

INTERNATIONAL STANDARD

NORME INTERNATIONALE

**Programmable controllers –
Part 3: Programming languages**

**Automates programmables –
Partie 3: Langages de programmation**

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013



THIS PUBLICATION IS COPYRIGHT PROTECTED

Copyright © 2013 IEC, Geneva, Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either IEC or IEC's member National Committee in the country of the requester.

If you have any questions about IEC copyright or have an enquiry about obtaining additional rights to this publication, please contact the address below or your local IEC member National Committee for further information.

Droits de reproduction réservés. Sauf indication contraire, aucune partie de cette publication ne peut être reproduite ni utilisée sous quelque forme que ce soit et par aucun procédé, électronique ou mécanique, y compris la photocopie et les microfilms, sans l'accord écrit de la CEI ou du Comité national de la CEI du pays du demandeur.

Si vous avez des questions sur le copyright de la CEI ou si vous désirez obtenir des droits supplémentaires sur cette publication, utilisez les coordonnées ci-après ou contactez le Comité national de la CEI de votre pays de résidence.

IEC Central Office
3, rue de Varembe
CH-1211 Geneva 20
Switzerland

Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00
info@iec.ch
www.iec.ch

About the IEC

The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications

The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

Useful links:

IEC publications search - www.iec.ch/searchpub

The advanced search enables you to find IEC publications by a variety of criteria (reference number, text, technical committee,...).

It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished

Stay up to date on all new IEC publications. Just Published details all new publications released. Available on-line and also once a month by email.

Electropedia - www.electropedia.org

The world's leading online dictionary of electronic and electrical terms containing more than 30 000 terms and definitions in English and French, with equivalent terms in additional languages. Also known as the International Electrotechnical Vocabulary (IEV) on-line.

Customer Service Centre - webstore.iec.ch/csc

If you wish to give us your feedback on this publication or need further assistance, please contact the Customer Service Centre: csc@iec.ch.

A propos de la CEI

La Commission Electrotechnique Internationale (CEI) est la première organisation mondiale qui élabore et publie des Normes internationales pour tout ce qui a trait à l'électricité, à l'électronique et aux technologies apparentées.

A propos des publications CEI

Le contenu technique des publications de la CEI est constamment revu. Veuillez vous assurer que vous possédez l'édition la plus récente, un corrigendum ou amendement peut avoir été publié.

Liens utiles:

Recherche de publications CEI - www.iec.ch/searchpub

La recherche avancée vous permet de trouver des publications CEI en utilisant différents critères (numéro de référence, texte, comité d'études,...).

Elle donne aussi des informations sur les projets et les publications remplacées ou retirées.

Just Published CEI - webstore.iec.ch/justpublished

Restez informé sur les nouvelles publications de la CEI. Just Published détaille les nouvelles publications parues. Disponible en ligne et aussi une fois par mois par email.

Electropedia - www.electropedia.org

Le premier dictionnaire en ligne au monde de termes électroniques et électriques. Il contient plus de 30 000 termes et définitions en anglais et en français, ainsi que les termes équivalents dans les langues additionnelles. Egalement appelé Vocabulaire Electrotechnique International (VEI) en ligne.

Service Clients - webstore.iec.ch/csc

Si vous désirez nous donner des commentaires sur cette publication ou si vous avez des questions contactez-nous: csc@iec.ch.

INTERNATIONAL STANDARD

NORME INTERNATIONALE

**Programmable controllers –
Part 3: Programming languages**

**Automates programmables –
Partie 3: Langages de programmation**

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

COMMISSION
ELECTROTECHNIQUE
INTERNATIONALE

PRICE CODE
CODE PRIX

XH

ICS 25.040; 35.240.50

ISBN 978-2-83220-661-4

**Warning! Make sure that you obtained this publication from an authorized distributor.
Attention! Veuillez vous assurer que vous avez obtenu cette publication via un distributeur agréé.**

CONTENTS

FOREWORD.....	7
1 Scope.....	9
2 Normative references	9
3 Terms and definitions	9
4 Architectural models	18
4.1 Software model	18
4.2 Communication model	19
4.3 Programming model	20
5 Compliance	22
5.1 General	22
5.2 Feature tables	22
5.3 Implementer's compliance statement	22
6 Common elements	24
6.1 Use of printed characters	24
6.1.1 Character set.....	24
6.1.2 Identifiers	24
6.1.3 Keywords	24
6.1.4 Use of white space	25
6.1.5 Comments	25
6.2 Pragma	26
6.3 Literals – External representation of data	26
6.3.1 General	26
6.3.2 Numeric literals and string literals.....	26
6.3.3 Character string literals	28
6.3.4 Duration literal.....	29
6.3.5 Date and time of day literal.....	30
6.4 Data types.....	30
6.4.1 General	30
6.4.2 Elementary data types (BOOL, INT, REAL, STRING, etc.).....	30
6.4.3 Generic data types	33
6.4.4 User-defined data types.....	34
6.5 Variables.....	47
6.5.1 Declaration and initialization of variables	47
6.5.2 Variable sections	49
6.5.3 Variable length ARRAY variables	51
6.5.4 Constant variables.....	53
6.5.5 Directly represented variables (%).....	54
6.5.6 Retentive variables (RETAIN, NON_RETAIN).....	56
6.6 Program organization units (POUs)	58
6.6.1 Common features for POU's	58
6.6.2 Functions.....	70
6.6.3 Function blocks	99
6.6.4 Programs.....	117
6.6.5 Classes	118

6.6.6	Interface	137
6.6.7	Object oriented features for function blocks	146
6.6.8	Polymorphism	152
6.7	Sequential Function Chart (SFC) elements	155
6.7.1	General	155
6.7.2	Steps	155
6.7.3	Transitions	157
6.7.4	Actions	160
6.7.5	Rules of evolution	168
6.8	Configuration elements	176
6.8.1	General	176
6.8.2	Tasks	180
6.9	Namespaces	186
6.9.1	General	186
6.9.2	Declaration	186
6.9.3	Usage	192
6.9.4	Namespace directive USING	192
7	Textual languages	195
7.1	Common elements	195
7.2	Instruction list (IL)	195
7.2.1	General	195
7.2.2	Instructions	195
7.2.3	Operators, modifiers and operands	196
7.2.4	Functions and function blocks	198
7.3	Structured Text (ST)	201
7.3.1	General	201
7.3.2	Expressions	201
7.3.3	Statements	203
8	Graphic languages	208
8.1	Common elements	208
8.1.1	General	208
8.1.2	Representation of variables and instances	209
8.1.3	Representation of lines and blocks	211
8.1.4	Direction of flow in networks	212
8.1.5	Evaluation of networks	213
8.1.6	Execution control elements	214
8.2	Ladder diagram (LD)	215
8.2.1	General	215
8.2.2	Power rails	216
8.2.3	Link elements and states	216
8.2.4	Contacts	216
8.2.5	Coils	218
8.2.6	Functions and function blocks	219
8.2.7	Order of network evaluation	219
8.3	Function Block Diagram (FBD)	219
8.3.1	General	219
8.3.2	Combination of elements	219
8.3.3	Order of network evaluation	220
Annex A (normative)	Formal specification of the languages elements	221

Annex B (informative) List of major changes and extensions of the third edition	228
Bibliography	229

Figure 1 – Software model	18
Figure 2 – Communication model	20
Figure 3 – Combination of programmable controller language elements	21
Figure 4 – Implementer's compliance statement (Example)	23
Figure 5 – Hierarchy of the generic data types	34
Figure 6 – Initialization by literals and constant expressions (Rules)	35
Figure 7 – Variable declaration keywords (Summary)	50
Figure 8 – Usage of <code>VAR_GLOBAL</code> , <code>VAR_EXTERNAL</code> and <code>CONSTANT</code> (Rules)	51
Figure 9 – Conditions for the initial value of a variable (Rules)	57
Figure 10 – Formal and non-formal representation of call (Examples)	63
Figure 11 – Data type conversion rules – implicit and/or explicit (Summary)	67
Figure 12 – Supported implicit type conversions	68
Figure 13 – Usage of function block input and output parameters (Rules)	108
Figure 14 – Usage of function block input and output parameters (Illustration of rules)	109
Figure 15 – Standard timer function blocks – timing diagrams (Rules)	116
Figure 16 – Overview of inheritance and interface implementation	119
Figure 17 – Inheritance of classes (Illustration)	128
Figure 18 – Interface with derived classes (Illustration)	138
Figure 19 – Inheritance of interface and class (Illustration)	143
Figure 20 – Function block with optional body and methods (Illustration)	149
Figure 21 – Inheritance of function block body with <code>SUPER()</code> (Example)	151
Figure 22 – <code>ACTION_CONTROL</code> function block – External interface (Summary)	165
Figure 23 – <code>ACTION_CONTROL</code> function block body (Summary)	166
Figure 24 – Action control (Example)	168
Figure 25 – SFC evolution (Rules)	174
Figure 26 – SFC errors (Example)	175
Figure 27 – Configuration (Example)	177
Figure 28 – <code>CONFIGURATION</code> and <code>RESOURCE</code> declaration (Example)	180
Figure 29 – Accessibility using namespaces (Rules)	189
Figure 30 – Common textual elements (Summary)	195

Table 1 – Character set	24
Table 2 – Identifiers	24
Table 3 – Comments	25
Table 4 – Pragma	26
Table 5 – Numeric literals	27
Table 6 – Character string literals	28
Table 7 – Two-character combinations in character strings	29
Table 8 – Duration literals	29
Table 9 – Date and time of day literals	30

