTION

FUNCTION Suma2  : USINT
  VAR_IN_OUT
    vector      : TMyUsintArray;
  END_VAR
  VAR_INPUT
    length      : INT;
  END_VAR
  VAR
    i           : INT;
    tmp         : USINT := 0;
  END_VAR

  FOR i := 1 TO length DO tmp := tmp + vector[i]; END_FOR;
  Suma2 := tmp;
END_FUNCTION

PROGRAM ExampleVarInOut
  VAR
    buffer      : TMyUsintArray := [1,2,3,4,5,6,7,8,9,10];
    result1,
    result2     : USINT;
  END_VAR

  result1 := Suma1( buffer, 10);        // 55
  result2 := Suma2( buffer, 10);        // 55

END_PROGRAM
```

The task of this example was to create a function that would calculate the sum of the input number of **USINT** array type elements.

The **Suma1** function uses **vector** as an input variable of the **VAR_INPUT** class, which means, that when calling this function, all values of all elements of the **buffer** array must be sent

to the **vector** input variable. In this case it means 100 bytes of data. The calculation is done above the **vector** variable.

The **Suma2** function has the **vector** variable defined in the **VAR_IN_OUT** class and so, during calling this function, it sends the **buffer** variable's address instead of the values of all elements. This means only 4 bytes instead of 100 bytes in case of the first case. The **vector** input variable contains the address of the **buffer** variable and the calculation is done above the **buffer** variable which is indirectly addressed to via the **vector** variable.

3.3.5 Simple-element and multi-element variables

From the point of view of data types, the variables can be divided into *simple-element* and *multi-element variables*. Simple element variables are of the basic type. Multi-element variables are of the types *array* and ***structure***. The IEC 61 131-3 standard sees these variables as multi-element variables.

3.3.5.1 Simple-element variables

A simple-element variable is defined as a variable representing a single data element of one of the elementary data types or user data types (value enumeration list, subrange or a type recursively derived that you can come recursively back to the value enumeration list or subranges or elementary data types). Examples of simple-element variables are stated in example 3.18.

Example 20 Simple-element variables

```
TYPE
  TColor        : (white, red, gree, black);
  TMyInt        : INT := 100;
END_TYPE

VAR_GLOBAL
  basicColor    : TColor := red;
  lunchTime     : TIME   := TIME#12:00:00;
END_VAR

PROGRAM ExapleSimpleVar
  VAR
    tmpBool     : BOOL;
    count1      : INT;
    count2      : TMyInt;
    currentTime : TIME;
  END_VAR
  VAR_TEMP
    count3      : REAL := 100.0;
  END_VAR

END_PROGRAM
```

3.3.5.2 Array

An **array** is a collection of data elements of the same data type referenced by one or more *subscripts* enclosed in brackets and separated by commas. A subscript shall be an expression of the types contained in generic type ANY_INT. The maximum number of subscripts (array dimension) is 4 and the maximum range of subscripts has to correspond to type INT.

Variables of the array type can be defined in two different ways. First, the derived array data type can be defined and then the variable of this type is created. This is the case of **rxMessage** variable in example 3.19. Or second, the array can be defined directly in the variable declaration, see **sintArray** variable in the same example. Rules stated in chapter 3.2.3.2 apply to both declaration types. Also the manner of entering the initialization values is the same.

Example 21 Variables array

```
TYPE
  TMessage    : ARRAY[0..99] OF BYTE;
END_TYPE

VAR_GLOBAL
  delay       : ARRAY [1..5] OF TIME := [ TIME#1h,
                                          T#10ms,
                                          time#3h_20m_15s,
                                          t#15h5m10ms,
                                          T#3d];
END_VAR

PROGRAM ExampleArrayVar
  VAR
    rxMessage   : TMessage;
    txMessage   : TMessage;
    sintArray   : ARRAY [1..2,1..4] OF SINT := [ 11, 12, 13, 14,
                                                 21, 22, 23, 24 ];
  END_VAR
  VAR_TEMP
    pause       : TIME;
    element     : SINT;
  END_VAR

   pause   := delay[3];              // 3h 20m 15s
   element := sintArray[2, 3];       // 23
END_PROGRAM
```

3.3.5.3 Structures

A **structured variable** is a variable which is declared to be of a type which has previously been specified to be a data structure, i.e., a data type consisting of a collection of named elements. The declaration of the derived type Structure is described in chapter 3.2.3.3.

The direct declaration of a structure is not supported in the variables declaration.

In example 3.20 the variable **`pressure`** is defined, which is of the **`Tmeasure`** structure type. Another structure, the **`Tlimit`**, is used in the definition of this type. The example also shows the initialization of all elements of the **`pressure`** variable, including immersed structures. It can also be seen in the example, how can the program access each element in the structure variable (e.g. **`presure.lim.low`**). The **`AT %XF10`** construction is explained in the following chapter.

Example 22 Structured variable

```
TYPE
  TLimit :
    STRUCT
      low, high : REAL;
    END_STRUCT;

  TMeasure :
    STRUCT
      lim       : TLimit;
      value     : REAL;
      failure   : BOOL;
    END_STRUCT;
END_TYPE

VAR_GLOBAL
  AT      %XF10 : REAL;
  presure       : TMeasure := ( lim     := ( low := 10, high := 100.0),
                                value   := 0,
                                failure := false);
END_VAR

PROGRAM ExampleStructVar

  presure.value := %XF10;                 // input sensor
  IF presure.value < presure.lim.low  OR
     presure.value > presure.lim.high
  THEN
    presure.failure := TRUE;
  ELSE
    presure.failure := FALSE;
  END_IF;
END_PROGRAM
```

3.3.6 Location of variables in the PLC memory

The compiler allocates variables in the PLC memory automatically. If it is necessary to locate a variable on a specific address, it is possible to specify such address in the declaration of variables using the keyword **`AT`** followed by the variable address.

Expressing the variable address

A special "**%**" sign is used for expressing the variable address; the location prefix and size prefix. These signs are followed by one or more characters of the UINT type separated by full stops.

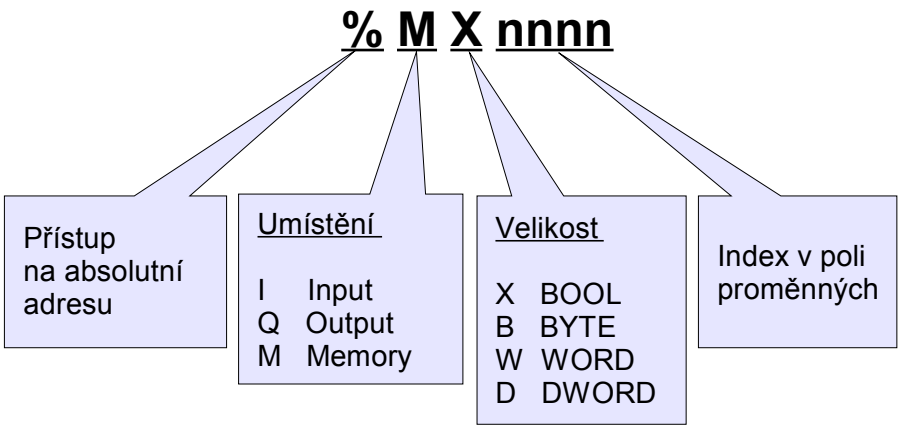


Figure 8 Direct address to PLC memory according to IEC

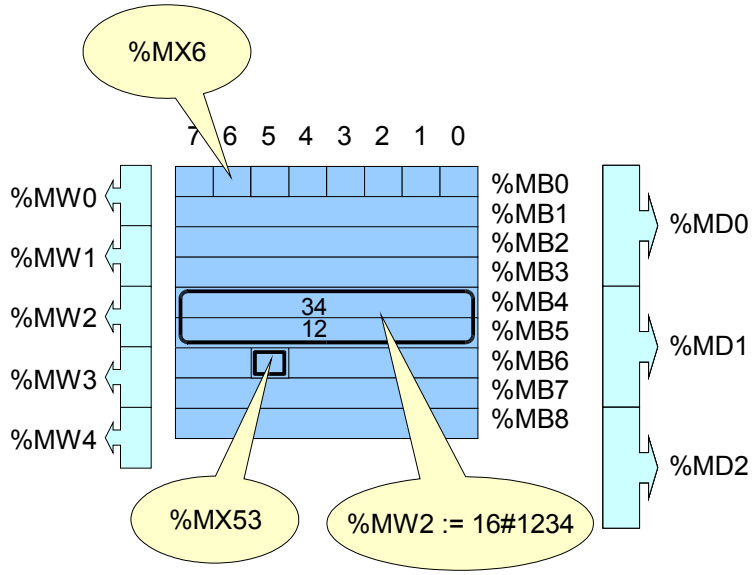


Figure 9 Marking PLC memory according to IEC

Direct addresses in PLC programs can be expressed in a traditional way used in the Mosaic programming language. The compiler automatically detects which manner of expression was used.

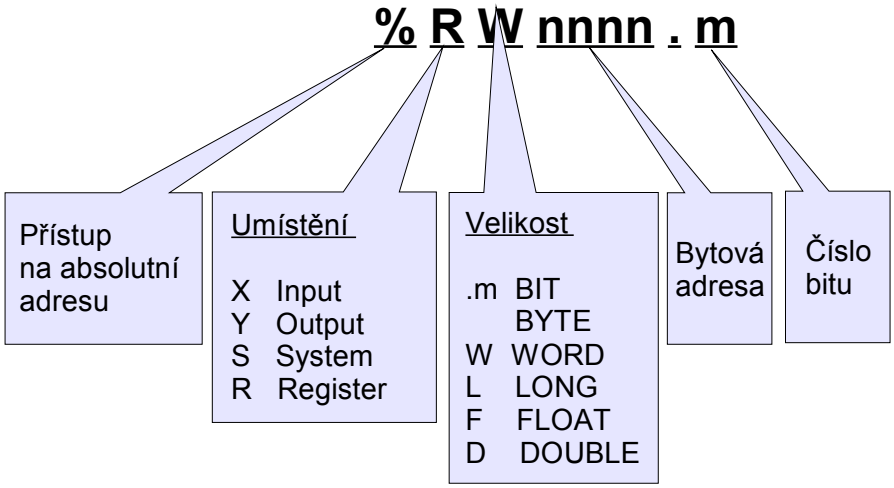


Figure 10 Traditional marking of direct addreses of variables in a PLC

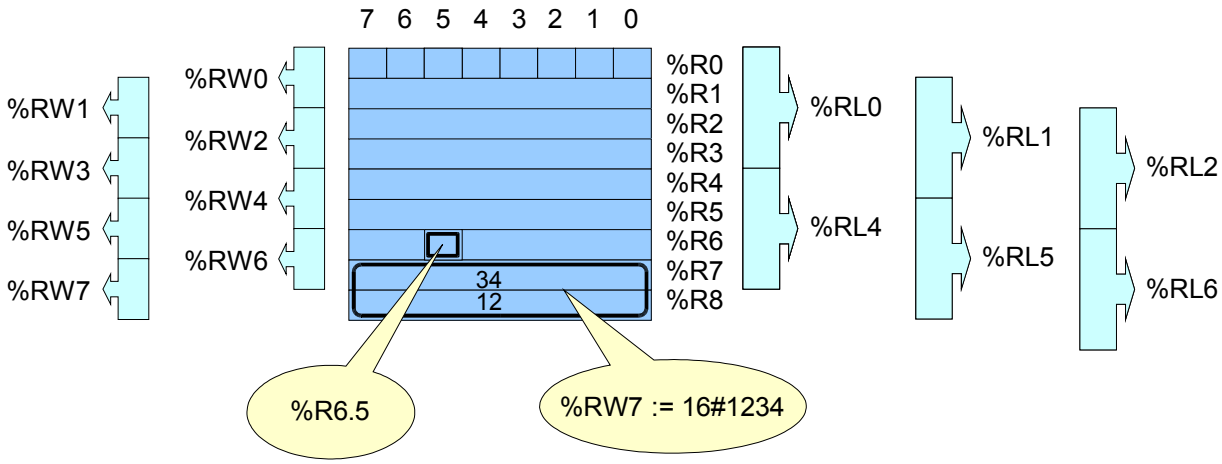


Figure 11 Traditional marking of PLC memory within the Mosaic environment

Expressions **`%MB10`** (according to IEC) and **`%R10`** (traditional) mark the same location in the memory. The expression **`%RW152.9`** marks the ninth bit of a **`WORD`** size variable located in the memory on the address **`152`**. By variables which occupy more than one byte in the memory, the least important byte is allocated to the lowest address and vice versa (Little Endian).

The specification of a direct address in the variables declaration can be used only in **VAR_GLOBAL** and **VAR_EXTERNAL** classes. The keyword **AT** which introduces the direct address variable is located between the variable name and data type specification.

Variables that have a stated direct address without the name of the variable, in their declaration, are named represented variables. When the program accesses these variables, their addresses are used instead of their names. This can be seen by the variables **%MB121** and **%R122** in example 3.21.

Example 23 Address specification in variables declaration

```
VAR_GLOBAL
  SymbolicVar AT %MB120 : USINT;
              AT %MB121 : USINT;
              AT %R122  : USINT := 242;
  counterOut  AT %Y0.0  : BOOL;              // PLC output
END_VAR

PROGRAM ExampleDirectVar
  VAR_EXTERNAL
    AT %S6              : USINT;             // second counter
    AT %X0.0, AT %X0.1  : BOOL;              // PLC input
  END_VAR
  VAR
    counter : CTU;
  END_VAR

