BS EN 61131-3:2013

BSI Standards Publication

Programmable controllers

Part 3: Programming languages

National foreword

This British Standard is the UK implementation of EN 61131-3:2013. It is identical to IEC 61131-3:2013. It supersedes BS EN 61131-3:2003, which will be withdrawn on 27 March 2016.

The UK participation in its preparation was entrusted by Technical Committee GEL/65, Measurement and control, to Subcommittee GEL/65/2, Elements of systems.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

Published by BSI Standards Limited 2013

ISBN 978 0 580 76605 3

ICS 25.040.40; 35.060; 35.240.50

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 May 2013.

Amendments issued since publication

Date	Text affected

EUROPEAN STANDARD

NORME EUROPÉENNE

EUROPÄISCHE NORM

EN 61131-3

May 2013

ICS 25.040; 35.240.50

Supersedes EN 61131-3:2003

English version

Programmable controllers - Part 3: Programming languages

(IEC 61131-3:2013)

Automates programmables -
Partie 3: Langages de programmation
(CEI 61131-3:2013)

Speicherprogrammierbare Steuerungen -
Teil 3: Programmiersprachen
(IEC 61131-3:2013)

This European Standard was approved by CENELEC on 2013-03-27. CENELEC members are bound to comply with the CEN/CENELEC Internal Regulations which stipulate the conditions for giving this European Standard the status of a national standard without any alteration.

Up-to-date lists and bibliographical references concerning such national standards may be obtained on application to the CEN-CENELEC Management Centre or to any CENELEC member.

This European Standard exists in three official versions (English, French, German). A version in any other language made by translation under the responsibility of a CENELEC member into its own language and notified to the CEN-CENELEC Management Centre has the same status as the official versions.

CENELEC members are the national electrotechnical committees of Austria, Belgium, Bulgaria, Croatia, Cyprus, the Czech Republic, Denmark, Estonia, Finland, Former Yugoslav Republic of Macedonia, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, the Netherlands, Norway, Poland, Portugal, Romania, Slovakia, Slovenia, Spain, Sweden, Switzerland, Turkey and the United Kingdom.

CENELEC
European Committee for Electrotechnical Standardization
Comité Européen de Normalisation Electrotechnique
Europäisches Komitee für Elektrotechnische Normung

Management Centre: Avenue Marnix 17, B - 1000 Brussels

Foreword

The text of document 65B/858/FDIS, future edition 3 of IEC 61131-3, prepared by IEC TC 65 "Industrial-process measurement, control and automation" was submitted to the IEC-CENELEC parallel vote and approved by CENELEC as EN 61131-3:2013.

The following dates are fixed:

• latest date by which the document has to be implemented at national level by publication of an identical national standard or by endorsement	(dop)	2013-12-27
• latest date by which the national standards conflicting with the document have to be withdrawn	(dow)	2016-03-27

This document supersedes EN 61131-3:2003.

EN 61131-3:2013 includes the following significant technical changes with respect to EN 61131-3:2003:

EN 61131-3:2013 is a compatible extension of EN 61131-3:2003. The main extensions are new data types and conversion functions, references, name spaces and the object oriented features of classes abd function blocks. See Annex B.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. CENELEC [and/or CEN] shall not be held responsible for identifying any or all such patent rights.

Endorsement notice

The text of the International Standard IEC 61131-3:2013 was approved by CENELEC as a European Standard without any modification.

In the official version, for Bibliography, the following notes have to be added for the standards indicated:

IEC 60848	NOTE Harmonised as EN 60848.
IEC 61499 series	NOTE Harmonised in EN 61499 series.

Annex ZA
(normative)

Normative references to international publications with their corresponding European publications

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

NOTE When an international publication has been modified by common modifications, indicated by (mod), the relevant EN/HD applies.

Publication	Year	Title	EN/HD	Year
IEC 61131-1	-	Programmable controllers - Part 1: General information	EN 61131-1	-
IEC 61131-5	-	Programmable controllers - Part 5: Communications	EN 61131-5	-
ISO/IEC 10646	2012	Information technology - Universal Coded Character Set (UCS)	-	-
ISO/IEC/IEEE 60559	-	Information technology - Microprocessor Systems - Floating-Point arithmetic	-	-

CONTENTS

PROGRAMMABLE CONTROLLERS –

Part 3: Programming languages

1 Scope

This part of IEC 61131 specifies syntax and semantics of programming languages for programmable controllers as defined in Part 1 of IEC 61131.

The functions of program entry, testing, monitoring, operating system, etc., are specified in Part 1 of IEC 61131.

This part of IEC 61131 specifies the syntax and semantics of a unified suite of programming languages for programmable controllers (PCs). This suite consists of two textual languages, Instruction List (IL) and Structured Text (ST), and two graphical languages, Ladder Diagram (LD) and Function Block Diagram (FBD).

An additional set of graphical and equivalent textual elements named Sequential Function Chart (SFC) is defined for structuring the internal organization of programmable controller programs and function blocks. Also, configuration elements are defined which support the installation of programmable controller programs into programmable controller systems.

In addition, features are defined which facilitate communication among programmable controllers and other components of automated systems.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC 61131-1, *Programmable controllers – Part 1: General information*

IEC 61131-5, *Programmable controllers – Part 5: Communications*

ISO/IEC 10646:2012, *Information technology – Universal Coded Character Set (UCS)*

ISO/IEC/IEEE 60559, *Information technology – Microprocessor Systems – Floating-Point arithmetic*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in IEC 61131-1 and the following apply.

3.1
absolute time
combination of time of day and date information

3.2
access path
association of a symbolic name with a variable for the purpose of open communication

3.3
action
Boolean variable or a collection of operations to be performed, together with an associated control structure

3.4
action block
graphical language element which utilizes a Boolean input variable to determine the value of a Boolean output variable or the enabling condition for an action, according to a predetermined control structure

3.5
aggregate
structured collection of data objects forming a data type

[SOURCE: ISO/AFNOR:1989]

3.6
array
aggregate that consists of data objects, with identical attributes, each of which may be uniquely referenced by subscripting

[SOURCE: ISO/AFNOR:1989]

3.7
assignment
mechanism to give a value to a variable or to an aggregate

[SOURCE: ISO/AFNOR:1989]

3.8
base type
data type, function block type or class from which further types are inherited/derived

3.9
based number
number represented in a specified base other than ten

3.10
binary coded decimal
BCD
encoding for decimal numbers in which each digit is represented by its own binary sequence

3.11
bistable function block
function block with two stable states controlled by one or more inputs

3.12
bit string
data element consisting of one or more bits

3.13
bit string literal
literal that directly represents a bit string value of data type BOOL, BYTE, WORD, DWORD, or LWORD

3.14
body
set of operations of the program organization unit

3.15
call
language construct causing the execution of a function, function block, or method

3.16
character string
aggregate that consists of an ordered sequence of characters

3.17
character string literal
literal that directly represents a character or character string value of data type CHAR, WCHAR, STRING, or WSTRING

3.18
class
program organization unit consisting of:

- the definition of a data structure,
- a set of methods to be performed upon the data structure, and

3.19
comment
language construct for the inclusion of text having no impact on the execution of the program

[SOURCE: ISO/AFNOR:1989]

3.20
configuration
language element corresponding to a programmable controller system

3.21
constant
language element which declares a data element with a fixed value

3.22
counter function block
function block which accumulates a value for the number of changes sensed at one or more specified inputs

3.23
data type
set of values together with a set of permitted operations

[SOURCE: ISO/AFNOR:1989]

3.24
date and time
date within the year and the time of day represented as a single language element

3.25
declaration
mechanism for establishing the definition of a language element

3.26
delimiter
character or combination of characters used to separate program language elements

3.27
derived class
class created by inheritance from another class

Note 1 to entry: Derived class is also named extended class or child class.

3.28
derived data type
data type created by using another data type

3.29
derived function block type
function block type created by inheritance from another function block type

3.30
direct representation
means of representing a variable in a programmable controller program from which an implementation-specified correspondence to a physical or logical location may be determined directly

3.31
double word
data element containing 32 bits

3.32
dynamic binding
situation in which the instance of a method call is retrieved during runtime according to the actual type of an instance or interface

3.33
evaluation
process of establishing a value for an expression or a function, or for the outputs of a network or function block instance, during program execution

3.34
execution control element
language element which controls the flow of program execution

3.35
falling edge
change from 1 to 0 of a Boolean variable

3.36
function
language element which, when executed, typically yields one data element result and possibly additional output variables

3.37
function block instance
instance of a function block type

3.38
function block type
language element consisting of:

- the definition of a data structure partitioned into input, output, and internal variables; and
- a set of operations or a set of methods to be performed upon the elements of the data structure when an instance of the function block type is called

3.39
function block diagram
network in which the nodes are function block instances, graphically represented functions or method calls, variables, literals, and labels

3.40
generic data type
data type which represents more than one type of data

3.41
global variable
variable whose scope is global

3.42
hierarchical addressing
direct representation of a data element as a member of a physical or logical hierarchy

EXAMPLE A point within a module which is contained in a rack, which in turn is contained in a cubicle, etc.

3.43
identifier
combination of letters, numbers, and underscore characters which begins with a letter or underscore and which names a language element

3.44
implementation
product version of a PLC or the programming and debugging tool provided by the Implementer

3.45
Implementer
manufacturer of the PLC or the programming and debugging tool provided to the user to program a PLC application

3.46
inheritance
creation of a new class, function block type or interface based on an existing class, function block type or interface, respectively

3.47
initial value
value assigned to a variable at system start-up

3.48
in-out variable
variable which is used to supply a value to a program organization unit and which is additionally used to return a value from the program organization unit

3.49
input variable
variable which is used to supply a value to a program organization unit except for class

3.50
instance
individual, named copy of the data structure associated with a function block type, class, or program type, which keeps its values from one call of the associated operations to the next

3.51
instance name
identifier associated with a specific instance

3.52
instantiation
creation of an instance

3.53
integer
integer number which may contain positive, null, and negative values

3.54
integer literal
literal which directly represents an integer value

3.55
interface
language element in the context of object oriented programming containing a set of method prototypes

3.56
keyword
lexical unit that characterizes a language element

3.57
label
language construction naming an instruction, network, or group of networks, and including an identifier

3.58
language element
any item identified by a symbol on the left-hand side of a production rule in the formal specification

3.59
literal
lexical unit that directly represents a value

[SOURCE: ISO/AFNOR:1989]

3.60
logical location
location of a hierarchically addressed variable in a schema which may or may not bear any relation to the physical structure of the programmable controller's inputs, outputs, and memory

3.61
long real
real number represented in a long word

3.62
long word
64-bit data element

3.63
method
language element similar to a function that can only be defined in the scope of a function block type and with implicit access to static variables of the function block instance or class instance

3.64
method prototype
language element containing only the signature of a method

3.65
named element
element of a structure which is named by its associated identifier

3.66
network
arrangement of nodes and interconnecting branches

3.67
numeric literal
literal which directly represents a numeric value i.e. an integer literal or real literal

3.68
operation
language element that represents an elementary functionality belonging to a program organization unit or method

3.69
operand
language element on which an operation is performed

3.70
operator
symbol that represents the action to be performed in an operation

3.71
override
keyword used with a method in a derived class or function block type for a method with the same signature as a method of the base class or function block type using a new method body

3.72
output variable
variable which is used to return a value from the program organization unit except for classes

3.73
parameter
variable which is used to provide a value to a program organization unit (as input or in-out parameter) or a variable which is used to return a value from a program organization unit (as output or in-out parameter)

3.74
reference
user-defined data containing the location address to a variable or to an instance of a function block of a specified type

3.75
power flow
symbolic flow of electrical power in a ladder diagram, used to denote the progression of a logic solving algorithm

3.76
pragma
language construct for the inclusion of text in a program organization unit which may affect the preparation of the program for execution

3.77
program
to design, write, and test user programs

3.78
program organization unit
function, function block, class, or program

3.79
real literal
literal directly representing a value of type `REAL` or `LREAL`

3.80
resource
language element corresponding to a “signal processing function” and its “man-machine interface” and “sensor and actuator interface functions”, if any

3.81
result
value which is returned as an outcome of a program organization unit

3.82
return
language construction within a program organization unit designating an end to the execution sequences in the unit

3.83
rising edge
change from 0 to 1 of a Boolean variable

3.84
scope
set of program organization units within which a declaration or label applies

3.85
semantics
relationships between the symbolic elements of a programming language and their meanings, interpretation and use

3.86
semigraphic representation
representation of graphic information by the use of a limited set of characters

3.87
signature
set of information defining unambiguously the identity of the parameter interface of a `METHOD` consisting of its name and the names, types, and order of all its parameters (i.e. inputs, outputs, in-out variables, and result type)

3.88
single-element variable
variable which represents a single data element

3.89
static variable
variable whose value is stored from one call to the next one

3.90
step
situation in which the behavior of a program organization unit with respect to its inputs and outputs follows a set of rules defined by the associated actions of the step

3.91
structured data type
aggregate data type which has been declared using a `STRUCT` or `FUNCTION_BLOCK` declaration

3.92
subscripting
mechanism for referencing an array element by means of an array reference and one or more expressions that, when evaluated, denote the position of the element

3.93
task
execution control element providing for periodic or triggered execution of a group of associated program organization units

3.94
time literal
literal representing data of type `TIME`, `DATE`, `TIME_OF_DAY`, or `DATE_AND_TIME`

3.95
transition
condition whereby control passes from one or more predecessor steps to one or more successor steps along a directed link

3.96
unsigned integer
integer number which may contain positive and null values

3.97
unsigned integer literal
integer literal not containing a leading plus (+) or minus (-) sign

3.98
user-defined data type
data type defined by the user

EXAMPLE Enumeration, array or structure.

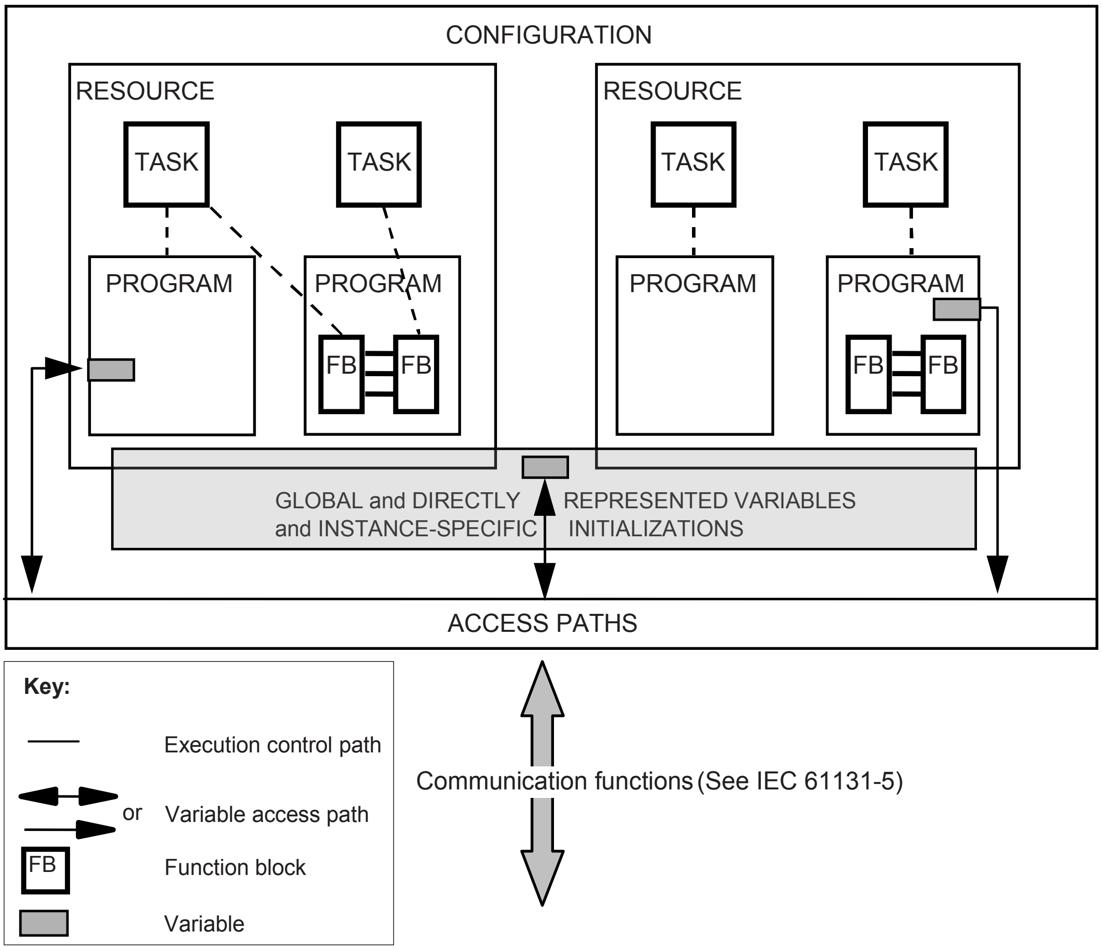
3.99
variable
software entity that may take different values, one at a time

4 Architectural models

4.1 Software model

The basic high-level language elements and their interrelationships are illustrated in Figure 1.

These consist of elements which are programmed using the languages defined in this standard, that is, programs and function block types, classes, functions, and configuration elements, namely, configurations, resources, tasks, global variables, access paths, and instance-specific initializations, which support the installation of programmable controller programs into programmable controller systems.



NOTE 1 Figure 1 is illustrative only. The graphical representation is not normative.

NOTE 2 In a configuration with a single resource, the resource need not be explicitly represented.

Figure 1 – Software model

A configuration is the language element which corresponds to a programmable controller system as defined in IEC 61131-1. A resource corresponds to a “signal processing function” and its “man-machine interface” and “sensor and actuator interface” functions (if any) as defined in IEC 61131-1.

A configuration contains one or more resources, each of which contains one or more programs executed under the control of zero or more tasks.

A program may contain zero or more function block instances or other language elements as defined in this part of IEC 61131.

A task is capable of causing, e.g. on a periodic basis, the execution of a set of programs and function block instances.

Configurations and resources can be started and stopped via the "operator interface", "programming, testing, and monitoring", or "operating system" functions defined in IEC 61131-1. The starting of a configuration shall cause the initialization of its global variables, followed by the starting of all the resources in the configuration. The starting of a resource shall cause the initialization of all the variables in the resource, followed by the enabling of all the tasks in the resource. The stopping of a resource shall cause the disabling of all its tasks, while the stopping of a configuration shall cause the stopping of all its resources.

Mechanisms for the control of tasks are defined in 6.8.2, while mechanisms for the starting and stopping of configurations and resources via communication functions are defined in IEC 61131-5.

Programs, resources, global variables, access paths (and their corresponding access privileges), and configurations can be loaded or deleted by the "communication function" defined in IEC 61131-1. The loading or deletion of a configuration or resource shall be equivalent to the loading or deletion of all the elements it contains.

Access paths and their corresponding access privileges are defined in this standard.

The mapping of the language elements onto communication objects shall be as defined in IEC 61131-5.

4.2 Communication model

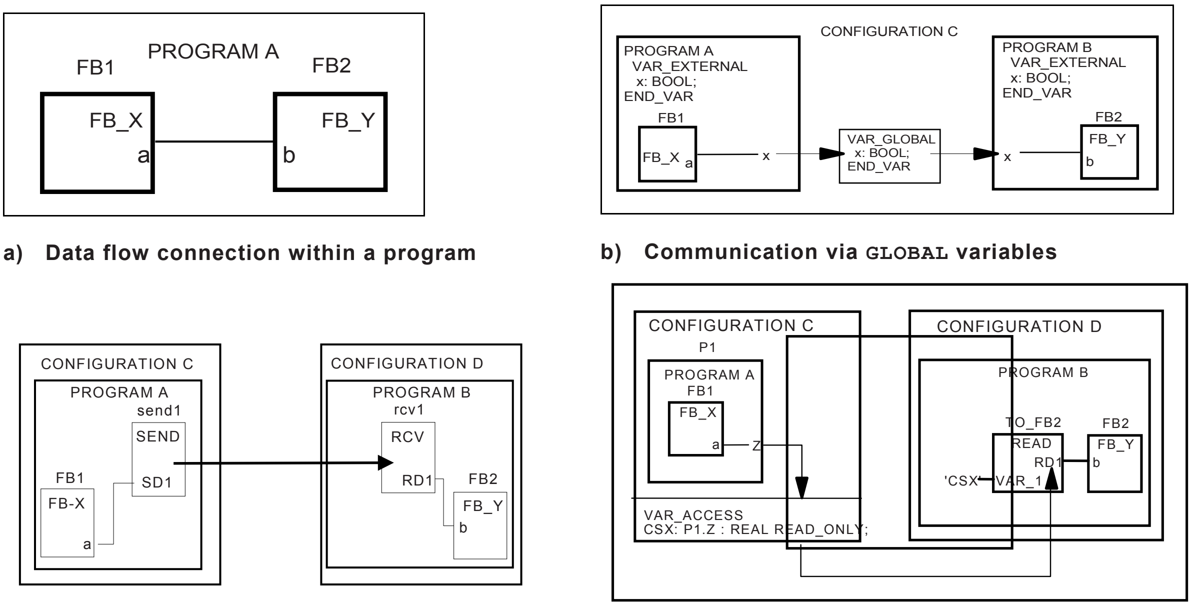
Figure 2 illustrates the ways that values of variables can be communicated among software elements.

As shown in Figure 2a), variable values within a program can be communicated directly by connection of the output of one program element to the input of another. This connection is shown explicitly in graphical languages and implicitly in textual languages.

Variable values can be communicated between programs in the same configuration via global variables such as the variable x illustrated in Figure 2b). These variables shall be declared as `GLOBAL` in the configuration, and as `EXTERNAL` in the programs.

As illustrated in Figure 2c), the values of variables can be communicated between different parts of a program, between programs in the same or different configurations, or between a programmable controller program and a non-programmable controller system, using the communication function blocks defined in IEC 61131-5.

In addition, programmable controllers or non-programmable controller systems can transfer data which is made available by access paths, as illustrated in Figure 2d), using the mechanisms defined in IEC 61131-5.



a) **Data flow connection within a program**

b) **Communication via GLOBAL variables**

c) **Communication function blocks**

d) **Communication via access paths**

NOTE 1 Figure 2 is illustrative only. The graphical representation is not normative.

NOTE 2 In these examples, configurations C and D are each considered to have a single resource.

NOTE 3 The details of the communication function blocks are not shown in Figure 2.

NOTE 4 Access paths can be declared on directly represented variables, global variables, or input, output, or internal variables of programs or function block instances.

NOTE 5 IEC 61131-5 specifies the means by which both PC and non-PC systems can use access paths for reading and writing of variables.

Figure 2 – Communication model

4.3 Programming model

In Figure 3 are the PLC Languages elements summarized. The combination of these elements shall obey the following rules:

1. Data types shall be declared, using the standard data types and any previously defined data types.
2. Functions can be declared using standard or user-defined data types, the standard functions and any previously defined functions.

   This declaration shall use the mechanisms defined for the IL, ST, LD or FBD language.
3. Function block types can be declared using standard and user-defined data types, functions, standard function block types and any previously defined function block types.

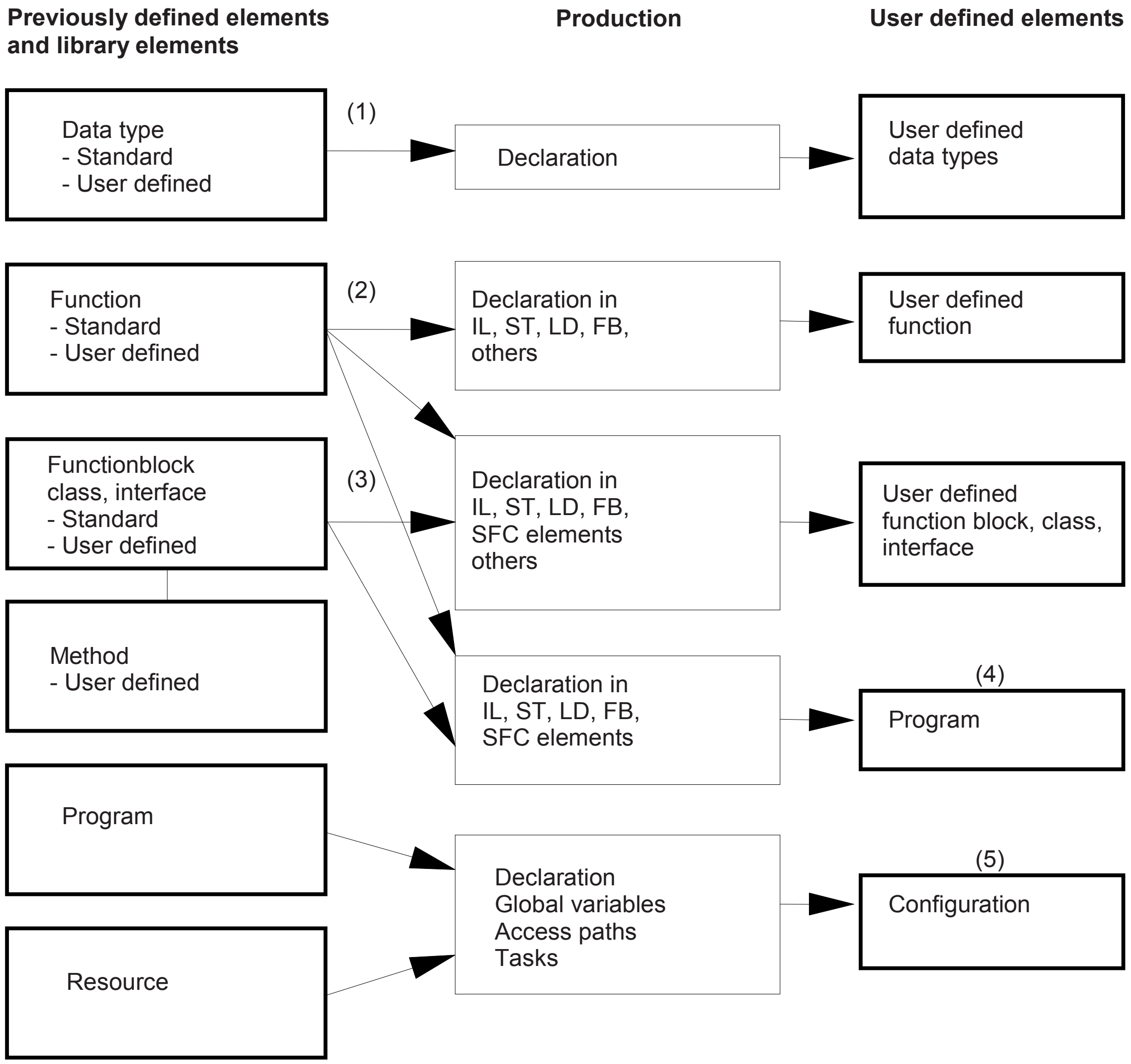
   These declarations shall use the mechanisms defined for the IL, ST, LD, or FBD language, and can include Sequential Function Chart (SFC) elements.

   Optionally, one may define object oriented function block types or classes which use methods and interfaces.
4. A program shall be declared using standard or user-defined data types, functions, function blocks and classes.

   This declaration shall use the mechanisms defined for the IL, ST, LD, or FBD language, and can include Sequential Function Chart (SFC) elements.
5. Programs can be combined into configurations using the elements that is, global variables, resources, tasks, and access paths.

Reference to “previously defined” data types, functions, and function blocks in the above rules is intended to imply that once such a previously defined element has been declared, its definition is available, for example, in a “library” of previously defined elements, for use in further definitions.

A programming language other than one of those defined in this standard may be used for programming of a function, function block type and methods.


Previously defined elements and library elements
Production
User defined elements
Data type
- Standard
- User defined
(1)
Declaration
User defined data types
Function
- Standard
- User defined
(2)
Declaration in IL, ST, LD, FB, others
User defined function
Functionblock class, interface
- Standard
- User defined
(3)
Declaration in IL, ST, LD, FB, SFC elements others
User defined function block, class, interface
Method
- User defined
Declaration in IL, ST, LD, FB, SFC elements
(4)
Program
Program
Declaration Global variables Access paths Tasks
(5)
Configuration
Resource

LD:	Ladder Diagram
FBD:	Function Block Diagram
IL:	Instruction List
ST:	Structured Text
Others:	Other programming languages

NOTE 1 The parenthesized numbers (1) to (5) refer to the corresponding paragraphs 1) through 5) above.

NOTE 2 Data types are used in all productions. For clarity, the corresponding linkages are omitted in this figure.

Figure 3 – Combination of programmable controller language elements

5 Compliance

5.1 General

A PLC programming and debugging tool (PADT), as defined in IEC 61131-1, which claims to comply, wholly or partially, with the requirements of this part of IEC 61131 shall do only as described below.

a) shall provide a subset of the features and provide the corresponding Implementer's compliance statement as defined below.

b) shall not require the inclusion of substitute or additional language elements in order to accomplish any of the features.

c) shall provide a document that specifies all Implementer specific extensions. These are any features accepted by the system that are prohibited or not specified.

d) shall provide a document that specifies all Implementer specific dependencies. This includes the implementation dependencies explicitly designated in this part of IEC 61131 and the limiting parameters like maximum length, number, size and range of value which are not explicitly here.

e) shall provide a document that specifies all errors that are detectable and reported by the implementation. This includes the errors explicitly designated in this part and the errors detectable during preparation of the program for execution and during execution of the program.

NOTE Errors occurring during execution of the program are only partially specified in this part of IEC 61131.

f) shall not use any of the standard names of data types, function or function block names defined in this standard for implementation-defined features whose functionality differs from that described in this part of IEC 61131.

5.2 Feature tables

All tables in this part of IEC 61131 are used for a special purpose in a common way. The first column contains the "feature number", the second column gives the "feature description", the following columns may contain examples or further information. This table structure is used in the Implementer's compliance statement.

5.3 Implementer's compliance statement

The Implementer may define any consistent subset of the features listed in the feature tables and shall declare the provided subset in the "Implementer's compliant statement".

The Implementer's compliance statement shall be included in the documentation accompanying the system, or shall be produced by the system itself.

The format of the Implementer's compliance statement shall provide the following information. Figure 4 shows an example.

- The general information including the Implementer name and address, the product name and version, the controller type and version and the date of issue.
- For each implemented feature the number of the corresponding feature table, the feature number and the applicable programming language.

  Optional is the title and subtitle of the feature table, the feature description, examples, Implementer's note etc.

Not implemented tables and features may be omitted.

IEC 61131-3 "PLC Programming Languages"

Implementer**:** Company name, address, etc.
Product: Product name, version, etc. Controller type specific subset, etc.
.
Date: 2012-05-01

This Product complies with the requirements of the standard for the following language features:

Feature No.	**Table number and title /** **Feature description**	**Compliantly** implemented in the language (✓)				**Implementer's note**
		LD	**FB D**	**ST**	**IL**	
	Table 1 – Character set					
1	ISO/IEC 10646:2012, Information technology – Universal Coded Character Set (UCS)	✓	✓	✓	✓	
2a	Lower case characters a: a, b, c, …	✓	✓	✓		No "ß, ü, ä, ö"
2b	Number sign: # See Table 5	✓				
2c	Dollar sign: $ See Table 6		✓			
	Table 2 – Identifiers					
1	Upper case letters and numbers: `IW215`					
2	Upper and lower case letters, numbers, embedded underscore					
3	Upper and lower case, numbers, leading or embedded underscore					
	Table 3 – Comments					
1	Single-line comment //…					
2a	Multi-line comment (* … *)					
2b	Multi-line comment /* … */					
3a	Nested comment (* ..(* .. *) ..*)					
3b	Nested comment /* .. /* .. */ .. */					
	Table 4 – Pragma					
1	Pragma with curly brackets { … }					
	Table 5 – Numeric literals					
1	Integer literal: `-12`					
2	Real literal: `-12.0`					
3	Real literals with exponent: `-1.34E-12`					
4	Binary literal: `2#1111_1111`					
5	Octal literal: `8#377`					
6	Hexadecimal literal: `16#FF`					
7	Boolean zero and one					
8	Boolean `FALSE` and `TRUE`					
9	Typed literal: `INT#-123`					
	Etc.					

Figure 4 – Implementer's compliance statement (Example)

6 Common elements

6.1 Use of printed characters

6.1.1 Character set

Table 1 shows the character set of the textual languages and textual elements of graphic languages. The characters are represented in terms of the ISO/IEC 10646.

Table 1 – Character set

No.	Description		
1	"ISO/IEC 10646		
2a	Lower case characters [a]:	a, b, c	
2b	Number sign:	#	See Table 5
2c	Dollar sign:	$	See Table 6

[a] When lower-case letters are supported, the case of letters shall not be significant in language elements except within comments as defined in 6.1.5, string literals as defined in 6.3.3, and variables of type STRING and WSTRING as defined in 6.3.3.

6.1.2 Identifiers

An identifier is a string of letters, digits, and underscores which shall begin with a letter or underscore character.

The case of letters shall not be significant in identifiers, for example, the identifiers `abcd`, `ABCD`, and `aBCd` shall be interpreted identically.

The underscore character shall be significant in identifiers, for example, `A_BCD` and `AB_CD` shall be interpreted as different identifiers. Multiple leading or multiple embedded underlines are not allowed; for example, the character sequences `__LIM_SW5 and LIM__SW5` are not valid identifiers. Trailing underscores are not allowed; for example, the character sequence `LIM_SW5_` is not a valid identifier.

At least six characters of uniqueness shall be supported in all systems which support the use of identifiers, for example, `ABCDE1` shall be interpreted as different from `ABCDE2` in all such systems. The maximum number of characters allowed in an identifier is an Implementer specific dependency.

Identifier features and examples are shown in Table 2.

Table 2 – Identifiers

No.	Description	Examples
1	Upper case letters and numbers: `IW215`	`IW215 IW215Z QX75 IDENT`
2	Upper and lower case letters, numbers, embedded underscore	All the above plus: `LIM_SW_5 LimSw5 abcd ab_Cd`
3	Upper and lower case, numbers, leading or embedded underscore	All the above plus: `_MAIN _12V7`

6.1.3 Keywords

Keywords are unique combinations of characters utilized as individual syntactic elements. Keywords shall not contain embedded spaces. The case of characters shall not be significant in keywords; for instance, the keywords FOR and for are syntactically equivalent. They shall not be used for any other purpose, for example, variable names or extensions.

6.1.4 Use of white space

The user shall be allowed to insert one or more characters of “white space” anywhere in the text of programmable controller programs except within keywords, literals, enumerated values, identifiers, directly represented variables or delimiter combinations for example, for comments. “White space” is defined as the SPACE character with encoded value 32 decimal, as well as non-printing characters such as tab, newline, etc. for which no encoding is given in IEC/ISO 10646.

6.1.5 Comments

There are different kinds of user comments listed in Table 3:

1. Single line comments start with the character combination // and end at the next following line feed, new line, form feed (page), or carriage return.

   In single-line comments the special character combinations (* and *) or /* and */ have no special meaning.

2. Multi-line comments shall be delimited at the beginning and end by the special character combinations (* and *), respectively.

   An alternative multi-line comment may be provided using the special character combinations /* and */.

   In multi-line comments the special character combination // has no special meaning.

Comments shall be permitted anywhere in the program where spaces are allowed, except within character string literals.

Comments shall have no syntactic or semantic significance in any of the languages defined in this standard. They are treated like a white space.

Nested comments use corresponding

- pairs of (*, *), e.g. (* ... (* NESTED *)... *) or
- pairs of /*, */, e.g. /* ... /* NESTED */... */.

Table 3 – Comments

No.	Description	Examples

1	Single-line comment with // ...	X:= 13; // comment for one line
		// a single line comments can start at
		// the first character position.

2a	Multi-line comment with (* ... *)	(* comment *) (*************************** A framed comment on three line ****************************)
2b	Multi-line comment with /* ... */	/* comment in one or more lines */
3a	Nested comment with (* .. (* .. *) ..*)	(* (* NESTED *) *)
3b	Nested comment with /* .. /* .. */ .. */	/* /* NESTED */ */

6.2 Pragma

As illustrated in Table 4, pragmas shall be delimited at the beginning and end by curly brackets { and }, respectively. The syntax and semantics of particular pragma constructions are Implementer specific. Pragmas shall be permitted anywhere in the program where spaces are allowed, except within character string literals.

Table 4 – Pragma

No.	Description	Examples

1	Pragma
	with { … }
	curly brackets
		`{VERSION 2.0}`
		`{AUTHOR JHC}`
		`{x:= 256, y:= 384}`

6.3 Literals – External representation of data

6.3.1 General

External representations of data in the various programmable controller programming languages shall consist of numeric literals, character string literals, and time literals.

The need to provide external representations for two distinct types of time-related data is recognized:

- duration data for measuring or controlling the elapsed time of a control event,
- and time of day data which may also include date information for synchronizing the beginning or end of a control event to an absolute time reference.

6.3.2 Numeric literals and string literals

There are two kinds of numeric literals: integer literals and real literals. A numeric literal is defined as a decimal number or a based number. The maximum number of digits for each kind of numeric literal shall be sufficient to express the entire range and precision of values of all the data types which are represented by the literal in a given implementation.

Single underscore characters "_" inserted between the digits of a numeric literal shall not be significant. No other use of underscore characters in numeric literals is allowed.

Decimal literals shall be represented in conventional decimal notation. Real literals shall be distinguished by the presence of a decimal point. An exponent indicates the integer power of ten by which the preceding number is to be multiplied to obtain the value represented. Decimal literals and their exponents can contain a preceding sign "+" or "–".

Literals can also be represented in base 2, 8, or 16. The base shall be in decimal notation. For base 16, an extended set of digits consisting of the letters A through F shall be used, with the conventional significance of decimal 10 through 15, respectively. Based numbers shall not contain a leading sign "+" or "–". They are interpreted as bit string literals.

Numeric literals which represent a positive integer may be used as bit string literals.

Boolean data shall be represented by integer literals with the value zero (0) or one (1), or the keywords FALSE or TRUE, respectively.

Numeric literal features and examples are shown in Table 5.

The data type of a Boolean or numeric literal can be specified by adding a type prefix to the literal, consisting of the name of an elementary data type and the “#” sign. For examples, see feature 9 in Table 5.

Table 5 – Numeric literals

No.	Description	Examples	Explanation
1	Integer literal	-12, 0, 123_4, +986	
2	Real literal	0.0, 0.4560, 3.14159_26	
3	Real literals with exponent	-1.34E-12, -1.34e-12 1.0E+6, 1.0e+6 1.234E6, 1.234e6	
4	Binary literal	 2#1111_1111 2#1110_0000	Base 2 literal 255 decimal 224 decimal
5	Octal literals	 8#377 8#340	Base 8 literal 255 decimal 224 decimal
6	Hexadecimal literal	 16#FF or 16#ff 16#E0 or 16#e0	Base 16 literal 255 decimal 224 decimal
7	Boolean zero and one	0 or 1	
8	Boolean FALSE and TRUE	FALSE TRUE	
9	Typed literal	INT#-123	INT representation of the decimal value -123
		INT#16#7FFF	INT representation of the decimal value 32767
		WORD#16#AFF	WORD representation of the hexadecimal value 0AFF
		WORD#1234	WORD representation of the decimal value 1234=16#4D2
		UINT#16#89AF	UINT representation of the hexadecimal value 89AF
		CHAR#16#41	CHAR representation of the ‘A’
		BOOL#0	
		BOOL#1	
		BOOL#FALSE	
		BOOL#TRUE	

NOTE 1 The keywords FALSE and TRUE correspond to Boolean values of 0 and 1, respectively.

NOTE 2 The feature 5 ‘Octal literals’ is deprecated and may not be included in the next edition of this part of IEC 61131.

6.3.3 Character string literals

Character string literals include single-byte or double-byte encoded characters.

- A single-byte character string literal is a sequence of zero or more characters prefixed and terminated by the single quote character ('). In single-byte character strings, the three-character combination of the dollar sign ($) followed by two hexadecimal digits shall be interpreted as the hexadecimal representation of the eight-bit character code, as shown in feature 1 of Table 6.
- A double-byte character string literal is a sequence of zero or more characters from the ISO/IEC 10646 character set prefixed and terminated by the double quote character ("). In double-byte character strings, the five-character combination of the dollar sign “$” followed by four hexadecimal digits shall be interpreted as the hexadecimal representation of the sixteen-bit character code, as shown in feature 2 of Table 6.

NOTE Relation of ISO/IEC 10646 and Unicode:

Although the character codes and encoding forms are synchronized between Unicode and ISO/IEC 10646, the Unicode Standard imposes additional constraints on implementations to ensure that they treat characters uniformly across platforms and applications. To this end, it supplies an extensive set of functional character specifications, character data, algorithms and substantial background material that is not in ISO/IEC 10646.

Two-character combinations beginning with the dollar sign shall be interpreted as shown in Table 7 when they occur in character strings.

Table 6 – Character string literals

No.	Description	Examples
	Single-byte characters or character strings with ‘ ‘	
1a	Empty string (length zero)	''
1b	String of length one or character CHAR containing a single character	'A'
1c	String of length one or character CHAR containing the “space” character	' '
1d	String of length one or character CHAR containing the “single quote” character	'$''
1e	String of length one or character CHAR containing the “double quote” character	'"'
1f	Support of two character combinations of Table 7	'RL'
1g	Support of a character representation with ‘$’ and two hexadecimal characters	'$0A'
	Double-byte characters or character strings with "" (NOTE)	
2a	Empty string (length zero)	""
2b	String of length one or character WCHAR containing a single character	"A"
2c	String of length one or character WCHAR containing the “space” character	" "
2d	String of length one or character WCHAR containing the “single quote” character	"'"
2e	String of length one or character WCHAR containing the “double quote” character	"$""
2f	Support of two character combinations of Table 7	"RL"
2h	Support of a character representation with ‘$’ and four hexadecimal characters	"$00C4"
	Single-byte typed characters or string literals with #	
3a	Typed string	STRING#'OK'
3b	Typed character	CHAR#'X'
	Double-byte typed string literals with # (NOTE)	
4a	Typed double-byte string (using “double quote” character)	WSTRING#"OK"
4b	Typed double-byte character (using “double quote” character)	WCHAR#"X"
4c	Typed double-byte string (using “single quote” character)	WSTRING#'OK'
4d	Typed double-byte character (using “single quote” character)	WCHAR#'X'

No.	Description	Examples
NOTE If a particular implementation supports feature 4 but not feature 2, the Implementer may specify Implementer specific syntax and semantics for the use of the double-quote character.		

Table 7 – Two-character combinations in character strings

No.	Description	Combinations
1	Dollar sign	$$
2	Single quote	$'
3	Line feed	$L or $l
4	Newline	$N or $n
5	Form feed (page)	$P or $p
6	Carriage return	$R or $r
7	Tabulator	$T or $t
8	Double quote	$"
NOTE 1 The “newline” character provides an implementation-independent means of defining the end of a line of data; for printing, the effect is that of ending a line of data and resuming printing at the beginning of the next line. NOTE 2 The $' combination is only valid inside single quoted string literals. NOTE 3 The $" combination is only valid inside double quoted string literals.		

6.3.4 Duration literal

Duration data shall be delimited on the left by the keyword T#, TIME# or LTIME#. The representation of duration data in terms of days, hours, minutes, seconds, and fraction of a second, or any combination thereof, shall be supported as shown in Table 8. The least significant time unit can be written in real notation without an exponent.

The units of duration literals can be separated by underscore characters.

“Overflow” of the most significant unit of a duration literal is permitted, for example, the notation T#25h_15m is permitted.

Time units, for example, seconds, milliseconds, etc., can be represented in upper- or lower-case letters.

As illustrated in Table 8, both positive and negative values are allowed for durations.

Table 8 – Duration literals

No.	Description	Examples
	Duration abbreviations	
1a	d	Day
1b	h	Hour
1c	m	Minute
1d	s	Second
1e	ms	Millisecond
1f	us (no µ available)	Microsecond
1g	ns	Nanoseconds

No.	Description	Examples
	Duration literals without underscore	
2a	short prefix	T#14ms T#-14ms LT#14.7s T#14.7m T#14.7h t#14.7d t#25h15m lt#5d14h12m18s3.5ms t#12h4m34ms230us400ns
2b	long prefix	TIME#14ms TIME#-14ms time#14.7s
	Duration literals with underscore	
3a	short prefix	t#25h_15m t#5d_14h_12m_18s_3.5ms LTIME#5m_30s_500ms_100.1us
3b	long prefix	TIME#25h_15m ltime#5d_14h_12m_18s_3.5ms LTIME#34s_345ns

6.3.5 Date and time of day literal

Prefix keywords for time of day and date literals shall be as shown in Table 9.

Table 9 – Date and time of day literals

No.	Description	Examples
1a	Date literal (long prefix)	DATE#1984-06-25, date#2010-09-22
1b	Date literal (short prefix)	D#1984-06-25
2a	Long date literal (long prefix)	LDATE#2012-02-29
2b	Long date literal (short prefix)	LD#1984-06-25
3a	Time of day literal (long prefix)	TIME_OF_DAY#15:36:55.36
3b	Time of day literal (short prefix)	TOD#15:36:55.36
4a	Long time of day literal (short prefix)	LTOD#15:36:55.36
4b	Long time of day literal (long prefix)	LTIME_OF_DAY#15:36:55.36
5a	Date and time literal (long prefix)	DATE_AND_TIME#1984-06-25-15:36:55.360227400
5b	Date and time literal (short prefix)	DT#1984-06-25-15:36:55.360_227_400
6a	Long date and time literal (long prefix)	LDATE_AND_TIME#1984-06-25-15:36:55.360_227_400
6b	Long date and time literal (short prefix)	LDT#1984-06-25-15:36:55.360_227_400

6.4 Data types

6.4.1 General

A data type is a classification which defines for literals and variables the possible values, the operations that can be done, and the way the values are stored.

6.4.2 Elementary data types (BOOL, INT, REAL, STRING, etc.)

6.4.2.1 Specification of elementary data types

A set of (pre-defined) elementary data types is specified by this standard.

The elementary data types, keyword for each data type, number of bits per data element, and range of values for each elementary data type shall be as shown in Table 10.

Table 10 – Elementary data types

No.	Description	Keyword	Default initial value	N (bits)[a]
1	Boolean	`BOOL`	`0, FALSE`	1[h]
2	Short integer	`SINT`	`0`	8[c]
3	Integer	`INT`	`0`	16[c]
4	Double integer	`DINT`	`0`	32[c]
5	Long integer	`LINT`	`0`	64[c]
6	Unsigned short integer	`USINT`	`0`	8[d]
7	Unsigned integer	`UINT`	`0`	16[d]
8	Unsigned double integer	`UDINT`	`0`	32[d]
9	Unsigned long integer	`ULINT`	`0`	64[d]
10	Real numbers	`REAL`	`0.0`	32[e]
11	Long reals	`LREAL`	`0.0`	64[f]
12a	Duration	`TIME`	`T#0s`	--[b]
12b	Duration	`LTIME`	`LTIME#0s`	64[m, q]
13a	Date (only)	`DATE`	NOTE	--[b]
13b	Long Date (only)	`LDATE`	`LDATE#1970-01-01`	64[n]
14a	Time of day (only)	`TIME_OF_DAY or TOD`	`TOD#00:00:00`	--[b]
14b	Time of day (only)	`LTIME_OF_DAY or LTOD`	`LTOD#00:00:00`	64[o, q]
15a	Date and time of Day	`DATE_AND_TIME or DT`	NOTE	--[b]
15b	Date and time of Day	`LDATE_AND_TIME or LDT`	`LDT#1970-01-01-00:00:00`	64[p, q]
16a	Variable-length single-byte character string	`STRING`	`''` (empty)	8[i, g, k, l]
16b	Variable-length double-byte character string	`WSTRING`	`""` (empty)	16[i, g, k, l]
17a	Single-byte character	`CHAR`	`'$00'`	8[g, l]
17b	Double-byte character	`WCHAR`	`"$0000"`	16[g, l]
18	Bit string of length 8	`BYTE`	`16#00`	8[j, g]
19	Bit string of length 16	`WORD`	`16#0000`	16[j, g]
20	Bit string of length 32	`DWORD`	`16#0000_0000`	32[j, g]
21	Bit string of length 64	`LWORD`	`16#0000_0000_0000_0000`	64[j, g]

NOTE Implementer specific because of special starting date different than 0001-01-01.

No.	Description	Keyword	Default initial value	N (bits) [a]

[a] Entries in this column shall be interpreted as specified in the table footnotes.

[b] The range of values and precision of representation in these data types is Implementer specific.

[c] The range of values for variables of this data type is from $-(2^{N-1})$ to $(2^{N-1})-1$.

[d] The range of values for variables of this data type is from 0 to $(2^{N})-1$.

[e] The range of values for variables of this data type shall be as defined in IEC 60559 for the basic single width floating-point format. Results of arithmetic instructions with denormalized values, infinity, or not-a-number values are Implementer specific.

[f] The range of values for variables of this data type shall be as defined in IEC 60559 for the basic double width floating-point format. Results of arithmetic instructions with denormalized values, infinity, or not-a-number values are Implementer specific.

[g] A numeric range of values does not apply to this data type.

[h] The possible values of variables of this data type shall be 0 and 1, corresponding to the keywords `FALSE` and `TRUE`, respectively.

[i] The value of `N` indicates the number of bits/character for this data type.

[j] The value of `N` indicates the number of bits in the bit string for this data type.

[k] The maximum allowed length of `STRING` and `WSTRING` variables is Implementer specific.

[l] The character encoding used for `CHAR`, `STRING`, `WCHAR`, and `WSTRING` is ISO/IEC 10646 (see 6.3.3).

[m] The data type `LTIME` is a signed 64-bit integer with unit of nanoseconds.

[n] The data type `LDATE` is a signed 64-bit integer with unit of nanoseconds with starting date 1970-01-01.

[o] The data type `LDT` is a signed 64-bit integer with unit of nanoseconds with starting date 1970-01-01-00:00:00.

[p] The data type `LTOD` is a signed 64-bit integer with unit of nanoseconds with starting time midnight with `TOD#00:00:00`.

[q] The update accuracy of the values of this time format is Implementer specific, i.e. the value is given in nanoseconds, but it may be updated every microsecond or millisecond.

6.4.2.2 Elementary data type strings (`STRING`, `WSTRING`)

The supported maximum length of elements of type `STRING` and `WSTRING` shall be Implementer specific values and define the maximum length of a `STRING` and `WSTRING` which is supported by the programming and debugging tool.

The explicit maximum length is specified by a parenthesized maximum length (which shall not exceed the Implementer specific supported maximum value) in the associated declaration.

Access to single characters of a string using elements of the data type `CHAR` or `WCHAR` shall be supported using square brackets and the position of the character in the string, starting with position 1.

It shall be an error if double byte character strings are accessed using single byte characters or if single byte character strings are accessed using double byte characters.

EXAMPLE 1 STRING, WSTRING and CHAR, WCHAR

a) Declaration

```
VAR
       String1:   STRING[10]:= 'ABCD';
       String2:   STRING[10]:= '';
       aWStrings: ARRAY [0..1] OF WSTRING:= [“1234”, “5678”];
       Char1:    CHAR;
       WChar1:   WCHAR;
END_VAR
```

b) Usage of STRING and CHAR

```
    Char1:= String1[2];            //is equivalent to Char1:= 'B';
    String1[3]:= Char1;            //results in String1:= 'ABBD '
    String1[4]:= 'B';              //results in String1:= 'ABBB'
    String1[1]:= String1[4];       //results in String1:= 'BBBB'
    String2:= String1[2];   (*results in String2:= 'B'
             if implicit conversion CHAR_TO_STRING has been implemented*)
```

c) Usage of WSTRING and WCHAR

```
    WChar1:= aWStrings[1][2];      //is equivalent to WChar1:= '6';
    aWStrings[1][3]:=WChar1;       //results in aWStrings[1]:= "5668"
    aWStrings[1][4]:= "6";  //results in aWStrings[1]:= "5666"
    aWStrings[1][1]:= aWStrings[1][4];    //results in String1:= "6666"
    aWStrings[0]:= aWStrings[1][4];       (* results in aWStrings[0]:= "6";
             if implicit conversion WCHAR_TO_WSTRING has been implemented *)
```

d) Equivalent functions (see 6.6.2.5.11)

```
   Char1:= String1[2];
    is equivalent to
   Char1:= STRING_TO_CHAR(Mid(IN:= String1, L:= 1, P:= 2));
   aWStrings[1][3]:= WChar1;
    is equivalent to
   REPLACE(IN1:= aWStrings[1], IN2:= WChar1, L:= 1, P:=3 );
```

e) Error cases

```
    Char1:= String1[2]; //mixing WCHAR, STRING
    String1[2]:= String2;
       //requires implicit conversion STRING_TO_CHAR which is not allowed
```

NOTE The data types for single characters (CHAR and WCHAR) can only contain one character. Strings can contain several characters; therefore strings may require additional management information which is not needed for single characters.

EXAMPLE 2

If type STR10 is declared by

```
TYPE STR10: STRING[10]:= 'ABCDEF'; END_TYPE
```

then maximum length of STR10 is 10 characters, default initial value is 'ABCDEF', and the initial length of data elements of type STR10 is 6 characters.

6.4.3 Generic data types

In addition to the elementary data types shown in Table 10, the hierarchy of generic data types shown in Figure 5 can be used in the specification of inputs and outputs of standard functions and function blocks. Generic data types are identified by the prefix “ANY”.

The use of generic data types is subject to the following rules:

1. The generic type of a directly derived type shall be the same as the generic type of the elementary type from which it is derived.
2. The generic type of a subrange type shall be ANY_INT.
3. The generic type of all other derived types defined in Table 11 shall be ANY_DERIVED.

The usage of generic data types in user-declared program organization units is beyond the scope of this standard.

Generic data types	Generic data types	Groups of elementary data types
ANY	g)	
ANY_DERIVED		
ANY_ELEMENTARY		
ANY_MAGNITUDE		
ANY_NUM		
ANY_REAL	h)	REAL, LREAL
ANY_INT	ANY_UNSIGNED	USINT, UINT, UDINT, ULINT
	ANY_SIGNED	SINT, INT, DINT, LINT
ANY_DURATION		TIME, LTIME
ANY_BIT		BOOL, BYTE, WORD, DWORD, LWORD
ANY_CHARS		
ANY_STRING		STRING, WSTRING
ANY_CHAR		CHAR, WCHAR
ANY_DATE		DATE_AND_TIME, LDT, DATE, TIME_OF_DAY, LTOD

Figure 5 – Hierarchy of the generic data types

6.4.4 User-defined data types

6.4.4.1 Declaration (`TYPE`)

6.4.4.1.1 General

The purpose of the user-defined data types is to be used in the declaration of other data types and in the variable declarations.

A user-defined type can be used anywhere a base type can be used.

User-defined data types are declared using the `TYPE...END_TYPE` textual construct.

A type declaration consists of

- the name of the type
- a ‘`:`’ (colon)
- the declaration of the type itself as defined in the following clauses.

EXAMPLE Type declaration

```
TYPE
      myDatatype1: <data type declaration with optional initialization>;
END_TYPE
```

6.4.4.1.2 Initialization

User-defined data types can be initialized with user-defined values. This initialization has priority over the default initial value.

The user-defined initialization follows the type declaration and starts with the assignment operator ‘`:=`’ followed by the initial value(s).

Literals (e.g. -123, 1.55, “abc”) or constant expressions (e.g. 12*24) may be used. The initial values used shall be of a compatible type i.e. the same type or a type which can be converted using implicit type conversion.

The rules according to Figure 6 shall apply for the initialization of data types.

Generic Data Type	Initialized by literal	Result
ANY_UNSIGNED	Non-negative integer literal or non-negative constant expression	Non-negative integer value
ANY_SIGNED	Integer literal or constant expression	Integer value
ANY_REAL	Numeric literal or constant expression	Numeric value
ANY_BIT	Unsigned integer literal or unsigned constant expression	Unsigned integer value
ANY_STRING	String literal	String value
ANY_DATE	Date and Time of Day literal	Date and Time of Day value
ANY_DURATION	Duration literal	Duration value

Figure 6 – Initialization by literals and constant expressions (Rules)

Table 11 defines the features of the declaration of user-defined data types and initialization.

Table 11 – Declaration of user-defined data types and initialization

No.	Description	Example	Explanation
1a 1b	Enumerated data types	TYPE ANALOG_SIGNAL_RANGE: (BIPOLAR_10V, UNIPOLAR_10V, UNIPOLAR_1_5V, UNIPOLAR_0_5V, UNIPOLAR_4_20_MA, UNIPOLAR_0_20_MA) := UNIPOLAR_1_5V; END_TYPE	Initialization

2a
2b
	Data types with named values	TYPE
		Colors: DWORD
		(Red := 16#00FF0000,
		Green:= 16#0000FF00,
		Blue := 16#000000FF,
		White:= Red OR Green OR Blue,
		Black:= Red AND Green AND Blue)
		:= Green;
		END_TYPE
			Initialization

3a
3b
	Subrange data types	TYPE
		ANALOG_DATA: INT(-4095 .. 4095):= 0;
		END_TYPE	

4a
4b
	Array data types	TYPE ANALOG_16_INPUT_DATA:
		ARRAY [1..16] OF ANALOG_DATA
		:= [8(-4095), 8(4095)];
		END_TYPE
			ANALOG_DATA see above.
			Initialization

5a
5b
	FB types and classes as array elements	TYPE
		TONs: ARRAY[1..50] OF TON
		:= [50(PT:=T#100ms)];
		END_TYPE
			FB TON as array element
			Initialization

No.	Description	Example	Explanation

6a
6b
	Structured data type	TYPE ANALOG_CHANNEL_CONFIGURATION:
		  STRUCT
		    RANGE: ANALOG_SIGNAL_RANGE;
		    MIN_SCALE: ANALOG_DATA:= -4095;
		    MAX_SCALE: ANALOG_DATA:= 4095;
		  END_STRUCT;
		END_TYPE
			ANALOG_SIGNAL_RANGE see above

7a
7b
	FB types and classes as structure elements	TYPE
		  Cooler: STRUCT
		    Temp: INT;
		    Cooling: TOF:= (PT:=T#100ms);
		END_TYPE
			FB TOF as structure element

8a
8b
	Structured data type with relative addressing AT	TYPE
		  Com1_data: STRUCT
		   head AT %B0: INT;
		   length AT %B2: USINT:= 26;
		   flag1 AT %X3.0: BOOL;
		   end AT %B25: BYTE;
		  END_STRUCT;
		END_TYPE
			Explicit layout without overlapping

9a	Structured data type with relative addressing AT and OVERLAP	TYPE
		  Com2_data: STRUCT OVERLAP
		   head AT %B0: INT;
		   length AT %B2: USINT;
		   flag2 AT %X3.3: BOOL;
		   data1 AT %B5: BYTE;
		   data2 AT %B5: REAL;
		   end AT %B19: BYTE;
		  END_STRUCT;
		END_TYPE
			Explicit layout with overlapping

10a
10b
	Directly represented elements of a structure – partly specified using “ * „	TYPE
		  HW_COMP: STRUCT;
		   IN AT %I*: BOOL;
		   OUT_VAR AT %Q*: WORD:= 200;
		   ITNL_VAR: REAL:= 123.0; // not located
		   END_STRUCT;
		END_TYPE
			Assigns the components of a structure to not yet located inputs and outputs, see NOTE 2

11a
11b
	Directly derived data types	TYPE
		  CNT: UINT;
		  FREQ: REAL:= 50.0;
		  ANALOG_CHANNEL_CONFIG:
		   ANALOG_CHANNEL_CONFIGURATION
		  := (MIN_SCALE:= 0, MAX_SCALE:= 4000);
		END_TYPE
			Initialization
			new initialization

12	Initialization using constant expressions	TYPE
		  PIx2: REAL:= 2 * 3.1416;
		END_TYPE
			Uses a constant expression

The declaration of data type is possible without initialization (feature a) or with (feature b) initialization. If only feature (a) is supported, the data type is initialized with the default initial value. If feature (b) is supported, the data type shall be initialized with the given value or default initial value, if no initial value is given.

Variables with directly represented elements of a structure – partly specified using “ * ” may not be used in the VAR_INPUT or VAR_IN_OUT sections.

6.4.4.2 Enumerated data type

6.4.4.2.1 General

The declaration of an enumerated data type specifies that the value of any data element of that type can only take one of the values given in the associated list of identifiers, as illustrated in Table 11.

The enumeration list defines an ordered set of enumerated values, starting with the first identifier of the list, and ending with the last one.

Different enumerated data types may use the same identifiers for enumerated values. The maximum allowed number of enumerated values is Implementer specific.

To enable unique identification when used in a particular context, enumerated literals may be qualified by a prefix consisting of their associated data type name and the hash sign (number sign) '#', similar to typed literals. Such a prefix shall not be used in an enumeration list.

It is an error if sufficient information is not provided in an enumerated literal to determine its value unambiguously (see example below).

EXAMPLE Enumerated date type

```
TYPE
 Traffic_light:   (Red, Amber, Green);
 Painting_colors: (Red, Yellow, Green, Blue):= Blue;
END_TYPE

VAR
 My_Traffic_light: Traffic_light:= Red;
END_VAR

IF My_Traffic_light = Traffic_light#Amber THEN ...      // OK
IF My_Traffic_light = Traffic_light#Red   THEN ...      // OK
IF My_Traffic_light = Amber THEN ...                    // OK - Amber is unique
IF My_Traffic_light = Red   THEN ...                    // ERROR - Red is not unique
```

6.4.4.2.2 Initialization

The default initial value of an enumerated data type shall be the first identifier in the associated enumeration list.

The user can initialize the data type with a user-defined value out of the list of its enumerated values. This initialization has priority.

As shown in Table 11 for ANALOG_SIGNAL_RANGE, the user-defined default initial value of the enumerated data type is the assigned value UNIPOLAR_1_5V.

The user-defined assignment of the initial value of the data type is a feature in Table 11.

6.4.4.3 Data type with named values

6.4.4.3.1 General

Related to the enumeration data type – where the values of enumerated identifiers are not known by the user – is an enumerated data type with named values. The declaration specifies the data type and assigns the values of the named values, as illustrated in Table 11.

Declaring named values does not limit the use of the value range of variables of these data types; i.e. other constants can be assigned, or can arise through calculations.

To enable unique identification when used in a particular context, named values may be qualified by a prefix consisting of their associated data type name and the hash sign (number sign) '#', similar to typed literals.

Such a prefix shall not be used in a declaration list. It is an error if sufficient information is not provided in an enumerated literal to determine its value unambiguously (see example below).

EXAMPLE Data type with named values

```
TYPE
 Traffic_light:   INT (Red:= 1, Amber := 2, Green:= 3):= Green;
 Painting_colors: INT (Red:= 1, Yellow:= 2, Green:= 3, Blue:= 4):= Blue;
END_TYPE

VAR
 My_Traffic_light: Traffic_light;
END_VAR

My_Traffic_light:= 27;       // Assignment from a constant
My_Traffic_light:= Amber + 1;      // Assignment from an expression
                                   // Note: This is not possible for enumerated values
My_Traffic_light:= Traffic_light#Red + 1;

IF My_Traffic_light   = 123 THEN ...                   // OK
IF My_Traffic_light   = Traffic_light#Amber THEN ...   // OK
IF My_Traffic_light   = Traffic_light#Red   THEN ...   // OK
IF My_Traffic_light   = Amber THEN ...                 // OK because Amber is unique
IF My_Traffic_light   = Red THEN ...                   // Error because Red is not unique
```

6.4.4.3.2 Initialization

The default value for a date type with named values is the first data element in the enumeration list. In the example above for `Traffic_light` this element is Red.

The user can initialize the data type with a user-defined value. The initialization is not restricted to named values, any value from within the range of the base data type may be used. This initialization has priority.

In the example, the user-defined initial value of the enumerated data type for `Traffic_light` is `Green`.

The user-defined assignment of the initial value of the data type is a feature in Table 11.

6.4.4.4 Subrange data type

6.4.4.4.1 General

A subrange declaration specifies that the value of any data element of that type can only take on values between and including the specified upper and lower limits, as illustrated in Table 11.

The limits of a subrange shall be literals or constant expressions.

EXAMPLE

```
TYPE
  ANALOG_DATA: INT(-4095 .. 4095):= 0;
END_TYPE
```

6.4.4.4.2 Initialization

The default initial values for data types with subrange shall be the first (lower) limit of the subrange.

The user can initialize the data type with a user-defined value out of the subrange. This initialization has priority.