Table 10 – Elementary data types	31
Table 11 – Declaration of user-defined data types and initialization	35
Table 12 – Reference operations	46
Table 13 – Declaration of variables	48
Table 14 – Initialization of variables	49
Table 15 – Variable-length <code>ARRAY</code> variables	52
Table 16 – Directly represented variables	54
Table 17 – Partial access of <code>ANY_BIT</code> variables	60
Table 18 – Execution control graphically using <code>EN</code> and <code>ENO</code>	65
Table 19 – Function declaration	72
Table 20 – Function call	74
Table 21 – Typed and overloaded functions	76
Table 22 – Data type conversion function	78
Table 23 – Data type conversion of numeric data types	80
Table 24 – Data type conversion of bit data types	82
Table 25 – Data type conversion of bit and numeric types	83
Table 26 – Data type conversion of date and time types	85
Table 27 – Data type conversion of character types	86
Table 28 – Numerical and arithmetic functions	87
Table 29 – Arithmetic functions	88
Table 30 – Bit shift functions	89
Table 31 – Bitwise Boolean functions	89
Table 32 – Selection functions ^d	90
Table 33 – Comparison functions	91
Table 34 – Character string functions	92
Table 35 – Numerical functions of time and duration data types	93
Table 36 – Additional functions of time data types <code>CONCAT</code> and <code>SPLIT</code>	94
Table 37 – Function for endianness conversion	98
Table 38 – Functions of enumerated data types	98
Table 39 – Validate functions	99
Table 40 – Function block type declaration	100
Table 41 – Function block instance declaration	104
Table 42 – Function block call	105
Table 43 – Standard bistable function blocks ^a	112
Table 44 – Standard edge detection function blocks	113
Table 45 – Standard counter function blocks	113
Table 46 – Standard timer function blocks	115
Table 47 – Program declaration	117
Table 48 – Class	120
Table 49 – Class instance declaration	122
Table 50 – Textual call of methods – Formal and non-formal parameter list	125
Table 51 – Interface	137
Table 52 – Assignment attempt	146

Table 53 – Object oriented function block	147
Table 54 – SFC step	156
Table 55 – SFC transition and transition condition	158
Table 56 – SFC declaration of actions	160
Table 57 – Step/action association	162
Table 58 – Action block.....	163
Table 59 – Action qualifiers.....	163
Table 60 – Action control features.....	168
Table 61 – Sequence evolution – graphical.....	169
Table 62 – Configuration and resource declaration	178
Table 63 – Task.....	182
Table 64 – Namespace	191
Table 65 – Nested namespace declaration options	192
Table 66 – Namespace directive USING.....	194
Table 67 – Parenthesized expression for IL language	197
Table 68 – Instruction list operators	197
Table 69 – Calls for IL language	199
Table 70 – Standard function block operators for IL language.....	201
Table 71 – Operators of the ST language.....	202
Table 72 – ST language statements.....	203
Table 73 – Graphic execution control elements.....	215
Table 74 – Power rails and link elements.....	216
Table 75 – Contacts.....	217
Table 76 – Coils.....	218

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013

INTERNATIONAL ELECTROTECHNICAL COMMISSION

PROGRAMMABLE CONTROLLERS –**Part 3: Programming languages****FOREWORD**

- 1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
- 2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
- 3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
- 4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
- 5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
- 6) All users should ensure that they have the latest edition of this publication.
- 7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
- 8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
- 9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC 61131-3 has been prepared by subcommittee 65B: Measurement and control devices, of IEC technical committee 65: Industrial-process measurement, control and automation.

This third edition of IEC 61131-3 cancels and replaces the second edition, published in 2003. This edition constitutes a technical revision.

This edition includes the following significant technical changes with respect to the previous edition:

This third edition is a compatible extension of the second edition. The main extensions are new data types and conversion functions, references, name spaces and the object oriented features of classes and function blocks. See Annex B.

The text of this standard is based on the following documents:

FDIS	Report on voting
65B/858/FDIS	65B/863/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

A list of all the parts in the IEC 61131 series, published under the general title *Programmable controllers* can be found on the IEC website.

The committee has decided that the contents of this publication will remain unchanged until the stability date indicated on the IEC web site under "<http://webstore.iec.ch>" in the data related to the specific publication. At this date, the publication will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013

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

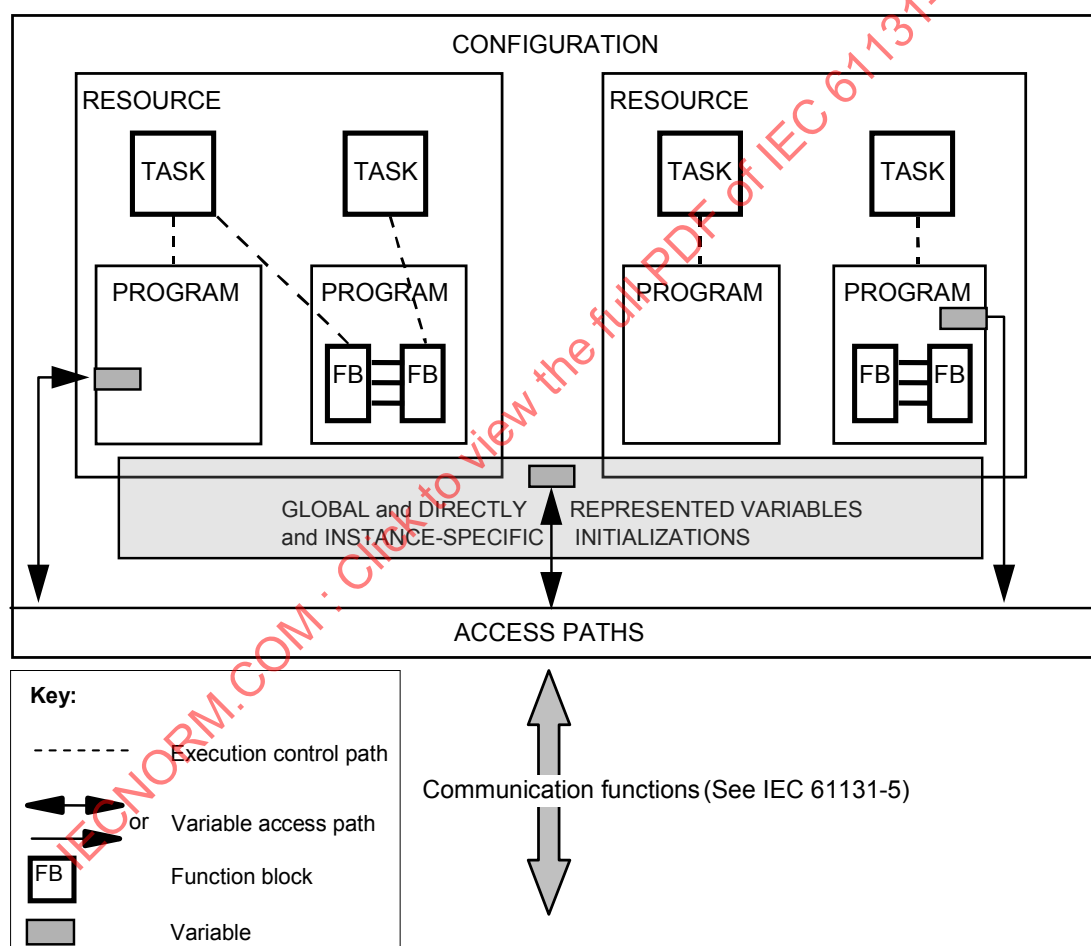
software entity that may take different values, one at a time

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

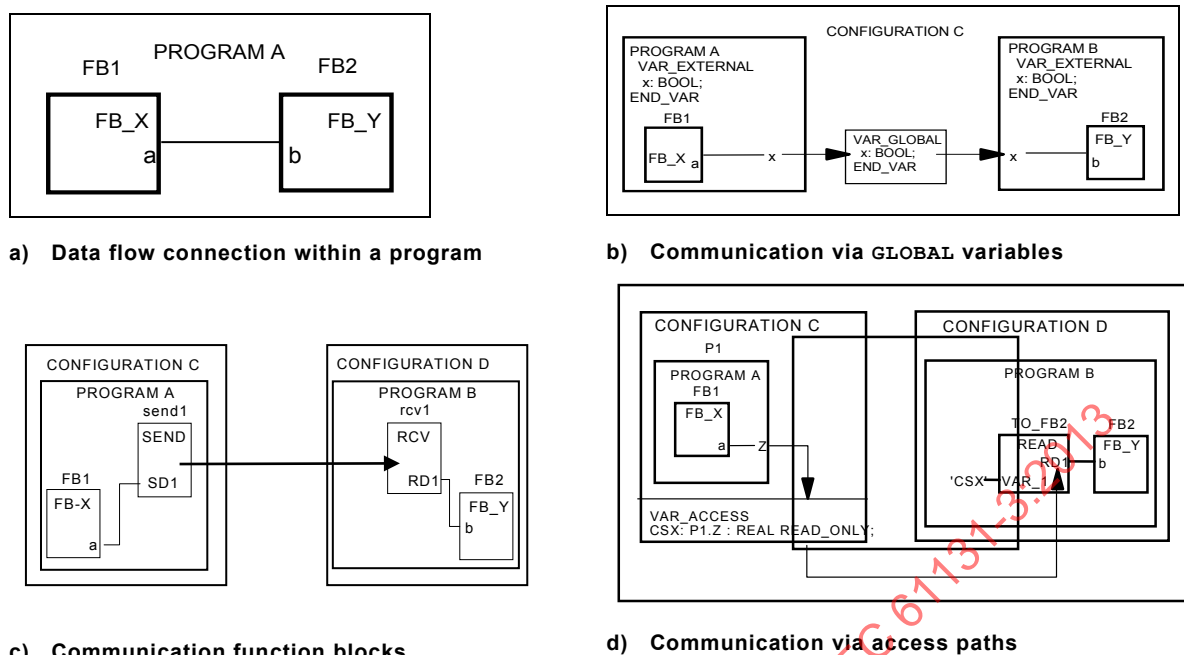
Figure 2 illustrates the ways that values of variables can be communicated among software elements.