  SymbolicVar := %MB121 + %R122;
  counter(CU := %X0.0, R := %X0.1, PV := 100, Q => counterOut);
END_PROGRAM
```

Direct addresses are used for declaring such variables, which location should not be changed during program editing. An example can be variables intended for visualization programs or variables which represent PLC inputs or outputs.

If an address is not stated in the variable declaration, the compiler will locate the symbolic variable into the PLC memory (assign an address). The compiler will also ensure that the variables do not overlap each other in the memory.

By directly represented variables, the programmer is the one who decides where the variables will be located in the memory and he has to ensure that no unwanted collision of variables occurs (their addresses overlap in the memory).

3.3.7 Variable initialization

The programming model according to the IEC 61 131 standard ensures that every variable gets an assigned value (initialization value) upon control system restart. This value can be:

- A value which the variable had at the moment of configuration element stop – typically during control system failure (retained value)
- Initial values specified by the user (stated in variable declaration)
- Predefined (default) initial values according to data type

The user can declare that a variable is to be ***retentive*** (this means its last value will be retained) by using the **RETAIN** qualifier. This qualifier can be used for global variables only.

It is possible to specify, within the frame of the declaration, variables using variable values. If the initialization value is not stated in the declaration, then the variable will be initialized using the initial value of the used data type.

Initial variable value

The initial value of a variable after system restart shall be determined according to the following rules:

- If the starting operation is a "warm restart", the initial values of retentive variables shall be their retained values.
- If the operation is a "cold restart", the initial values of retentive variables shall be the user-specified initial values.
- Non-retained variables shall be initialized to the user-specified initial values, or to the default value for the associated data type of any variable for which no initial value is specified by the user.
- Variables which represent inputs of the programmable control system shall be initialized in an implementation-dependent manner.
- Variables representing system outputs shall be initialized with the value 0, which corresponds to the “no power supply” state

By **VAR_EXTERNAL** class variables, the initial values cannot be assigned, because they are really only links to variables which are declared on a different place in the program. It is not also possible to declare initialization by **VAR_IN_OUT** class variables, because these variables contain only pointers to variables but not variables themselves.

Example 24 Variable initialization

```
TYPE
  MY_REAL : REAL := 100.0;
END_TYPE

VAR_GLOBAL RETAIN
  remanentVar1 : BYTE;
  remanentVar2 : BYTE := 56;
END_VAR

PROGRAM ExampleInitVar
  VAR
    localVar1  : REAL;
    localVar2  : REAL := 12.5;
    localVar3  : MY_REAL;
  END_VAR
  VAR_TEMP
    tempVar1   : BYTE;
    tempVar2   : REAL;
  END_VAR

    tempVar1   := remanentVar1 AND remanentVar2;
    tempVar2   := localVar1 + localVar2;
END_PROGRAM
```

During a cold system restart, the backed up variable **remanentVar1** will have a initialization value of 0 according to the initialization value of the **BYTE** data type. The variable **remanentVar2** will have a initialization value of 56, because this value is preset in the variables declaration.

During a warm system restart, the variables **remanentVar1** and **remanentVar2** will have such values, that these variables had when the system was switched off.

The variable **localVar1** will independently on the type of restart have a initialization value of 0, because the initialization value is not stated in the variables and so a predefined initialization value according to the **REAL** data type will be used. The variable **localVar2** will after restart always have an initialization value of **12.5**. The variable **localVar3** will after restart have an initialization value of **100.0**, because it is the initialization value derived from the **MY_REAL** data type.

3.4 Program organization units

Program Organization Units (POUs) are the ***function***, ***function block*** and ***program***.

These program organization units can be delivered by the manufacturer, or programmed by the user.

Program organization units are ***not recursive***; that is, the invocation of a program organization unit must not cause the invocation of another program organization unit of the same type!
In other words, the POU cannot call itself.

3.4.1 Function

For the purposes of programmable controller programming languages, a *function* is defined as a program organization unit which, when executed, yields exactly one data element, which is considered to be the function result, and arbitrarily many additional output elements (the function result can be multi-valued, e.g., an array or structure). The invocation of a function can be used in textual languages as an operand in an expression.

Functions contain no internal state information, i.e., invocation of a function with the same arguments (input parameters) shall always yield the same values (output).

Function declaration

Function declaration contains these elements

- Key words FUNCTION followed by the name of the declared function, colon and value data type, which the function will have to return
- A VAR_INPUT definition, specifying the names and types of the function's input variables
- Definition of local VAR variables or VAR_TEMP, specifying the names and types of internal variable functions
- A VAR CONSTANT definition
- The function body stated in the ST language. The function body specifies operations that should be executed above the input parameters for the purporse of assigning one

or more variable values, which have the same name as the function and which represent the return value of the function
- The terminating keyword END_FUNCTION.

Function calling

It is possible to call a function stating the function followed by the handed over parameters in round brackets in the ST programming language. The number of data types of handed over parameters must correspond to the input variables in the function definition. If the names of the function input variables are not stated in the calling, then the order of the parameters must exactly correspond to the order of input variables in the function definition. If the parameters are assigned to the names of the input parameters (formal call), then the order of parameters during calling is not important.

Example 25 Function definition and its calling in ST

```
FUNCTION MyFunction : REAL
  VAR_INPUT
    r, h    : REAL;
  END_VAR
  VAR CONSTANT
    PI      : REAL := 3.14159;
  END_VAR

  IF r > 0.0 AND h > 0.0
  THEN MyFunction := PI * r**2 * h;
  ELSE MyFunction := 0.0;
  END_IF;
END_FUNCTION


PROGRAM ExampleFunction
  VAR
    v1, v2  : REAL;
  END_VAR

  v1 := MyFunction( h := 2.0, r := 1.0);
  v2 := MyFunction( 1.0, 2.0);
END_PROGRAM
```

The **MyFunction** function in example 3.23 has two input variables defined **r** and **h** of the **REAL** type. The return value of this function is of the **REAL** type and is represented by the name **MyFunction**. In the calling of this function **v1 := MyFunction(h := 2.0, r := 1.0)** the names of the input variables are stated. In this case the order of input parameters in brackets is not important. The calling **v2 := MyFunction(1.0, 2.0)** does not contain names of input variables and so the input parameters are expected to be in such an order, in which the input variables are declared in the function declaration. Both callings showed in the example are equal and give the same result.

3.4.1.1 Standard functions

Standard functions usable in all programming languages for PLCs are described in detail in the IEC 61 131-3 standard, section 2.5.1.5. A summary of standard functions that are supported by the Mosaic environment compiler is shown in this sectio

Function Overloading

A function or operation is said to be ***overloaded*** when it can operate on input data elements of various types within a generic type designator. For instance, an overloaded addition function on generic type ANY_NUM can operate on data of types LREAL, REAL, DINT, INT, and SINT.

When a programmable control system supports an overloaded standard function, this function can apply to all data types of the given generic type which are supported by that system.

The information on what functions are overloaded are specified below. User-defined functions cannot be overloaded.

If all formal input parameters of a standard function are of the same generic type, then also all current parameters shall be of the same type. If necessary, functions for type *conversion* can be used for this purpose. The output value of a function will then be of the same type as current outputs.

Extensible functions

Some standard functions are ***extensible***, that is they are allowed to have a variable number of inputs, and shall be considered as applying the indicated operation to all its inputs. The maximum number of inputs of an extensible function is not limited.

Standard function division

Standard functions are divided into several basic groups:

- Functions for type conversion
- Numerical functions
- numerical functions of one variable
- arithmetic functions of more variables
- Bit string functions
- bit rotation
- Boolean functions
- Selection functions
- Comparison functions
- Character string functions
- Functions of time date types
- Functions of enumerated data types

The column with the name of **Ovr** in the following tables specifies whether a function is overloaded. The column with the name of **Ext** specifies whether the given function is extensible. See the Annex for the correct specification of standard functions.

Table.13 Standard functions, type conversion group

Standard functions, type conversion group					
Function name	**Input data type**	**Output data type**	**Function description**	**Ovr**	**Ext**
…_TO_…	ANY	ANY	Conversion of data type in the first position to data type specified on the second position	yes	no
TRUNC	ANY_REAL	ANY_INT	„Truncation“	yes	no

Table.14 Standard functions, one variable numerical function group

Standard functions, one variable numerical function group				
Function name	**Input / Output data type**	**Function description**	**Ovr**	**Ext**
ABS	ANY_NUM / ANY_NUM	Absolute value	ano	ne
SQRT	ANY_REAL / ANY_REAL	Square root	ano	ne
LN	ANY_REAL / ANY_REAL	Natural logarithm	ano	ne
LOG	ANY_REAL / ANY_REAL	Logarithm base 10	ano	ne
EXP	ANY_REAL / ANY_REAL	Natural exponential function	ano	ne
SIN	ANY_REAL / ANY_REAL	Sine of input angle in radians	ano	ne
COS	ANY_REAL / ANY_REAL	Cosine of input angle in radians	ano	ne
TAN	ANY_REAL / ANY_REAL	Tangent of input angle in radians	ano	ne
ASIN	ANY_REAL / ANY_REAL	Principal arc sine	ano	ne
ACOS	ANY_REAL / ANY_REAL	Principal arc cosine	ano	ne
ATAN	ANY_REAL / ANY_REAL	Principal arc tangent	ano	ne

Table.15 Standard functions, numerical function group - arithmetic functions of more variables

Standard functions, numerical function group - arithmetic functions of more variables					
Function name	**Input / Output data type**	**Symbol**	**Function description**	**Ovr**	**Ext**
ADD	ANY_NUM, .. ANY_NUM / ANY_NUM	+	Sum OUT:=IN1+ IN2+…+INn	yes	yes
MUL	ANY_NUM, .. ANY_NUM / ANY_NUM	*	Multiplication OUT:=IN1* IN2*…*INn	yes	yes
SUB	ANY_NUM, ANY_NUM / ANY_NUM	-	Subtraction OUT:=IN1-IN2	yes	no
DIV	ANY_NUM, ANY_NUM / ANY_NUM	/	Division OUT:=IN1/IN2	yes	no
MOD	ANY_NUM, ANY_NUM / ANY_NUM		Modulo OUT:=IN1 modulo IN2	yes	no
EXPT	ANY_REAL, ANY_NUM / ANY_REAL	**	Exponetiation OUT:=IN1**IN2	yes	no
MOVE	ANY_NUM / ANY_NUM	:=	Movement, assignment OUT:=IN	yes	no

Table.16 Standard functions, bit string function group - bit rotation

Standard functions, bit string function group - bit rotation				
Function name	**Input / Output data type**	**Function description**	**Ovr**	**Ext**
SHL	ANY_BIT, N / ANY_BIT	Shift left OUT := IN left shifted by N bits, zero-filled on right	yes	no
SHR	ANY_BIT, N / ANY_BIT	Shift right OUT := IN right shifted by N bits, zero-filled on left	yes	no
ROR	ANY_BIT, N / ANY_BIT	Rotate right OUT := IN right rotated by N bits, circular	yes	no
ROL	ANY_BIT, N / ANY_BIT	Rotate left OUT := IN left rotated by N bits, circular	yes	no

Table.17 Standard functions, bit string function group - Boolean functions

Standard functions, bit string function group - Boolean functions					
Function name	**Input / Output data type**	**Symbol**	**Function description**	**Ovr**	**Ext**
AND	ANY_BIT, .. ANY_BIT / ANY_BIT	&	Logical product, „conjunc-tion“, OUT:=IN1& IN2&…&INn	yes	yes
OR	ANY_BIT, .. ANY_BIT / ANY_BIT		Logical add, „or“, inclusive OR, OUT:=IN1 OR IN2 OR … OR INn	yes	yes
XOR	ANY_BIT, .. ANY_BIT / ANY_BIT		Exclusive add, „either - or“, exclusive OR OUT:=IN1 XOR IN2 XOR … XOR INn	yes	yes
NOT	ANY_BIT / ANY_BIT		Negation, „no“,	yes	no

			OUT:=NOT IN1		

Table.18 Standard functions, selection function group

Standard functions, selection function group				
Function name	**Input / Output data type**	**Function description**	**Ovr**	**Ext**
SEL	BOOL, ANY, ANY / ANY	Binary selection OUT := IN0 if G = 0 OUT := IN1 if G = 1	yes	no
MAX	ANY, .. ANY / ANY	Maximum OUT := MAX(IN1, IN2, .. INn)	yes	yes
MIN	ANY, .. ANY / ANY	Minimum OUT := MIN(IN1, IN2, .. INn)	yes	yes
LIMIT	MN, ANY, MX / ANY	Limiter OUT := MIN(MAX(IN, MN), MX)	yes	no

Table.19 Standard functions - comparison function group

Standard functions, comparison function group				
Function name	Input / Output data type	Function description	Ovr	Ext
GT	ANY, .. ANY / BOOL	Decreasing sequence OUT:=(IN1> IN2)& (IN2>IN3)&…&(INn-1>INn)	yes	yes
GE	ANY, .. ANY / BOOL	Monotonic sequence (downwards) OUT:=(IN1>= IN2)& (IN2>=IN3)&…&(INn-1>=INn)	yes	yes
EQ	ANY, .. ANY / BOOL	Equality OUT:=(IN1= IN2)& (IN2=IN3)&…&(INn-1=INn)	yes	yes
LE	ANY, .. ANY / BOOL	Monotonic sequence (upwards) OUT:=(IN1<= IN2)&	yes	yes

LT	ANY, .. ANY / BOOL	(IN2<=IN3)&…&(INn-1<=INn) Increasing sequence OUT:=(IN1< IN2)& (IN2<IN3)&…&(INn-1<INn)	yes	yes
NE	ANY, ANY / BOOL	Inequality OUT := (IN1<>IN2)	yes	no

Table.20 Standard functions, function group above character string

Standard functions, function group above character string				
Function name	**Input / Output data type**	**Function description**	**Ovr**	**Ext**
LEN	STRING / INT	OUT := LEN(IN); String length IN	no	no
LEFT	STRING, ANY_INT / STRING	OUT := LEFT(IN, L); From the IN input string shift L characters from the left into the output string	yes	no
RIGHT	STRING, ANY_INT / STRING	OUT := RIGHT(IN, L); From the IN input string shift L characters from the right into the output string	yes	no
MID	STRING, ANY_INT, ANY_INT / STRING	OUT := MID(IN, L, P); From the IN input string shift L characters from the P numbered character into the output string	yes	no
CONCAT	STRING, …. STRING /STRING	OUT := CONCAT(IN1, IN2, ...); Connection of individual input strings into the output string	no	yes
INSERT	STRING, STRING, ANY_INT / STRING	OUT := INSERT(IN1, IN2, P); Insertion of string IN2 intor string IN1 beginning from the P numbered position	yes	yes
DELETE	STRING, ANY_INT, ANY_INT / STRING	OUT := DELETE(IN, L, P); Deletion of L characters from the string beginning from the P numbered position	yes	yes
REPLACE	STRING, STRING, ANY_INT, ANY_INT / STRING	OUT := REPLACE(IN1, IN2, L, P); Replacement of L characters of the IN1 string with characters from the IN2 string, replacement from position P	yes	yes
FIND	STRING, STRING / INT	OUT := FIND(IN1, IN2); Find position of first character of the first occurance of string IN2 in string IN1	yes	yes

Table.21 Standard functions, function group with date and time types

Standard functions, function group with date and time types					
Function name	**IN1**	**IN2**	**OUT**	**Ovr**	**Ext**
ADD_TIME	TIME	TIME	TIME	no	no
ADD_TOD_TIME	TIME_OF_DAY	TIME	TIME_OF_DAY	no	no
ADD_DT_TIME	DATE_AND_TIME	TIME	DATE_AND_TIME	no	no
SUB_TIME	TIME	TIME	TIME	no	no
SUB_DATE_DATE	DATE	DATE	TIME	no	no
SUB_TOD_TIME	TIME_OF_DAY	TIME	TIME_OF_DAY	no	no
SUB_TOD_TOD	TIME_OF_DAY	TOD	TIME	no	no
SUB_DT_TIME	DATE_AND_TIME	TIME	DATE_AND_TIME	no	no
SUB_DT_DT	DATE_AND_TIME	DT	TIME		
MULTIME	TIME	ANY_NUM	TIME	yes	no
DIVTIME	TIME	ANY_NUM	TIME	yes	no
CONCAT_DATE_TOD	DATE	TIME_OF_ DAY	DATE_AND_TIME	no	no
Type conversion function					
DATE_AND_TIME_TO_TIME_OF_DAY, DAT_TO_TIME DATE_AND_TIME_TO_DATE, DAT_TO_DATE					

TOD … TIME_OF_DATE
DT … DATE_AND_TIME

3.4.2 Function blocks

For the purposes of programming according to IEC 61 131-3, a ***function block*** is a program organization unit which, when executed, yields one or more values. Multiples, named ***instances*** (copies) of a function block can be created. Each instance has an associated identifier (the *instance name*), and a data structure containing its output and internal variables. All the values of the output variables and the necessary internal variables of this data structure persist from one execution of the function block to the next. Therefore, invocation of a function block with the same arguments (input variables) need not always yield the same output values. The function block instances are created by using a declared type of a function block within VAR or VAR_GLOBAL structure.

Any function block type which has already been declared can be used in the declaration of another function block type or program type.

The scope of an instance of a function block is local to the program organization unit in which it is *instantiated* (i.e. where its named copy is created), unless it is declared to be global.

The following example describes the procedure for the declaration of a function block, creation of its instance in a program and its invocation (execution).

Example 26 Function block in language ST