For instance, as shown in the example in Table 11, the default initial value of elements of type `ANALOG_DATA` is `-4095`, while with explicit initialization, the default initial value is zero (as declared).

6.4.4.5 Array data type

6.4.4.5.1 General

The declaration of an array data type specifies that a sufficient amount of data storage shall be allocated for each element of that type to store all the data which can be indexed by the specified index subrange(s), as illustrated in Table 11.

An array is a collection of data elements of the same data type. Elementary and user-defined data types, function block types and classes can be used as type of an array element. This collection of data elements is referenced by one or more subscripts enclosed in brackets and separated by commas. It shall be an error if the value of a subscript is outside the range specified in the declaration of the array.

NOTE This error can be detected only at runtime for a computed index.

The maximum number of array subscripts, maximum array size and maximum range of subscript values are Implementer specific.

The limits of the index subrange(s) shall be literals or constant expressions. Arrays with variable length are defined in 6.5.3.

In the ST language a subscript shall be an expression yielding a value corresponding to one of the sub-types of generic type `ANY_INT`.

The form of subscripts in the IL language and the graphic languages defined in Clause 8 is restricted to single-element variables or integer literals.

EXAMPLE

a) Declaration of an array

```
VAR myANALOG_16: ARRAY [1..16] OF ANALOG_DATA
     := [8(-4095), 8(4095)];     // user-defined initial values
END_VAR
```

b) Usage of array variables in the ST language could be:

```
OUTARY[6,SYM]:= INARY[0] + INARY[7] - INARY[i] * %IW62;
```

6.4.4.5.2 Initialization

The default initial value of each array element is the initial value defined for the data type of the array elements.

The user can initialize an array type with a user-defined value. This initialization has priority.

The user-defined initial value of an array is assigned in form of a list which may use parentheses to express repetitions.

During initialization of the array data types, the rightmost subscript of an array shall vary most rapidly with respect to filling the array from the list of initialization values.

EXAMPLE Initialization of an array

```
A: ARRAY [0..5] OF INT:= [2(1, 2, 3)]
```

is equivalent to the initialization sequence `1, 2, 3, 1, 2, 3`.

If the number of initial values given in the initialization list exceeds the number of array entries, the excess (rightmost) initial values shall be ignored. If the number of initial values is less than the number of array entries, the remaining array entries shall be filled with the default initial values for the corresponding data type. In either case, the user shall be warned of this condition during preparation of the program for execution.

The user-defined assignment of the initial value of the data type is a feature in Table 11.

6.4.4.6 Structured data type

6.4.4.6.1 General

The declaration of a structured data type (STRUCT) specifies that this data type shall contain a collection of sub-elements of the specified types which can be accessed by the specified names, as illustrated in Table 11.

An element of a structured data type shall be represented by two or more identifiers or array accesses separated by single periods " . ". The first identifier represents the name of the structured element, and subsequent identifiers represent the sequence of element names to access the particular data element within the data structure. Elementary and user-defined data types, function block types and classes can be used as type of a structure element.

For instance, an element of data type ANALOG_CHANNEL_CONFIGURATION as declared in Table 11 will contain a RANGE sub-element of type ANALOG_SIGNAL_RANGE, a MIN_SCALE sub-element of type ANALOG_DATA, and a MAX_SCALE element of type ANALOG_DATA.

The maximum number of structure elements, the maximum amount of data that can be contained in a structure, and the maximum number of nested levels of structure element addressing are Implementer specific.

Two structured variables are assignment compatible only if they are of the same data type.

EXAMPLE Declaration and usage of a structured data type and structured variable

a) Declaration of a structured data type

```
TYPE
  ANALOG_SIGNAL_RANGE:
    (BIPOLAR_10V,
     UNIPOLAR_10V);

  ANALOG_DATA: INT (-4095 .. 4095);

  ANALOG_CHANNEL_CONFIGURATION:
    STRUCT
      RANGE:     ANALOG_SIGNAL_RANGE;
      MIN_SCALE: ANALOG_DATA;
      MAX_SCALE: ANALOG_DATA;
    END_STRUCT;

END_TYPE
```

b) Declaration of a structured variable

```
VAR
 MODULE_CONFIG:  ANALOG_CHANNEL_CONFIGURATION;
 MODULE_8_CONF:  ARRAY [1..8] OF ANALOG_CHANNEL_CONFIGURATION;
END_VAR
```

c) Usage of structured variables in the ST language:

```
    MODULE_CONFIG.MIN_SCALE:= -2047;
    MODULE_8_CONF[5].RANGE:= BIPOLAR_10V;
```

6.4.4.6.2 Initialization

The default values of the components of a structure are given by their individual data types.

The user can initialize the components of the structure with user-defined values. This initialization has priority.

The user can also initialize a previously defined structure using a list of assignments to the components of the structure. This initialization has a higher priority than the default initialization and the initialization of the components.

EXAMPLE Initialization of a structure

a) Declaration with initialization of a structured data type

```
TYPE
  ANALOG_SIGNAL_RANGE:
       (BIPOLAR_10V,
        UNIPOLAR_10V):= UNIPOLAR_10V;
  ANALOG_DATA: INT (-4095 .. 4095);
  ANALOG_CHANNEL_CONFIGURATION:
    STRUCT
      RANGE:     ANALOG_SIGNAL_RANGE;
      MIN_SCALE: ANALOG_DATA:= -4095;
      MAX_SCALE: ANALOG_DATA:= 4096;
    END_STRUCT;
  ANALOG_8BI_CONFIGURATION:
    ARRAY [1..8] OF ANALOG_CHANNEL_CONFIGURATION
     := [8((RANGE:= BIPOLAR_10V))];
END_TYPE
```

b) Declaration with initialization of a structured variable

```
VAR
 MODULE_CONFIG:  ANALOG_CHANNEL_CONFIGURATION
  := (RANGE:= BIPOLAR_10V, MIN_SCALE:= -1023);
 MODULE_8_SMALL:  ANALOG_8BI_CONFIGURATION
  := [8 ((MIN_SCALE:= -2047, MAX_SCALE:= 2048))];
END_VAR
```

6.4.4.7 Relative location for elements of structured data types (AT)

6.4.4.7.1 General

The locations (addresses) of the elements of a structured type can be defined relative to the beginning of the structure.

In this case the name of each component of this structure shall be followed by the keyword AT and a relative location. The declaration may contain gaps in the memory layout.

The relative location consists of a '%' (percent), the location qualifier and a bit or byte location. A byte location is an unsigned integer literal denoting the byte offset. A bit location consists of a byte offset, followed by a '.' (point), and the bit offset as unsigned integer literal out of the range of 0 to 7. White spaces are not allowed within the relative location.

The components of the structure shall not overlap in their memory layout, except if the keyword OVERLAP has been given in the declaration.

Overlapping of strings is beyond the scope of this standard.

NOTE Counting of bit offsets starts with 0 at the rightmost bit. Counting of byte offsets starts at the beginning of the structure with byte offset 0.

EXAMPLE Relative location and overlapping in a structure

```
TYPE
  Com1_data: STRUCT
   head       AT %B0:       INT;          // at location 0
   length     AT %B2:       USINT:= 26;   // at location 2
   flag1      AT %X3.0:     BOOL;         // at location 3.0
   end        AT %B25:      BYTE;         // at 25, leaving a gap
  END_STRUCT;

  Com2_data: STRUCT OVERLAP
   head       AT %B0:       INT;          // at location 0
   length     AT %B2:       USINT;        // at location 2
   flag2      AT %X3.3:     BOOL;         // at location 3.3
   data1      AT %B5:       BYTE;         // at locations 5, overlapped
   data2      AT %B5:       REAL;         // at locations 5 to 8
   end        AT %B19:      BYTE;         // at 19, leaving a gap
  END_STRUCT;

  Com_data: STRUCT OVERLAP  // C1 and C2 overlap
   C1 at %B0: Com1_data;
   C2 at %B0: Com2_data;

 END_STRUCT;
END_TYPE
```

6.4.4.7.2 Initialization

Overlapped structures cannot be initialized explicitly.

6.4.4.8 Directly represented components of a structure – partly specified using “ * ”

The asterisk notation “*” in Table 11 can be used to denote not yet fully specified locations for directly represented components of a structure.

EXAMPLE Assigning of the components of a structure to not yet located inputs and outputs.

```
 TYPE

    HW_COMP: STRUCT;
     IN       AT %I*: BOOL;
     VAL      AT %I*: DWORD;
     OUT      AT %Q*: BOOL;
     OUT_VAR  AT %Q*: WORD;
     ITNL_VAR: REAL; // not located
     END_STRUCT;

  END_TYPE
```

In the case that a directly represented component of a structure is used in a location assignment in the declaration part of a program, a function block type, or a class, an asterisk “*” shall be used in place of the size prefix and the unsigned integer(s) in the concatenation to indicate that the direct representation is not yet fully specified.

The use of this feature requires that the location of the structured variable so declared shall be fully specified inside the `VAR_CONFIG...END_VAR` construction of the configuration for every instance of the containing type.

Variables of this type shall not be used in a `VAR_INPUT`, `VAR_IN_OUT`, or `VAR_TEMP` section.

It is an error if any of the full specifications in the `VAR_CONFIG...END_VAR` construction is missing for any incomplete address specification expressed by the asterisk notation ”*” in any instance of programs or function block types which contain such incomplete specifications.

6.4.4.9 Directly derived data type

6.4.4.9.1 General

A user-defined data type may be directly derived from an elementary data type or a previously user-defined data type.

This may be used to define new type-specific initial values.

EXAMPLE Directly derived data type

```
TYPE
 myInt1123:     INT:= 123;
 myNewArrayType: ANALOG_16_INPUT_DATA  := [8(-1023), 8(1023)];
 Com3_data:   Com2_data:= (head:= 3, length:=40);

END_TYPE

.R1: REAL:= 1.0;
 R2: R1;
```

6.4.4.9.2 Initialization

The default initial value is the initial value of the data type the new data type is derived from.

The user can initialize the data type with a user-defined value. This initialization has priority.

The user-defined initial value of the elements of structure can be declared in a parenthesized list following the data type identifier. Elements for which initial values are not listed in the initial value list shall have the default initial values declared for those elements in the original data type declaration.

EXAMPLE 1 User-defined data types - usage

Given the declaration of `ANALOG_16_INPUT_DATA` in Table 11

and the declaration `VAR INS: ANALOG_16_INPUT_DATA; END_VAR`

the variables `INS[1]` through `INS[16]` can be used anywhere a variable of type `INT` could be used.

EXAMPLE 2

Similarly, given the definition of `Com_data` in Table 11

and additionally the declaration `VAR telegram: Com_data; END_VAR`

the variable `telegram.length` can be used anywhere a variable of type `USINT` could be used.

EXAMPLE 3

This rule can also be applied recursively:

Given the declarations of `ANALOG_16_ INPUT_CONFIGURATION`, `ANALOG_CHANNEL_CONFIGURATION` and `ANALOG_DATA` in Table 11

and the declaration `VAR CONF: ANALOG_16_INPUT_CONFIGURATION; END_VAR`

the variable `CONF.CHANNEL[2].MIN_SCALE` can be used anywhere that a variable of type `INT` could be used.

6.4.4.10 References

6.4.4.10.1 Reference declaration

A reference is a variable that shall only contain a reference to a variable or to an instance of a function block. A reference may have the value `NULL`, i.e. it refers to nothing.

References shall be declared to a defined data type using the keyword REF_TO and a data type – the reference data type. The reference data type shall already be defined. It may be an elementary data type or a user defined data type.

NOTE References without binding to a data type are beyond the scope of this part of IEC 61131.

EXAMPLE 1

```
TYPE
 myArrayType:       ARRAY[0..999] OF INT;
 myRefArrType:      REF_TO myArrayType;             // Definition of a reference
 myArrOfRefType:    ARRAY [0..12] OF myRefArrType; // Definition of an array of refer-
  ences
END_TYPE

VAR
 myArray1:          myArrayType;
 myRefArr1:         myRefArrType;                   // Declararion of a reference
 myArrOfRef:        myArrOfRefType;                 // Declararion of an array of refer-
  ences
END_VAR
```

The reference shall reference only variables of the given reference data type. References to data types which are directly derived are treated as aliases to references to the base data type. The direct derivation may be applied several times.

EXAMPLE 2

```
TYPE
 myArrType1:  ARRAY[0..999] OF INT;
 myArrType2:  myArrType1;
 myRefType1:  REF_TO myArrType1;
 myRefType2:  REF_TO myArrType2;
END_TYPE
```

myRefType1 and myRefType2 can reference variables of type ARRAY[0..999] OF INT and of the derived data types.

The reference data type of a reference can also be a function block type or a class. A reference of a base type can also refer to instances derived from this base type.

EXAMPLE 3

```
CLASS F1 ...              END_CLASS;
CLASS F2 EXTENDS F1 ...   END_CLASS;

TYPE
  myRefF1:   REF_TO F1;
  myRefF2:   REF_TO F2;
END_TYPE
```

References of type myRefF1 can reference instances of class F1 and F2 and derivations of both. Where references of myRefF2 cannot reference instances of F1, only instances of F2 and derivations of it, because F1 may not support methods and variables of the extended class F2.

6.4.4.10.2 Initialization of references

References can be initialized using the value NULL (default) or the address of an already declared variable, instance of a function block or class.

EXAMPLE

```
FUNCTION_BLOCK F1 ... END_FUNCTION_BLOCK;

VAR
 myInt:     INT;
 myRefInt: REF_TO INT:= REF(myInt);
 myF1:      F1;
 myRefF1:  REF_TO F1:= REF(myF1);
END_VAR
```

6.4.4.10.3 Operations on references

The REF() operator returns a reference to the given variable or instance. The reference data type of the returned reference is the data type of the given variable. Applying the REF() operator to a temporary variable (e.g. variables of any VAR_TEMP section and any variables inside functions) is not permitted.

A reference can be assigned to another reference if the reference data type is equal to the base type or is a base type of the reference data type of the assigned reference.

References can be assigned to parameters of functions, function blocks and methods in a call if the reference data type of the parameter is equal to the base type or is a base type of the reference data type. References shall not be used as in-out variables.

If a reference is assigned to a reference of the same data type, then the latter references the same variable. In this context, a directly derived data type is treated like its base data type.

If a reference is assigned to a reference of the same function block type or of a base function block type, then this reference references the same instance, but is still bound to its function block type; i.e. can only use the variables and methods of its reference data type.

Dereferencing shall be done explicitly.

A reference can be dereferenced using a succeeding '^' (caret).

A dereferenced reference can be used in the same way as using a variable directly.

Dereferencing a NULL reference is an error.

NOTE 1 Possible checks of NULL references can be done at compile time, by the runtime system, or by the application program.

The construct REF() and the dereferencing operator '^' shall be used in the graphical languages in the definition of the operands.

NOTE 2 Reference arithmetic is not recommended and is beyond the scope of this part of IEC 61131.

EXAMPLE 1

```
TYPE
S1: STRUCT
  SC1: INT;
  SC2: REAL;
  END_STRUCT;
A1: ARRAY[1..99] OF INT;
END_TYPE

VAR
 myS1: S1;
 myA1: A1;
 myRefS1: REF_TO S1:= REF(mySl);
 myRefA1: REF_TO A1:= REF(myA1);
 myRefInt: REF_TO INT:= REF(myA1[1]);
END_VAR

myRefS1^.SC1:= myRefA1^[12];     // in this case, equivalent to S1.SC1:= A1[12];
myRefInt:= REF(A1[11]);

S1.SC1:= myRefInt^;              // assigns the value of A1[11] to S1.SC1
```

EXAMPLE 2

Graphical representation of the statements of Example 1

```
                  +----------+
                  |   MOVE   |
------------------|EN     ENO|
   myRefA1^[12]---|IN     OUT|--- myRefS1^.SC1
                  +----------+

                  +----------+                          +----------+
                  |   MOVE   |                          |   MOVE   |
------------------|EN     ENO|--------------------------|EN     ENO|
    REF(A1[11])---|IN     OUT|--- myRefInt  myRefInt^---|IN     OUT|--- S1.SC1
                  +----------+                          +----------+
```

Table 12 defines the features for reference operations.

Table 12 – Reference operations

No	Description	Example
	Declaration	

1	Declaration of a reference type	`TYPE`
		`  myRefType: REF_TO INT;`
		`END_TYPE`

	Assignment and comparison	
2a	Assignment reference to reference	<reference>:= <reference> `myRefType1:= myRefType2;`
2b	Assignment reference to parameter of function, function block and method	`myFB (a:= myRefS1);` The types shall be equal!
2c	Comparison with NULL	`IF myInt = NULL THEN ...`
	Referencing	
3a	`REF`(<variable>) Provides of the typed reference to the variable	`myRefA1:= REF (A1);`

No	Description	Example
3b	`REF`(<function block instance>) Provides the typed reference to the function block or class instance	`myRefFB1:= REF(myFB1)`
	Dereferencing	
4	<reference>^ Provides the content of the variable or the content of the instance to which the reference variable contains the reference	`myInt:= myA1Ref^[12];`

6.5 Variables

6.5.1 Declaration and initialization of variables

6.5.1.1 General

The variables provide a means of identifying data objects whose contents may change, for example, data associated with the inputs, outputs, or memory of the programmable controller.

In contrast to the literals which are the external representations of data, variables may change their value over time.

6.5.1.2 Declaration

Variables are declared inside of one of the variable sections.

A variable can be declared using

- an elementary data type or
- a previously user-defined type or
- a reference type or
- an instantly user-defined type within the variable declaration.

A variable can be

- a single-element variable, i.e. a variable whose type is either
  - an elementary type or
  - a user-defined enumeration or subrange type or
  - a user-defined type whose “parentage”, defined recursively, is traceable to an elementary, enumeration or subrange type.
- a multi-element variable, i.e. a variable which represents an `ARRAY` or a `STRUCT`
- a reference, i.e. a variable that refers to another variable or function block instance.

A variable declaration consists of

- a list of variable names which are declared
- a “:” (colon) and
- a data type with an optional variable-specific initialization.

EXAMPLE

```
TYPE
   myType: ARRAY [1..9] OF INT;    // previously user-defined data type
END_TYPE
```

```
VAR
  myVar1, myVar1a: INT;           // two variables using an elementary type
  myVar2: myType;                 // using a previously user-defined type
  myVar3: ARRAY [1..8] OF REAL;   // using an instantly user-defined type
END_VAR
```

6.5.1.3 Initialization of variables

The default initial value(s) of a variable shall be

1. the default initial value(s) of the underlying elementary data types as defined in Table 10,
2. NULL, if the variable is a reference,
3. the user-defined value(s) of the assigned data type;
   this value is optionally specified by using the assignment operator ":=" in the TYPE declaration defined in Table 11,
4. the user-defined value(s) of the variable;
   this value is optionally specified by using the assignment operator ":=" in the VAR declaration (Table 14).

This user-defined value may be a literal (e.g. -123, 1.55, "abc") or a constant expression (e.g. 12*24).

Initial values cannot be given in VAR_EXTERNAL declarations.

Initial values can also be specified by using the instance-specific initialization feature provided by the VAR_CONFIG...END_VAR construct. Instance-specific initial values always override type-specific initial values.

Table 13 – Declaration of variables

No.	Description	Example	Explanation

1	Variable with elementary data type	VAR
		 MYBIT: BOOL;
		 OKAY: STRING[10];
		 VALVE_POS AT %QW28: INT;
		END_VAR
			Allocates a memory bit to the Boolean variable MYBIT.
			Allocates memory to contain a string with a maximum length of 10 characters.

2	Variable with user-defined data type	VAR
		 myVAR: myType;
		END_VAR
			Declaration of a variable with a user data type.

3	Array	VAR
		 BITS: ARRAY[0..7] OF BOOL;
		 TBT: ARRAY [1..2, 1..3] OF INT;
		 OUTA AT %QW6: ARRAY[0..9] OF INT;
		END_VAR	

4	Reference	VAR
		 myRefInt: REF_TO INT;
		END_VAR
			Declaration of a variable to be a reference

Table 14 – Initialization of variables

No.	Description	Example	Explanation

1	Initialization of a variable with elementary data type	VAR
		MYBIT: BOOL := 1;
		OKAY: STRING[10] := 'OK';
		VALVE_POS AT %QW28: INT:= 100;
		END_VAR
			Allocates a memory bit to the Boolean variable MYBIT with an initial value of 1.
			Allocates memory to contain a string with a maximum length of 10 characters.
			After initialization, the string has a length of 2 and contains the two-byte sequence of characters 'OK' (decimal 79 and 75 respectively), in an order appropriate for printing as a character string.

2	Initialization of a variable with user-defined data type	TYPE myType: ... END_TYPE VAR myVAR: myType:= ... // initialization END_VAR	Declaration of a user data type with or without an initialization. Declaration with a prior initialization of a variable with a user data type.

3	Array	VAR
		BITS: ARRAY[0..7] OF BOOL
		:=[1,1,0,0,0,1,0,0];
		TBT: ARRAY [1..2, 1..3] OF INT
		:= [9,8,3(10),6];
		OUTARY AT %QW6: ARRAY[0..9] OF INT := [10(1)];
		END_VAR
			Allocates 8 memory bits to contain initial values
			BITS[0]:= 1, BITS[1]:= 1,...,
			BITS[6]:= 0, BITS[7]:= 0.
			Allocates a 2-by-3 integer array TBT with initial values
			TBT[1,1]:= 9, TBT[1,2]:= 8,
			TBT[1,3]:= 10, TBT[2,1]:= 10,
			TBT[2,2]:= 10, TBT[2,3]:= 6.

4	Declaration and initialization of constants	VAR CONSTANT
		PI: REAL:= 3.141592;
		PI2: REAL:= 2.0*PI;
		END_VAR
			constant
			symbolic constant PI

5	Initialization using constant expressions	VAR
		PIx2: REAL:= 2.0 * 3.1416;
		END_VAR
			Uses a constant expression

6	Initialization of a reference	VAR
		myRefInt: REF_TO INT
		:= REF(myINT);
		END_VAR
			Initializes myRefInt to refer to the variable myINT.

6.5.2 Variable sections

6.5.2.1 General

Each declaration of a program organization unit (POU), i.e. function block, function and program and additionally the method, starts with zero or more declaration parts which specify the names, types (and, if applicable, the physical or logical location and initialization) of the variables used in the organization unit.

The declaration part of the POU may contain various VAR sections depending on the kind of the POU.

The variables can be declared within the various VAR ... END_VAR textual constructions including qualifiers like RETAIN or PUBLIC, if applicable. The qualifiers for variable sections are summarized in Figure 7.

Keyword	**Variable usage**
VAR sections: depending on the POU type (see for function, function block, program) or method	
VAR	Internal to entity (function, function block, etc.)
VAR_INPUT	Externally supplied, not modifiable within entity
VAR_OUTPUT	Supplied by entity to external entities
VAR_IN_OUT	Supplied by external entities, can be modified within entity and supplied to external entity
VAR_EXTERNAL	Supplied by configuration via VAR_GLOBAL
VAR_GLOBAL	Global variable declaration
VAR_ACCESS	Access path declaration
VAR_TEMP	Temporary storage for variables in function blocks, methods and programs
VAR_CONFIG	Instance-specific initialization and location assignment.
(END_VAR)	Terminates the various VAR sections above.
Qualifiers: may follow the keywords above	
RETAIN	Retentive variables
NON_RETAIN	Non-retentive variables
PROTECTED	Only access from inside the own entity and its derivations (default)
PUBLIC	Access allowed from all entities
PRIVATE	Only access from the own entity
INTERNAL	Only access within the same namespace
CONSTANT [a]	Constant (variable cannot be modified)

NOTE The usage of these keywords is a feature of the program organization unit or configuration element in which they are used.

[a] Function block instances shall not be declared in variable sections with a CONSTANT qualifier.

Figure 7 – Variable declaration keywords (Summary)

- **VAR**

  The variables declared in the VAR ... END_VAR section persist from one call of the program or function block instance to another.

  Within functions the variables declared in this section do not persist from one call of the function to another.

- **VAR_TEMP**

  Within program organization units, variables can be declared in a VAR_TEMP...END_VAR section.

  For functions and methods, the keywords VAR and VAR_TEMP are equivalent.

  These variables are allocated and initialized with a type specific default value at each call, and do not persist between calls.

- **VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT**

  The variables declared in these sections are the formal parameters of functions, function block types, programs, and methods.

- **VAR_GLOBAL and VAR_EXTERNAL**

  Variables declared within a VAR_GLOBAL section can be used within another POU if these are re-declared there within a VAR_EXTERNAL section.

Figure 8 shows the usage of the usage of the VAR_GLOBAL, VAR_EXTERNAL and CONSTANT.

Declaration in the element containing the variable	Declaration in the element using the variable	Allowed?
VAR_GLOBAL X	VAR_EXTERNAL CONSTANT X	Yes
VAR_GLOBAL X	VAR_EXTERNAL X	Yes
VAR_GLOBAL CONSTANT X	VAR_EXTERNAL CONSTANT X	Yes
VAR_GLOBAL CONSTANT X	VAR_EXTERNAL X	No

NOTE The use of the VAR_EXTERNAL section in a contained element may lead to unanticipated behaviors, for instance, when the value of an external variable is modified by another contained element in the same containing element.

Figure 8 – Usage of VAR_GLOBAL, VAR_EXTERNAL and CONSTANT (Rules)

- **VAR_ACCESS**

  Variables declared within a VAR_ACCESS section can be accessed using the access path given in the declaration.

- **VAR_CONFIG**

  The VAR_CONFIG...END_VAR construction provides a means to assign instance specific locations to symbolically represented variables using the asterisk notation "*" or to assign instance specific initial values to symbolically represented variables, or both.

6.5.2.2 Scope of the declarations

The scope (range of validity) of the declarations contained in the declaration part shall be local to the program organization unit in which the declaration part is contained. That is, the declared variables shall not be accessible to other program organization units except by an explicit parameter passing via variables which have been declared as inputs or outputs of those units.

The exception to this rule is the case of variables which have been declared to be global. Such variables are only accessible to a program organization unit via a VAR_EXTERNAL declaration. The type of a variable declared in a VAR_EXTERNAL block shall agree with the type declared in the VAR_GLOBAL block of the associated program, configuration or resource.

It shall be an error if:

- any program organization unit attempts to modify the value of a variable that has been declared with the CONSTANT qualifier or in a VAR_INPUT section;
- a variable declared as VAR_GLOBAL CONSTANT in a configuration element or program organization unit (the "containing element") is used in a VAR_EXTERNAL declaration (without the CONSTANT qualifier) of any element contained within the containing element as illustrated below.

The maximum number of variables allowed in a variable declaration block is Implementer specific.

6.5.3 Variable length ARRAY variables

Variable-length arrays can only be used

- as input, output or in-out variables of functions and methods,
- as in-out variables of function blocks.

The count of array dimensions of actual and formal parameter shall be the same. They are specified using an asterisk as an undefined subrange specification for the index ranges.

Variable-length arrays provide the means for programs, functions, function blocks, and methods to use arrays of different index ranges.

To handle variable-length arrays, the following standard functions shall be provided (Table 15).

Table 15 – Variable-length ARRAY variables

No.	Description	Examples

1	Declaration `using *`
	`ARRAY [*, *, . . . ] OF data type`
		`VAR_IN_OUT`
		`  A: ARRAY [*, *] OF INT;`
		`END_VAR;`

	Standard functions LOWER_BOUND / UPPER_BOUND	

2a	Graphical representation	Get lower bound of an array:
		`          +-------------+`
		`          ! LOWER_BOUND !`
		`ARRAY ----! ARR         !--- ANY_INT`
		`ANY_INT --! DIM         !`
		`          +-------------+`
		Get upper bound of an array:
		`          +-------------+`
		`          ! UPPER_BOUND !`
		`ARRAY ----! ARR         !--- ANY_INT`
		`ANY_INT---! DIM         !`
		`          +-------------+`

2b	Textual representation	Get lower bound of the 2nd dimension of the array A:
		`low2:= LOWER_BOUND (A, 2);`
		Get upper bound of the 2nd dimension of the array A:
		`up2:= UPPER_BOUND (A, 2);`

EXAMPLE 1

```
A1: ARRAY [1..10] OF INT:= [10(1)];

A2: ARRAY [1..20, -2..2] OF INT:= [20(5(1))];
          // according array initialization 6.4.4.5.2


LOWER_BOUND (A1, 1)      →  1
UPPER_BOUND (A1, 1)      → 10
LOWER_BOUND (A2, 1)      →  1
UPPER_BOUND (A2, 1)      → 20
LOWER_BOUND (A2, 2)      → -2
UPPER_BOUND (A2, 2)      →  2
LOWER_BOUND (A2, 0)      → error
LOWER_BOUND (A2, 3)      → error
```

EXAMPLE 2 Array Summation

```
FUNCTION SUM: INT;
VAR_IN_OUT A: ARRAY [*] OF INT; END_VAR;
VAR i, sum2: DINT; END_VAR;

sum2:= 0;
FOR i:= LOWER_BOUND(A,1) TO UPPER_BOUND(A,1)
  sum2:= sum2 + A[i];
END_FOR;
SUM:= sum2;
END_FUNCTION

 // SUM (A1)    → 10
 // SUM (A2[2]) → 5
```

EXAMPLE 3 Matrix multiplication

```
    FUNCTION MATRIX_MUL
    VAR_INPUT
      A: ARRAY [*, *] OF INT;
      B: ARRAY [*, *] OF INT;
    END_VAR;

    VAR_OUTPUT C: ARRAY [*, *] OF INT; END_VAR;
    VAR i, j, k, s: INT; END_VAR;

    FOR i:= LOWER_BOUND(A,1)   TO UPPER_BOUND(A,1)
      FOR j:= LOWER_BOUND(B,2) TO UPPER_BOUND(B,2)
      s:= 0;
          FOR k:= LOWER_BOUND(A,2) TO UPPER_BOUND(A,2)
          s:= s + A[i,k] * B[k,j];
          END_FOR;
      C[i,j]:= s;
      END_FOR;
    END_FOR;
    END_FUNCTION

    // Usage:
    VAR
      A: ARRAY [1..5, 1..3] OF INT;
      B: ARRAY [1..3, 1..4] OF INT;
      C: ARRAY [1..5, 1..4] OF INT;
    END_VAR

MATRIX_MUL (A, B, C);
```

6.5.4 Constant variables

Constant variables are variables which are defined inside a variable section which contains the keyword CONSTANT. The rules defined for expressions shall apply.

EXAMPLE Constant variables

```
VAR CONSTANT
  Pi:      REAL:= 3.141592;
  TwoPi:   REAL:= 2.0*Pi;
END_VAR
```

6.5.5 Directly represented variables (%)

6.5.5.1 General

Direct representation of a single-element variable shall be provided by a special symbol formed by the concatenation of

- a percent sign "%"and
- location prefixes I, Q or M and
- a size prefix X (or none), B, W, D, or L and
- one or more (see below hierarchical addressing) unsigned integers that shall be separated by periods ".".

EXAMPLE

```
%MW1.7.9
%ID12.6
%QL20
```

The Implementer shall specify the correspondence between the direct representation of a variable and the physical or logical location of the addressed item in memory, input or output.

NOTE The use of directly represented variables in the bodies of functions, function block types, methods, and program types limits the reusability of these program organization unit types, for example between programmable controller systems in which physical inputs and outputs are used for different purposes.

The use of directly represented variables is permitted in the body of functions, function blocks, programs, methods, and in configurations and resource.

Table 16 defines the features for directly represented variables.

The use of directly represented variables in the body of POUs and methods is deprecated functionality.

Table 16 – Directly represented variables

No.	Description		Example	Explanation
	Location (NOTE 1)			
1	Input location	I	%IW215	Input word 215
2	Output location	Q	%QB7	Output byte 7
3	Memory location	M	%MD48	Double word at memory loc. 48
	Size			
4a	Single bit size	X	%IX1	Input data type BOOL
4b	Single bit size	None	%I1	Input data type BOOL
5	Byte (8 bits) size	B	%IB2	Input data type BYTE
6	Word (16 bits) size	W	%IW3	Input data type WORD
7	Double word (32 bits) size	D	%ID4	Input data type DWORD
8	Long (quad) word (64 bits) size	L	%IL5	Input data type LWORD

No.	Description	Example	Explanation
	Addressing		
9	Simple addressing `%IX1`	`%IB0`	1 level
10	Hierarchical addressing using "." `%QX7.5`	`%QX7.5` `%MW1.7.9`	Implementer defined e.g.: 2 levels, ranges 0..7 3 levels, ranges 1..16
11	Partly specified variables using asterisk "*" (NOTE 2)	`%M*`	

NOTE 1 National standardization organizations can publish tables of translations of these prefixes.

NOTE 2 The use of an asterisk in this table needs the feature `VAR_CONFIG` and vice versa.

6.5.5.2 Directly represented variables – hierarchical addressing

When the simple (1 level) direct representation is extended with additional integer fields separated by periods, it shall be interpreted as a hierarchical physical or logical address with the leftmost field representing the highest level of the hierarchy, with successively lower levels appearing to the right.

EXAMPLE Hierarchical address

```
%IW2.5.7.1
```

For instance, this variable represents the first "channel" (word) of the seventh "module" in the fifth "rack" of the second "I/O bus" of a programmable controller system. The maximum number of levels of hierarchical addressing is Implementer specific.

The use of the hierarchical addressing to permit a program in one programmable controller system to access data in another programmable controller shall be considered as a Implementer specific extension of the language.

6.5.5.3 Declaration of directly represented variables (AT)

Declaration of the directly represented variables as defined in Table 16 (e.g. `%IW6`) can be given a symbolic name and a data type by using the `AT` keyword.

Variables with user-defined data types e.g. an array can be assigned an "absolute" memory by using `AT`. The location of the variable defines the start address of the memory location and does not need to be of equal or bigger size than the given direct representation, i.e. empty memory or overlapping is permitted.

EXAMPLE Usage of direct representation.

```
VAR                                            Name and type for an input
 INP_0 AT %I0.0: BOOL;
 AT %IB12: REAL;
 PA_VAR AT %IB200: PA_VALUE;                   Name and user-defined type for an input location be-
                                               ginning at %IB200
 OUTARY AT %QW6: ARRAY[0..9] OF INT;           Array of 10 integers to be allocated to contiguous out-
END_VAR                                        put locations starting at %QW6
```

For all kinds of variables defined in Table 13, an explicit (user-defined) memory allocation can be declared using the keyword `AT` in combination with the directly represented variables (e.g. `%MW10`).

If this feature is not supported in one or more variable declarations, then it should be stated in the Implementer's compliance statement.

NOTE Initialization of system inputs (e.g. %IW10) is Implementer specific.

6.5.5.4 Directly represented variables – partly specified using “ * ”

The asterisk notation “*” can be used in address assignments inside programs, and function block types to denote not yet fully specified locations for directly represented variables.

EXAMPLE

```
VAR
  C2 AT %Q*: BYTE;
END_VAR
```

Assigns not yet located output byte to bitstring variable C2 of one byte length.

In the case that a directly represented variable is used in a location assignment to an internal variable in the declaration part of a program or a function block type, an asterisk “*” shall be used in place of the size prefix and the unsigned integer(s) in the concatenation to indicate that the direct representation is not yet fully specified.

Variables of this type shall not be used in the `VAR_INPUT` and `VAR_IN_OUT` section.

The use of this feature requires that the location of the variable so declared shall be fully specified inside the `VAR_CONFIG...END_VAR` construction of the configuration for every instance of the containing type.

It is an error if any of the full specifications in the `VAR_CONFIG...END_VAR` construction is missing for any incomplete address specification expressed by the asterisk notation “*” in any instance of programs or function block types which contain such incomplete specifications.

6.5.6 Retentive variables (RETAIN, NON_RETAIN)

6.5.6.1 General

When a configuration element (resource or configuration) is “started” as “warm restart” or “cold restart” according to Part 1 of the IEC 61131 series, each of the variables associated with the configuration element and its programs has a value depending on the starting operation of the configuration element and the declaration of the retain behavior of the variable.

The retentive behavior can declare for all variables contained in the variable sections `VAR_INPUT`, `VAR_OUTPUT`, and `VAR` of functions blocks and programs to be either retentive or non-retentive by using the `RETAIN` or `NON_RETAIN` qualifier specified in Figure 7. The usage of these keywords is an optional feature.

Figure 9 below shows the conditions for the initial value of a variable.

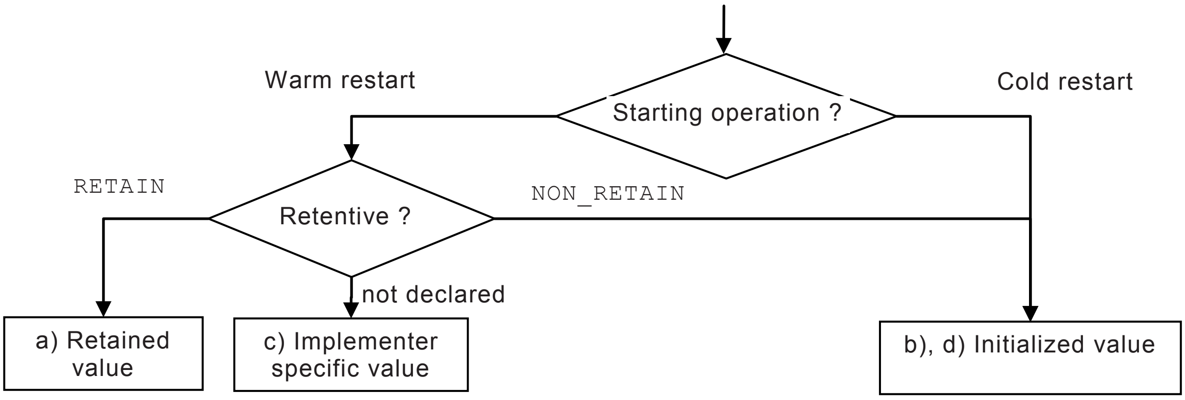


Figure 9 – Conditions for the initial value of a variable (Rules)

1. If the starting operation is a "warm restart" as defined in IEC 61131-1, the initial value of all variables in a variable section with RETAIN qualifier shall be the retained values. These are the values the variables had when the resource or configuration was stopped.
2. If the starting operation is a "warm restart", the initial value of all variables in a variable section with NON_RETAIN qualifier shall be initialized.
3. If the starting operation is a "warm restart" and there is no RETAIN and NON_RETAIN qualifier given, the initial values are Implementer specific.
4. If the starting operation is a "cold restart", the initial value of all variables in a VAR section with RETAIN and NON_RETAIN qualifier shall be initialized as defined below.

6.5.6.2 Initialization

The variables are initialized using the variable-specific user-defined values.

If no value is defined the type-specific user-defined initial value is used. If none is defined the type-specific default initial value is used, defined in Table 10.

Following further rules apply:

- Variables which represent inputs of the programmable controller system as defined in IEC 61131-1 shall be initialized in an Implementer specific manner.
- The RETAIN and NON_RETAIN qualifiers may be used for variables declared in static VAR, VAR_INPUT, VAR_OUTPUT, and VAR_GLOBAL sections but not in VAR_IN_OUT section.
- The usage of RETAIN and NON_RETAIN in the declaration of function block, class, and program instances is allowed. The effect is that all variables of the instance are treated as RETAIN or NON_RETAIN, except if:
  - the variable is explicitly declared as RETAIN or NON_RETAIN in the function block, class, or program type definition;
  - the variable itself is a function block type or a class. In this case, the retain declaration of the used function block type or class is applied.

The usage of RETAIN and NON_RETAIN for instances of structured data types is allowed. The effect is that all structure elements, also those of nested structures, are treated as RETAIN or NON_RETAIN.

EXAMPLE

```
VAR RETAIN
 AT %QW5: WORD:= 16#FF00;
 OUTARY AT %QW6: ARRAY[0..9] OF INT:= [10(1)];
 BITS: ARRAY[0..7] OF BOOL:= [1,1,0,0,0,1,0,0];
END_VAR

VAR NON_RETAIN
 BITS: ARRAY[0..7] OF BOOL;
 VALVE_POS AT %QW28: INT:= 100;
END_VAR
```

6.6 Program organization units (POUs)

6.6.1 Common features for POUs

6.6.1.1 General

The program organization units (POU) defined in this part of IEC 61131 are function, function block, class, and program. Function blocks and classes may contain methods.

A POU contains for the purpose of modularization and structuring a well-defined portion of the program. The POU has a defined interface with inputs and outputs and may be called and executed several times.

NOTE The above mentioned parameter interface is not the same as the interface defined in the context of object orientation.

POUs and methods can be delivered by the Implementer or programmed by the user.

A POU which has already been declared can be used in the declaration of other POUs as shown in Figure 3.

The recursive call of POUs and methods is Implementer specific.

The maximum number of POUs, methods and instances for a given resource are Implementer specific.

6.6.1.2 Assignment and expression

6.6.1.2.1 General

The language constructs of assignment and expression are used in the textual and (partially) in the graphical languages.

6.6.1.2.2 Assignment

An assignment is used to write the value of a literal, a constant expression, a variable, or an expression (see below) to another variable. This latter variable may be any kind of variable, like e.g. an input or an output variable of a function, method, function block, etc.

Variables of the same data type can always be assigned. Additionally the following rules apply:

- A variable or a constant of type `STRING` or `WSTRING` can be assigned to another variable of type `STRING` or `WSTRING` respectively. If the source string is longer than the target string the result is Implementer specific;
- A variable of a subrange type can be used anywhere a variable of its base type can be used. It is an error if the value of a subrange type falls outside the specified range of values;

- A variable of a derived type can be used anywhere a variable of its base type can be used;
- Additional rules for arrays may be defined by the Implementer.

Implicit and explicit data type conversion may be applied to adapt the data type of the source to the data type of the target:

a) In textual form (also partially applicable to graphical languages) the assignment operator may be

":= " which means the value of the expression on the right side of the operator is written to the variable on the left side of the operator or

" => " which means the value on the left side of the operator is written to the variable on the right side of the operator.

The "=>" operator is only used in the parameter list of calls of functions, methods, function blocks, etc. and only to pass VAR_OUTPUT parameter back to the caller.

EXAMPLE

```
A:= B + C/2;
Func (in1:= A, out2 => x);
A_struct1:= B_Struct1;
```

NOTE For assignment of user-defined data types (STUCTURE, ARRAY) see Table 72.

b) In graphical form

the assignment is visualized as a graphical connection line from a source to a target, in principle from left to right; e.g. from a function block output to a function block input or from the graphical "location" of a variable/constant to a function input or from an function output to the graphical "location" of a variable.

The standard function MOVE is one of the graphical representations of an assignment.

6.6.1.2.3 Expression

An expression is a language construct that consists of a defined combination of operands, like literals, variables, function calls, and operators like (+, -, *, /) and yields one value which may be multi-valued.

Implicit and explicit data type conversion may be applied to adapt the data types of an operation of the expression.

a) In textual form (also partially applicable in graphical languages), the expression is executed in a defined order depending on the precedence as specified in the language.

EXAMPLE `... B + C / 2 * SIN(x) ...`

b) In graphical form, the expression is visualized as a network of graphical blocks (function blocks, functions, etc.) connected with lines.

6.6.1.2.4 Constant expression

A constant expression is a language construct that consists of a defined combination of operands like literals, constant variables, enumerated values and operators like (+, -, *) and yields one value which may be multi-valued.

6.6.1.3 Partial access to ANY_BIT variables

For variables of the data type ANY_BIT (BYTE, WORD, DWORD, LWORD) a partial access to a bit, byte, word and double word of the variable is defined in Table 17.

In order to address the part of the variable, the symbol '%' and the size prefix as defined for directly represented variables in Table 16 (X, none, B, W, D, L) are used in combination with an integer literal (0 to max) for the address within the variable. The literal 0 refers to the least significant part and max refers to the most significant part. The '%X' is optional in the case of accessing bits.

EXAMPLE Partial access to ANY_BIT

```
VAR
  Bo: BOOL;
  By: BYTE;
  Wo: WORD;
  Do: DWORD;
  Lo: LWORD;
END_VAR;

  Bo:= By.%X0;// bit 0 of By
  Bo:= By.7;  // bit 7 of By; %X is the default and may be omitted.
  Bo:= Lo.63  // bit 63 of Lo;

  By:= Wo.%B1;// byte 1 of Wo;
  By:= Do.%B3;// byte 3 of Do;
```

Table 17 – Partial access of ANY_BIT variables

No.	Description	Data Type	Example and Syntax (NOTE 2)
	Data Type - Access to		myVAR_12.%X1; yourVAR1.%W3;
1a	BYTE – bit VB2.%X0	BOOL	<variable_name>.%X0 to <variable_name>.%X7
1b	WORD – bit VW3.%X15	BOOL	<variable_name>.%X0 to <variable_name>.%X15
1c	DWORD – bit	BOOL	<variable_name>.%X0 to <variable_name>.%X31
1d	LWORD – bit	BOOL	<variable_name>.%X0 to <variable_name>.%X63
2a	WORD – byte VW4.%B0	BYTE	<variable_name>.%B0 to <variable_name>.%B1
2b	DWORD – byte	BYTE	<variable_name>.%B0 to <variable_name>.%B3
2c	LWORD – byte	BYTE	<variable_name>.%B0 to <variable_name>.%B7
3a	DWORD – word	WORD	<variable_name>.%W0 to <variable_name>.%W1
3b	LWORD – word	WORD	<variable_name>.%W0 to <variable_name>.%W3
4	LWORD – dword VL5.%D1	DWORD	<variable_name>.%D0 to <variable_name>.%D1
The bit access prefix %X may be **omitted** according to Table 16, e.g. By1.%X7 is equivalent to By1.7. Partial access shall not be used with a direct variable e.g. %IB10.			

6.6.1.4 Call representation and rules

6.6.1.4.1 General

A call is used to execute a function, a function block instance, or a method of a function block or class. As illustrated in Figure 10 a call can be represented in a textual or graphical form.

1. Where no names are given for input variables of standard functions, the default names IN1, IN2, ... shall apply in top-to-bottom order. When a standard function has a single unnamed input, the default name IN shall apply.
2. It shall be an error if any VAR_IN_OUT variable of any call within a POU is not "properly mapped".
   A VAR_IN_OUT variable is "properly mapped" if

- it is connected graphically at the left, or
- it is assigned using the ":=" operator in a textual call, to a variable declared (without the CONSTANT qualifier) in a VAR_IN_OUT, VAR, VAR_TEMP, VAR_OUTPUT, or

VAR_ EXTERNAL block of the containing program organization unit, or to a “properly mapped” VAR_IN_OUT of another contained call.

3. A “properly mapped” (as shown in rule above) VAR_IN_OUT variable of a call can
   - be connected graphically at the right, or
   - be assigned using the “:=” operator in a textual assignment statement to a variable declared in a VAR, VAR_OUTPUT or VAR_EXTERNAL block of the containing program organization unit.

   It shall be an error if such a connection would lead to an ambiguous value of the variable so connected.

4. The name of a function block instance may be used as an input if it is declared as a VAR_INPUT, or as VAR_IN_OUT.

   The instance can be used inside the called entity in the following way:
   - if declared as VAR_INPUT the function block variables can only be read,
   - if declared as VAR_IN_OUT the function block variables can be read and written and the function block can be called.

6.6.1.4.2 Textual languages

The features for the textual call are defined in Table 20. The textual call shall consist of the name of the called entity followed by a list of parameters.

In the ST language the parameters shall be separated by commas and this list shall be delimited on the left and right by parentheses.

The parameter list of a call shall provide the actual values and may assign them to the corresponding formal parameters names (if any):

- Formal call

  The parameter list has the form of a set of assignments of actual values to the formal parameter names (formal parameter list), that is:

  a) assignments of values to input and in-out variables using the ":=" operator, and

  b) assignments of the values of output variables to variables using the "=>" operator.

  The formal parameter list may be complete or incomplete. Any variable to which no value is assigned in the list shall have the initial value, if any, assigned in the declaration of the called entity, or the default value for the associated data type.

  The ordering of parameters in the list shall not be significant.

  The execution control parameters EN and ENO may be used.

  EXAMPLE 1

  ```
  A:= LIMIT(EN:= COND, IN:= B, MN:= 0, MX:= 5, ENO => TEMPL); // Complete

  A:= LIMIT(IN:= B, MX:= 5);                                   // Incomplete
  ```

- Non-formal call

  The parameter list shall contain exactly the same number of parameters, in exactly the same order and of the same data types as given in the function definition, except the execution control parameters EN and ENO.

  EXAMPLE 2

  ```
  A:= LIMIT(B, 0, 5);
  ```

  This call is equivalent to the complete call in the example above, but without EN/ENO.

6.6.1.4.3 Graphical languages

In the graphic languages the call of functions shall be represented as graphic blocks according to the following rules:

1. The form of the block shall be rectangular.
2. The size and proportions of the block may vary depending on the number of inputs and other information to be displayed.
3. The direction of processing through the block shall be from left to right (input parameters on the left and output parameters on the right).
4. The name or symbol of the called entity, as specified below, shall be located inside the block.
5. Provision shall be made for input and output variable names appearing at the inside left and right sides of the block respectively.
6. An additional input EN and/or output ENO may be used. If present, they shall be shown at the uppermost positions at the left and right side of the block, respectively.
7. The function result shall be shown at the uppermost position at the right side of the block, except if there is an ENO output, in which case the function result shall be shown at the next position below the ENO output. Since the name of the called entity itself is used for the assignment of its output value, no output variable name shall be shown at the right side of the block, i.e. for the function result.
8. Parameter connections (including function result) shall be shown by signal flow lines.
9. Negation of Boolean signals shall be shown by placing an open circle just outside of the input or output line intersection with the block. In the character set this may be represented by the upper case alphabetic “O”, as shown in Table 20. The negation is performed outside the POU.
10. All inputs and outputs (including function result) of a graphically represented function shall be represented by a single line outside the corresponding side of the block, even though the data element may be a multi-element variable.
11. Results and outputs (VAR_OUTPUT) can be connected to a variable, used as input to other calls, or can be left unconnected.

Graphical example (FBD)	Textual example (ST)	Explanation

a)
`    +--------+`
`
	ADD |`
	`B---|        |--A`
	`C---|        |`
	`D---|        |`
	`    +--------+`
		`A:= ADD(B,C,D); //Function or`
		
		`A:= B + C + D; // Operators`
			Non-formal parameter list
			`(B, C, D)`

b)
`    +-------+`
`
	SHL |`
	`B---|IN     |--A`
	`C---|N      |`
	`    +-------+`
		`A:= SHL(IN:= B, N:= C);`	Formal parameter names
			`IN, N`

c)
`        +-------+`
`
	SHL |`
	`ENABLE--|EN  ENO|O-NO_ERR`
	`     B--|IN     |--A`
	`     C--|N      |`
	`        +-------+`
		`A:= SHL(`
		`      EN:= ENABLE,`
		`      IN:= B,`
		`      N := C,`
		`      NOT ENO => NO_ERR);`
			Formal parameter names
			
			Use of `EN` input
			and negated `ENO` output

d)
`   +-------+`
`
	INC |`
	`
		|--A`
		`X--|V-----V|--X`
		`   +-------+`
			`A:= INC(V:= X);`	User-defined `INC` function
				
				Formal parameter names `V` for `VAR_IN_OUT`

The examples illustrate both the graphical and equivalent textual use, including the use of a standard function (`ADD`) without defined formal parameter names; a standard function (`SHL`) with defined formal parameter names; the same function with additional use of `EN` input and negated `ENO` output; and a user-defined function (`INC`) with defined formal parameter names.		

Figure 10 – Formal and non-formal representation of call (Examples)

6.6.1.5 Execution control (EN, ENO)

As shown in Table 18, an additional Boolean `EN` (Enable) input or `ENO` (Enable Out) output, or both, can be provided by the Implementer or the user according to the declarations.

```
VAR_INPUT    EN:    BOOL:= 1;    END_VAR
VAR_OUTPUT   ENO:   BOOL;        END_VAR
```

When these variables are used, the execution of the operations defined by the POU shall be controlled according to the following rules:

1. If the value of EN is FALSE then the POU shall not be executed. In addition, ENO shall be reset to FALSE. The Implementer shall specify the behavior in this case in detail, see the examples below.
2. Otherwise, if the value of EN is TRUE, ENO is set to TRUE and the POU implementation shall be executed. The POU may set ENO to a Boolean value according to the result of the execution.
3. If any error occurs during the execution of one of the POU, the ENO output of that POU shall be reset to FALSE (0) by the programmable controller system, or the Implementer shall specify other disposition of such an error.
4. If the ENO output is evaluated to FALSE (0), the values of all POU outputs (VAR_OUTPUT, VAR_IN_OUT and function result) are Implementer specific.
5. The input EN shall only be set as an actual value as a part of a call of a POU.
6. The output ENO shall only be transferred to a variable as a part of a call of a POU.
7. The output ENO shall only be set inside its POU.

8. Use of the parameters EN/ENO in the function REF() to get a reference to EN/ENO is an error.

Behavior different from normal POU execution can be implemented in the case of `EN` being `FALSE`.This shall be specified by the Implementer. See examples below.

EXAMPLE 1 Internal implementation

The input `EN` is evaluated inside the POU.

If `EN` is `FALSE`, `ENO` is set to False and the POU returns immediately or performs a subset of operations depending on this situation.

All given input and in-out parameters are evaluated and set in the instance of the POU (except for functions).

The validity of the in-out parameters is checked.

EXAMPLE 2 External implementation

The input `EN` is evaluated outside the POU. If `EN` is False, only ENO is set to False and the POU is not called.

The input and in-out parameters are not evaluated and not set in the instance of the POU. The validity of the in-out parameters is not checked.

The input `EN` is not assigned outside the POU separately from the call.

The following figure and examples illustrate the usage with and without `EN/ENO`:

```
                  myInst
                +--------+
       cond     |  myFB  |         X
-----| |------|EN    ENO|---------( )
         v1 ---|A       B|--- v2
         v3 ---|C------C|---
                +--------+
```

EXAMPLE 3 Internal implementation

```
myInst (EN:= cond, A:= v1, C:= v3, B=> v2, ENO=> X);
```

where the body of myInst starts in principle with

```
IF NOT EN THEN...        // perform a subset of operations
                         // depending on the situation
ENO:= 0; RETURN; END_IF;
```

EXAMPLE 4 External implementation

```
IF cond THEN myInst (A:= v1, C:= v3, B=> v2, ENO=> X)
ELSE X:= 0; END_IF;
```

Table 18 shows the features for the call of POU without and with `EN/ENO`.

Table 18 – Execution control graphically using EN and ENO

<table>
<tr><th>No.</th><th>Description [a]</th><th>Example[b]</th></tr>
<tr><td>1</td><td>Usage without EN and ENO</td><td>Shown for a function in FBD and ST
<pre>
          +------+
      A---| + |---C
      B---| |
          +------+

C:= ADD(IN1:= A, IN2:= B);
</pre></td></tr>
<tr><td>2</td><td>Usage of EN only
(without ENO)</td><td>Shown for a function in FBD and ST
<pre>
          +------+
ADD_EN----|EN |
      A---| + |---C
      B---| |
          +------+

C:= ADD(EN:= ADD_EN. IN1:= A, IN2:= B);
</pre></td></tr>
<tr><td>3</td><td>Usage of ENO only
(without EN)</td><td>Shown for a function in FBD and ST
<pre>
          +------+
          | ENO|---ADD_OK
      A---| + |---C
      B---| |
          +------+

C:= ADD(IN1:= A, IN2:= B, ENO => ADD_OK);
</pre></td></tr>
<tr><td>4</td><td>Usage of EN and ENO</td><td>Shown for a function in LD and ST
<pre>
         +-------+ |
| ADD_EN | + | ADD_OK |
+---||---|EN ENO|---( )---+
A---		---C
B---		
         +-------+ |

C:= ADD(EN:= ADD_EN, IN1:= a, IN2:= IN2, EN => ADD_OK);
</pre></td></tr>
<tr><td colspan="3">[a] The Implementer shall specify in which of the languages the feature is supported; i.e. in an implementation it may be prohibited to use EN and/or ENO.
[b] The languages chosen for demonstrating the features above are given only as examples.</td></tr>
</table>

6.6.1.6 Data type conversion

Data type conversion is used to adapt data types for the use in expressions, assignments and parameter assignments.

The representation and the interpretation of the information stored in a variable are dependent of the declared data type of the variable. There are two cases where type conversion is used.

- In an assignment
  of a data value of a variable to another variable of a different data type.

  This is applicable with the assignment operators ":=" and "=>" and with the assignment of variables declared as parameters, i.e. inputs, outputs, etc. of functions, function blocks, methods, and programs. Figure 11 shows the conversion rules from a source data type to a target data type.

  EXAMPLE 1
  ```
  A:= B;                     // Variable assignment
  FB1 (x:= z, v => W);       // Parameter assignment
  ```

- In an expression (see 7.3.2 for ST language)
  consisting of operators like "+" and operands like literals and variables with the same or different data types.

EXAMPLE 2 `... SQRT( B + (C * 1.5)); // Expression`

- Explicit data type conversion
  is done by usage of the conversion functions.
- Implicit data type conversion
  has the following application rules:

  1. shall keep the value and accuracy of the data types,
  2. may be applied for typed functions,
  3. may be applied for assignments of an expression to a variable,

EXAMPLE 3

```
myUDInt:= myUInt1 * myUInt2;
     /* The multiplication has a UINT result
             which is then implicitly converted to an UDINT at the assignment */
```

4. may be applied for the assignment of an input parameter,
5. may be applied for the assignment of an output parameter,
6. shall not be applied for the assignment to in-out parameters,
7. may be applied so that operands and results of an operation or overloaded function get the same data type.

EXAMPLE 4

```
myUDInt:= myUInt1 * myUDInt2;
     // myUInt1 is implicitly converted to a UDINT, the multiplication has a UDINT result
```

8. The Implementer shall define the rules for non-typed literals.

NOTE The user can use typed literals to avoid ambiguities.

EXAMPLE 5

```
IF myWord = NOT (0) THEN ...;       // Ambiguous comparison with 16#FFF, 16#0001, 16#00FF, etc.
IF myWord = NOT (WORD#0) THEN ...; // Ambiguous comparison with 16#FFFF
```

Figure 11 shows the two alternatives “implicit” and “explicit” conversion of the source data type to a target data type.

Source Data Type		Target Data Type																											
		real		integer				unsigned				bit					date & times								char			
		LREAL	REAL	LINT	DINT	INT	SINT	ULINT	UDINT	UINT	USINT	LWORD	DWORD	WORD	BYTE	BOOL	LTIME	TIME	LDT	DT	LDATE	DATE	LTOD	TOD	WSTRING	STRING	WCHAR	CHAR
real	LREAL	▨	e	e	e	e	e	e	e	e	e	e	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	REAL	i	▨	e	e	e	e	e	e	e	e	-	e	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
integer	LINT	e	e	▨	e	e	e	e	e	e	e	e	e	e	e	-	-	-	-	-	-	-	-	-	-	-	-	-
	DINT	i	e	i	▨	e	e	e	e	e	e	e	e	e	e	-	-	-	-	-	-	-	-	-	-	-	-	-
	INT	i	i	i	i	▨	e	e	e	e	e	e	e	e	e	-	-	-	-	-	-	-	-	-	-	-	-	-
	SINT	i	i	i	i	i	▨	e	e	e	e	e	e	e	e	-	-	-	-	-	-	-	-	-	-	-	-	-
unsigned	ULINT	e	e	e	e	e	e	▨	e	e	e	e	e	e	e	-	-	-	-	-	-	-	-	-	-	-	-	-
	UDINT	i	e	i	e	e	e	i	▨	e	e	e	e	e	e	-	-	-	-	-	-	-	-	-	-	-	-	-
	UINT	i	i	i	i	e	e	i	i	▨	e	e	e	e	e	-	-	-	-	-	-	-	-	-	-	-	-	-
	USINT	i	i	i	i	i	e	i	i	i	▨	e	e	e	e	-	-	-	-	-	-	-	-	-	-	-	-	-
bit	LWORD	e	-	e	e	e	e	e	e	e	e	▨	e	e	e	-	-	-	-	-	-	-	-	-	-	-	-	-
	DWORD	-	e	e	e	e	e	e	e	e	e	i	▨	e	e	-	-	-	-	-	-	-	-	-	-	-	-	-
	WORD	-	-	e	e	e	e	e	e	e	e	i	i	▨	e	-	-	-	-	-	-	-	-	-	-	-	e	-
	BYTE	-	-	e	e	e	e	e	e	e	e	i	i	i	▨	-	-	-	-	-	-	-	-	-	-	-	-	e
	BOOL	-	-	e	e	e	e	e	e	e	e	i	i	i	i	▨	-	-	-	-	-	-	-	-	-	-	-	-
date & times	LTIME	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	▨	e	-	-	-	-	-	-	-	-	-	-
	TIME	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	i	▨	-	-	-	-	-	-	-	-	-	-
	LDT	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	▨	e	e	e	e	e	-	-	-	-
	DT	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	i	▨	e	e	e	e	-	-	-	-
	LDATE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	▨	e	-	-	-	-	-	-
	DATE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		i	▨	-	-	-	-	-	
	LTOD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	▨	e	-	-	-	-
	TOD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	i	▨	-	-	-	-
char	WSTRING	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-	-	▨	e	-	-
	STRING (NOTE)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-	-	e	▨	-	e
	WCHAR	-	-	-	-	-	-	-	-	-	-	e	e	e	-	-	-	-	-	-	-		-	-	i	-	▨	e
	CHAR (NOTE)	-	-	-	-	-	-	-	-	-	e	e	e	e	e	-	-	-	-	-	-		-	-	-	i	e	▨

Key

▨	No data type conversion necessary
-	No implicit or explicit data type conversion defined by this standard. The implementation may support additional Implementer specific data type conversions.
i	Implicit data type conversion; however, explicit type conversion is additionally allowed.
e	Explicit data type conversion applied by the user (standard conversion functions) may be used to accept loss of accuracy, mismatch in the range or to effect possible Implementer dependent behavior.

NOTE Conversions of STRING to WSTRING and CHAR to WCHAR are not implicit, to avoid conflicts with the used character set.

Figure 11 – Data type conversion rules – implicit and/or explicit (Summary)

The following Figure 12 shows the data type conversions which are supported by implicit type conversion. The arrows present the possible conversion paths; e.g. BOOL can be converted to BYTE, BYTE can be converted to WORD, etc.

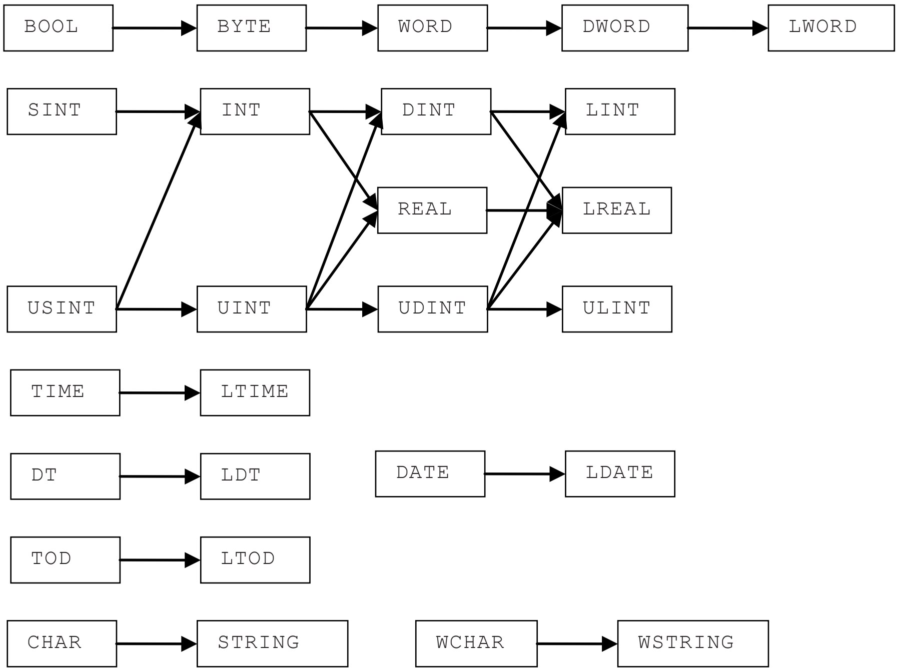


Figure 12 – Supported implicit type conversions

The following example shows examples of the data type conversion.