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.



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.

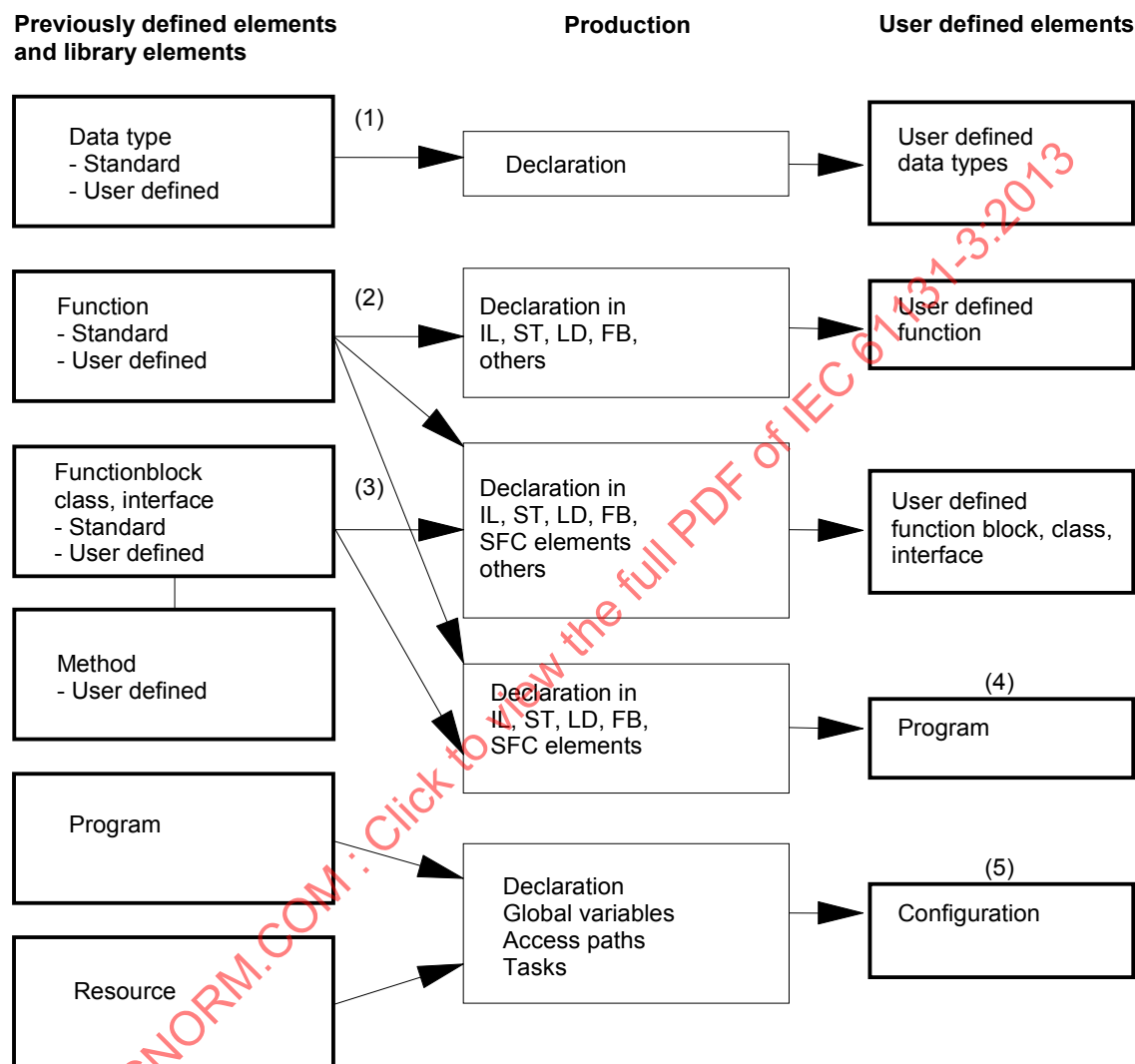
4. A program shall be declared using standard or user-defined data types, functions, function blocks and classes.

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.



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: I W 2 1 5					
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 <code>// ..</code>	<code>X:= 13; // comment for one line</code> <code>// a single line comments can start at</code> <code>// the first character position.</code>
2a	Multi-line comment with <code>(* ... *)</code>	<code>(* comment *)</code> <code>(*****</code> <code> A framed comment on three line</code> <code>*****)</code>
2b	Multi-line comment with <code>/* ... */</code>	<code>/* comment in one</code> <code>or more lines */</code>
3a	Nested comment with <code>(* .. (* .. *) .. *)</code>	<code>(* (* NESTED *) *)</code>
3b	Nested comment with <code>/* .. /* .. */ .. */</code>	<code>/* /* NESTED */ */</code>

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 ' ' (NOTE)	
1a	Empty string (length zero)	' '
1b	String of length one or character <code>CHAR</code> containing a single character	'A'
1c	String of length one or character <code>CHAR</code> containing the "space" character	' '
1d	String of length one or character <code>CHAR</code> containing the "single quote" character	'\$''
1e	String of length one or character <code>CHAR</code> containing the "double quote" character	'"'
1f	Support of two character combinations of Table 7	'\$R\$L'
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 <code>WCHAR</code> containing a single character	"A"
2c	String of length one or character <code>WCHAR</code> containing the "space" character	" "
2d	String of length one or character <code>WCHAR</code> containing the "single quote" character	"' "
2e	String of length one or character <code>WCHAR</code> containing the "double quote" character	"\$"
2f	Support of two character combinations of Table 7	"\$R\$L"
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	h)	REAL, LREAL
	ANY_REAL		
	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	<pre> 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 </pre>	Initialization
2a 2b	Data types with named values	<pre> 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 </pre>	Initialization
3a 3b	Subrange data types	<pre> TYPE ANALOG_DATA: INT(-4095 .. 4095) := 0; END_TYPE </pre>	
4a 4b	Array data types	<pre> TYPE ANALOG_16_INPUT_DATA: ARRAY [1..16] OF ANALOG_DATA := [8(-4095), 8(4095)]; END_TYPE </pre>	ANALOG_DATA see above. Initialization
5a 5b	FB types and classes as array elements	<pre> TYPE TONs: ARRAY[1..50] OF TON := [50(PT:=T#100ms)]; END_TYPE </pre>	FB TON as array element Initialization

No.	Description	Example	Explanation
6a 6b	Structured data type	<pre> TYPE ANALOG_CHANNEL_CONFIGURATION: STRUCT RANGE: ANALOG_SIGNAL_RANGE; MIN_SCALE: ANALOG_DATA:= -4095; MAX_SCALE: ANALOG_DATA:= 4095; END_STRUCT; END_TYPE </pre>	ANALOG_SIGNAL_RANGE see above
7a 7b	FB types and classes as structure elements	<pre> TYPE Cooler: STRUCT Temp: INT; Cooling: TOF:= (PT:=T#100ms); END_TYPE </pre>	FB TOF as structure element
8a 8b	Structured data type with relative addressing AT	<pre> 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 </pre>	Explicit layout without overlapping
9a	Structured data type with relative addressing AT and OVERLAP	<pre> 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 </pre>	Explicit layout with overlapping
10a 10b	Directly represented elements of a structure – partly specified using “*”	<pre> 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 </pre>	Assigns the components of a structure to not yet located inputs and outputs, see NOTE 2
11a 11b	Directly derived data types	<pre> TYPE CNT: UINT; FREQ: REAL:= 50.0; ANALOG_CHANNEL_CONFIG: ANALOG_CHANNEL_CONFIGURATION := (MIN_SCALE:= 0, MAX_SCALE:= 4000); END_TYPE </pre>	Initialization new initialization
12	Initialization using constant expressions	<pre> TYPE PIx2: REAL:= 2 * 3.1416; END_TYPE </pre>	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 data 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 data 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(myS1);
  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

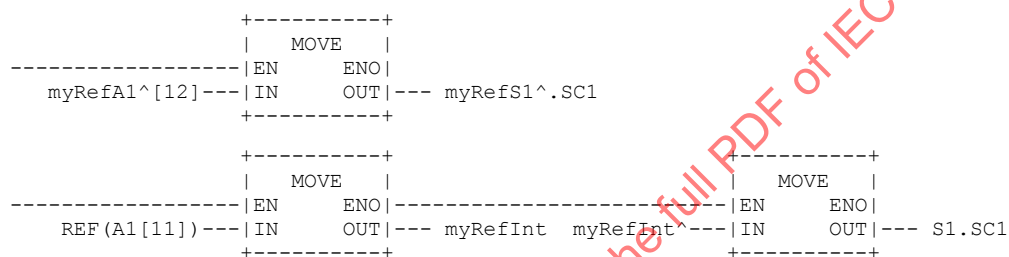


Table 12 defines the features for reference operations.

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	<pre>VAR MYBIT: BOOL; OKAY: STRING[10]; VALVE_POS AT %QW28: INT; END_VAR</pre>	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	<pre>VAR myVAR: myType; END_VAR</pre>	Declaration of a variable with a user data type.
3	Array	<pre>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</pre>	
4	Reference	<pre>VAR myRefInt: REF_TO INT; END_VAR</pre>	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	<pre> VAR MYBIT: BOOL := 1; OKAY: STRING[10] := 'OK'; VALVE_POS AT %QW28: INT:= 100; END_VAR </pre>	Allocates a memory bit to the Boolean variable <code>MYBIT</code> 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	<pre> TYPE myType: ... END_TYPE VAR myVAR: myType:= ... // initialization END_VAR </pre>	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	<pre> 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 </pre>	Allocates 8 memory bits to contain initial values <code>BITS[0] := 1, BITS[1] := 1, ..., BITS[6] := 0, BITS[7] := 0.</code> Allocates a 2-by-3 integer array <code>TBT</code> with initial values <code>TBT[1,1] := 9, TBT[1,2] := 8, TBT[1,3] := 10, TBT[2,1] := 10, TBT[2,2] := 10, TBT[2,3] := 6.</code>
4	Declaration and initialization of constants	<pre> VAR CONSTANT PI: REAL:= 3.141592; PI2: REAL:= 2.0*PI; END_VAR </pre>	constant symbolic constant PI
5	Initialization using constant expressions	<pre> VAR PIx2: REAL:= 2.0 * 3.1416; END_VAR </pre>	Uses a constant expression
6	Initialization of a reference	<pre> VAR myRefInt: REF_TO INT := REF(myINT); END_VAR </pre>	Initializes <code>myRefInt</code> to refer to the variable <code>myINT</code> .