```
FUNCTION_BLOCK fbStartStop              // FB declaration
  VAR_INPUT
    start       : BOOL R_EDGE;          // input variable
    stop        : BOOL R_EDGE;
  END_VAR
  VAR_OUTPUT
    output      : BOOL;                 // output variable
  END_VAR

  output := (output OR start) AND not stop;
END_FUNCTION_BLOCK


PROGRAM ExampleFB
  VAR
    StartStop   : fbStartStop;          // FB instance
    running     : BOOL;
  END_VAR

  // invocation of function block instance
  StartStop( stop := FALSE, start := TRUE, output => running);

  // alternative FB invocation
  StartStop.start := TRUE;
  StartStop.stop  := FALSE;
  StartStop();
  running         := StartStop.output;

  // call with incomplete parameter list
  StartStop( start := TRUE);
  running := StartStop.output;

END_PROGRAM
```

The input and output variables of an instance of a function block can be represented as elements of the structure data type.

If an instance of a function block is global, it can be then also declared as retentive. In that case, it is valid only for internal and output parameters of a function block.

From outside, only input and output parameters of a function block are accessible, that is, the internal variables of a function block are hidden to the user of the function block.. The assignment of a value from outside to an output variable of a function block is not allowed, this value is assigned from inside by the function block itself. The assignment of a value of an input of a function block is allowed anywhere in a superior POU (typically it is part of invocation of a function block).

Function block declaration

- The delimiting keywords for declaration of function blocks are FUNCTION_BLOCK...END_FUNCTION_BLOCK.
- A function block can have more than one output parameter declared textually by the construct VAR_OUTPUT.
- The values of variables which are passed to the function block via a VAR_IN_OUT VAR_EXTERNAL construct can be modified from within the function block.
- In textual declarations, the R_EDGE and F_EDGE qualifiers can be used to indicate an edge-detection function on Boolean inputs. This shall cause the implicit declaration of a function block of type R_TRIG or F_TRIG, respectively, to perform the required edge detection. By doing this, the default declaration of the function block R_TRIG or F_TRIG is invoked.
- The construct defined for initialization of functions is used also for declaration of the default inputs of the function block and for the initial values of its internal and output variables.

By means of the structureVAR_IN_OUT, only variables can be passed to the function block (function blocks instances cannot be passed). "Cascading" of VAR_IN_OUT constructions is permitted.

3.4.2.1 Standard function blocks

Standard function blocks are defined in detail in the standard IEC 61 131-3 section 2.5.2.3.

Standard function blocks can be divided into the following groups (see Table 3.22).

- Bistable elements
- Edge detection
- Counters
- Timers

Standard function blocks are stored in library Standard_FBs_*.mlb.

Table.22 Overview of standard function blocks

Name of standard function block	Name of input parameter	Name of output parameter	Description
Bistable elements (flip-flop circuits)			
SR	S1, R	Q1	dominant setting (closing)
RS	S, R1	Q1	dominant deletion (opening)
Edge detection			
R_TRIG	CLK	Q	Rising edge detection
F_TRIG	CLK	Q	Falling edge detection
Counters			
CTU	CU, R, PV	Q, CV	Upward counter
CTD	CD, LD, PV	Q, CV	Down counter
CTUD	CU, CD, R, LD, PV	QU, QD, CV	Reversible counter
Timers			
TP	IN, PT	Q, ET	Pulse timer
TON (T--0)	IN, PT	Q, ET	Timer ON - rising edge delay
TOF (0--T)	IN, PT	Q, ET	Timer OFF - falling edge delay

The names, meaning and data types of variables used with standard function blocks:

Name of input / output	Meaning	Data type
R	Resetting input	BOOL
S	Setting input	BOOL
R1	Dominant resetting input	BOOL
S1	Dominant setting input	BOOL
Q	Output (standard)	BOOL
Q1	Output (only with flip-flop circuits)	BOOL
CLK	Clock (synchronization) signal	BOOL
CU	Input for up-counting	BOOL
CD	Input for down counting	BOOL
LD	Counter preset (initial value)	INT
PV	Counter preset (end value)	INT
QD	Output (down counter)	BOOL
QU	Output (upward counter)	BOOL
CV	Current value (counter)	INT
IN	Input (timer)	BOOL
PT	Timer preset	TIME
ET	Timer current value	TIME
PDT	Preset - date and time	DT
CDT	Current value - date and time	DT

3.4.3 Programs

A ***program*** is defined in IEC 1131-1 as a "logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a programmable control system."

In other words, functions and function blocks can be compared to subroutines, while a POU program is the main program. The declaration and usage of *programs* is identical to the declaration and usage of function blocks with the following differences:

- The delimiting keywords for program declarations are PROGRAM...END_PROGRAM.
- *Programs* can only be instantiated within *resources*, as defined in section 3.5, while *function blocks* can only be instantiated within *programs* or other *function blocks*.
- Programs can invocate functions and function blocks, while the invocation of programs from functions or function blocks is not possible.

Example 27 POU Program in language ST

```
PROGRAM test
  VAR
    motor1 : fbMotor;
    motor2 : fbMotor;
  END_VAR

  motor1( startMotor := sb1, stopMotor := sb2,
          wye => km1, delta => km2);
  motor2( startMotor := sb3, stopMotor := sb4,
          Wye => km3, delta => km4);

END_PROGRAM
```

When writing a program in language ST, it should be realized that a POU *program* as well as a function block is only "an instruction", in which a data structure and algorithms performed on this data structure. To perform a defined program it is necessary to produce its instance and associate with a program to some of standard tasks, in which it then will be performed. These operations are described in the following section.

3.5 Configuration elements

Configuration elements describe run-time features of programs and associate the execution of programs with the concrete PLC hardware. They represent the top "rule" of the entire program for a PLC.

When programming PLC systems the following configuration elements are used:

- ***Configuration*** – specifies a PLC system which shall execute all programmed POUs.
- ***Resource*** – specifies a processor module in a PLC which shall ensure program execution.
- ***Task*** – assigns a task (process) within which the relevant *PROGRAM* POU shall be executed.

Example 28

```
CONFIGURATION Plc1
  RESOURCE CPM
    TASK FreeWheeling(Number := 0);
    PROGRAM prg WITH FreeWheeling : test ();
  END_RESOURCE
END_CONFIGURATION
```

In the MOSAIC development environment all configuration elements are generated automatically after the configuration dialogs have been filled-in.

3.5.1 Configuration

The configuration marks a PLC system which provides resources for executing the user program. In other words, a configuration marks control systems for which a user program is intended for.

Configuration declaration

- The keywords delimiting c*onfiguration* are CONFIGURATION….END_ CONFIGURATION
- The keyword CONFIGURATION is followed by configuration naming, in the MOSAIC development environment the name of configuration corresponds to the name of project
- *Configuration* serves as a frame for *Resource* definition.

Example 29

```
CONFIGURATION configuration_name

  // resource declaration

END_CONFIGURATION
```

3.5.2 Resources

A resource defines which module within the PLC provides the calculation operation for the user program execution. In the PLCs of TC700 series it is always the system processor module.

Resource declaration

- The keywords delimiting *a resource* are RESOURCE….END_ RESOURCE.
- The keyword RESOURCE is followed by resource naming, in the MOSAIC development environment it is "CPM" by default.
- *Resources* can be declared only within *configuration.*

Example 30

```
CONFIGURATION Plc1
  RESOURCE CPM

    // task declaration

    // assignment of programs to declared tasks
  END_RESOURCE
END_CONFIGURATION
```

3.5.3 Tasks

For the purposes of IEC 61 131-3, a *task* is defined as an execution control element which is capable of invoking, either on a periodic basis or upon the occurrence of the rising edge of a specified Boolean variable, the execution of a set of program organization units. These can include *programs* and *function blocks* declared within them. For the Mosaic environment, the term "task" is identical with the conventionally used term "process".

Task declarations

- The keyword for a task is TASK
- The keyword TASK is followed by task naming.
- The task name is followed by task features, concretely with the number of the corresponding process.
- *Tasks* can be declared only within *resource* declaration.

Assigning of programs to tasks

- Program association with a concrete task begins with the keyword PROGRAM, by which the instance of the given program is produced automatically.
- The keyword PROGRAM is followed by the program instance name.

- The keyword WITH specifies the task name to which the program shall be associated with.
- The name of the associated program including input and output parameters specification follows the colon.
- More programs can be associated with one task, the order of their execution within the task corresponds to the order as they were associated.
- Program association can be declared only within the *Resource* declaration.

Example 31

```
CONFIGURATION Plc1
  RESOURCE CPM
    TASK FreeWheeling(Number := 0);
    PROGRAM prg WITH FreeWheeling : test ();
  END_RESOURCE
END_CONFIGURATION
```

4 TEXT LANGUAGES

The IEC 61 131-3 standard defines two text languages: IL, Instruction List and ST, Structured Text. Both these languages are supported by Mosaic compilers.

4.1 IL Instruction list language

The instruction list language is a low level assembler type language. This language belongs to line orientated languages.

4.1.1 Instructions in IL

The list of instruction contains a sequence of *instructions*. Each instruction (statement) begins on a new line and contains an *operator* which can be supported by a modifier and if it is necessary for the specific instructions, it can also contain one or more operands separated by comas. Instead of operands, random data representations defined for literals can be used (see chapter 3.1.2) and variables (see chapter 3.3).

Example 32 Program in IL

```
VAR_GLOBAL
  AT %X1.2       : BOOL;
  AT %Y2.0       : BOOL;
END_VAR

PROGRAM Example_IL
  VAR
    tmp1, tmp2   : BOOL;
  END_VAR

Step1:  LD       %X1.2    // load bit from PLC input
        AND      tmp1     (* AND temporary variable *)
        ST       %Y2.0    (* store to PLC output *)
                          (* empty instruction *)
Step2:                    (* label *)
        LDN      tmp2
END_PROGRAM
```

4.1.2 Operators, modifiers and operands

Standard operators, together with acceptable modifiers are stated Table. 4.1 to Table. 4.4. If not stated differently in tables, the semantics of operators is as follows:

```
result := result OPERATOR operand
```

That means that the expression value, which is evaluated, is replaced by its new value, which is processed from its current value using operators or operands, e.g. instruction **AND %X1** is interpreted as:

```
result:= result AND %X1
```

Comparing operators are interpreted with current values left from the comparing sign and operand on the right from the comparing sign. The result of comparing is a bool variable, e.g. instruction **LT %IW32** will have a bool result of "1", if the current result is smaller than the value 32 of the input word; in all other cases the result will be a bool "0"

The **N** modifier marks the bool negation of the operand. E.g. instruction **ORN %X1.5** is interpreted as:

```
result:= result OR NOT %X1.5
```

The modifier of the left bracket **(** means that the operator should be "deffered", i.e. the operator execution deffered until the operator of the right bracket is found **)**. E.g. instruction sequence

```
AND(        %X1.1
OR          %X1.3
)
```

is interpreted as:

```
result:= result AND (%X1.1 OR %X1.3)
```

Table 23 Operators and modifiers for ANY_BIT data type

ANY_BIT operators		
Operator	**Modifier**	**Function description**
LD	**N**	Setting of current results to the value equal to the operand
AND	**N, (**	Bool AND
OR	**N, (**	Bool OR
XOR	**N, (**	Bool XOR
ST	**N**	Saves the current result on the location of the operand
S		Sets a bool operand to "1" The operation is executed only if the current result is a bool "1"
R		Deletes the bool operand to "0" The operation is executed only if the current result is a bool "1"
)		Evaluation of a deffered operation

Some operators can have more modifiers added at once. On example operator **AND** it has four different forms as seen on Table. 4.2.

Table 24 AND operator modifiers

AND	Bool AND
AND(	Deferred bool AND
ANDN	Bool AND with negated operand
ANDN(	Deffered bool AND with negated result

Table 25 Operators and modifiers for ANY_NUM data type

ANY_NUM Operators		
Operator	**Modifier**	**Function description**
LD	**N**	Setting of current results to the value equal to the operand
ST	**N**	Saves the current result on the location of the operand
ADD	**(**	Add operand to result
SUB	**(**	Deduct operand from result
MUL	**(**	Multiply result with operand
DIV	**(**	Divide result with operand
GT	**(**	Compare result > operand
GE	**(**	Compare result >= operand
EQ	**(**	Compare result = operand
NE	**(**	Compare result <> operand
LE	**(**	Compare result <= operand
LT	**(**	Compare result < operand
)		Evaluation of last deffered operation

Table 26 Operators and modifiers for jumps and calls

ANY_BIT Operators		
Operator	**Modifier**	**Function description**
JMP	**C, N**	Jump to identifier
CAL	**C, N**	Function block call
Func_name		Function call
RET	**C, N**	Return from function or function block

The **C** modifier means, that the assigned instruction may only be executed in the case that the currently evaluated result is a bool “1” (or bool “0”, if the operator has an **N** modifier added).

4.1.3 IL language user function definition

Example 33 IL language user function definition