EXAMPLE 6 Explicit vs. implicit type conversion

1) Type declaration

```
VAR
  PartsRatePerHr: REAL;
  PartsDone:      INT;
  HoursElapsed:   REAL;
  PartsPerShift:  INT;
  ShiftLength:    SINT;
END_VAR
```

2) Usage in ST language

a) Explicit type conversion

```
PartsRatePerHr:= INT_TO_REAL(PartsDone) / HoursElapsed;
PartsPerShift := REAL_TO_INT(SINT_TO_REAL(ShiftLength)*PartsRatePerHr);
```

b) Explicit overloaded type conversion

```
PartsRatePerHr:= TO_REAL(PartsDone) / HoursElapsed;
PartsPerShift := TO_INT(TO_REAL(ShiftLength)*PartsRatePerHr);
```

c) Implicit type conversion

```
PartsRatePerHr:= PartsDone / HoursElapsed;
PartsPerShift := TO_INT(ShiftLength * PartsRatePerHr);
```

3) Usage in FBD language

a) Explicit type conversion

```
            +-------------+  +----------+   +---+  +-------------+
PartsDone  -| INT_TO_REAL |--| DIV_REAL |---| * |--| REAL_TO_INT |--- PartsPerShift
            +-------------+  |          |   |   |  |             |
                             |          |   |   |  +-------------+
HoursElapsed   --------------|          |   |   |
                             +----------+   |   |
                                            |   |
             +--------------+               |   |
ShiftLength -| SINT_TO_REAL |---------------|   |
             +--------------+               +---+
```

b) Explicit overloaded type conversion

```
             +-------------+   +----------+   +---+   +--------+
PartsDone   -| TO_REAL     |---| DIV_REAL |---| * |---| TO_INT |--- PartsPerShift
             +-------------+   |          |   |   |   |        |
                               |          |   |   |   +--------+
HoursElapsed ------------------|          |   |   |
                               +----------+   |   |
                                              |   |
             +--------------+                 |   |
ShiftLength -| TO_REAL      |-----------------|   |
             +--------------+                 +---+
```

c) Implicit type conversion with typed functions

```
                       +----------+   +----------+   +--------+
PartsDone -------------| DIV_REAL |---| MUL_REAL |---| TO_INT |--- PartsPerShift
                       |          |   |          |   |        |
                       |          |   |          |   +--------+
HoursElapsed-----------|          |   |          |
                       +----------+   |          |
                                      |          |
                                      |          |
ShiftLength --------------------------|          |
                                      +----------+
```

6.6.1.7 Overloading

6.6.1.7.1 General

A language element is said to be overloaded when it can operate on input data elements of various types within a generic data type; e.g. ANY_NUM, ANY_INT.

The following standard language elements which are provided by the manufacturer may have generic overloading as a special feature:

- Standard functions

  These are overloaded standard functions (e.g. ADD, MUL) and overloaded standard conversion functions (e.g. TO_REAL, TO_INT).

- Standard methods

  This part of IEC 61131 does not define standard methods within standard classes or function block types. However, they may be supplied by the Implementer.

- Standard function blocks

  This part of IEC 61131 does not define standard function blocks, except some simple ones like counters.
  However, they may be defined by other parts of IEC 61131 and may be supplied by the Implementer.

- Standard classes

  This part of IEC 61131 does not define standard classes. However, they may be defined in other parts of IEC 61131 and may be supplied by the Implementer.

- Operators
  These are e.g. “+” and “*” in ST language; ADD, MUL in IL language.

6.6.1.7.2 Data type conversion

When a programmable controller system supports an overloaded language element, this language element shall apply to all suitable data types of the given generic type which are supported by that system.

The suitable data types for each language element are defined in the related features tables. The following examples illustrate the details:

EXAMPLE 1

> This standard defines for the ADD function the generic data type ANY_NUM for a number of inputs of the same kind and one result output.
> The Implementer specifies for this generic data type ANY_NUM of the PLC system the related elementary data types REAL and INT.

EXAMPLE 2

> This standard defines for the bit-shift function LEFT the generic data type ANY_BIT for one input and the result output and the generic data type ANY_INT for another input.
> The Implementer specifies for these two generic data types of the PLC system:
> ANY_BIT represents e.g. the elementary data types BYTE and WORD;
> ANY_INT represents e.g. the elementary data types INT and LINT.

An overloaded language element shall operate on the defined elementary data types according the following rules:

- The data types of inputs and the outputs/result shall be of the same type; this is applicable for the inputs and outputs/result of the same kind.
  The same kind means parameters, operands and the result equally used like the inputs of an addition or multiplication.
  More complex combinations shall be Implementer specific.
- If the data types of the inputs and outputs of the same kind have not the same type then the conversion in the language element is Implementer specific.
- The implicit type conversion of an expression and of the assignment follows the sequence of evaluation of the expression. See example below.
- The data type of the variable to store the result of the overloaded function does not influence the data type of the result of the function or operation.

NOTE The user can explicitly specify the result type of the operation by using typed functions.

EXAMPLE 3

```
int3  := int1   + int2   (* Addition is performed as an integer operation *)
dint1:= int1    + int2;  (* Addition is performed as an integer operation, then the result is converted
                            to a DINT and assigned to dint1 *)
dint1:= dint2  + int3;   (* int3 is converted to a DINT, the addition is performed as a DINT addition *)
```

6.6.2 Functions

6.6.2.1 General

A function is a programmable organization unit (POU) which does not store its state; i.e. inputs, internals and outputs/result.

The common features of POUs apply for functions if not stated otherwise.

The Function execution

- delivers typically a temporary result which may be a one-data element or a multi-valued array or structure,
- delivers possibly output variable(s) which may be multi-valued,
- may change the value of in-out and VAR_EXTERNAL variable(s).

A function with result may be called in an expression or as a statement.

A function without result shall not be called inside an expression.

6.6.2.2 Function declaration

The declaration of a function shall consist of the following elements as defined in Table 19: These features are declared in a similar manner as described for the function blocks.

Following rules for the declaration of a function shall be applied as given in the Table 19:

1. The declarations begin with the keyword FUNCTION followed by an identifier specifying the name of the function.
2. If a result is available a colon ':', and followed by the data type of the value to be returned by the function shall be given or if no function result is available, the colon and data type shall be omitted.
3. The constructs with VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT, if required, specifying the names and data types of the function parameters.
4. The values of the variables which are passed to the function via a VAR_EXTERNAL construct can be modified from within the function block.
5. The values of the constants which are passed to the function via a VAR_EXTERNAL CONSTANT construct cannot be modified from within the function.
6. The values of variables which are passed to the function via a VAR_IN_OUT construct can be modified from within the function.
7. The variable-length arrays may be used as VAR_INPUT, VAR_OUTPUT and VAR_IN_OUT.
8. The input, output, and temporary variables may be initialized.
9. EN/ENO inputs and outputs may be used as described.
10. A VAR...END_VAR construct and also the VAR_TEMP...END_VAR, if required, specifying the names and types of the internal temporary variables.
    In contrast to function blocks, the variables declared in the VAR section are not stored.
11. If the generic data types (e.g. ANY_INT) are used in the declaration of standard function variables, then the rules for using the actual types of the parameters of such functions shall be part of the function definition.
12. The variable initialization constructs can be used for the declaration of initial values of function inputs and initial values of their internal and output variables.
13. The keyword END_FUNCTION terminates the declaration.

Table 19 – Function declaration

No.	Description	Example
1a	Without result `FUNCTION ... END_FUNCTION`	`FUNCTION myFC ... END_FUNCTION`

1b	With result
	`FUNCTION <name>: <data type>`
	`END _FUNCTION`
		`FUNCTION myFC: INT ... END_FUNCTION`

2a	Inputs `VAR_INPUT...END_VAR`	`VAR_INPUT IN: BOOL; T1: TIME; END_VAR`
2b	Outputs `VAR_OUTPUT...END_VAR`	`VAR_OUTPUT OUT: BOOL; ET_OFF: TIME; END_VAR`
2c	In-outs `VAR_IN_OUT...END_VAR`	`VAR_IN_OUT A: INT; END_VAR`
2d	Temporary variables `VAR_TEMP...END_VAR`	`VAR_TEMP I: INT; END_VAR`
2e	Temporary variables `VAR...END_VAR`	`VAR B: REAL; END_VAR` For compatibility reason a difference to function blocks: VARs are static in function blocks (stored)!

2f	External variables
	`VAR_EXTERNAL...END_VAR`
		`VAR_EXTERNAL B: REAL; END_VAR`
		Corresponding to
		`VAR_GLOBAL B: REAL...`

2g	External constants
	`VAR_EXTERNAL CONSTANT...END_VAR`
		`VAR_EXTERNAL CONSTANT B: REAL; END_VAR`
		Corresponding to
		`VAR_GLOBAL B: REAL`

3a	Initialization of inputs	`VAR_INPUT MN: INT:= 0;`
3b	Initialization of outputs	`VAR_OUTPUT RES: INT:= 1;`
3c	Initialization of temporary variables	`VAR I: INT:= 1;`
--	`EN/ENO` inputs and outputs	Defined in Table 18

EXAMPLE

```
// Parameter interface specification

  FUNCTION SIMPLE_FUN: REAL
  VAR_INPUT
       A, B: REAL;
       C:    REAL:= 1.0;
  END_VAR
   VAR_IN_OUT COUNT: INT;
  END_VAR
```

```
// Parameter interface specification
                 FUNCTION
              +-------------+
              |  SIMPLE_FUN |
      REAL----|A            |----REAL
      REAL----|B            |
      REAL----|C            |
      INT-----|COUNT---COUNT|----INT
              +-------------+
```

```
 // Function body specification

  VAR COUNTP1: INT; END_VAR
  COUNTP1:= ADD(COUNT, 1);
  COUNT  := COUNTP1


  SIMPLE_FUN:= A*B/C; // result
  END_FUNCTION
```

```
 // Function body specification
              +---+
              |ADD|---         +----+
       COUNT--|   |---COUNTP1--|:= |---COUNT
           1--|   |            +----+
              +---+   +---+
                  A---| * |   +---+
                  B---|   |---| / |-SIMPLE_FUN
                      +---+   |   |
                  C-----------|   |
                              +---+
 END_FUNCTION
```

a) Function declaration and body (ST and FBD) – NOTE

```
VAR_GLOBAL DataArray: ARRAY [0..100]
OF INT; END_VAR
```

// External interface

```
FUNCTION SPECIAL_FUN

VAR_INPUT
  FirstIndex: INT;
  LastIndex:  INT;
END_VAR

VAR_OUTPUT
  Sum: INT;
END_VAR

VAR_EXTERNAL DataArray:
    ARRAY [0..100] OF INT;
END_VAR
```

// no function result, but output Sum

```
       +------------------+
       |   SPECIAL_FUN    |
INT----|FirstIndex     Sum|----INT
INT----|LastIndex         |
       +------------------+
```

```
VAR I: INT; Sum: INT:= 0; END_VAR

  FOR i:= FirstIndex TO LastIndex
    DO Sum:= Sum + DataArray[i];
  END_FOR

END_FUNCTION
```

// Function body – Not graphically shown

b) Function declaration and body (without function result – with Var output)

NOTE In a), the input variable is given a defined default value of 1.0 to avoid a “division by zero” error if the input is not specified when the function is called, for example, if a graphical input to the function is left unconnected.

6.6.2.3 Function call

A call of a function can be represented in a textual or graphical form.

Since the input variables, the output variables and the result of a function are not stored, the assignment to the inputs, the access to the outputs and to the result shall be immediate with the call of the function.

If a variable-length array is used as a parameter, the parameter shall be connected to the static variable.

A function shall not contain any internal state information, i.e.

- it does not store any of the input, internal (temporary) and output element(s) from one call to the next;

- the call of a function with the same parameters (VAR_INPUT and VAR_IN_OUT) and the same values of VAR_EXTERNAL will always yield the same value of its output variables, in-out variables, external variables and its function result, if any.

NOTE 1 Some functions, typically provided as system functions by the Implementer, may yield different values; e.g. TIME(), RANDOM().

Table 20 – Function call

No.	Description	Example

1a	Complete formal call (textual only)
	NOTE 1 This is used if EN/ENO is necessary in calls.
		`A:= LIMIT(   EN:= COND,`
		`              IN:= B,`
		`              MN:= 0,`
		`              MX:= 5,`
		`              ENO => TEMPL);`

1b	Incomplete formal call (textual only)
	NOTE 2 This is used if EN/ENO is not necessary in calls.
		`A:= LIMIT(   IN:= B,`
		`              MX:= 5);`
		NOTE 3 MN variable will have the default value 0 (zero).

2	Non-formal call (textual only) (fix order and complete) NOTE 4 This is used for call of standard functions without formal names.	`A:= LIMIT(B, 0, 5);` NOTE 5 This call is equivalent to 1a, but without EN/ENO.

3	Function without function result	`FUNCTION myFun       // no type declararion`
		`VAR_INPUT  x: INT;   END_VAR;`
		`VAR_OUTPUT y: REAL;  END_VAR;`
		`myFun(150, var);     // Call`

4	Graphical representation	`        +-------+`
		`        |  FUN  |`
		`    a --|EN  ENO|--`
		`    b --|IN1    |-- result`
		`    c --|IN2  Q1|--out`
		`        |     Q2|`
		`        +-------+`

5	Usage of negated boolean input and output in graphical representation	`        +-------+`
		`        |  FUN  |`
		`    a -o|EN  ENO|--`
		`    b --|IN1    |-- result`
		`    c --|IN2  Q1|o- out`
		`        |     Q2|`
		`        +-------+`
		NOTE 6 The use of these constructs is forbidden for in-out variables.

6	Graphical usage of VAR_IN_OUT	`      +------------+`
		`      |   myFC1    |`
		`  a --|In1     Out1|-- d`
		`  b --|Inout--Inout|-- c`
		`      +------------+`

EXAMPLE Function call

Call

```
VAR
  X, Y, Z, Res1, Res2: REAL;
  En1, V: BOOL;
END_VAR

Res1:= DIV(In1:= COS(X), In2:= SIN(Y), ENO => EN1);
Res2:= MUL(SIN(X), COS(Y));
Z   := ADD(EN:= EN1, IN1:= Res1, IN2:= Res2, ENO => V);
          +-----+      +------+     +------+
    X --+-| COS |--+  -|EN ENO|-----|EN ENO|-- V
        | |     |  |   |      |     |      |
        | +-----+  +---|  DIV |-----|  ADD |-- Z
        |              |      |     |      |
        | +-----+      |      |   +-|      |
   Y -+---| SIN |------|      |   | +------+
      | | |     |      +------+   |
      | | +-----+                 |
      | |                         |
      | | +-----+      +------+   |
      | +-| SIN |--+  -|EN ENO|-  |
      |   |     |  |   |      |   |
      |   +-----+  +---| MUL  |---+
      |                |      |
      |   +-----+      |      |
      +---| COS |------|      |
          |     |      +------+
          +-----+
```

a) Standard functions call with result and EN/ENO

Declaration

```
FUNCTION My_function                                   // no type, no result
      VAR_INPUT In1:              REAL; END_VAR
      VAR_OUTPUT Out1, Out2:      REAL; END_VAR
      VAR_TEMP Tmp1:              REAL; END_VAR        // VAR_TEMP allowed
      VAR_EXTERNAL Ext:           BOOL; END_VAR


      // Function body

END_FUNCTION
```

Call textual and graphical

```
My_Function (In1:= a, Out1 => b; Out2 => c);

                    +------------+
                    | My_Function|                     // Without result
                a --|In1     Out1|-- b
                    |        Out2|-- c                 // With 2 outputs
                    +------------+
```

b) Function declaration and call without result but with output variables

Call textual and graphical

```
myFC1 (In1:= a, Inout:= b, Out1 => Tmp1);       // Usage of a temporary variable
d:= myFC2 (In1:= Tmp1, Inout:= b);              // b stored in inout; Assignment to c
c:= b;                                          // b assigned to c


                    +------------+      +------------+
                    |   myFC1    |      |    myFC2   |
                a --|In1     Out1|------|In1         |-- d    // Result
                b --|Inout--Inout|------|Inout--Inout|-- c    // Assignment to c
                    +------------+      |            |
                                        +------------+
```

c) Function call with graphical representation of in-out variables

Call textual and graphical

```
~~My_Function (In1:= a, Out1+Out2 => d);~~          // not permitted in ST
My_Function (In1:= a, Out1 => Tmp1, Out2 => Tmp2);
d:= Tmp1 + Tmp2;

                    +------------+      +---------+
                    | My_Function|      |    +    |-- d
                a --|In1     Out1|------|In1      |
                    |        Out2|------|In2      |
                    +------------+      +---------+
```

d) Function call without result but with expression of output variables

NOTE 2 These examples show two different representations of the same functionality. It is not required to support any automatic transformation between the two forms of representation.

6.6.2.4 Typed and overloading functions

A function which normally represents an overloaded operator is to be typed. This shall be done by appending a "_" (underscore) character followed by the required type, as shown in Table 21. The typed function is performed using the type as data type for its inputs and outputs. Implicit or explicit type conversion may apply.

An overloaded conversion function of the form `TO_xxx` or `TRUNC_xxx` with `xxx` as the typed elementary output type can be typed by preceding the required elementary data and a following "underscore" character.

Table 21 – Typed and overloaded functions

No.	Description	Example

1a	Overloaded function
	ADD (ANY_Num to ANY_Num)
		`          +---------+`
		`
			ADD |`
			`ANY_NUM --|         |-- ANY_NUM`
			`ANY_NUM --|         |`
			`   .    --|         |`
			`   .    --|         |`
			`ANY_NUM --|         |`
			`          +---------+`

1b	Conversion of inputs
	`ANY_ELEMENT TO_INT`
		`                  +---------+`
		`ANY_ELEMENTARY---|  TO_INT |----INT`
		`                  +---------+`

2a [a]	Typed functions:
	`ADD_INT`
		`      +---------+`
		`
			ADD_INT |`
			`INT --|         |-- INT`
			`INT --|         |`
			` .  --|         |`
			` .  --|         |`
			`INT --|         |`
			`      +---------+`

2b [a]	Conversion:
	`WORD_TO_INT`
		`        +-----------+`
		`WORD----|WORD_TO_INT|---INT`
		`        +-----------+`

NOTE The overloading of non-standard functions or function block types is beyond the scope of this standard.		
[a] If feature 2 is supported, the Implementer provides an additional table showing which functions are overloaded and which are typed in the implementation.		

EXAMPLE 1 Typed and overloaded functions

```
VAR                                    +---+
   A: INT;                         A --| + |-- C
   B: INT;                         B --|   |
   C: INT;                             +---+
END_VAR
                                  C:= A+B;
```

NOTE 1 Type conversion is not required in the example shown above.

```
                            +-----------+   +---+             +--------+   +---+
VAR                     A --|INT_TO_REAL|---| + |-- C     A---|TO_REAL|---|ADD|---C
   A: INT;                  +-----------+   |   |             +--------+   |   |
   B: REAL;             B ------------------|   |         B----------------|   |
   C: REAL;                                 +---+                          +---+
END_VAR                   C:= INT_TO_REAL(A)+B;             C:= TO_REAL(A) + B;

                                                              +---+   +--------+
VAR                         +---+   +-----------+         A---|ADD|---|TO_REAL|-- C
   A: INT;              A --| + |---|INT_TO_REAL|-- C     B---|   |   +--------+
   B: INT;              B --|   |   +-----------+             +---+
   C: REAL;                 +---+                          C:= TO_REAL(A+B);
END_VAR
                          C:= INT_TO_REAL(A+B);
```

a) Type declaration (ST) **b) Usage** (FBD and ST)

EXAMPLE 2 Explicit and implicit type conversion with typed functions

```
VAR                                 +---------+
  A: INT;                       A---| ADD_INT |---C
  B: INT;                       B---|         |
  C: INT;                           +---------+
END_VAR
                                  C:= ADD_INT(A, B);
```

NOTE 2 Type conversion is not required in the example shown above.

```
VAR                           Explicit type conversion
  A: INT;                                    +------------+  +----------+
  B: REAL;                                A--|INT_TO_REAL|--| ADD_REAL |-- C
  C: REAL;                                   +------------+  |          |
END_VAR                                   B-----------------|          |
                                                             +----------+

                                  C:= ADD_REAL(INT_TO_REAL(A), B);

VAR                           Implicit type conversion
  A: INT;                                                   +----------+
  B: REAL;                                 A --------------| ADD_REAL |-- C
  C: REAL;                                                  |          |
END_VAR                                    B --------------|          |
                                                            +----------+

                                  C:= ADD_REAL(A,B);

VAR                           Explicit type conversion
  A: INT;                                      +---------+  +-----------+
  B: INT;                                  A --| ADD_INT |--|INT_TO_REAL|-- C
  C: REAL;                                     |         |  +-----------+
END_VAR                                    B --|         |
                                               +---------|

                                  C:= INT_TO_REAL(ADD_INT(A, B));

VAR                           Implicit type conversion
  A: INT;                                      +---------+
  B: INT;                                  A --| ADD_INT |-- C
  C: REAL;                                     |         |
END_VAR                                    B --|         |
                                               +---------|

                                  C:= ADD_INT(A, B);
```

a) Type declaration (ST) **b) Usage** (FBD and ST)

6.6.2.5 Standard functions

6.6.2.5.1 General

A standard function specified in this subclause to be extensible is allowed to have two or more inputs to which the indicated operation is to be applied, for example, extensible addition shall give at its output the sum of all its inputs. The maximum number of inputs of an extensible function is an Implementer specific. The actual number of inputs effective in a formal call of an extensible function is determined by the formal input name with the highest position in the sequence of variable names.

EXAMPLE 1

The statement `X:= ADD(Y1, Y2, Y3);`
is equivalent to `X:= ADD(IN1:= Y1, IN2:= Y2, IN3:= Y3);`

EXAMPLE 2

The statement `I:= MUX_INT(K:=3, IN0:= 1, IN2:= 2, IN4:= 3);`
is equivalent to `I:= 0;`

6.6.2.5.2 Data type conversion functions

As shown in Table 22, type conversion functions shall have the form `*_TO_**`, where “*” is the type of the input variable `IN`, and “**” the type of the output variable `OUT`, for example, `INT_TO_REAL`. The effects of type conversions on accuracy, and the types of error**s** that may arise during execution of type conversion operations, are Implementer specific.

Table 22 – Data type conversion function

<table>
<tr><th>No.</th><th>Description</th><th>Graphical form</th><th>Usage example</th></tr>
<tr><td>1a</td><td>Typed conversion
input_TO_output</td><td>+-----------+
B ---| *_TO_** |--- A
+-----------+
(*) - Input data type, e.g., INT
(**) - Output data type, e.g., REAL</td><td>A:= INT_TO_REAL(B);</td></tr>
<tr><td>1b[a,b,e]</td><td>Overloaded conversion
TO_output</td><td>+-----------+
B ---| TO_** |--- A
+-----------+
- Input data type, e.g., INT
(**) - Output data type, e.g., REAL</td><td>A:= TO_REAL(B);</td></tr>
<tr><td>2a[c]</td><td>“Old” overloaded truncation
TRUNC</td><td>+-----------+
ANY_REAL ---| TRUNC |--- ANY_INT
+-----------+</td><td><strong>Deprecated</strong></td></tr>
<tr><td>2b[c]</td><td>Typed truncation
input_TRUNC_output</td><td>+-----------+
ANY_REAL ---|*_TRUNC_** |--- ANY_INT
+-----------+</td><td>A:=
REAL_TRUNC_INT(B);</td></tr>
<tr><td>2c[c]</td><td>Overloaded truncation
TRUNC_output</td><td>+-----------+
ANY_REAL ---| TRUNC_** |--- ANY_INT
+-----------+</td><td>A:= TRUNC_INT(B);</td></tr>
<tr><td>3a[d]</td><td>Typed
input_BCD_TO_output</td><td>+-----------+
* ---|*_BCD_TO_**|--- **
+-----------+</td><td>A:=
WORD_BCD_TO_INT(B);</td></tr>
<tr><td>3b[d]</td><td>Overloaded
BCD_TO_output</td><td>+-----------+
* ----| BCD_TO_** |--- **
+-----------+</td><td>A:= BCD_TO_INT(B);</td></tr>
<tr><td>4a[d]</td><td>Typed
input_TO_BCD_output</td><td>+-----------+
** ----|**_TO_BCD_*|--- *
+-----------+</td><td>A:=
INT_TO_BCD_WORD(B);</td></tr>
<tr><td>4b[d]</td><td>Overloaded
TO_BCD_output</td><td>+-----------+
* ----| TO_BCD_** |--- **
+-----------+</td><td>A:= TO_BCD_WORD(B);</td></tr>
</table>

No.	Description	Graphical form	Usage example
NOTE Usage examples are given in the ST language.			

[a] A statement of conformance to feature 1 of this table shall include a list of the specific type conversions supported, and a statement of the effects of performing each conversion.

[b] Conversion from type REAL or LREAL to SINT, INT, DINT or LINT shall round according to the convention of IEC 60559, according to which, if the two nearest integers are equally near, the result shall be the nearest even integer, e.g.:

```
REAL_TO_INT ( 1.6) is equivalent to  2
REAL_TO_INT (-1.6) is equivalent to -2

REAL_TO_INT ( 1.5) is equivalent to  2
REAL_TO_INT (-1.5) is equivalent to -2

REAL_TO_INT ( 1.4) is equivalent to  1
REAL_TO_INT (-1.4) is equivalent to -1

REAL_TO_INT ( 2.5) is equivalent to  2
REAL_TO_INT (-2.5) is equivalent to -2.
```

[c] The function TRUNC_* is used for truncation toward zero of a REAL or LREAL yielding a variable of one of the integer types, for instance

```
TRUNC_INT ( 1.6)  is equivalent to INT#1
TRUNC_INT (-1.6)  is equivalent to INT#-1

TRUNC_SINT ( 1.4)  is equivalent to SINT#1
TRUNC_SINT (-1.4)  is equivalent to SINT#-1.
```

[d] The conversion functions *_BCD_TO_** and **_TO_BCD_* shall perform conversions between variables of type BYTE, WORD, DWORD, and LWORD and variables of type USINT, UINT, UDINT and ULINT (represented by "*" and "**" respectively), when the corresponding bit-string variables contain data encoded in BCD format. For example, the value of USINT_TO_BCD_BYTE(25) would be 2#0010_0101, and the value of WORD_BCD_TO_UINT (2#0011_0110_1001) would be 369.

[e] When an input or output of a type conversion function is of type STRING or WSTRING, the character string data shall conform to the external representation of the corresponding data, as specified in 6.3.3, in the character set defined in 6.1.1.

6.6.2.5.3 Data type conversion of numeric data types

Numeric data type conversion uses the following rules:

1. The source data type is extended to its largest data type of the same data type category holding its value.
2. Then the result is converted to the largest data type of data type category to which the target data type belongs to.
3. Then the result is converted to the target data type.

   If the value of the source variable does not fit into the target data type, i.e. the value range is too small, then value of the target variable is Implementer specific.

NOTE The implementation of the conversion function can use a more efficient procedure.

EXAMPLE

X:= REAL_TO_INT (70_000.4)

1. REAL value (70_000.4) converted to LREAL value (70_000.400_000..).
2. LREAL value (70_000.4000_000..) converted to LINT value (70_000). Here rounded to an integer.
3. LINT value (70_000) converted to INT value. Here Implementer specific because INT can maximal hold 65.536.

This results in a variable of the target data type which holds the same value as the source variable, if the target data type is able to hold this value. When converting a floating point number, normal rounding rules are applied i.e. rounding to the nearest integer and if this is ambiguous, to the nearest even integer.

The data type BOOL used as a source data type is treated like an unsigned integer data type which can only hold the values 0 and 1.

Table 23 describes the conversion functions with conversion details as result of the rules above.

Table 23 – Data type conversion of numeric data types

No	Conversion Function	Conversion Details
1	LREAL _TO_ REAL	Conversion with rounding, value range errors give an Implementer specific result
2	LREAL _TO_ LINT	Conversion with rounding, value range errors give an Implementer specific result
3	LREAL _TO_ DINT	Conversion with rounding, value range errors give an Implementer specific result
4	LREAL _TO_ INT	Conversion with rounding, value range errors give an Implementer specific result
5	LREAL _TO_ SINT	Conversion with rounding, value range errors give an Implementer specific result
6	LREAL _TO_ ULINT	Conversion with rounding, value range errors give an Implementer specific result
7	LREAL _TO_ UDINT	Conversion with rounding, value range errors give an Implementer specific result
8	LREAL _TO_ UINT	Conversion with rounding, value range errors give an Implementer specific result
9	LREAL _TO_ USINT	Conversion with rounding, value range errors give an Implementer specific result
10	REAL _TO_ LREAL	Value preserving conversion
11	REAL _TO_ LINT	Conversion with rounding, value range errors give an Implementer specific result
12	REAL _TO_ DINT	Conversion with rounding, value range errors give an Implementer specific result
13	REAL _TO_ INT	Conversion with rounding, value range errors give an Implementer specific result
14	REAL _TO_ SINT	Conversion with rounding, value range errors give an Implementer specific result
15	REAL _TO_ ULINT	Conversion with rounding, value range errors give an Implementer specific result
16	REAL _TO_ UDINT	Conversion with rounding, value range errors give an Implementer specific result
17	REAL _TO_ UINT	Conversion with rounding, value range errors give an Implementer specific result
18	REAL _TO_ USINT	Conversion with rounding, value range errors give an Implementer specific result
19	LINT _TO_ LREAL	Conversion with potential loss of accuracy
20	LINT _TO_ REAL	Conversion with potential loss of accuracy
21	LINT _TO_ DINT	Value range errors give an Implementer specific result
22	LINT _TO_ INT	Value range errors give an Implementer specific result
23	LINT _TO_ SINT	Value range errors give an Implementer specific result
24	LINT _TO_ ULINT	Value range errors give an Implementer specific result
25	LINT _TO_ UDINT	Value range errors give an Implementer specific result
26	LINT _TO_ UINT	Value range errors give an Implementer specific result
27	LINT _TO_ USINT	Value range errors give an Implementer specific result
28	DINT _TO_ LREAL	Value preserving conversion
29	DINT _TO_ REAL	Conversion with potential loss of accuracy
30	DINT _TO_ LINT	Value preserving conversion
31	DINT _TO_ INT	Value range errors give an Implementer specific result
32	DINT _TO_ SINT	Value range errors give an Implementer specific result
33	DINT _TO_ ULINT	Value range errors give an Implementer specific result
34	DINT _TO_ UDINT	Value range errors give an Implementer specific result
35	DINT _TO_ UINT	Value range errors give an Implementer specific result

No	Conversion Function	Conversion Details
36	DINT _TO_ USINT	Value range errors give an Implementer specific result
37	INT _TO_ LREAL	Value preserving conversion
38	INT _TO_ REAL	Value preserving conversion
39	INT _TO_ LINT	Value preserving conversion
40	INT _TO_ DINT	Value preserving conversion
41	INT _TO_ SINT	Value range errors give an Implementer specific result
42	INT _TO_ ULINT	Value range errors give an Implementer specific result
43	INT _TO_ UDINT	Value range errors give an Implementer specific result
44	INT _TO_ UINT	Value range errors give an Implementer specific result
45	INT _TO_ USINT	Value range errors give an Implementer specific result
46	SINT _TO_ LREAL	Value preserving conversion
47	SINT _TO_ REAL	Value preserving conversion
48	SINT _TO_ LINT	Value preserving conversion
49	SINT _TO_ DINT	Value preserving conversion
50	SINT _TO_ INT	Value preserving conversion
51	SINT _TO_ ULINT	Value range errors give an Implementer specific result
52	SINT _TO_ UDINT	Value range errors give an Implementer specific result
53	SINT _TO_ UINT	Value range errors give an Implementer specific result
54	SINT _TO_ USINT	Value range errors give an Implementer specific result
55	ULINT _TO_ LREAL	Conversion with potential loss of accuracy
56	ULINT _TO_ REAL	Conversion with potential loss of accuracy
57	ULINT _TO_ LINT	Value range errors give an Implementer specific result
58	ULINT _TO_ DINT	Value range errors give an Implementer specific result
59	ULINT _TO_ INT	Value range errors give an Implementer specific result
60	ULINT _TO_ SINT	Value range errors give an Implementer specific result
61	ULINT _TO_ UDINT	Value range errors give an Implementer specific result
62	ULINT _TO_ UINT	Value range errors give an Implementer specific result
63	ULINT _TO_ USINT	Value range errors give an Implementer specific result
64	UDINT _TO_ LREAL	Value preserving conversion
65	UDINT _TO_ REAL	Conversion with potential loss of accuracy
66	UDINT _TO_ LINT	Value preserving conversion
67	UDINT _TO_ DINT	Value range errors give an Implementer specific result
68	UDINT _TO_ INT	Value range errors give an Implementer specific result
69	UDINT _TO_ SINT	Value range errors give an Implementer specific result
70	UDINT _TO_ ULINT	Value preserving conversion
71	UDINT _TO_ UINT	Value range errors give an Implementer specific result
72	UDINT _TO_ USINT	Value range errors give an Implementer specific result
73	UINT _TO_ LREAL	Value preserving conversion
74	UINT _TO_ REAL	Value preserving conversion
75	UINT _TO_ LINT	Value preserving conversion
76	UINT _TO_ DINT	Value preserving conversion
77	UINT _TO_ INT	Value range errors give an Implementer specific result
78	UINT _TO_ SINT	Value range errors give an Implementer specific result

No	Conversion Function	Conversion Details
79	UINT _TO_ ULINT	Value preserving conversion
80	UINT _TO_ UDINT	Value preserving conversion
81	UINT _TO_ USINT	Value range errors give an Implementer specific result
82	USINT _TO_ LREAL	Value preserving conversion
83	USINT _TO_ REAL	Value preserving conversion
84	USINT _TO_ LINT	Value preserving conversion
85	USINT _TO_ DINT	Value preserving conversion
86	USINT _TO_ INT	Value preserving conversion
87	USINT _TO_ SINT	Value range errors give an Implementer specific result
88	USINT _TO_ ULINT	Value preserving conversion
89	USINT _TO_ UDINT	Value preserving conversion
90	USINT _TO_ UINT	Value preserving conversion

6.6.2.5.4 Data type conversion of bit data types

This data type conversion uses the following rules:

1. Data type conversion is done as binary transfer.
2. If the source data type is smaller than the target data type the source value is stored into the rightmost bytes of the target variable and the leftmost bytes are set to zero.
3. If the source data type is bigger than the target data type only the rightmost bytes of the source variable are stored into the target data type.

EXAMPLE

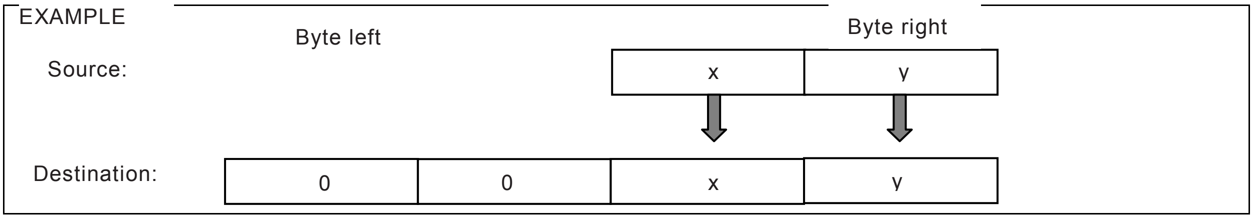


Table 24 describes the conversion functions with conversion details as result of the rules above.

Table 24 – Data type conversion of bit data types

No.	Conversion Function			Conversion Details
1	LWORD	_TO_	DWORD	Binary transfer of the rightmost bytes into the target
2	LWORD	_TO_	WORD	Binary transfer of the rightmost bytes into the target
3	LWORD	_TO_	BYTE	Binary transfer of the rightmost bytes into the target
4	LWORD	_TO_	BOOL	Binary transfer of the rightmost bit into the target
5	DWORD	_TO_	LWORD	Binary transfer into the rightmost bytes of the target, leftmost target bytes are set to zero
6	DWORD	_TO_	WORD	Binary transfer of the rightmost bytes into the target
7	DWORD	_TO_	BYTE	Binary transfer of the rightmost bytes into the target

No.	Conversion Function			Conversion Details
8	DWORD	_TO_	BOOL	Binary transfer of the rightmost bit into the target
9	WORD	_TO_	LWORD	Binary transfer into the rightmost bytes of the target, leftmost target bytes are set to zero
10	WORD	_TO_	DWORD	Binary transfer into the rightmost bytes of the target, leftmost target bytes are set to zero
11	WORD	_TO_	BYTE	Binary transfer of the rightmost bytes into the target
12	WORD	_TO_	BOOL	Binary transfer of the rightmost bit into the target
13	BYTE	_TO_	LWORD	Binary transfer into the rightmost bytes of the target, leftmost target bytes are set to zero
14	BYTE	_TO_	DWORD	Binary transfer into the rightmost bytes of the target, leftmost target bytes are set to zero
15	BYTE	_TO_	WORD	Binary transfer into the rightmost bytes of the target, leftmost target bytes are set to zero
16	BYTE	_TO_	BOOL	Binary transfer of the rightmost bit into the target
17	BYTE	_TO_	CHAR	Binary transfer
18	BOOL	_TO_	LWORD	Results in value 16#0 or 16#1
19	BOOL	_TO_	DWORD	Results in value 16#0 or 16#1
20	BOOL	_TO_	WORD	Results in value 16#0 or 16#1
21	BOOL	_TO_	BYTE	Results in value 16#0 or 16#1
22	CHAR	_TO_	BYTE	Binary transfer
23	CHAR	_TO_	WORD	Binary transfer into the rightmost bytes of the target, leftmost target bytes are set to zero
24	CHAR	_TO_	DWORD	Binary transfer into the rightmost bytes of the target, leftmost target bytes are set to zero
25	CHAR	_TO_	LWORD	Binary transfer into the rightmost bytes of the target, leftmost target bytes are set to zero
26	WCHAR	_TO_	WORD	Binary transfer
27	WCHAR	_TO_	DWORD	Binary transfer into the rightmost bytes of the target, leftmost target bytes are set to zero
28	WCHAR	_TO_	LWORD	Binary transfer into the rightmost bytes of the target, leftmost target bytes are set to zero

6.6.2.5.5 Data type conversion of bit to numeric types

These data type conversions use the following rules:

1. Data type conversion is done as binary transfer.
2. If the source data type is smaller than the target data type the source value is stored into the rightmost bytes of the target variable and the leftmost bytes are set to zero.

   EXAMPLE 1 `X: SINT:= 18; W: WORD; W:= SINT_TO_WORD(X);` and W gets `16#0012`.
3. If the source data type is bigger than the target data type only the rightmost bytes of the source variable are stored into the target data type.

   EXAMPLE 2 `W: WORD: = 16#1234; X: SINT; X:= W;` and X gets 54 (=16#34).

Table 25 describes the conversion functions with conversion details as result of the rules above.

Table 25 – Data type conversion of bit and numeric types

No.	Conversion Function			Conversion Details
1	LWORD	_TO_	LREAL	Binary transfer

No.	Conversion Function			Conversion Details
2	DWORD	_TO_	REAL	Binary transfer
3	LWORD	_TO_	LINT	Binary transfer
4	LWORD	_TO_	DINT	Binary transfer of the rightmost bytes into the target
5	LWORD	_TO_	INT	Binary transfer of the rightmost bytes into the target
6	LWORD	_TO_	SINT	Binary transfer of the rightmost byte into the target
7	LWORD	_TO_	ULINT	Binary transfer
8	LWORD	_TO_	UDINT	Binary transfer of the rightmost bytes into the target
9	LWORD	_TO_	UINT	Binary transfer of the rightmost bytes into the target
10	LWORD	_TO_	USINT	Binary transfer of the rightmost byte into the target
11	DWORD	_TO_	LINT	Binary transfer into the rightmost bytes of the target
12	DWORD	_TO_	DINT	Binary transfer
13	DWORD	_TO_	INT	Binary transfer of the rightmost bytes into the target
14	DWORD	_TO_	SINT	Binary transfer of the rightmost byte into the target
15	DWORD	_TO_	ULINT	Binary transfer into the rightmost bytes of the target
16	DWORD	_TO_	UDINT	Binary transfer
17	DWORD	_TO_	UINT	Binary transfer of the rightmost bytes into the target
18	DWORD	_TO_	USINT	Binary transfer of the rightmost byte into the target
19	WORD	_TO_	LINT	Binary transfer into the rightmost bytes of the target
20	WORD	_TO_	DINT	Binary transfer into the rightmost bytes of the target
21	WORD	_TO_	INT	Binary transfer
22	WORD	_TO_	SINT	Binary transfer of the rightmost byte into the target
23	WORD	_TO_	ULINT	Binary transfer into the rightmost bytes of the target
24	WORD	_TO_	UDINT	Binary transfer into the rightmost bytes of the target
25	WORD	_TO_	UINT	Binary transfer
26	WORD	_TO_	USINT	Binary transfer of the rightmost byte into the target
27	BYTE	_TO_	LINT	Binary transfer into the rightmost bytes of the target
28	BYTE	_TO_	DINT	Binary transfer into the rightmost bytes of the target
29	BYTE	_TO_	INT	Binary transfer into the rightmost bytes of the target
30	BYTE	_TO_	SINT	Binary transfer
31	BYTE	_TO_	ULINT	Binary transfer into the rightmost bytes of the target
32	BYTE	_TO_	UDINT	Binary transfer into the rightmost bytes of the target
33	BYTE	_TO_	UINT	Binary transfer into the rightmost bytes of the target
34	BYTE	_TO_	USINT	Binary transfer
35	BOOL	_TO_	LINT	Results in value 0 or 1
36	BOOL	_TO_	DINT	Results in value 0 or 1
37	BOOL	_TO_	INT	Results in value 0 or 1
38	BOOL	_TO_	SINT	Results in value 0 or 1
39	BOOL	_TO_	ULINT	Results in value 0 or 1
40	BOOL	_TO_	UDINT	Results in value 0 or 1
41	BOOL	_TO_	UINT	Results in value 0 or 1
42	BOOL	_TO_	USINT	Results in value 0 or 1
43	LREAL	_TO_	LWORD	Binary transfer
44	REAL	_TO_	DWORD	Binary transfer
45	LINT	_TO_	LWORD	Binary transfer
46	LINT	_TO_	DWORD	Binary transfer of the rightmost bytes into the target
47	LINT	_TO_	WORD	Binary transfer of the rightmost bytes into the target

No.	Conversion Function			Conversion Details
48	LINT	_TO_	BYTE	Binary transfer of the rightmost byte into the target
49	DINT	_TO_	LWORD	Binary transfer into the rightmost bytes of the target, rest = 0
50	DINT	_TO_	DWORD	Binary transfer
51	DINT	_TO_	WORD	Binary transfer of the rightmost bytes into the target
52	DINT	_TO_	BYTE	Binary transfer of the rightmost byte into the target
53	INT	_TO_	LWORD	Binary transfer into the rightmost bytes of the target, rest = 0
54	INT	_TO_	DWORD	Binary transfer into the rightmost bytes of the target, rest = 0
55	INT	_TO_	WORD	Binary transfer
56	INT	_TO_	BYTE	Binary transfer of the rightmost byte into the target
57	SINT	_TO_	LWORD	Binary transfer into the rightmost bytes of the target, rest = 0
58	SINT	_TO_	DWORD	Binary transfer into the rightmost bytes of the target, rest = 0
59	SINT	_TO_	WORD	Binary transfer
60	SINT	_TO_	BYTE	Binary transfer
61	ULINT	_TO_	LWORD	Binary transfer
62	ULINT	_TO_	DWORD	Binary transfer of the rightmost bytes into the target
63	ULINT	_TO_	WORD	Binary transfer of the rightmost bytes into the target
64	ULINT	_TO_	BYTE	Binary transfer of the rightmost byte into the target
65	UDINT	_TO_	LWORD	Binary transfer into the rightmost bytes of the target, rest = 0
66	UDINT	_TO_	DWORD	Binary transfer
67	UDINT	_TO_	WORD	Binary transfer of the rightmost bytes into the target
68	UDINT	_TO_	BYTE	Binary transfer of the rightmost byte into the target
69	UINT	_TO_	LWORD	Binary transfer into the rightmost bytes of the target, rest = 0
70	UINT	_TO_	DWORD	Binary transfer into the rightmost bytes of the target, rest = 0
71	UINT	_TO_	WORD	Binary transfer
72	UINT	_TO_	BYTE	Binary transfer of the rightmost byte into the target
73	USINT	_TO_	LWORD	Binary transfer into the rightmost bytes of the target, rest = 0
74	USINT	_TO_	DWORD	Binary transfer into the rightmost bytes of the target, rest = 0
75	USINT	_TO_	WORD	Binary transfer
76	USINT	_TO_	BYTE	Binary transfer

6.6.2.5.6 Data type conversion of date and time types

Table 26 shows the data type conversion of date and time types.

Table 26 – Data type conversion of date and time types

No.	Conversion Function			Conversion Details
1	LTIME	_TO_	TIME	Value range errors give an Implementer specific result and a possible loss of precision may occur.

No.	Conversion Function			Conversion Details
2	TIME	_TO_	LTIME	Value range errors give an Implementer specific result and a possible loss of precision may occur.
3	LDT	_TO_	DT	Value range errors give an Implementer specific result and a possible loss of precision may occur.
4	LDT	_TO_	DATE	Converts only the contained date, a value range error gives an Implementer specific result.
5	LDT	_TO_	LTOD	Converts only the contained time of day.
6	LDT	_TO_	TOD	Converts only the contained time of day, a possible loss of precision may occur.
7	DT	_TO_	LDT	Value range errors give an Implementer specific result and a possible loss of precision may occur.
8	DT	_TO_	DATE	Converts only the contained date, a value range error gives an Implementer specific result.
9	DT	_TO_	LTOD	Converts only the contained time of day, a value range error gives an Implementer specific result.
10	DT	_TO_	TOD	Converts only the contained time of day, a value range error gives an Implementer specific result.
11	LTOD	_TO_	TOD	Value preserving conversion
12	TOD	_TO_	LTOD	Value range errors give an Implementer specific result and a possible loss of precision may occur.

6.6.2.5.7 Data type conversion of character types

Table 27 shows the data type conversion of character types.

Table 27 – Data type conversion of character types

No.	Conversion Function			Conversion Details
1	WSTRING	_TO_	STRING	The characters which are supported by the Implementer with the data type STRING are converted; others are converted in an Implementer-dependency**.**
2	WSTRING	_TO_	WCHAR	The first character of the string is transferred; if the string is empty the target variable is undefined.
3	STRING	_TO_	WSTRING	Converts the characters of the string as defined by the implementer to the appropriate ISO/IEC 10646 (UTF-16) character.
4	STRING	_TO_	CHAR	The first character of the string is transferred; if the string is empty the target variable is undefined.
5	WCHAR	_TO_	WSTRING	Gives a string of actual size of one character.
6	WCHAR	_TO_	CHAR	The characters which are supported by the Implementer with the data type CHAR are converted, the others are converted in an Implementer specific way.
7	CHAR	_TO_	STRING	Gives a string of actual size of one character.
8	CHAR	_TO_	WCHAR	Converts a character as defined by the Implementer to the appropriate UTF-16 character.

6.6.2.5.8 Numerical and arithmetic functions

The standard graphical representation, function names, input and output variable types, and function descriptions of functions of a single numeric variable shall be as defined in Table 28. These functions shall be overloaded on the defined generic types, and can be typed. For these functions, the types of the input and output shall be the same.

The standard graphical representation, function names and symbols, and descriptions of arithmetic functions of two or more variables shall be as shown in Table 29. These functions shall be overloaded on all numeric types, and can be typed.

The accuracy of numerical functions shall be expressed in terms of one or more Implementer specific dependencies.

It is an error if the result of evaluation of one of these functions exceeds the range of values specified for the data type of the function output, or if division by zero is attempted.

Table 28 – Numerical and arithmetic functions

No.	**Description** (Function name)	**I/O type**	**Explanation**

	Graphical form
	`    +---------+`
	`* --|   **    |-- *`
	`    +---------+`
	`(*)` - Input/Output (I/O) type
	`(**)` - Function name
			Usage example in ST
			`A:= SIN(B);`
			(ST language)

	General functions		
1	`ABS(x)`	`ANY_NUM`	Absolute value
2	`SQRT(x)`	`ANY_REAL`	Square root
	Logarithmic functions		
3	`LN(x)`	`ANY_REAL`	Natural logarithm
4	`LOG(x)`	`ANY_REAL`	Logarithm base 10
5	`EXP(x)`	`ANY_REAL`	Natural exponential
	Trigonometric functions		
6	`SIN(x)`	`ANY_REAL`	Sine of input in radians
7	`COS(x)`	`ANY_REAL`	Cosine in radians
8	`TAN(x)`	`ANY_REAL`	Tangent in radians
9	`ASIN(x)`	`ANY_REAL`	Principal arc sine
10	`ACOS(x)`	`ANY_REAL`	Principal arc cosine
11	`ATAN(x)`	`ANY_REAL`	Principal arc tangent

12	`ATAN2(y, x)`
	`          +-------+`
	`          | ATAN2 |`
	`ANY_REAL--|Y      |--ANY_REAL`
	`ANY_REAL--|X      |`
	`          +-------+`
		`ANY_REAL`	Angle in between the positive x-axis of a plane and the point given by the coordinates (x, y) on it. The angle is positive for counter-clockwise angles (upper half-plane, y > 0), and negative for clockwise angles (lower half-plane, y < 0).

Table 29 – Arithmetic functions

No. a,b	Description	Name	Symbol (Operator)	Explanation
	Graphical form			**Usage example in ST**

	`      +-----+`
	`ANY_NUM --| *** |-- ANY_NUM`
	`ANY_NUM --|     |`
	`   .    --|     |`
	`   .    --|     |`
	`ANY_NUM --|     |`
	`      +-----+`
	
	`(***)` - Name or Symbol
				as function call:
				`A:= ADD(B, C, D);`
				or
				as operator (symbol)
				`A:= B + C + D;`

	Extensible arithmetic functions			
1 [c]	Addition	ADD	+	OUT:= IN1 + IN2 +... + INn
2	Multiplication	MUL	*	OUT:= IN1 * IN2 *... * INn
	Non-extensible arithmetic functions			
3 [c]	Subtraction	SUB	-	OUT:= IN1 - IN2
4 [d]	Division	DIV	/	OUT:= IN1 / IN2
5 [e]	Modulo	MOD		OUT:= IN1 modulo IN2
6 [f]	Exponentiation	EXPT	**	OUT:= $\mathrm{IN1}^{\mathrm{IN2}}$
7 [g]	Move	MOVE	:=	OUT:= IN

NOTE 1 Non-blank entries in the Symbol column are suitable for use as operators in textual languages.

NOTE 2 The notations IN1, IN2, ..., INn refer to the inputs in top-to-bottom order; OUT refers to the output.

NOTE 3 Usage examples and descriptions are given in the ST language.

[a] When the representation of a function is supported with a name, this is indicated by the suffix "n" in the compliance statement.
For example, "1n" represents the notation "ADD".

[b] When the representation of a function is supported with a symbol, this is indicated by the suffix "s" in the compliance statement. For example, "1s" represents the notation "+".

[c] The generic type of the inputs and outputs of these functions is ANY_MAGNITUDE.

[d] The result of division of integers shall be an integer of the same type with truncation toward zero, for instance, 7/3 = 2 and (-7)/3 = -2.

[e] IN1 and IN2 shall be of generic type ANY_INT for this function. The result of evaluating this MOD function shall be the equivalent of executing the following statements in the ST:

```
IF (IN2 = 0)
  THEN OUT:=0;
  ELSE OUT:=IN1 - (IN1/IN2)*IN2;
END_IF
```

[f] IN1 shall be of type ANY_REAL, and IN2 of type ANY_NUM for this EXPT function. The output shall be of the same type as IN1.

[g] The MOVE function has exactly one input (IN) of type ANY and one output (OUT) of type ANY.

6.6.2.5.9 Bit string and bitwise Boolean functions

The standard graphical representation, function names and descriptions of shift functions for a single bit-string variable shall be as defined in Table 30. These functions shall be overloaded on all bit-string types, and can be typed.

The standard graphical representation, function names and symbols, and descriptions of bitwise Boolean functions shall be as defined in Table 31. These functions shall be extensible, except for NOT, and overloaded on all bit-string types, and can be typed.

Table 30 – Bit shift functions

No.	Description	Name	Explanation
	Graphical form		**Usage example** [a]

	`           +------+`
	`           | *** |`
	`ANY_BIT --|IN    |-- ANY_BIT`
	`ANY_INT --|N     |`
	`           +------+`
	(***) - Function Name
			`A:= SHL(IN:=B, N:=5);`
			(ST language)

1	Shift left	SHL	OUT:= IN left-shifted by N bits, zero-filled on right
2	Shift right	SHR	OUT:= IN right-shifted by N bits, zero-filled on left
3	Rotation left	ROL	OUT:= IN left-rotated by N bits, circular
4	Rotation right	ROR	OUT:= IN right-rotated by N bits, circular

NOTE 1 The notation OUT refers to the function output.

EXAMPLE

```
        IN:= 2#0001_1001 of type BYTE, N = 3

        SHL(IN, 3) = 2#1100_1000
        SHR(IN, 3) = 2#0000_0011
        ROL(IN, 3) = 2#1100_1000
        ROR(IN, 3) = 2#0010_0011
```

NOTE 2 IN of type BOOL (one bit) does not make sense.

[a] It is an error if the value of the N input is less than zero.

Table 31 – Bitwise Boolean functions

No. a,b	Description	Name	Symbol	Explanation (NOTE 3)

	Graphical form
	`           +------+`
	`ANY_BIT --| *** |-- ANY_BIT`
	`ANY_BIT --|      |`
	`   :    --|      |`
	`   :    --|      |`
	`ANY_BIT --|      |`
	`           +------+`
	(***) - Name or symbol
				Usage examples (NOTE 5)
				`A:= AND(B, C, D);`
				or
				`A:= B & C & D;`

1	And	AND	& (NOTE 1)	`OUT:= IN1 & IN2 &... & INn`
2	Or	OR	>=1 (NOTE 2)	`OUT:= IN1 OR IN2 OR... OR INn`
3	Exclusive Or	XOR	=2k+1 (NOTE 2)	`OUT:= IN1 XOR IN2 XOR... XOR INn`
4	Not	NOT		`OUT:= NOT IN1 (NOTE 4)`

NOTE 1 This symbol is suitable for use as an operator in textual languages, as shown in Table 68 and Table 71.

NOTE 2 This symbol is not suitable for use as an operator in textual languages.

NOTE 3 The notations IN1, IN2,..., INn refer to the inputs in top-to-bottom order; OUT refers to the output.

NOTE 4 Graphic negation of signals of type BOOL can also be accomplished.

NOTE 5 Usage examples and descriptions are given in the ST language.

[a] When the representation of a function is supported with a name, this shall be indicated by the suffix “n” in the compliance statement. For example, “1n” represents the notation “AND”.

[b] When the representation of a function is supported with a symbol, this shall be indicated by the suffix “s” in the compliance statement. For example, “1s” represents the notation “&”.

6.6.2.5.10 Selection and comparison functions

Selection and comparison functions shall be overloaded on all data types. The standard graphical representations, function names and descriptions of selection functions shall be as shown in Table 32.

The standard graphical representation, function names and symbols, and descriptions of comparison functions shall be as defined in Table 33. All comparison functions (except NE) shall be extensible.

Comparisons of bit string data shall be made bitwise from the leftmost to the rightmost bit, and shorter bit strings shall be considered to be filled on the left with zeros when compared to longer bit strings; that is, comparison of bit string variables shall have the same result as comparison of unsigned integer variables.

Table 32 – Selection functions [d]

No.	Description	Name	Graphical form	Explanation/ Example
1	Move [a, d] (assignment)	MOVE	+--------+ | MOVE | ANY --| |- ANY +--------+	OUT:= IN

2	Binary selection [d]	SEL	+-------+
			| SEL |
			BOOL --|G |- ANY
			ANY --|IN0 |
			ANY --|IN1 |
			+-------+
				OUT:= IN0 if G = 0
				OUT:= IN1 if G = 1
				EXAMPLE 1
				A:= SEL (G := 0,
				IN0:= X,
				IN1:= 5);

3	Extensible maximum function	MAX	+-------+
			| MAX |
			ANY_ELEMENTARY --| |- ANY_ELEMENTARY
			: --| |
			ANY_ELEMENTARY --| |
			+-------+
				OUT:=
				MAX(IN1, IN2, ..., INn);
				EXAMPLE 2
				A:= MAX(B, C , D);

4	Extensible minimum function	MIN	+-------+ | MIN | ANY_ELEMENTARY --| |- ANY_ELEMENTARY : --| | ANY_ELEMENTARY --| | +-------+	OUT:= MIN (IN1, IN2,..., Nn) EXAMPLE 3 A:= MIN(B, C, D);

5	Limiter	LIMIT	+-------+
			| LIMIT |
			ANY_ELEMENTARY --|MN |- ANY_ELEMENTARY
			ANY_ELEMENTARY --|IN |
			ANY_ELEMENTARY --|MX |
			+-------+
				OUT:= MIN
				(MAX(IN, MN),MX);
				EXAMPLE 4
				A:= LIMIT(IN:= B,
				MN:= 0,
				MX:= 5);

<table>
<tr><td>6</td><td>Extensible [b, c, d, e]
multiplexer</td><td>MUX</td><td>+--------+
| MUX |
ANY_ELEMENTARY --|K |- ANY_ELEMENTARY
ANY_ELEMENTARY --| |
ANY_ELEMENTARY --| |
+--------+</td><td>a, b, c:
Select one of N inputs depending on input K

EXAMPLE 5
A:= MUX(0, B, C, D);
would have the same effect as
A:= B;</td></tr>
</table>

NOTE 1 The notations IN1, IN2,..., INn refer to the inputs in top-to-bottom order; OUT refers to the output.

NOTE 2 Usage examples and descriptions are given in the ST language.

[a] The MOVE function has exactly one input IN of type ANY and one output OUT of type ANY.

[b] The unnamed inputs in the MUX function shall have the default names IN0, IN1,..., INn-1 in top-to-bottom order, where n is the total number of these inputs. These names may, but need not, be shown in the graphical representation.

[c] The MUX function can be typed in the form MUX_*_**, where * is the type of the K input and ** is the type of the other inputs and the output.

[d] It is allowed, but not required, that the Implementer support selection among variables of user-defined data types, in order to claim compliance with this feature.

[e] It is an error if the actual value of the K input of the MUX function is not within the range {0 … n-1}.

Table 33 – Comparison functions

<table>
<tr><th>No.</th><th>Description</th><th>Name [a]</th><th>Symbol [b]</th><th>Explanation
(For 2 or more operands extensible)</th></tr>
<tr><td></td><td><b>Graphical form</b></td><td></td><td></td><td><b>Usage examples</b></td></tr>
<tr><td></td><td>+-----+
ANY_ELEMENTARY --| *** |-- BOOL
: --| |
ANY_ELEMENTARY --| |
+-----+

(***) Name or Symbol</td><td></td><td></td><td>A:= GT(B, C, D); // Function name
or
A:= (B>C) & (C>D); // Symbol</td></tr>
<tr><td>1</td><td>Decreasing sequence</td><td>GT</td><td>&gt;</td><td>OUT:= (IN1>IN2)& (IN2>IN3) &.. & (INn-1 &gt; INn)</td></tr>
<tr><td>2</td><td>Monotonic sequence:</td><td>GE</td><td>&gt;=</td><td>OUT:= (IN1>=IN2)&(IN2>=IN3)&.. & (INn-1 &gt;= INn)</td></tr>
<tr><td>3</td><td>Equality</td><td>EQ</td><td>=</td><td>OUT:= (IN1=IN2)&(IN2=IN3) &.. & (INn-1 = INn)</td></tr>
<tr><td>4</td><td>Monotonic sequence</td><td>LE</td><td>&lt;=</td><td>OUT:= (IN1<=IN2)&(IN2<=IN3)&.. & (INn-1 &lt;= INn)</td></tr>
<tr><td>5</td><td>Increasing sequence</td><td>LT</td><td>&lt;</td><td>OUT:= (IN1<IN2)& (IN2<IN3) &.. & (INn-1 &lt; INn)</td></tr>
<tr><td>6</td><td>Inequality</td><td>NE</td><td>&lt;&gt;</td><td>OUT:= (IN1<>IN2) (non-extensible)</td></tr>
</table>

NOTE 1 The notations IN1, IN2,..., INn refer to the inputs in top-to-bottom order; OUT refers to the output.

NOTE 2 All the symbols shown in this table are suitable for use as operators in textual languages.

NOTE 3 Usage examples and descriptions are given in the ST language.

NOTE 4 Standard comparison functions may be defined language dependant too e.g. ladder.

[a] When the representation of a function is supported with a name, this shall be indicated by the suffix “n” in the compliance statement. For example, “1n” represents the notation “GT”.

[b] When the representation of a function is supported with a symbol, this shall be indicated by the suffix “s” in the compliance statement. For example, “1s” represents the notation “>“.

6.6.2.5.11 Character string functions.

Table 33 shall be applicable to character strings. Instead of a single-character string a variable of data type CHAR or WCHAR respectively may be used.

For the purposes of comparison of two strings of unequal length, the shorter string shall be considered to be extended on the right to the length of the longer string by characters with the value zero. Comparison shall proceed from left to right, based on the numeric value of the character codes in the character set.

EXAMPLE

The character string 'Z' is greater than the character string 'AZ' ('Z' > 'A'),
and 'AZ' is greater than 'ABC' ('A' = 'A' and 'Z' > 'B').

The standard graphical representations, function names and descriptions of additional functions of character strings shall be as shown in Table 34.For the purpose of these operations, character positions within the string shall be considered to be numbered 1, 2, ..., L, beginning with the leftmost character position, where L is the length of the string.

It shall be an error if:

- the actual value of any input designated as ANY_INT in Table 34 is less than zero;
- the evaluation of the function results in an attempt to (1) access a non-existent character position in a string, or (2) produce a string longer than the Implementer specific maximum string length;
- the arguments of data type STRING or CHAR and arguments of data type WSTRING or WCHAR are mixed at the same function.

Table 34 – Character string functions

No.	Description	Graphical form	Example

1	String length	`            +---------+`
		`ANY_STRING--|   LEN   |-- ANY_INT`
		`            +---------+`
			String length
			A:= LEN('ASTRING');
			..is equivalent to A:= 7;

2	Left	`            +---------+`
		`ANY_STRING--|   LEN   |-- ANY_INT`
		`            +---------+`
			Leftmost L characters of IN
			A:= LEFT(IN:='ASTR', L:=3);
			is equivalent to A:= 'AST';

3	Right	`            +---------+`
		`
			RIGHT |`
			`ANY_STRING--|IN       |-- ANY_STRING`
			`ANY_INT   --|L        |`
			`            +---------+`
				Rightmost L characters of IN
				A:= RIGHT(IN:='ASTR', L:=3);
				is equivalent to A:= 'STR';

4	Middle	`            +---------+`
		`
			MID |`
			`ANY_STRING--|IN       |-- ANY_STRING`
			`ANY_INT   --|L        |`
			`ANY_INT   --|P        |`
			`            +---------+`
				L characters of IN, beginning at the P-th character position
				A:= MID(IN:='ASTR', L:=2, P:=2);
				is equivalent to A:= 'ST';

No.	Description	Graphical form	Example

5	Extensible concatenation	`            +---------+`
		`            | CONCAT  |`
		` ANY_CHARS--|         |-- ANY_STRING`
		`      :  --|         |`
		` ANY_CHARS--|         |`
		`            +---------+`
			Extensible concatenation
			`A:= CONCAT('AB','CD','E');`
			is equivalent to `A:= 'ABCDE';`

6	Insert	`            +---------+`
		`            | INSERT  |`
		`ANY_STRING--|IN1      |-- ANY_STRING`
		`ANY_CHARS --|IN2      |`
		`ANY_INT-----|P        |`
		`            +---------+`
			Insert `IN2` into `IN1` after the `P`-th character position
			`A:= INSERT(IN1:='ABC', IN2:='XY', P=2);`
			is equivalent to `A:= 'ABXYC';`

7	Delete	`            +---------+`
		`            | DELETE  |`
		`ANY_STRING--|IN       |-- ANY_STRING`
		`ANY_INT   --|L        |`
		`ANY_INT   --|P        |`
		`            +---------+`
			`L` characters of `IN`, beginning at the `P`-th character position
			`A:= DELETE(IN:='ABXYC', L:=2, P:=3);`
			is equivalent to `A:= 'ABC';`

8	Replace	`            +---------+`
		`            | REPLACE |`
		`ANY_STRING--|IN1      |-- ANY_STRING`
		`ANY_CHARS --|IN2      |`
		`ANY_INT   --|L        |`
		`ANY_INT   --|P        |`
		`            +---------+`
			Replace `L` characters of `IN1` by `IN2`, starting at the `P`-th character position.
			`A:= REPLACE(IN1:='ABCDE', IN2:='X', L:=2, P:=3);`
			is equivalent to `A:= 'ABXE';`

9	Find	`            +---------+`
		`            |  FIND   |`
		`ANY_STRING--|IN1      |-- ANY_INT`
		`ANY_CHARS --|IN2      |`
		`            +---------+`
			Find the character position of the beginning of the first occurrence of `IN2` in `IN1`. If no occurrence of `IN2` is found, then `OUT`:= 0.
			`A:= FIND(IN1:='ABCBC', IN2:='BC');`
			is equivalent to `A:= 2;`

NOTE 1 The examples in this table are given in the ST language. NOTE 2 All inputs of `CONCAT` are of `ANY_CHARS` i.e. can also be of type `CHAR` or `WCHAR`. NOTE 3 The input `IN2` of the functions `INSERT`, `REPLACE`, `FIND` are of `ANY_CHARS` i.e. can also be of type `CHAR` or `WCHAR`.			