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: <pre> +-----+ ! LOWER_BOUND ! ARRAY ----! ARR !--- ANY_INT ANY_INT --! DIM ! +-----+</pre> Get upper bound of an array: <pre> +-----+ ! UPPER_BOUND ! ARRAY ----! ARR !--- ANY_INT ANY_INT ---! DIM ! +-----+</pre>
2b	Textual representation	Get lower bound of the 2nd dimension of the array A: <pre>low2:= LOWER_BOUND (A, 2);</pre> Get upper bound of the 2nd dimension of the array A: <pre>up2:= UPPER_BOUND (A, 2);</pre>

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 POU 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
  INP_0 AT %I0.0: BOOL;
  AT %IB12: REAL;
  PA_VAR AT %IB200: PA_VALUE;
```

Name and type for an input

```
OUTARY AT %QW6: ARRAY[0..9] OF INT;
END_VAR
```

Name and user-defined type for an input location beginning at %IB200

Array of 10 integers to be allocated to contiguous output 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.

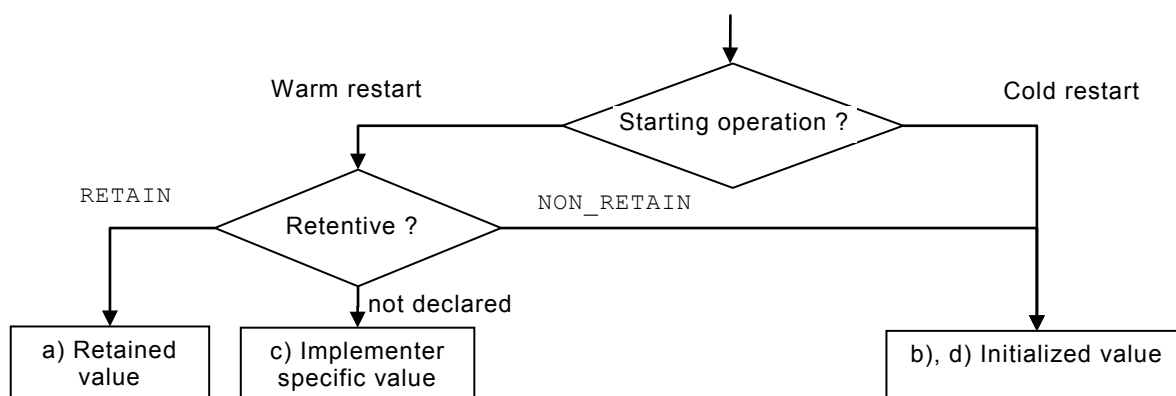


Figure 9 – Conditions for the initial value of a variable (Rules)

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 (`STRUCTURE`, `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.

- 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.
- 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.

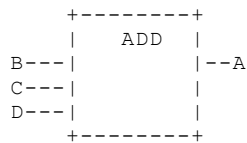
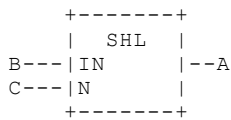
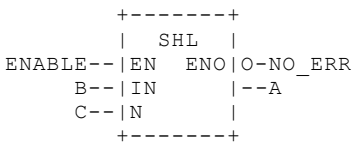
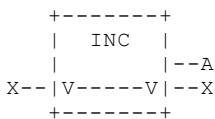
Graphical example (FBD)	Textual example (ST)	Explanation
a) 	<pre>A:= ADD(B,C,D); //Function or</pre> <pre>A:= B + C + D; // Operators</pre>	Non-formal parameter list (B, C, D)
b) 	<pre>A:= SHL(IN:= B, N:= C);</pre>	Formal parameter names IN, N
c) 	<pre>A:= SHL(</pre> <pre> EN:= ENABLE,</pre> <pre> IN:= B,</pre> <pre> N := C,</pre> <pre> NOT ENO => NO_ERR);</pre>	Formal parameter names Use of EN input and negated ENO output
d) 	<pre>A:= INC(V:= X);</pre>	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.

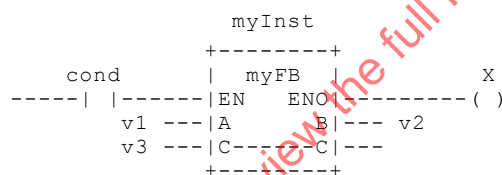
EXAMPLE 2 External implementation

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:



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

No.	Description ^a	Example ^b
1	Usage without EN and ENO	Shown for a function in FBD and ST <pre> +-----+ A--- + ---C B--- +-----+ </pre> C := ADD(IN1:= A, IN2:= B);
2	Usage of EN only (without ENO)	Shown for a function in FBD and ST <pre> +-----+ ADD_EN--- EN A--- + ---C B--- +-----+ </pre> C := ADD(EN:= ADD_EN, IN1:= A, IN2:= B);
3	Usage of ENO only (without EN)	Shown for a function in FBD and ST <pre> +-----+ ENO ---ADD_OK A--- + ---C B--- +-----+ </pre> C := ADD(IN1:= A, IN2:= B, ENO=> ADD_OK);
4	Usage of EN and ENO	Shown for a function in LD and ST <pre> +-----+ ADD_EN + ADD_OK +--- --- EN ENO ---()---+ A--- ---C B--- +-----+ </pre> C := ADD(EN:= ADD_EN, IN1:= a, IN2:= IN2, EN => ADD_OK);
^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.		

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	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		
	REAL	i		e	e	e	e	e	e	e	-	e	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		
integer	LINT	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	-	-	-	-	-	-	-	-	-	-	-	-			
	INT	i	i	i	i		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	-	-	-	-	-	-	-	-	-	-	-	-			
	UDINT	i	e	i	e	e	e	i		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	-	-	-	-	-	-	-	-	-	-	-	-			
	DWORD	-	e	e	e	e	e	e	e	e	i		e	e	-	-	-	-	-	-	-	-	-	-	-	-			
	WORD	-	-	e	e	e	e	e	e	e	i	i		e	-	-	-	-	-	-	-	-	-	-	e	-			
	BYTE	-	-	e	e	e	e	e	e	e	i	i	i		-	-	-	-	-	-	-	-	-	-	-	e			
	BOOL	-	-	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.

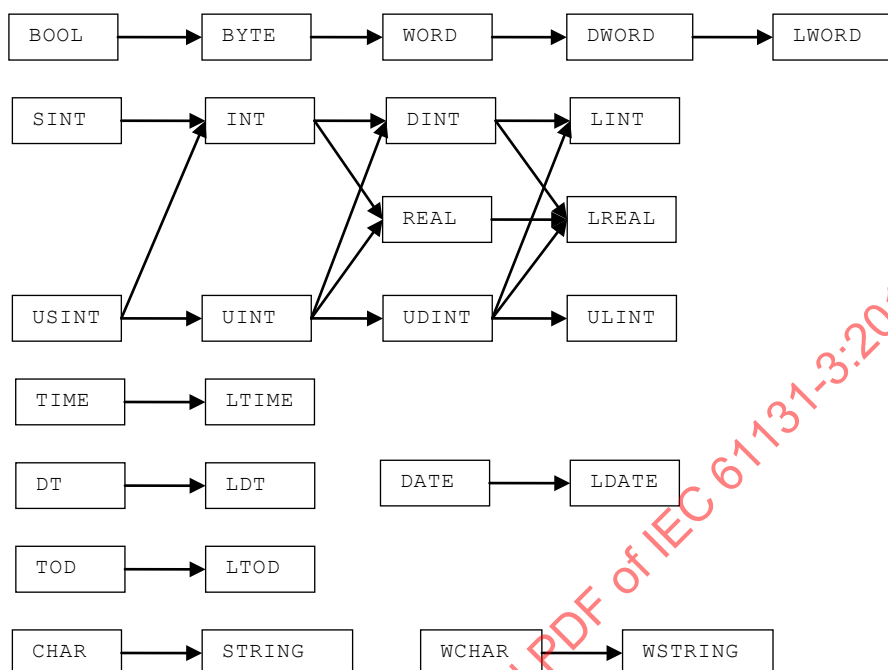


Figure 12 – Supported implicit type conversions

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);
  