```
FUNCTION UserFun : INT
  VAR_INPUT
    val     :     INT;            // input value
    minVal  :     INT;            // minimun
    maxVal  :     INT;            // maximum
  END_VAR

        LD      val               // load input value
        GE      minVal            // test if val >= minVal
        JMPC    NXT_TST           // jump if OK
        LD      minVal            // low limit value
        JMP     VAL_OK
NXT_TST:
        LD      val               // load input value
        GT      maxVal            // test if val > maxVal
        JMPCN   VAL_OK            // jump if not
        LD      maxVal            // high limit value
VAL_OK: ST      UserFun           // return value
END_FUNCTION
```

4.1.4 Calling function in IL language

Functions in the IL language are called by placing the function name into the operator array. The current result is used as the first function parameter. If further parameters are needed, they are entered into the operand array and are separated using a coma. The value returned by the function after successfully executing the instruction **RET** or after achieving the physical end of the function becomes the current value.

Other two possibilities for calling a relative function are the same as in the ST language. The function name is followed by a list of parameters handed over into the function in round brackets. The list of parameters can be with parameter names (formal call) or without (informal call).

In example 4.3 all described calls can be found. Called functions are user defined in example 4.2.

Example 34 Function calling in IL

```
VAR_GLOBAL
  AT %YW10 : INT;
END_VAR

PROGRAM Example_IL1
  VAR
     count     : INT;
  END_VAR
```

```
            // calling function, first parameter is current result
            LD      Count
            UserFun 100, 1000
            ST      %YW10

            // calling function using an informal call
            UserFun( Count, 100, 1000)
            ST      %YW10

            // calling function using a formal call
            UserFun( val := Count, minVal := 100, maxVal := 1000)
            ST      %YW10

END_PROGRAM
```

4.1.5 Calling a function block in IL

Function blocks in the IL language are called conditionally or unconditionally using the operator **CAL** (Call). As shown in the following example, the function block can be called in two ways.

The function block can be called by its name followed by a list of parameters (formal call). A second possibility is saving the parameters into relevant memory locations of the function block instance and then calling them (informal call). Both methods can be combined.

Example 35 Calling a function block in IL

```
VAR_GLOBAL
  in1   AT %X1.0   : BOOL;
  out1  AT %Y1.0   : BOOL;
END_VAR

PROGRAM Example_IL2
  VAR
    timer       : TON;
    timerValue  : TIME;
  END_VAR

  // calling FB using an informal call
  LD    in1
  ST    timer.IN        // parameter IN
  LD    T#10m12s
  ST    timer.PT        // parameter PT
  CAL   timer           // calling FB TON
  LD    timer.ET
  ST    timerValue      // timer value
  LD    timer.Q
  ST    out1            // timer output

  // calling FB using an informal call
  CAL   timer( IN := in1, PT := T#10m12s, Q => out1, ET => timerValue)
```

```
    // another way
    LD    in1
    ST    timer.IN
    CAL   timer( PT := T#10m12s, ET => timerValue)
    LD    timer.Q
    ST    out1

END_PROGRAM
```

4.2 ST structured text language

The ST language is one of the languages defined by IEC 61 131-3. It is a high-performance higher programming language originating from well-known languages such as Ada, Pascal and C. It is object-oriented and contains all substantial elements of a modern programming language including branching (IF-THEN-ELSE and CASE OF) and iteration statements (FOR, WHILE a REPEAT). These elements can be nested. This language is a perfect tool to define complex function blocks.

An algorithm written in the ST language can be divided into ***statements***. Statements are used to calculate and assign value, to control program execution flow and to invocate or to end or terminate a POU. A part of a program calculating a value is called an ***expression***. The expressions produce the value necessary to execute statements.

4.2.1 Expressions

An *expression* is a construct which, when evaluated, yields a value corresponding to one of the data types defined in section 3.2.

An expression is composed of an operator and an operand. An *operand* can be a literal, a variable, a function invocation another expression.

The ST language operators are shown in Table 4.5.

Table.27 Operators in language ST

Operator	Operation	Priority
( )	Parentheses	Highest
**	Exponentiation	
- **NOT**	Sign Complement	
* / **MOD**	Multiplication Division Modulo	
+ -	Addition Subtraction	
<, >, <= ,>=	Comparison	
= <>	Equality Inequality	
&, AND	Boolean AND	
XOR	Boolean exclusive OR	
OR	Boolean OR	Lowest

To operands of operators apply the same limitations as to the inputs of the corresponding functions defined in section 3.4.1.1. As an example, the result of the expression A**B is the same as the result of the function EXP(A, B) as it is defined in Table 3.14

The evaluation of an expression consists of applying the operators to the operands in a sequence defined by the operator priority shown in Table 4.5. The operator with highest precedence in an expression shall be applied first, followed by the operator of next lower precedence, etc., until evaluation is complete. Operators of equal precedence shall be applied as written in the expression from left to right.

-

Example 36 Operator priority during expression evaluation

```
PROGRAM EXAMPLE
  VAR                                  // local variables
    A       : INT := 2;
    B       : INT := 4;
    C       : INT := 5;
    D       : INT := 8;
    X,Y     : INT;
    Z       : REAL;
  END_VAR

  X := A + B - C * ABS(D);             // X = -34
  Y := (A + B - C) * ABS(D);           // Y = 8
  Z := INT_TO_REAL( Y);
END_PROGRAM
```

By the evaluation of the expression A + B- C * ABS(D) in example 4.5 we receive the value of -34. If another order of evaluation than stated is required, parentheses have to be used. Then, for the same values of variables, when evaluating the expression (A + B - C) * ABS(D), we receive the value of 8.

The functions are invoked as elements of the expressions comprising of the name of the function, followed by a list of arguments in parentheses.

When an operator has two operands, the leftmost operand shall be evaluated first. For example, in the expression COS(Y) * SIN(X) the expression COS(Y) shall be evaluated first, followed by SIN(X), followed by evaluation of the product.

Boolean expressions may be evaluated only to the extent necessary to determine the unambiguous resultant values. For instance, if C<=D, then only the expression (C>D) can be evaluated from the expression (C>D) & (F<A). Its value is Boolean zero with respect to the presumption and this is sufficient for the entire logical product to be Boolean zero. It is not then necessary to evaluate the second expression (F < A).

When an operator in an expression can be represented as one of the overloaded functions defined in section 3.4.1.1, conversion of operands and results shall follow the rule and examples given in this section.

4.2.2 Summary of statements in the ST language

A list of statements of the ST language is summarized in Table 4.6. The statements are terminated with a semicolon. The character of the line end is treated in this language in the same manner as the space character.

Table.28 List of examples of language ST

Statement	Description	Example	Note
:=	Assignment	A **:=** 22;	Assignment of a value calculated on the right side to the identifier on the left side
	Function block invocation	**InstanceFB**(par1 := 10, par2 := 20);	Function block invocation with parameters passing
IF	Selection statement	**IF** A > 0 **THEN** B := 100; **ELSE** B := 0; **END_IF;**	Selection of an alternative conditional to a Boolean expression
CASE	Selection statement	**CASE** kod **OF** 1 : A := 11; 2 : A := 22; **ELSE** A := 99; **END_CASE;**	Command block selection conditional to the value of the expression „code“

FOR	Iteration statement FOR	**FOR** i := 0 **TO** 10 **BY** 2 **DO**
		j := j + i;
		END_FOR;
			A multi-loop of a statement block with initial and end condition and increment value

WHILE	Iteration statement WHILE	**WHILE** i > 0 **DO**
		n := n * 2;
		END_WHILE;
			A multi-loop of a statement block with condition of termination of loop at the beginning

REPEAT	Iteration statement REPEAT	**REPEAT**
		k := k + i;
		UNTIL i < 20;
		END_REPEAT;
			A multi-loop of a statement block with condition of termination of loop at the end

EXIT	Loop termination	**EXIT;**	Premature termination of iteration statement
RETURN	Return	**RETURN**;	Leaving of a POU being just executed and return to the invoking POU
;	Empty statement	**;;**	

4.2.2.1 Assignment statement

The assignment statement replaces the current value of a single or multi-element variable by the result of evaluating an expression. An assignment statement consists of a variable reference on the left-hand side, followed by the *assignment operator* ":=", followed by the expression to be evaluated.

The assignment statement is very powerful. It can assign a simple variable, but also a whole data structure. As it can be seen in example 4.6 where the assignment statement A := B is used to replace the value of simple variable A by the current value of variable B (both variables are of basic type INT). However, assignment can be successfully used also for multi-element variables AA := BB and then, all items of the multi-element variable AA are rewritten by the items of the multi-element variable BB. The variables shall be of the same data type.

Example 37 Assignment of a simple and multi-element variable

```
TYPE
  tyRECORD : STRUCT
    serialNumber     : UDINT;
    colour           : (red, green, white, blue);
    quality          : USINT;
  END_STRUCT;
END_TYPE

PROGRAM EXAMPLE
  VAR                        // local variables
    A, B   : INT;
    AA, BB : tyZAZNAM;
  END_VAR

  A  := B;                     // simple variable assignment
  AA := BB;                    // multi-element variable assignment
END_PROGRAM
```

The assignment statement can also be used for the assignment of the function return value by placing the name of the function on the left-hand side of the assignment operator in the function declaration body. The function return value shall be the result of the last evaluation of this assignment statement.

Example 38 Assignment of function return value

```
FUNCTION EXAMPLE : REAL
  VAR_INPUT                    // input variables
    F, G   : REAL;
    S      : REAL  := 3.0;
  END_VAR

  EXAMPLE := F * G / S;        // function return value
END_FUNCTION
```

4.2.2.2 Function block call statement

Function blocks shall be invoked by a statement consisting of the name of the function block instance followed by a parenthesised list of arguments with assigned values.

The order of the parameters in the list if on no importance when invoking a function block. For each function block invocation all input variables do not need to be assigned. If a parameter does not have a value assigned before function block invocation, then the last assigned value shall be used (or the initial value, if no assignment has been performed yet).

Example 39 Function block call statement

```
// function block declaration
FUNCTION_BLOCK fb_RECTANGLE
  VAR_INPUT
    A,B                : REAL;          // input variables
  END_VAR
  VAR_OUTPUT
    perimeter, surface      : REAL;          // output variables
  END_VAR

  perimeter := 2.0 * (A + B); surface := A * B;
END_FUNCTION_BLOCK

// global variables
VAR_GLOBAL
  RECTANGLE : fb_RECTANGLE;             // global FB instance
END_VAR

// program declaration
PROGRAM main
  VAR
    o,s     : REAL;                   // local variables
  END_VAR

  // FB invocation with complete list of parameters
  RECTANGLE   ( A := 2.0, B := 3.0, obvod => o , plocha => s);
  IF   o > 20.0 THEN
    ....
  END_IF;

  // FB invocation with incomplete list of parameters
  RECTANGLE   ( B := 4.0, A := 2.5);
  IF   RECTANGLE . perimeter   > 20.0 THEN
    ....
  END_IF;
END_PROGRAM
```

4.2.2.3 IF statement

The IF statement specifies that a group of statements is to be executed only if the associated Boolean expression evaluates to be true (TRUE). If the condition is false, then either no statement is to be executed, or the statement group following the ELSE keyword (or the ELSIF keyword if its associated Boolean condition is true) is to be executed.

Example 40 IF statement

```
FUNCTION EXAMPLE : INT
  VAR_INPUT
    code      : INT;                    // input variable
  END_VAR

  IF code  < 10 THEN EXAMPLE:= 0;       //  at code < 10 function
                                            returns 0
  ELSIF code < 100 THEN EXAMPLE:= 1;   //   at 9 < code < 100 function
                                            returns 1
  ELSE EXAMPLE:= 2;                   // at code > 99 function returns 2
  END_IF;
END_FUNCTION
```

4.2.2.4 CASE statement

The CASE statement consists of an expression which shall evaluate to a variable of type INT (the "selector"), and a list of statement groups, each group being labelled by one or more integer or enumerated values or ranges of integer values, as applicable. It specifies that the first group of statements, one of whose ranges contains the computed value of the selector, shall be executed . If the value of the selector does not occur in a range of any case, the statement sequence following the keyword ELSE (if it occurs in the CASE statement) shall be executed. Otherwise, none of the statement sequences shall be executed.

Example 41 CASE statement

```
FUNCTION EXAMPLE: INT
  VAR_INPUT
    kod     : INT;              // input variable
  END_VAR

  CASE kod OF
    10      : EXAMPLE := 0;     // at code = 10 function
                                  returns 0
    20,77   : EXAMPLE := 1;     // at code  = 20 or code = 77 function
                                  returns 1
    21..55 : EXAMPLE := 2;      // at 20 < code < 56 function returns 2
    100     : EXAMPLE := 3;     // at code  = 100 function returns 3
  ELSE
              EXAMPLE := 4;     // otherwise function returns 4
  END_CASE;
END_FUNCTION
```

4.2.2.5 FOR statement

The FOR statement is used if the number of iterations can be predetermined, otherwise the WHILE or REPEAT constructs shall be used.

The FOR statement indicates that a statement sequence shall be repeatedly executed, up to the END_FOR keyword, while a progression of values is assigned to the FOR loop control variable. The control variable, initial value, and final value are expressions of the same integer type (e.g., SINT, INT, or DINT) and shall not be altered by any of the repeated statements. The FOR statement increments the control variable up or down from an initial value to a final value in increments determined by the value of an expression (this value defaults to 1). The test for the termination condition is made at the beginning of each iteration, so that the statement sequence is not executed if the initial value exceeds the final value.

Example 42 FOR statement

```
FUNCTION FACTORIAL : UDINT
  VAR_INPUT
    code     : USINT;                    // input variable
  END_VAR
  VAR_TEMP
    i        : USINT;                   // auxiliary variable
    tmp      : UDINT := 1;              // auxiliary variable
  END_VAR