6.6.2.5.12 Date and duration functions

In addition to the comparison and selection functions, the combinations of input and output time and duration data types shown in Table 35 shall be allowed with the associated functions.

It shall be an error if the result of evaluating one of these functions exceeds the Implementer specific range of values for the output data type.

Table 35 – Numerical functions of time and duration data types

No.	Description (function name)	Symbol	IN1	IN2	OUT
1a	`ADD`	+	`TIME, LTIME`	`TIME, LTIME`	`TIME, LTIME`
1b	`ADD_TIME`	+	`TIME`	`TIME`	`TIME`
1c	`ADD_LTIME`	+	`LTIME`	`LTIME`	`LTIME`
2a	`ADD`	+	`TOD, LTOD`	`LTIME`	`TOD, LTOD`
2b	`ADD_TOD_TIME`	+	`TOD`	`TIME`	`TOD`
2c	`ADD_LTOD_LTIME`	+	`LTOD`	`LTIME`	`LTOD`
3a	`ADD`	+	`DT, LDT`	`TIME, LTIME`	`DT, LDT`
3b	`ADD_DT_TIME`	+	`DT`	`TIME`	`DT`

No.	Description (function name)	Symbol	IN1	IN2	OUT
3c	ADD_LDT_LTIME	+	LDT	LTIME	LDT
4a	SUB	-	TIME, LTIME	TIME, LTIME	TIME, LTIME
4b	SUB_TIME	-	TIME	TIME	TIME
4c	SUB_LTIME	-	LTIME	LTIME	LTIME
5a	SUB	-	DATE	DATE	TIME
5b	SUB_DATE_DATE	-	DATE	DATE	TIME
5c	SUB_LDATE_LDATE	-	LDATE	LDATE	LTIME
6a	SUB	-	TOD, LTOD	TIME, LTIME	TOD, LTOD
6b	SUB_TOD_TIME	-	TOD	TIME	TOD
6c	SUB_LTOD_LTIME	-	LTOD	LTIME	LTOD
7a	SUB	-	TOD, LTOD	TOD, LTOD	TIME, LTIME
7b	SUB_TOD_TOD	-	TOD	TOD	TIME
7c	SUB_TOD_TOD	-	LTOD	LTOD	LTIME
8a	SUB	-	DT, LDT	TIME, LTIME	DT, LDT
8b	SUB_DT_TIME	-	DT	TIME	DT
8c	SUB_LDT_LTIME	-	LDT	LTIME	LDT
9a	SUB	-	DT, LDT	DT, LDT	TIME, LTIME
9b	SUB_DT_DT	-	DT	DT	TIME
9c	SUB_LDT_LDT	-	LDT	LDT	LTIME
10a	MUL	*	TIME, LTIME	ANY_NUM	TIME, LTIME
10b	MUL_TIME	*	TIME	ANY_NUM	TIME
10c	MUL_LTIME	*	LTIME	ANY_NUM	LTIME
11a	DIV	/	TIME, LTIME	ANY_NUM	TIME, LTIME
11b	DIV_TIME	/	TIME	ANY_NUM	TIME
11c	DIV_LTIME	/	LTIME	ANY_NUM	LTIME
NOTE These standard functions support overloading but only within the both sets of data types (TIME, DT, DATE, TOD) and (LTIME, LDT, DATE, LTOD).					

EXAMPLE

The ST language statements

```
X:= DT#1986-04-28-08:40:00;
Y:= DT_TO_TOD(X);
W:= DT_TO_DATE(X);
```

have the same result as the statement with “extracted” data.

```
X:= DT#1986-04-28-08:40:00;
Y:= TIME_OF_DAY#08:40:00;
W:= DATE#1986-04-28;
```

Concatenate and split functions as shown in Table 36 are defined to handle date and time. Additionally, a function to get the day of the week is defined.

It shall be an error if the result of evaluating one of these functions exceeds the Implementer specific range of values for the output data type.

Table 36 – Additional functions of time data types CONCAT and SPLIT

No.	Description	Graphical form	Example
	Concatenate time data types		

<table>
<tr><th>No.</th><th>Description</th><th>Graphical form</th><th>Example</th></tr>
<tr><td>1a</td><td>CONCAT_DATE
_TOD</td><td>   +-----------------+
        | CONCAT_DATE_TOD |
DATE --|DATE |--DT
 TOD --|TOD |
        +-----------------+</td><td>Concatenate a date:
VAR
 myD: DATE;
END_VAR
myD:= CONCAT_DATE_TOD
 (D#2010-03-12, TOD#12:30:00);</td></tr>
<tr><td>1b</td><td>CONCAT_DATE
_LTOD</td><td>   +-----------------+
        | CONCAT_DATE_LTOD|
DATE --|DATE |--LDT
LTOD --|LTOD |
        +-----------------+</td><td>Concatenate a date and a time of day:
VAR
 myD: DATE;
END_VAR
myD:= CONCAT_DATE_LTOD
 (D#2010-03-12,
   TOD#12:30:12.1223452);</td></tr>
<tr><td>2</td><td>CONCAT_DATE</td><td>   +-------------+
           | CONCAT_DATE |
ANY_INT --|YEAR |--DATE
ANY_INT --|MONTH |
ANY_INT --|DAY |
           +-------------+</td><td>Concatenate a date and time of day:
VAR
 myD: DATE;
END_VAR
myD:= CONCAT_DATE (2010,3,12);</td></tr>
<tr><td>3a</td><td>CONCAT_TOD</td><td>   +-------------+
           | CONCAT_TOD |
ANY_INT --|HOUR |--TOD
ANY_INT --|MINUTE |
ANY_INT --|SECOND |
ANY_INT --|MILLISECOND |
           +-------------+</td><td>Concatenate a time of day:
VAR
 myTOD: TOD;
END_VAR
myTD:= CONCAT_TOD (16,33,12,0);</td></tr>
<tr><td>3b</td><td>CONCAT_LTOD</td><td>   +-------------+
           | CONCAT_LTOD |
ANY_INT --|HOUR |--LTOD
ANY_INT --|MINUTE |
ANY_INT --|SECOND |
ANY_INT --|MILLISECOND |
           +-------------+</td><td>Concatenate a time of day:
VAR
 myTOD: LTOD;
END_VAR
myTD:= CONCAT_TOD (16,33,12,0);</td></tr>
<tr><td>4a</td><td>CONCAT_DT</td><td>   +-------------+
           | CONCAT_DT |
ANY_INT --|YEAR |--DT
ANY_INT --|MONTH |
ANY_INT --|DAY |
ANY_INT --|HOUR |
ANY_INT --|MINUTE |
ANY_INT --|SECOND |
ANY_INT --|MILLISECOND |
           +-------------+</td><td>Concatenate a time of day:
VAR
 myDT: DT;
 Day: USINT;
END_VAR
Day := 17;
myDT:= CONCAT_DT
       (2010,3,Day,12,33,12,0);</td></tr>
<tr><td>4b</td><td>CONCAT_LDT</td><td>   +-------------+
           | CONCAT_LDT |
ANY_INT --|YEAR |--LDT
ANY_INT --|MONTH |
ANY_INT --|DAY |
ANY_INT --|HOUR |
ANY_INT --|MINUTE |
ANY_INT --|SECOND |
ANY_INT --|MILLISECOND |
           +-------------+</td><td>Concatenate a time of day:
VAR
 myDT: LDT;
 Day: USINT;
END_VAR
Day := 17;
myDT:= CONCAT_LDT
       (2010,3,Day,12,33,12,0);</td></tr>
</table>

<table>
<tr><th>No.</th><th>Description</th><th>Graphical form</th><th>Example</th></tr>
<tr><td></td><td><b>Split time data types</b></td><td></td><td></td></tr>
<tr><td>5</td><td>SPLIT_DATE</td><td>   +-------------+
        | SPLIT_DATE |
  DATE--|IN   YEAR|-- ANY_INT
        | MONTH|-- ANY_INT
        | DAY|-- ANY_INT
        +-------------+
See NOTE 2</td><td>Split a date:
VAR
myD: DATE:= DATE#2010-03-10;
 myYear: UINT;
 myMonth,
 myDay: USINT;
END_VAR
SPLIT_DATE
(myD,myYear,myMonth,myDay);</td></tr>
<tr><td>6a</td><td>SPLIT_TOD</td><td>   +-------------+
        | SPLIT_TOD |
   TOD--|IN   HOUR|-- ANY_INT
        | MINUTE|-- ANY_INT
        | SECOND|-- ANY_INT
        | MILLISECOND|-- ANY_INT
        +-------------+
See NOTE 2</td><td>Split a time of day:
VAR myTOD: TOD:= TOD#14:12:03;
 myHour, myMin, mySec: USINT;
 myMilliSec: UINT;
END_VAR
SPLIT_TOD(myTOD, myHour, myMin, mySec,myMilliSec);</td></tr>
<tr><td>6b</td><td>SPLIT_LTOD</td><td>   +-------------+
        | SPLIT_LTOD |
  LTOD--|IN   HOUR|-- ANY_INT
        | MINUTE|-- ANY_INT
        | SECOND|-- ANY_INT
        | MILLISECOND|-- ANY_INT
        +-------------+
See NOTE 2</td><td>Split a time of day:
VAR myTOD: LTOD:= TOD#14:12:03;
 myHour, myMin, mySec: USINT;
 myMilliSec: UINT;
END_VAR
SPLIT_TOD(myTOD, myHour, myMin, mySec,myMilliSec);</td></tr>
<tr><td>7a</td><td>SPLIT_DT</td><td>   +-------------+
        | SPLIT_DT |
    DT--|IN   YEAR|-- ANY_INT
        | MONTH|-- ANY_INT
        | DAY|-- ANY_INT
        | HOUR|-- ANY_INT
        | MINUTE|-- ANY_INT
        | SECOND|-- ANY_INT
        | MILLISECOND|-- ANY_INT
        +-------------+
See NOTE 2</td><td>Split a date:
VAR myDT: DT
  := DT#2010-03-10-14:12:03:00;
 myYear, myMilliSec: UINT;
 myMonth, myDay, myHour, myMin,
  mySec: USINT;
END_VAR
SPLIT_DT(myDT, myYear, myMonth, myDay,
myHour,myMin,mySec,myMilliSec);</td></tr>
<tr><td>7b</td><td>SPLIT_LDT</td><td>   +-------------+
        | SPLIT_LDT |
   LDT--|IN   YEAR|-- ANY_INT
        | MONTH|-- ANY_INT
        | DAY|-- ANY_INT
        | HOUR|-- ANY_INT
        | MINUTE|-- ANY_INT
        | SECOND|-- ANY_INT
        | MILLISECOND|-- ANY_INT
        +-------------+
See NOTE 2</td><td>Split a date:
VAR myDT: LDT
  := DT#2010-03-10-14:12:03:00;
 myYear, myMilliSec: UINT;
 myMonth, myDay, myHour, myMin,
  mySec: USINT;
END_VAR
SPLIT_DT(myDT, myYear, myMonth, myDay,
myHour,myMin,mySec,myMilliSec);</td></tr>
<tr><td></td><td><b>Get day of the week</b></td><td></td><td></td></tr>
<tr><td>8</td><td>DAY_OF_WEEK</td><td>   +-------------+
        | DAY_OF_WEEK |
  DATE--|IN |- ANY_INT
        +-------------+
See NOTE 2</td><td>Get the day of week:
VAR myD: DATE:= DATE#2010-03-10;
 myDoW: USINT;
END_VAR
myDoW:= DAY_OF_WEEK(myD);</td></tr>
<tr><td></td><td colspan="3">The function DAY_OF_WEEK returns 0 for Sunday, 1 for Monday, …, 6 for Saturday</td></tr>
<tr><td colspan="4">NOTE 1 The data type at input YEAR is at least a 16 bit type to be able to support a valid year value.
NOTE 2 The Implementer specifies the provided data types for the ANY_INT outputs.
NOTE 3 The Implementer may define additional inputs or outputs according to the supported precision, e.g. microsecond and nanosecond.</td></tr>
</table>

6.6.2.5.13 Functions for endianess conversion

The endianess conversion functions convert to and from the Implementer specific, internally used endianess of the PLC from and to the requested endianess.

Endianess is the ordering of the bytes within a longer data type or variable.

The data values of data in big endian are placed in the memory locations beginning with the leftmost byte first and the rightmost byte last.

The data values of data in little endian are placed in the memory locations beginning with the rightmost byte first and the leftmost byte last.

Independently of the endianess the bit offset 0 addresses the rightmost bit of a data type.

Using the partial access with the lower number returns the lower value part independently of the specified endianess.

EXAMPLE 1 Endianess

```
TYPE D: DWORD:= 16#1234_5678; END_TYPE;
```

Memory layout
for big endian: 16#12, 16#34, 16#56, 16#78
for little endian: 16#78, 16#56, 16#34, 16#12.

EXAMPLE 2 Endianess

```
TYPE L: ULINT:= 16#1234_5678_9ABC_DEF0; END_TYPE;
```

Memory layout
for big endian: 16#12, 16#34,16#56, 16#78, 16#9A, 16#BC, 16#DE, 16#F0
for little endian: 16#F0, 16#DE, 16#BC, 16#9A, 16#78, 16#56, 16#34, 16#12.

The following data types shall be supported as inputs or outputs of the endianess conversion functions:

- `ANY_INT` with size greater than or equal to 16 bits
- `ANY_BIT` with size greater than or equal to 16 bits
- `ANY_REAL`
- `WCHAR`
- `TIME`
- arrays of these data types
- structures containing components of these data types

Other data types are not converted but may be contained in the structures to convert.

Table 37 shows the functions for endianess conversion.

Table 37 – Function for endianess conversion

<table>
<tr><th>No.</th><th>Description</th><th>Graphical form</th><th>Textual form</th></tr>
<tr><td>1</td><td>TO_BIG_ENDIAN</td><td>+------------------+
| TO_BIG_ENDIAN |
ANY --|IN |--ANY
+------------------+</td><td>Conversion to big endian data format
A:= TO_BIG_ENDIAN(B);</td></tr>
<tr><td>2</td><td>TO_LITTLE_ENDIAN</td><td>+------------------+
|.TO_LITTLE_ENDIAN |
ANY --|IN   . |--ANY
+------------------+</td><td>Conversion to little endian data format
B:= TO_LITTLE_ENDIAN(A);</td></tr>
<tr><td>3</td><td>BIG_ENDIAN_TO</td><td>+------------------+
| FROM_BIG_ENDIAN |
ANY --|IN |--ANY
+------------------+</td><td>Conversion from big endian data format
A:= FROM_BIG_ENDIAN(B);</td></tr>
<tr><td>4</td><td>LITTLE_ENDIAN_TO</td><td>+------------------+
|FROM_LITTLE_ENDIAN|
ANY --|IN |--ANY
+------------------+</td><td>Conversion from little endian data format
A:= FROM_LITTLE_ENDIAN(B);</td></tr>
<tr><td colspan="4">The data types on the input and output side shall be the same.
NOTE   In the case the variable is already in the requested data format, the function does not change the data representation.</td></tr>
</table>

6.6.2.5.14 Functions of enumerated data types

The selection and comparison functions listed in Table 38 can be applied to inputs which are of an enumerated data type.

Table 38 – Functions of enumerated data types

No.	Description/ Function name	Symbol	Feature No. x in Table y
1	SEL		Feature 2, Table 32
2	MUX		Feature 6, Table 32
3[a]	EQ	=	Feature 3, Table 33
4[a]	NE	<>	Feature 6, Table 33
NOTE The provisions of Notes 1 and 2 of Table 33 apply to this table.			
[a] The provisions of footnotes a and b of Table 33 apply to this feature.			

6.6.2.5.15 Validate functions

The validate functions check if the given input parameter contains a valid value.

The overloaded function IS_VALID is defined for the data types REAL and LREAL. In the case the real number is Not-a-Number (NaN) or infinite (+Inf, -Inf) the result of the validate function is FALSE.

The Implementer may support additional data types with the validate function IS_VALID. The result of these extensions is Implementer specific.

The overloaded function IS_VALID_BCD is defined for the data types BYTE, WORD, DWORD, and LWORD. In the case the value does not conform to the BCD definition, the result of the validate function is FALSE.

Table 39 shows the list of features of the the validate functions.

Table 39 – Validate functions

No.	Function	Graphical form	Example

1	`IS_VALID`	`            +--------------+`
		`          . |   IS_VALID   |`
		`ANY_REAL--|IN            |--BOOL`
		`            +--------------+`
			Validity of a REAL
			`VAR R: REAL; END_VAR`
			`IF IS_VALID(R) THEN ...`

2	`IS_VALID_BCD`	`            +--------------+`
		`            | IS_VALID_BCD |`
		`-ANY_BIT--|IN            |--BOOL`
		`            +--------------+`
			Validity test for a BCD word
			`VAR W: WORD; END_VAR`
			`IF IS_VALID_BCD(W) THEN ...`

6.6.3 Function blocks

6.6.3.1 General

A function block is a programmable organization unit (POU) which represents for the purpose of modularization and structuring a well-defined portion of the program.

The function block concept is realized by the function block type and the function block instance:

- Function block type consists of
  - the definition of a data structure partitioned into input, output, and internal variables; and
  - a set of operations to be performed upon the elements of the data structure when an instance of the function block type is called.
- Function block instance
  - It is a multiple, named usage (instances) of a function block type.
  - Each instance shall have an associated identifier (the instance name), and a data structure containing the static input, output, and internal variables.

    The static variables shall keep their value from one execution of the function block instance to the next; therefore, call of a function block instance with the same input parameters need not always yield the same output values.

The common features of POUs apply for function blocks.

- Object oriented function block

  The function block can be extended by a set of object oriented features.

  The object oriented function block is also a superset of the class.

6.6.3.2 Function block type declaration

The function block type shall be declared in a similar manner as described for functions.

The features of the function block type declaration are defined in Table 40:

1) The keyword FUNCTION_BLOCK, followed by an identifier specifying the name of the function block being declared.
2) A set of operations that constitutes the body.
3) The terminating keyword END_FUNCTION_BLOCK after the function block body.
4) The construct with VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT, if required, specifying the names and types of the variables.

5) The values of the variables which are declared via a VAR_EXTERNAL construct can be modified from within the function block.

6) The values of the constants which are declared via a VAR_EXTERNAL CONSTANT construct cannot be modified from within the function block.

7) The variable-length arrays may be used as VAR_IN_OUT.

8) The input, output and static variables may be initialized.

9) EN/ENO inputs and outputs shall be declared similar as input and output variables.

The following features are specific for function blocks (different to functions):

10) A VAR...END_VAR construct and also the VAR_TEMP...END_VAR, if required, specifying the names and types of the function block's internal variables. In contrast to functions the variables declared in the VAR section are static.

11) Variables of the VAR section (static) may be declared PUBLIC or PRIVATE. The access specifier PRIVATE is default. A public variable may be accessed from outside the FB using the syntax like the access to FB outputs.

12) The RETAIN or NON_RETAIN qualifier can be used for and input, output, and internal variables of a function block, as shown in Table 40.

13) In textual declarations, the R_EDGE and F_EDGE qualifiers shall 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 in this function block to perform the required edge detection. For an example of this construction, see Table 40.

14) In graphical declarations, the falling and rising edges detection the construction illustrated shall be used. When the character set is used, the “greater than” ‘>‘ or “less than” ‘<‘ character shall be in line with the edge of the function block.

15) The asterisk ‘*‘ notation as defined in Table 16 may be used in the declaration of internal variables of a function block.

16) If the generic data types are used in the type declaration of standard function block inputs and outputs, then the rules for inferring the actual types of the outputs of such function block types shall be part of the function block type definition.

17) Instances of other function blocks, classes and object oriented function blocks can be declared in all variable sections except the VAR_TEMP section.

18) A function block instance declared inside a function block type should not use the same name as a function of the same name scope to avoid ambiguities.

Table 40 – Function block type declaration

No.	Description	Example
1	Declaration of function block type FUNCTION_BLOCK ... END_FUNCTION_BLOCK	FUNCTION_BLOCK myFB ... END_FUNCTION_BLOCK
2a	Declaration of inputs VAR_INPUT ... END_VAR	VAR_INPUT IN: BOOL; T1: TIME; END_VAR
2b	Declaration of outputs VAR_OUTPUT ... END_VAR	VAR_OUTPUT OUT: BOOL; ET_OFF: TIME; END_VAR
2c	Declaration of in-outs VAR_IN_OUT ... END_VAR	VAR_IN_OUT A: INT; END_VAR
2d	Declaration of temporary variables VAR_TEMP ... END_VAR	VAR_TEMP I: INT; END_VAR
2e	Declaration of **static** variables VAR ... END_VAR	VAR B: REAL; END_VAR
2f	Declaration of external variables VAR_EXTERNAL ... END_VAR	VAR_EXTERNAL B: REAL; END_VAR Corresponding to VAR_GLOBAL B: REAL

<table>
<tr><th>No.</th><th>Description</th><th>Example</th></tr>
<tr><td>2g</td><td>Declaration of external variables
VAR_EXTERNAL CONSTANT ... END_VAR</td><td>VAR_EXTERNAL CONSTANT B: REAL; END_VAR
Corresponding to
VAR_GLOBAL B: REAL</td></tr>
<tr><td>3a</td><td>Initialization of inputs</td><td>VAR_INPUT MN: INT:= 0;</td></tr>
<tr><td>3b</td><td>Initialization of outputs</td><td>VAR_OUTPUT RES: INT:= 1;</td></tr>
<tr><td>3c</td><td>Initialization of static variables</td><td>VAR B: REAL:= 12.1;</td></tr>
<tr><td>3d</td><td>Initialization of temporary variables</td><td>VAR_TEMP I: INT:= 1;</td></tr>
<tr><td>-</td><td>EN/ENO inputs and outputs</td><td>Defined in Table 18</td></tr>
<tr><td>4a</td><td>Declaration of RETAIN qualifier on input variables</td><td>VAR_INPUT RETAIN X: REAL; END_VAR</td></tr>
<tr><td>4b</td><td>Declaration of RETAIN qualifier on output variables</td><td>VAR_OUTPUT RETAIN X: REAL; END_VAR</td></tr>
<tr><td>4c</td><td>Declaration of NON_RETAIN qualifier on input variables</td><td>VAR_INPUT NON_RETAIN X: REAL; END_VAR</td></tr>
<tr><td>4d</td><td>Declaration of NON_RETAIN qualifier on output variables</td><td>VAR_OUTPUT NON_RETAIN X: REAL; END_VAR</td></tr>
<tr><td>4e</td><td>Declaration of RETAIN qualifier on static variables</td><td>VAR RETAIN X: REAL; END_VAR</td></tr>
<tr><td>4f</td><td>Declaration of NON_RETAIN qualifier on static variables</td><td>VAR NON_RETAIN X: REAL; END_VAR</td></tr>
<tr><td>5a</td><td>Declaration of RETAIN qualifier on local FB instances</td><td>VAR RETAIN TMR1: TON; END_VAR</td></tr>
<tr><td>5b</td><td>Declaration of NON_RETAIN qualifier on local FB instances</td><td>VAR NON_RETAIN TMR1: TON; END_VAR</td></tr>
<tr><td>6a</td><td>Textual declaration of
- rising edge inputs</td><td>FUNCTION_BLOCK AND_EDGE
VAR_INPUT X: BOOL R_EDGE;
          Y: BOOL F_EDGE;
END_VAR
VAR_OUTPUT Z: BOOL; END_VAR
 Z:= X AND Y; (* ST language example *)
END_FUNCTION_BLOCK</td></tr>
<tr><td>6b</td><td>- falling edge inputs (textual)</td><td>See above</td></tr>
<tr><td>7a</td><td>Graphical declaration of
- rising edge inputs (>)</td><td>FUNCTION_BLOCK
  (* External interface *)
          +----------+
          | AND_EDGE |
    BOOL-->X   Z|--BOOL
          | |
    BOOL--<Y |
          | |
          +----------+

  (* FB body *)
          +-----+
          | & |
       X--| |--Z
       Y--| |
          +-----+
END_FUNCTION_BLOCK</td></tr>
<tr><td>7b</td><td>Graphical declaration of
- falling edge inputs (<)</td><td>See above</td></tr>
<tr><td colspan="3">NOTE The features 1-3 of this table are equivalent to functions, see Table 19.</td></tr>
</table>

Examples of FB type declaration are shown below.

EXAMPLE 1 Function block type declaration

```
FUNCTION_BLOCK DEBOUNCE

(*** External Interface ***)
  VAR_INPUT
    IN: BOOL;                       (* Default = 0 *)
    DB_TIME: TIME:= t#10ms;         (* Default = t#10ms *)
  END_VAR


  VAR_OUTPUT
    OUT: BOOL;                      (* Default = 0 *)
    ET_OFF: TIME;                   (* Default = t#0s *)
  END_VAR


  VAR DB_ON: TON;                   (** Internal Variables **)
    DB_OFF:  TON;                   (**  and FB Instances  **)
    DB_FF:   SR;
  END_VAR


(*** Function Block Body ***)
  DB_ON (IN:= IN, PT:= DB_TIME);
  DB_OFF(IN:= NOT IN, PT:= DB_TIME);
  DB_FF (S1:= DB_ON.Q, R:= DB_OFF.Q);
  OUT:= DB_FF.Q1;
  ET_OFF:= DB_OFF.ET;

 END_FUNCTION_BLOCK
```

a) Textual declaration (ST language)

```
FUNCTION_BLOCK
(* External Parameter-Interface *)
                +--------------+
                |   DEBOUNCE   |
         BOOL---|IN         OUT|---BOOL
         TIME---|DB_TIME ET_OFF|---TIME
                +--------------+


(* Function block type body *)
              DB_ON       DB_FF
             +-----+     +----+
             | TON |     | SR |
 IN----+------|IN  Q|-----|S1 Q|---OUT
       |  +---|PT ET|  +--|R   |
       |  |   +-----+  |  +----+
       |  |            |
       |  |    DB_OFF  |
       |  |   +-----+  |
       |  |   | TON |  |
       +--|--O|IN  Q|--+
 DB_TIME--+---|PT ET|--------------ET_OFF
             +-----+

                                    END_FUNCTION_BLOCK
```

b) Graphical declaration (FBD language)

The example below shows the declaration and graphical usage of in-out variables in function blocks as given in Table 40.

EXAMPLE 2

```
      +-------+
      | ACCUM |
INT---|A-----A|--INT
INT---|X      |
      +-------+
        +---+
    A---| + |---A
    X---|   |
        +---+
```

```
FUNCTION_BLOCK ACCUM
   VAR_IN_OUT A: INT; END_VAR
   VAR_INPUT  X: INT; END_VAR
   A:= A+X;
END_FUNCTION_BLOCK
```

a) Graphical and textual declaration of function block type and function

```
               ACC1
             +-------+
             | ACCUM |
ACC----------|A-----A|---ACC
     +---+   |       |
X1---| * |---|X      |
X2---|   |   +-------+
     +---+
```

```
VAR
    ACC: INT;
    X1:  INT;
    X2:  INT;
END_VAR
```

This declaration is assumed: the effect of execution:

```
ACC:= ACC+X1*X2;
```

b) Allowed usage of function block instance and function

```
             ACC1                     ACC2
           +-------+                +-------+
           | ACCUM |                | ACCUM |
ACC--------|A-----A|----------------|A-----A|--ACC
    +---+  |       |       +---+    |       |
X1--| * |--|X      |  X3---| * |---|X       |
X2--|   |  +-------+  X4---|   |    +-------+
    +---+                  +---+
```

Declarations as in b) are assumed for `ACC, X1, X2, X3`, and `X4;`

the effect of execution is
`ACC:= ACC+X1*X2+X3*X4;`

c) Allowed usage of function block instance

```
               ACC1
             +-------+
             | ACCUM |
X3-----------|A-----A|---X4
     +---+   |       |
X1---| * |---|X      |
X2---|   |   +-------+
     +---+
```

```
VAR
    X1: INT;
    X2: INT;
    X3: INT;
    X4: INT;
END_VAR
```

The declaration is assumed: the effect of execution:

```
X3:= X3+X1*X2;
X4:= X3;
```

d) Allowed usage of function block instance and function – with assignment to an output

```
                ACC1
     +---+   +-------+
X1---| * |   | ACCUM |
X2---|   |---|A-----A|---ACC
     +---+   |       |
X3-----------|X      |
             +-------+
```

NOT **ALLOWED !**
Connection to in-out variable `A` is not a variable or a function block name (see preceding text)

e) Disallowed usage of FB instance

The following example shows the function block `AND_EDGE` used in Table 40.

EXAMPLE 3 Function block type declaration AND_EDGE

The declaration of function block `AND_EDGE` in the above examples in Table 40 is equivalent to:

```
FUNCTION_BLOCK AND_EDGE
  VAR_INPUT
    X: BOOL;
    Y: BOOL;
  END_VAR
  VAR
    X_TRIG: R_TRIG;
    Y_TRIG: F_TRIG;
  END_VAR
  VAR_OUTPUT
    Z: BOOL;
  END_VAR

  X_TRIG(CLK:= X);
  Y_TRIG(CLK:= Y);
  Z:= X_TRIG.Q AND Y_TRIG.Q;
END_FUNCTION_BLOCK
```

See Table 44 for the definition of the edge detection function blocks `R_TRIG` and `F_TRIG`.

6.6.3.3 Function block instance declaration

The function block instance shall be declared in a similar manner as described for structured variables.

When a function block instance is declared, the initial values for the inputs, outputs or public variables of the function block instance can be declared in a parenthesized list following the assignment operator that follows the function block type identifier as shown in Table 41.

Elements for which initial values are not listed in the above described initialization list shall have the default initial values declared for those elements in the function block type declaration.

Table 41 – Function block instance declaration

No.	Description	Example

1	Declaration of FB instance(s)	`VAR`
		`FB_instance_1, FB_instance_2: my FB_Type;`
		`T1, T2, T3: TON;`
		`END_VAR`

2	Declaration of FB instance with initialization of its variables	`VAR`
		`TempLoop: PID:=    (PropBand:= 2.5,`
		`Integral:= T#5s);`
		`END_VAR`
		Allocates initial values to inputs and outputs of a function block instance.

6.6.3.4 Function block call

6.6.3.4.1 General

The call of an instance of a function block can be represented in a textual or graphical form.

The features of the function block call (including the formal and the non-formal call) are similar to those of the functions with the following extensions:

1. The textual call of a FB shall consist of the instance name of the function block followed by a list of parameters.

2. In the graphical representation, the instance name of the function block shall be located above the block.
3. The input variables and output variables of an instance of a function block are stored and can be represented as elements of structured data types. Therefore the assignment of the inputs and the access to the outputs of a function block can be
   a) immediate within the call of the function block; this is the typical usage or
   b) separate from the call. These separate assignments shall become effective with the next call of the function block.
   c) unassigned or unconnected inputs of a function block shall keep their initialized values or the values from the latest previous call, if any.

It is possible that no actual parameter is specified for an in-out variable or a function block instance used as an input variable of another function block instance. However, the instance shall be provided with a valid value which is stored, e.g. via initialization or former call, before used in the function block (body) or by a method, otherwise it causes a runtime error.

Further rules apply for the function block call:

4. If a function block instance is called with EN=0, the Implementer shall specify if the input and in-out variables are set in the instance.
5. The name of a function block instance can be used as the input to a function block instance if declared as an input variable in a VAR_INPUT declaration, or as an input/output variable of a function block instance in a VAR_IN_OUT declaration.
6. The output values of a different function block instance whose name is passed into the function block via a VAR_INPUT, VAR_IN_OUT, or VAR_EXTERNAL construct can be accessed, but not modified, from within the function block.
7. A function block whose instance name is passed into the function block via a VAR_IN_OUT or VAR_EXTERNAL construction can be called from inside the function block.
8. Only variables or function block instance names can be passed into a function block via the VAR_IN_OUT construct.

   This is to prevent the inadvertent modifications of such outputs. However, "cascading" of VAR_IN_OUT constructions is permitted.

The following Table 42 contains the features of the function block call.

Table 42 – Function block call

No.	Description	Example

1	Complete formal call (textual only)
	
	Is used if EN/ENO is necessary in calls.
		`YourCTU(      EN:= not B,`
		`               CU:= r,`
		`               PV:= c1,`
		`               ENO=> next,`
		`               Q  => out,`
		`               CV => c2);`

2	Incomplete formal call (textual only)	`YourCTU(      Q  => out,`
		`               CV => c2);`
		
		`EN`, `CU`, `PV` variable will have the value of the last call or an initial value, if never called before.

3	Graphical call	`          YourCTU`
		`         +-------+`
		`
			CTU |`
			`     B --|EN  ENO|-- next`
			`     r --|CU    Q|-- out`
			`    c1 --|PV   CV|-- c2`
			`         +-------+`

<table>
<tr><th>No.</th><th>Description</th><th>Example</th></tr>
<tr><td>4</td><td>Graphical call with negated boolean input and output</td><td>   YourCTU
    +-------+
    | CTU |
B -0|EN ENO|-- next
r --|CU   Q|0- out
c1 --|PV   CV|-- c2
    +-------+
The use of these constructs is forbidden for in-out variables.</td></tr>
<tr><td>5a</td><td>Graphical call with usage of VAR_IN_OUT</td><td></td></tr>
<tr><td>5b</td><td>Graphical call with assignment of VAR_IN_OUT to a variable</td><td></td></tr>
<tr><td>6a</td><td>Textual call with separate assignment of input
FB_Instance.Input:= x;</td><td>YourTon.IN:= r;
YourTon.PT:= t;
YourTon(not Q => out);</td></tr>
<tr><td>6b</td><td>Graphical call with separate assignment of input</td><td>   +------+
r--| MOVE |--YourCTU.CU
   +------+
   +------+
c--| MOVE |--YourCTU.PV
   +------+
     YourCTU
    +-------+
    | CTU |
  1--|EN ENO|-- next
   --|CU   Q|0- out
   --|PV   CV|--
    +-------+</td></tr>
<tr><td>7</td><td>Textual output read after FB call
x:= FB_Instance.Output;</td><td></td></tr>
<tr><td>8a</td><td>Textual output assigned in FB call</td><td></td></tr>
<tr><td>8b</td><td>Textual output assigned in FB call with negation</td><td></td></tr>
<tr><td>9a</td><td>Textual call with function block instance name as input</td><td>VAR_INPUT I_TMR: TON; END_VAR
 EXPIRED:= I_TMR.Q;
It is assumed that the variables EXPIRED and A_VAR have been declared of type BOOL in this example and in the following examples.</td></tr>
<tr><td>9b</td><td>Graphical call with function block instance name as input</td><td>a</td></tr>
<tr><td>10a</td><td>Textual call with function block instance name as VAR_IN_OUT</td><td>VAR_IN_OUT IO_TMR: TOF; END_VAR
 IO_TMR (IN:=A_VAR, PT:= T#10S);
 EXPIRED:= IO_TMR.Q;</td></tr>
<tr><td>10b</td><td>Graphical call with function block instance name as VAR_IN_OUT</td><td></td></tr>
<tr><td>11a</td><td>Textual call with function block instance name as external variable</td><td>VAR_EXTERNAL EX_TMR: TOF; END_VAR
 EX_TMR(IN:= A_VAR, PT:= T#10S);
 EXPIRED:= EX_TMR.Q;</td></tr>
<tr><td>11b</td><td>Graphical call with function block instance name as external variable</td><td></td></tr>
</table>

EXAMPLE Function block call with immediate and separate parameter assignment

```
        YourCTU                      YourCTU (EN:= not b,
       +-------+                              CU:= r,
       |  CTU  |                              PV:= c,
   B -0|EN  ENO|--                            not Q => out);
   r --|CU    Q|0-out
   c --|PV   CV|--
       +-------+
```

a) FB call with immediate assignment of inputs (typical usage)

```
      +-------+
   r--| MOVE |--YourCTU.CU            YourCTU.CU:= r;
      +------+                        YourCTU.PV:= V;
      +------+
   c--| MOVE |--YourCTU.PV            YourCTU(not Q => out);
      +------+

            YourCTU
           +--------+
           |  CTU   |
         --|EN  ENO|--
         --|CU    Q|0-out
         --|PV   CV|--
           +--------+
```

b) FB call with separate assignment of input

```
              YourCTU
             +--------+
                                       VAR a, b, r, out: BOOL;
   +----+    |  CTU   |                    YourCTU: CTU; END_VAR
a--| NE |---0|EN  ENO|--
b--|    | r--|CU    Q|0-out             YourCTU (EN := NOT (a <> b),
   +----+  --|PV   CV|--                         CU := r,
             +--------+                          NOT Q => out);
```

c) FB call with immediate access to output (typical usage)
Also negation in call is permitted

```
               FF75
             +------+
             |  SR  |                VAR FF75: SR; END_VAR (* Declaration *)
      bIn1---|S1  Q1|---bOut3         FF75(S1:= bIn1,      (* call *)
      bIn2---|R     |                       R:= bIn2);
             +------+
                                      bOut3:= FF75.Q1;    (* Assign Output *)
```

d) FB call with textual separate output assignment (after call)

```
                                     VAR
            TONs[12]                   TONs: array [0..100] OF TON;
            +--------+                 i: INT;
            |  TON   |               END_VAR
   bIn1   --|IN    Q|--
   T#10ms --|PT   ET|--
            +--------+
                                      TON[12](IN:= bIn1, PT:= T#10ms);
             TONs[i]
            +-------+
            |  TON  |
   bIn1   --|IN    Q|--               TON[i](IN:= bIn1, PT:= T#20ms);
   T#20ms --|PT   ET|--
            +-------+
```

e) FB call using an instance array

```
                                     TYPE
        myCooler.Cooling               Cooler: STRUCT
            +-------+                    Temp: INT;
            |  TOF  |                    Cooling: TOF;
   bIn1   --|IN    Q|--                END_STRUCT;
   T#30s  --|PT   ET|--              END_TYPE
            +-------+
                                     VAR
                                       myCooler: Cooler;
                                     END_VAR

                                      myCooler.Cooling(IN:= bIn1, PT:= T#30s);
```

f) FB call using an instance as structure element

6.6.3.4.2 Usage of input and output parameters

Figure 13 and Figure 14 summarize the rules for usage of input and output parameter of a function block in the context of the call of this function block. This assignment to input and in-out parameters shall become effective with the next call of the FB.

```
FUNCTION_BLOCK FB_TYPE;

VAR_INPUT  In:      REAL; END_VAR
VAR_OUTPUT Out:     REAL; END_VAR
VAR_IN_OUT In_out: REAL; END_VAR
VAR        M:       REAL; END_VAR

END_FUNCTION-BLOCK

VAR FB_INST: FB_TYPE; A, B, C: REAL; END_VAR
```

	Usage	a) Inside function block	b) Outside function block
1.	Input read	M:= In;	~~A:= In;~~ Not allowed (NOTES 1 and 2)

2.	Input assignment	~~In:= M;~~ Not allowed (NOTE 1)	// Call with immediate parameter assignment
			FB_INST(In:= A);
			// Separate assignment (NOTE 4)
			FB_INST.In:= A;

3.	Output read	M:= Out;	// Call with immediate parameter assignment
			FB_INST(Out => B);
			// Separate assignment
			B:= FB_INST.Out;

4.	Output assignment	Out:= M;	~~FB_INST.Out:= B;~~ Not Allowed (NOTE 1)
5.	In-out read	M:= In_out;	~~FB_INST(In_out=> C);~~ Not allowed ~~C:= FB_INST.In_out;~~Not allowed

6.	In-out assignment	In_out:= M; (NOTE 3)	// Call with immediate parameter assignment
			FB_INST(In_out:= C);
			~~FB_INST.In_out:= C;~~Not allowed

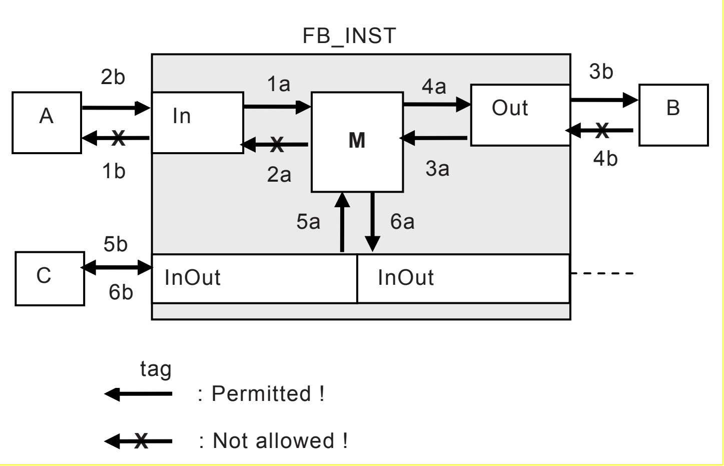
NOTE 1 Those usages listed as "not allowed" in this table could lead to Implementer specific unpredictable side effects.

NOTE 2 Reading and writing (assignment) of input, output parameters and internal variables of a function block may be performed by the "communication function", "operator interface function", or the "programming, testing, and monitoring functions" defined in IEC 61131-1.

NOTE 3 Modification within the function block of a variable declared in a VAR_IN_OUT block is permitted.

Figure 13 – Usage of function block input and output parameters (Rules)

The usage of input and output parameters defined by the rules of Figure 13 is illustrated in Figure 14.



The tags 1a, 1b, etc are the rules from Figure 13.

Figure 14 – Usage of function block input and output parameters (Illustration of rules)

The following examples shows examples of the graphical usage of function block names as parameters and external variable.

EXAMPLES Graphical usage of function block names as parameter and external variables

```
  FUNCTION_BLOCK

  (* External interface *)

          +---------------+
          |   INSIDE_A    |
   TON---|I_TMR  EXPIRED|---BOOL
          +---------------+


  (* Function block body *)

                +-------+
                | MOVE  |
   I_TMR.Q---  |       |---EXPIRED
                +-------+

END_FUNCTION_BLOCK
  FUNCTION_BLOCK
  (* External interface *)

           +---------------+
           |  EXAMPLE_A    |
   BOOL---|GO         DONE|---BOOL
           +---------------+


  (* Function block body *)
              E_TMR

               +------+                   I_BLK
               | TON  |             +---------------+
         GO---|IN   Q|             |    INSIDE_A    |
   t#100ms---|PT  ET|      E_TMR---|I_TMR  EXPIRED|---DONE
               +------+             +---------------+

END_FUNCTION_BLOCK
```

a) Function block name as an input variable (NOTE)

```
  FUNCTION_BLOCK

  (* External interface *)

          +--------------+
          |   INSIDE_B   |
    TON---|I_TMR----I_TMR|---TON
    BOOL--|TMR_GO EXPIRED|---BOOL
          +--------------+

  (* Function block body *)

             I_TMR
            +-----+
            | TON |
   TMR_GO---|IN  Q|---EXPIRED
            |PT ET|
            +-----+
END_FUNCTION_BLOCK
```

```
  FUNCTION_BLOCK

  (* External interface *)

           +---------------+
           |  EXAMPLE_B    |
    BOOL---|GO        DONE|---BOOL
           +---------------+


  (* Function block body *)

                E_TMR
               +-----+                  I_BLK
               | TON |             +----------------+
               |IN  Q|             |   INSIDE_B     |
     t#100ms---|PT ET|     E_TMR---|I_TMR-----I_TMR|
               +-----+     GO------|TMR_GO  EXPIRED|---DONE
                                   +----------------+

END_FUNCTION_BLOCK
```

b) Function block name as an in-out variable

```
  FUNCTION_BLOCK

  (* External interface *)

          +--------------+
          |   INSIDE_C   |
    BOOL--|TMR_GO EXPIRED|---
          +--------------+


  VAR_EXTERNAL X_TMR: TON; END_VAR

  (* Function block body *)

             X_TMR
            +-----+
            | TON |
   TMR_GO---|IN  Q|---EXPIRED
            |PT ET|
            +-----+
END_FUNCTION_BLOCK
```

```
  PROGRAM
  (* External interface *)
           +--------------+
           |  EXAMPLE_C   |
    BOOL---|GO        DONE|---BOOL
           +--------------+


  VAR_GLOBAL X_TMR: TON; END_VAR


  (* Program body *)
               I_BLK
          +---------------+
          |   INSIDE_C    |
    GO---|TMR_GO  EXPIRED|---DONE
          +---------------+
END_PROGRAM
```

c) Function block name as an external variable

NOTE I_TMR is here not represented graphically since this would imply call of I_TMR within INSIDE_A, which is forbidden by rules 3) and 4) of Figure 13.

6.6.3.5 Standard function blocks

6.6.3.5.1 General

Definitions of standard function blocks common to all programmable controller programming languages are given below. The Implementer may provide additional standard function blocks.

Where graphical declarations of standard function blocks are shown in this subclause, equivalent textual declarations can also be written, as for example in Table 44.

Standard function blocks may be overloaded and may have extensible inputs and outputs. The definitions of such function block types shall describe any constraints on the number and data types of such inputs and outputs. The use of such capabilities in non-standard function blocks is beyond the scope of this standard.

6.6.3.5.2 Bistable elements

The graphical form and function block body of standard bistable elements are shown in Table 43.

Table 43 – Standard bistable function blocks[a]

<table>
<tr><th>No.</th><th>Description/Graphical form</th><th>Function block body</th></tr>
<tr><td>1a</td><td>Bistable function block (set dominant): <code>SR(S1,R,Q1)</code></td><td></td></tr>
<tr><td></td><td><pre>
       +-----+
       | SR |
BOOL---|S1 Q1|---BOOL
BOOL---|R |
       +-----+
</pre></td><td><pre>
                 +-----+
S1 -------------| >=1 |-- Q1
        +---+ | |
R ----O| & |----| |
Q1 -----| | | |
        +---+   +-----+
</pre></td></tr>
<tr><td>1b</td><td>Bistable function block (set dominant)
with long input names: <code>SR(SET1, RESET, Q1)</code></td><td></td></tr>
<tr><td></td><td><pre>
       +--------+
       | SR |
BOOL---|SET1 Q1|---BOOL
BOOL---|RESET |
       +--------+
</pre></td><td><pre>
                  +-----+
SET1 ------------| >=1 |-- Q1
         +---+ | |
 RESET -O| & |----| |
 Q1 ----| | | |
         +---+   +-----+
</pre></td></tr>
<tr><td>2a</td><td>Bistable function block (reset dominant): <code>RS(S, R1, Q1)</code></td><td></td></tr>
<tr><td></td><td><pre>
       +-----+
       | RS |
BOOL---|S Q1|---BOOL
BOOL---|R1 |
       +-----+
</pre></td><td><pre>
                   +---+
R1 --------------O| & |-- Q1
        +-----+ | |
S -----| >=1 |----| |
Q1 -----| | | |
        +-----+   +---+
</pre></td></tr>
<tr><td>2b</td><td>Bistable function block (reset dominant)
with long input names: <code>RS(SET,RESET1, Q1)</code></td><td></td></tr>
<tr><td></td><td><pre>
       +--------+
       | RS |
BOOL---|SET   Q1|---BOOL
BOOL---|R1 |
       +--------+
</pre></td><td><pre>
                      +---+
RESET1 --------------0| & |-- Q1
           +-----+ | |
   SET ----| >=1 |----| |
   Q1 ----| | +---+
           +-----+
</pre></td></tr>
<tr><td colspan="3">[a] The initial state of the output variable <code>Q1</code> shall be the normal default value of zero for Boolean variables.</td></tr>
</table>

6.6.3.5.3 Edge detection (`R_TRIG` and `F_TRIG`)

The graphic representation of standard rising- and falling-edge detecting function blocks shall be as shown in Table 44. The behaviors of these blocks shall be equivalent to the definitions given in this table. This behavior corresponds to the following rules:

1. The `Q` output of an `R_TRIG` function block shall stand at the `BOOL#1` value from one execution of the function block to the next, following the `0` to `1` transition of the `CLK` input, and shall return to `0` at the next execution.
2. The `Q` output of an `F_TRIG` function block shall stand at the `BOOL#1` value from one execution of the function block to the next, following the `1` to `0` transition of the `CLK` input, and shall return to 0 at the next execution.

Table 44 – Standard edge detection function blocks

<table>
<tr><th>No.</th><th>Description/Graphical form</th><th>Definition
(ST language)</th></tr>
<tr><td>1</td><td>Rising edge detector: <code>R_TRIG(CLK, Q)</code></td><td></td></tr>
<tr><td></td><td>

```
       +--------+
       | R_TRIG |
BOOL --|CLK    Q|-- BOOL
       +--------+
```

</td><td>

```
FUNCTION_BLOCK R_TRIG
  VAR_INPUT  CLK: BOOL; END_VAR
  VAR_OUTPUT Q: BOOL;    END_VAR
  VAR M: BOOL; END_VAR
   Q:= CLK AND NOT M;
   M:= CLK;
END_FUNCTION_BLOCK
```

</td></tr>
<tr><td>2</td><td>Falling edge detector: <code>F_TRIG(CLK, Q)</code></td><td></td></tr>
<tr><td></td><td>

```
       +--------+
       | F_TRIG |
BOOL --|CLK    Q|-- BOOL
       +--------+
```

</td><td>

```
FUNCTION_BLOCK F_TRIG
  VAR_INPUT CLK:  BOOL; END_VAR
  VAR_OUTPUT Q:   BOOL; END_VAR
  VAR M: BOOL;    END_VAR
   Q:= NOT CLK AND NOT M;
   M:= NOT CLK;
END_FUNCTION_BLOCK
```

</td></tr>
<tr><td colspan="3">NOTE When the CLK input of an instance of the R_TRIG type is connected to a value of BOOL#1, its Q output will stand at BOOL#1 after its first execution following a “cold restart”. The Q output will stand at BOOL#0 following all subsequent executions. The same applies to an F_TRIG instance whose CLK input is disconnected or is connected to a value of FALSE.</td></tr>
</table>

6.6.3.5.4 Counters

The graphic representations of standard counter function blocks, with the types of the associated inputs and outputs, shall be as shown in Table 45. The operation of these function blocks shall be as specified in the corresponding function block bodies.

Table 45 – Standard counter function blocks

<table>
<tr><th>No.</th><th>Description/Graphical form</th><th>Function block body
(ST language)</th></tr>
<tr><td></td><td><b>Up-Counter</b></td><td></td></tr>
<tr><td>1a</td><td><code>CTU_INT(CU, R, PV, Q, CV)</code> or <code>CTU(..)</code></td><td></td></tr>
<tr><td></td><td>

```
           +-----+
           | CTU |
   BOOL--->CU  Q|---BOOL
   BOOL---|R    |
    INT---|PV CV|---INT
           +-----+
```

and also:

```
          +------------+
          |   CTU_INT  |
  BOOL--->CU          Q|---BOOL
  BOOL---|R            |
   INT---|PV         CV|---INT
          +------------+
```

</td><td>

```
VAR_INPUT CU: BOOL R_EDGE; ...
```

/* The edge is evaluated internally by the data type R_EDGE */

```
IF R
THEN CV:= 0;
ELSIF CU AND (CV < PVmax)
     THEN
     CV:= CV+1;
END_IF;
 Q:= (CV >= PV);
```

</td></tr>
<tr><td>1b</td><td><code>CTU_DINT   PV, CV: DINT</code></td><td>see 1a</td></tr>
<tr><td>1c</td><td><code>CTU_LINT   PV, CV:</code> LINT</td><td>see 1a</td></tr>
<tr><td>1d</td><td><code>CTU_UDINT   PV, CV: UDINT</code></td><td>see 1a</td></tr>
<tr><td>1e</td><td><code>CTU_ULINT(CD, LD, PV, CV) PV, CV: ULINT</code></td><td>see 1a</td></tr>
<tr><td></td><td><b>Down-counters</b></td><td></td></tr>
<tr><td>2a</td><td><code>CTD_INT(CD, LD, PV, Q, CV)</code> or <code>CTD</code></td><td></td></tr>
</table>

<table>
<tr><th>No.</th><th>Description/Graphical form</th><th>Function block body
(ST language)</th></tr>
<tr><td></td><td><pre>
        +-----+
        | CTD |
 BOOL--->CD Q|---BOOL
 BOOL---|LD |
  INT---|PV CV|---INT
        +-----+
</pre>
and also:
<pre>
        +-----------+
        | CTD_INT |
 BOOL--->CD   Q|---BOOL
 BOOL---|LD |
  INT---|PV   CV|---INT
        +-----------+
</pre></td><td><pre>
VAR_INPUT CU: BOOL R_EDGE; ...
</pre>
// The edge is evaluated internally by the data type R_EDGE
<pre>
IF LD
THEN CV:= PV;
ELSIF CD AND (CV > PVmin)
   THEN CV:= CV-1;
END_IF;
 Q:= (CV <= 0);
</pre></td></tr>
<tr><td>2b</td><td>CTD_DINT   PV, CV: DINT</td><td>See 2a</td></tr>
<tr><td>2c</td><td>CTD_LINT   PV, CV: LINT</td><td></td></tr>
<tr><td>2d</td><td>CTD_UDINT   PV, CV: UDINT</td><td>See 2a</td></tr>
<tr><td>2e</td><td>CTD_ULINT   PV, CV: UDINT</td><td>See 2a</td></tr>
<tr><td></td><td><b>Up-down counters</b></td><td></td></tr>
<tr><td>3a</td><td>CTUD_INT(CD, LD, PV, Q, CV) or CTUD(..)</td><td></td></tr>
<tr><td></td><td><pre>
        +-----------+
        | CTUD |
 BOOL--->CU   QU|---BOOL
 BOOL--->CD   QD|---BOOL
 BOOL---|R |
 BOOL---|LD |
  INT---|PV   CV|---INT
        +-----------+
</pre>
and also:
<pre>
        +-----------+
        | CTUD_INT |
 BOOL--->CU   QU|---BOOL
 BOOL--->CD   QD|---BOOL
 BOOL---|R |
 BOOL---|LD |
  INT---|PV   CV|---INT
        +-----------+
</pre></td><td><pre>
VAR_INPUT CU, CD: BOOL R_EDGE; ...
</pre>
// Edge is evaluated internally by the data type R_EDGE
<pre>
IF R
THEN CV:= 0;
ELSIF LD
  THEN CV:= PV;
  ELSE
     IF NOT (CU AND CD)
     THEN
        IF CU AND (CV < PVmax)
        THEN CV:= CV+1;
        ELSIF CD AND (CV > PVmin)
           THEN CV:= CV-1;
        END_IF;
     END_IF;
 END_IF;
 QU:= (CV >= PV);
 QD:= (CV <= 0);
</pre></td></tr>
<tr><td>3b</td><td>CTUD_DINT   PV, CV: DINT</td><td>See 3a</td></tr>
<tr><td>3c</td><td>CTUD_LINT   PV, CV: LINT</td><td>See 3a</td></tr>
<tr><td>3d</td><td>CTUD_UDINT   PV, CV: UDINT</td><td>See 3a</td></tr>
<tr><td>3e</td><td>CTUD_ULINT   PV, CV: ULINT</td><td>See 3a</td></tr>
<tr><td colspan="3">NOTE The numerical values of the limit variables PVmin and PVmax are Implementer specific.</td></tr>
</table>

6.6.3.5.5 Timers

The graphic form for standard timer function blocks shall be as shown in Table 46. The operation of these function blocks shall be as defined in the timing diagrams given in Figure 15.

The standard timer function blocks may be used overloaded with TIME or LTIME, or the base data type for the standard timer may be specified as TIME or LTIME.

Table 46 – Standard timer function blocks

No.	Description	Symbol	Graphical form

1a	Pulse, overloaded `TP`	*** is: `TP`	`+-------+`
			`|  ***  |`
			`BOOL---|IN    Q|---BOOL`
			`TIME---|PT   ET|---TIME`
			`+-------+`
			`PT` see NOTE
			`IN:` Input (Start)
			`PT:` Preset Time
			`Q:` Output
			`ET:` Elapsed Time

1b	Pulse using TIME	`TP_TIME`	
1c	Pulse using LTIME	`TP_LTIME`	
2a	On-delay, overloaded `TON`	`TON`	
2b	On-delay using TIME	`TON_TIME`	
2c	On-delay using LTIME	`TON_LTIME`	
2d [a]	On-delay, overloaded (Graphical)	`T---0`	
3a	Off-delay, overloaded `TOF`	`TOF`	
3b	Off-delay using TIME	`TOF_TIME`	
3c	Off-delay using LTIME	`TOF_LTIME`	
3d [a]	Off-delay, overloaded (Graphical)	`0---T`	

NOTE The effect of a change in the value of the `PT` input during the timing operation, e.g., the setting of `PT` to `t#0s` to reset the operation of a `TP` instance, is an Implementer specific parameter.

[a] In textual languages, features 2b and 3b shall not be used.

Figure 15 below shows the timing diagrams of the standard timer function blocks.

```
      +--------+     ++ ++   +--------+
 IN   |        |     || ||   |        |
    --+        +-----++-++---+        +---------
      t0       t1    t2 t3    t4       t5

      +----+         +----+  +----+
 Q    |    |         |    |  |    |
    --+    +---------+    +--+    +-------------
      t0   t0+PT    t2 t2+PT t4  t4+PT

   PT      +---+          +       +---+
   :     /    |         /|      /    |
ET :    /     |        / |     /     |
   :   /      |       /  |    /      |
   :  /       |      /   |   /       |
    0-+        +-----+    +--+        +---------
      t0       t1    t2       t4       t5
```

a) Pulse (PT) timing

```
     +--------+        +---+   +--------+
IN   |        |        |   |   |        |
   --+        +--------+   +---+        +-------------
     t0       t1       t2  t3  t4       t5

          +---+                     +---+
Q         |   |                     |   |
   -------+   +---------------------+   +-------------
       t0+PT  t1                t4+PT   t5

  PT      +---+                     +---+
  :     /    |            +       /    |
ET:    /     |           /|      /     |
  :   /      |          / |     /      |
  :  /       |         /  |    /       |
   0-+        +--------+   +---+        +-------------
     t0       t1       t2  t3  t4       t5
```

b) On-delay (TON) timing

```
       +--------+        +---+   +--------+
IN     |        |        |   |   |        |
    ---+        +--------+   +---+        +-----------
       t0       t1       t2  t3  t4       t5

       +-------------+   +---------------------+
Q      |             |   |                     |
    ---+             +---+                     +------
       t0          t1+PT t2                  t5+PT

  PT                 +---+                     +------
  :                /    |       +            /
ET:               /     |      /|           /
  :              /      |     / |          /
  :             /       |    /  |         /
   0------------+        +---+   +--------+
                t1           t3           t5
```

c) Off-delay (TOF) timing

Figure 15 – Standard timer function blocks – timing diagrams (Rules)

6.6.3.5.6 Communication function blocks

Standard communication function blocks for programmable controllers are defined in IEC 61131-5. These function blocks provide programmable communications functionality such as device verification, polled data acquisition, programmed data acquisition, parametric control, interlocked control, programmed alarm reporting, and connection management and protection.

6.6.4 Programs

A program is defined in IEC 61131-1 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 PLC-system."

The declaration and usage of programs is identical to that of function blocks with the additional features shown in Table 47 and the following differences:

1. The delimiting keywords for program declarations shall be `PROGRAM...END_PROGRAM`.
2. A program can contain a `VAR_ACCESS...END_VAR` construction, which provides a means of specifying named variables which can be accessed by some of the communication services specified in IEC 61131-5. An access path associates each such variable with an input, output or internal variable of the program.
3. Programs can only be instantiated within resources while function blocks can only be instantiated within programs or other function blocks.
4. A program can contain location assignments in the declarations of its global and internal variables. Location assignments with partly specified direct representation can only be used in the declaration of internal variables of a program.
5. The object-orientation features for programs are beyond the scope of this part of IEC 61131.

Table 47 – Program declaration

No.	Description	Example
1	Declaration of a program `PROGRAM ... END_PROGRAM`	`PROGRAM myPrg ... END_PROGRAM`
2a	Declaration of inputs `VAR_INPUT ... END_VAR`	`VAR_INPUT IN: BOOL; T1: TIME; END_VAR`
2b	Declaration of outputs `VAR_OUTPUT ... END_VAR`	`VAR_OUTPUT OUT: BOOL; ET_OFF: TIME; END_VAR`
2c	Declaration of in-outs `VAR_IN_OUT ... END_VAR`	`VAR_IN_OUT A: INT; END_VAR`
2d	Declaration of temporary variables `VAR_TEMP ... END_VAR`	`VAR_TEMP I: INT; END_VAR`
2e	Declaration of static variables `VAR ... END_VAR`	`VAR B: REAL; END_VAR`

2f	Declaration of external variables
	`VAR_EXTERNAL ... END_VAR`
		`VAR_EXTERNAL B: REAL; END_VAR`
		Corresponding to
		`VAR_GLOBAL B: REAL`

2g	Declaration of external variables
	`VAR_EXTERNAL CONSTANT ... END_VAR`
		`VAR_EXTERNAL CONSTANT B: REAL; END_VAR`
		Corresponding to
		`VAR_GLOBAL B: REAL`

3a	Initialization of inputs	`VAR_INPUT  MN:  INT:= 0;`
3b	Initialization of outputs	`VAR_OUTPUT RES: INT:= 1;`
3c	Initialization of static variables	`VAR B: REAL:= 12.1;`
3d	Initialization of temporary variables	`VAR_TEMP I: INT:= 1;`
4a	Declaration of `RETAIN` qualifier on input variables	`VAR_INPUT RETAIN  X: REAL; END_VAR`
4b	Declaration of `RETAIN` qualifier on output variables	`VAR_OUTPUT RETAIN  X: REAL; END_VAR`
4c	Declaration of `NON_RETAIN` qualifier on input variables	`VAR_INPUT NON_RETAIN  X: REAL; END_VAR`

<table>
<tr><th>No.</th><th>Description</th><th>Example</th></tr>
<tr><td>4d</td><td>Declaration of NON_RETAIN qualifier on output variables</td><td>VAR_OUTPUT NON_RETAIN X: REAL; END_VAR</td></tr>
<tr><td>4e</td><td>Declaration of RETAIN qualifier on static variables</td><td>VAR RETAIN X: REAL; END_VAR</td></tr>
<tr><td>4f</td><td>Declaration of NON_RETAIN qualifier on static variables</td><td>VAR NON_RETAIN X: REAL; END_VAR</td></tr>
<tr><td>5a</td><td>Declaration of RETAIN qualifier on local FB instances</td><td>VAR RETAIN TMR1: TON; END_VAR</td></tr>
<tr><td>5b</td><td>Declaration of NON_RETAIN qualifier on local FB instances</td><td>VAR NON_RETAIN TMR1: TON; END_VAR</td></tr>
<tr><td>6a</td><td>Textual declaration of
- rising edge inputs</td><td>PROGRAM AND_EDGE
VAR_INPUT X: BOOL R_EDGE;
          Y: BOOL F_EDGE;
END_VAR
VAR_OUTPUT Z: BOOL; END_VAR
 Z:= X AND Y; (* ST language example *)
END_PROGRAM</td></tr>
<tr><td>6b</td><td>Textual declaration of
- falling edge inputs (textual)</td><td>See above</td></tr>
<tr><td>7a</td><td>Graphical declaration of
- rising edge inputs (&gt;)</td><td>PROGRAM
      (* External interface *)
          +----------+
          | AND_EDGE |
    BOOL-->X   Z|--BOOL
          | |
    BOOL--<Y |
          | |
          +----------+

      (* FB body *)
          +-----+
          | & |
       X--| |--Z
       Y--| |
          +-----+
END_PROGRAM</td></tr>
<tr><td>7b</td><td>Graphical declaration of
- falling edge inputs (&lt;)</td><td>See above</td></tr>
<tr><td>8a</td><td>VAR_GLOBAL...END_VAR declaration within a PROGRAM</td><td>VAR_GLOBAL z1: BYTE; END_VAR</td></tr>
<tr><td>8b</td><td>VAR_GLOBAL CONSTANT declarations within PROGRAM type declarations</td><td>VAR_GLOBAL CONSTANT z2: BYTE; END_VAR</td></tr>
<tr><td>9</td><td>VAR_ACCESS...END_VAR declaration within a PROGRAM</td><td>VAR_ACCESS
  ABLE: STATION_1.%IX1.1: BOOL READ_ONLY;
  BAKER: STATION_1.P1.x2: UINT READ_WRITE;
END_VAR</td></tr>
<tr><td colspan="3">NOTE   The features 2a to 7b are equivalent to the same features of Table 40 for function blocks.</td></tr>
</table>

6.6.5 Classes

6.6.5.1 General

The language element class supports the object oriented paradigm, and is characterized by the following concepts:

- Definition of a data structure partitioned into public and internal variables,
- Method to be performed upon the elements of the data structure,
- Classes, consisting of methods (algorithms) and data structures

- Interface with method prototypes and implementation of interfaces,
- Inheritance of interfaces and classes,
- Instantiation of classes.