```

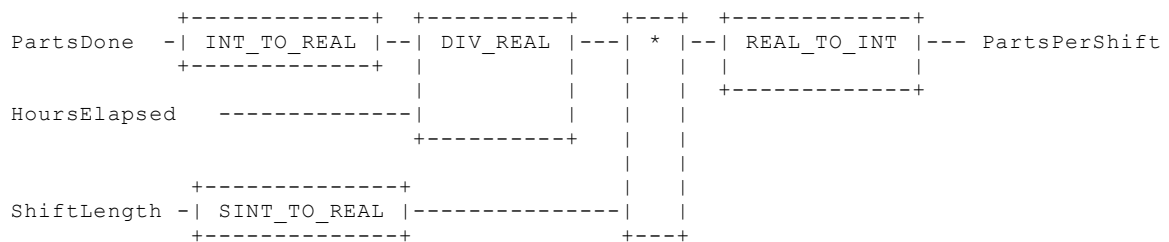
c) Implicit type conversion

```

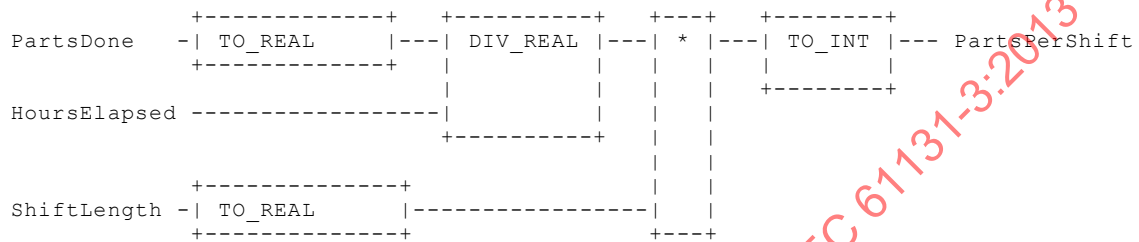
PartsRatePerHr := PartsDone / HoursElapsed;
PartsPerShift := TO_INT(ShiftLength * PartsRatePerHr);
  
```

3) Usage in FBD language

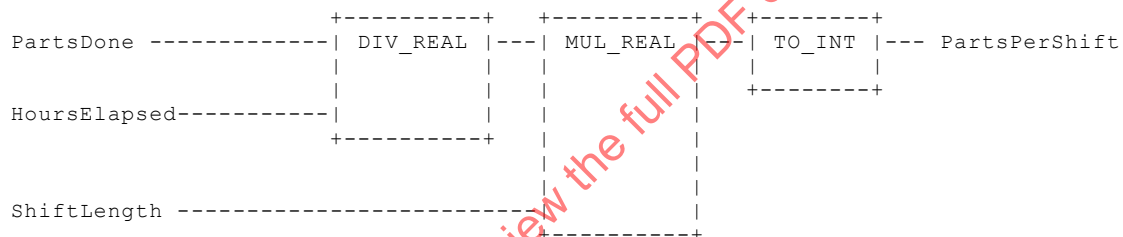
a) Explicit type conversion



b) Explicit overloaded type conversion



c) Implicit type conversion with typed functions



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      // 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

// Function body specification
VAR COUNTPl: INT; END_VAR
COUNTPl:= ADD(COUNT, 1);
COUNT := COUNTPl

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

```
FUNCTION
+-----+
| SIMPLE_FUN |
REAL----|A      |----REAL
REAL----|B      |
REAL----|C      |
INT----|COUNT---COUNT|----INT
+-----+

// Function body specification
+----+
|ADD|---+-----+
COUNT--| |---COUNTPl--|:= |---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

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

VAR I: INT; Sum: INT:= 0; END_VAR
FOR i:= FirstIndex TO LastIndex
DO Sum:= Sum + dataArray[i];
END_FOR

END_FUNCTION
```

```
// External interface

// no function result, but output Sum
+-----+
| SPECIAL_FUN |
INT----|FirstIndex   Sum|----INT
INT----|LastIndex    |
+-----+
```

```
// 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	<pre> +-----+ FUN a -- EN ENO -- b -- IN1 -- result c -- IN2 Q1 --out Q2 +-----+ </pre>
5	Usage of negated boolean input and output in graphical representation	<pre> +-----+ FUN a -o EN ENO -- b -- IN1 -- result c -- IN2 Q1 o- out Q2 +-----+ </pre> NOTE 6 The use of these constructs is forbidden for in-out variables.
6	Graphical usage of VAR_IN_OUT	<pre> +-----+ myFC1 a -- In1 Out1 -- d b -- Inout--Inout -- c +-----+ </pre>

EXAMPLE Function call

Call

VAR

X, Y, Z, Res1, Res2: REAL;

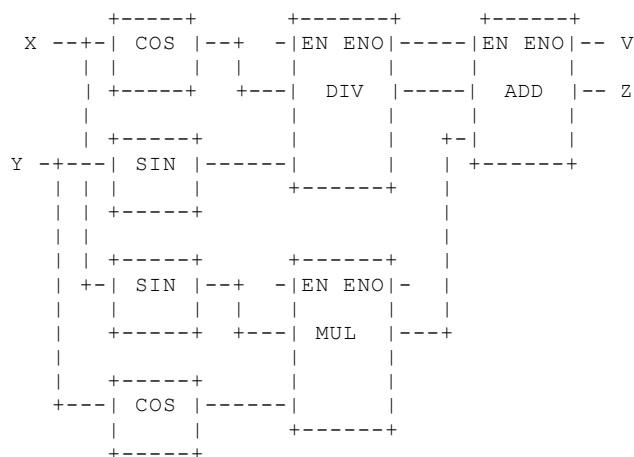
En1, V: BOOL;

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);



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);



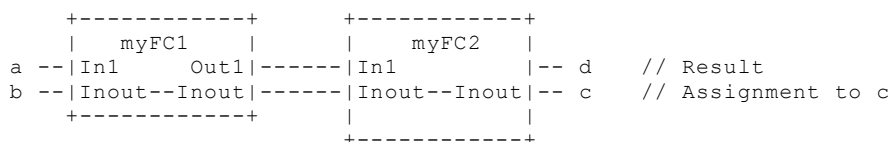
b) Function declaration and call without result but with output variables

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

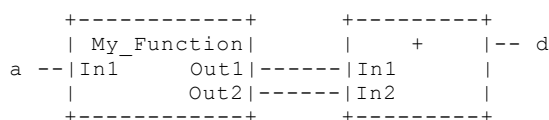
```



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;
```



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)	<pre> +-----+ ADD +-----+ ANY_NUM -- ANY_NUM -- . . ANY_NUM -- +-----+ </pre>
1b	Conversion of inputs ANY_ELEMENT TO_INT	<pre> +-----+ TO_INT ---INT +-----+ </pre>
2a ^a	Typed functions: ADD_INT	<pre> +-----+ ADD_INT +-----+ INT -- INT -- . . INT -- +-----+ </pre>
2b ^a	Conversion: WORD_TO_INT	<pre> +-----+ WORD_TO_INT ---INT +-----+ </pre>
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;
  B: INT;
  C: INT;
END_VAR

      +---+
A --| + |-- C
B --|   |
      +---+

C := A+B;

```

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

```

VAR
  A: INT;
  B: REAL;
  C: REAL;
END_VAR

      +-----+ +---+
A --|INT_TO_REAL|---| + |-- C
      +-----+   |
B -----|   |
      +-----+
C := INT_TO_REAL(A) + B;

      +-----+ +---+
A ---|TO_REAL|---|ADD|---C
      +-----+   |
B -----|   |
      +-----+
C := TO_REAL(A) + B;

VAR
  A: INT;
  B: INT;
  C: REAL;
END_VAR

      +---+ +-----+
A --| + |---|INT_TO_REAL|--- C
B --|   | +-----+
      +---+
C := INT_TO_REAL(A+B);

      +---+ +-----+
A ---|ADD|---|TO_REAL|--- C
B ---|   | +-----+
      +---+
C := 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;
  B: INT;
  C: INT;
END_VAR

      +-----+
A ---| ADD_INT |---C
B ---|         |
      +-----+

C := ADD_INT(A, B);

```

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

Explicit type conversion

```

VAR
  A: INT;
  B: REAL;
  C: REAL;
END_VAR

      +-----+ +-----+
A ---|INT_TO_REAL|---|ADD_REAL|--- C
      +-----+   |
B -----|         |
      +-----+

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

```

Implicit type conversion

```

VAR
  A: INT;
  B: REAL;
  C: REAL;
END_VAR

      +-----+
A -----| ADD_REAL |--- C
B -----|         |
      +-----+

C := ADD_REAL(A, B);

```

Explicit type conversion

```

VAR
  A: INT;
  B: INT;
  C: REAL;
END_VAR

      +-----+ +-----+
A --| ADD_INT |---|INT_TO_REAL|--- C
      |         | +-----+
B --|         |
      +-----+

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