  FOR i := 1 TO code DO
    tmp := tmp * USINT_TO_UDINT( i);
  END_FOR;
  FACTORIAL := tmp;
END_FUNCTION
```

4.2.2.6 WHILE statement

The WHILE statement causes the sequence of statements up to the END_WHILE keyword to be executed repeatedly until the associated Boolean expression is false. If the expression is initially false, then the group of statements is not executed at all. For instance, the loop FOR...END_FOR can be rewritten using the WHILE...END_WHILE construction. The example 4.17 can be rewritten by using the WHILE statement as follows:

Example 43 WHILE statement

```
FUNCTION FACTORIAL : UDINT
  VAR_INPUT
    code     : USINT;                    // input variable
  END_VAR
  VAR_TEMP
    i        : USINT;                   // auxiliary variable
    tmp      : UDINT := 1;              // auxiliary variable
  END_VAR

  i := kod;
  WHILE i <> 0 DO
    tmp := tmp * USINT_TO_UDINT( i);  i := i - 1;
  END_WHILE;
  FACTORIAL := tmp;
END_FUNCTION
```

It shall be a cycle error if a WHILE statement is used in an algorithm for which satisfaction of the loop termination condition or execution of an EXIT statement cannot be guaranteed.

4.2.2.7 REPEAT statement

The REPEAT statement causes the sequence of statements up to the UNTIL keyword to be executed repeatedly (and at least once) until the associated Boolean condition is true. For instance, the loop WHILE...END_WHILE can be rewritten using the REPEAT...END_REPEAT construction shown in the following example:

Example 44 REPEAT statement

```
FUNCTION FACTORIAL : UDINT
  VAR_INPUT
    code     : USINT;                    // input variable
  VAR_TEMP
    i        : USINT := 1;              // auxiliary variable
    tmp      : UDINT := 1;              // auxiliary variable
  END_VAR

  REPEAT
    tmp := tmp * USINT_TO_UDINT( i);  i := i + 1;
  UNTIL i > kod
  END_REPEAT;
  FACTORIAL := tmp;
END_FUNCTION
```

It shall be a cycle error if a REPEAT statement is used in an algorithm for which satisfaction of the loop termination condition or execution of an EXIT statement cannot be guaranteed.

4.2.2.8 EXIT statement

The EXIT statement is used to terminate iterations before the termination condition is satisfied.

When the EXIT statement is located within nested iterative constructs (statements FOR, WHILE, REPEAT), exit shall be from the innermost loop in which the EXIT is located, that is, control shall pass to the next statement after the first loop terminator (END_FOR, END_WHILE, or END_REPEAT) following the EXIT statement.

Example 45 EXIT statement

```
FUNCTION FACTORIAL : UDINT
  VAR_INPUT
    code    : USINT;                    // input variable
  END_VAR
  VAR_TEMP
    i       : USINT;                   // auxiliary variable
    tmp     : UDINT := 1;              // auxiliary variable
  END_VAR

  FOR i := 1 TO code
    IF i > 13 THEN
      tmp := 16#FFFF_FFFF; EXIT;
    END_IF;
    tmp := tmp * USINT_TO_UDINT( i);
  END_FOR;
  FACTORIAL := tmp;
END_FUNCTION
```

For calculation of the factorial for number greater than 13, the result will be greater than the maximum number that can be stored in the variable of UDINT type. This case in example 4.14 is treated by means of the EXIT statement.

4.2.2.9 RETURN statement

The RETURN statement shall provide early exit from a function, function block or program. When a RETURN statement is used in a function, the function output shall be setup (a variable having the same name as the function) before the RETURN statement is executed. Otherwise, the function output value shall not be defined.

When a RETURN statement is used in a function block, the programmer should ensure setting of the output variables of the function blocks before the statement is executed. The output variables that have not been setup, shall contain the value corresponding to the initialization value for the corresponding data type or the value set in the preceding function block invocation.

Example 46 RETURN statement

```
FUNCTION FACTORIAL : UDINT
  VAR_INPUT
    code    : USINT;                    // input variable
  END_VAR
  VAR_TEMP
    i       : USINT;                   // auxiliary variable
    tmp     : UDINT := 1;              // auxiliary variable
  END_VAR