NOTE The terms class and object used in IT programming languages like C#, C++, Java, UML etc., correspond with the terms type and instance used in PLC programming languages of this standard. This is shown below.

IT Programming Languages: C#, C++, Java, UML		**PLC languages of the standard**	
Class	(= type of a class)	Type	of a function block and class
Object	(= instance of a class)	Instance	of a function block and class

The following Figure 16 illustrates the inheritance of interface and classes using the mechanisms of implementation and extension. This is defined in this 6.6.5.

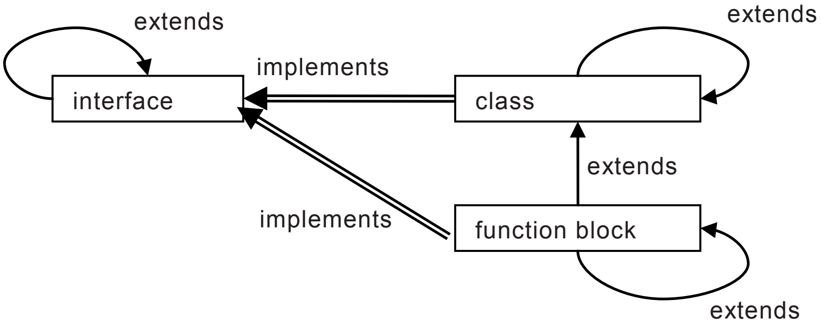


Figure 16 – Overview of inheritance and interface implementation

A class is a POU designed for object oriented programming. A class contains essentially variables and methods. A class shall be instantiated before its methods can be called or its variables can be accessed.

6.6.5.2 Class declaration

The features of the class declaration are defined in Table 48:

1. The keyword CLASS, followed by an identifier specifying the name of the class being declared.
2. The terminating keyword END_CLASS.
3. The values of the variables which are declared via a VAR_EXTERNAL construct can be modified from within the class.
4. The values of the constants which are declared via a VAR_EXTERNAL CONSTANT construct cannot be modified from within the class.
5. A VAR...END_VAR construct, if required, specifying the names and types of the variables of the class.
6. The variables may be initialized.
7. Variables of the VAR section (static) may be declared PUBLIC. A public variable may be accessed from outside the class using the same syntax as for the access to FB outputs.
8. The RETAIN or NON_RETAIN qualifier can be used for internal variables of a class.
9. The asterisk ‘*‘ notation as defined in Table 16 may be used in the declaration of internal variables of a class.
10. Variables may be PUBLIC, PRIVATE, INTERNAL, or PROTECTED. The access specifier PROTECTED is default.
11. A class may support inheritance of other classes to extend a base class.
12. A class may implement one or more interfaces.
13. Instances of other function blocks, classes and object oriented function blocks can be declared in the variable sections VAR and VAR_EXTERNAL.

14. A class instance declared inside a class should not use the same name as a function (of the same name scope) to avoid ambiguities.

The class has the following differences to the function block:

- The keywords FUNCTION_BLOCK and END_FUNCTION_BLOCK are replaced by CLASS and END_CLASS respectively.
- Variables are only declared in the VAR section. The sections VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT, and VAR_TEMP are not allowed.
- A class has no body. A class may define only methods.
- A call of an instance of a class is not possible. Only the methods of a class can be called.

The implementer of classes shall provide an inherently consistent subset of features defined in the following Table 48.

Table 48 – Class

No.	**Description** Keyword	**Explanation**
1	CLASS ... END_CLASS	Class definition
1a	FINAL specifier	Class cannot be used as a base class
	Adapted from function block	
2a	Declaration of variables VAR ... END_VAR	VAR B: REAL; END_VAR
2b	Initialization of variables	VAR B: REAL:= 12.1; END_VAR
3a	RETAIN qualifier on internal variables	VAR RETAIN X: REAL; END_VAR
3b	NON_RETAIN qualifier on internal variables	VAR NON_RETAIN X: REAL; END_VAR
4a	VAR_EXTERNAL declarations within class type declarations	For equivalent example see Table 40
4b	VAR_EXTERNAL CONSTANT declarations within class type declarations	For equivalent example see Table 40
	Methods and specifiers	
5	METHOD...END_METHOD	Method definition
5a	PUBLIC specifier	Method may be called from anywhere
5b	PRIVATE specifier	Method may only be called from inside the defining POU
5c	INTERNAL specifier	Method may only be called from inside the same namespace
5d	PROTECTED specifier	Method may only be called from inside the defining POU and its derivations (default)
5e	FINAL specifier	Method shall not be overridden
	Inheritance	- these features are the same as in Table 53 feature inheritance
6	EXTENDS	Class inherits from class (NOTE: no inheritance from FB)
7	OVERRIDE	Method overrides base method – see dynamic name binding
8	ABSTRACT	Abstract class – at least one method is abstract Abstract method – this method is abstract
	Access reference	
9a	THIS	Reference to own methods
9b	SUPER	Access reference to method in base class

No.	Description Keyword	Explanation
	Variable access specifiers	
10a	PUBLIC specifier	The variable may be accessed from anywhere
10b	PRIVATE specifier	The variable may only be accessed from inside the defining POU
10c	INTERNAL specifier	The variable may only be accessed from inside the same namespace
10d	PROTECTED specifier	The variable may only be accessed from inside the defining POU and its derivations (default)
	Polymorphism	
11a	with VAR_IN_OUT	VAR_IN_OUT of a (base) class may be assigned an instance of a derived class
11b	with reference	A reference to a (base) class may be assigned the address of an instance of a derived class

The example below illustrates the features of the class declaration and its usage.

EXAMPLE Class declaration

```
Class CCounter
  VAR
    m_iCurrentValue: INT;          (* Default = 0 *)
    m_bCountUp: BOOL:=TRUE;
  END_VAR
  VAR PUBLIC
    m_iUpperLimit: INT:=+10000;
    m_iLowerLimit: INT:=-10000;
  END_VAR

METHOD Count (* Only body *)
    IF (m_bCountUp AND m_iCurrentValue<m_iUpperLimit) THEN
        m_iCurrentValue:= m_iCurrentValue+1;
    END_IF;
    IF (NOT m_bCountUp AND m_iCurrentValue>m_iLowerLimit) THEN
        m_iCurrentValue:= m_iCurrentValue-1;
    END_IF;
END_METHOD

METHOD SetDirection
  VAR_INPUT
     bCountUp: BOOL;
  END_VAR
  m_bCountUp:=bCountUp;
END_METHOD

END_CLASS
```

6.6.5.3 Class instance declaration

A class instance shall be declared in a similar manner as defined for structured variables.

When a class instance is declared, the initial values for the public variables of the class instance can be assigned in a parenthesized initialization list following the assignment operator that follows the class identifier as shown in Table 49.

Elements which are not assigned in the initialization list shall have the initial values of the class declaration.

Table 49 – Class instance declaration

No.	Description	Example

1	Declaration of class instance(s) with default initialization	`VAR`
		` MyCounter1: CCounter;`
		`END_VAR`

2	Declaration of class instance with initialization of its public variables	`VAR`
		`MyCounter2: CCounter:=`
		`(m_iUpperLimit:=20000;`
		`        m_iLowerLimit:=-20000);`
		`END_VAR`

6.6.5.4 Methods of a class

6.6.5.4.1 General

For the purpose of the programmable controller languages the concept of methods well known in the object oriented programming is adopted as a set of optional language elements defined within the class definition.

Methods may be applied to define the operations to be performed on the class instance data.

6.6.5.4.2 Signature

For the purpose of this standard the term signature is defined in Clause 3 as a set of information defining unambiguously the identity of the parameter interface of a METHOD.

A signature consists of

- name of method,
- result type,
- variable names, data types and the order of all its parameters, i.e. inputs, outputs, in-out variables.

The local variables are not a part of the signature. VAR_EXTERNAL and constant variables are not relevant for the signature.

The access specifiers like PUBLIC or PRIVATE are not relevant for the signature.

6.6.5.4.3 Method declaration and execution

A class may have a set of methods.

The declaration of a method shall comply with the following rules:

1. The methods are declared within the scope of a class.
2. A method may be defined in any of the programming languages specified in this standard.
3. In the textual declaration the methods are listed after the declaration of the variables of the class.
4. A method may declare its own VAR_INPUT, internal temporary variables VAR and VAR_TEMP, VAR_OUTPUT, VAR_IN_OUT and a method result.

   The keywords VAR_TEMP and VAR have the same meaning and are both permitted for the internal variables. (VAR is used in functions).

5. The method declaration shall contain one of the following access specifiers: `PUBLIC`, `PRIVATE`, `INTERNAL`, and `PROTECTED`. If no access specifier is given, the method will be `PROTECTED` by default.
6. The method declaration may contain the additional keyword `OVERRIDE` or `ABSTRACT`.

   NOTE 1 Overloading of methods is not in the scope of this part of IEC 61131.

   The execution of a method shall comply with the following rules:

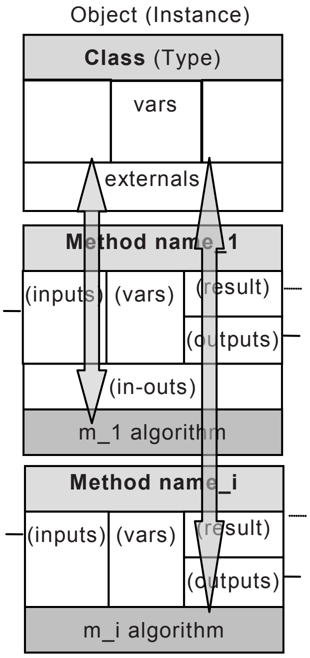
7. When executed, a method may read its inputs and calculates its outputs and its result using its temporary variables.
8. The method result is assigned to the method name.
9. All method variables and the result are temporary (like the variables of a function), i.e. the values are not stored from one method execution to the next. Therefore the evaluation of the method output variables is only possible in the immediate context of the method call.
10. The variable names of each method and of the class shall be different (unique).

The names of local variables of different methods may be the same.

11. All methods have read/write access to the static and external variables declared in the class.
12. All variables and results may be multi-valued, i.e. an array or a structure. As it is defined for functions the method result may be used as an operand in an expression.
13. When executed, a method may use other methods defined within this class. Methods of this class instance shall be called using the `THIS` keyword.

The following example illustrates the simplified declaration of a class with two methods and the call of the method.

EXAMPLE 1



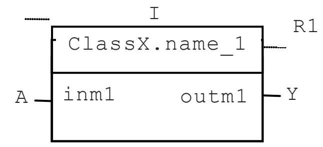
NOTE 2

The algorithms of the methods have access to their own data and to the class data.

(Temporary parameters are parenthesized.)

Declaration of the class (type) with methods:

```
CLASS name
 VAR vars; END_VAR
 VAR_EXTERNAL externals; END_VAR

METHOD name_1
 VAR_INPUT   inputs;   END_VAR
 VAR_OUTPUT  outputs;  END_VAR
END_METHOD

METHOD name_i
 VAR_INPUT   inputs;   END_VAR
 VAR_OUTPUT  outputs;  END_VAR
END_METHOD
END_CLASS
```



NOTE 3
This graphical representation of the method is for illustration only.

Call of a method:

a) Usage of the result: (result is optional)

```
 R1:= I.method1(inm1:= A, outm1 => Y);
```

b) Usage of call: (no result declared)

```
 I.method1(inm1:= A, outm1 => Y);
```

Assignment of method inputs from outside:
~~I.inm1 := A;~~ // Not permitted;

Read of method outputs from outside:
~~Y:= I.outm1;~~ // Not permitted,

EXAMPLE 2

Class COUNTER with two methods for counting up. Method UP5 shows how to call a method of the same class.

```
CLASS COUNTER
  VAR
    CV: UINT;                          // Current value of counter
    Max: UINT:= 1000;
  END_VAR

  METHOD PUBLIC UP: UINT               // Method for count up by inc
  VAR_INPUT INC: UINT; END_VAR         // Increment
  VAR_OUTPUT QU: BOOL; END_VAR         // Upper limit detection

    IF CV <= Max - INC                 // Count up of current value
       THEN CV:= CV + INC;
            QU:= FALSE;                // Upper limit reached
       ELSE QU:= TRUE;
    END_IF                             // Result of method
    UP:= CV;
  END_METHOD

  METHOD PUBLIC UP5: UINT              // Count up by 5
  VAR_OUTPUT QU: BOOL; END_VAR         // Upper limit reached
   UP5:= THIS.UP(INC:= 5, QU => QU);   // Internal method call
  END_METHOD
END_CLASS
```

6.6.5.4.4 Method call representation

The methods can be called in textual languages (Table 50) and in graphical languages.

In all language representations there are two different cases of calls of a method.

a) Internal call

   of a method of the own class instance. The method name shall be preceded by '`THIS.`'

   This call may be issued by another method.

b) External call

   of a method of an instance of another class. The method name shall be preceded by the instance name and '`.`'.

   This call may be issued by a method or a function block body where the instance is declared.

NOTE The following syntax is used:

- The syntax `A()` is used to call a global function A.
- The syntax `THIS.A()` is used to call a method of the own instance.
- The syntax `I1.A()` is used to call a method A of another instance I1.

6.6.5.4.5 Textual call representation

A method with result shall be called as an operand of an expression.

A method without result shall not be called inside an expression.

The method can be called formal or non-formal.

The external call of a method additionally needs the name of the external class instance.

EXAMPLE 1 `... class_instance_name.method_name(parameters)`

The internal call of a method is using `THIS` instead of the instance name.

EXAMPLE 2 `... THIS.method_name (parameters)`

Table 50 – Textual call of methods – Formal and non-formal parameter list

No.	Description	Example

1a	Complete formal call (textual only)
	Shall be used if `EN/ENO` is necessary in calls.
		`A:= COUNTER.UP(EN:= TRUE, INC:= B,`
		`START:= 1, ENO=> %MX1, QU => C);`

1b	Incomplete formal call (textual only) Shall be used if EN/ENO is not necessary in calls.	`A:= COUNTER.UP(INC:= B, QU => C);` `START` variable will have the default value `0` (zero).
2	Non-formal call (textual only) (fix order and complete)	`A:= COUNTER.UP(B, 1, C);` This call is equivalent to 1a, but without `EN/ENO`.

6.6.5.4.6 Graphical representation

The graphical representation of a method call is similar to the representation of a function or function block. It is a rectangular block with inputs on the left and outputs on the right side of the block.

The method calls may support `EN` and `ENO` as defined in Table 18.

- The internal call shows the class name and the method name separated with a period inside a block.

  The keyword `THIS` shall be located above the block.

- The external call shows the class name and the method name separated with a period inside a block.

  The class instance name shall be located above the block.

6.6.5.4.7 Error

The usage of a method output independent of the method call shall be treated as an error. See the example below.

EXAMPLE Internal and external method call

```
VAR
  CT:    COUNTER;
  LIMIT: BOOL;
  VALUE: UINT;
END_VAR
```

1) In Structured Text (ST)

a) Internal call of a method:

```
VALUE:= THIS.UP (INC:= 5, QU => LIMIT);
```

b) External call of a method:

```
VALUE:= CT.UP (INC:= 5, QU => LIMIT);
```

2) In Function Block Diagram (FBD)

a) Internal call of a method

```
            THIS
       +------------+
 On ---| COUNTER.UP |---VALUE
       |------------|
  5 ---|INC         |
       |          QU|---LIMIT
       +------------+
```

Called in a class by another method

THIS is mandatory
Method UP returns the result

The graphical representation is for illustration only
The variable On enables the method call

b) External call of a method:

```
             CT
       +------------+
 On ---| COUNTER.UP |---VALUE
       |------------|
  5 ---|INC         |
       |          QU|---LIMIT
       +------------+
```

CT is a class instance declared within another class or FB

Called by a method or function block body

Method UP returns the result

The graphical representation is for illustration only

The variable On enables the method call

3) Error: Method output usage without graphical and textual call

```
    |      ~~CT.UP~~       VALUE
    |-------| |---------( )---
    |
 VALUE:= ~~CT.UP~~;
```

This evaluation of the method output
is NOT possible because a method does not store its outputs from one execution to the next.

6.6.5.5 Class inheritance (EXTENDS, SUPER, OVERRIDE, FINAL)

6.6.5.5.1 General

For the purpose of the PLC languages the concept of inheritance defined in the general object oriented programming is here adapted as a way to create new elements.

The inheritance of classes is shown in Figure 17. Based on an existing class one or more classes may be derived. This may be repeated multiple times.

NOTE "Multiple inheritance" is not supported.

A derived (child) class typically extends the base (parent) class by additional methods.

The term "base" class stands for all "ancestors", i.e. for the parent and their parent classes etc.

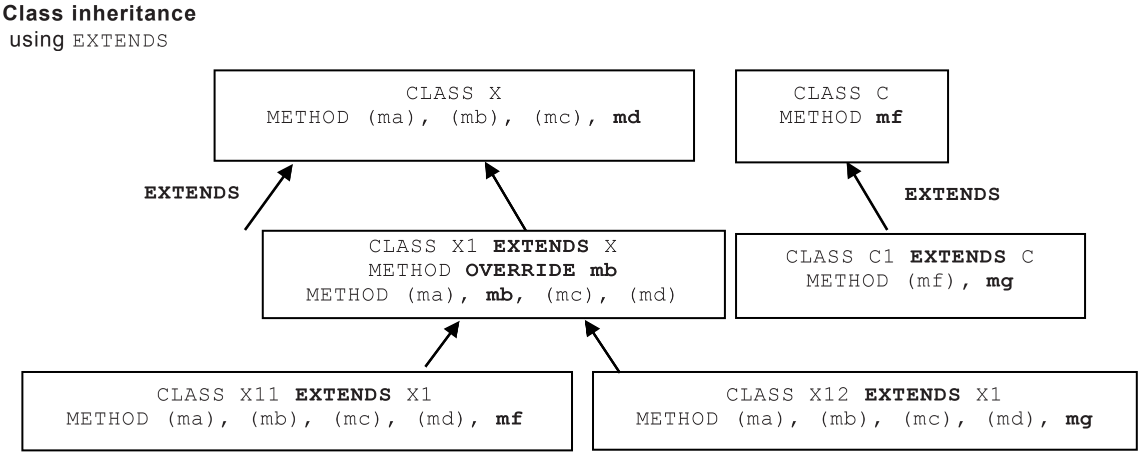


Figure 17 – Inheritance of classes (Illustration)

6.6.5.5.2 EXTENDS of class

A class may be derived from one already existing class (base class) using the keyword EXTENDS.

EXAMPLE `CLASS X1 EXTENDS X;`

The following rules shall apply:

1. The derived class inherits without further declarations all methods (if any) from its base class with the following exceptions.
   - PRIVATE methods are not inherited.
   - INTERNAL methods are not inherited outside the namespace.
2. The derived class inherits all variables (if any) from its base class.
3. A derived class inherits only from one base class.

   Multi-inheritance is not supported by this standard.

   NOTE A class can implement (using the keyword IMPLEMENTS) one or more interface(s).
4. The derived class may extend the base class, i.e. it may have own methods and variables in addition to the inherited methods, variables of the base class and thus create new functionality.
5. The class used as a base class may itself be a derived class. Then it passes also on to a derived class the methods and variables it has inherited.

   This may be repeated several times.
6. If the definition of the base class is changed, all derived classes (and their children) also change their functionality.

6.6.5.5.3 OVERRIDE a method

A derived class may override (replace) one or more inherited method(s) by an own implementation of the method(s). In order to override the base methods, the following rules apply:

1. The method that overrides an inherited method shall have the same signature (method name and variables) within the scope of the derived class.
2. The method that overrides an inherited method shall have the following features:
   - The keyword OVERRIDE follows the keyword METHOD.

- The derived class has access to the base method which is `PUBLIC` or `PROTECTED` or `INTERNAL` in the same namespace.
- The new method shall have the same access specifiers. But the method specifier `FINAL` may be used for an overridden method.

EXAMPLE `METHOD OVERRIDE mb;`

6.6.5.5.4 **`FINAL` for classes and methods**

A method with the specifier `FINAL` shall not be overridden.

A class with the specifier `FINAL` cannot be a base class.

EXAMPLE 1 `METHOD FINAL mb;`

EXAMPLE 2 `CLASS FINAL c1;`

6.6.5.5.5 Errors for `EXTENDS, SUPER, OVERRIDE, FINAL`

The following situation shall be treated as an error:

1. The derived class defines a variable with the name of a variable already contained in its base class, whether defined there or inherited. This rule does not apply on PRIVATE variables.
2. The derived class defines a method with the name of a variable already contained in its base class.
3. The derived class is derived from its own base class, whether directly or indirectly, i.e. recursion is not permitted.
4. The class defines a method with the keyword OVERRIDE which is not overriding a method of a base class.

EXAMPLE Inheritance and override

A class that extends the class LIGHTROOM.

```
CLASS LIGHTROOM
VAR LIGHT: BOOL; END_VAR

METHOD PUBLIC DAYTIME
  LIGHT:= FALSE;
END_METHOD

METHOD PUBLIC NIGHTTIME
  LIGHT:= TRUE;
END_METHOD
END_CLASS


CLASS LIGHT2ROOM EXTENDS LIGHTROOM
VAR LIGHT2: BOOL; END_VAR               // Second light

METHOD PUBLIC OVERRIDE DAYTIME
  LIGHT := FALSE;                       // Access to parent’s variable
  LIGHT2:= FALSE;                       //   specific implementation
END_METHOD

METHOD PUBLIC OVERRIDE NIGHTTIME
  LIGHT := TRUE;                        // Access to parent’s variable
  LIGHT2:= TRUE;                        //   specific implementation
END_METHOD

END_CLASS
```

6.6.5.6 Dynamic name binding (OVERRIDE)

Name binding is the association of a method name with a method implementation. The binding of a name (e.g. by the compiler) before the execution of the program is called static or “early” binding**.** A binding performed while the program is executed is called dynamic or “late” binding.

In case of an internal method call, the overriding feature with the keyword OVERRIDE causes a difference between the static and dynamic form of name binding:

- Static binding

  associates the method name to the method implementation of the class with an internal method call or contains the method doing the internal method call.

- Dynamic binding

  associates the method name to the method implementation of the actual type of the class instance.

EXAMPLE 1 Dynamic name binding

Overriding with effect on the binding.

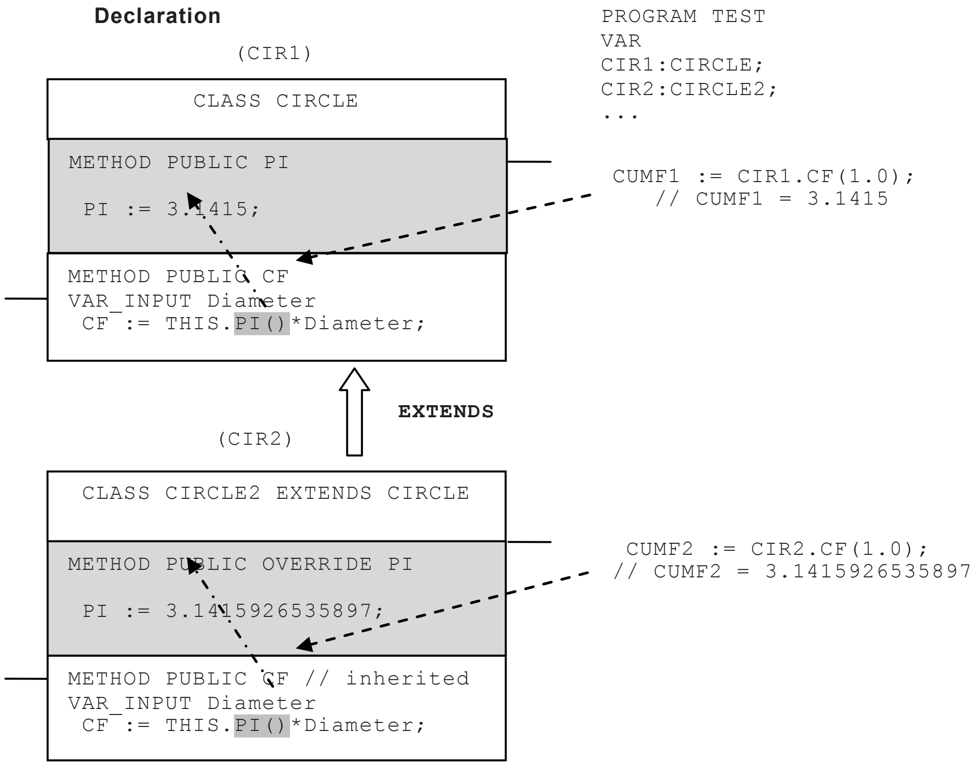
```
// Declaration
CLASS CIRCLE

METHOD PUBLIC PI: LREAL            // Method yields less accurate PI
  PI:= 3.1415;
END_METHOD

METHOD PUBLIC CF: LREAL            // Method yields circumference
  VAR_INPUT DIAMETER: LREAL; END_VAR
  CF:= THIS.PI() * DIAMETER;              // Internal call of method PI
END_METHOD                                // using dynamic binding of PI
END_CLASS


CLASS CIRCLE2 EXTENDS CIRCLE       // Class with method overriding PI

METHOD PUBLIC OVERRIDE PI: LREAL   // Method yields more accurate PI
  PI:= 3.1415926535897;
END_METHOD
END_CLASS

PROGRAM TEST
VAR
  CIR1:    CIRCLE;                        // Instance of CIRCLE
  CIR2:    CIRCLE2;                       // Instance of CIRCLE2
  CUMF1:   LREAL;
  CUMF2:   LREAL;
  DYNAMIC: BOOL;
END_VAR

  CUMF1:= CIR1.CF(1.0);                   // Call of method CIR1
  CUMF2:= CIR2.CF(1.0);                   // Call of method CIR2
  DYNAMIC:= CUMF1 <> CUMF2;        // Dynamic binding results in True

END_PROGRAM
```

In this example the class CIRCLE contains an internal call of its method PI with low accuracy to calculate the circumference (CF) of a circle.

The derived class CIRCLE2 overrides this method with a more accurate definition of PI.

The call of the method PI() refers either to CIRCLE.PI or to CIRCLE2.PI, according to the type of the instance on which the call of CF was performed. Here CUMF2 is more accurate than CUMF1.

EXAMPLE 2
Illustration of the textual example above (simplified)



```
CLASS CIRCLE

METHOD PUBLIC PI

 PI := 3.1415;

METHOD PUBLIC CF
VAR_INPUT Diameter
 CF := THIS.PI()*Diameter;
```

```
CLASS CIRCLE2 EXTENDS CIRCLE

METHOD PUBLIC OVERRIDE PI

 PI := 3.1415926535897;

METHOD PUBLIC CF // inherited
VAR_INPUT Diameter
 CF := THIS.PI()*Diameter;
```

```
PROGRAM TEST
VAR
CIR1:CIRCLE;
CIR2:CIRCLE2;
...

 CUMF1 := CIR1.CF(1.0);
    // CUMF1 = 3.1415

  CUMF2 := CIR2.CF(1.0);
 // CUMF2 = 3.1415926535897
```

6.6.5.7 Method call of own and base class (THIS, SUPER)

6.6.5.7.1 General

To access a method defined inside or outside the own class there are the keywords THIS and SUPER available.

6.6.5.7.2 THIS

THIS is a reference to the own class instance.

With the keyword THIS a method of the own class instance can be called by another method of this class instance.

THIS may be passed to a variable of the type of an INTERFACE.

The keyword THIS cannot be used with another instance e.g., the expression myInstance.THIS is not allowed.

EXAMPLE Usage of keyword THIS.

These examples are copied from examples above for convenience.

```
INTERFACE ROOM
   METHOD DAYTIME   END_METHOD          // Called during day-time
   METHOD NIGHTTIME END_METHOD          // Called during night-time
END_INTERFACE

FUNCTION_BLOCK ROOM_CTRL                //
  VAR_INPUT
    RM: ROOM;                           // Interface ROOM as type of input variable
  END_VAR
  VAR_EXTERNAL
    Actual_TOD: TOD;                    // Global time definition
  END_VAR
  IF (RM = NULL)                        // Important: test valid reference!
  THEN RETURN;
  END_IF;
  IF Actual_TOD >= TOD#20:15 OR Actual_TOD <= TOD#6:00
  THEN RM.NIGHTTIME();                  // call method of RM
  ELSE RM.DAYTIME();
  END_IF;
END_FUNCTION_BLOCK
```

// Applies keyword THIS to assign the own instance

```
CLASS DARKROOM IMPLEMENTS ROOM          // ROOM see above
VAR_EXTERNAL
  Ext_Room_Ctrl: ROOM_CTRL;             // ROOM_CTRL see above
END_VAR

METHOD PUBLIC DAYTIME;    END_METHOD
METHOD PUBLIC NIGHTTIME;  END_METHOD

METHOD PUBLIC EXT_1
  Ext_Room_Ctrl(RM:= THIS);             // Call Ext_Room_Ctrl with own instance
END_METHOD
END_CLASS
```

6.6.5.7.3 SUPER

SUPER offers access to methods of the base class implementation.

With the keyword SUPER a method which is valid in the base (parent) class instance can be called. Thus, static name binding takes place.

The keyword SUPER cannot be used with another instance e.g., the expression my-Room.SUPER.DAYTIME() is not allowed.

The keyword SUPER cannot be used to access further derived methods e.g., the expression SUPER.SUPER.aMethod is not supported.

EXAMPLE Usage of the keyword SUPER and polymorphism.

LIGHT2ROOM using SUPER as alternative implementation to the example above.
Some previous examples are copied here for convenience.

```
INTERFACE ROOM
   METHOD DAYTIME   END_METHOD  // Called during day-time
   METHOD NIGHTTIME END_METHOD  // Called during night-time
END_INTERFACE
```

```
CLASS LIGHTROOM IMPLEMENTS ROOM
VAR LIGHT: BOOL; END_VAR

METHOD PUBLIC DAYTIME
  LIGHT:= FALSE;
END_METHOD

METHOD PUBLIC NIGHTTIME
  LIGHT:= TRUE;
END_METHOD
END_CLASS


FUNCTION_BLOCK ROOM_CTRL
  VAR_INPUT
     RM: ROOM;                       // Interface ROOM as type of a variable
  END_VAR

  VAR_EXTERNAL
     Actual_TOD: TOD;                // Global time definition
  END_VAR

  IF (RM = NULL)                     // Important: test valid reference!
  THEN RETURN;
  END_IF;
  IF  Actual_TOD >= TOD#20:15 OR
     Actual_TOD <= TOD#06:00
  THEN RM.NIGHTTIME();               // Call method of RM (dynamic binding) to
                                     // either LIGHTROOM.NIGHTTIME
                                     // or LIGHT2ROOM.NIGHTTIME)
  ELSE RM.DAYTIME();
  END_IF;
END_FUNCTION_BLOCK
```

// Applies keyword SUPER to call a method of the base class

```
CLASS LIGHT2ROOM EXTENDS LIGHTROOM      // See above
VAR LIGHT2: BOOL; END_VAR               // Second light

METHOD PUBLIC OVERRIDE DAYTIME
  SUPER.DAYTIME();                   // Call of method in LIGHTROOM
  LIGHT2:= TRUE;
END_METHOD

METHOD PUBLIC OVERRIDE NIGHTTIME
  SUPER.NIGHTTIME()                  // Call of method in LIGHTROOM
  LIGHT2:= FALSE;
END_METHOD
END_CLASS
```

// Usage of polymorphism and dynamic binding

```
PROGRAM C
VAR
  MyRoom1: LIGHTROOM;                // See above
  MyRoom2: LIGHT2ROOM;               // See above
  My_Room_Ctrl: ROOM_CTRL;           // See above
END_VAR

 My_Room_Ctrl(RM:= MyRoom1);      // Calls in My_Room_Ctrl call methods of LIGHTROOM
 My_Room_Ctrl(RM:= MyRoom2);      // Calls in My_Room_Ctrl call methods of LIGHT2ROOM
END_PROGRAM
```

6.6.5.8 ABSTRACT class and ABSTRACT method

6.6.5.8.1 General

The ABSTRACT modifier may be used with classes or with single methods. The Implementer shall declare the implementation of these features according Table 48.

6.6.5.8.2 Abstract class

The use of the `ABSTRACT` modifier in a class declaration indicates that a class is intended to be a base type of other classes to be used for inheritance.

EXAMPLE `CLASS ABSTRACT A1`

The abstract class has the following features:

- An abstract class cannot be instantiated.
- An abstract class shall contain at least one abstract method.

A (non-abstract) class derived from an abstract class shall include actual implementations of all inherited abstract methods.

An abstract class may be used as a type of an input or in-out parameter.

6.6.5.8.3 Abstract method

All methods in an abstract class that are marked as `ABSTRACT` shall be implemented by classes that derive from the abstract class, if the derived class itself is not marked as `ABSTRACT`.

Methods of a class which are inherited from an interface shall get the keyword `ABSTRACT` if they are not yet implemented.

The keyword `ABSTRACT` shall not be used in combination with the keyword `OVERRIDE`.

The keyword `ABSTRACT` can only be used on methods of an abstract class.

EXAMPLE `METHOD PUBLIC ABSTRACT M1`

6.6.5.9 Method access specifiers (PROTECTED, PUBLIC, PRIVATE, INTERNAL)

For each method it shall be defined from where the call of the method is permitted. The accessibility of a method is defined by using one of the following access specifiers following the keyword `METHOD`.

- **PROTECTED**

  If inheritance is implemented then the access specifier `PROTECTED` is applicable. It indicates for methods that they are only accessible from inside a class and from inside all derived classes.

  `PROTECTED` is default and may be omitted.

  NOTE If inheritance is not supported, the default access specifier `PROTECTED` has the same effect as `PRIVATE`.

- **PUBLIC**

  The access specifier `PUBLIC` indicates for methods that they are accessible at any place where the class can be used.

- **PRIVATE**

  The access specifier `PRIVATE` indicates for methods that they are only accessible from inside the class itself.

- **INTERNAL**

If namespace is implemented then the access specifier INTERNAL is applicable. It indicates for methods that they are only accessible from within the NAMESPACE, in which the class is declared.

The access to method prototypes is implicitly always PUBLIC; therefore no access specifier is used on method prototypes.

All improper uses shall be treated as errors.

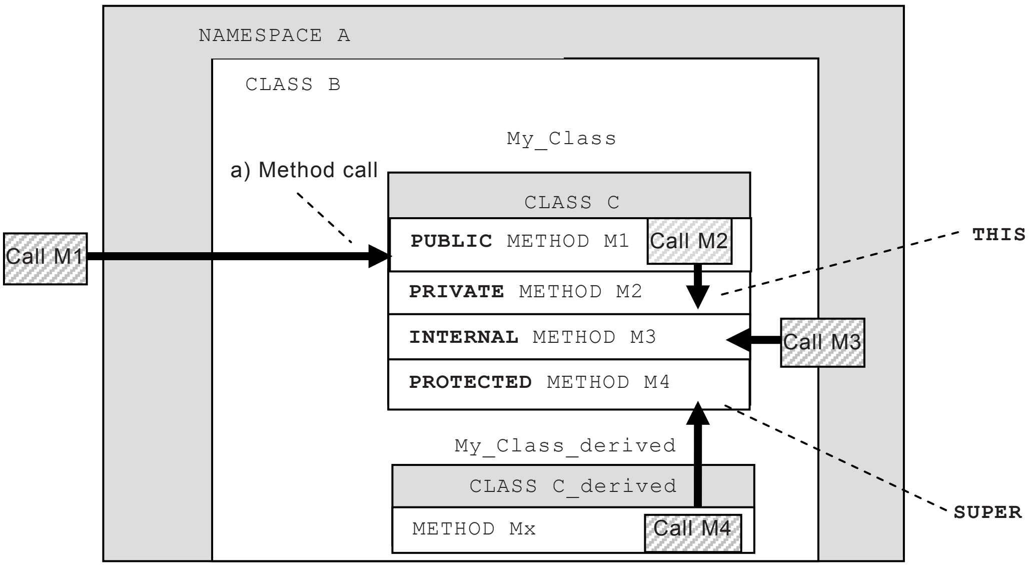
EXAMPLE Access specifier for methods.

Illustration of the accessibility (call) of methods defined in class C:

a) Access specifiers: PUBLIC, PRIVATE, INTERNAL, PROTECTED

- PUBLIC	M1 accessible by call M1 from inside class B (also class C)
- PRIVATE	M2 accessible by call M2 from inside class C only
- INTERNAL	M3 accessible by call M3 from inside NAMESPACE A (also class B , class C)
- PROTECTED	M4 accessible by call M4 from inside class C_derived (also class C)

b) Method calls inside/outside:
- M2 is called from inside class C – with keyword THIS.
- M1, M3 and M4 are class C called from outside class C – with keyword SUPER for M4.



6.6.5.10 Variable access specifiers (PROTECTED, PUBLIC, PRIVATE, INTERNAL)

For the VAR section it shall be defined from where the access of the variables of this section is permitted. The accessibility of the variables is defined by using one of the following access specifiers following the keyword VAR.

NOTE The access specifiers can be combined with other specifiers like RETAIN or CONSTANT in any order.

- **PROTECTED**

  If inheritance is implemented the access specifier PROTECTED is applicable. It indicates for variables that they are only accessible from inside a class and from inside all derived classes. PROTECTED is default and may be omitted.

  If inheritance is implemented but not used, PROTECTED has the same effect as PRIVATE.

- **PUBLIC**

The access specifier PUBLIC indicates for variables that they are accessible at any place where the class can be used.

- **PRIVATE**

  The access specifier PRIVATE indicates for variables that they are only accessible from inside the class itself.

  If inheritance is not implemented, PRIVATE is default and may be omitted.

- **INTERNAL**

  If namespace is implemented the access specifier INTERNAL is applicable. It indicates for variables that they are only accessible from within the NAMESPACE, in which the class is declared.

All improper uses shall be treated as errors.

6.6.6 Interface

6.6.6.1 General

In the object oriented programming the concept of interface is introduced to provide for separation of the interface specification from its implementation as a class. This allows different implementations of a common interface specification.

An interface definition starts with the keyword INTERFACE followed by the interface name and ends with the keyword END_INTERFACE (see Table 51).

The interface may contain a set of (implicitly public) method prototypes.

6.6.6.2 Usage of interface

The interface specification may be used in two ways:

a) In a class declaration.

   This specifies which methods the class shall implement; e.g. for reuse of the interface specification like illustrated in Figure 18.

b) As a type of a variable.

   Variables whose type is interface are references to instances of classes and shall be assigned before usage. Interfaces shall not be used as in-out variables.

Table 51 – Interface

No.	**Description** Keyword	**Explanation**
1	INTERFACE ... END_INTERFACE	Interface definition
	Methods and specifiers	
2	METHOD...END_METHOD	Method definition
	Inheritance	
3	EXTENDS	Interface inherits from interface
	Usage of interface	
4a	IMPLEMENTS interface	Implements an interface in a class declaration
4b	IMPLEMENTS multi-interfaces	Implements more than one interface in a class declaration
4c	Interface as type of a variable	Referencing an implementation (function block instance) of the interface

6.6.6.3 Method prototype

A method prototype is a restricted method declaration for the use with an interface. It contains the method name, VAR_INPUT, VAR_OUTPUT and VAR_IN_OUT variables and the method result. A method prototype definition does not contain any algorithm (code) and temporary variables; i.e. it does not yet include the implementation.

The access to method prototypes is implicitly always PUBLIC; therefore no access specifier is used in method prototypes.

Illustration of INTERFACE general_drive with
a) method prototypes (no algorithm)
b) class drive_A and class drive_B: IMPLEMENTS the INTERFACE general_drive.
These classes have methods with different algorithms.

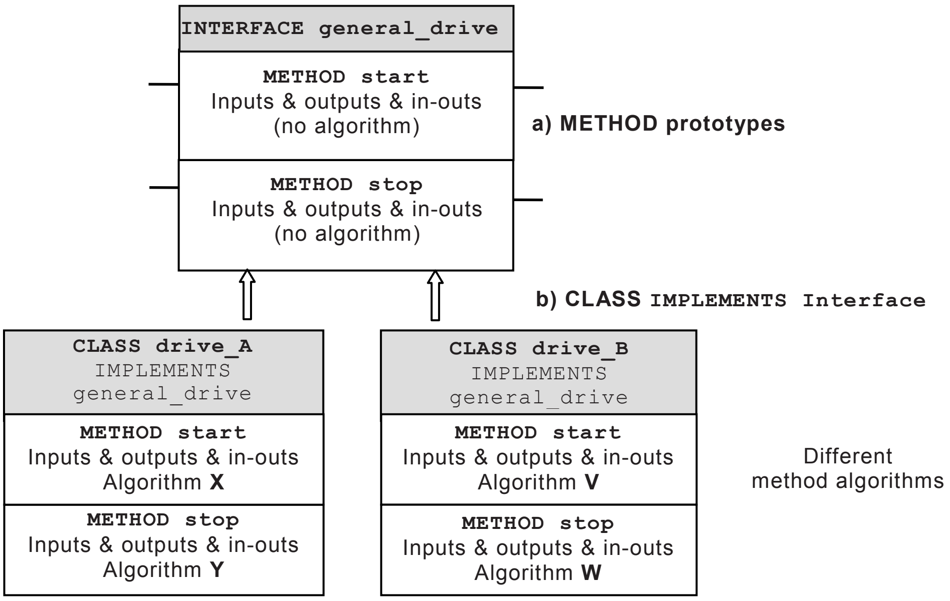


Figure 18 – Interface with derived classes (Illustration)

6.6.6.4 Usage of interface in a class declaration (IMPLEMENTS)

6.6.6.4.1 General

A class can implement one or more INTERFACE(s) by using the keyword IMPLEMENTS.

EXAMPLE `CLASS B IMPLEMENTS A1, A2;`

The class shall implement the algorithms of all methods specified by the method prototype(s) that are contained in the INTERFACE specification(s).

A class which does not implement all method prototypes shall be marked as ABSTRACT and cannot be instantiated.

NOTE The implementation of a method prototype can have additional temporary variables in the method.

6.6.6.4.2 Errors

The following situations shall be treated as an error:

1. If a class does not implement all methods defined in the base (parent) interface and the class is instantiated.

2. If a class implements a method with the same name as defined in the interface but with a different signature.
3. If a class implements a method with the same name as defined in the interface but not with the access specifier `PUBLIC` or `INTERNAL`.

6.6.6.4.3 Example

The example below illustrates the declaration of an interface in a class and the usage by an external method call.

EXAMPLE Class implements an interface

// Declaration

```
INTERFACE ROOM
  METHOD DAYTIME   END_METHOD             // Called in day-time
  METHOD NIGHTTIME END_METHOD             // in night-time
END_INTERFACE


CLASS LIGHTROOM IMPLEMENTS ROOM
  VAR LIGHT: BOOL; END_VAR

  METHOD PUBLIC DAYTIME
    LIGHT:= FALSE;
  END_METHOD

  METHOD PUBLIC NIGHTTIME
    LIGHT:= TRUE;
  END_METHOD
END_CLASS
```

// Usage (by an external method call)

```
PROGRAM A
  VAR  MyRoom: LIGHTROOM; END_VAR; // class instantiation
  VAR_EXTERNAL Actual_TOD: TOD; END_VAR;  // global time definition
  IF Actual_TOD >= TOD#20:15 OR Actual_TOD <= TOD#6:00
   THEN MyRoom.NIGHTTIME();
   ELSE MyRoom.DAYTIME();
  END_IF;
END_PROGRAM
```

6.6.6.5 Usage of interface as type of a variable

6.6.6.5.1 General

An interface may be used as the type of a variable. This variable is then a reference to an instance of a class implementing this interface. The variable shall be assigned to an instance of a class before it can be used. This rule applies for all cases where variables may be used.

The following values may be assigned to a variable of a type `INTERFACE`:

1. An instance of a class implementing the interface.
2. An instance of a class which is derived (by `EXTENDS`) from a class implementing the interface.
3. Another variable of the same or derived type `INTERFACE`.
4. The special value `NULL` indicating an invalid reference. This is also the initial value of the variable, if not initialized otherwise.

A variable of a type of an `INTERFACE` may be compared for equality with another variable of the same type. The result shall be `TRUE`, if the variables reference the same instance or if both variables equal to `NULL`.

6.6.6.5.2 Error

The variable of type interface shall be assigned before usage to verify that a valid class instance is assigned. Otherwise a runtime error will occur.

NOTE To avoid a runtime error the programming tool could provide a default “dummy” method. Another way is to check in advance if it is assigned.

6.6.6.5.3 Example

Examples 1 and 2 illustrate the declaration and usage of interfaces as type of a variable.

EXAMPLE 1 Function block type with calls of the methods of an interface

// Declaration

```
INTERFACE ROOM
  METHOD DAYTIME   END_METHOD      // called during day-time
  METHOD NIGHTTIME END_METHOD      // called during night-time
END_INTERFACE

CLASS LIGHTROOM IMPLEMENTS ROOM
  VAR LIGHT: BOOL; END_VAR

  METHOD PUBLIC DAYTIME
    LIGHT:= FALSE;
  END_METHOD

  METHOD PUBLIC NIGHTTIME
    LIGHT:= TRUE;
  END_METHOD
END_CLASS


FUNCTION_BLOCK ROOM_CTRL
  VAR_INPUT RM: ROOM; END_VAR
                            // Interface ROOM as type of (input) variable
  VAR_EXTERNAL
   Actual_TOD: TOD; END_VAR // Global time definition


  IF (RM = NULL)            // Important: test valid reference!
  THEN RETURN;
  END_IF;
  IF  Actual_TOD >= TOD#20:15 OR
      Actual_TOD <= TOD#06:00
  THEN RM.NIGHTTIME();      // Call method of RM
  ELSE RM.DAYTIME();
  END_IF;
END_FUNCTION_BLOCK
```

// Usage

```
PROGRAM B
  VAR                              // Instantiations
    My_Room:         LIGHTROOM;    // See LIGHTROOM IMPLEMENTS ROOM
    My_Room_Ctrl:    ROOM_CTRL;    // See ROOM_CTRL above
  END_VAR

  My_Room_Ctrl(RM:= My_Room);
                     // Calling FB with passing class instance as input
END_PROGRAM
```

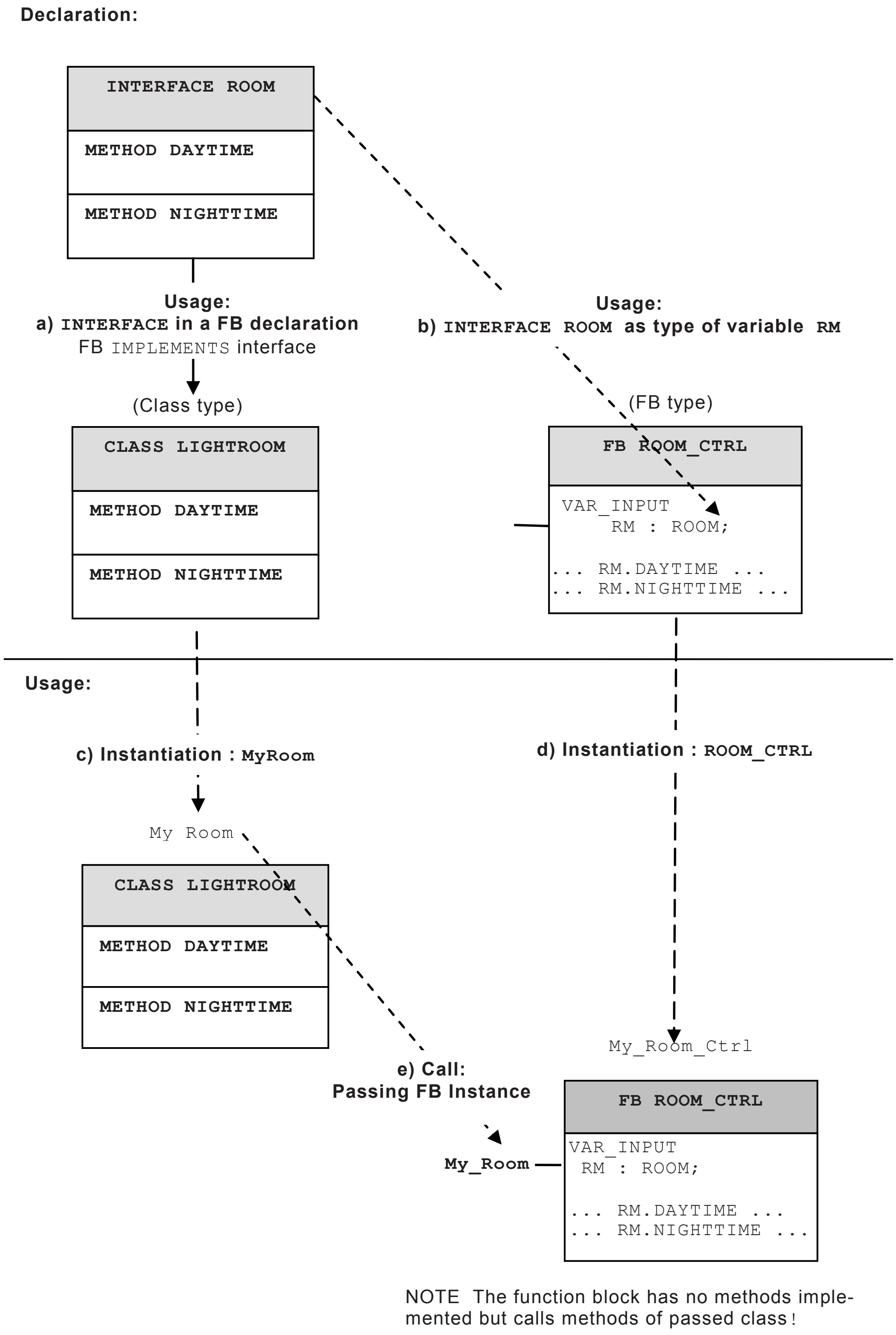
In this example a function block declares a variable of the type of an interface as parameter. The call of the function block instance passes (as function block input, output, in-out, or result) an instance (reference) of a class implementing the interface to this variable. Then the method called in the class uses the methods of the passed class instance. By this usage it is possible to pass instances of different classes implementing the interface.

Declaration:
Interface ROOM with two methods and class LIGHTROOM implementing the interface.

The function block ROOM_CTRL with input variable RM which has the type of interface ROOM.
ROOM_CTRL calls methods of the passed class which implements the interface.

Usage:
Program B instantiates the class My_Room and the function block My_Room_Ctrl
and calls the function block My_Room_Ctrl with passing the class My_Room to input variable RM of type interface ROOM.

EXAMPLE 2 Illustration of the relationship of Example 1 above.



6.6.6.6 Interface inheritance (EXTENDS)

6.6.6.6.1 General

For the purpose of the PLC languages the concept of inheritance and implementation defined in the general object oriented programming is here adopted as a way to create new elements as illustrated in Figure 19 a), b), c) below.

a) Interface inheritance

A derived (child) interface EXTENDS a base (parent) interface that has already been defined or

b) Class implementation

A derived class IMPLEMENTS one or more interface(s) that has/have already been defined or

c) Class inheritance

A derived class EXTENDS base class that has already been defined.

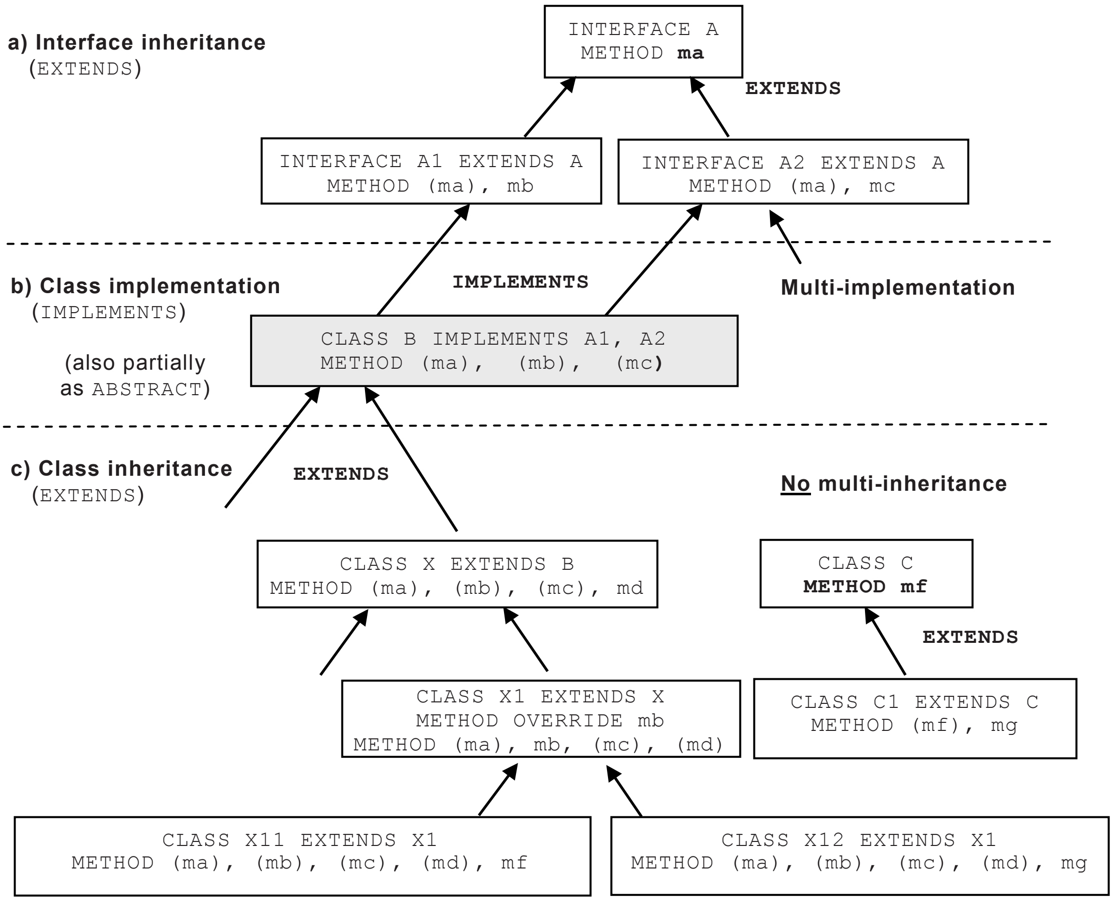


Illustration of the hierarchy of inheritance

a) Interface inheritance using keyword EXTENDS
b) Class implementation of interface(s) using keyword IMPLEMENTS
c) Class inheritance using keyword EXTENDS and OVERRIDE

Figure 19 – Inheritance of interface and class (Illustration)

The interface inheritance as shown in Figure 19 a) is the first of three inheritance/ implementation levels. Based on an existing interface one or more interfaces may be derived.

An interface may be derived from one or more already existing interface(s) (base interfaces) using the keyword EXTENDS.

EXAMPLE `INTERFACE A1 EXTENDS A`

The following rules shall apply:

1. The derived (child) interface inherits without further declarations all method prototypes from its base (parent) interfaces.
2. A derived interface can inherit from an arbitrary number of base interfaces.
3. The derived interface may extend the set of prototype methods; i.e. it may have method prototypes in addition to its base interface and thus create new functionality.
4. The interface used as a base interface, may itself be a derived interface. Then it passes on to its derived interfaces also the method prototypes it inherited.

   This may be repeated multiple times.
5. If the base interface changes its definition, all derived interfaces (and their children) have also this changed functionality.

6.6.6.6.2 Error

The following situation shall be treated as error:

1. An interface defines an additional method prototype (according rule 3) with the same name of a method prototype of one of its base interfaces.
2. An interface is its own base interface, whether directly or indirectly, i.e. recursion is not permitted.

NOTE The `OVERRIDE` feature, as defined in 6.6.5.5 for classes, is not applicable for interfaces.

6.6.6.7 Assignment attempt

6.6.6.7.1 General

The assignment attempt is used to check if the instance implements the given interface (Table 52). This is applicable for classes and function block types.

If the referenced instance is of a class or function block type that implements the interface, the result is a valid reference to this instance. Otherwise the result is `NULL`.

The assignment attempt syntax can also be used for safe casts from interface references to references to classes (or function block types), or from one reference to a base type to a reference to a derived type (downcast).

The result of an assignment attempt shall be checked to be unequal to `NULL` before used.

6.6.6.7.2 Textual representation

In Instruction List (IL) the operator “`ST?`” (Store) is used as shown in the following example.

EXAMPLE 1

```
LD  interface2       // in IL

ST? interface1
```

In Structured Text (ST) the operator “`?=`” is used as shown in the following example.

EXAMPLE 2

```
interface1 ?= interface2;   // in ST
```

6.6.6.7.3 Graphical representation

In graphical languages the following function is used:

EXAMPLE 1

```
                +---------------+
interface2 ---|      ?=       |--- interface1
                +---------------+
```

EXAMPLE 2 Assignment attempt with interface references

A successful and a failing assignment attempt with interface references
// Declaration

```
CLASS C IMPLEMENTS ITF1, ITF2
END_CLASS
```

// Usage

```
PROGRAM A
  VAR
     inst: C;
     interf1: ITF1;
     interf2: ITF2;
     interf3: ITF3;
  END_VAR

interf1:= inst;             // interf1 contains now a valid reference
interf2 ?= interf1;         // interf2 will contain a valid reference
                            // equal to interf2:= inst;
interf3 ?= interf1;         // interf3 will be NULL

END_PROGRAM
```

EXAMPLE 3 Assignment attempt with references

// Declaration

```
CLASS ClBase IMPLEMENTS ITF1, ITF2
END_CLASS

CLASS ClDerived EXTENDS ClBase
END_CLASS
```

// Usage

```
PROGRAM A
  VAR
     instbase: ClBase;
     instderived:ClDerived;
     rinstBase1, pinstBase2: REF_TO ClBase;
     rinstDerived1, rinstDerived2: REF_TO ClDerived;
     rinstDerived3, rinstDerived4: REF_TO ClDerived;
     interf1: ITF1;
     interf2: ITF2;
     interf3: ITF3;
  END_VAR

rinstBase1:= REF(instBase); // rinstbase1 references base class
rinstBase2:= REF(instDerived); // rinstbase2 references derived class

rinstDerived1 ?= rinstBase1; // rinstDerived1 == NULL
rinstDerived2 ?= rinstBase2; // rinstDerived2 will contain a valid
                             // reference to instDerived
interf1:= instbase;    // interf1 is a reference to base class
interf2:= instderived; // interf2 is a reference to derived class

rinstDerived3 ?= interf1;  // rinstDerived3 == NULL
rinstDerived4 ?= interf2;  // rinstDerived4 will contain a valid
                           // reference to instDerived
END_PROGRAM
```

The result of an assignment attempt shall be checked to be unequal to `NULL` before used.

Table 52 – Assignment attempt

No.	Description	Example
1	Assignment attempt with interfaces using `?=`	See above
2	Assignment attempt with references using `?=`	See above

6.6.7 Object oriented features for function blocks

6.6.7.1 General

The function block concept of IEC 61131-3:2003 is extended to support the object oriented paradigm using the concepts as defined for classes.

- Methods used additionally in function blocks
- Interfaces implemented additionally by function blocks
- Inheritance additionally of function blocks

For the object oriented function blocks all features of the function blocks defined in Table 40 are applicable.

Additionally the Implementer of object oriented function blocks shall provide an inherently consistent subset of the object oriented function block features defined in the following Table 53.

Table 53 – Object oriented function block

No.	**Description** Keyword	**Explanation**
1	Object oriented function block	Object oriented extension of the function block concept
1a	FINAL specifier	Function block cannot be used as a base function block.
	Methods and specifiers	
5	METHOD...END_METHOD	Method definition
5a	PUBLIC specifier	Method may be called from anywhere.
5b	PRIVATE specifier	Method may only be called from inside the defining POU.
5c	INTERNAL specifier	Method may only be called from inside the same namespace.
5d	PROTECTED specifier	Method may only be called from inside the defining POU and its derivations (default).
5e	FINAL specifier	Method shall not be overridden.
	Usage of interface	
6a	IMPLEMENTS interface	Implements an interface in a function block declaration
6b	IMPLEMENTS multi-interfaces	Implements more than one interface in a function block declaration
6c	Interface as type of a variable	Referencing an implementation (function block instance) of the interface
	Inheritance	
7a	EXTENDS	Function block inherits from base function bloc.k
7b	EXTENDS	Function block inherits from base class.
8	OVERRIDE	Method overrides base method – see dynamic name binding.
9	ABSTRACT	Abstract function block – at least one method is abstract. Abstract method – this method is abstract.
	Access reference	
10a	THIS	Reference to **own methods**
10b	SUPER	Access reference to method in base function block
10c	SUPER()	Access reference to body in base function block
	Variable access specifiers	
11a	PUBLIC specifier	The variable may be accessed from anywhere.
11b	PRIVATE specifier	The variable may only be accessed from inside the defining POU.
11c	INTERNAL specifier	The variable may only be accessed from inside the same namespace.
11d	PROTECTED specifier	The variable may only be accessed from inside the defining POU and its derivations (default).
	Polymorphism	
12a	with VAR_IN_OUT with equal signature	VAR_IN_OUT of a (base) FB type may be assigned an instance of a derived FB type without additional VAR_IN_OUT, VAR_INPUT or VAR_OUTPUT-variables.
12b	With VAR_IN_OUT with compatible signature	VAR_IN_OUT of a (base) FB type may be assigned an instance of a derived FB type without additional VAR_IN_OUT-variables.
12c	with reference with equal signature	A reference to a (base) FB type may be assigned the address of an instance of a derived FB type without additional VAR_IN_OUT, VAR_INPUT or VAR_OUTPUT-variables.

No.	**Description** Keyword	**Explanation**
12d	with reference with compatible signature	A reference to a (base) FB type may be assigned the address of an instance of a derived FB type without additional `VAR_IN_OUT` – variables.

6.6.7.2 Methods for function blocks

6.6.7.2.1 General

The concept of methods is adopted as a set of optional language elements defined within the function block type definition.

Methods may be applied to define the operations to be performed on the function block instance data.

6.6.7.2.2 Variants of a function block

A function block may have a function block body and additionally a set of methods. Since the FB body and/or the methods may be omitted, there are three variants of the function block. This is shown in the example in Figure 20 a), b), c).

a) Function block with a FB body only.

   This function block is known from the IEC 61131-3: 2003.

   In this case the function block has no methods implemented. The elements of the function block (inputs, outputs, etc.) and the call of the function block are shown in the example in Figure 20 a).

b) Function block with FB body and methods.

   Methods shall support the access to their own locally defined variables as well as to variables defined in the function block declaration sections of the var_inputs, the var_outputs or the vars.