```

Implicit type conversion

```

VAR
  A: INT;
  B: INT;
  C: REAL;
END_VAR

      +-----+
A --| ADD_INT |--- C
      |         |
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 errors that may arise during execution of type conversion operations, are Implementer specific.

Table 22 – Data type conversion function

No.	Description	Graphical form	Usage example
1a	Typed conversion input_TO_output	<pre> B ---+-----+ * TO ** --- A +-----+ (*) - Input data type, e.g., INT (**) - Output data type, e.g., REAL </pre>	<code>A:= INT_TO_REAL(B);</code>
1b ^{a,b,e}	Overloaded conversion TO_output	<pre> B ---+-----+ TO ** --- A +-----+ - Input data type, e.g., INT (**) - Output data type, e.g., REAL </pre>	<code>A:= TO_REAL(B);</code>
2a ^c	"Old" overloaded truncation TRUNC	<pre> ANY_REAL ---+-----+ TRUNC --- ANY_INT +-----+ </pre>	Deprecated
2b ^c	Typed truncation input_TRUNC_output	<pre> ANY_REAL ---+-----+ * TRUNC ** --- ANY_INT +-----+ </pre>	<code>A:= REAL_TRUNC_INT(B);</code>
2c ^c	Overloaded truncation TRUNC_output	<pre> ANY_REAL ---+-----+ TRUNC ** --- ANY_INT +-----+ </pre>	<code>A:= TRUNC_INT(B);</code>
3a ^d	Typed input_BCD_TO_output	<pre> * ---+-----+ * BCD TO ** --- ** +-----+ </pre>	<code>A:=</code> <code>WORD_BCD_TO_INT(B);</code>
3b ^d	Overloaded BCD_TO_output	<pre> * ---+-----+ BCD TO ** --- ** +-----+ </pre>	<code>A:= BCD_TO_INT(B);</code>
4a ^d	Typed input_TO_BCD_output	<pre> ** ---+-----+ ** TO BCD * --- * +-----+ </pre>	<code>A:=</code> <code>INT_TO_BCD_WORD(B);</code>
4b ^d	Overloaded TO_BCD_output	<pre> * ---+-----+ TO BCD ** --- ** +-----+ </pre>	<code>A:= TO_BCD_WORD(B);</code>

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	<code>LREAL _TO_ REAL</code>	Conversion with rounding, value range errors give an Implementer specific result
2	<code>LREAL _TO_ LINT</code>	Conversion with rounding, value range errors give an Implementer specific result
3	<code>LREAL _TO_ DINT</code>	Conversion with rounding, value range errors give an Implementer specific result
4	<code>LREAL _TO_ INT</code>	Conversion with rounding, value range errors give an Implementer specific result
5	<code>LREAL _TO_ SINT</code>	Conversion with rounding, value range errors give an Implementer specific result
6	<code>LREAL _TO_ ULINT</code>	Conversion with rounding, value range errors give an Implementer specific result
7	<code>LREAL _TO_ UDINT</code>	Conversion with rounding, value range errors give an Implementer specific result
8	<code>LREAL _TO_ UINT</code>	Conversion with rounding, value range errors give an Implementer specific result
9	<code>LREAL _TO_ USINT</code>	Conversion with rounding, value range errors give an Implementer specific result
10	<code>REAL _TO_ LREAL</code>	Value preserving conversion
11	<code>REAL _TO_ LINT</code>	Conversion with rounding, value range errors give an Implementer specific result
12	<code>REAL _TO_ DINT</code>	Conversion with rounding, value range errors give an Implementer specific result
13	<code>REAL _TO_ INT</code>	Conversion with rounding, value range errors give an Implementer specific result
14	<code>REAL _TO_ SINT</code>	Conversion with rounding, value range errors give an Implementer specific result
15	<code>REAL _TO_ ULINT</code>	Conversion with rounding, value range errors give an Implementer specific result
16	<code>REAL _TO_ UDINT</code>	Conversion with rounding, value range errors give an Implementer specific result
17	<code>REAL _TO_ UINT</code>	Conversion with rounding, value range errors give an Implementer specific result
18	<code>REAL _TO_ USINT</code>	Conversion with rounding, value range errors give an Implementer specific result
19	<code>LINT _TO_ LREAL</code>	Conversion with potential loss of accuracy
20	<code>LINT _TO_ REAL</code>	Conversion with potential loss of accuracy
21	<code>LINT _TO_ DINT</code>	Value range errors give an Implementer specific result
22	<code>LINT _TO_ INT</code>	Value range errors give an Implementer specific result
23	<code>LINT _TO_ SINT</code>	Value range errors give an Implementer specific result
24	<code>LINT _TO_ ULINT</code>	Value range errors give an Implementer specific result
25	<code>LINT _TO_ UDINT</code>	Value range errors give an Implementer specific result
26	<code>LINT _TO_ UINT</code>	Value range errors give an Implementer specific result
27	<code>LINT _TO_ USINT</code>	Value range errors give an Implementer specific result
28	<code>DINT _TO_ LREAL</code>	Value preserving conversion
29	<code>DINT _TO_ REAL</code>	Conversion with potential loss of accuracy
30	<code>DINT _TO_ LINT</code>	Value preserving conversion
31	<code>DINT _TO_ INT</code>	Value range errors give an Implementer specific result
32	<code>DINT _TO_ SINT</code>	Value range errors give an Implementer specific result
33	<code>DINT _TO_ ULINT</code>	Value range errors give an Implementer specific result
34	<code>DINT _TO_ UDINT</code>	Value range errors give an Implementer specific result
35	<code>DINT _TO_ UINT</code>	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.

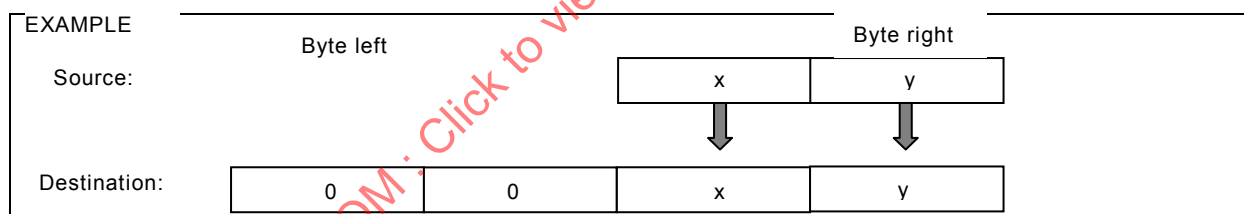


Table 24 describes the conversion functions with conversion details as result of the rules above.

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 <pre> +-----+ * -- ** -- * +-----+ (*) - Input/Output (I/O) type (**) - Function name </pre>		Usage example in ST <pre> A := SIN(B); (ST language) </pre>
	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) <pre> +-----+ ATAN2 ANY_REAL-- Y --ANY_REAL ANY_REAL-- X +-----+ </pre>	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
	<pre> +-----+ ANY_NUM -- *** -- ANY_NUM ANY_NUM -- . -- . -- ANY_NUM -- +-----+ </pre> (***) - 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:= IN1 ^{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: <pre> IF (IN2 = 0) THEN OUT:=0; ELSE OUT:=IN1 - (IN1/IN2)*IN2; END_IF </pre> 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
	<pre> +-----+ *** ANY_BIT -- IN -- ANY_BIT ANY_INT -- N +-----+ (***) - Function Name </pre>		<pre> A:= SHL(IN:=B, N:=5); (ST language) </pre>
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 <pre> 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 </pre> 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			Usage examples (NOTE 5)
	<pre> +-----+ *** ANY_BIT -- IN1 -- ANY_BIT ANY_BIT -- IN2 : : ANY_BIT -- INn +-----+ (***) - Name or symbol </pre>			<pre> A:= AND(B, C, D); or A:= B & C & D; </pre>
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	<pre> +-----+ MOVE ANY -- - ANY +-----+ </pre>	OUT:= IN
2	Binary selection ^d	SEL	<pre> +-----+ SEL BOOL -- G - ANY ANY -- IN0 ANY -- IN1 +-----+ </pre>	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	<pre> +-----+ MAX ANY_ELEMENTARY -- - ANY_ELEMENTARY : -- ANY_ELEMENTARY -- +-----+ </pre>	OUT:= MAX(IN1, IN2, ..., INn); EXAMPLE 2 A:= MAX(B, C, D);
4	Extensible minimum function	MIN	<pre> +-----+ MIN ANY_ELEMENTARY -- - ANY_ELEMENTARY : -- ANY_ELEMENTARY -- +-----+ </pre>	OUT:= MIN (IN1, IN2, ..., Nn) EXAMPLE 3 A:= MIN(B, C, D);
5	Limiter	LIMIT	<pre> +-----+ LIMIT ANY_ELEMENTARY -- MN - ANY_ELEMENTARY ANY_ELEMENTARY -- IN ANY_ELEMENTARY -- MX +-----+ </pre>	OUT:= MIN (MAX(IN, MN), MX); EXAMPLE 4 A:= LIMIT(IN:= B, MN:= 0, MX:= 5);

6	Extensible ^b , c, d, e multiplexer	MUX	<pre> +-----+ MUX ANY_ELEMENTARY -- K -- ANY_ELEMENTARY ANY_ELEMENTARY -- ANY_ELEMENTARY -- +-----+ </pre>	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;
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

No.	Description	Name ^a	Symbol ^b	Explanation (For 2 or more operands extensible)
	Graphical form			Usage examples
	<pre> +-----+ ANY_ELEMENTARY -- *** -- BOOL : ANY_ELEMENTARY -- +-----+ </pre> (***) Name or Symbol			A:= GT(B, C, D); // Function name or A:= (B>C) & (C>D); // Symbol
1	Decreasing sequence	GT	>	OUT:= (IN1>IN2) & (IN2>IN3) &... & (INn-1 > INn)
2	Monotonic sequence:	GE	>=	OUT:= (IN1>=IN2) & (IN2>=IN3) &... & (INn-1 >= INn)
3	Equality	EQ	=	OUT:= (IN1=IN2) & (IN2=IN3) &... & (INn-1 = INn)
4	Monotonic sequence	LE	<=	OUT:= (IN1<=IN2) & (IN2<=IN3) &... & (INn-1 <= INn)
5	Increasing sequence	LT	<	OUT:= (IN1<IN2) & (IN2<IN3) &... & (INn-1 < INn)
6	Inequality	NE	<>	OUT:= (IN1<>IN2) (non-extensible)

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	<pre> +-----+ ANY_STRING -- LEN -- ANY_INT +-----+</pre>	String length <code>A := LEN('ASTRING');</code> ..is equivalent to <code>A := 7;</code>
2	Left	<pre> +-----+ ANY_STRING -- LEN -- ANY_INT +-----+</pre>	Leftmost L characters of IN <code>A := LEFT(IN:='ASTR', L:=3);</code> is equivalent to <code>A := 'AST';</code>
3	Right	<pre> +-----+ RIGHT ANY_STRING -- IN -- ANY_STRING ANY_INT -- L +-----+</pre>	Rightmost L characters of IN <code>A := RIGHT(IN:='ASTR', L:=3);</code> is equivalent to <code>A := 'STR';</code>
4	Middle	<pre> +-----+ MID ANY_STRING -- IN -- ANY_STRING ANY_INT -- L ANY_INT -- P +-----+</pre>	L characters of IN, beginning at the P-th character position <code>A := MID(IN:='ASTR', L:=2, P:=2);</code> is equivalent to <code>A := 'ST';</code>

No.	Description	Graphical form	Example
5	Extensible concatenation	<pre> +-----+ CONCAT ANY_CHARS-- -- ANY_STRING : -- ANY_CHARS-- +-----+ </pre>	Extensible concatenation A:= CONCAT('AB','CD','E'); is equivalent to A:= 'ABCDE';
6	Insert	<pre> +-----+ INSERT ANY_STRING-- IN1 -- ANY_STRING ANY_CHARS -- IN2 ANY_INT---- P +-----+ </pre>	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	<pre> +-----+ DELETE ANY_STRING-- IN -- ANY_STRING ANY_INT -- L ANY_INT -- P +-----+ </pre>	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	<pre> +-----+ REPLACE ANY_STRING-- IN1 -- ANY_STRING ANY_CHARS -- IN2 ANY_INT -- L ANY_INT -- P +-----+ </pre>	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	<pre> +-----+ FIND ANY_STRING-- IN1 -- ANY_INT ANY_CHARS -- IN2 +-----+ </pre>	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_LTOD_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		

No.	Description	Graphical form	Example
1a	CONCAT_DATE_TOD	<pre> +-----+ CONCAT_DATE_TOD DATE -- DATE --DT TOD -- TOD +-----+</pre>	Concatenate a date: <pre> VAR myD: DATE; END_VAR myD:= CONCAT_DATE_TOD (D#2010-03-12, TOD#12:30:00);</pre>
1b	CONCAT_DATE_LTOD	<pre> +-----+ CONCAT_DATE_LTOD DATE -- DATE --LDT LTOD -- LTOD +-----+</pre>	Concatenate a date and a time of day: <pre> VAR myD: DATE; END_VAR myD:= CONCAT_DATE_LTOD (D#2010-03-12, TOD#12:30:12.1223452);</pre>
2	CONCAT_DATE	<pre> +-----+ CONCAT_DATE ANY_INT -- YEAR --DATE ANY_INT -- MONTH ANY_INT -- DAY +-----+</pre>	Concatenate a date and time of day: <pre> VAR myD: DATE; END_VAR myD:= CONCAT_DATE (2010,3,12);</pre>
3a	CONCAT_TOD	<pre> +-----+ CONCAT_TOD ANY_INT -- HOUR --TOD ANY_INT -- MINUTE ANY_INT -- SECOND ANY_INT -- MILLISECOND +-----+</pre>	Concatenate a time of day: <pre> VAR myTOD: TOD; END_VAR myTD:= CONCAT_TOD (16,33,12,0);</pre>
3b	CONCAT_LTOD	<pre> +-----+ CONCAT_LTOD ANY_INT -- HOUR --LTOD ANY_INT -- MINUTE ANY_INT -- SECOND ANY_INT -- MILLISECOND +-----+</pre>	Concatenate a time of day: <pre> VAR myTOD: LTOD; END_VAR myTD:= CONCAT_TOD (16,33,12,0);</pre>
4a	CONCAT_DT	<pre> +-----+ CONCAT_DT ANY_INT -- YEAR --DT ANY_INT -- MONTH ANY_INT -- DAY ANY_INT -- HOUR ANY_INT -- MINUTE ANY_INT -- SECOND ANY_INT -- MILLISECOND +-----+</pre>	Concatenate a time of day: <pre> VAR myDT: DT; Day: USINT; END_VAR Day := 17; myDT:= CONCAT_DT (2010,3,Day,12,33,12,0);</pre>
4b	CONCAT_LDT	<pre> +-----+ CONCAT_LDT ANY_INT -- YEAR --LDT ANY_INT -- MONTH ANY_INT -- DAY ANY_INT -- HOUR ANY_INT -- MINUTE ANY_INT -- SECOND ANY_INT -- MILLISECOND +-----+</pre>	Concatenate a time of day: <pre> VAR myDT: LDT; Day: USINT; END_VAR Day := 17; myDT:= CONCAT_LDT (2010,3,Day,12,33,12,0);</pre>

No.	Description	Graphical form	Example
Split time data types			
5	SPLIT_DATE	<pre> +-----+ SPLIT_DATE DATE-- IN YEAR -- ANY_INT MONTH -- ANY_INT DAY -- ANY_INT +-----+ </pre> See NOTE 2	Split a date: <pre> VAR myD: DATE:= DATE#2010-03-10; myYear: UINT; myMonth, myDay: USINT; END_VAR SPLIT_DATE (myD,myYear,myMonth,myDay); </pre>
6a	SPLIT_TOD	<pre> +-----+ SPLIT_TOD TOD-- IN HOUR -- ANY_INT MINUTE -- ANY_INT SECOND -- ANY_INT MILLISECOND -- ANY_INT +-----+ </pre> See NOTE 2	Split a time of day: <pre> VAR myTOD: TOD:= TOD#14:12:03; myHour, myMin, mySec: USINT; myMilliSec: UINT; END_VAR SPLIT_TOD(myTOD, myHour, myMin, mySec,myMilliSec); </pre>
6b	SPLIT_LTOD	<pre> +-----+ SPLIT_LTOD LTOD-- IN HOUR -- ANY_INT MINUTE -- ANY_INT SECOND -- ANY_INT MILLISECOND -- ANY_INT +-----+ </pre> See NOTE 2	Split a time of day: <pre> VAR myTOD: LTOD:= TOD#14:12:03; myHour, myMin, mySec: USINT; myMilliSec: UINT; END_VAR SPLIT_TOD(myTOD, myHour, myMin, mySec,myMilliSec); </pre>
7a	SPLIT_DT	<pre> +-----+ 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 +-----+ </pre> See NOTE 2	Split a date: <pre> 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); </pre>
7b	SPLIT_LDT	<pre> +-----+ 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 +-----+ </pre> See NOTE 2	Split a date: <pre> 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); </pre>
Get day of the week			
8	DAY_OF_WEEK	<pre> +-----+ DAY_OF_WEEK DATE-- IN -- ANY_INT +-----+ </pre> See NOTE 2	Get the day of week: <pre> VAR myD: DATE:= DATE#2010-03-10; myDoW: USINT; END_VAR myDoW:= DAY_OF_WEEK(myD); </pre>
The function DAY_OF_WEEK returns 0 for Sunday, 1 for Monday, ..., 6 for Saturday			

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. micro-second and nanosecond.

6.6.2.5.13 Functions for endianness conversion

The endianness conversion functions convert to and from the Implementer specific, internally used endianness of the PLC from and to the requested endianness.

Endianness 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 endianness 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 endianness.

EXAMPLE 1 Endianness