  IF code > 13 THEN FACTORIAL := 16#FFFF_FFFF; RETURN; END_IF;
  i := code;
  WHILE i <> 0 DO
    tmp := tmp * USINT_TO_UDINT( i);  i := i - 1;
  END_WHILE;
  FACTORIAL := tmp;
END_FUNCTION
```

For calculation of the factorial for number greater than 13, the result will be greater than the maximum number that can be stored in the variable of UDINT type. This case in example 4.15 is treated by means of the RETURN statement.

5 GRAPHIC LANGUAGES

The IEC 61 131-3 standard defines two graphic languages: LD, Ladder Diagram and FBD, Function Block Diagram. Both are supported by the Mosaic programming environment.

5.1 Mutual graphic language elements

In the same way as text languages do, each POU declaration in a graphical language contains a declaration and execution part. The declaration part is absolutely the same as the one text languages have, the execution part is divided into so called networks. Each network contains the following elements:

- Network identifier
- Network notes
- Network graphics

Network identifiers

Every network can feature an identifier which is a user defined identifier ended by a colon. The identifier can be the target of a jump during branching a high performance POU program. The scope of a network and its identifier is *local* within the frame of the programming unit in which the network was located into. Network identifiers are not necessary.

Every network has an order number in the Mosaic programming environment. This number is generated automatically and serves for better orientation in complicated POUs. After inserting a new network, the networks are automatically re-numbered. The graphic editor enables quick search for networks in the POU according to numbers.

Network notes

Between network identifiers and graphics a network note can be placed. This note can have several lines and can contain characters of national alphabets. Network notes are not necessary.

Network graphics

The network graphics contain graphical elements interconnected via connection lines. A graphical element can e.g. be a open contact, timer block or output coil. Connection lines determine the information flow, e.g. from the timer output to the output coil. Every graphic element can voluntarily feature a note.

Flow direction in networks

Graphic languages are used for representing flows of “intended amounts” through one or more networks representing a control algorithm. These intended amounts can be understood to be:

- *“energy flow”*, analogously to the energy flow in electromechanical relay systems which are standardly used in relay diagrams
- *“signal flow”*, analogously to the signal flow between signal processing systems which are standardly used in function block diagrams

The relevant “intended amount” flows along the lines between network elements according

to the following rules:

- Energy flows in the LD language flow from left to right.
- Signal flows in the FBD language flow from outputs (right side) of function blocks to the inputs (left side) of other connected function blocks

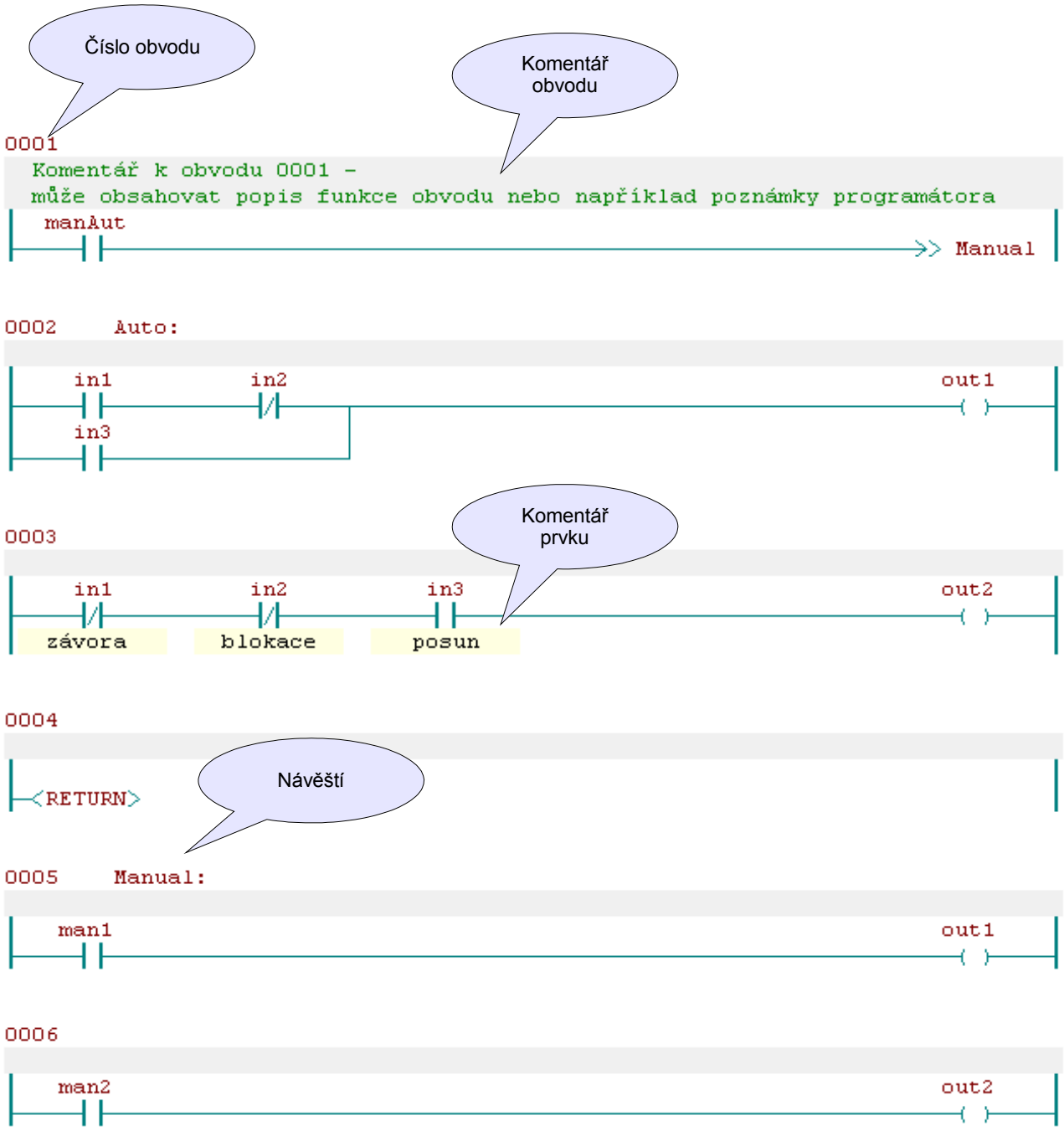


Figure 12 Mutual graphic language elements

5.2 LD ladder diagram language

The ladder diagram language originates from electromechanical relay circuits and is based on graphic representation of relay logics. This language is mainly intended for processing bool signals.

As already mentioned, the execution parts of the POUs in the LD language are made from networks. Networks are, in the LD language, bordered by so called power rails on the left and right side. The logical one (TRUE) “leads” from the left power rail to all graphic elements connected to it, usually switch and brake contacts. Depending on their state, the logical one is let through or not into the following elements connected in the network. The last element on the right is usually the output one and is connected to the right power rail. A coil typically represents the output element.

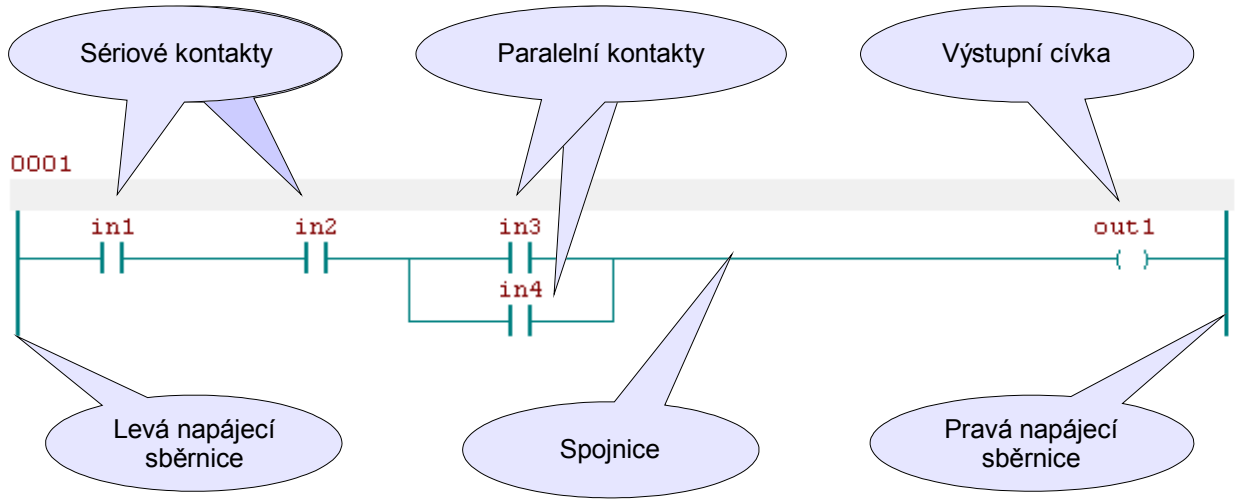


Figure 13 Serial and parallel connection of network elements

5.2.1 LD language graphic elements

The LD language network can contain following graphic elements:

- power rail
- connection lines
- contacts and coils
- graphic elements for controlling program executions (jumps)
- graphic elements for calling functions or function blocks

Graphic elements can be connected in serial or parallel order. Figure 5.2 shows variables **`in1`** and **`in2`** connected in serial (AND) with parallel connected variables **`in3`** and **`in4`** (OR). These variables are called contacts and the program tests their values (reads them). The variable **`out1`** is the name of the coil and the program writes into it.

The network shown on figure 5.2 is executed via the expression **`out1 := in1 AND in2 AND (in3 OR in4);`**

5.2.1.1 Power rail

The network, in the LD language, is bordered from the left by a vertical line called the *left power rail* and from the right by a vertical line called the *right power rail.* The left power rail is always “ON”. The right power rail has no defined state.

5.2.1.2 LD language connection lines

Connection line elements can be vertical or horizontal. The state of a connection line can be “ON” or “OFF”, depending on the bool value 1 or 0. The term *connection line state* is a synonym for the term *energy flow.*

A horizontal connection line is indicated by a horizontal line. A horizontal connection line hands over the state of the element which is adjacent left to the element to the adjacent element on the right from it.

A vertical connection line is indicated by a vertical line intersecting one or more horizontal connection lines on each side. The state of vertical connection lines represents inclusive OR states of ON horizontal connection lines on its left side, i.e. the state of vertical connection lines will be:

- OFF, if the states of all connected horizontal connection lines on its left side are OFF
- ON, if the state of one or more connected connection lines on its left side are ON

The state of the vertical connection lines is copied to all connected horizontal connection lines to the right of it. The state of vertical connection lines is not copied to any connected horizontal connection line to the left of it.

Table 29 FBD language connection lines

Graphic object	Name	Function
	Horizontal connection lines	Horizontal connection lines copy the state of elements connected on the left from it into elements right from it
	Vertical connection line with horizontal connections	The state of the left horizontal connection line is copied to all horizontal connection lines to the right
	Vertical connection line with horizontal OR connections	The state of the right horizontal connection line is the result of the OR logic function of the states of all left horizontal connection lines

5.2.1.3 Contacts and coils

Contacts enable logic operations between the state of a left horizontal connection line and a variable, which is assigned to the contact. The type of logic operation depends on the type of contact. The resulting value is handed over to the right connection line. The contact does not affect the value assigned to the bool variable.

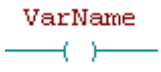
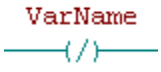
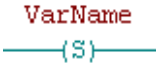
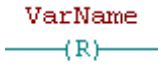
Contacts may be open or closed. An open contact is disconnected in idle mode, the same as an electromechanical contact, (variable value is FALSE) and after switching on power it switches (variable value is TRUE). The function of the closed contact is exactly opposite. It is switched in its idle state (without power), i.e. the tested value is TRUE and after supplying power the contact is opened (tested value FALSE). The function of contacts in the LD language is explained in Table 5.2.

Table 30 LD language contacts

Graphic object	Name	Function
	Open contact	Right power rail := left power rail AND VarName; (Copies the state of the left power rail into the right power rail if the state of the variable **VarName** is **TRUE**, otherwise it writes **FALSE** into the rail)
	Closed contact	Right power rail:= left power rail AND NOT Var-Name; (Copies the state of the left power rail into the right power rail if the state of the variable **VarName** is **FALSE**, otherwise it writes **FALSE** into the rail)

The coil copies the left connection line into the right connection line and saves this state into the assigned bool variable. Coil types and function are listed in Table 5.3.

Table 31 LD language coils

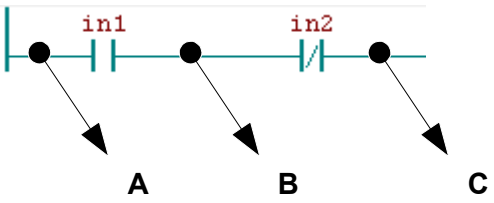
Graphic object	Name	Function
	Coil	Variable := left connection line; (Copies the state of the left connection line into the variable **VarName** as well as to the right connection line)
	Negated coil	Variable:= NOT left connection line; (Copies the negation of the state of the left connection line into the variable **VarName** as well as to the right connection line)
	Set coil	Sets the value **TRUE** into the variable **VarName** if the state of the left connection line is **TRUE**, otherwise it leaves the variable in its original state. The state of the right connection line copies the state of the left connection line.
	Reset coil	Sets the value **FALSE** into the variable **VarName** if the state of the left connection line is **TRUE**, otherwise it leaves the variable in its original state. The state of the right connection line copies the state of the left connection line.

Evaluating energy flows in networks

The energy flows in networks are evaluated from left to right. During program calculations, the individual networks in the POU are evaluated in the sequence top to bottom.

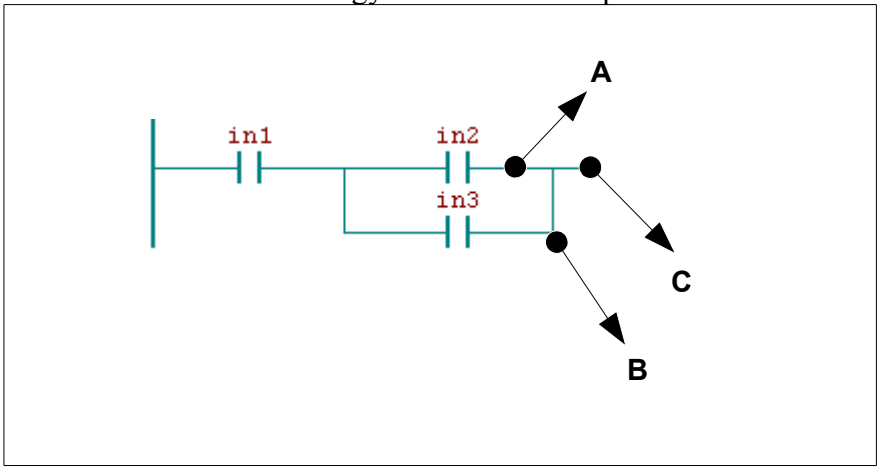
An example of evaluating serial contacts is shown in Table 5.4. The stated network executes the expression **`C := in1 AND NOT in2`**. An example of evaluating parallel contacts is shown in Table 5.5. The stated network executes the expression **`C := in1 AND (in2 OR in3)`**.

Table 32 Energy flow evaluation-serial contacs



in1	in2	NOT in2	A	B	C
0	0	1	1	0	0
0	1	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0

Table 33 Energy flow evaluation-parallel contacs



in1	in2	in3	A	B	C
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

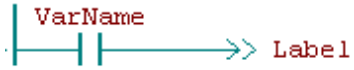
5.2.1.4 LD language program execution controlling

For controlling program execution, we have two possibilities in the LD language: jump to a specific network in a current POU and termination of POU. Graphic symbols are shown Table 5.6.

Jumps are symbolized with a horizontal line ended with a double arrow. Handing over of program on a given identifier is done, if the bool value of the connection line is 1 (TRUE). The connection line for the jump condition can begin by a bool variable, a bool function output or function block or by the left power rail. An unconditioned jump is a special case of conditioned jump. The goal of the jump is the network identifier within the POU, in which the jump shows up. It cannot be jumped outside of one POU.

Conditional returns from functions and function blocks are implemented using the construction RETURN. Program execution is handed back to the calling POU, if the bool input is 1 (TRUE). Program execution will carry on in its standard running, if the bool input is 0. Unconditional return are created on physical ends of functions or function blocks or by the help of RETURN, which is connected to the left power rail.

Table 34 LD language handing over of program control

Graphic object	Name	Function
	Unconditional jump	Jump to a network with a identifier **`Label`**
	Conditional jump	Jump to a network with a identifier **`Label`** if the variable **`VarName`** has a **`TRUE`** value, otherwise the program carries on executing the next network
	Unconditonal Return	Terminates POU and returns control to calling POU. The POU is also terminated if all of its functions are executed
	Conditional return from POU	Terminates POU and returns control to calling POU if the variable **`VarName`** has a **`TRUE`** value, otherwise the program continues executing following networks

5.2.1.5 LD language function and function block calling

The LD language supports function and function block calling. The called POUs are represented in the diagram by a rectangle. Input variables are represented by a connection line from the left, output variables by a connection line from the right. Names of input and output formal parameters are stated inside the rectangles opposite the connection lines, over which current parameter values (variables or constants) are connected. By expandable function (e.g. ADD, XOR, etc.), the names of input parameters are not stated. A function or function block name is stated in the upper part of the rectangle. The function block instance name is stated above the rectangle. Function rectangles are drawn green, function blocks blue.

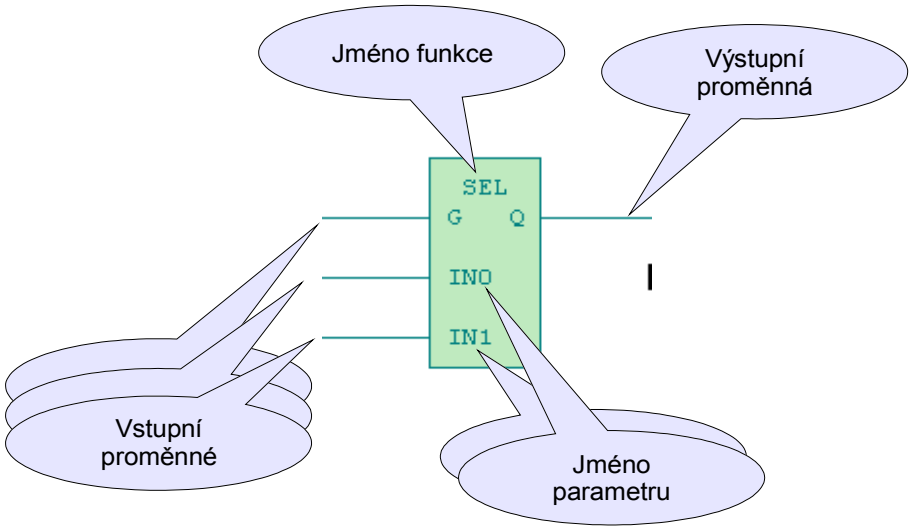
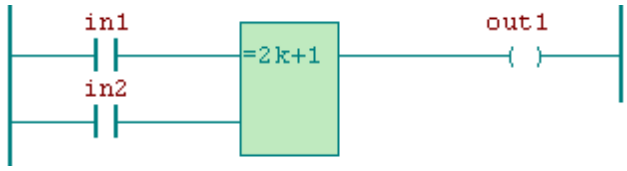
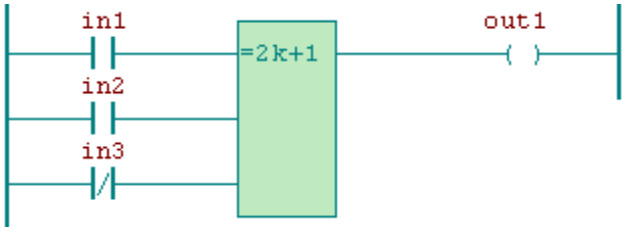
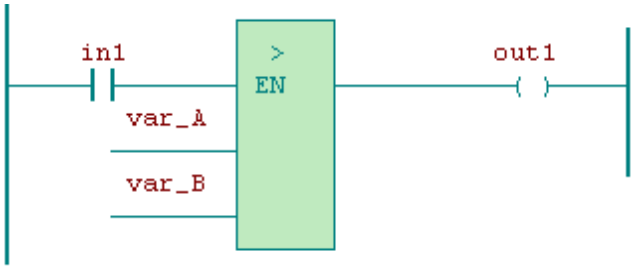
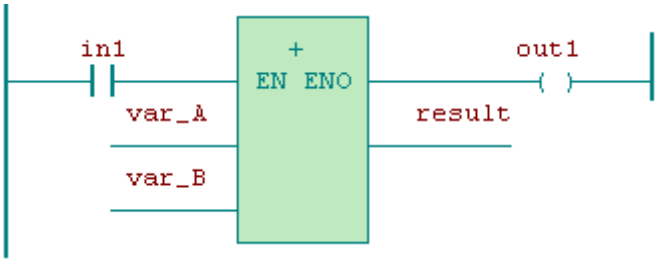


Figure 14 LD language graphic function representation

Function calling