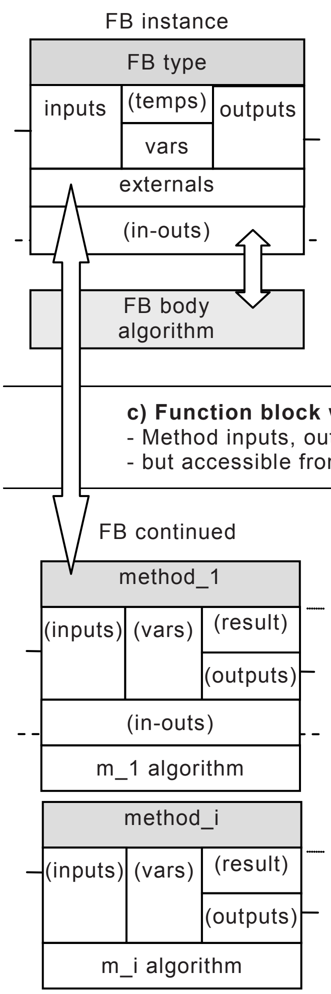
c) Function block with methods only.

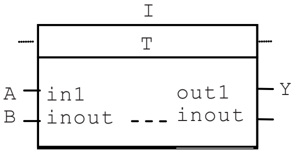
   In this case this function block has an empty function block body implemented. The elements of the function block and the call of a method are shown in the example in Figure 20 b)

   In this case this function block can also be declared as a class.

Illustration of the elements and the call of a function block with body and/or methods.
The example also shows the permitted and not permitted assignments and reads of inputs and outputs.

a) Function block with body only / Function block call:
- FB inputs, outputs are static and are accessible from outside
- also independent of the FB call.





This graphical representation of the method is for illustration only.

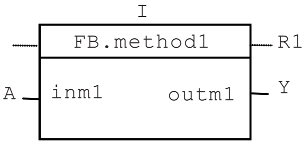
Temporary parameters are parenthesized.

`I (in1:= A, inout:= B, out1 => Y);`

Assignment of inputs from outside:
`I.in1 := A;`
~~I.inout:= B;~~ // Not permitted. In call only!

Read of outputs from outside:
`Y:= I.out1;` // Permitted! Different to b)

c) Function block with methods only (i.e. empty body) **/ Method call:**
- Method inputs, outputs, vars, and result are temporary (not static)
- but accessible from outside – in call only!



This graphical representation of the method is for illustration only.

Temporary parameters are parenthesized.

`R1:= I.method1(inm1:= A, outm1 => Y);`
or // usage of the result is optional

`I.method1(inm1:= A, outm1 => Y);`

Assignment of method inputs from outside:
~~I.inm1 := A;~~ // Not permitted; in call only!

Read of method outputs from outside:
~~Y:= I.outm1;~~ // Not permitted.In call only

b) Combined function block with body and methods: including a) and c)

Figure 20 – Function block with optional body and methods (Illustration)

6.6.7.2.3 Method declaration and execution

A function block may have a set of methods as illustrated in Figure 20 c).

The declaration of a method shall comply with the following rules additionally to the rules concerning methods of a class:

1. The methods are declared within the scope of a function block type.
2. In the textual declaration the methods are listed between the function block declaration part and the function block body.

The execution of a method shall comply with the following rules additionally to the methods of a class:

3. All methods have read/write access to the static variables declared in the function block: Inputs (if not of data type `BOOL R_EDGE` or `BOOL F_EDGE`), outputs, static variables and externals.

4. A method has no access to the temporary FB variables VAR_TEMP and the VAR_IN_OUT variables.
5. The method variables are not accessible by the FB body (algorithm).

6.6.7.2.4 Method call representation

The methods can be called as defined for classes in textual languages and in graphical languages.

6.6.7.2.5 Method access specifiers (PROTECTED, PUBLIC, PRIVATE, INTERNAL)

For each method it shall be defined from where the call of the method is permitted.

6.6.7.2.6 Variable access specifiers (PROTECTED, PUBLIC, PRIVATE, INTERNAL)

For the VAR section it shall be defined from where the access of the variables of this section is permitted.

The access to input and output variables is implicitly always PUBLIC, therefore no access specifier is used on input and output variable sections. Output variables are implicitly read-only. In-out variables can only be used in the function block body and within the call statement. The access to variables of the VAR_EXTERNAL section is implicitly always PROTECTED; therefore no access specifier shall be used on these variables.

6.6.7.2.7 Function block inheritance (EXTENDS, SUPER, OVERRIDE, FINAL)

6.6.7.2.8 General

The inheritance of function block is like the inheritance of classes. Based on an existing class or function block type one or more function block types may be derived. This may be repeated multiple times.

6.6.7.2.9 SUPER() in the body of a derived function block

The derived function blocks and their base function block may each have a function block body. The function block body is not automatically inherited from the base function block. It is empty by default. It can be called using SUPER().

In this case the rules above for EXTENDS of a function block and additionally the following rules apply:

1. The body (if any) of the derived function block type will be executed when the function block is called.
2. To execute additionally the body of the base function block (if any) in the derived function block the call of SUPER() shall be used. The call of SUPER() has no parameters.

The call SUPER() shall occur once in the function block body and shall not be in a loop.

3. The names of the variables in the base and the derived function blocks shall be unique.
4. The call of the function block shall be bound dynamically.
   a) A derived function block type can be used in all places where its base function block type can be used.
   b) A derived function block type can be used in all places where its base class type can be used.
5. SUPER() may only be called in the function block body, not in the method of a function block.

Figure 21 shows examples for SUPER():

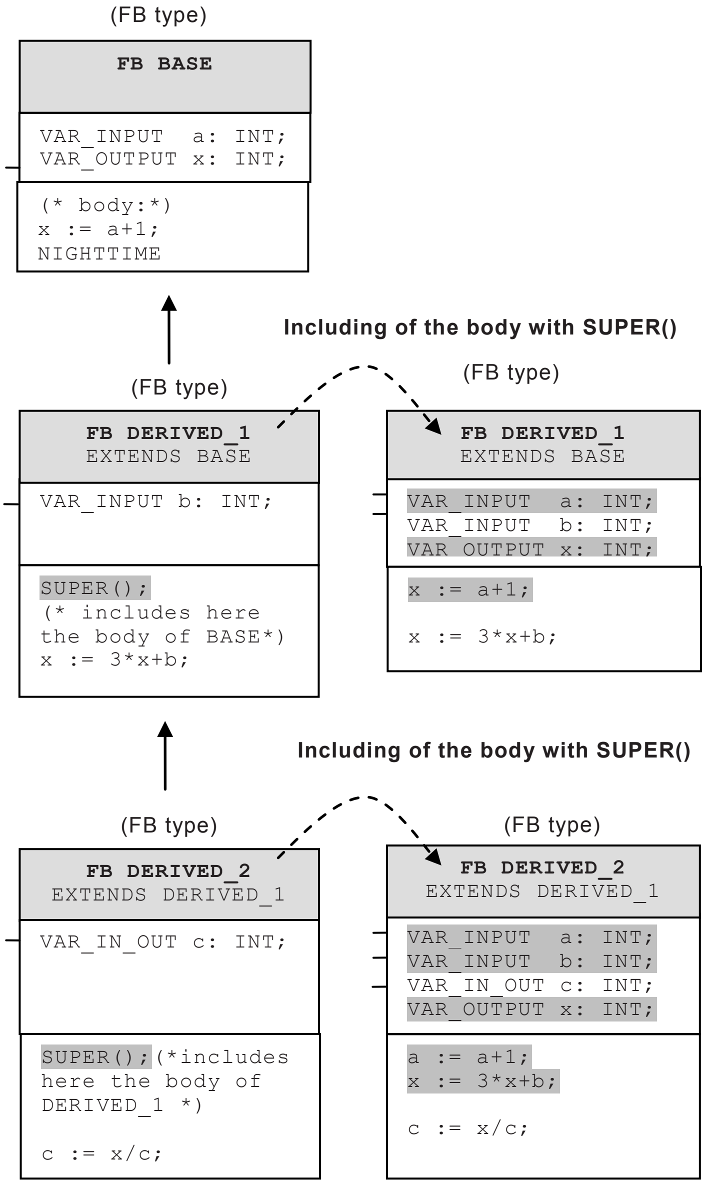


Figure 21 – Inheritance of function block body with `SUPER()` (Example)

6.6.7.2.10 `OVERRIDE` a method

A derived function block type may override (replace) one or more inherited method(s) by an own implementation of the method(s).

6.6.7.2.11 `FINAL` for function blocks and methods

A method with the specifier `FINAL` shall not be overridden.

A function block with the specifier `FINAL` cannot be a base function block.

6.6.7.3 Dynamic name binding (`OVERRIDE`)

Name binding is the association of a method name or function block name with a method or a function block implementation and is used as defined in 6.6.5.6 also for methods of function blocks.

6.6.7.4 Method call of own and base FB (THIS, SUPER) and polymorphism

To access a method defined inside or outside the own function block there are the keywords THIS and SUPER available.

6.6.7.5 ABSTRACT function block and ABSTRACT method

The ABSTRACT modifier may also be used with function blocks. The Implementer shall declare the implementation of these features.

6.6.7.6 Method access specifiers (PROTECTED, PUBLIC, PRIVATE, INTERNAL)

For each method it shall be defined from where the call of the method is permitted, as defined for classes.

6.6.7.7 Variable access specifiers (PROTECTED, PUBLIC, PRIVATE, INTERNAL)

For the VAR section it shall be defined from where the access of the variables of this section is permitted as defined in reference to classes.

The access to input and output variables is implicitly always PUBLIC, therefore no access specifier is used on input and output variable sections. Output variables are implicitly read-only. In-out variables can only be used in the function block body and within the call statement. The access to variables of the VAR_EXTERNAL section is implicitly always PROTECTED; therefore no access specifier shall be used on these variables.

6.6.8 Polymorphism

6.6.8.1 General

There are four cases in which polymorphism takes place, as shown in 6.6.8.2, 6.6.8.3, 6.6.8.4 and 6.6.8.5 below.

6.6.8.2 Polymorphism with INTERFACE

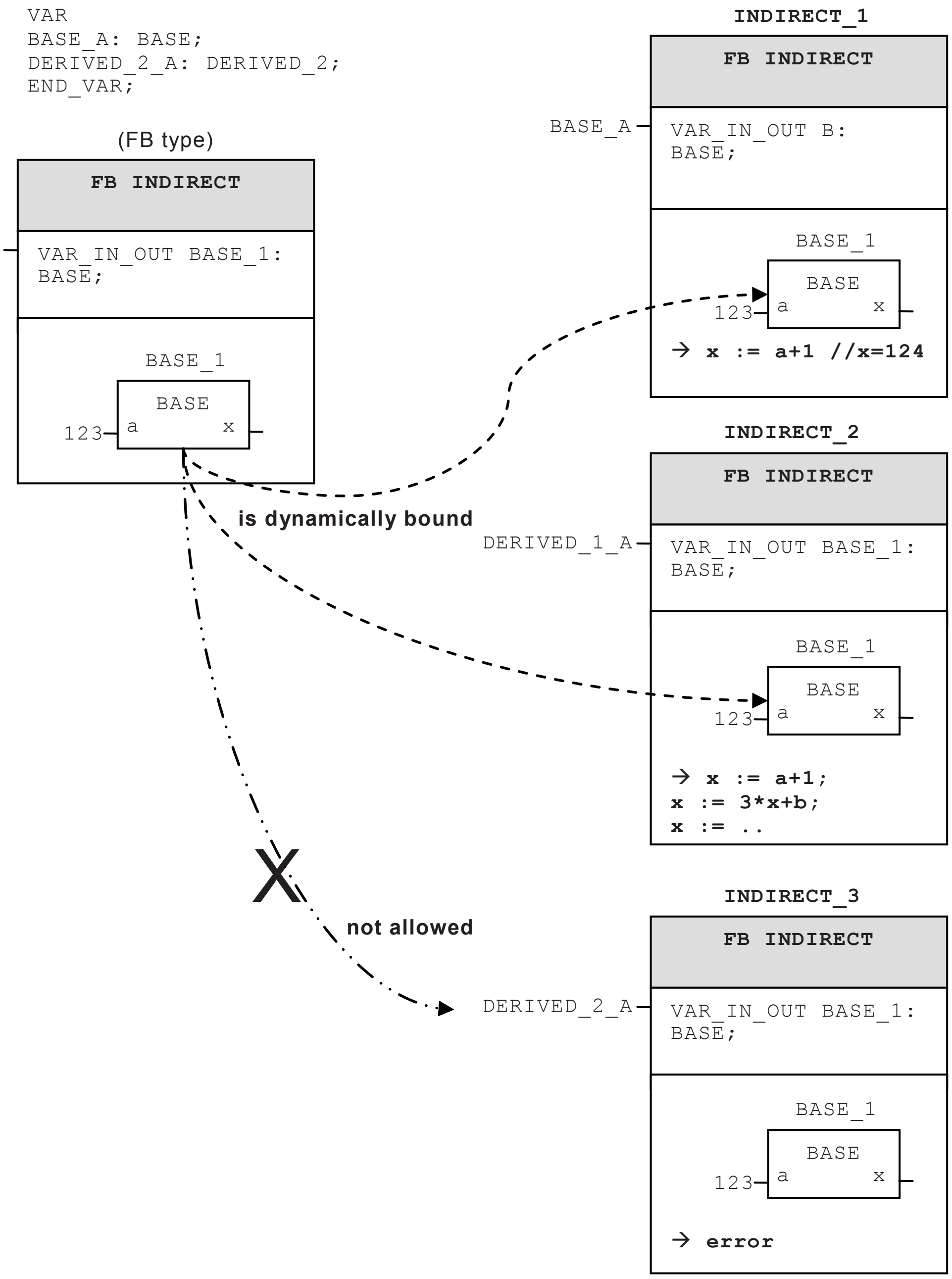
Since an interface cannot be instantiated, only derived types may be assigned to an interface reference. Thus, any call of a method via an interface reference is a case of dynamic binding.

6.6.8.3 Polymorphism with VAR_IN_OUT

An in-out variable of a type may be assigned an instance of a derived function block type, if the derived function block type has no additional in-out variables. Whether or not an instance of a derived function block type with additional input and output variables can be assigned is Implementer specific.

Thus, the call of a function block and the call of function block methods via a VAR_IN_OUT-instance are cases of dynamic binding.

EXAMPLE 1 Dynamic binding of function block calls



```
VAR
BASE_A: BASE;
DERIVED_2_A: DERIVED_2;
END_VAR;
```

If the derived function blocks added an in-out variable, then dynamic binding of the function block call would result in `INDIRECT_3` in the evaluation of the not assigned in-out variable c and would cause a runtime error. Therefore this assignment of the instance of the derived function blocks is an error.

EXAMPLE 2

```
CLASS LIGHTROOM
  VAR LIGHT: BOOL; END_VAR
  METHOD PUBLIC SET_DAYTIME
  VAR_INPUT: DAYTIME: BOOL; END_VAR
     LIGHT:= NOT(DAYTIME);
  END_METHOD
END_CLASS
```

```
CLASS LIGHT2ROOM EXTENDS LIGHTROOM
  VAR  LIGHT2: BOOL; END_VAR                    // Second light

  METHOD PUBLIC OVERRIDE SET_DAYTIME
   VAR_INPUT: DAYTIME: BOOL; END_VAR
    SUPER.SET_DAYTIME(DAYTIME);                 // Call of LIGHTROOM.SET_DAYTIME
    LIGHT2:= NOT(DAYTIME);
  END_METHOD
END_CLASS

FUNCTION_BLOCK ROOM_CTRL
  VAR_IN_OUT RM: LIGHTROOM; END_VAR
  VAR_EXTERNAL Actual_TOD: TOD; END_VAR   // Global time definition
             // In this case the class method to call is bound dynamically.
             // RM may refer to a derived class!

    RM.SET_DAYTIME(DAYTIME:= (Actual_TOD <= TOD#20:15) AND (Actual_TOD >= TOD#6:00));
END_FUNCTION_BLOCK
```

// Usage of polymorphism and dynamic binding with reference

```
PROGRAM D
VAR
  MyRoom1: LIGHTROOM;
  MyRoom2: LIGHT2ROOM;
  My_Room_Ctrl: ROOM_CTRL;
END_VAR

   My_Room_Ctrl(RM:= MyRoom1);
   My_Room_Ctrl(RM:= MyRoom2);
END_PROGRAM;
```

6.6.8.4 Polymorphism with reference

An instance of a derived type may be assigned to a reference to a base class.

A variable with a type may be assigned a reference to a derived function block type, if the derived function block type has no additional in-out variables. Whether or not a reference to derived function block type with additional input and output variables can be assigned is Implementer specific.

Thus, the call of a function block and the call of function block methods via a dereferentiation of a reference are cases of dynamic binding.

EXAMPLE 1 Alternative implementation of the lightroom example

```
FUNCTION_BLOCK LIGHTROOM
VAR LIGHT: BOOL; END_VAR
VAR_INPUT: DAYTIME: BOOL; END_VAR
LIGHT:= NOT(DAYTIME);
END_FUNCTION_BLOCK

FUNCTION_BLOCK LIGHT2ROOM EXTENDS LIGHTROOM
VAR LIGHT2: BOOL; END_VAR         // Second light
SUPER();                          // Call of LIGHTROOM
LIGHT2:= NOT(DAYTIME);
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK ROOM_CTRL
  VAR_INPUT RM: REF_TO LIGHTROOM; END_VAR
  VAR_EXTERNAL Actual_TOD: TOD; END_VAR // Global time definition


// in this case the function block to call is bound dynamically
// RM may refer to a derived function block type!

IF RM <> NULL THEN
      RM^.DAYTIME:= (Actual_TOD <= TOD#20:15) AND (Actual_TOD >= TOD#6:00));
END_IF
END_FUNCTION_BLOCK
```

// Usage of polymorphism and dynamic binding with reference

```
PROGRAM D
VAR
  MyRoom1: LIGHTROOM;             // see above
  MyRoom2: LIGHT2ROOM;            // see above
  My_Room_Ctrl: ROOM_CTRL;        // see above
END_VAR

My_Room_Ctrl(RM:= REF(MyRoom1));
My_Room_Ctrl(RM:= REF(MyRoom2));
END_PROGRAM;
```

6.6.8.5 Polymorphism with THIS

During runtime, THIS can hold a reference to the current function block type or to all of its derived function block types. Thus, any call of a function block method via THIS is a case of dynamic binding.

NOTE In special circumstances, e.g. if a function block type or a method is FINAL, or if there are no derived function block types, the type of an in-out variable, a reference or THIS can well be determined during compile time. In this case no dynamic binding is necessary.

6.7 Sequential Function Chart (SFC) elements

6.7.1 General

Subclause 6.7 defines sequential function chart (SFC) elements for use in structuring the internal organization of a programmable controller program organization unit, written in one of the languages defined in this standard, for the purpose of performing sequential control functions. The definitions in 6.7 are derived from IEC 60848, with the changes necessary to convert the representations from a documentation standard to a set of execution control elements for a programmable controller program organization unit.

The SFC elements provide a means of partitioning a programmable controller program organization unit into a set of steps and transitions interconnected by directed links. Associated with each step is a set of actions, and with each transition is associated a transition condition.

Since SFC elements require storage of state information, the program organization units which can be structured using these elements are function blocks and programs.

If any part of a program organization unit is partitioned into SFC elements, the entire program organization unit shall be so partitioned. If no SFC partitioning is given for a program organization unit, the entire program organization unit shall be considered to be a single action which executes under the control of the calling entity.

6.7.2 Steps

A step represents a situation in which the behavior of a program organization unit with respect to its inputs and outputs follows a set of rules defined by the associated actions of the step. A step is either active or inactive. At any given moment, the state of the program organization unit is defined by the set of active steps and the values of its internal and output variables.

As shown in Table 54, a step shall be represented graphically by a block containing a step name in the form of an identifier or textually by a STEP...END_STEP construction. The directed link(s) into the step can be represented graphically by a vertical line attached to the top of the step. The directed link(s) out of the step can be represented by a vertical line attached to the bottom of the step. Alternatively, the directed links can be represented textually by the TRANSITION... END_TRANSITION construct.

The step flag (active or inactive state of a step) can be represented by the logic value of a Boolean structure element ***.X, where *** is the step name, as shown in Table 54. This Boolean variable has the value 1 when the corresponding step is active and 0 when it is inactive. The state of this variable is available for graphical connection at the right side of the step as shown in Table 54.

Similarly, the elapsed time, ***.T, since initiation of a step can be represented by a structure element of type TIME, as shown in Table 54. When a step is deactivated, the value of the step elapsed time shall remain at the value it had when the step was deactivated. When a step is activated, the value of the step elapsed time shall be reset to t#0s.

The scope of step names, step flags, and step times shall be local to the program organization unit in which the steps appear.

The initial state of the program organization unit is represented by the initial values of its internal and output variables, and by its set of initial steps, i.e., the steps which are initially active. Each SFC network, or its textual equivalent, shall have exactly one initial step.

An initial step can be drawn graphically with double lines for the borders. When the character set defined in 6.1.1 is used for drawing, the initial step shall be drawn as shown in Table 54.

For system initialization the default initial elapsed time for steps is t#0s, and the default initial state is BOOL#0 for ordinary steps and BOOL#1 for initial steps. However, when an instance of a function block or a program is declared to be retentive for instance the states and (if supported) elapsed times of all steps contained in the program or function block shall be treated as retentive for system initialization.

The maximum number of steps per SFC and the precision of step elapsed time are implementation dependencies.

It shall be an error if:

1. an SFC network does not contain exactly one initial step;
2. a user program attempts to assign a value directly to the step state or the step time.

Table 54 – SFC step

<table>
<tr><th>No.</th><th>Description</th><th>Representation</th></tr>
<tr><td>1a</td><td>Step – graphical form with directed links</td><td> |
+------+
| *** |
+------+
   |</td></tr>
<tr><td>1b</td><td>Initial step – graphical form with directed link</td><td> |
+=======+
|| *** ||
|| ||
+=======+
    |</td></tr>
<tr><td>2a</td><td>Step – textual form without directed links</td><td>STEP ***:
  (* Step body *)
END_STEP</td></tr>
</table>

<table>
<tr><th>No.</th><th>Description</th><th>Representation</th></tr>
<tr><td>2b</td><td>Initial step – textual form without directed links</td><td><pre>INITIAL_STEP ***:

  (* Step body *)
END_STEP</pre></td></tr>
<tr><td>3a [a]</td><td>Step flag – general form ***.X = BOOL#1 when *** is active, BOOL#0 otherwise</td><td><pre>***.X</pre></td></tr>
<tr><td>3b [a]</td><td>Step flag – direct connection of Boolean variable ***.X to right side of step</td><td><pre> |
+-----+
| *** |----
+-----+
    |</pre></td></tr>
<tr><td>4 [a]</td><td>Step elapsed time – general form ***.T = a variable of type TIME</td><td><pre>***.T</pre></td></tr>
</table>

NOTE 1 The upper directed link to an initial step is not present if it has no predecessors.

NOTE 2 *** = step name

[a] When feature 3a, 3b, or 4 is supported, it shall be an error if the user program attempts to modify the associated variable. For example, if S4 is a step name, then the following statements would be errors in the ST language defined in 7.3:

```
S4.X:= 1; (* ERROR *)
S4.T:= t#100ms; (* ERROR *)
```

6.7.3 Transitions

A transition represents the condition whereby control passes from one or more steps preceding the transition to one or more successor steps along the corresponding directed link. The transition shall be represented by a horizontal line across the vertical directed link.

The direction of evolution following the directed links shall be from the bottom of the predecessor step(s) to the top of the successor step(s).

Each transition shall have an associated transition condition which is the result of the evaluation of a single Boolean expression. A transition condition which is always true shall be represented by the symbol 1 or the keyword TRUE.

A transition condition can be associated with a transition by one of the following means, as shown in Table 55:

a) By placing the appropriate Boolean expression in the ST language physically or logically adjacent to the vertical directed link.

b) By a ladder diagram network in the LD language physically or logically adjacent to the vertical directed link.

c) By a network in the FBD language defined in 8.3, physically or logically adjacent to the vertical directed link.

d) By a LD or FBD network whose output intersects the vertical directed link via a connector.

e) By a TRANSITION...END_TRANSITION construct using the ST language. This shall consist of:

- the keywords TRANSITION FROM followed by the step name of the predecessor step (or, if there is more than one predecessor, by a parenthesized list of predecessor steps);
- the keyword TO followed by the step name of the successor step (or, if there is more than one successor, by a parenthesized list of successor steps);

- the assignment operator (`:=`), followed by a Boolean expression in the ST language, specifying the transition condition;
- the terminating keyword `END_TRANSITION`.

f) By a `TRANSITION...END_TRANSITION` construct using the IL language. This shall consist of:

- the keywords `TRANSITION FROM` followed by the step name of the predecessor step (or, if there is more than one predecessor, by a parenthesized list of predecessor steps), followed by a colon (`:`);
- the keyword `TO` followed by the step name of the successor step (or, if there is more than one successor, by a parenthesized list of successor steps);
- beginning on a separate line, a list of instructions in the IL language, the result of whose evaluation determines the transition condition;
- the terminating keyword `END_TRANSITION` on a separate line.

g) By the use of a transition name in the form of an identifier to the right of the directed link. This identifier shall refer to a `TRANSITION...END_TRANSITION` construction defining one of the following entities, whose evaluation shall result in the assignment of a Boolean value to the variable denoted by the transition name:

- a network in the LD or FBD language;
- a list of instructions in the IL language;
- an assignment of a Boolean expression in the ST language.

The scope of a transition name shall be local to the program organization unit in which the transition is located.

It shall be an error if any “side effect” (for instance, the assignment of a value to a variable other than the transition name) occurs during the evaluation of a transition condition.

The maximum number of transitions per SFC and per step is Implementer specific.

Table 55 – SFC transition and transition condition

<table>
<tr><th>No.</th><th>Description</th><th>Example</th></tr>
<tr><td>1[a]</td><td>Transition condition physically or logically adjacent to the transition using ST language</td><td><pre>
   |
+-----+
|STEP7|
+-----+
   |
   + bvar1 & bvar2
   |
+-----+
|STEP8|
+-----+
   |
</pre></td></tr>
<tr><td>2[a]</td><td>Transition condition physically or logically adjacent to the transition using LD language</td><td><pre>
   |
+-----+
|STEP7|
+-----+
   |
   + bvar1 & bvar2
   |
+-----+
|STEP8|
+-----+
   |
</pre></td></tr>
</table>

<table>
<tr><th>No.</th><th>Description</th><th>Example</th></tr>
<tr><td>3[a]</td><td>Transition condition physically or logically adjacent to the transition using FBD language</td><td> |
                     +-----+
                     |STEP7|
         +--------+ +-----+
         | & | |
bvar1 ---| |-----+
bvar2 ---| | |
         +--------+ +-----+
                     |STEP8|
                     +-----+
                        |</td></tr>
<tr><td>4[a]</td><td>Use of connector</td><td> |
                     +-----+
                     |STEP7|
                     +-----+
                        |
   >TRANX>--------------+
                        |
                     +-----+
                     |STEP8|
                     +-----+
                        |</td></tr>
<tr><td>5[a]</td><td>Transition condition: Using LD language</td><td> | bvar1 bvar2
    +---||-----||---->TRANX>
    |</td></tr>
<tr><td>6[a]</td><td>Transition condition: Using FBD language</td><td>   +--------+
           | & |
  bvar1 ---| |-->TRANX>
  bvar2 ---| |
           +--------+</td></tr>
<tr><td>7[b]</td><td>Textual equivalent of feature 1 using ST language</td><td>STEP STEP7: END_STEP
TRANSITION FROM STEP7 TO STEP8
 := bvar1 & bvar2;
END_TRANSITION
STEP STEP8: END_STEP</td></tr>
<tr><td>8[b]</td><td>Textual equivalent of feature 1 using IL language</td><td>STEP STEP7: END_STEP
TRANSITION FROM STEP7 TO STEP 8:
  LD bvar1
  AND bvar2
END_TRANSITION

STEP STEP8: END_STEP</td></tr>
<tr><td>9[a]</td><td>Use of transition name</td><td> |
+-----+
|STEP7|
+-----+
  |
  + TRAN7 TO STEP8
  |
+-----+
|STEP8|
+-----+
|</td></tr>
<tr><td>10[a]</td><td>Transition condition using LD language</td><td>TRANSITION TRAN78 FROM STEP7 TO STEP8:
| |
| bvar1 bvar2   TRAN78 |
+---||-----||------( )---+
| |
END_TRANSITION</td></tr>
<tr><td>11[a]</td><td>Transition condition using FBD language</td><td>TRANSITION TRAN78 FROM STEP7 TO STEP8:
         +--------+
         | & |
bvar1 ---| |--TRAN78
bvar2 ---| |
         +--------+
END_TRANSITION</td></tr>
</table>

No.	Description	Example

12[b]	Transition condition using IL language	TRANSITION TRAN78 FROM STEP7 TO STEP8:
		      LD bvar1
		      AND bvar2
		END_TRANSITION

13[b]	Transition condition using ST language	TRANSITION TRAN78 FROM STEP7 TO STEP8
		 := bvar1 & bvar2;
		END_TRANSITION

a If feature 1 of Table 54 is supported, then one or more of features 1, 2, 3, 4, 5, 6, 9, 10 or 11 of this table shall be supported. b If feature 2 of Table 54 is supported, then one or more of features 7, 8, 12 or 13 of this table shall be supported.		

6.7.4 Actions

6.7.4.1 General

An action can be a Boolean variable, a collection of instructions in the IL language, a collection of statements in the ST language , a collection of rungs in the LD language, a collection of networks in the FBD language or a sequential function chart (SFC) organized .

Actions shall be declared via one or more of the mechanisms defined in 6.7.4.1 and shall be associated with steps via textual step bodies or graphical action blocks. Control of actions shall be expressed by action qualifiers.

It shall be an error if the value of a Boolean variable used as the name of an action is modified in any manner other than as the name of one or more actions in the same SFC.

A programmable controller implementation which supports SFC elements shall provide one or more of the mechanisms defined in Table 56 for the declaration of actions. The scope of the declaration of an action shall be local to the program organization unit containing the declaration.

6.7.4.2 Declaration

Zero or more actions shall be associated with each step. A step which has zero associated actions shall be considered as having a “WAIT” function, that is, waiting for a successor transition condition to become true.

Table 56 – SFC declaration of actions

No.	Description[a,b]	Example
1	Any Boolean variable declared in a VAR or VAR_OUTPUT block, or their graphical equivalents, can be an action.	
2l	Graphical declaration in LD language	+---+ | ACTION_4 | +---+ | | bvar1 bvar2 S8.X bOut1 | | | +---||------||----||------()----+ | | | | | | | +-------+ | | | +----|EN ENO| bvar2 | | | | C--| LT |-----------(S)----+ | | | D--| | | | | | +-------+ | | +---+

2s	Inclusion of SFC elements in action	`+-----------------------------------------+`
		`|              OPEN_VALVE_1               |`
		`+-----------------------------------------+`
		`|           | ...                         |`
		`| +=================+                     |`
		`| || VALVE_1_READY ||                     |`
		`| +=================+                     |`
		`|           |                             |`
		`|           + STEP8.X                     |`
		`|           |                             |`
		`| +-----------------+  +---+-----------+ |`
		`| | VALVE_1_OPENING |--| N |VALVE_1_FWD| |`
		`| +-----------------+  +---+-----------+ |`
		`|           | ...                         |`
		`+-----------------------------------------+`

2f	Graphical declaration in FBD language	`+-----------------------------------------+`
		`|                ACTION_4                 |`
		`+-----------------------------------------+`
		`|                 +---+                   |`
		`|          bvar1--| & |                   |`
		`|          bvar2--|   |-- bOut1            |`
		`|    S8.X---------|   |                   |`
		`|                 +---+    FF28           |`
		`|                         +----+          |`
		`|                         | SR |          |`
		`|               +------+  |  Q1|- bOut2   |`
		`|            C--|  LT  |--|S1  |          |`
		`|            D--|      |  +----+          |`
		`|               +------+                  |`
		`+-----------------------------------------+`

3s	Textual declaration in ST language	`ACTION ACTION_4:`
		`  bOut1:= bvar1 & bvar2 & S8.X;`
		`  FF28(S1:= (C<D));`
		`  bOut2:= FF28.Q;`
		`END_ACTION`

3i	Textual declaration in IL language	`ACTION     ACTION_4:`
		`  LD         S8.X`
		`  AND        bvar1`
		`  AND        bvar2`
		`  ST         bOut1`
		`  LD         C`
		`  LT         D`
		`  S1         FF28`
		`  LD         FF28.Q`
		`  ST         bOut2`
		`END_ACTION`

NOTE The step flag `S8.X` is used in these examples to obtain the desired result such that, when `S8` is deactivated, `bOut2:= 0`.

[a] If feature 1 of Table 54 is supported, then one or more of the features in this table, or feature 4 of Table 57, shall be supported.

[b] If feature 2 of Table 54 is supported, then one or more of features 1, 3s, or 3i of this table shall be supported.

6.7.4.3 Association with steps

A programmable controller implementation which supports SFC elements shall provide one or more of the mechanisms defined in Table 57 for the association of actions with steps. The maximum number of action blocks per step is an **implementation** dependency.

Table 57 – Step/action association

<table>
<tr><th>No.</th><th>Description</th><th>Example</th></tr>
<tr><td>1</td><td>Action block physically or logically adjacent to the step</td><td>

```
    |
  +----+  +-----+----------+---+
  | S8 |--|  L  | ACTION_1 |DN1|
  +----+  |t#10s|          |   |
    |     +-----+----------+---+
    + DN1
    |
```

</td></tr>
<tr><td>2</td><td>Concatenated action blocks physically or logically adjacent to the step</td><td>

```
    |
   +----+  +-----+---------------------+---+
   | S8 |--|  L  |      ACTION_1       |DN1|
   +----+  |t#10s|                     |   |
     |     +-----+---------------------+---+
     +DN1  |  P  |      ACTION_2       |   |
     |     +-----+---------------------+---+
     |     |  N  |      ACTION_3       |   |
     |     +-----+---------------------+---+
```

</td></tr>
<tr><td>3</td><td>Textual step body</td><td>

```
STEP S8:
  ACTION_1(L,t#10s,DN1);
  ACTION_2(P);
  ACTION_3(N);
END_STEP
```

</td></tr>
<tr><td>4 [a]</td><td>Action block "d" field</td><td>

```
     +-----+----------------------+---+
 ----| N   |      ACTION_4        |   |---
     +-----+----------------------+---+
     | bOut1:= bvar1 & bvar2 & S8.X; |
     | FF28 (S1:= (C<D));            |
     | bOut2:= FF28.Q;               |
     +-----+----------------------+---+
```

</td></tr>
<tr><td colspan="3">When feature 4 is used, the corresponding action name cannot be used in any other action block.</td></tr>
</table>

6.7.4.4 Action blocks

As shown in Table 58, an action block is a graphical element for the combination of a Boolean variable with one of the action qualifiers to produce an enabling condition, according to the rules for an associated action.

The action block provides a means of optionally specifying Boolean “indicator” variables, indicated by the “c” field in Table 58, which can be set by the specified action to indicate its completion, timeout, error conditions, etc. If the “c” field is not present, and the “b” field specifies that the action shall be a Boolean variable, then this variable shall be interpreted as the “c” variable when required. If the “c” field is not defined, and the “b” field does not specify a Boolean variable, then the value of the “indicator” variable is considered to be always `FALSE`.

When action blocks are concatenated graphically as illustrated in Table 57, such concatenations can have multiple indicator variables, but shall have only a single common Boolean input variable, which shall act simultaneously upon all the concatenated blocks.

The use of the “indicator”-variable is deprecated.

As well as being associated with a step, an action block can be used as a graphical element in the LD or FBD.

Table 58 – Action block

<table>
<tr><th>No.</th><th>Description</th><th>Graphical form/example</th></tr>
<tr><td>1 [a]</td><td>"a": Qualifier as per 6.7.4.5</td><td rowspan="3">   +------+-----------------+------+
---| "a" | "b" | "c" |---
   +------+-----------------+------+
   | "d" |
   | |
   +-------------------------------+</td></tr>
<tr><td>2</td><td>"b": Action name</td></tr>
<tr><td>3 [b]</td><td>"c": Boolean "indicator" variables (**deprecated**)</td></tr>
<tr><td></td><td>**"d": Action using:**</td><td rowspan="5"></td></tr>
<tr><td>4i</td><td>IL language</td></tr>
<tr><td>4s</td><td>ST language</td></tr>
<tr><td>4l</td><td>LD language</td></tr>
<tr><td>4f</td><td>FBD language</td></tr>
<tr><td>5l</td><td>Use of action blocks LD</td><td>| S8.X   bIn1   +---+-------+---+   OK1 |
+--| |----| |----| N | ACT1 |DN1|--( )--+
| +---+-------+---+ |</td></tr>
<tr><td>5f</td><td>Use of action blocks in FBD</td><td>   +----+   +---+-------+------+
S8.X ---| & |----| N | ACT1 | DN1 |---OK1
bIn1 ---| | +---+-------+------+
           +----+</td></tr>
<tr><td colspan="3">Field “a” can be omitted when the qualifier is “N”.
Field “c” can be omitted when no indicator variable is used.</td></tr>
</table>

6.7.4.5 Action qualifiers

Associated with each step/action association or each occurrence of an action block shall be an action qualifier. The value of this qualifier shall be one of the values listed in Table 59. In addition, the qualifiers L, D, SD, DS, and SL shall have an associated duration of type TIME.

Table 59 – Action qualifiers

No.	Description	Qualifier
1	Non-stored (null qualifier)	None
2	**N**on-stored	N
3	overriding **R**eset	R
4	**S**et (**S**tored)	S
5	time **L**imited	L
6	time **D**elayed	D
7	**P**ulse	P
8	**S**tored and time **D**elayed	SD
9	**D**elayed and **S**tored	DS
10	**S**tored and time **L**imited	SL
11	**P**ulse (rising edge)	P1
12	**P**ulse (falling edge)	P0

6.7.4.6 Action control

The control of actions shall be functionally equivalent to the application of the following rules:

a) Associated with each action shall be the functional equivalent of an instance of the ACTION_CONTROL function block defined in Figure 22 and Figure 23. If the action is declared as a Boolean variable, the Q output of this block shall be the state of this Boolean variable. If the action is declared as a collection of statements or networks, then this collection shall be executed continually while the A (activation) output of the ACTION_CONTROL function block stands at BOOL#1. In this case, the state of the output Q (called the "action flag") can be accessed within the action by reading a read-only Boolean variable which has the form of a reference to the Q output of a function block instance whose instance name is the same as the corresponding action name, for example, ACTION1.Q.

   The Implementer may opt for a simpler implementation as shown in Figure 23 b). In this case, if the action is declared as a collection of statements or networks, then this collection shall be executed continually while the Q output of the ACTION_ CONTROL function block stands at BOOL#1. In any case, the Implementer shall specify which one of the features given in Table 60 is supported.

   NOTE 1 The condition Q=FALSE will ordinarily be used by an action to determine that it is being executed for the final time during its current activation.

   NOTE 2 The value of Q will always be FALSE during execution of actions called by P0 and P1 qualifiers.

   NOTE 3 The value of A will be TRUE for only one execution of an action called by a P1 or P0 qualifier. For all other qualifiers, A will be true for one additional execution following the falling edge of Q.

   NOTE 4 Access to the functional equivalent of the Q or A outputs of an ACTION_CONTROL function block from outside of the associated action is an Implementer specific feature.

b) A Boolean input to the ACTION_CONTROL block for an action shall be said to have an association with a step or with an action block, if the corresponding qualifier is equivalent to the input name (N, R, S, L, D, P, P0, P1, SD, DS, or SL). The association shall be said to be active if the associated step is active, or if the associated action block's input has the value BOOL#1. The active associations of an action are equivalent to the set of active associations of all inputs to its ACTION_CONTROL function block.

   A Boolean input to an ACTION_CONTROL block shall have the value BOOL#1 if it has at least one active association and the value BOOL#0 otherwise.

c) The value of the T input to an ACTION_CONTROL block shall be the value of the duration portion of a time-related qualifier (L, D, SD, DS, or SL) of an active association. If no such association exists, the value of the T input shall be t#0s.

d) It shall be an error if one or more of the following conditions exist:

   - More than one active association of an action has a time-related qualifier (L, D, SD, DS, or SL).
   - The SD input to an ACTION_CONTROL block has the BOOL#1 when the Q1 output of its SL_FF block has the value BOOL#1.
   - The SL input to an ACTION_CONTROL block has the value BOOL#1 when the Q1 output of its SD_FF block has the value BOOL#1.

e) It is not required that the ACTION_CONTROL block itself be implemented, but only that the control of actions be equivalent to the preceding rules. Only those portions of the action control appropriate to a particular action need be instantiated, as illustrated in Figure 24. In particular, note that simple MOVE (:=) and Boolean OR functions suffice for control of Boolean variable actions if the latter's associations have only “N” qualifiers.

Figure 22 and Figure 23 summarize the parameter interface and the body of the ACTION_CONTROL function block. Figure 24 shows an example of the action control.

```
       +----------------+
       | ACTION_CONTROL |
BOOL   |N              Q|---BOOL
BOOL---|R              A|---BOOL
BOOL---|S               |
BOOL---|L               |
BOOL---|D               |
BOOL---|P               |
BOOL---|P1              |
BOOL---|P0              |
BOOL---|SD              |
BOOL---|DS              |
BOOL---|SL              |
TIME---|T               |
       +----------------+
```

a) With “final scan” logic

```
       +----------------+
       | ACTION_CONTROL |
BOOL   |N              Q|---BOOL
BOOL---|R               |
BOOL---|S               |
BOOL---|L               |
BOOL---|D               |
BOOL---|P               |
BOOL---|P1              |
BOOL---|P0              |
BOOL---|SD              |
BOOL---|DS              |
BOOL---|SL              |
TIME---|T               |
       +----------------+
```

b) Without “final scan” logic

NOTE These interfaces are not visible to the user.

Figure 22 – ACTION_CONTROL function block – External interface (Summary)

```
                                                          +---+
   +-----------------------------------------------------O| & |---Q
   |                                             +-----+  |   |
N--|---------------------------------------------| >=1 |--|   |
   |                       S_FF                  |     |  +---+
R--+                      +----+                 |     |
   |                      | RS |                 |     |
S--|----------------------|S Q1|-----------------|     |
   +----------------------|R1  |                 |     |
   |                      +----+  +---+          |     |
L--|---------+--------------------| & |----------|     |
   |         |       L_TMR    +--O|   |          |     |
   |         |      +-----+   |   +---+          |     |
   |         |      | TON |   |                  |     |
   |         +------|IN  Q|---+      D_TMR       |     |
   |  +-------------|PT   |         +-----+      |     |
   |  |             +-----+         | TON |      |     |
D--|--|-----------------------------|IN  Q|------|     |
   |  +-----------------------------|PT   |      |     |
   |  |              P_TRIG         +-----+      |     |
   |  |            +--------+                    |     |
   |  |            | R_TRIG |                    |     |
P--|--|------------|CLK    Q|--------------------|     |
   |  |    SD_FF   +--------+   SD_TMR           |     |
   |  |   +----+                +-----+          |     |
   |  |   | RS |                | TON |          |     |
SD-|--|---|S Q1|----------------|IN  Q|----------|     |
   +--|---|R1  |   +------------|PT   |          |     |
   |  |   +----+   |   DS_TMR   +-----+  DS_FF   |     |
   |  +------------+   +-----+          +----+   |     |
   |  |                | TON |          | RS |   |     |
DS-|--|----------------|IN  Q|----------|S Q1|---|     |
   |  +----------------|PT   |      +---|R1  |   |     |
   |  |                +-----+      |   +----+   |     |
   +--|-----------------------------+            |     |
   |  |         SL_FF                            |     |
   |  |        +----+                            |     |
   |  |        | RS |                     +---+  |     |
SL-|--|--------|S Q1|--+------------------| & |--|     |
   +--|--------|R1  |  |     SL_TMR   +--O|   |  +-----+
      |        +----+  |    +-----+   |   +---+
      |                |    | TON |   |
      |                +----|IN  Q|---+                     +-----+
T-----+---------------------|PT   |      +--------+         | >=1 |
                            +-----+      | F_TRIG |     Q---|     |---A
                +--------+           Q---|CLK    Q|---------|     |
                | R_TRIG |               +--------+         |     |
P1--------------|CLK    Q|----------------------------------|     |
                +--------+     +--------+                   |     |
                               | F_TRIG |                   |     |
P0-----------------------------|CLK    Q|-------------------|     |
                               +--------+                   +-----+
```

a) Body with “final scan” logic

```
                                                          +---+
   +-----------------------------------------------------O| & |---Q
   |                                             +-----+  |   |
N--|---------------------------------------------| >=1 |--|   |
   |                       S_FF                  |     |  +---+
R--+                      +----+                 |     |
   |                      | RS |                 |     |
S--|----------------------|S Q1|-----------------|     |
   +----------------------|R1  |                 |     |
   |                      +----+  +---+          |     |
L--|---------+--------------------| & |----------|     |
   |         |       L_TMR    +--O|   |          |     |
   |         |      +-----+   |   +---+          |     |
   |         |      | TON |   |                  |     |
   |         +------|IN  Q|---+      D_TMR       |     |
   |  +-------------|PT   |         +-----+      |     |
   |  |             +-----+         | TON |      |     |
D--|--|-----------------------------|IN  Q|------|     |
   |  +-----------------------------|PT   |      |     |
   |  |              P_TRIG         +-----+      |     |
   |  |            +--------+                    |     |
   |  |            | R_TRIG |                    |     |
P--|--|------------|CLK    Q|--------------------|     |
   |  |    SD_FF   +--------+   SD_TMR           |     |
   |  |   +----+                +-----+          |     |
   |  |   | RS |                | TON |          |     |
SD-|--|---|S Q1|----------------|IN  Q|----------|     |
   +--|---|R1  |   +------------|PT   |          |     |
   |  |   +----+   |   DS_TMR   +-----+  DS_FF   |     |
   |  +------------+   +-----+          +----+   |     |
   |  |                | TON |          | RS |   |     |
DS-|--|----------------|IN  Q|----------|S Q1|---|     |
   |  +----------------|PT   |      +---|R1  |   |     |
   |  |                +-----+      |   +----+   |     |
   +--|-----------------------------+            |     |
   |  |         SL_FF                            |     |
   |  |        +----+                            |     |
   |  |        | RS |                     +---+  |     |
SL-|--|--------|S Q1|--+------------------| & |--|     |
   +--|--------|R1  |  |     SL_TMR   +--O|   |  |     |
      |        +----+  |    +-----+   |   +---+  |     |
      |                |    | TON |   |          |     |
      |                +----|IN  Q|---+          |     |
T-----+---------------------|PT   |              |     |
          +--------+        +-----+              |     |
          | R_TRIG |                             |     |
P1--------|CLK    Q|-----------------------------|     |
          +--------+     +--------+              |     |
                         | F_TRIG |              |     |
P0-----------------------|CLK    Q|--------------|     |
                         +--------+              +-----+
```

b) Body without “final scan” logic

NOTE 1 Instances of these function block types are not visible to the user.

NOTE 2 The external interfaces of these function block types are given above.

Figure 23 – ACTION_CONTROL function block body (Summary)

```
   |
+-----+   +---+------------+----------------+
| S22 |---| N | HV_BREAKER | HV_BRKR_CLOSED |
+-----+   +---+------------+----------------+
   |      | S | START_INDICATOR             |
   |      +---+-----------------------------+
   + HV_BRKR_CLOSED
   |
+-----+   +----+---------------+
| S23 |---| SL | RUNUP_MONITOR |
+-----+   |t#1m|               |
   |      +----+---------------+
   |      | D  | START_WAIT    |
   |      |t#1s|               |
   |      +----+---------------+
   + START_WAIT
   |
+-----+   +-----+-----------------+------------------+
| S24 |---| N   | ADVANCE_STARTER | STARTER_ADVANCED |
+-----+   +-----+-----------------+------------------+
   |      | L   | START_MONITOR                      |
   |      |t#30s|                                    |
   |      +-----+------------------------------------+
   + STARTER_ADVANCED
   |
+-----+   +-----+-----------------+-------------------+
| S26 |---| N   | RETRACT_STARTER | STARTER_RETRACTED |
+-----+   +-----+-----------------+-------------------+
   |
   |
   + STARTER_RETRACTED
   |
+-----+   +-----+-----------------+
| S27 |---| R   | START_INDICATOR |
+-----+   +-----+-----------------+
   |      | R   | RUNUP_MONITOR   |
   |      +-----+-----------------+
```

a) SFC representation

```
S22.X--------------------------------------------------HV_BREAKER

S24.X---------------------------------------------ADVANCE_STARTER

S26.X---------------------------------------------RETRACT_STARTER

                      START_INDICATOR_S_FF
                            +----+
                            | RS |
S22.X-----------------------|S Q1|----------------START_INDICATOR
S27.X-----------------------|R1  |
                            +----+

                       START_WAIT_D_TMR
                            +-----+
                            | TON |
S23.X-----------------------|IN  Q|---------------------START_WAIT
t#1s------------------------|PT   |
                            +-----+

RUNUP_MONITOR_SL_FF
        +----+
        | RS |                              +---+
S23.X---|S Q1|--+-----------------------------| & |--RUNUP_MONITOR
S27.X---|R1  |  |   RUNUP_MONITOR_SL_TMR  +--O|   |
        +----+  |         +-----+         |   +---+
                |         | TON |         |
                +---------|IN  Q|---------+
t#1m----------------------|PT   |
                          +-----+

                                                +---+
S24.X------------+-----------------------------| & |---START_MONITOR
                  | START_MONITOR_L_TMR  +---O|   |
                  |         +-----+       |    +---+
                  |         | TON |       |
                  +---------|IN  Q|-------+
t#30s----------------------|PT   |
                            +-----+
```

b) Functional equivalent

NOTE The complete SFC network and its associated declarations are not shown in this example.

Figure 24 – Action control (Example)

Table 60 shows the two possible action control features.

Table 60 – Action control features

No.	Description	Reference
1	With final scan	per Figure 22 a) and Figure 23 a)
2	Without final scan	per Figure 22 b) and Figure 23 b)
These two features are mutually exclusive, i.e., only one of the two shall be supported in a given SFC implementation.		

6.7.5 Rules of evolution

The initial situation of a SFC network is characterized by the initial step which is in the active state upon initialization of the program or function block containing the network.

Evolutions of the active states of steps shall take place along the directed links when caused by the clearing of one or more transitions.

A transition is enabled when all the preceding steps, connected to the corresponding transition symbol by directed links, are active. The crossing of a transition occurs when the transition is enabled and when the associated transition condition is true.

The clearing of a transition causes the deactivation (or "resetting") of all the immediately preceding steps connected to the corresponding transition symbol by directed links, followed by the activation of all the immediately following steps.

The alternation step/transition and transition/step shall always be maintained in SFC element connections, that is:

- Two steps shall never be directly linked; they shall always be separated by a transition.
- Two transitions shall never be directly linked; they shall always be separated by a step.

When the clearing of a transition leads to the activation of several steps at the same time, the sequences to which these steps belong are called simultaneous sequences. After their simultaneous activation, the evolution of each of these sequences becomes independent. In order to emphasize the special nature of such constructs, the divergence and convergence of simultaneous sequences shall be indicated by a double horizontal line.

It shall be an error if the possibility can arise that non-prioritized transitions in a selection divergence, as shown in feature 2a of Table 61, are simultaneously true. The user may make provisions to avoid this error as shown in features 2b and 2c of Table 61.

Table 61 defines the syntax and semantics of the allowed combinations of steps and transitions.

The clearing time of a transition may theoretically be considered as short as one may wish, but it can never be zero. In practice, the clearing time will be imposed by the programmable controller implementation. For the same reason, the duration of a step activity can never be considered to be zero.

Several transitions which can be cleared simultaneously shall be cleared simultaneously, within the timing constraints of the particular programmable controller implementation and the priority constraints defined in Table 61.

Testing of the successor transition condition(s) of an active step shall not be performed until the effects of the step activation have propagated throughout the program organization unit in which the step is declared.

Figure 25 illustrates the application of these rules. In this figure, the active state of a step is indicated by the presence of an asterisk (*) in the corresponding block. This notation is used for illustration only, and is not a required language feature.

The application of the rules given in this subclause cannot prevent the formulation of “unsafe” SFCs, such as the one shown in Figure 26 a), which may exhibit uncontrolled proliferation of tokens. Likewise, the application of these rules cannot prevent the formulation of “unreachable” SFCs, such as the one shown in Figure 26 b), which may exhibit “locked up” behavior. The programmable controller system shall treat the existence of such conditions as errors.

The maximum allowed widths of the “divergence” and “convergence” constructs in Table 61 are Implementer specific.

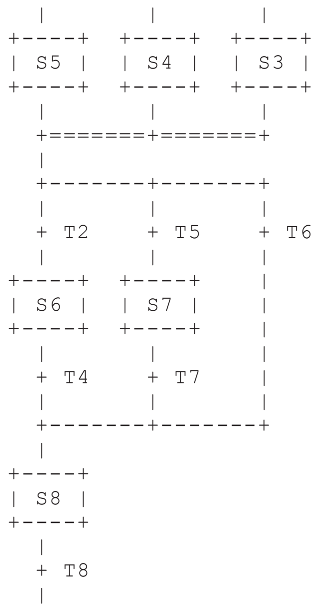
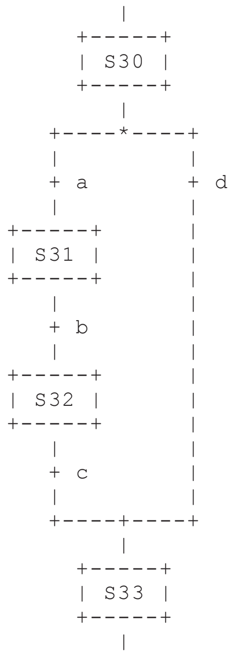
Table 61 – Sequence evolution – graphical

No.	Description	Explanation	Example
1	Single sequence	The alternation step-transition is repeated in series.	| +----+ | S3 | +----+ | + c | +----+ | S4 | +----+ | An evolution from step S3 to step S4 takes place if and only if step S3 is in the active state and the transition condition c is TRUE

2a	Divergence of sequence with left to right priority	A selection between several sequences is represented by as many transition symbols, under the horizontal line, as there are different possible evolutions. The asterisk denotes left-to-right priority of transition evaluations.	|
			     +----+
			     | S5 |
			     +----+
			        |
			   +----*----+-...
			   | |
			   + e + f
			   | |
			+----+ +----+
			| S6 | | S8 |
			+----+ +----+
			   | |
			An evolution takes place from S5 to S6 if S5 is active and the transition condition e is TRUE (independent of the value of f), or from S5 to S8 only if S5 is active and f is TRUE and e is FALSE

<table>
<tr><th>No.</th><th>Description</th><th>Explanation</th><th>Example</th></tr>
<tr><td>2b</td><td>Divergence of sequence with numbered branches</td><td>The asterisk (“ * ”), followed by numbered branches, indicates a user-defined priority of transition evaluation, with the lowest-numbered branch having the highest priority.</td><td><pre>
        |
     +----+
     | S5 |
     +----+
        |
  +-----*-----+-...
  |2 |1
  + e   + f
  | |
+----+   +----+
| S6 | | S8 |
+----+   +----+
  | |
</pre>An evolution takes place from S5 to S8 if S5 is active and the transition condition f is TRUE (independent of the value of e), or from S5 to S6 only if S5 is active and e is TRUE and f is FALSE.</td></tr>
<tr><td>2c</td><td>Divergence of sequence with mutual exclusion</td><td>The connection (“ + ”) of the branch indicates that the user shall assure that transition conditions are mutually exclusive.</td><td><pre>
        |
      +----+
      | S5 |
      +----+
        |
  +-----+-----+-...
  | |
  +e   +NOT e & f
  | |
+----+   +----+
| S6 | | S8 |
+----+   +----+
  | |
</pre>An evolution takes place from S5 to S6 if S5 is active and the transition condition e is TRUE, or from S5 to S8 only if S5 is active and e is FALSE and f is TRUE.</td></tr>
<tr><td>3</td><td>Convergence of sequence</td><td>The end of a sequence selection is represented by as many transition symbols, above the horizontal line, as there are selection paths to be ended.</td><td><pre>
  | |
+----+   +----+
| S7 | | S9 |
+----+   +----+
  | |
  + h   + j
  | |
  +-----+-----+-...
        |
     +------+
     | S10 |
     +------+
        |
</pre>An evolution takes place from S7 to S10 if S7 is active and the transition condition h is TRUE, or from S9 to S10 if S9 is active and j is TRUE.</td></tr>
</table>

<table>
<tr><th>No.</th><th>Description</th><th>Explanation</th><th>Example</th></tr>
<tr><td>4a</td><td>Simultaneous divergence after a single transition</td><td>The double horizontal line of synchronization can be preceded by a single transition condition.</td><td><pre>
         |
      +-----+
      | S11 |
      +-----+
         |
         + b
         |
   +=====+=====+=...
   | |
+-----+   +-----+
| S12 | | S14 |
+-----+   +-----+
   | |
</pre>An evolution takes place from S11 to S12, S14, ..., if S11 is active and the transition condition b associated to the common transition is TRUE.
After the simultaneous activation of S12, S14, etc., the evolution of each sequence proceeds independently.</td></tr>
<tr><td>4b</td><td>Simultaneous divergence after conversion</td><td>The double horizontal line of synchronization can be preceded by a sequence selection convergence.</td><td><pre>
   | |
+----+ +----+
| S2 | | S5 |
+----   +----+
   | |
   + T2   + T6
   | |
   +-------+
   |
   +=======+=======+
   | | |
+----+ +----+ +----+
| S3 | | S6 | | S7 |
+----+ +----+ +----+
</pre>An evolution takes place to the steps S3, S6 and S7 if S2 is active and the transition T2 is TRUE or S5 is active and the transition T6 is true.</td></tr>
<tr><td>4c</td><td>Simultaneous convergence
before one transition</td><td>Double lines of simultaneous convergence can be followed by a single transition.</td><td><pre>
   | |
+-----+   +-----+
| S13 | | S15 |
+-----+   +-----+
   | |
   +=====+=====+=...
         |
         + d
         |
      +-----+
      | S16 |
      +-----+
         |
</pre>An evolution takes place from S13, S15, ... to S16 only if all steps above and connected to the double horizontal line are active and the transition condition d associated to the common transition is TRUE.</td></tr>
</table>

No.	Description	Explanation	Example
4d	Simultaneous convergence before a sequence selection	Double lines of simultaneous convergence can be followed by a sequence selection divergence.	 S5 S4 S3 T2 T5 T6 S6 S7 T4 T7 S8 T8 An evolution takes place from S5, S4 and S3 to one of the steps S6, S7 or S8 only if all steps above and connected to the double horizontal line are active and the transition condition T2, T5 or T6 is TRUE, respectively.
5a,b, c	Sequence skip	A “sequence skip” is a special case of sequence selection (feature 2) in which one or more of the branches contain no steps. Features 5a, 5b, and 5c correspond to the representation options given in features 2a, 2b, and 2c, respectively.	 (feature 5a shown) An evolution takes place from S30 to S33 if “a” is `FALSE` and `d` is `TRUE`, that is, the sequence (S31, S32) will be skipped.

<table>
<tr><th>No.</th><th>Description</th><th>Explanation</th><th>Example</th></tr>
<tr><td>6a, b, c</td><td>Sequence loop</td><td>A “sequence loop” is a special case of sequence selection (feature 2) in which one or more of the branches return to a preceding step. Features 6a, 6b, and 6c correspond to the representation options given in features 2a, 2b, and 2c, respectively.</td><td>

```
    |
 +-----+
 | S30 |
 +-----+
    |
    + a
    |
    +---------+
    |         |
 +-----+      |
 | S31 |      |
 +-----+      |
    |         |
    + b       |
    |         |
 +-----+      |
 | S32 |      |
 +-----+      |
    |         |
    *-----+   |
    |     |   |
    + c   + d |
    |     |   |
 +-----+  +---+
 | S33 |
 +-----+
    |
```

(feature 6a shown)
An evolution takes place from S32 to S31 if “c” is false and “d” is TRUE, that is, the sequence (S31, S32) will be repeated.</td></tr>
<tr><td>7</td><td>Directional arrows</td><td>When necessary for clarity, the “less than” (<) character of the character set defined 6.1.1 can be used to indicate right-to-left control flow, and the “greater than” (>) character to represent left-to-right control flow.
When this feature is used, the corresponding character shall be located between two “-” characters, that is, in the character sequence “-<-” or “->-”as shown in the accompanying example.</td><td>

```
    |
 +-----+
 | S30 |
 +-----+
    |
    + a
    |
    +----<----+
    |         |
 +-----+      |
 | S31 |      |
 +-----+      |
    |         |
    + b       |
    |         |
 +-----+      |
 | S32 |      |
 +-----+      |
    |         |
    *-----+   |
    |     |   |
    + c   + d |
    |     |   |
 +-----+  +->-+
 | S33 |
 +-----+
    |
```

</td></tr>
</table>

```
   |                   |         |         |
+------+            +------+  +------+  +------+
|STEP10|            |STEP9 |  |STEP13|  |STEP22|
|      |            |      |  |  *   |  |  *   |
+------+            +------+  +------+  +------+
   |                   |         |         |
   + X             ====+=========+=========+====
   |                             |
+------+                         + X
|STEP11|                         |
|      |                ====+====+===+====
+------+                    |        |
   |                    +------+  +------+
                        |STEP15|  |STEP16|
                        |      |  |      |
                        +------+  +------+
                            |        |
```

a) Transition not enabled (NOTE 2)

```
    |                  |        |         |
+------+           +-----+  +------+  +------+
|STEP10|           |STEP9|  |STEP13|  |STEP22|
|  *   |           |  *  |  |  *   |  |  *   |
+------+           +-----+  +------+  +------+
   |                  |        |         |
   + X            ====+========+=========+=====
   |                           |
+------+                       + X
|STEP11|                       |
|      |              ====+====+====+====
+------+                  |         |
   |                   +------+  +------+
                       |STEP15|  |STEP16|
                       |      |  |      |
                       +------+  +------+
                          |         |
```

b) Transition enabled but not cleared (X = 0)

```
   |                    |        |         |
+------+            +-----+  +------+  +------+
|STEP10|            |STEP9|  |STEP13|  |STEP22|
|      |            |     |  |      |  |      |
+------+            +-----+  +------+  +------+
   |                   |        |          |
   + X             ====+========+==========+====
   |                            |
+------+                        + X
|STEP11|                        |
|  *   |               ====+====+===+====
+------+                   |        |
   |                   +------+  +------+
                       |STEP15|  |STEP16|
                       |   *  |  |  *   |
                       +------+  +------+
                           |        |
```

c) Transition cleared (X = 1)

NOTE 1 In this figure, the active state of a step is indicated by the presence of an asterisk (*) in the corresponding block. This notation is used for illustration only, and is not a required language feature.

NOTE 2 In a), the value of the Boolean variable X may be either TRUE or FALSE.