```
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 Endianness

```
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 endianness 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 endianness conversion.

Table 37 – Function for endianness conversion

No.	Description	Graphical form	Textual form
1	TO_BIG_ENDIAN	<pre> +-----+ TO_BIG_ENDIAN ANY -- IN --ANY +-----+ </pre>	Conversion to big endian data format A:= TO_BIG_ENDIAN(B);
2	TO_LITTLE_ENDIAN	<pre> +-----+ TO_LITTLE_ENDIAN ANY -- IN --ANY +-----+ </pre>	Conversion to little endian data format B:= TO_LITTLE_ENDIAN(A);
3	BIG_ENDIAN_TO	<pre> +-----+ FROM_BIG_ENDIAN ANY -- IN --ANY +-----+ </pre>	Conversion from big endian data format A:= FROM_BIG_ENDIAN(B);
4	LITTLE_ENDIAN_TO	<pre> +-----+ FROM_LITTLE_ENDIAN ANY -- IN --ANY +-----+ </pre>	Conversion from little endian data format A:= FROM_LITTLE_ENDIAN(B);
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.			

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	<pre> +-----+ . IS_VALID ANY_REAL-- IN --BOOL +-----+ </pre>	Validity of a REAL <pre> VAR R: REAL; END_VAR IF IS_VALID(R) THEN ... </pre>
2	IS_VALID_BCD	<pre> +-----+ IS_VALID_BCD -ANY_BIT-- IN --BOOL +-----+ </pre>	Validity test for a BCD word <pre> VAR W: WORD; END_VAR IF IS_VALID_BCD(W) THEN ... </pre>

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

No.	Description	Example
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;
-	EN/ENO inputs and outputs	Defined in Table 18
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
4d	Declaration of NON_RETAIN qualifier on output variables	VAR_OUTPUT NON_RETAIN X: REAL; END_VAR
4e	Declaration of RETAIN qualifier on static variables	VAR RETAIN X: REAL; END_VAR
4f	Declaration of NON_RETAIN qualifier on static variables	VAR NON_RETAIN X: REAL; END_VAR
5a	Declaration of RETAIN qualifier on local FB instances	VAR RETAIN TMR1: TON; END_VAR
5b	Declaration of NON_RETAIN qualifier on local FB instances	VAR NON_RETAIN TMR1: TON; END_VAR
6a	Textual declaration of - rising edge inputs	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
6b	- falling edge inputs (textual)	See above
7a	Graphical declaration of - rising edge inputs (>)	FUNCTION_BLOCK (* External interface *) +-----+ AND_EDGE BOOL-->X Z --BOOL BOOL--<Y +-----+ (* FB body *) +-----+ & X-- --Z Y-- +-----+ END_FUNCTION_BLOCK
7b	Graphical declaration of - falling edge inputs (<)	See above
NOTE The features 1-3 of this table are equivalent to functions, see Table 19.		

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 |
+-----+
| IN | OUT |
|---|---|
| DB_TIME | ET_OFF |
|---|---|
+-----+

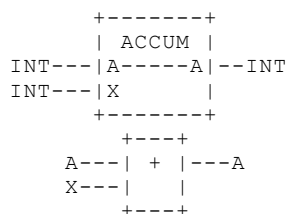
(* Function block type body *)
DB_ON DB_FF
+-----+ +-----+
| TON | | SR |
|---|---| |---|---|
IN---+---| IN Q |---| S1 Q |---OUT
| | +---| PT ET | +---| R |
| | +-----+ | +-----+
| | DB_OFF |
| | +-----+ |
| | TON |
|---|---O IN Q |---+
DB_TIME---+---| PT ET |-----ET_OFF
+-----+
```