If a function has at least one BOOL type input then this input is connected to the left power rail of the network. If the function has a BOOL type output then this input is connected to the right power rail of the network. Otherwise implicit bool variables EN and ENO are used for connecting function into a network. The EN is a input variable of the BOOL type which conditions the function calling. If a TRUE value is sent to the EN input, the function calling is executed. Otherwise the function will not be called. In every case, the EN input value copies itself into the ENO output function. The connection of the ENO output is not necessary. The use of N / ENO is typical in e.g. arithmetic functions.

Table 35 LD function calling

Network	Description
	Calling of standard function **XOR** Network executes expression `out1 := IN1 XOR in2`
	Calling of standard function **XOR** with expanded numer of inputs Network executes expression `out1 := in1 XOR in2 XOR NOT in3`
	Calling the **GT** function while using the implicit **EN** input. The implicit **ENO** output is not used. If the **EN** input has a **TRUE** value, network executes expression `out1 := var_A > var_B` Otherwise the variable value **out1** is not calculated.
	Calling the **ADD** function while using the implicit **EN** input and implicit **ENO** output. If the **EN** input has a **TRUE** value, network executes expression `result := var_A + var_B` Otherwise the variable value **result** is not calculated. The **ENO** output copies the **EN** input state.

Calling function blocks

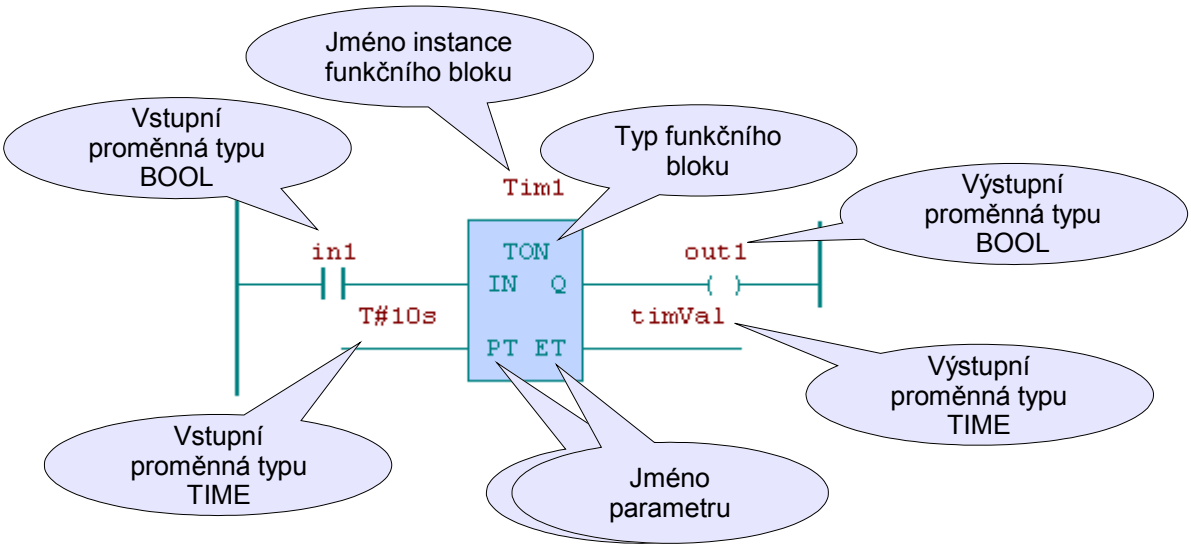
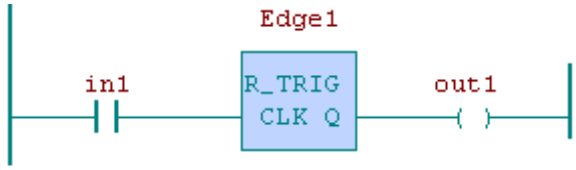
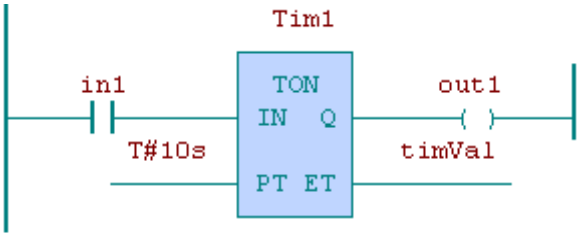
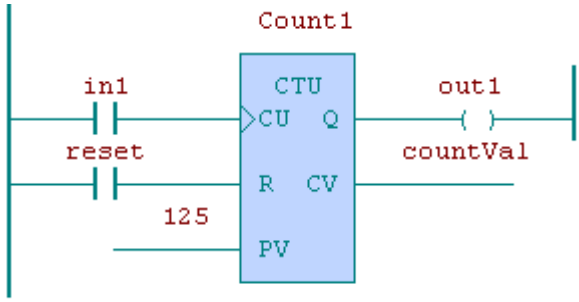
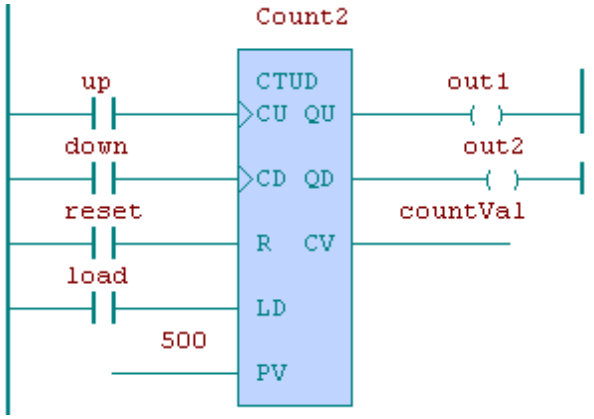


Figure 15 LD language function block calling

When calling a function block in the LD language, similar rules as when calling function apply. For us to be able to connect a function block into a network in the LD language, it has to have some input of the BOOL type (because the signal flow in a LD network starts from the left power rail to which it is possible to connect only BOOL type elements). If a function block does not have any BOOL type input, it is possible to use the implicit input EN (enable) which conditions actions of the function block. All functions and function blocks automatically contain this input which is ensured by the programming environment. The EN input will be available for user defined function blocks, even in the case, when the block definition does not state such a input. The same applies for the implicit ENO (Enable Output) output. The EN is copied into the ENO output in the same why as by functions.

Table 36 LD calling function blocks

Network	Description
	Calling a standard function block **R_TRIG** The output **out1** is set only when the variable **in1** changes from 0 to a value of 1 (rising edge)
	Calling a standard function block **TON** The **PT** input variable (timer preselection) is of the **TIME** type and is not connected to the left power rail. In this case the constant **T#10s** is written into this variable (10 seconds)
	Calling a standard function block **CTU** The **CU** input is defined in the **CTU** function block flowingly: **VAR_INPUT** **CU : BOOL R_EDGE;** **END_VAR** Because of this, the input connection line of this signal is terminated by the sign rising edge evaluation 
	Calling a standard function block **CTUD** **CU** and **CD** inputs are of the **BOOL** type and have rising edge detectors. The **PV** input (Preset Value) is not of the **BOOL** type and thus it is not connected to the power rail. In this case, the constant **500** is written into the input. The **CV** output is also not a **BOOL** type and thus it is not connected to the power rail. Its value is entered into the variable **countVal**.

5.3 FBD language

The Function Block Diagram language is based on connecting function blocks and functions. Functions and function blocks are represented in the FBD in the same way as in the LD, as rectangles. The difference is, that the LD language can transfer only BOOL type values via the connection lines but the FBD language can transfer values of random types between graphic elements.

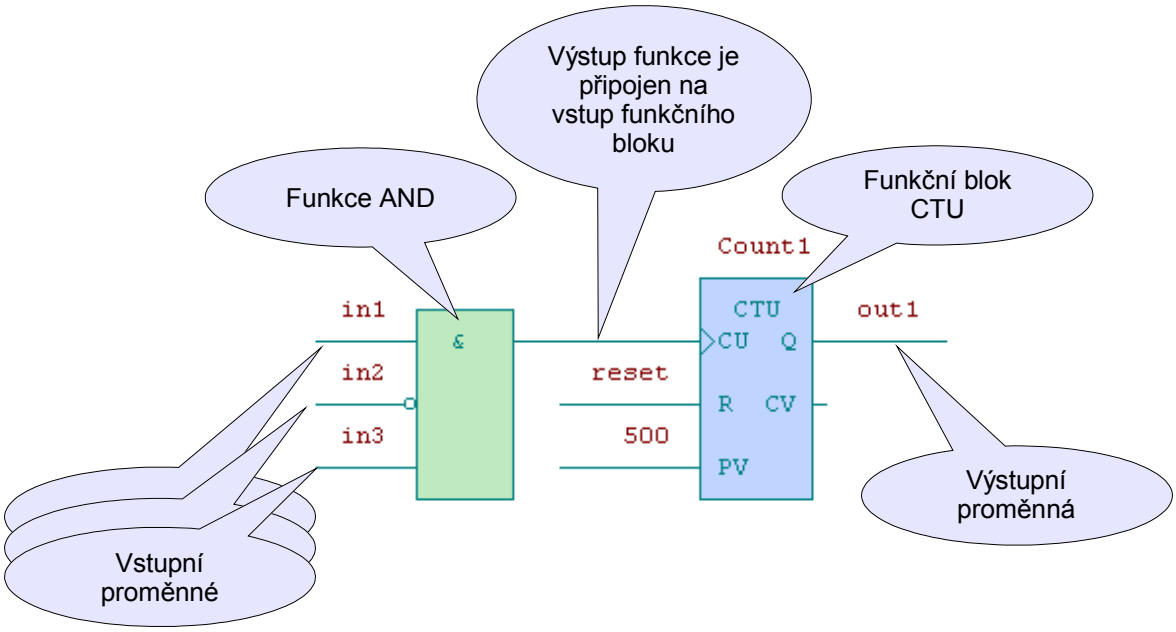


Figure 16 FBD language network graphics

5.3.1 FBD language graphic elements

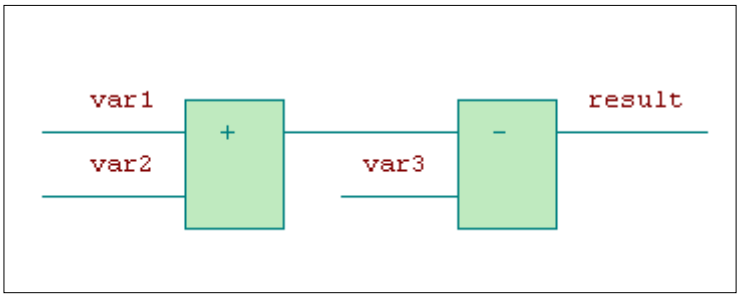
A network in the FBD language may contain the following graphic elements:

- connection lines
- graphic elements for controlling program jumps
- graphic elements for calling functions or function blocks

The FBD language does not contain any other graphic elements as contacts or coils as the LD language. FBD language elements are connected using connection lines of signal flow. Outputs of function blocks are not interconnected. The “wired OR” function is not allowed in the FBD language. The bool OR block is used instead.

The network may be drawn in two ways using the FBD, as seen in Figure 5.6. The manner of display can be whenever changed. The circuit executes the expression **`result := (var1 + var2) - var3`**.

A



B

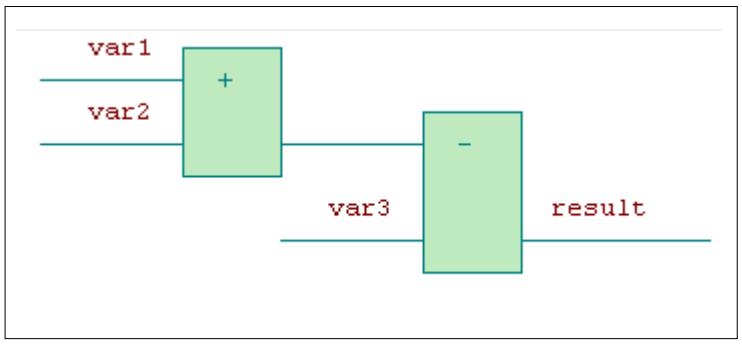
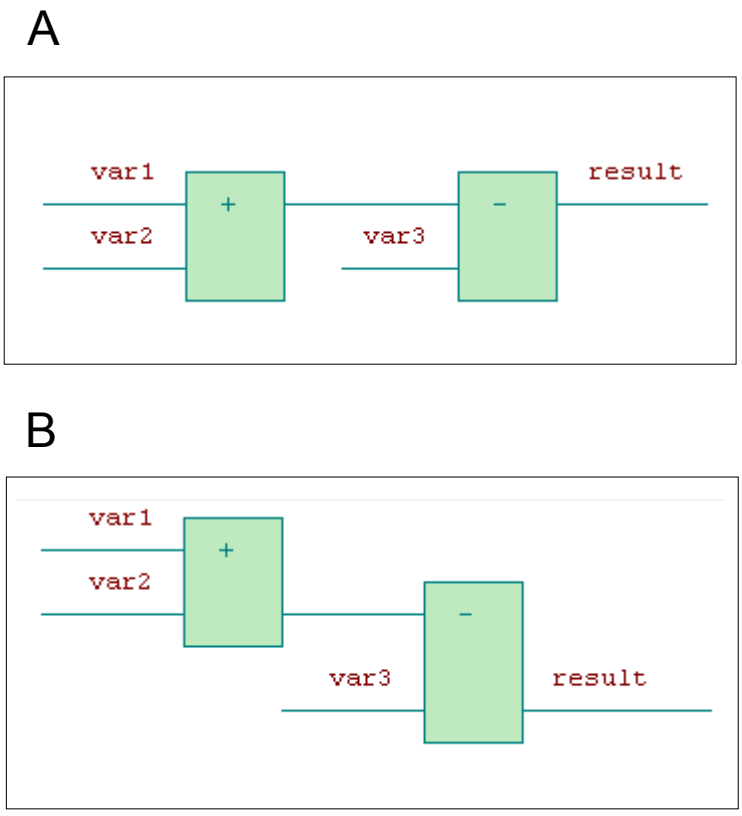


Figure 17 Language FBD manners of network

Evaluating signal flows in networks

The signal flow in circuits is evaluated from left to right. During program calculations, the individual POU networks are evaluated from top to bottom. In the network in Figure 5.6 you can see the sum of variables **`var1`** and **`var2`** and then the deduction of variable **`var3`**. The result will be saved into the variable **`result`**.



5.3.1.1 FBD language connection lines

Connection line elements may be horizontal or vertical. The state of the connections represent the value of the connected variable. The term *connection line state* is a synonym for the term *signal flow.*

A horizontal connection line is indicated by a horizontal line. A horizontal connection line hands over the state of the element which is adjacent left to the element to the adjacent element on the right from it.

The vertical connection line contains vertical lines connecting one or more horizontal connection lines on the right side. The state of the vertical connection lines is copied to all connected

horizontal connection lines to the right of it. The state of vertical connection lines is not copied to any connected horizontal connection line to the left of it.

Table 37 FBD language connection lines

Graphic object	Name	Function
	Horizontal connection lines	Horizontal connection lines copy the state of elements connected on the left from it into elements right from it
	Vertical connection line with horizontal connections	The state of the left horizontal connection line is copied to all horizontal connection lines to the right
	Vertical connection line with horizontal OR connections	This construction (known as wired OR) is not allowed in the FBD language. Instead, the standard OR BOOL function is used.

5.3.1.2 FBD language program execution controlling

For controlling program execution, we have two possibilities, identical to the LD language: jump to a specific network in a current POU and termination of POU. Graphic symbols for the FBD are shown in Table 5.10.

Jumps are symbolized with a horizontal line ended with a double arrow. Handing over of program on a given identifier is done, if the bool value of the connection line is 1 (TRUE). The connection line for the jump condition can begin by a bool variable, a bool function output or function block. If the condition is not stated, then we are talking about a unconditioned jump. The goal of the jump is the network identifier within the POU, in which the jump shows up. It cannot be jumped outside of one POU.

Conditional returns from functions and function blocks are implemented using the construction RETURN. Program execution is handed back to the calling POU, if the bool input is 1 (TRUE). Program execution will carry on in its standard running, if the bool input is 0. Unconditional returns are created on physical ends of functions or function blocks or by the help of the unconditioned element RETURN.

Table.38 FBD language handing over of program control

Graphic object	Name	Function
	Unconditional jump	Jump to a network with a identifier

		**Label**
	Conditional jump	Jump to a network with a identifier **Label** if the variable **VarName** has a **TRUE** value, otherwise the program carries on executing the next network
	Unconditonal Return	Terminates POU and returns control to calling POU. The POU is also terminated if all of its functions are executed
	Conditional return from POU	Terminates POU and returns control to calling POU if the variable **VarName** has a **TRUE** value, otherwise the program continues executing following networks

5.3.1.3 FBD language function and function block calling

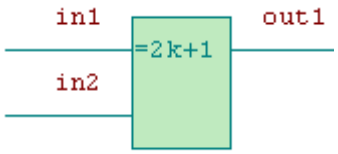
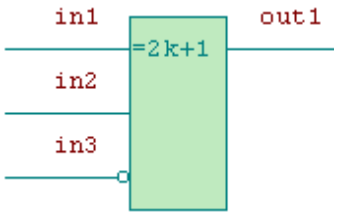
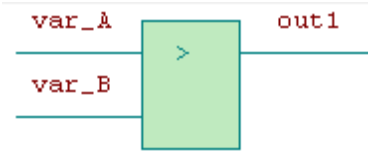
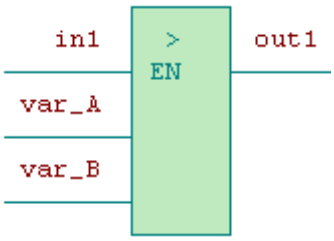
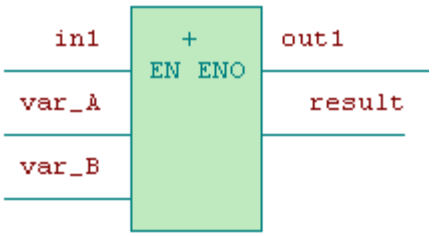
The graphic representation of functions and function blocks is very similar. These POUs are represented in the diagram by a rectangle identically as in the LD language. Input variables are represented by a connection line from the left, output variables by a connection line from the right. Names of input and output formal parameters are stated inside the rectangles opposite the connection lines, over which current parameter values (variables or constants) are connected. By expandable functions (e.g. ADD, XOR, etc.), the names of input parameters are not stated. A function or function block name is stated in the upper part of the rectangle. The function block instance name is stated above the rectangle. Function rectangles are drawn green, function blocks blue.

In the FBD language a function or function block does not have to have any BOOL type input, so that it can be connected to a network. It is thus not necessary to use the implicit EN input, but it is not prohibited. The same applies to the ENO output. If EN and ENO are used, their purpose and behavior is the same as in the LD language.

Function calling

Examples of function calling in the FBD language are shown in Table 5.11.

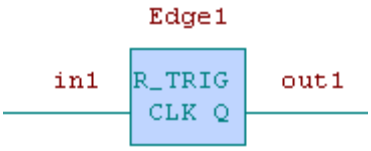
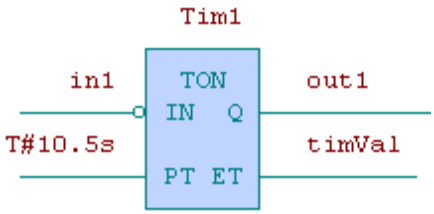
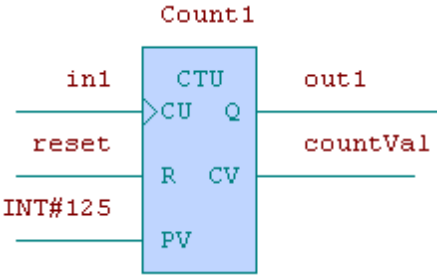
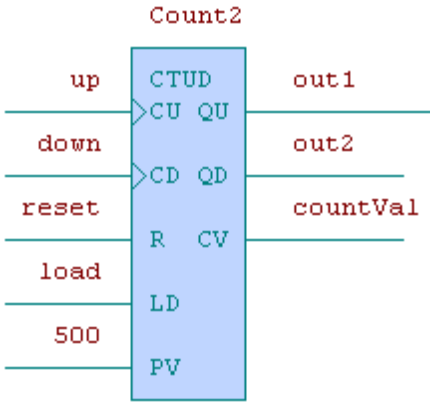
Table.39 FBD language function calling

Network	Description
	Calling of standard function **XOR** Network executes expression **out1 := IN1 XOR in2**
	Calling of standard function **XOR** with expanded numer of inputs Network executes expression **out1 := in1 XOR in2 XOR NOT in3**
	Calling the **GT** function without using **EN** and **ENO** Network executes expression **out1 := var_A > var_B**
	Calling the **GT** function while using the implicit **EN** input. The implicit **ENO** output is not used. If the **EN** input has a **TRUE** value, network executes expression **out1 := var_A > var_B** Otherwise the variable value **out1** is not calculated.
	Calling the **ADD** function while using the implicit **EN** input and implicit **ENO** output. If the **EN** input has a **TRUE** value, network executes expression **result := var_A + var_B** Otherwise the variable value **result** is not calculated. The **ENO** output copies the **EN** input state.

Calling function blocks

Examples of function calling in the FBD language are shown in Table 5.12.

Table.40 FBD language function block calling

Network	Description
	Calling a standard function block **R_TRIG** The output **out1** is set only when the variable **in1** changes from 0 to a value of 1 (rising edge)
	Calling a standard function block **TON** The input variable **in1** is negated. The **PT** input variable (timer preselection) is of the **TIME** type and the constant **T#10.5s** is written into this variable (10,5 seconds)
	Calling a standard function block **CTU** The **CU** input is defined in the **CTU** function block fo lowingly: **VAR_INPUT** **CU : BOOL R_EDGE;** **END_VAR** Because of this, the input connection line of this signal is terminated by the sign rising edge evaluation 
	Calling a standard function block **CTUD** **CU** and **CD** inputs are of the **BOOL** type and have rising edge detectors. The constant **500** is written into the **PV** input (Preset Value). The **CV** output value is entered into the variable **countVal**.

6 APPENDIXES

6.1 Directives

Programs written in one of the text languages may contain directives for the compiler, which will enable to control the compiler's work. Directives are entered into a vinculum.

For example the directive **{$DEFINE new_name}** defines a name "new_name".

6.1.1 PUBLIC directive

The directive **{PUBLIC}** is used to mark public variables. The description of such a variable will be saved into a file with the extension ".pub" during assembly. This file serves for transferring definitions of variables into visualization programs etc.

These directives may be used within the frame of declarations of data type or in the frame of declarations of variables.

Syntax of expression is:

```
TYPE MyINT {PUBLIC} : INT; END_TYPE
VAR
  Var1 {PUBLIC} : BOOL;
  Var2 {PUBLIC} AT %R2000 : BYTE;
END_VAR
```

6.1.2 Directives for conditional program compilation

For conditional program compilation the following directives are used:

```
{$IF <expression>}
{$IFDEF <name>}
{$IFNDEF <name>}
{$DEFINE <name>}
{$UNDEF <name>}
{$END_IF}
{$ELSE}
{$DEFINED(<name>)}
{$ELSEIF <name>}
```

These directives can be used in the declaration and execution part of a program.

6.1.2.1 $IF … $ELSE … $END_IF directives

The **{$IF <expression>}** directives are intended for conditional program assembly if the expression is fulfilled. A branch can be also conditioned using **{$ELSE}**. The conditioned part of the assembled program is finished by using the **{$END_IF}** directive. The expression must contain only variables defined as **VAR_GLOBAL CONSTANT**, constants or **{$DEFINED(<name>)}**. Operators in expression may only be:

'>' - larger
'<' - smaller
'=' - equal
NOT - negation in expression
AND - bool multiplication
OR - bool sum
')' - bracket
'(' - bracket

Syntax of expression is:
{$IF <expression>} .... [{$ELSE}....] {$END_IF}

6.1.2.2 $IFDEF and $IFNDEF directives

These directives are intended for conditioned assemblies. The program following the directive **{$IFDEF <name>}** is assembled providing that the name stated in the directive exists (is defined). On the other hand a program stated behind the directive **{$IFNDEF <name>}** will be assembled only if the name stated in the directive is not defined. These directives can be combined with **{$ELSE}** and **{$ELSEIF}** directives and so create alternatively assembled program parts. The end of conditional program assembly is marked with the **{$END_IF}** directive.

Syntax of expression is:
{$IFDEF <name>} .... [{$ELSE}....] {$END_IF}
{$IFNDEF <name>} .... [{$ELSE}....] {$END_IF}

6.1.2.3 $DEFINE and $UNDEF directives

These directives are intended for adding or removing a name definition. The **{$DEFINE <name>}** directive adds a name **<name>** definition. The name can then be used in the directives **{$IFDEF <name>}** and **{$IFNDEF <name>}**. The **{$UNDEF <name>}** directive cancels the name definition stated in the directive.

Syntax of expression is:
{$DEFINE <name>}
{$UNDEF <name>}

6.1.2.4 DEFINED directive

This directive is used for testing name **<name>** definition validity and it can be used in combination with the **{$IF <expression>}** directive as a part of the expression.

Syntax of expression is:
```
DEFINED (name)
```

Example:
```
{$IF DEFINED( ALFA) OR DEFINED( BETA)}
  VAR counter : INT; END_VAR
{$ELSE}
  VAR counter : DINT; END_VAR
{$END_IF}
```

6.1.3 ASM and END_ASM directives

The **{ASM}** directive is used for inserting a program in mnemocode into a program in one of the IEC languages. The end of the inserted mnemocode is marked with the **{END_ASM}** directive.

Syntax of expression is:
```
{ASM}
{END_ASM}
```

6.1.4 ST_WARNING directive

The **{ST_WARNING}** directive is used for suppressing warnings of ST compiler. The **{ST_WARNING OFF}** directive marks a place in a program from which ST warning messages will be suppressed. The **{ST_WARNING ON}** directive marks a place in a program from which ST warning messages will be again issued.

Syntax of expression is:
```
{ST_WARNING ON}
{ST_WARNING OFF}
```

6.1.5 OFFSET_REG directive

The **{OFFSET_REG=10000}** directive is used to set base addresses in a %R (%M) memory, where variables and instances will be mapped. The first variable will be located on a one level higher address than stated in the directive (%R10001). The directive **{END_OFFSET_REG}** will terminate the shifted variable allocation. Allocation of variables in PLC memory will carry on with addresses 2 levels higher than before the directive **{OFFSET_REG=..}** was used.

Important!
These directives will disable the automatic variable address overlay check. When variables overlay, the compiler will not prompt error!

Syntax of expression is:
{OFFSET_REG=xxx} where xxx is the address %R, where new mapping will occur
{END_OFFSET_REG}

6.2 Reserved keywords

The following table lists keywords which use is reserved by IEC 61 131-3 standard programming languages and they cannot be used for user defined symbols.

Table 41 Reserved keywords

A	ABS ANY ANY_NUM AT	ACOS ANY_BIT ANY_REAL ATAN	ACTION ANY_DATE ARRAY	ADD ANY_INT ASIN
B	BOOL	BY	BYTE	
C	CAL CD CONFIGURATION CTU	CALC CDT CONSTANT CTUD	CALCN CLK COS CU	CASE CONCAT CTD CV
D	D DINT DT	DATE DIV DWORD	DATE_AND_TIME DO	DELETE DS
E	ELSE END_CONFIGURATION END_IF END_STEP END_VAR EQ EXPT	ELSIF END_FOR END_PROGRAM END_STRUCT END_WHILE ET	END_ACTION END_FUNCTION END_REPEAT END_TRANSITION EN EXIT	END_CASE END_FUNCTION_BLOCK END_RESOURCE END_TYPE ENO EXP
F	FALSE FOR	F_EDGE FROM	F_TRIG FUNCTION	FIND FUNCTION_BLOCK
G	GE	GT		
I	IF INT	IN INTERVAL	INITIAL_STEP	INSERT
J	JMP	JMPC	JMPCN	
L	L LEFT LN LWORD	LD LEN LOG	LDN LIMIT LREAL	LE LINT LT
M	MAX MOVE	MID MUL	MIN MUX	MOD
N	N	NE	NEG	NOT
O	OF	ON	OR	ORN
P	P PV	PRIORITY	PROGRAM	PT
Q	Q	QI	QU	QD
R	R READ_WRITE REPLACE RETC ROL R_EDGE	RI REAL RESOURCE RETCN ROR	R_TRIG RELEASE RET RETURN RS	READ_ONLY REPEAT RETAIN RIGHT RTC

S	S SEMA SINGLE SR STRING	ST SHL SINT ST STRUCT	SD SHR SL STEP SUB	SEL SIN SQRT STN
T	TAN TIME_OF_DAY TON TYPE	TASK TO TP	THEN TOD TRANSITION	TIME TOF TRUE
U	UDINT USINT	UINT	ULINT	UNTIL
V	VAR VAR_INPUT	VAR_ACCESS VAR_IN_OUT	VAR_EXTERNAL VAR_OUTPUT	VAR_GLOBAL
W	WHILE	WITH	WORD	
X	XOR	XORN		

CONTENTS

1 INTRODUCTION....3

1.1 The IEC 61 131 standard....3

1.2 Terminology....3

1.3 The basic idea of the IEC 61 131-3 standard....4

1.3.1 Mutual features....4

1.3.2 Programming languages....6

2 BASIC TERMS....8

2.1 Basic program blocks....8

2.2 POU variables declaration....10

2.3 POU executive part....11

2.4 Program example....12

3 COMMON ELEMENTS....14

3.1 Basic elements....14

3.1.1 Identifiers....15

3.1.2 Literals....17

3.1.2.1 Numeric literals....17

3.1.2.2 Character string literals....18

3.1.2.3 Time literals....20

3.2 Date type....21

3.2.1 Elementary data types....21

3.2.2 Generic data types....23

3.2.3 Derived data types....23

3.2.3.1 Simple derived data types....24

3.2.3.2 Derived array data type....25

3.2.3.3 Derived data type Structure....28

3.2.3.4 Combining structures and arrays in derived data types....30

3.2.4 Data type Pointer....31

3.3 Variables....33

3.3.1 Variables declaration....33

3.3.1.1 Variable classes....34

3.3.1.2 Qualifiers in variables declaration....36

3.3.2 Global variables....37

3.3.3 Local variables....38

3.3.4 Input and output variables....39

3.3.5 Simple-element and multi-element variables....41

3.3.5.1 Simple-element variables....41

3.3.5.2 Array....42

3.3.5.3 Structures....43

3.3.6 Location of variables in the PLC memory....44

3.3.7 Variable initialization....46

3.4 Program organization units....48

3.4.1 Function....48

3.4.1.1 Standard functions....50

3.4.2 Function blocks....57

3.4.2.1 Standard function blocks....58

3.4.3 Programs....60

3.5 Configuration elements....61

3.5.1 Configuration....61

3.5.2 Resources....62

3.5.3 Tasks....62

4 Text languages....64

4.1 IL Instruction list language....64
4.1.1 Instructions in IL....64
4.1.2 Operators, modifiers and operands....64
4.1.3 IL language user function definition....67
4.1.4 Calling function in IL language....67
4.1.5 Calling a function block in IL....68

4.2 ST structured text language....70
4.2.1 Expressions....70
4.2.2 Summary of statements in the ST language....72
4.2.2.1 Assignment statement....73
4.2.2.2 Function block call statement....74
4.2.2.3 IF statement....75
4.2.2.4 CASE statement....75
4.2.2.5 FOR statement....76
4.2.2.6 WHILE statement....76
4.2.2.7 REPEAT statement....77
4.2.2.8 EXIT statement....78
4.2.2.9 RETURN statement....78

5 Graphic languages....80

5.1 Mutual graphic language elements....80

5.2 LD ladder diagram language....82
5.2.1 LD language graphic elements....82
5.2.1.1 Power rail....83
5.2.1.2 LD language connection lines....83
5.2.1.3 Contacts and coils....84
5.2.1.4 LD language program execution controlling....86
5.2.1.5 LD language function and function block calling....87

5.3 FBD language....91
5.3.1 FBD language graphic elements....91
5.3.1.1 FBD language connection lines....93
5.3.1.2 FBD language program execution controlling....93
5.3.1.3 FBD language function and function block calling....94

6 APPENDIXES....97

6.1 Directives....97
6.1.1 PUBLIC directive....97
6.1.2 Directives for conditional program compilation....97
6.1.2.1 $IF … $ELSE … $END_IF directives....98
6.1.2.2 $IFDEF and $IFNDEF directives....98
6.1.2.3 $DEFINE and $UNDEF directives....98
6.1.2.4 DEFINED directive....99
6.1.3 ASM and END_ASM directives....99
6.1.4 ST_WARNING directive....99
6.1.5 OFFSET_REG directive....100

6.2 Reserved keywords....101