Figure 25 – SFC evolution (Rules)

```
+----------------------+
|                      |
|                   +=====+
|                   || A ||
|                   +=====+
|                      |
|                      + t1
|                      |
|     ======+==========+============+=======
|           |                       |
|        +-----+                 +-----+
|        |  B  |                 |  C  |
|        +-----+                 +-----+
|           |                       |
|           |                       *--------+
|           |                       |        |
|           |                       + t2     + t3
|           |                       |        |
|           |                     +---+    +---+
|           |                     | D |    | E |
|           |                     +---+    +---+
|           |                       |        |
|        ===+==========+============+===     |
|                      |                     |
|                      + t4                  + t5
|                      |                     |
|                    +---+                 +---+
|                    | F |                 | G |
|                    +---+                 +---+
|                      |                     |
|                      + t6                  + t7
|                      |                     |
+----------------------+---------------------+
```

a) SFC error: an “unsafe” SFC

```
+----------------------+
|                      |
|                   +=====+
|                   || A ||
|                   +=====+
|                      |
|                      + t1
|                      |
|     ======+==========+============+=======
|           |                       |
|        +-----+                 +-----+
|        |  B  |                 |  C  |
|        +-----+                 +-----+
|           |                       |
|           |                       *--------+
|           |                       |        |
|           |                       + t2     + t3
|           |                       |        |
|           |                     +---+    +---+
|           |                     | D |    | E |
|           |                     +---+    +---+
|           |                       |        |
|        ===+==========+============+===     |
|                      |                     |
|                      + t4                  + t5
|                      |                     |
|                    +---+                 +---+
|                    | F |                 | G |
|                    +---+                 +---+
|                      |                     |
|                  ====+==========+==========+===
|                                 |
|                                 + t6
|                                 |
+---------------------------------+
```

b) SFC error: an “unreachable” SFC

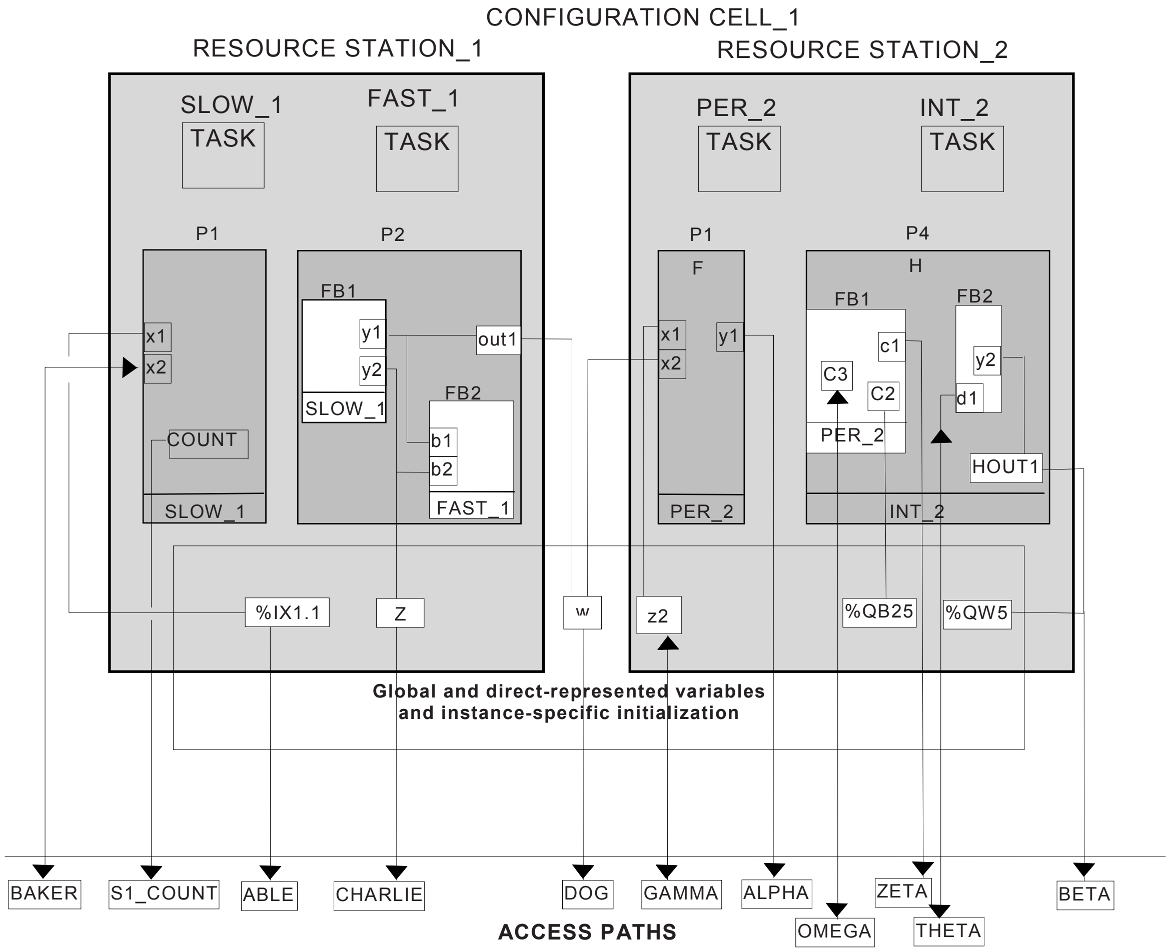
Figure 26 – SFC errors (Example)

6.8 Configuration elements

6.8.1 General

A configuration consists of resources, tasks (which are defined within resources), global variables, access paths and instance specific initializations. Each of these elements is defined in detail in this 6.8.

A graphic example of a simple configuration is shown in Figure 27 a). Skeleton declarations for the corresponding function blocks and programs are given in Figure 27 b). The declaration of the example in Figure 27 is shown in Figure 28.



a) Graphical representation

```
FUNCTION_BLOCK A
  VAR_OUTPUT
    y1: UINT;
    y2: BYTE;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK C
  VAR_OUTPUT
    c1: BOOL;
  END_VAR
  VAR
    C2 AT %Q*: BYTE;
    C3: INT;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK B
  VAR_INPUT
    b1: UINT;
    b2: BYTE;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK D
  VAR_INPUT
    d1: BOOL;
  END_VAR
  VAR_OUTPUT
    y2: INT;
  END_VAR
END_FUNCTION_BLOCK
```

```
PROGRAM F
  VAR_INPUT
    x1: BOOL;
    x2: UINT;
  END_VAR
  VAR_OUTPUT
    y1: BYTE;
  END_VAR
  VAR
    COUNT: INT;
    TIME1: TON;
  END_VAR
END_PROGRAM

PROGRAM G
  VAR_OUTPUT
    out1: UINT;
  END_VAR
  VAR_EXTERNAL
    z1: BYTE;
  END_VAR
  VAR
    FB1: A;
    FB2: B;
  END_VAR

  FB1(...);
  out1:= FB1.y1;
  z1:= FB1.y2;
  FB2(b1:= FB1.y1, b2:= FB1.y2);
END_PROGRAM

PROGRAM H
  VAR_OUTPUT
    HOUT1: INT;
  END_VAR
  VAR
    FB1: C;
    FB2: D;
  END_VAR

  FB1(...);
  FB2(...);
  HOUT1:= FB2.y2;
END_PROGRAM
```

b) Skeleton function block and program declarations

Figure 27 – Configuration (Example)

Table 62 enumerates the language features for declaration of configurations, resources, global variables, access paths and instance specific initializations.

- **Tasks**

  Figure 27 provides examples of the `TASK` features, corresponding to the example configuration shown in Figure 27 a) and the supporting declarations in Figure 27 b).

- **Resources**

  The `ON` qualifier in the `RESOURCE...ON...END_RESOURCE` construction is used to specify the type of “processing function” and its “man-machine interface” and “sensor and actuator interface” functions upon which the resource and its associated programs and tasks are to be implemented. The Implementer shall supply an Implementer specific resource library of such elements, as illustrated in Figure 3. Associated with each element in this library shall be an identifier (the resource type name) for use in resource declaration.

  NOTE 1 The `RESOURCE...ON...END_RESOURCE` construction is not required in a configuration with a single resource.

- **Global variables**

  The scope of a `VAR_GLOBAL` declaration shall be limited to the configuration or resource in which it is declared, with the exception that an access path can be declared to a global variable in a resource using feature 10d in Table 62.

- **Access paths**

  VAR_ACCESS...END_VAR construction provides a means of specifying variable names which can be used for remote access by some of the communication services specified in IEC 61131-5. An access path associates each such variable name with a global variable, a directly represented variable or any input, output, or internal variable of a program or function block.

  The association shall be accomplished by qualifying the name of the variable with the complete hierarchical concatenation of instance names, beginning with the name of the resource (if any), followed by the name of the program instance (if any), followed by the name(s) of the function block instance(s) (if any). The name of the variable is concatenated at the end of the chain. All names in the concatenation shall be separated by dots. If such a variable is a multi-element variable (structure or array) then an access path can also be specified to an element of the variable.

  It shall not be possible to define access paths to variables that are declared in VAR_TEMP, VAR_EXTERNAL or VAR_IN_OUT declarations.

  The direction of the access path can be specified as READ_WRITE or READ_ONLY, indicating that the communication services can both read and modify the value of the variable in the first case, or read but not modify the value in the second case. If no direction is specified, the default direction is READ_ONLY.

  Access to variables that are declared CONSTANT or to function block inputs that are externally connected to other variables shall be READ_ONLY.

  NOTE 2 The effect of using READ_WRITE access to function block output variables is Implementer specific.

- **Configurations**

  The VAR_CONFIG...END_VAR construction provides a means to assign instance specific locations to symbolically represented variables, which are nominated for the respective purpose by using the asterisk notation "*" or to assign instance specific initial values to symbolically represented variables, or both.

  The assignment shall be accomplished by qualifying the name of the object to be located or initialized with the complete hierarchical concatenation of instance names, beginning with the name of the resource (if any), followed by the name of the program instance, followed by the name(s) of the function block instance(s) (if any). The name of the variable to be located or initialized is concatenated at the end of the chain, followed by the name of the component of the structure (if the variable is structured). All names in the concatenation shall be separated by dots. The location assignment or the initial value assignment follows the syntax and the semantics.

  Instance specific initial values provided by the VAR_CONFIG...END_VAR construction always prevail type specific initial values. It shall not be possible to define instance specific initializations to variables which are declared in VAR_TEMP, VAR_EXTERNAL, VAR CONSTANT or VAR_IN_OUT declarations.

Table 62 – Configuration and resource declaration

No.	Description
1	CONFIGURATION...END_CONFIGURATION
2	VAR_GLOBAL...END_VAR within CONFIGURATION
3	RESOURCE...ON ...END_RESOURCE
4	VAR_GLOBAL...END_VAR within RESOURCE
5a	Periodic TASK
5b	Non-periodic TASK
6a	WITH for PROGRAM to TASK association
6b	WITH for FUNCTION_BLOCK to TASK association
6c	PROGRAM with no TASK association

No.	Description
7	Directly represented variables in VAR_GLOBAL
8a	Connection of directly represented variables to PROGRAM inputs
8b	Connection of GLOBAL variables to PROGRAM inputs
9a	Connection of PROGRAM outputs to directly represented variables
9b	Connection of PROGRAM outputs to GLOBAL variables
10a	VAR_ACCESS...END_VAR
10b	Access paths to directly represented variables
10c	Access paths to PROGRAM inputs
10d	Access paths to GLOBAL variables in RESOURCEs
10e	Access paths to GLOBAL variables in CONFIGURATIONs
10f	Access paths to PROGRAM outputs
10g	Access paths to PROGRAM internal variables
10h	Access paths to function block inputs
10i	Access paths to function block outputs
11a	VAR_CONFIG...END_VAR to variables This feature shall be supported if the feature “partly defined” with “*” in Table 16 is supported.
11b	VAR_CONFIG...END_VAR to components of structures
12a	VAR_GLOBAL CONSTANT in RESOURCE
12b	VAR_GLOBAL CONSTANT in CONFIGURATION
13a	VAR_EXTERNAL in RESOURCE
13b	VAR_EXTERNAL CONSTANT in RESOURCE

The following figure shows the declaration of the example in Figure 27.

```
Program code                                                using feature of
                                                               Table 62

CONFIGURATION CELL_1                                              1
  VAR_GLOBAL   w: UINT;   END_VAR                                 2
  RESOURCE STATION_1 ON PROCESSOR_TYPE_1                          3
    VAR_GLOBAL   z1: BYTE;   END_VAR                              4
    TASK SLOW_1(INTERVAL:= t#20ms, PRIORITY:= 2);                 5a
    TASK FAST_1(INTERVAL:= t#10ms, PRIORITY:= 1);                 5a
    PROGRAM P1 WITH SLOW_1:                                       6a
                     F(x1:= %IX1.1);                              8a
    PROGRAM P2: G(OUT1 => w,                                      9b
                     FB1 WITH SLOW_1,                             6b
                     FB2 WITH FAST_1);                            6b
  END_RESOURCE                                                    3
  RESOURCE STATION_2 ON PROCESSOR_TYPE_2                          3
    VAR_GLOBAL   z2      : BOOL;                                  4
                     AT %QW5: INT ;                               7
    END_VAR                                                       4
    TASK PER_2(INTERVAL:= t#50ms, PRIORITY:= 2);                  5a
    TASK INT_2(SINGLE:= z2,       PRIORITY:= 1);                  5b
    PROGRAM P1 WITH PER_2:                                                  6a
                     F(x1:= z2, x2:= w);                                    8b
    PROGRAM P4 WITH INT_2:                                                  6a
                     H(HOUT1 => %QW5,                                       9a
                     FB1 WITH  PER_2);                                      6b
  END_RESOURCE                                                              3
  VAR_ACCESS                                                                10a
    ABLE      : STATION_1.%IX1.1          : BOOL READ_ONLY;                 10b
    BAKER     : STATION_1.P1.x2           : UINT READ_WRITE;                10c
    CHARLIE   : STATION_1.z1              : BYTE;                           10d
    DOG       : w                         : UINT READ_ONLY;                 10e
    ALPHA     : STATION_2.P1.y1           : BYTE READ_ONLY;                 10f
    BETA      : STATION_2.P4.HOUT1        : INT READ_ONLY;                  10f
    GAMMA     : STATION_2.z2              : BOOL READ_WRITE;                10d
    S1_COUNT  : STATION_1.P1.COUNT        : INT;                            10g
    THETA     : STATION_2.P4.FB2.d1       : BOOL READ_WRITE;                10h
    ZETA      : STATION_2.P4.FB1.c1       : BOOL READ_ONLY;                 10i
    OMEGA     : STATION_2.P4.FB1.C3       : INT READ_WRITE;                 10k
  END_VAR                                                                   10a
  VAR_CONFIG                                                                11
    STATION_1.P1.COUNT: INT:= 1;
    STATION_2.P1.COUNT: INT:= 100;
    STATION_1.P1.TIME1: TON:= (PT:= T#2.5s);
    STATION_2.P1.TIME1: TON:= (PT:= T#4.5s);
    STATION_2.P4.FB1.C2 AT %QB25: BYTE;
  END_VAR
END_CONFIGURATION                                                           1
```

NOTE 1 Graphical and semigraphic representation of these features is allowed but is beyond the scope of this part of IEC 61131.

NOTE 2 It is an error if the data type declared for a variable in a `VAR_ACCESS` statement is not the same as the data type declared for the variable elsewhere, e.g., if variable `BAKER` is declared of type `WORD` in the above examples.

Figure 28 – `CONFIGURATION` and `RESOURCE` declaration (Example)

6.8.2 Tasks

For the purposes of this part of the IEC 61131 series, a task is defined as an execution control element which is capable of calling, 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, which can include programs and function blocks whose instances are specified in the declaration of programs.

The maximum number of tasks per resource and task interval resolution is Implementer specific.

Tasks and their association with program organization units can be represented graphically or textually using the `WITH` construction, as shown in Table 63, as part of resources within configurations. A task is implicitly enabled or disabled by its associated resource according to the mechanisms. The control of program organization units by enabled tasks shall conform to the following rules:

a) The associated program organization units shall be scheduled for execution upon each rising edge of the `SINGLE` input of the task.

b) If the `INTERVAL` input is non-zero, the associated program organization units shall be scheduled for execution periodically at the specified interval as long as the `SINGLE` input stands at zero (0). If the `INTERVAL` input is zero (the default value), no periodic scheduling of the associated program organization units shall occur.

c) The `PRIORITY` input of a task establishes the scheduling priority of the associated program organization units, with zero (0) being highest priority and successively lower priorities having successively higher numeric values. As shown in Table 63, the priority of a program organization unit (that is, the priority of its associated task) can be used for pre-emptive or non-pre-emptive scheduling.

   - In non-pre-emptive scheduling, processing power becomes available on a resource when execution of a program organization unit or operating system function is complete. When processing power is available, the program organization unit with highest scheduled priority shall begin execution. If more than one program organization unit is waiting at the highest scheduled priority, then the program organization unit with the longest waiting time at the highest scheduled priority shall be executed.

   - In pre-emptive scheduling, when a program organization unit is scheduled, it can interrupt the execution of a program organization unit of lower priority on the same resource, that is, the execution of the lower-priority unit can be suspended until the execution of the higher-priority unit is completed. A program organization unit shall not interrupt the execution of another unit of the same or higher priority.

Depending on schedule priorities, a program organization unit might not begin execution at the instant it is scheduled. However, in the examples shown in Table 63, all program organization units meet their deadlines, that is, they all complete execution before being scheduled for re-execution. The Implementer shall provide information to enable the user to determine whether all deadlines will be met in a proposed configuration.

d) A program with no task association shall have the lowest system priority. Any such program shall be scheduled for execution upon “starting” of its resource and shall be re-scheduled for execution as soon as its execution terminates.

e) When a function block instance is associated with a task, its execution shall be under the exclusive control of the task, independent of the rules of evaluation of the program organization unit in which the task-associated function block instance is declared.

f) Execution of a function block instance which is not directly associated with a task shall follow the normal rules for the order of evaluation of language elements for the program organization unit (which can itself be under the control of a task) in which the function block instance is declared.

   NOTE 1 Classes instances cannot be associated with a task.

   NOTE 2 The methods of a function block or of a class are executed in the POU they are called.

g) The execution of function blocks within a program shall be synchronized to ensure that data concurrency is achieved according to the following rules:

   - If a function block receives more than one input from another function block, then when the former is executed, all inputs from the latter shall represent the results of the same evaluation.

   - If two or more function blocks receive inputs from the same function block, and if the “destination” blocks are all explicitly or implicitly associated with the same task, then the inputs to all such “destination” blocks at the time of their evaluation shall represent the results of the same evaluation of the “source” block.

Provision shall be made for storage of the outputs of functions or function blocks which have explicit task associations, or which are used as inputs to program organization units which have explicit task associations, as necessary to satisfy the rules given above.

It shall be an error if a task fails to be scheduled or to meet its execution deadline because of excessive resource requirements or other task scheduling conflicts.

Table 63 – Task

<table>
<tr><th>No.</th><th>Description</th><th>Examples</th></tr>
<tr><td>1a</td><td>Textual declaration of periodic TASK</td><td>(feature 5a of Table 62)</td></tr>
<tr><td>1b</td><td>Textual declaration of non-periodic TASK</td><td>(feature 5b of Table 62)</td></tr>
<tr><td></td><td>Graphical representation of TASKs (general form)</td><td><pre>
        TASKNAME
       +---------+
       | TASK |
BOOL---|SINGLE |
TIME---|INTERVAL |
UINT---|PRIORITY |
       +---------+
</pre></td></tr>
<tr><td>2a</td><td>Graphical representation of periodic TASKs (with INTERVAL)</td><td><pre>
         SLOW_1   FAST_1
        +---------+   +---------+
        | TASK | | TASK |
      --|SINGLE | ---|SINGLE |
t#20ms--|INTERVAL | t#10ms---|INTERVAL |
     2--|PRIORITY | 1---|PRIORITY |
        +---------+   +---------+
</pre></td></tr>
<tr><td>2b</td><td>Graphical representation of non-periodic TASK (with SINGLE)</td><td><pre>
         INT_2
        +---------+
        | TASK |
    z2--|SINGLE |
      --|INTERVAL |
     1--|PRIORITY |
        +---------+
</pre></td></tr>
<tr><td>3a</td><td>Textual association with PROGRAMs</td><td>(feature 6a of Table 62)</td></tr>
<tr><td>3b</td><td>Textual association with function blocks</td><td>(feature 6b of Table 62)</td></tr>
<tr><td>4a</td><td>Graphical association with PROGRAMs</td><td><pre>
RESOURCE STATION_2

      P1   P4
  +-------+   +-------+
F		H
  +-------+   +-------+
  | PER_2 | | INT_2 |
  +-------+   +-------+

END_RESOURCE
</pre></td></tr>
<tr><td>4b</td><td>Graphical association with function blocks within PROGRAMs</td><td><pre>
 P2
   +---+
   | G |
   | |
   | FB1   FB2 |
   | +------+   +------+ |
	A		B	
+------+   +------+
	SLOW_1		FAST_1	
+------+   +------+
   +---+

END_RESOURCE
</pre></td></tr>
<tr><td>5a</td><td>Non-preemptive scheduling</td><td>See Figure 28</td></tr>
<tr><td>5b</td><td>Preemptive scheduling</td><td>See Figure 28</td></tr>
<tr><td colspan="3">NOTE 1 Details of RESOURCE and PROGRAM declarations are not shown.
NOTE 2 The notation X@Y indicates that program organization unit X is scheduled or executing at priority Y.</td></tr>
</table>

The following examples show non-preemptive and preemptive scheduling defined in Table 63, 5a and 5b.

EXAMPLES 1 Non-preemptive and preemptive scheduling

1. Non-preemptive scheduling

- RESOURCE STATION_1 as configured in Figure 28
- Execution times: P1 = 2 ms; P2 = 8 ms
- P2.FB1 = P2.FB2 = 2 ms (see NOTE 1)
- STATION_1 starts at t = 0

Schedule (repeats every 40 ms)

t(ms)	Executing	Waiting
0	P2.FB2@1	P1@2, P2.FB1@2, P2
2	P1@2	P2.FB1@2, P2
4	P2.FB1@2	P2
6	P2	
10	P2	P2.FB2@1
14	P2.FB2@1	P2
16	P2	(P2 restarts)
20	P2	P2.FB2@1, P1@2, P2.FB1@2
24	P2.FB2@1	P1@2, P2.FB1@2, P2
26	P1@2	P2.FB1@2, P2
28	P2.FB1@2	P2
30	P2.FB2@1	P2
32	P2	
40	P2.FB2@1	P1@2, P2.FB1@2, P2

- RESOURCE STATION_2 as configured in Figure 28
- Execution times: P1 = 30 ms, P4 = 5 ms, P4.FB1 = 10 ms
- INT_2 is triggered at t = 25, 50, 90,... ms
- STATION_2 starts at t = 0

Schedule

t(ms)	Executing	Waiting
0	P1@2	P4.FB1@2
25	P1@2	P4.FB1@2, P4@1
30	P4@1	P4.FB1@2
35	P4.FB1@2	
50	P4@1	P1@2, P4.FB1@2
55	P1@2	P4.FB1@2
85	P4.FB1@2	
90	P4.FB1@2	P4@1
95	P4@1	
100	P1@2	P4.FB1@2

2. Preemptive scheduling See Table 63, 5b

- RESOURCE STATION_1 as configured in Figure 28
- Execution times: P1 = 2 ms; P2 = 8 ms; P2.FB1 = P2.FB2 = 2 ms
- STATION_1 starts at t = 0

Schedule

t(ms)	Executing	Waiting
0	P2.FB2@1	P1@2, P2.FB1@2, P2
2	P1@2	P2.FB1@2, P2
4	P2.FB1@2	P2

6	P2	
10	P2.FB2@1	P2
12	P2	
16	P2	(P2 restarts)
20	P2.FB2@1	P1@2, P2.FB1@2, P2
- RESOURCE STATION_2 as configured in Figure 28 - Execution times: P1 = 30 ms, P4 = 5 ms, P4.FB1 = 10 ms (NOTE 2) - INT_2 is triggered at t = 25, 50, 90,... ms - STATION_2 starts at t = 0		
Schedule		
t(ms)	**Executing**	**Waiting**
0	P1@2	P4.FB1@2
25	P4@1	P1@2, P4.FB1@2
30	P1@2	P4.FB1@2
35	P4.FB1@2	
50	P4@1	P1@2, P4.FB1@2
55	P1@2	P4.FB1@2
85	P4.FB1@2	
90	P4@1	P4.FB1@2
95	P4.FB1@2	
100	P1@2	P4.FB1@2
NOTE 1 The execution times of P2.FB1 and P2.FB2 are not included in the execution time of P2. NOTE 2 The execution time of P4.FB1 is not included in the execution time of P4.		

EXAMPLES 2 Task associations to function block instances

```
RESOURCE R1

PROGRAM X
      Y1                  Y2
    +-----+             +-----+
    |  Y  |             |  Y  |
 ---|A   C|----+--------|A   C|---
 ---|B   D|----|--+-----|B   D|---
    +-----+    |  |     +-----+
    |slow1|    |  |     |fast1|
    +-----+    |  |     +-----+
               |  |
               |  |    Y3
               |  |  +-----+
               |  |  |  Y  |
               +--|--|A   C|---
                  +--|B   D|---
                     +-----+
                     |fast1|
                     +-----+
END_PROGRAM
```

a) Function blocks with explicit task associations

```
             fast1                                    slow1
         +-----------+                            +-----------+
         |   TASK    |                            |   TASK    |
t#10ms---|INTERVAL   |                   t#20ms---|INTERVAL   |
     1---|PRIORITY   |                        2---|PRIORITY   |
         +-----------+                            +-----------+

P1

PROGRAM X
      Y1                  Y2
    +-----+             +-----+
    |  Y  |             |  Y  |
 ---|A   C|----+--------|A   C|---
 ---|B   D|----|--+-----|B   D|---
    +-----+    |  |     +-----+
    |fast1|    |  |
    +-----+    |  |
               |  |
               |  |    Y3
               |  |  +-----+
               |  |  |  Y  |
               +--|--|A   C|---
                  +--|B   D|---
                     +-----+
END_PROGRAM

slow1
```

b) Function blocks with implicit task associations

```
RESOURCE R1

             fast1                                 slow1
         +-----------+                         +-----------+
         |   TASK    |                         |   TASK    |
t#10ms---|INTERVAL   |                t#20ms---|INTERVAL   |
     1---|PRIORITY   |                     2---|PRIORITY   |
         +-----------+                         +-----------+
P1
PROGRAM X
      Y1                  Y2
    +-----+             +-----+
    |  Y  |             |  Y  |
 ---|A   C|----+--------|A   C|---
 ---|B   D|----|--+-----|B   D|---
    +-----+    |  |     +-----+
    |fast1|    |  |     |slow1|
    +-----+    |  |     +-----+
               |  |
               |  |     Y3
               |  |  +-----+
               |  |  |  Y  |
               +--|--|A   C|---
                  +--|B   D|---
                     +-----+
                     |slow1|
                     +-----+
END_PROGRAM
```

c) Explicit task associations equivalent to b)

NOTE 3 The graphical representations in these examples are illustrative only and are not normative.

6.9 Namespaces

6.9.1 General

For the purposes of programmable controller programming languages, a namespace is a language element combining other language elements to a combined entity.

The same name of a language element declared within a namespace may also be used within other namespaces.

Namespaces and types that have no enclosing namespace are members of the global namespace. The global namespace includes the names declared in the global scope. All standard functions and function blocks are elements of the global namespace.

Namespaces may be nested.

Namespaces and types declared within a namespace are members of that namespace. The members of the namespace are in the local scope of the namespace.

With namespaces a library concept can be implemented as well as a module concept. Namespaces can be used to avoid identifier ambiguities. A typical application of namespace is in the context of the object oriented programming features.

6.9.2 Declaration

A namespace declaration starts with the keyword `NAMESPACE` optionally followed by the access specifier `INTERNAL`, the name of the namespace and ends with the keyword `END_NAMESPACE`. A namespace contains a set of language elements, each optionally followed by the following access specifier:

- `INTERNAL` for an access only within the namespace itself.

The access specifier can be applied to the declaration of the following language elements:

- user-defined data types - using keyword `TYPE`,
- functions,
- programs,
- function block types and their variables and methods,
- classes and their variables and methods,

- interfaces,
- namespaces.

If no access specifier is given, the language elements of the namespace are accessible from outside the namespace, i.e. a namespace is public by default.

Examples 1 and 2 show the namespace declaration and the nested namespace declaration.

EXAMPLE 1 Namespace declaration

```
NAMESPACE Timers

         FUNCTION INTERNAL TimeTick: DWORD
          // ...declaration and operations here
         END_FUNCTION

       // other namespace elements without specifier are PUBLIC by Default
         TYPE
           LOCAL_TIME: STRUCT
             TIMEZONE: STRING [40];
             DST:      BOOL;  // Daylight saving time
             TOD:      TOD;
             END_STRUCT;
         END_TYPE;
         ...
         FUNCTION_BLOCK TON
          // ... declaration and operations here
         END_FUNCTION_BLOCK
         ...
         FUNCTION_BLOCK TOF
          // ... declaration and operations here
         END_FUNCTION_BLOCK

END_NAMESPACE (*Timers*)
```

EXAMPLE 2 Nested namespace declaration

```
NAMESPACE Standard   // Namespace = PUBLIC by Default

  NAMESPACE Timers   // Namespace = PUBLIC by Default

         FUNCTION INTERNAL TimeTick: DWORD
          // ...declaration and operations here
         END_FUNCTION

       // other namespace elements without specifier are PUBLIC by Default
         TYPE
           LOCAL_TIME: STRUCT
             TIMEZONE: STRING [40];
             DST:      BOOL;  // Daylight saving time
             TOD:      TOD;
             END_STRUCT;
         END_TYPE;
         ...
         FUNCTION_BLOCK TON  // defines an implementation of TON with a new name
          // ... declaration and operations here
         END_FUNCTION_BLOCK
         ...
         FUNCTION_BLOCK TOF  // defines an implementation of TOF with a new name
          // ... declaration and operations here
         END_FUNCTION_BLOCK

         CLASS A
           METHOD INTERNAL M1
           ...
           END_METHOD
           METHOD PUBLIC M2 // PUBLIC is given here to replace the default of PROTECTED
           ...
           END_METHOD
         END_CLASS

         CLASS INTERNAL B
           METHOD INTERNAL M1
           ...
           END_METHOD
           METHOD PUBLIC M2
           ...
           END_METHOD
         END_CLASS

  END_NAMESPACE (*Timers*)
  NAMESPACE Counters
         FUNCTION_BLOCK CUP
          // ... declaration and operations here
         END_FUNCTION_BLOCK
         ...
         FUNCTION_BLOCK CDOWN
          // ... declaration and operations here
         END_FUNCTION_BLOCK
  END_NAMESPACE (*Counters*)
END_NAMESPACE (*Standard*)
```

The accessibility on namespace elements, methods and variables of function blocks from inside and outside the namespace depends on the access specifiers of the variable or method together with the namespace specifier at the namespace declaration and the language elements.

The rules of accessibility are summarized in Figure 29.

Namespace specifier	**Public (default, no specifier)**		**INTERNAL**		
Access specifier of language element, variable or method	Access from outside the namespace	Access from inside the namespace but outside the POU	Access from outside the namespace		Access from inside the namespace but outside the POU
			All Namespaces except parent namespace	Parent namespace	
PRIVATE	No	No	No	No	No
PROTECTED	No	No	No	No	No
INTERNAL	No	Yes	No	No	Yes
PUBLIC	Yes	Yes	No	Yes	Yes

Figure 29 – Accessibility using namespaces (Rules)

In the case of hierarchical namespaces, the outside namespace can additionally restrict the access; it cannot allow additional access to entities which are already internal of the inner namespace.

EXAMPLE 3 Nested namespaces and access specifiers

```
NAMESPACE pN1
    NAMESPACE pN11
       FUNCTION pF1  ... END_FUNCTION            // accessible from everywhere
       FUNCTION INTERNAL iF2  ... END_FUNCTION   // accessible in pN11
       FUNCTION_BLOCK pFB1                       // accessible from everywhere
           VAR PUBLIC pVar1: REAL: ... END_VAR   // accessible from everywhere
           VAR INTERNAL iVar2: REAL ... END_VAR  // accessible in pN11
           ...
       END_FUNCTION_BLOCK
       FUNCTION_BLOCK INTERNAL iFB2              // accessible in pN11
           VAR PUBLIC pVar3: REAL: ... END_VAR   // accessible in pN11
           VAR INTERNAL iVar4: REAL ... END_VAR  // accessible in pN11
           ...
       END_FUNCTION_BLOCK
       CLASS pC1
           VAR PUBLIC pVar5: REAL: ... END_VAR   // accessible from everywhere
           VAR INTERNAL iVar6: REAL ... END_VAR  // accessible in pN11
           METHOD pM1   ... END_METHOD           // accessible from everywhere
           METHOD INTERNAL iM2   ... END_METHOD  // accessible in pN11
       END_CLASS
       CLASS INTERNAL iC2
           VAR PUBLIC pVar7: REAL: ... END_VAR   // accessible in pN11
           VAR INTERNAL iVar8: REAL ... END_VAR  // accessible in pN11
           METHOD pM3   ... END_METHOD           // accessible in pN11
           METHOD INTERNAL iM4   ... END_METHOD  // accessible in pN11
       END_CLASS
    END_NAMESPACE
    NAMESPACE INTERNAL iN12
       FUNCTION pF1  ... END_FUNCTION            // accessible in pN1
       FUNCTION INTERNAL iF2  ... END_FUNCTION   // accessible in iN12
       FUNCTION_BLOCK pFB1                       // accessible in pN1
           VAR PUBLIC pVar1: REAL: ... END_VAR   // accessible in pN1
           VAR INTERNAL iVar2: REAL ... END_VAR  // accessible in iN12
           ...
       END_FUNCTION_BLOCK
       FUNCTION_BLOCK INTERNAL iFB2              // accessible in iN12
           VAR PUBLIC pVar3: REAL: ... END_VAR   // accessible in iN12
           VAR INTERNAL iVar4: REAL ... END_VAR  // accessible in iN12
           ...
       END_FUNCTION_BLOCK
       CLASS pC1
           VAR PUBLIC pVar5: REAL: ... END_VAR   // accessible in pN1
           VAR INTERNAL iVar6: REAL ... END_VAR  // accessible in iN12
           METHOD pM1   ... END_METHOD           // accessible in pN1
           METHOD INTERNAL iM2   ... END_METHOD  // accessible in iN12
       END_CLASS
       CLASS INTERNAL iC2
           VAR PUBLIC pVar7: REAL: ... END_VAR   // accessible in iN12
           VAR INTERNAL iVar8: REAL ... END_VAR  // accessible in iN12
           METHOD pM3   ... END_METHOD           // accessible in iN12
           METHOD INTERNAL iM4   ... END_METHOD  // accessible in iN12
       END_CLASS
    END_NAMESPACE
END_NAMESPACE
```

Table 64 shows the features defined for namespace.

Table 64 – Namespace

No	Description	Example

1a	Public namespace (without access specifier)	`NAMESPACE name`
		`   declaration(s)`
		`   declaration(s)`
		`END_NAMESPACE`
		
		All containing elements are accessible according to their access specifiers.

1b	Internal namespace (with `INTERNAL` specifier)	`NAMESPACE INTERNAL name`
		`   declaration(s)`
		`   declaration(s)`
		`END_NAMESPACE`
		
		All containing elements without any specifier or the access specifier `PUBLIC` are accessible in the namespace one level above.

2	Nested namespaces	See Example 2

3	Variable access specifier `INTERNAL`	`CLASS C1`
		`  VAR INTERNAL myInternalVar: INT; END_VAR`
		`  VAR PUBLIC   myPublicVar: INT; END_VAR`
		`END_CLASS`

4	Method access specifier `INTERNAL`	`CLASS C2`
		`  METHOD INTERNAL myInternalMethod: INT; ... END_METHOD`
		`  METHOD PUBLIC   myPublicMethod: INT; ... END_METHOD`
		`END_CLASS`

5	Language element with access specifier `INTERNAL`:
	User-defined data types
	- using keyword `TYPE`
	Functions
	Function block types
	Classes
	Interfaces
		`CLASS INTERNAL`
		`  METHOD INTERNAL myInternalMethod: INT; ... END_METHOD`
		`  METHOD PUBLIC   myPublicMethod: INT; ... END_METHOD`
		`END_CLASS`
		
		`CLASS`
		`  METHOD INTERNAL myInternalMethod: INT; ... END_METHOD`
		`  METHOD PUBLIC   myPublicMethod: INT; ... END_METHOD`
		`END_CLASS`

The name of a namespace may be a single identifier or a fully qualified name consisting of a sequence of namespace identifiers separated by dots ("."). The latter form permits the declaration of a nested namespace without lexically nesting several namespace declarations. It also supports the extension of an existing namespace with further language elements by a further declaration.

Lexically nested namespaces are declared by multiple namespace declarations with the keyword `NAMESPACE` textually nested as shown in the first of the three features in Table 65. All three features contribute language elements to the same namespace `Standard.Timers.HighResolution`. The second feature shows the extension of the same namespace declared by a fully qualified name. The third feature mixes the namespace declaration by fully qualified name and by lexically nested `NAMESPACE` keywords to add another POU to the namespace.

Table 65 shows the features defined for nested namespace declaration options.

Table 65 – Nested namespace declaration options

No	Description	Example

1	Lexically nested namespace declaration
	Equivalent to feature 2 of Table 64
		`NAMESPACE Standard`
		`    NAMESPACE Timers`
		`        NAMESPACE HighResolution`
		`            FUNCTION PUBLIC TimeTick: DWORD`
		`             // ...declaration and operations here`
		`            END_FUNCTION`
		`        END_NAMESPACE (*HighResolution*)`
		`    END_NAMESPACE (*Timers*)`
		`END_NAMESPACE (*Standard*)`

2	Nested namespace declaration by fully qualified name	`NAMESPACE Standard.Timers.HighResolution`
		`          FUNCTION PUBLIC TimeResolution: DWORD`
		`           // ...declaration and operations here`
		`          END_FUNCTION`
		`END_NAMESPACE (*Standard.Timers.HighResolution*)`

3	Mixed lexically nested namespace and namespace nested by fully qualified name	`NAMESPACE Standard.Timers`
		`    NAMESPACE HighResolution`
		`          FUNCTION PUBLIC TimeLimit: DWORD`
		`           // ...declaration and operations here`
		`          END_FUNCTION`
		`    END_NAMESPACE (*HighResolution*)`
		`END_NAMESPACE (*Standard.Timers*)`

NOTE Multiple namespace declarations with the same fully qualified name contribute to the same namespace. In the examples of this Table the functions `TimeTick`, `TimeResolution`, and `TimeLimit` are members of the same namespace `Standard.Timers.HighResolution` even though they are defined in separate namespace declarations; e.g. in different Structured Text program files.		

6.9.3 Usage

Elements of a namespace can be accessed from outside the namespace by preceding the name of the namespace and a following “**.**”. This is not necessary from within the namespace but permitted.

Language elements declared with an `INTERNAL` access specifier cannot be accessed from outside the namespace except the own namespace.

Elements in nested namespaces can be accessed by naming all parent namespaces as shown in the example.

EXAMPLE

Usage of a Timer `TON` from the namespace Standard.Timers.

```
FUNCTION_BLOCK Uses_Timer

VAR

   Ton1: Standard.Timers.TON;
       (* starts timer with rising edge, resets timer with falling edge *)
   Ton2: PUBLIC.TON; (* uses the standard timer *)

bTest: BOOL;

END_VAR

   Ton1(In:= bTest, PT:= t#5s);

END_FUNCTION_BLOCK
```

6.9.4 Namespace directive `USING`

A `USING` namespace directive may be given following the name of a namespace, a POU, the name and result declaration of a function or a method.

If the USING directive is used within a function block, class or structure it shall immediately follow the type name.

If the USING directive is used within a function or a method it shall immediately follow the result type declaration of the function or method.

A USING directive starts with the keyword USING followed by one or a list of fully qualified names of namespaces as shown in Table 64, feature 2. It enables the use of the language elements contained in the specified namespaces immediately in the enclosing namespace resp. POU. The enclosing namespace might be the global namespace, too.

Within member declarations in a namespace that contains a USING namespace directive, the types contained in the given namespace can be referenced directly. In the example shown below, within member declarations of the namespace Infeed, the type members of Standard.Timers are directly available, and thus function block Uses_Timer can declare an instance variable of function block TON without qualification.

Examples 1 and 2 below show the usage of the namespace directive USING.

EXAMPLE 1 Namespace directive USING

```
NAMESPACE Counters
          FUNCTION_BLOCK CUP
           // ... declaration and operations here
          END_FUNCTION_BLOCK
END_NAMESPACE (*Standard.Counters*)

NAMESPACE Standard.Timers
          FUNCTION_BLOCK TON
           // ... declaration and operations here
          END_FUNCTION_BLOCK
END_NAMESPACE (*Standard.Timers*)

NAMESPACE Infeed
FUNCTION_BLOCK Uses_Std
        USING Standard.Timers;
 VAR
   Ton1: TON;
  (* starts timer with rising edge, resets timer with falling edge *)
   Cnt1: Counters.CUP;
   bTest: BOOL;
 END_VAR
   Ton1(In:= bTest, PT:= t#5s);
END_FUNCTION_BLOCK
END_NAMESPACE
```

A USING namespace directive enables the types contained in the given namespace, but specifically does not enable types contained in nested namespaces. The using namespace directive enables the types contained in Standard, but not types of the namespaces nested in Standard. Thus, the reference to Timers.TON in the declaration of Uses_Timer results in a compile-time error because no members named Standard are in scope.

EXAMPLE 2 Invalid import of nested namespaces

```
NAMESPACE Standard.Timers
          FUNCTION_BLOCK TON
           // ... declaration and operations here
          END_FUNCTION_BLOCK
END_NAMESPACE (*Standard.Timers*)

NAMESPACE Infeed
        USING Standard;
        USING Standard.Counters;

        FUNCTION_BLOCK Uses_Timer
        VAR
           ~~Ton1: Timers.TON~~; // ERROR: Nested namespaces are not imported
          (* starts timer with rising edge, resets timer with falling edge *)
           bTest: BOOL;
        END_VAR
           Ton1(In:= bTest, PT:= t#5s);
        END_FUNCTION_BLOCK
END_NAMESPACE (*Standard.Timers.HighResolution*)
```

For usage of language elements of a namespace in the global namespace the keyword USING and the namespace identifiers shall be used.

Table 66 shows the features defined for the namespace directive USING.

Table 66 – Namespace directive USING

No	Description	Example

1	USING in global namespace	`USING Standard.Timers;`
		` FUNCTION PUBLIC TimeTick: DWORD`
		`  VAR`
		`   Ton1: TON;`
		`  END_VAR // ...declaration and operations here`
		` END_FUNCTION`

2	USING in other namespace	`NAMESPACE Standard.Timers.HighResolution`
		`    USING Counters;`
		`          FUNCTION PUBLIC TimeResolution: DWORD`
		`           // ...declaration and operations here`
		`          END_FUNCTION`
		`END_NAMESPACE (*Standard.Timers.HighResolution*)`

3	USING in POUs
	• Functions
	• Function block types
	• Classes
	• Methods
	• Interfaces
		`FUNCTION_BLOCK Uses_Std`
		`    USING Standard.Timers, Counters;`
		` VAR`
		`   Ton1: TON;`
		`  (* starts timer with rising edge, resets timer with`
		`falling edge *)`
		`   Cnt1: CUP;`
		`   bTest: BOOL;`
		` END_VAR`
		`   Ton1(In:= bTest, PT:= t#5s);`
		`END_FUNCTION_BLOCK`
		
		`FUNCTION myFun: INT`
		`    USING Lib1, Lib2;`
		`    USING Lib3;`
		` VAR ....`
		`....`
		`END_FUNCTION`

7 Textual languages

7.1 Common elements

The textual languages defined in this standard are IL (Instruction List) and ST (Structured Text). The sequential function chart (SFC) can be used in conjunction with either of these languages.

Subclause 7.2 defines the semantics of the IL language, whose syntax is given in Annex A. Subclause 7.3 defines the semantics of the ST language, whose syntax is given.

The textual elements specified in Clause 6 shall be common to the textual languages (IL and ST) defined in this Clause 7. In particular, the following program structuring elements in Figure 30 shall be common to textual languages:

```
TYPE         ...END_TYPE
VAR          ...END_VAR
VAR_INPUT    ...END_VAR
VAR_OUTPUT   ...END_VAR
VAR_IN_OUT   ...END_VAR
VAR_EXTERNAL ...END_VAR
VAR_TEMP     ...END_VAR
VAR_ACCESS   ...END_VAR
VAR_GLOBAL   ...END_VAR
VAR_CONFIG   ...END_VAR
FUNCTION     ...END_FUNCTION
FUNCTION_BLOCK...END_FUNCTION_BLOCK
PROGRAM      ...END_PROGRAM
METHOD       ...END_METHOD
STEP         ...END_STEP
TRANSITION   ...END_TRANSITION
ACTION       ...END_ACTION
NAMESPACE    ...END_NAMESPACE
```

Figure 30 – Common textual elements (Summary)

7.2 Instruction list (IL)

7.2.1 General

This language is outdated as an assembler like language. Therefore it is deprecated and will not be contained in the next edition of this standard.

7.2.2 Instructions

An instruction list is composed of a sequence of instructions. Each instruction shall begin on a new line and shall contain an operator with optional modifiers, and, if necessary for the particular operation, one or more operands separated by commas. Operands can be of any of the data representations for literals, for enumerated values, and for variables.

The instruction can be preceded by an identifying label followed by a colon (:). Empty lines can be inserted between instructions.

EXAMPLE The fields of an instruction list

LABEL	OPERATOR	OPERAND	COMMENT
`START:`	`LD`	`%IX1`	`(* PUSH BUTTON *)`
	`ANDN`	`%MX5`	`(* NOT INHIBITED *)`
	`ST`	`%QX2`	`(* FAN ON *)`

7.2.3 Operators, modifiers and operands

7.2.3.1 General

Standard operators with their allowed modifiers and operands shall be as listed in Table 68.

7.2.3.2 "Current result"

Unless otherwise defined in Table 68 the semantics of the operators shall be

```
result:= result OP operand
```

That is, the value of the expression being evaluated is replaced by its current value operated upon by the operator with respect to the operand.

EXAMPLE 1 The instruction `AND %IX1` is interpreted as `result:= result AND %IX1`.

The comparison operators shall be interpreted with the current result to the left of the comparison and the operand to the right, with a Boolean result.

EXAMPLE 2 The instruction `GT %IW10` will have the Boolean result `1` if the current result is greater than the value of Input Word 10, and the Boolean result `0` otherwise.

7.2.3.3 Modifier

The modifier "`N`" indicates bitwise Boolean negation (one's complement) of the operand.

EXAMPLE 1 The instruction `ANDN %IX2` is interpreted as `result:= result AND NOT %IX2`.

It shall be an error if the current result and operand are not of same data type, or if the result of a numerical operation exceeds the range of values for its data type.

The left parenthesis modifier "`(`" indicates that evaluation of the operator shall be deferred until a right parenthesis operator "`)`" is encountered. In Table 67, two equivalent forms of a parenthesized sequence of instructions are shown. Both features in Table 67 shall be interpreted as

```
result:= result AND (%IX1 OR %IX2)
```

An operand shall be a literal as defined in 6.3, an enumerated value or a variable.

The function `REF()` and the dereferencing operator "`^`" shall be used in the definition of the operands, Table 67 shows the parenthesized expression.

Table 67 – Parenthesized expression for IL language

No.	Descriptionˆ	Example
1	Parenthesized expression beginning with explicit operator:	AND(LD %IX1 (NOTE) OR %IX2)
2	Parenthesized expression (short form)	AND(%IX1 OR %IX2)

NOTE In feature 1 the LD operator may be modified or the LD operation may be replaced by another operation or function call respectively.

The modifier “C” indicates that the associated instruction shall be performed only if the value of the currently evaluated result is Boolean 1 (or Boolean 0 if the operator is combined with the “N” modifier). Table 68 shows the Instruction list operators.

Table 68 – Instruction list operators

No.	**Description** Operator[a]	**Modifier** (see NOTE)	Explanation
1	LD	N	Set current result equal to operand
2	ST	N	Store current result to operand location
3	S[e] , R[e]		Set operand to 1 if current result is Boolean 1 Reset operand to 0 if current result is Boolean 1
4	AND	N, (	Logical AND
5	&	N, (	Logical AND
6	OR	N, (	Logical OR
7	XOR	N, (	Logical exclusive OR
8	NOT[d]		Logical negation (one's complement)
9	ADD	(	Addition
10	SUB	(	Subtraction
11	MUL	(	Multiplication
12	DIV	(	Division
13	MOD	(	Modulo-division
14	GT	(	Comparison: >
15	GE	(	Comparison: >=
16	EQ	(	Comparison: =
17	NE	(	Comparison: <>
18	LE	(	Comparison: <=
9	LT	(	Comparison: <
20	JMP[b]	C, N	Jump to label
21	CAL[c]	C, N	Call function block (see Table 69)
22	RET[f]	C, N	Return from called function, function block or program
23	)		Evaluate deferred operation
24	ST?		Assignment attempt Store with test

See preceding text for explanation of modifiers and evaluation of expressions.

[a] Unless otherwise noted, these operators shall be either overloaded or typed.

[b] The operand of a JMP instruction shall be the label of an instruction to which execution is to be transferred. When a JMP instruction is contained in an ACTION... END_ACTION construct, the operand shall be a label within the same construct.

[c] The operand of this instruction shall be the name of a function block instance to be called.

[d] The result of this operation shall be the bitwise Boolean negation (one's complement) of the current result.

[e] The type of the operand of this instruction shall be BOOL.

[f] This instruction does not have an operand.

7.2.4 Functions and function blocks

7.2.4.1 General

The general rules and features for function calls and for function block calls apply also in IL.

The features for the call of function blocks and functions are defined in Table 69.

7.2.4.2 Function

Functions shall be called by placing the function name in the operator field. The parameters may be given all together in one operand field or each parameter in an operand field line by line.

In case of the non-formal call the first parameter of a function need not to be contained in the parameter, but the current result shall be used as the first parameter of the function. Additional parameters (starting with the second one), if required, shall be given in the operand field, separated by commas, in the order of their declaration.

Functions may have a result. As shown in features 3 in Table 69 the successful execution of a RET instruction or upon reaching the end of the POU the POU delivers the result as the “current result”.

If a function is called which does not have a result, the “current result” is undefined.

7.2.4.3 Function block

Functions block shall be called by placing the keyword CAL in the operator field and the function block instance name in the operand field. The parameters may be given all together or each parameter may be placed in an operand field.

Function blocks can be called conditionally and unconditionally via the EN operator.

All parameter assignments defined in a parameter list of a conditional function block call shall only be performed together with the call, if the condition is true.

If a function block instance is called, the “current result” is undefined.

7.2.4.4 Methods

Methods shall be called by placing the function block instance name, followed by a single period “.”, and the method name in the operator field. The parameters may be given all together in one operand field or each parameter in an operand field line by line.

In case of the non-formal call the first parameter of a method need not to be contained in the parameter, but the current result shall be used as the first parameter of the function. Additional parameters (starting with the second one), if required, shall be given in the operand field, separated by commas, in the order of their declaration.

Methods may have a result. As shown in features 4 in Table 69 the successful execution of a RET instruction or upon reaching the end of the POU the POU delivers the result as the "current result".

If a method is called which does not have a result, the "current result" is undefined.

Table 69 shows the alternative calls of the IL language.

Table 69 – Calls for IL language

No.	**Description**	**Example** (NOTE)
1a	Function block call with non-formal parameter list	CAL C10(%IX10, FALSE, A, OUT, B) CAL CMD_TMR(%IX5, T#300ms, OUT, ELAPSED)

1b	Function block call with formal parameter list	CAL C10(// FB instance name
		    CU := %IX10,
		    R := FALSE,
		    PV := A,
		    Q => OUT,
		    CV => B)
		CAL CMD_TMR(
		    IN := %IX5,
		    PT := T#300ms,
		    Q => OUT,
		    ET => ELAPSED,
		    ENO => ERR)

2	Function block call with load/store of standard input parameters	LD A ADD 5 ST C10.PV LD %IX10 ST C10.CU CAL C10 // FB instance name LD C10.CV // current result

3a	Function call with formal parameter list	LIMIT(// Function name
		 EN := COND,
		 IN := B,
		 MN := 1,
		 MX := 5,
		 ENO => TEMPL
		)
		ST A // Current result new

3b	Function call with non-formal parameter list	LD 1 // set current result LIMIT B, 5 // and use it as IN ST A // new current result

4a	Method call with formal parameter list	FB_INST.M1(// Method name
		 EN := COND,
		 IN := B,
		 MN := 1,
		 MX := 5,
		 ENO => TEMPL
		)
		ST A // Current result new

4b	Method call with non-formal parameter list	LD 1 // set current result FB_INST.M1 B, 5 // and use it as IN ST A // new current result

No.	Description	**Example** (NOTE)

NOTE A declaration such as

```
VAR
  C10    : CTU;
  CMD_TMR: TON;
  A, B   : INT;
  ELAPSED: TIME;
  OUT, ERR, TEMPL, COND: BOOL;
END_VAR
```

is assumed in the above examples.

The standard input operators of standard function blocks defined in Table 70 can be used in conjunction with feature 2 (load/store) in Table 69. This call is equivalent to a `CAL` with a parameter list, which contains only one variable with the name of the input operator.

Parameters, which are not supplied, are taken from the last assignment or, if not present, from initialization. This feature supports problem situations, where events are predictable and therefore only one variable can change from one call to the next.

EXAMPLE 1

Together with the declaration

```
VAR C10: CTU; END_VAR
```

the instruction sequence

```
LD     15
PV     C10
```

gives the same result as

```
CAL    C10(PV:=15)
```

The missing inputs `R` and `CU` have values previously assigned to them. Since the `CU` input detects a rising edge, only the `PV` input value will be set by this call; counting cannot happen because an unsupplied parameter cannot change. In contrast to this, the sequence

```
LD     %IX10
CU     C10
```

results in counting at maximum in every second call, depending on the change rate of the input `%IX10`. Every call uses the previously set values for `PV` and `R`.

EXAMPLE 2

With bistable function blocks, taking a declaration

```
VAR FORWARD: SR; END_VAR
```

this results into an implicit conditional behavior. The sequence

```
LD      FALSE
S1      FORWARD
```

does not change the state of the bistable FORWARD. A following sequence

```
LD      TRUE
R       FORWARD
```

resets the bistable.

Table 70 – Standard function block operators for IL language

No.	Function block	Input operator	Output operator
1	SR	S1,R	Q
2	RS	S,R1	Q
3	F/R_TRIG	CLK	Q
4	CTU	CU,R,PV	CV, Q, also RESET
5	CTD	CD,PV	CV, Q
6	CTUD	CU,CD,R,PV	CV, QU,QD, also RESET
7	TP	IN,PT	CV, Q
8	TON	IN,PT	CV, Q
9	TOF	IN,PT	CV, Q
NOTE LD (Load) is not necessary as a Standard Function Block input operator, because the LD functionality is included in PV.			

Parameters, which are not supplied, are taken from the last assignment or, if not present, from initialization. This feature supports problem situations, where events are predictable and therefore only one variable can change from one call to the next.

7.3 Structured Text (ST)

7.3.1 General

The textual programming language “Structured Text, ST” is derived from the programming language Pascal for the usage in this standard.

7.3.2 Expressions

In the ST language, the end of a textual line shall be treated the same as a space (SP) character.

An expression is a construct which, when evaluated, yields a value corresponding to one of the data types. The maximum allowed length of expressions is an Implementer specific.

Expressions are composed of operators and operands. An operand shall be a literal, an enumerated value, a variable, a call of function with result, call of method with result, call of function block instance with result or another expression.

The operators of the ST language are summarized in Table 71.

The Implementer shall define explicit and implicit type conversions.

The evaluation of an expression shall apply the following rules:

1. The operators apply the operands in a sequence defined by the operator precedence shown in Table 71. 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.

   EXAMPLE 1

   If A, B, C, and D are of type INT with values 1, 2, 3, and 4, respectively, then
   `A+B-C*ABS(D)`
   is calculated to -9, and
   `(A+B-C)*ABS(D)`
   is calculated to 0.

2. Operators of equal precedence shall be applied as written in the expression from left to right.

   EXAMPLE 2
   `A+B+C` is evaluated as `(A+B)+C`.

3. When an operator has two operands, the leftmost operand shall be evaluated first.

   EXAMPLE 3
   In the expression
   `SIN(A)*COS(B)` the expression `SIN(A)` is evaluated first,
   followed by `COS(B)`, followed by evaluation of the product.

4. Boolean expressions may be evaluated only to the extent necessary to determine the resultant value including possible side effects. The extent to which a Boolean expression is evaluated is Implementer specific.

   EXAMPLE 4
   For the expression `(A>B) & (C<D)` it is sufficient, if
   `A<=B` , to evaluate only `(A>B)`, to decide
   that the value of the expression is `FALSE`.

5. Functions and methods may be called as elements of expressions consisting of the function or method name followed by a parenthesized list of parameters.
6. When an operator in an expression can be represented as one of the overloaded functions, conversion of operands and results shall follow the rule and examples given here.

The following conditions in the execution of operators shall be treated as errors:

a) An attempt is made to divide by zero.
b) Operands are not of the correct data type for the operation.
c) The result of a numerical operation exceeds the range of values for its data type.

Table 71 – Operators of the ST language

No.	Description Operation[a]	Symbol	Example	Precedence
1	Parentheses	(expression)	`(A+B/C),  (A+B)/C,  A/(B+C)`	11 (Highest)

2	Evaluation of result of function and method – if a result is declared	Identifier (parameter list)	`LN(A), MAX(X,Y),`
			`myclass.my_method(x)`
				10

3	Dereference	^	`R^`	9
4	Negation	-	`-A, - A`	8
5	Unary Plus	+	`+B, + B`	8
5	Complement	NOT	`NOT C`	8
7	Exponentiation[b]	**	`A**B, B ** B`	7

No.	Description Operation[a]	Symbol	Example	Precedence
8	Multiply	`*`	`A*B,  A * B`	6
9	Divide	`/`	`A/B,  A / B / D`	6
10	Modulo	`MOD`	`A MOD B`	6
11	Add	`+`	`A+B,  A + B + C`	5
12	Subtract	`-`	`A-B,  A - B - C`	5
13	Comparison	`< , > , <= , >=`	`A<B` ~~A < B < C~~	4
14	Equality	`=`	`A=B,  A=B & B=C`	4
15	Inequality	`<>`	`A<>B, A <> B`	4
16a	Boolean `AND`	`&`	`A&B, A & B, A & B & C`	3
16b	Boolean `AND`	`AND`	`A AND B`	3
17	Boolean Exclusive `OR`	`XOR`	`A XOR B`	2
18	Boolean `OR`	`OR`	`A OR B`	1 (Lowest)

[a] The same rules apply to the operands of these operators as to the inputs of the corresponding standard functions.

[b] The result of evaluating the expression `A**B` shall be the same as the result of evaluating the function `EXPT(A,B)`.

7.3.3 Statements

7.3.3.1 General

The statements of the ST language are summarized in Table 72. The maximum allowed length of statements is an Implementer specific.

Table 72 – ST language statements

No.	Description	Examples
1	**Assignment** Variable:= expression;	
1a	Variable and expression of elementary data type	`A:= B;  CV:= CV+1; C:= SIN(X);`
1b	Variables and expression of different elementary data types with implicit type conversion according Figure 11	`A_Real:= B_Int;`

1c	Variable and expression of user-defined type	`A_Struct1:= B_Struct1;`
		`C_Array1 := D_Array1;`

1d	Instances of function block type	`A_Instance1:= B_Instance1;`
	Call	
2a [b]	Function call	`FCT(17);`

2b [b]	Function block call and FB output usage	`CMD_TMR(IN:= bIn1, PT:= T#300ms);`
		`A:= CMD_TMR.Q;`

2c [b]	Method call	`FB_INST.M1(17);`
3	`RETURN`	`RETURN;`

No.	Description	Examples
	Selection	

4	IF ...
	THEN ...
	  ELSIF ...
	  THEN ...
	ELSE ...END_IF
		D:= B*B - 4.0*A*C;
		IF D < 0.0
		THEN NROOTS:= 0;
		    ELSIF D = 0.0
		    THEN
		      NROOTS:= 1;
		      X1:= - B/(2.0*A);
		    ELSE
		      NROOTS:= 2;
		      X1:= (- B + SQRT(D))/(2.0*A);
		      X2:= (- B - SQRT(D))/(2.0*A);
		END_IF;

5	CASE ... OF
	    ...
	    ELSE ...
	END_CASE
		TW:= WORD_BCD_TO_INT(THUMBWHEEL);
		TW_ERROR:= 0;
		CASE TW OF
		  1,5: DISPLAY:= OVEN_TEMP;
		  2: DISPLAY:= MOTOR_SPEED;
		  3: DISPLAY:= GROSS - TARE;
		  4,6..10: DISPLAY:= STATUS(TW - 4);
		  ELSE DISPLAY := 0;
		         TW_ERROR:= 1;
		END_CASE;
		QW100:= INT_TO_BCD(DISPLAY);

	Iteration	

6	FOR ... TO ... BY ... DO
	    ...
	END_FOR
		J:= 101;
		FOR I:= 1 TO 100 BY 2 DO
		    IF WORDS[I] = 'KEY' THEN
		      J:= I;
		      EXIT;
		    END_IF;
		END_FOR;

7	WHILE ... DO
	...
	END_WHILE
		J:= 1;
		WHILE J <= 100 & WORDS[J] <> 'KEY' DO
		  J:= J+2;
		END_WHILE;

8	REPEAT ...
	 UNTIL ...
	END_REPEAT
		J:= -1;
		REPEAT
		 J:= J+2;
		 UNTIL J = 101 OR WORDS[J] = 'KEY'
		END_REPEAT;

9 [a]	CONTINUE	J:= 1;
		WHILE (J <= 100 AND WORDS[J] <> 'KEY') DO
		..IF (J MOD 3 = 0) THEN
		    CONTINUE;
		  END_IF;
		(* if j=1,2,4,5,7,8, ... then this statement*);
		   ...
		END_WHILE;

10 [a]	EXIT an iteration	EXIT; (see also in feature 6)
11	Empty Statement	;

a If the EXIT or CONTINUE statement (feature 9 or 11) is supported, then it shall be supported for all of the iteration statements (FOR, WHILE, REPEAT) which are supported in the implementation.

b If the function, function block type, or method provides a result and the call is not in an expression of an assignment, the result is discarded.

7.3.3.2 Assignment (Comparison, result, call)

7.3.3.2.1 General

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

For instance, the statement

```
A:= B;
```

would be used to replace the single data value of variable `A` by the current value of variable `B` if both were of type `INT` or the variable `B` can implicitly be converted to type `INT`.

If `A` and `B` are multi-element variables the data types of `A` and `B` shall be the same. In this case the elements of the variable `A` get the values of the elements of variable `B`.

For instance, if both `A` and `B` were of type `ANALOG_CHANNEL_CONFIGURATION` then the values of all the elements of the structured variable `A` would be replaced by the current values of the corresponding elements of variable `B`.

7.3.3.2.2 Comparison

A comparison returns its result as a Boolean value. A comparison shall consist of a variable reference on the left-hand side, followed by a comparison operator, followed by a variable reference on the right-hand side. The variables can be single or multi-element variables.

The comparison

```
A = B
```

would be used to compare the data value of variable `A` by the value of variable `B` if both were of the same data type or one of the variables can implicitly be converted to the data type of the other one.

If `A` and `B` are multi-element variables the data types of `A` and `B` shall be the same. In this case the values of the elements of the variable `A` is compared to the values of the elements of variable `B`.

7.3.3.2.3 Result

An assignment is also used to assign the result of a function, function block type, or method. If a result is defined for this POU at least one assignment to the name of this POU shall be made. The value returned shall be the result of the most recent evaluation of such an assignment. It is an error to return from the evaluation with an `ENO` value of `TRUE`, or with a non-existent `ENO` output, unless at least one such assignment has been made.

7.3.3.2.4 Call

Function, method, and function block control statements consist of the mechanisms for calling this POU and for returning control to the calling entity before the physical end of the POU.

- **`FUNCTION`**

  Function shall be called by a statement consisting of the name of the function followed by a parenthesized list of parameters as illustrated in Table 72.

  The rules and features defined in 6.6.1.7 for function calls apply.

- **FUNCTION_BLOCK**

  Function blocks shall be called by a statement consisting of the name of the function block instance followed by a parenthesized list of parameters, as illustrated in Table 72.

- **METHOD**

  Methods shall be called by a statement consisting of the name of the instance followed by '.' and the method name and a parenthesized list of parameters.

- **RETURN**

  The RETURN statement shall provide early exit from a function, function block or program (for example, as the result of the evaluation of an IF statement).

7.3.3.3 Selection statements (IF, CASE)

7.3.3.3.1 General

Selection statements include the IF and CASE statements. A selection statement selects one (or a group) of its component statements for execution, based on a specified condition. Examples of selection statements are given in Table 72.

7.3.3.3.2 IF

The IF statement specifies that a group of statements is to be executed only if the associated Boolean expression evaluates to the value 1 (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.

7.3.3.3.3 CASE

The CASE statement consists of an expression which shall evaluate to a variable of elementary data type (the “selector”), and a list of statement groups, each group being labeled by one or more literals, enumerated values, or subranges, as applicable. The data types of these labels shall match to the data type of the selector variable i.e. the selector variable shall be able to be compared with the labels.

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.

The maximum allowed number of selections in CASE statements is an Implementer specific.

7.3.3.4 Iteration statements (WHILE, REPEAT, EXIT, CONTINUE , FOR)

7.3.3.4.1 General

Iteration statements specify that the group of associated statements shall be executed repeatedly.

The WHILE and REPEAT statements shall not be used to achieve inter-process synchronization, for example as a "wait loop" with an externally determined termination condition. The SFC elements shall be used for this purpose.

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

The FOR statement is used if the number of iterations can be determined in advance; otherwise, the WHILE or REPEAT constructs are used.

7.3.3.4.2 FOR

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 shall be expressions of the same integer type (for example, 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. If the BY construct is omitted the increment value defaults to 1.

EXAMPLE

The FOR loop specified by

```
FOR I:= 3 TO 1 STEP -1 DO ...;
```

terminates when the value of the variable I reaches 0.

The test for the termination condition is made at the beginning of each iteration, so that the statement sequence is not executed if the value of the control variable exceeds the final value i.e. the value of the control variable is greater respectively less than the final value if the increment value is positive respectively negative. The value of the control variable after completion of the FOR loop is Implementer specific.

The iteration is terminated when the value of the control variable is outside the range specified by the TO construct.

A further example of the usage of the FOR statement is given in feature 6 of Table 72. In this example, the FOR loop is used to determine the index J of the first occurrence (if any) of the string 'KEY' in the odd-numbered elements of an array of strings WORDS with a subscript range of (1..100). If no occurrence is found, J will have the value 101.

7.3.3.4.3 WHILE

The WHILE statement causes execution of the sequence of statements up to the END_WHILE keyword. The statements are repeatedly executed 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 FOR...END_FOR example can be rewritten using the WHILE...END_WHILE construction shown in Table 72.

7.3.3.4.4 REPEAT

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 WHILE...END_WHILE example can be rewritten using the WHILE ...END_WHILE construct also shown in Table 72.

7.3.3.4.5 CONTINUE

The CONTINUE statement shall be used to jump over the remaining statements of the iteration loop in which the CONTINUE is located after the last statement of the loop right before the loop terminator (END_FOR, END_WHILE, or END_REPEAT).

EXAMPLE

After executing the statements, the value of the variable if the value of the Boolean variable FLAG=0, and SUM=9 if FLAG=1.

```
SUM:= 0;
FOR I:= 1 TO 3 DO
  FOR J:= 1 TO 2 DO
    SUM:= SUM + 1;
    IF FLAG THEN

       CONTINUE;

    END_IF;
    SUM:= SUM + 1;
  END_FOR;
  SUM:= SUM + 1;
END_FOR;
```

7.3.3.4.6 EXIT

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

When the EXIT statement is located within nested iterative constructs, 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

After executing of the statements, the value of the variable SUM=15 if the value of the Boolean variable FLAG= 0, and SUM=6 if FLAG=1.

```
SUM:= 0;
FOR I:= 1 TO 3 DO
  FOR J:= 1 TO 2 DO
    SUM:= SUM + 1;

    IF FLAG THEN

        EXIT;
    END_IF;
    SUM:= SUM + 1;
  END_FOR;
  SUM:= SUM + 1;
END_FOR;
```

8 Graphic languages

8.1 Common elements

8.1.1 General

The graphic languages defined in this standard are LD (Ladder Diagram) and FBD (Function Block Diagram). The sequential function chart (SFC) elements can be used in conjunction with either of these languages.

The elements apply to both the graphic languages in this standard, that is, LD and FBD, and to the graphic representation of sequential function chart (SFC) elements.

8.1.2 Representation of variables and instances

All supported data types shall be accessible as operands or parameters in the graphical languages.

All supported declarations of instances shall be accessible in the graphical languages.

The usage of expression as parameters or as subscript of arrays is beyond the scope of this part of the IEC 61131 series.

EXAMPLE

```
TYPE                                  Type declarations
  SType: STRUCT
    x: BOOL;
    a: INT;
    t: TON;
    END_STRUCT;
END_TYPE;
```

```
VAR                                   Variable declarations
  x: BOOL;
  i: INT;
  Xs: ARRAY [1..10] OF BOOL;
  S:  SType;
  Ss: ARRAY [0..3]  OF SType;
  t:  TON;
  Ts: ARRAY [0..20] OF TON;
END_VAR
```

a) Type and variable declarations

Uses an operand:

```
               +--------+
      x        |  myFct |
------| |------|IN      |
               +--------+
```

as an elementary variable

```
               +--------+
    Xs[3]      |  myFct |
------| |------|IN      |
               +--------+
```

as an array element with constant subscript

```
               +--------+
    Xs[i]      |  myFct |
------| |------|IN      |
               +--------+
```

as an array element with variable subscript

```
               +--------+
     S.x       |  myFct |
------| |------|IN      |
               +--------+
```

as an element of a structure

```
               +--------+
   Ss[3].x     |  myFct |
------| |------|IN      |
               +--------+
```

as an element of a structured array

b) Representation of operands

Instance used as a parameter:

```
               +--------+
     t.Q       | myFct2 |
------| |------|aTON    |
               +--------+
```

as a normal instance

```
               +--------+
   Ts[10].Q    | myFct2 |
------| |------|aTON    |
               +--------+
```

as an array element with constant subscript

```
               +--------+
    Ts[i].Q    | myFct2 |
------| |------|aTON    |
               +--------+
```

as an array element with variable subscript

```
               +--------+
     S.t       | myFct2 |
------| |------|aTON    |
               +--------+
```

as an element of a structure

```
               +--------+
   Ss[2].t     | myFct2 |
------| |------|aTON    |
               +--------+
```

as an element of a structured array

c) Representation of an instance as parameter

```
                    t
               +--------+
        x      |  TON   |
------| |------|IN     Q|
               |PT    ET|
               +--------+
```

Instance as:

plain instance

```
                  Ts[12]
               +--------+
        x      |  TON   |
------| |------|IN     Q|
               |PT    ET|
               +--------+
```

array element with constant subscript

```
                   Ts[i]
               +--------+
        x      |  TON   |
------| |------|IN     Q|
               |PT    ET|
               +--------+
```

array element with variable subscript

```
                   s.t
               +--------+
        x      |  TON   |
------| |------|IN     Q|
               |PT    ET|
               +--------+
```

element of a structure

```
                 Ss[i].t
               +--------+
        x      |  TON   |
------| |------|IN     Q|
               |PT    ET|
               +--------+
```

element of a structured array

d) Representation of an instance call

8.1.3 Representation of lines and blocks

The usage of letters, semigraphic or graphic for the representation of graphical elements is Implementer specific and not a normative requirement.

The graphic language elements defined in this Clause 8 are drawn with line elements using characters from the character set. Examples are shown below.

Lines can be extended by the use of connector. No storage of data or association with data elements shall be associated with the use of connectors; hence, to avoid ambiguity, it shall be an error if the identifier used as a connector label is the same as the name of another named element within the same program organization unit.

Any restrictions on network topology in a particular implementation shall be expressed as Implementer specific.

```
EXAMPLES  Graphical elements

Horizontal lines                                        -----

                                                          |
Vertical lines                                            |
                                                          |
Horizontal/vertical connection (node)                   --+--
                                                          |
                                                          |
Line crossings without connection (no node)           ----|----
                                                          |

                                                      |   |
                                                  ----+   +----
Connected and non-connected corners (nodes)           |   |
                                                  ------+ +----
                                                      | | |

                                                         |
                                                     +--------+
                                                  ---|        |
Blocks with connecting lines                         |        |---
                                                  ---|        |
                                                     +--------+
                                                         |

                                                ---------->OTTO>
Connectors and continuation                     >OTTO>----------
```

8.1.4 Direction of flow in networks

A network is defined as a maximal set of interconnected graphic elements, excluding the left and right rails in the case of networks in the LD language. Provision shall be made to associate with each network or group of networks in a graphic language a network label delimited on the right by a colon (:). This label shall have the form of an identifier or an unsigned decimal integer. The scope of a network and its label shall be local to the program organization unit in which the network is located.

Graphic languages are used to represent the flow of a conceptual quantity through one or more networks representing a control plan, that is:

- “Power flow”,
  analogous to the flow of electric power in an electromechanical relay system, typically used in relay ladder diagrams.

  Power flow in the LD language shall be from left to right.

- “Signal flow”,
  analogous to the flow of signals between elements of a signal processing system, typically used in function block diagrams.

  Signal flow in the FBD language shall be from the output (right-hand) side of a function or function block to the input (left-hand) side of the function or function block(s) so connected.

- “Activity flow”,
  analogous to the flow of control between elements of an organization, or between the steps of an electromechanical sequencer, typically used in sequential function charts.

  Activity flow between the SFC elements shall be from the bottom of a step through the appropriate transition to the top of the corresponding successor step(s).

8.1.5 Evaluation of networks

8.1.5.1 General

The order in which networks and their elements are evaluated is not necessarily the same as the order in which they are labeled or displayed. Similarly, it is not necessary that all networks be evaluated before the evaluation of a given network can be repeated.

However, when the body of a program organization unit consists of one or more networks, the results of network evaluation within the said body shall be functionally equivalent to the observance of the following rules:

a) No element of a network shall be evaluated until the states of all of its inputs have been evaluated.

b) The evaluation of a network element shall not be complete until the states of all of its outputs have been evaluated.

c) The evaluation of a network is not complete until the outputs of all of its elements have been evaluated, even if the network contains one of the execution control elements.

d) The order in which networks are evaluated shall conform to the provisions for the LD language and for the FBD language.

8.1.5.2 Feedback path

A feedback path is said to exist in a network when the output of a function or function block is used as the input to a function or function block which precedes it in the network; the associated variable is called a feedback variable.

For instance, the Boolean variable `RUN` is the feedback variable in the example shown below. A feedback variable can also be an output element of a function block data structure.

Feedback paths can be utilized in the graphic languages defined, subject to the following rules:

a) Explicit loops such as the one shown in the example below a) shall only appear in the FBD language.

b) It shall be possible for the user to utilize an Implementer specific means to determine the order of execution of the elements in an explicit loop, for instance by selection of feedback variables to form an implicit loop as shown in the example below b).

c) Feedback variables shall be initialized by one of the mechanisms. The initial value shall be used during the first evaluation of the network. It shall be an error if a feedback variable is not initialized.

d) Once the element with a feedback variable as output has been evaluated, the new value of the feedback variable shall be used until the next evaluation of the element.

EXAMPLE Feedback path

```
                     +---+
            ENABLE---| & |-----RUN---+
                 +---|   |           |
         +---+   |   +---+           |
START1---|>=1|---+                   |
START2---|   |                       |
      +--|   |                       |
      |  +---+                       |
      +------------------------------+
```

a) Explicit loop

```
                     +---+
            ENABLE---| & |-----RUN
                 +---|   |
         +---+   |   +---+
START1---|>=1|---+
START2---|   |
   RUN---|   |
         +---+
```

b) Implicit loop

```
|  START1    ENABLE     RUN   |
+---| |----+---| |------( )---+
|  START2   |                 |
+---| |----+                  |
|   RUN     |                 |
+---| |----+                  |
|                             |
```

c) LD language equivalent

8.1.6 Execution control elements

Transfer of program control in the LD and FBD languages shall be represented by the graphical elements shown in Table 73.

Jumps shall be shown by a Boolean signal line terminated in a double arrowhead. The signal line for a jump condition shall originate at a Boolean variable, at a Boolean output of a function or function block, or on the power flow line of a ladder diagram. A transfer of program control to the designated network label shall occur when the Boolean value of the signal line is 1 (TRUE); thus, the unconditional jump is a special case of the conditional jump.

The target of a jump shall be a network label within the program organization unit body or method body within which the jump occurs. If the jump occurs within an ACTION ...END_ACTION construct, the target of the jump shall be within the same construct.

Conditional returns from functions and function blocks shall be implemented using a RETURN construction as shown in Table 73. Program execution shall be transferred back to the calling entity when the Boolean input is 1 (TRUE), and shall continue in the normal fashion when the Boolean input is 0 (FALSE). Unconditional returns shall be provided by the physical end of the function or function block, or by a RETURN element connected to the left rail in the LD language, as shown in Table 73.

Table 73 – Graphic execution control elements

<table>
<tr><th>No.</th><th>Description</th><th>Explanation</th></tr>
<tr><td></td><td><b>Unconditional jump</b></td><td></td></tr>
<tr><td>1a</td><td>FBD language</td><td><pre>1---->>LABELA</pre></td></tr>
<tr><td>1b</td><td>LD language</td><td><pre>|
+---->>LABELA
|</pre></td></tr>
<tr><td></td><td><b>Conditional jump</b></td><td></td></tr>
<tr><td>2a</td><td>FBD language</td><td>Example:
jump condition, jump target<pre>
 X---->>LABELB

        +---+
bvar0---| & |--->>NEXT
bvar50--| |
        +---+
NEXT:
        +---+
bvar5---|>=1|---bOut0
bvar60--| |
        +---+</pre></td></tr>
<tr><td>2b</td><td>LD language</td><td>Example:
jump condition, jump target<pre>
| X
+-| |---->>LABELB
|
|
| bvar0   bvar50
+---| |-----| |--->>NEXT
|
|
NEXT:
| bvar5   bOut0 |
+----| |----+----( )---+
| bvar60 | |
+----| |----+ |
| |</pre></td></tr>
<tr><td></td><td><b>Conditional return</b></td><td></td></tr>
<tr><td>3a</td><td>LD language</td><td><pre>| X
+--| |---<RETURN>
|</pre></td></tr>
<tr><td>3b</td><td>FBD language</td><td><pre>X---<RETURN></pre></td></tr>
<tr><td></td><td><b>Unconditional return</b></td><td></td></tr>
<tr><td>4</td><td>LD language</td><td><pre>|
+---<RETURN>
|</pre></td></tr>
</table>

8.2 Ladder diagram (LD)

8.2.1 General

Subclause 8.2 defines the LD language for ladder diagram programming of programmable controllers.

A LD program enables the programmable controller to test and modify data by means of standardized graphic symbols. These symbols are laid out in networks in a manner similar to a “rung” of a relay ladder logic diagram. LD networks are bounded on the left and right by power rails.

The usage of letters, semigraphic or graphic for the representation of graphical elements is Implementer specific and not a normative requirement.

8.2.2 Power rails

As shown in Table 74, the LD network shall be delimited on the left by a vertical line known as the left power rail, and on the right by a vertical line known as the right power rail. The right power rail may be explicit or implied.

8.2.3 Link elements and states

As shown in Table 74, link elements may be horizontal or vertical. The state of the link element shall be denoted “ON” or “OFF”, corresponding to the literal Boolean values 1 or 0, respectively. The term link state shall be synonymous with the term power flow.

The state of the left rail shall be considered ON at all times. No state is defined for the right rail.

A horizontal link element shall be indicated by a horizontal line. A horizontal link element transmits the state of the element on its immediate left to the element on its immediate right.

The vertical link element shall consist of a vertical line intersecting with one or more horizontal link elements on each side. The state of the vertical link shall represent the inclusive OR of the ON states of the horizontal links on its left side, that is, the state of the vertical link shall be:

- OFF if the states of all the attached horizontal links to its left are OFF;
- ON if the state of one or more of the attached horizontal links to its left is ON.

The state of the vertical link shall be copied to all of the attached horizontal links on its right. The state of the vertical link shall not be copied to any of the attached horizontal links on its left.

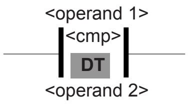
Table 74 – Power rails and link elements

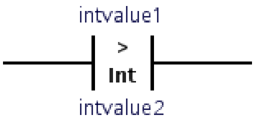
<table>
<tr><th>No.</th><th>Description</th><th>Symbol</th></tr>
<tr><td>1</td><td>Left power rail
(with attached horizontal link)</td><td>|
+---
|</td></tr>
<tr><td>2</td><td>Right power rail
(with attached horizontal link)</td><td> |
---+
   |</td></tr>
<tr><td>3</td><td>Horizontal link</td><td>-----------</td></tr>
<tr><td>4</td><td>Vertical link
(with attached horizontal links)</td><td> |
----+----
----+
    |
    +----</td></tr>
</table>

8.2.4 Contacts

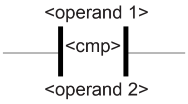
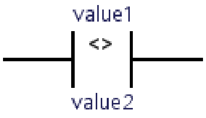
A contact is an element which imparts a state to the horizontal link on its right side which is equal to the Boolean AND of the state of the horizontal link at its left side with an appropriate function of an associated Boolean input, output, or memory variable. A contact does not modify the value of the associated Boolean variable. Standard contact symbols are given in Table 75.

Table 75 – Contacts

<table>
<tr><th>No.</th><th>Description</th><th>Explanation, Symbol</th></tr>
<tr><td></td><td><b>Static contacts</b></td><td></td></tr>
<tr><td>1</td><td>Normally open contact</td><td>***
--| |--
The state of the left link is copied to the right link if the state of the associated Boolean variable (indicated by "***") is ON. Otherwise, the state of the right link is OFF.</td></tr>
<tr><td>2</td><td>Normally closed contact</td><td>***
--|/|--
The state of the left link is copied to the right link if the state of the associated Boolean variable is OFF. Otherwise, the state of the right link is OFF.</td></tr>
<tr><td></td><td><b>Transition-sensing contacts</b></td><td></td></tr>
<tr><td>3</td><td>Positive transition-sensing contact</td><td>***
--|P|--
The state of the right link is ON from one evaluation of this element to the next when a transition of the associated variable from OFF to ON is sensed at the same time that the state of the left link is ON. The state of the right link shall be OFF at all other times.</td></tr>
<tr><td>4</td><td>Negative transition-sensing contact</td><td>***
--|N|--
The state of the right link is ON from one evaluation of this element to the next when a transition of the associated variable from ON to OFF is sensed at the same time that the state of the left link is ON. The state of the right link shall be OFF at all other times.</td></tr>
<tr><td>5a</td><td>Compare contact (typed)</td><td>

The state of the right link is ON from one evaluation of this element to the next when the left link is ON and the <cmp> result of the operands 1 and 2 is true.
The state of the right link shall be OFF otherwise.
< cmp> may be substituted by one of the compare functions that are valid for the given data type.
DT is the data type of both given operands.
Example:


If the left link is ON and (intvalue1 > intvalue2) the right link switches to ON. Both intvalue1 and intvalue2 are of the data type INT</td></tr>
</table>

No.	Description	Explanation, Symbol
5b	Compare contact, (overloaded)	 The state of the right link is ON from one evaluation of this element to the next when the left link is ON and the <cmp> result of the operands 1 and 2 is true. The state of the right link shall be OFF otherwise. <cmp> may be substituted by one of the compare functions that are valid for the operands data type. The rules defined in 6.6.1.7 shall apply. Example:  If the left link is ON and (value1 <> value2) the right link switches to ON.

8.2.5 Coils

A coil copies the state of the link on its left to the link on its right without modification, and stores an appropriate function of the state or transition of the left link into the associated Boolean variable. Standard coil symbols are given in Table 76.

EXAMPLE

In the rung shown below, the value of the Boolean output is always TRUE, while the value of outputs c, d and e upon completion of an evaluation of the rung is equal to the value of the input b.

```
|    a     b       c     d    |
+--( )--|  |--+--( )---( )--+
|             |      e      |
|             +-----( )-----+
```

Table 76 – Coils

No.	Description	Explanation, Symbol
	Momentary coils	
1	Coil	*** --( )– The state of the left link is copied to the associated Boolean variable and to the right link.
2	Negated coil	*** --(/)– The state of the left link is copied to the right link. The inverse of the state of the left link is copied to the associated Boolean variable, that is, if the state of the left link is OFF, then the state of the associated variable is ON, and vice versa.

No.	Description	Explanation, Symbol
	Latched coils	
3	Set (latch) coil	*** --(S)- The associated Boolean variable is set to the ON state when the left link is in the ON state, and remains set until reset by a RESET coil.
4	Reset (unlatch) coil .	*** --(R)- The associated Boolean variable is reset to the OFF state when the left link is in the ON state, and remains reset until set by a SET coil.
	Transition-sensing coils	
8	Positive transition-sensing coil	*** --(P)- The state of the associated Boolean variable is ON from one evaluation of this element to the next when a transition of the left link from OFF to ON is sensed. The state of the left link is always copied to the right link.
9	Negative transition-sensing coil	*** --(N)- The state of the associated Boolean variable is ON from one evaluation of this element to the next when a transition of the left link from ON to OFF is sensed. The state of the left link is always copied to the right link.

8.2.6 Functions and function blocks

The representation of functions, methods, and function blocks in the LD language shall be with the following exceptions:

a) Actual variable connections may optionally be shown by writing the appropriate data or variable outside the block adjacent to the formal variable name on the inside.

b) At least one Boolean input and one Boolean output shall be shown on each block to allow for power flow through the block.

8.2.7 Order of network evaluation

Within a program organization unit written in LD, networks shall be evaluated in top to bottom order as they appear in the ladder diagram, except as this order is modified by the execution control elements.

8.3 Function Block Diagram (FBD)

8.3.1 General

Subclause 8.3 defines FBD, a graphic language for the programming of programmable controllers which is consistent, as far as possible, with IEC 60617-12. Where conflicts exist between this standard and IEC 60617-12, the provisions of this standard shall apply for the programming of programmable controllers in the FBD language.

8.3.2 Combination of elements

Elements of the FBD language shall be interconnected by signal flow lines following the conventions of 8.1.4.

Outputs of function blocks shall not be connected together. In particular, the “wired-OR” construct of the LD language is not allowed in the FBD language; an explicit Boolean “OR” block is required instead, as shown in the example below.

EXAMPLE Boolean OR

```
|    a       c   |
+---| |--+--( )--+
|    b   |       |
+---| |--+       |
|                |
```

a) “Wired-OR” in LD language

```
    +-----+
a---| >=1 |---c
b---|     |
    +-----+
```

b) Function in FBD language

8.3.3 Order of network evaluation

When a program organization unit written in the FBD language contains more than one network, the Implementer shall provide Implementer specific means by which the user may determine the order of execution of networks.

Annex A
(normative)

Formal specification of the languages elements

The syntax of the textual languages are defined in a variant of the "Extended BNF" (Extended Backus Naur Form.)

The syntax of this EBNF variant is as follows:

For the purposes of this Annex A, terminal textual symbols consist of the appropriate character string enclosed in paired single quotes. For example, a terminal symbol represented by the character string ABC is represented by 'ABC'.

Non-terminal textual symbols shall be represented by strings of lower-case letters, numbers, and the underline character (_), beginning with an upper-case letter.

Production rules

The production rules for textual languages are of the form
non_terminal_symbol: extended_structure;
This rule can be read as: "A non_terminal_symbol can consist of an extended_structure."

Extended structures can be constructed according to the following rules:

Any terminal symbol is an extended structure.

Any non-terminal symbol is an extended structure.

If S is an extended structure, then the following expressions are also extended structures:

(S)	meaning S itself
(S)*	closure, meaning zero or more concatenations of S.
(S)+	closure, meaning one or more concatenations of S.
(S)?	option, meaning zero or one occurrence of S.

If S1 and S2 are extended structure, then the following expressions are extended structures:

S1 | S2	alternation, meaning a choice of S1 or S2.
S1 S2	concatenation, meaning S1 followed by S2.

Concatenation precedes alternation, that is,

S1 | S2 S3	is equivalent to S1 | (S2 S3),
S1 S2 | S3	is equivalent to (S1 S2) | S3.

If S is an extended structure that denotes a single character or an alternation of single characters, then the following is also an extended structure:

~(S)	negation, meaning any single character that is not in S.

Negation precedes closure or option, that is,

~(S)*	is equivalent to (~(S))*.

The following symbols are used to denote certain characters or classes of characters:

.	Any single character
\'	The "single quote" character
\n	Newline
\r	Carriage return
\t	Tabulator

Comments within the grammar start with double slashes and end at the end of the line:
// This is a comment

// Table 1 - Character sets
// Table 2 - Identifiers

```
Letter                        : 'A'..'Z' | '_';
Digit                         : '0'..'9';
Bit                           : '0'..'1';
Octal_Digit                   : '0'..'7';
Hex_Digit                     : '0'..'9' | 'A'..'F';
Identifier                    : Letter ( Letter | Digit )*;
```

// Table 3 - Comments

```
Comment                       : '//' ~( '\n' | '\r' )* '\r' ? '\n' {$channel=HIDDEN;}
                              | '(*' ( options{greedy=false;}: . )* '*)' {$channel=HIDDEN;}
                              | '/*' ( options{greedy=false;}: . )* '*/' {$channel=HIDDEN;};
WS                            : ( ' ' | '\t' | '\r' | '\n' ) {$channel=HIDDEN;}; // white space
EOL                           : '\n';
```

// Table 4 - Pragma

```
Pragma                        : '{' ( options{greedy=false;}: . )* '}' {$channel=HIDDEN;};
```

// Table 5 - Numeric literal

```
Constant                      : Numeric_Literal | Char_Literal | Time_Literal | Bit_Str_Literal | Bool_Literal;
Numeric_Literal               : Int_Literal | Real_Literal;
Int_Literal                   : ( Int_Type_Name '#' )? ( Signed_Int | Binary_Int | Octal_Int | Hex_Int );
Unsigned_Int                  : Digit ( '_' ? Digit )*;
Signed_Int                    : ( '+' | '-' )? Unsigned_Int;
Binary_Int                    : '2#' ( '_' ? Bit )+;
Octal_Int                     : '8#' ( '_' ? Octal_Digit )+;
Hex_Int                       : '16#' ( '_' ? Hex_Digit )+;
Real_Literal                  : ( Real_Type_Name '#' )? Signed_Int '.' Unsigned_Int ( 'E' Signed_Int )?;
Bit_Str_Literal               : ( Multibits_Type_Name '#' )? ( Unsigned_Int | Binary_Int | Octal_Int | Hex_Int );
Bool_Literal                  : ( Bool_Type_Name '#' )? ( '0' | '1' | 'FALSE' | 'TRUE' );
```

// Table 6 - Character String literals
// Table 7 - Two-character combinations in character strings

```
Char_Literal                  : ( 'STRING#' )? Char_Str;
Char_Str                      : S_Byte_Char_Str | D_Byte_Char_Str;
S_Byte_Char_Str               : '\'' S_Byte_Char_Value + '\'';
D_Byte_Char_Str               : '"' D_Byte_Char_Value + '"';
S_Byte_Char_Value             : Common_Char_Value | '$\'' | '"' | '$' Hex_Digit Hex_Digit;
D_Byte_Char_Value             : Common_Char_Value | '\'' | '$"' | '$' Hex_Digit Hex_Digit Hex_Digit Hex_Digit;
Common_Char_Value             : ' ' | '!' | '#' | '%' | '&' | '('..'/' | '0'..'9' | ':'..'@' | 'A'..'Z' | '['..'`' | 'a'..'z' | '{'..'~'
                              | '$$' | '$L' | '$N' | '$P' | '$R' | '$T';
                                        // any printable characters except $, " and '
```

// Table 8 - Duration literals
// Table 9 – Date and time of day literals

```
Time_Literal                  : Duration | Time_Of_Day | Date | Date_And_Time;
Duration                      : ( Time_Type_Name | 'T' | 'LT' ) '#' ( '+' | '-' )? Interval;
Fix_Point                     : Unsigned_Int ( '.' Unsigned_Int )?;
Interval                      : Days | Hours | Minutes | Seconds | Milliseconds | Microseconds | Nanoseconds;
Days                          : ( Fix_Point 'd' ) | ( Unsigned_Int 'd' '_' ? )? Hours ?;
Hours                         : ( Fix_Point 'h' ) | ( Unsigned_Int 'h' '_' ? )? Minutes ?;
Minutes                       : ( Fix_Point 'm' ) | ( Unsigned_Int 'm' '_' ? )? Seconds ?;
Seconds                       : ( Fix_Point 's' ) | ( Unsigned_Int 's' '_' ? )? Milliseconds ?;
Milliseconds                  : ( Fix_Point 'ms' ) | ( Unsigned_Int 'ms' '_' ? )? Microseconds ?;
Microseconds                  : ( Fix_Point 'us' ) | ( Unsigned_Int 'us' '_' ? )? Nanoseconds ?;
Nanoseconds                   : Fix_Point 'ns';
Time_Of_Day                   : ( Tod_Type_Name | 'LTIME_OF_DAY' ) '#' Daytime;
Daytime                       : Day_Hour ':' Day_Minute ':' Day_Second;
Day_Hour                      : Unsigned_Int;
Day_Minute                    : Unsigned_Int;
Day_Second                    : Fix_Point;
Date                          : ( Date_Type_Name | 'D' | 'LD' ) '#' Date_Literal;
Date_Literal                  : Year '-' Month '-' Day;
Year                          : Unsigned_Int;
Month                         : Unsigned_Int;
Day                           : Unsigned_Int;
Date_And_Time                 : ( DT_Type_Name | 'LDATE_AND_TIME' ) '#' Date_Literal '-' Daytime;
```

// Table 10 - Elementary data types

```
Data_Type_Access              : Elem_Type_Name | Derived_Type_Access;
Elem_Type_Name                : Numeric_Type_Name | Bit_Str_Type_Name
                              | String_Type_Name | Date_Type_Name | Time_Type_Name;
Numeric_Type_Name             : Int_Type_Name | Real_Type_Name;
Int_Type_Name                 : Sign_Int_Type_Name | Unsign_Int_Type_Name;
Sign_Int_Type_Name            : 'SINT' | 'INT' | 'DINT' | 'LINT';
Unsign_Int_Type_Name          : 'USINT' | 'UINT' | 'UDINT' | 'ULINT';
```

```
Real_Type_Name              : 'REAL' | 'LREAL';
String_Type_Name            : 'STRING' ( '[' Unsigned_Int ']' )? | 'WSTRING' ( '[' Unsigned_Int ']' )? | 'CHAR' | 'WCHAR';
Time_Type_Name              : 'TIME' | 'LTIME';
Date_Type_Name              : 'DATE' | 'LDATE';
Tod_Type_Name               : 'TIME_OF_DAY' | 'TOD' | 'LTOD';
DT_Type_Name                : 'DATE_AND_TIME' | 'DT' | 'LDT';
Bit_Str_Type_Name           : Bool_Type_Name | Multibits_Type_Name;
Bool_Type_Name              : 'BOOL';
Multibits_Type_Name         : 'BYTE' | 'WORD' | 'DWORD' | 'LWORD';
```

// Table 11 - Declaration of user-defined data types and initialization

```
Derived_Type_Access         : Single_Elem_Type_Access | Array_Type_Access | Struct_Type_Access
                            | String_Type_Access | Class_Type_Access | Ref_Type_Access | Interface_Type_Access;
String_Type_Access          : ( Namespace_Name '.' )* String_Type_Name;
Single_Elem_Type_Access     : Simple_Type_Access | Subrange_Type_Access | Enum_Type_Access;
Simple_Type_Access          : ( Namespace_Name '.' )* Simple_Type_Name;
Subrange_Type_Access        : ( Namespace_Name '.' )* Subrange_Type_Name;
Enum_Type_Access            : ( Namespace_Name '.' )* Enum_Type_Name;
Array_Type_Access           : ( Namespace_Name '.' )* Array_Type_Name;
Struct_Type_Access          : ( Namespace_Name '.' )* Struct_Type_Name;
Simple_Type_Name            : Identifier;
Subrange_Type_Name          : Identifier;
Enum_Type_Name              : Identifier;
Array_Type_Name             : Identifier;
Struct_Type_Name            : Identifier;

Data_Type_Decl              : 'TYPE' ( Type_Decl ';' )+ 'END_TYPE';
Type_Decl                   : Simple_Type_Decl | Subrange_Type_Decl | Enum_Type_Decl
                            | Array_Type_Decl | Struct_Type_Decl
                            | Str_Type_Decl | Ref_Type_Decl;
Simple_Type_Decl            : Simple_Type_Name ':' Simple_Spec_Init;
Simple_Spec_Init            : Simple_Spec ( ':=' Constant_Expr )?;
Simple_Spec                 : Elem_Type_Name | Simple_Type_Access;
Subrange_Type_Decl          : Subrange_Type_Name ':' Subrange_Spec_Init;
Subrange_Spec_Init          : Subrange_Spec ( ':=' Signed_Int )?;
Subrange_Spec               : Int_Type_Name '(' Subrange ')' | Subrange_Type_Access;
Subrange                    : Constant_Expr '..' Constant_Expr;
Enum_Type_Decl              : Enum_Type_Name ':' ( ( Elem_Type_Name ? Named_Spec_Init ) | Enum_Spec_Init );
Named_Spec_Init             : '(' Enum_Value_Spec ( ',' Enum_Value_Spec )* ')' ( ':=' Enum_Value )?;
Enum_Spec_Init              : ( ( '(' Identifier ( ',' Identifier )* ')' ) | Enum_Type_Access ) ( ':=' Enum_Value )?;
Enum_Value_Spec             : Identifier ( ':=' ( Int_Literal | Constant_Expr ) )?;
Enum_Value                  : ( Enum_Type_Name '#' )? Identifier;
Array_Type_Decl             : Array_Type_Name ':' Array_Spec_Init;
Array_Spec_Init             : Array_Spec ( ':=' Array_Init )?;
Array_Spec                  : Array_Type_Access | 'ARRAY' '[' Subrange ( ',' Subrange )* ']' 'OF' Data_Type_Access;
Array_Init                  : '[' Array_Elem_Init ( ',' Array_Elem_Init )* ']';
Array_Elem_Init             : Array_Elem_Init_Value | Unsigned_Int '(' Array_Elem_Init_Value ? ')';
Array_Elem_Init_Value       : Constant_Expr | Enum_Value | Struct_Init | Array_Init;
Struct_Type_Decl            : Struct_Type_Name ':' Struct_Spec;
Struct_Spec                 : Struct_Decl | Struct_Spec_Init;
Struct_Spec_Init            : Struct_Type_Access ( ':=' Struct_Init )?;
Struct_Decl                 :'STRUCT' 'OVERLAP' ? ( Struct_Elem_Decl ';' )+ 'END_STRUCT';
Struct_Elem_Decl            : Struct_Elem_Name ( Located_At Multibit_Part_Access ? )? ':'
                            ( Simple_Spec_Init | Subrange_Spec_Init | Enum_Spec_Init | Array_Spec_Init
                            | Struct_Spec_Init );
Struct_Elem_Name            : Identifier;
Struct_Init                 : '(' Struct_Elem_Init ( ',' Struct_Elem_Init )* ')';
Struct_Elem_Init            : Struct_Elem_Name ':=' ( Constant_Expr | Enum_Value | Array_Init | Struct_Init | Ref_Value );
Str_Type_Decl               : String_Type_Name ':' String_Type_Name ( ':=' Char_Str )?;
```

// Table 16 - Directly represented variables

```
Direct_Variable             : '%' ( 'I' | 'Q' | 'M' ) ( 'X' | 'B' | 'W' | 'D' | 'L' )? Unsigned_Int ( '.' Unsigned_Int )*;
```

// Table 12 - Reference operations

```
Ref_Type_Decl               : Ref_Type_Name ':' Ref_Spec_Init;
Ref_Spec_Init               : Ref_Spec ( ':=' Ref_Value )?;
Ref_Spec                    : 'REF_TO' + Data_Type_Access;
Ref_Type_Name               : Identifier;
Ref_Type_Access             : ( Namespace_Name '.' )* Ref_Type_Name;
Ref_Name                    : Identifier;
Ref_Value                   : Ref_Addr | 'NULL';
Ref_Addr                    : 'REF' '(' ( Symbolic_Variable | FB_Instance_Name | Class_Instance_Name ) ')';
Ref_Assign                  : Ref_Name ':=' ( Ref_Name | Ref_Deref | Ref_Value );
Ref_Deref                   : Ref_Name '^' +;
```

// Table 13 - Declaration of variables/Table 14 – Initialization of variables

```
Variable                    : Direct_Variable | Symbolic_Variable;
```

```
Symbolic_Variable         : ( ( 'THIS' '.' ) | ( Namespace_Name '.' )+ )? ( Var_Access | Multi_Elem_Var );
Var_Access                : Variable_Name | Ref_Deref;
Variable_Name             : Identifier;
Multi_Elem_Var            : Var_Access ( Subscript_List | Struct_Variable )+;
Subscript_List            : '[' Subscript ( ',' Subscript )* ']';
Subscript                 : Expression;
Struct_Variable           : '.' Struct_Elem_Select;
Struct_Elem_Select        : Var_Access;
Input_Decls               : 'VAR_INPUT' ( 'RETAIN' | 'NON_RETAIN' )? ( Input_Decl ';' )* 'END_VAR';
Input_Decl                : Var_Decl_Init | Edge_Decl | Array_Conform_Decl;
Edge_Decl                 : Variable_List ':' 'BOOL' ( 'R_EDGE' | 'F_EDGE' );
Var_Decl_Init             : Variable_List ':' ( Simple_Spec_Init | Str_Var_Decl | Ref_Spec_Init )
                          | Array_Var_Decl_Init | Struct_Var_Decl_Init | FB_Decl_Init | Interface_Spec_Init;
Ref_Var_Decl              : Variable_List ':' Ref_Spec;
Interface_Var_Decl        : Variable_List ':' Interface_Type_Access;
Variable_List             : Variable_Name ( ',' Variable_Name )*;
Array_Var_Decl_Init       : Variable_List ':' Array_Spec_Init;
Array_Conformand          : 'ARRAY' '[' '*' ( ',' '*' )* ']' 'OF' Data_Type_Access;
Array_Conform_Decl        : Variable_List ':' Array_Conformand;
Struct_Var_Decl_Init      : Variable_List ':' Struct_Spec_Init;
FB_Decl_No_Init           : FB_Name ( ',' FB_Name )* ':' FB_Type_Access;
FB_Decl_Init              : FB_Decl_No_Init ( ':=' Struct_Init )?;
FB_Name                   : Identifier;
FB_Instance_Name          : ( Namespace_Name '.' )* FB_Name '^' *;
Output_Decls              : 'VAR_OUTPUT' ( 'RETAIN' | 'NON_RETAIN' )? ( Output_Decl ';' )* 'END_VAR';
Output_Decl               : Var_Decl_Init | Array_Conform_Decl;
In_Out_Decls              : 'VAR_IN_OUT' ( In_Out_Var_Decl ';' )* 'END_VAR';
In_Out_Var_Decl           : Var_Decl | Array_Conform_Decl | FB_Decl_No_Init;
Var_Decl                  : Variable_List ':' ( Simple_Spec | Str_Var_Decl | Array_Var_Decl | Struct_Var_Decl );
Array_Var_Decl            : Variable_List ':' Array_Spec;
Struct_Var_Decl           : Variable_List ':' Struct_Type_Access;
Var_Decls                 : 'VAR' 'CONSTANT' ? Access_Spec ? ( Var_Decl_Init ';' )* 'END_VAR';
Retain_Var_Decls          : 'VAR' 'RETAIN' Access_Spec ? ( Var_Decl_Init ';' )* 'END_VAR';
Loc_Var_Decls             : 'VAR' ( 'CONSTANT' | 'RETAIN' | 'NON_RETAIN' )? ( Loc_Var_Decl ';' )* 'END_VAR';
Loc_Var_Decl              : Variable_Name ? Located_At ':' Loc_Var_Spec_Init;
Temp_Var_Decls            : VAR_TEMP' ( ( Var_Decl | Ref_Var_Decl | Interface_Var_Decl ) ';' )* 'END_VAR';
External_Var_Decls        : 'VAR_EXTERNAL' 'CONSTANT' ? ( External_Decl ';' )* 'END_VAR';
External_Decl             : Global_Var_Name ':'
                          ( Simple_Spec | Array_Spec | Struct_Type_Access | FB_Type_Access | Ref_Type_Access );
Global_Var_Name           : Identifier;
Global_Var_Decls          : 'VAR_GLOBAL' ( 'CONSTANT' | 'RETAIN' )? ( Global_Var_Decl ';' )* 'END_VAR';
Global_Var_Decl           : Global_Var_Spec ':' ( Loc_Var_Spec_Init | FB_Type_Access );
Global_Var_Spec           : ( Global_Var_Name ( ',' Global_Var_Name )* ) | ( Global_Var_Name Located_At );
Loc_Var_Spec_Init         : Simple_Spec_Init | Array_Spec_Init | Struct_Spec_Init | S_Byte_Str_Spec | D_Byte_Str_Spec;
Located_At                : 'AT' Direct_Variable;
Str_Var_Decl              : S_Byte_Str_Var_Decl | D_Byte_Str_Var_Decl;
S_Byte_Str_Var_Decl       : Variable_List ':' S_Byte_Str_Spec;
S_Byte_Str_Spec           : 'STRING' ( '[' Unsigned_Int ']' )? ( ':=' S_Byte_Char_Str )?;
D_Byte_Str_Var_Decl       : Variable_List ':' D_Byte_Str_Spec;
D_Byte_Str_Spec           : 'WSTRING' ( '[' Unsigned_Int ']' )? ( ':=' D_Byte_Char_Str )?;
Loc_Partly_Var_Decl       : 'VAR' ( 'RETAIN' | 'NON_RETAIN' )? Loc_Partly_Var * 'END_VAR';
Loc_Partly_Var            : Variable_Name 'AT' '%' ( 'I' | 'Q' | 'M' ) '*' ':' Var_Spec ';';
Var_Spec                  : Simple_Spec | Array_Spec | Struct_Type_Access
                          | ( 'STRING' | 'WSTRING' ) ( '[' Unsigned_Int ']' )?;

// Table 19 - Function declaration
Func_Name                 : Std_Func_Name | Derived_Func_Name;
Func_Access               : ( Namespace_Name '.' )* Func_Name;
Std_Func_Name             : 'TRUNC' | 'ABS' | 'SQRT' | 'LN' | 'LOG' | 'EXP'
                          | 'SIN' | 'COS' | 'TAN' | 'ASIN' | 'ACOS' | 'ATAN' | 'ATAN2 '
                          | 'ADD' | 'SUB' | 'MUL' | 'DIV' | 'MOD' | 'EXPT' | 'MOVE '
                          | 'SHL' | 'SHR' | 'ROL' | 'ROR'
                          | 'AND' | 'OR' | 'XOR' | 'NOT'
                          | 'SEL' | 'MAX' | 'MIN' | 'LIMIT' | 'MUX '
                          | 'GT' | 'GE' | 'EQ' | 'LE' | 'LT' | 'NE'
                          | 'LEN' | 'LEFT' | 'RIGHT' | 'MID' | 'CONCAT' | 'INSERT' | 'DELETE' | 'REPLACE' | 'FIND';
                                  // incomplete list
Derived_Func_Name         : Identifier;
Func_Decl                 : 'FUNCTION' Derived_Func_Name ( ':' Data_Type_Access )? Using_Directive *
                          ( IO_Var_Decls | Func_Var_Decls | Temp_Var_Decls )* Func_Body 'END_FUNCTION';
IO_Var_Decls              : Input_Decls | Output_Decls | In_Out_Decls;
Func_Var_Decls            : External_Var_Decls | Var_Decls;
Func_Body                 : Ladder_Diagram | FB_Diagram | Instruction_List | Stmt_List | Other_Languages;
```

// Table 40 – Function block type declaration
// Table 41 - Function block instance declaration

```
FB_Type_Name               : Std_FB_Name | Derived_FB_Name;
FB_Type_Access             : ( Namespace_Name '.' )* FB_Type_Name;
Std_FB_Name                : 'SR' | 'RS' | 'R_TRIG' | 'F_TRIG' | 'CTU'| 'CTD' | 'CTUD' | 'TP' | 'TON' | 'TOF';
                                    // incomplete list
Derived_FB_Name            : Identifier;
FB_Decl                    : 'FUNCTION_BLOCK' ( 'FINAL' | 'ABSTRACT' )? Derived_FB_Name Using_Directive *
                             ( 'EXTENDS' ( FB_Type_Access | Class_Type_Access ) )?
                             ( 'IMPLEMENTS' Interface_Name_List )?
                             ( FB_IO_Var_Decls | Func_Var_Decls | Temp_Var_Decls | Other_Var_Decls )*
                             ( Method_Decl )* FB_Body ? 'END_FUNCTION_BLOCK';
FB_IO_Var_Decls            : FB_Input_Decls | FB_Output_Decls | In_Out_Decls;
FB_Input_Decls             : 'VAR_INPUT' ( 'RETAIN' | 'NON_RETAIN' )? ( FB_Input_Decl ';' )* 'END_VAR';
FB_Input_Decl              : Var_Decl_Init | Edge_Decl | Array_Conform_Decl;
FB_Output_Decls            : 'VAR_OUTPUT' ( 'RETAIN' | 'NON_RETAIN' )? ( FB_Output_Decl ';' )* 'END_VAR';
FB_Output_Decl             : Var_Decl_Init | Array_Conform_Decl;
Other_Var_Decls            : Retain_Var_Decls | No_Retain_Var_Decls | Loc_Partly_Var_Decl;
No_Retain_Var_Decls        : 'VAR' 'NON_RETAIN' Access_Spec ? ( Var_Decl_Init ';' )* 'END_VAR';
FB_Body                    : SFC | Ladder_Diagram | FB_Diagram | Instruction_List | Stmt_List | Other_Languages;
Method_Decl                : 'METHOD' Access_Spec ( 'FINAL' | 'ABSTRACT' )? 'OVERRIDE' ?
                              Method_Name ( ':' Data_Type_Access )?
                             ( IO_Var_Decls | Func_Var_Decls | Temp_Var_Decls )* Func_Body 'END_METHOD';
Method_Name                : Identifier;
```

// Table 48 - Class
// Table 50 Textual call of methods – Formal and non-formal parameter list

```
Class_Decl                 : 'CLASS' ( 'FINAL' | 'ABSTRACT' )? Class_Type_Name Using_Directive *
                             ( 'EXTENDS' Class_Type_Access )? ( 'IMPLEMENTS' Interface_Name_List )?
                             ( Func_Var_Decls | Other_Var_Decls )* ( Method_Decl )* 'END_CLASS';
Class_Type_Name            : Identifier;
Class_Type_Access          : ( Namespace_Name '.' )* Class_Type_Name;
Class_Name                 : Identifier;
Class_Instance_Name        : ( Namespace_Name '.' )* Class_Name '^' *;
Interface_Decl             : 'INTERFACE' Interface_Type_Name Using_Directive *
                             ( 'EXTENDS' Interface_Name_List )? Method_Prototype * 'END_INTERFACE';
Method_Prototype           : 'METHOD' Method_Name ( ':' Data_Type_Access )? IO_Var_Decls * 'END_METHOD';
Interface_Spec_Init        : Variable_List ( ':=' Interface_Value )?;
Interface_Value            : Symbolic_Variable | FB_Instance_Name | Class_Instance_Name | 'NULL';
Interface_Name_List        : Interface_Type_Access ( ',' Interface_Type_Access )*;
Interface_Type_Name        : Identifier;
Interface_Type_Access      : ( Namespace_Name '.' )* Interface_Type_Name;
Interface_Name             : Identifier;
Access_Spec                : 'PUBLIC' | 'PROTECTED' | 'PRIVATE' | 'INTERNAL';
```

// Table 47 - Program declaration

```
Prog_Decl                  : 'PROGRAM' Prog_Type_Name
                             ( IO_Var_Decls | Func_Var_Decls | Temp_Var_Decls | Other_Var_Decls
                             | Loc_Var_Decls | Prog_Access_Decls )* FB_Body 'END_PROGRAM';
Prog_Type_Name             : Identifier;
Prog_Type_Access           : ( Namespace_Name '.' )* Prog_Type_Name;
Prog_Access_Decls          : 'VAR_ACCESS' ( Prog_Access_Decl ';' )* 'END_VAR';
Prog_Access_Decl           : Access_Name ':' Symbolic_Variable Multibit_Part_Access ?
                              ':' Data_Type_Access Access_Direction ?;
```

// Table 54 - 61 - Sequential Function Chart (SFC)

```
SFC                        : Sfc_Network +;
Sfc_Network                : Initial_Step ( Step | Transition | Action )*;
Initial_Step               : 'INITIAL_STEP' Step_Name ':' ( Action_Association ';' )* 'END_STEP';
Step                       : 'STEP' Step_Name ':' ( Action_Association ';' )* 'END_STEP';
Step_Name                  : Identifier;
Action_Association         : Action_Name '(' Action_Qualifier ? ( ',' Indicator_Name )* ')';
Action_Name                : Identifier;
Action_Qualifier           : 'N' | 'R' | 'S' | 'P' | ( ( 'L' | 'D' | 'SD' | 'DS' | 'SL' ) ',' Action_Time );
Action_Time                : Duration | Variable_Name;
Indicator_Name             : Variable_Name;
Transition                 : 'TRANSITION' Transition_Name ? ( '(' 'PRIORITY' ':=' Unsigned_Int ')' )?
                             'FROM' Steps 'TO' Steps ':' Transition_Cond  'END_TRANSITION';
Transition_Name            : Identifier;
Steps                      : Step_Name | '(' Step_Name ( ',' Step_Name )+ ')';
Transition_Cond            : ':=' Expression ';' | ':' ( FBD_Network | LD_Rung ) | ':=' IL_Simple_Inst;
Action                     : 'ACTION' Action_Name ':' FB_Body 'END_ACTION';
```

// Table 62 - Configuration and resource declaration

```
Config_Name              : Identifier;
Resource_Type_Name       : Identifier;
Config_Decl              : 'CONFIGURATION' Config_Name Global_Var_Decls ?
                          ( Single_Resource_Decl | Resource_Decl + ) Access_Decls ? Config_Init ?
                          'END_CONFIGURATION';
Resource_Decl            : 'RESOURCE' Resource_Name 'ON' Resource_Type_Name
                          Global_Var_Decls ? Single_Resource_Decl
                          'END_RESOURCE';
Single_Resource_Decl     : ( Task_Config ';' )* ( Prog_Config ';' )+;
Resource_Name            : Identifier;
Access_Decls             : 'VAR_ACCESS' ( Access_Decl ';' )* 'END_VAR';
Access_Decl              : Access_Name ':' Access_Path ':' Data_Type_Access Access_Direction ?;
Access_Path              : ( Resource_Name '.' )? Direct_Variable
                         | ( Resource_Name '.' )? ( Prog_Name '.' )?
                         ( ( FB_Instance_Name | Class_Instance_Name ) '.' )* Symbolic_Variable;
Global_Var_Access        : ( Resource_Name '.' )? Global_Var_Name ( '.' Struct_Elem_Name )?;
Access_Name              : Identifier;
Prog_Output_Access       : Prog_Name '.' Symbolic_Variable;
Prog_Name                : Identifier;
Access_Direction         : 'READ_WRITE' | 'READ_ONLY';
Task_Config              : 'TASK' Task_Name Task_Init;
Task_Name                : Identifier;
Task_Init                : '(' ( 'SINGLE' ':=' Data_Source ',' )?
                         ( 'INTERVAL' ':=' Data_Source ',' )?
                         'PRIORITY' ':=' Unsigned_Int ')';
Data_Source              : Constant | Global_Var_Access | Prog_Output_Access | Direct_Variable;
Prog_Config              : 'PROGRAM' ( 'RETAIN' | 'NON_RETAIN' )? Prog_Name ( 'WITH' Task_Name )? ':'
                          Prog_Type_Access ( '(' Prog_Conf_Elems ')' )?;
Prog_Conf_Elems          : Prog_Conf_Elem ( ',' Prog_Conf_Elem )*;
Prog_Conf_Elem           : FB_Task | Prog_Cnxn;
FB_Task                  : FB_Instance_Name 'WITH' Task_Name;
Prog_Cnxn                : Symbolic_Variable ':=' Prog_Data_Source | Symbolic_Variable '=>' Data_Sink;
Prog_Data_Source         : Constant | Enum_Value | Global_Var_Access | Direct_Variable;
Data_Sink                : Global_Var_Access | Direct_Variable;
Config_Init              : 'VAR_CONFIG' ( Config_Inst_Init ';' )* 'END_VAR';
Config_Inst_Init         : Resource_Name '.' Prog_Name '.' ( ( FB_Instance_Name | Class_Instance_Name ) '.' )*
                         ( Variable_Name Located_At ? ':' Loc_Var_Spec_Init
                         | ( ( FB_Instance_Name ':' FB_Type_Access )
                         | ( Class_Instance_Name ':' Class_Type_Access ) ) ':=' Struct_Init );
```

// Table 64 - Namespace

```
Namespace_Decl           : 'NAMESPACE' 'INTERNAL' ? Namespace_H_Name Using_Directive * Namespace_Elements
                          'END_NAMESPACE';
Namespace_Elements       : ( Data_Type_Decl | Func_Decl | FB_Decl
                         | Class_Decl | Interface_Decl | Namespace_Decl )+;
Namespace_H_Name         : Namespace_Name ( '.' Namespace_Name )*;
Namespace_Name           : Identifier;
Using_Directive          : 'USING' Namespace_H_Name ( ',' Namespace_H_Name )* ';';
POU_Decl                 : Using_Directive *
                         ( Global_Var_Decls | Data_Type_Decl | Access_Decls
                         | Func_Decl | FB_Decl | Class_Decl | Interface_Decl
                         | Namespace_Decl )+;
```

// Table 67 - 70 - Instruction List (IL)

```
Instruction_List         : IL_Instruction +;
IL_Instruction           : ( IL_Label ':' )? ( IL_Simple_Operation | IL_Expr | IL_Jump_Operation
                         | IL_Invocation | IL_Formal_Func_Call
                         | IL_Return_Operator )? EOL +;
IL_Simple_Inst           : IL_Simple_Operation | IL_Expr | IL_Formal_Func_Call;
IL_Label                 : Identifier;
IL_Simple_Operation      : IL_Simple_Operator IL_Operand ? | Func_Access IL_Operand_List ?;
IL_Expr                  : IL_Expr_Operator '(' IL_Operand ? EOL + IL_Simple_Inst_List ? ')';
IL_Jump_Operation        : IL_Jump_Operator IL_Label;
IL_Invocation            : IL_Call_Operator ((( FB_Instance_Name | Func_Name | Method_Name | 'THIS '
                         | ( ( 'THIS' '.' ( ( FB_Instance_Name | Class_Instance_Name ) '.' )* ) Method_Name ) )
                         ( '(' ( ( EOL + IL_Param_List ? ) | IL_Operand_List ? ) ')' )? ) | 'SUPER' '(' ')' );
IL_Formal_Func_Call      : Func_Access '(' EOL + IL_Param_List ? ')';
IL_Operand               : Constant | Enum_Value | Variable_Access;
IL_Operand_List          : IL_Operand ( ',' IL_Operand )*;
IL_Simple_Inst_List      : IL_Simple_Instruction +;
IL_Simple_Instruction    : ( IL_Simple_Operation | IL_Expr | IL_Formal_Func_Call ) EOL +;
IL_Param_List            : IL_Param_Inst * IL_Param_Last_Inst;
IL_Param_Inst            : ( IL_Param_Assign | IL_Param_Out_Assign ) ',' EOL +;
IL_Param_Last_Inst       : ( IL_Param_Assign | IL_Param_Out_Assign ) EOL +;
IL_Param_Assign          : IL_Assignment ( IL_Operand | ( '(' EOL + IL_Simple_Inst_List ')' ) );
IL_Param_Out_Assign      : IL_Assign_Out_Operator Variable_Access;
```

```
IL_Simple_Operator          : 'LD' | 'LDN' | 'ST' | 'STN' | 'ST?' | 'NOT' | 'S' | 'R'
                            | 'S1' | 'R1' | 'CLK' | 'CU' | 'CD' | 'PV'
                            | 'IN' | 'PT' | IL_Expr_Operator;
IL_Expr_Operator            : 'AND' | '&' | 'OR' | 'XOR' | 'ANDN' | '&N' | 'ORN'
                            | 'XORN' | 'ADD' | 'SUB' | 'MUL' | 'DIV'
                            | 'MOD' | 'GT' | 'GE' | 'EQ' | 'LT' | 'LE' | 'NE';
IL_Assignment               : Variable_Name ':=';
IL_Assign_Out_Operator      : 'NOT' ? Variable_Name '=>';
IL_Call_Operator            : 'CAL' | 'CALC' | 'CALCN';
IL_Return_Operator          : 'RT' | 'RETC' | 'RETCN';
IL_Jump_Operator            : 'JMP' | 'JMPC' | 'JMPCN';
```

// Table 71 - 72 - Language Structured Text (ST)

```
Expression                  : Xor_Expr ( 'OR' Xor_Expr )*;
Constant_Expr               : Expression;
                                    // a constant expression must evaluate to a constant value at compile time
Xor_Expr                    : And_Expr ( 'XOR' And_Expr )*;
And_Expr                    : Compare_Expr ( ( '&' | 'AND' ) Compare_Expr )*;
Compare_Expr                : ( Equ_Expr ( ( '=' | '<>' ) Equ_Expr )* );
Equ_Expr                    : Add_Expr ( ( '<' | '>' | '<=' | '>=' ) Add_Expr )*;
Add_Expr                    : Term ( ( '+' | '-' ) Term )*;
Term                        : Power_Expr ( '*' | '/' | 'MOD' Power_Expr )*;
Power_Expr                  : Unary_Expr ( '**' Unary_Expr )*;
Unary_Expr                  : '-' | '+' | 'NOT' ? Primary_Expr;
Primary_Expr                : Constant | Enum_Value | Variable_Access | Func_Call | Ref_Value| '(' Expression ')';
Variable_Access             : Variable Multibit_Part_Access ?;
Multibit_Part_Access        : '.' ( Unsigned_Int | '%' ( 'X' | 'B' | 'W' | 'D' | 'L' ) ? Unsigned_Int );
Func_Call                   : Func_Access '(' ( Param_Assign ( ',' Param_Assign )* )? ')';
Stmt_List                   : ( Stmt ? ';' )*;
Stmt                        : Assign_Stmt | Subprog_Ctrl_Stmt | Selection_Stmt | Iteration_Stmt;
Assign_Stmt                 : ( Variable ':=' Expression ) | Ref_Assign | Assignment_Attempt;
Assignment_Attempt          : ( Ref_Name | Ref_Deref ) '?=' ( Ref_Name | Ref_Deref | Ref_Value );
Invocation                  : ( FB_Instance_Name | Method_Name | 'THIS'
                             | ( ( 'THIS' '.' )? ( ( ( FB_Instance_Name | Class_Instance_Name ) '.' )+ ) Method_Name ) )
                            '(' ( Param_Assign ( ',' Param_Assign )* )? ')';
Subprog_Ctrl_Stmt           : Func_Call | Invocation | 'SUPER' '(' ')' | 'RETURN';
Param_Assign                : ( ( Variable_Name ':=' )? Expression ) | Ref_Assign | ( 'NOT' ? Variable_Name '=>' Variable );
Selection_Stmt              : IF_Stmt | Case_Stmt;
IF_Stmt                     : 'IF' Expression 'THEN' Stmt_List ( 'ELSIF' Expression 'THEN' Stmt_List )* ( 'ELSE' Stmt_List )?
                             'END_IF';
Case_Stmt                   : 'CASE' Expression 'OF' Case_Selection + ( 'ELSE' Stmt_List )? 'END_CASE';
Case_Selection              : Case_List ':' Stmt_List;
Case_List                   : Case_List_Elem ( ',' Case_List_Elem )*;
Case_List_Elem              : Subrange | Constant_Expr;
Iteration_Stmt              : For_Stmt | While_Stmt | Repeat_Stmt | 'EXIT' | 'CONTINUE';
For_Stmt                    : 'FOR' Control_Variable ':=' For_List 'DO' Stmt_List 'END_FOR';
Control_Variable            : Identifier;
For_List                    : Expression 'TO' Expression ( 'BY' Expression )?;
While_Stmt                  : 'WHILE' Expression 'DO' Stmt_List 'END_WHILE';
Repeat_Stmt                 : 'REPEAT' Stmt_List 'UNTIL' Expression 'END_REPEAT';
```

// Table 73 - 76 - Graphic languages elements

```
Ladder_Diagram              : LD_Rung *;
LD_Rung                     : 'syntax for graphical languages not shown here';
FB_Diagram                  : FBD_Network *;
FBD_Network                 : 'syntax for graphical languages not shown here';
```

```
// Not covered here
Other_Languages             : 'syntax for other languages not shown here';
```

Annex B
(informative)

List of major changes and extensions of the third edition

This standard is fully compatible with IEC 61131-3:2003. The following list shows the major changes and extensions:

Editorial improvements: Structure, numbering, order, wording, examples, feature tables

Terms and definitions like class, method, reference, signature

Compliance table format

New major features

- Data types with explicit layout
- Type with named values
- Elementary data types
- Reference, functions and operations with reference; Validate
- Partial access to `ANY_BIT`
- Variable-length `ARRAY`
- Initial value assignment
- Type conversion rules: Implicit – explicit
- Function – call rules, without function result
- Type conversion functions of numerical, bitwise Data, etc.
- Functions of concatenate and split of time and date
- Class, including method, interface, etc.
- Object-oriented FB, including method, interface, etc.
- Namespaces
- Structured Text: `CONTINUE`, etc.
- Ladder Diagram: Contacts for compare (typed and overloaded)
- ANNEX A - Formal specification of language elements

Deletions (of informative parts)

- ANNEX - Examples
- ANNEX - Interoperability with IEC 61499

Deprecations

- Octal literal
- Use of directly represented variables in the body of POUs and methods
- Overloaded truncation `TRUNC`
- Instruction list (IL)
- “Indicator” variable of action block

Bibliography

IEC 60050 (all parts), *International Electrotechnical Vocabulary* (available at http://www.electropedia.org)

IEC 60848, *GRAFCET specification language for sequential function charts*

IEC 60617, *Graphical symbols for diagrams* (available at http://std.iec.ch/iec60617)

IEC 61499 (all parts), *Function blocks*

ISO/IEC 14977:1996*, Information technology – Syntactic Metalanguage – Extended BNF*

ISO/AFNOR:1989, *Dictionary of computer science*

____

This page deliberately left blank

This page deliberately left blank

British Standards Institution (BSI)

BSI is the national body responsible for preparing British Standards and other standards-related publications, information and services.

BSI is incorporated by Royal Charter. British Standards and other standardization products are published by BSI Standards Limited.

About us

We bring together business, industry, government, consumers, innovators and others to shape their combined experience and expertise into standards -based solutions.

The knowledge embodied in our standards has been carefully assembled in a dependable format and refined through our open consultation process. Organizations of all sizes and across all sectors choose standards to help them achieve their goals.

Information on standards

We can provide you with the knowledge that your organization needs to succeed. Find out more about British Standards by visiting our website at bsigroup.com/standards or contacting our Customer Services team or Knowledge Centre.

Buying standards

You can buy and download PDF versions of BSI publications, including British and adopted European and international standards, through our website at bsigroup.com/shop, where hard copies can also be purchased.

If you need international and foreign standards from other Standards Development Organizations, hard copies can be ordered from our Customer Services team.

Subscriptions

Our range of subscription services are designed to make using standards easier for you. For further information on our subscription products go to bsigroup.com/subscriptions.

With **British Standards Online (BSOL)** you'll have instant access to over 55,000 British and adopted European and international standards from your desktop. It's available 24/7 and is refreshed daily so you'll always be up to date.

You can keep in touch with standards developments and receive substantial discounts on the purchase price of standards, both in single copy and subscription format, by becoming a **BSI Subscribing Member**.

PLUS is an updating service exclusive to BSI Subscribing Members. You will automatically receive the latest hard copy of your standards when they're revised or replaced.

To find out more about becoming a BSI Subscribing Member and the benefits of membership, please visit bsigroup.com/shop.

With a **Multi-User Network Licence (MUNL)** you are able to host standards publications on your intranet. Licences can cover as few or as many users as you wish. With updates supplied as soon as they're available, you can be sure your documentation is current. For further information, email bsmusales@bsigroup.com.

Revisions

Our British Standards and other publications are updated by amendment or revision.

We continually improve the quality of our products and services to benefit your business. If you find an inaccuracy or ambiguity within a British Standard or other BSI publication please inform the Knowledge Centre.

Copyright

All the data, software and documentation set out in all British Standards and other BSI publications are the property of and copyrighted by BSI, or some person or entity that owns copyright in the information used (such as the international standardization bodies) and has formally licensed such information to BSI for commercial publication and use. Except as permitted under the Copyright, Designs and Patents Act 1988 no extract may be reproduced, stored in a retrieval system or transmitted in any form or by any means – electronic, photocopying, recording or otherwise – without prior written permission from BSI. Details and advice can be obtained from the Copyright & Licensing Department.

Useful Contacts:

Customer Services
Tel: +44 845 086 9001
Email (orders): orders@bsigroup.com
Email (enquiries): cservices@bsigroup.com

Subscriptions
Tel: +44 845 086 9001
Email: subscriptions@bsigroup.com

Knowledge Centre
Tel: +44 20 8996 7004
Email: knowledgecentre@bsigroup.com

Copyright & Licensing
Tel: +44 20 8996 7070
Email: copyright@bsigroup.com

BSI Group Headquarters

389 Chiswick High Road London W4 4AL UK

bsi.

...making excellence a habit.™