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

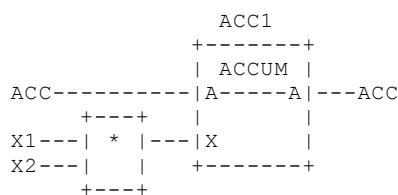


```

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



```

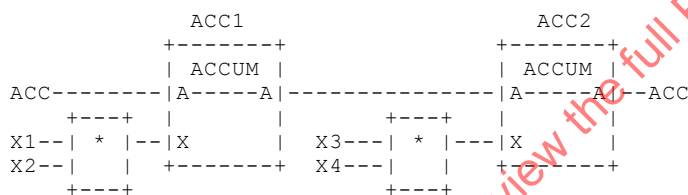
VAR
  ACC: INT;
  X1:  INT;
  X2:  INT;
END_VAR

```

This declaration is assumed: the effect of execution:

```
ACC:= ACC+X1*X2;
```

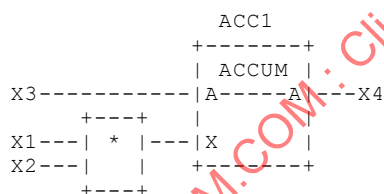
b) Allowed usage of function block instance and function



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



```

VAR
  X1: INT;
  X2: INT;
  X3: INT;
  X4: INT;
END_VAR

```

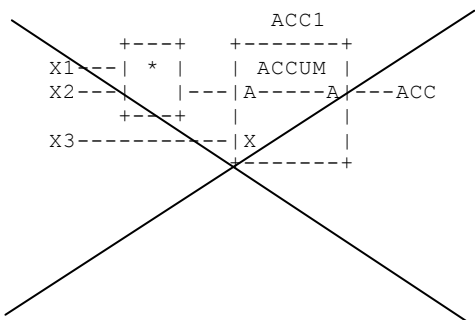
The declaration is assumed: the effect of execution:

```

X3:= X3+X1*X2;
X4:= X3;

```

d) Allowed usage of function block instance and function – with assignment to an output



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)	<pre>VAR FB_instance_1, FB_instance_2: my_FB_Type; T1, T2, T3: TON; END_VAR</pre>
2	Declaration of FB instance with initialization of its variables	<pre>VAR TempLoop: PID:= (PropBand:= 2.5, Integral:= T#5s); END_VAR</pre> 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.	<pre> YourCTU(EN:= not B, CU:= r, PV:= c1, ENO=> next, Q => out, CV => c2); </pre>
2	Incomplete formal call (textual only)	<pre> YourCTU(Q => out, CV => c2); </pre> EN, CU, PV variable will have the value of the last call or an initial value, if never called before.
3	Graphical call	<pre> YourCTU +-----+ CTU +-----+ B -- EN ENO -- next r -- CU Q -- out c1 -- PV CV -- c2 +-----+ </pre>

No.	Description	Example
4	Graphical call with negated boolean input and output	<pre> YourCTU +-----+ CTU B -0 EN ENO -- next r -- CU Q 0- out c1 -- PV CV -- c2 +-----+ </pre> The use of these constructs is forbidden for in-out variables.
5a	Graphical call with usage of VAR_IN_OUT	
5b	Graphical call with assignment of VAR_IN_OUT to a variable	
6a	Textual call with separate assignment of input FB_Instance.Input:= x;	<pre> YourTon.IN:= r; YourTon.PT:= t; YourTon(not Q => out); </pre>
6b	Graphical call with separate assignment of input	<pre> +-----+ r-- MOVE --YourCTU.CU +-----+ +-----+ c-- MOVE --YourCTU.PV +-----+ </pre> <pre> YourCTU +-----+ CTU 1-- EN ENO -- next -- CU Q 0- out -- PV CV -- +-----+ </pre>
7	Textual output read after FB call x:= FB_Instance.Output;	
8a	Textual output assigned in FB call	
8b	Textual output assigned in FB call with negation	
9a	Textual call with function block instance name as input	<pre> VAR_INPUT I_TMR: TON; END_VAR EXPIRED:= I_TMR.Q; </pre> It is assumed that the variables EXPIRED and A_VAR have been declared of type BOOL in this example and in the following examples.
9b	Graphical call with function block instance name as input	a
10a	Textual call with function block instance name as VAR_IN_OUT	<pre> VAR_IN_OUT IO_TMR: TOF; END_VAR IO_TMR (IN:=A_VAR, PT:= T#10S); EXPIRED:= IO_TMR.Q; </pre>
10b	Graphical call with function block instance name as VAR_IN_OUT	
11a	Textual call with function block instance name as external variable	<pre> VAR_EXTERNAL EX_TMR: TOF; END_VAR EX_TMR(IN:= A_VAR, PT:= T#10S); EXPIRED:= EX_TMR.Q; </pre>
11b	Graphical call with function block instance name as external variable	

EXAMPLE Function block call with immediate and separate parameter assignment

```

      YourCTU
    +-----+
    |   CTU   |
B -0|EN  ENO|--
r --|CU    Q|0-out
c --|PV    CV|--
    +-----+

```

```

YourCTU (EN:= not b,
         CU:= r,
         PV:= c,
         not Q => out);

```

a) FB call with immediate assignment of inputs (typical usage)

```

    +-----+
r--| MOVE |--YourCTU.CU
    +-----+
    +-----+
c--| MOVE |--YourCTU.PV
    +-----+

```

```

YourCTU.CU:= r;
YourCTU.PV:= V;

YourCTU(not Q => out);

```

```

      YourCTU
    +-----+
    |   CTU   |
--|EN  ENO|--
--|CU    Q|0-out
--|PV    CV|--
    +-----+

```

b) FB call with separate assignment of input

```

      YourCTU
    +-----+
    |   CTU   |
a--| NE |---0|EN  ENO|--
b--|   |r--|CU    Q|0-out
    +-----+
    +-----+

```

```

VAR a, b, r, out: BOOL;
YourCTU: CTU; END_VAR

YourCTU (EN := NOT (a <> b),
         CU := r,
         NOT Q => out);

```

c) FB call with immediate access to output (typical usage)

Also negation in call is permitted

```

      FF75
    +-----+
    |   SR   |
bIn1---|S1  Q1|---bOut3
bIn2---|R   |
    +-----+

```

```

VAR FF75: SR; END_VAR (* Declaration *)

FF75(S1:= bIn1,      (* call *)
     R:= bIn2);

bOut3:= FF75.Q1;    (* Assign Output *)

```

d) FB call with textual separate output assignment (after call)

```

      TONs[12]
    +-----+
    |   TON   |
bIn1  --|IN   Q|--
T#10ms --|PT  ET|--
    +-----+

```

```

VAR
  TONs: array [0..100] OF TON;
  i: INT;
END_VAR

TON[12](IN:= bIn1, PT:= T#10ms);

```

```

      TONs[i]
    +-----+
    |   TON   |
bIn1  --|IN   Q|--
T#20ms --|PT  ET|--
    +-----+

```

```

TON[i](IN:= bIn1, PT:= T#20ms);

```

e) FB call using an instance array

```

      myCooler.Cooling
    +-----+
    |   TOF   |
bIn1  --|IN   Q|--
T#30s  --|PT  ET|--
    +-----+

```

```

TYPE
  Cooler: STRUCT
    Temp: INT;
    Cooling: TOF;
  END_STRUCT;
END_TYPE

VAR
  myCooler: Cooler;
END_VAR

myCooler.Cooling(IN:= bIn1, PT:= T#30s);

```

f) FB call using an instance as structure element

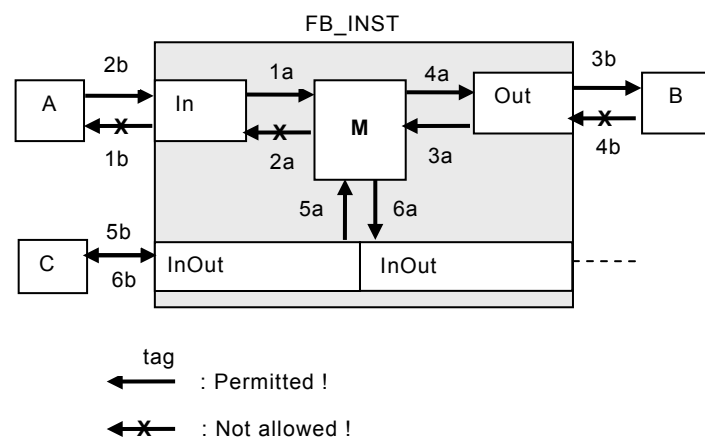
6.6.3.4.2 Usage of input and output parameters

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.

<pre> 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 </pre>		
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
+-----+
|  TON  |
GO---| IN  Q |
t#100ms---| PT ET |
+-----+

I_BLK
+-----+
|  INSIDE_A  |
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
      +-----+
      | TON |
      | IN  Q|
t#100ms---| PT ET|
          +-----+

      I_BLK
      +-----+
      |   INSIDE_B   |
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

No.	Description/Graphical form	Function block body
1a	Bistable function block (set dominant): SR(S1, R, Q1)	
	<pre> +-----+ SR S1 Q1 ---BOOL R +-----+ </pre>	<pre> +-----+ S1 ----- >=1 --- Q1 +-----+ R -----0 & --- Q1 ----- --- +-----+ </pre>
1b	Bistable function block (set dominant) with long input names: SR(SET1, RESET, Q1)	
	<pre> +-----+ SR SET1 Q1 ---BOOL RESET +-----+ </pre>	<pre> +-----+ SET1 ----- >=1 --- Q1 +-----+ RESET -0 & --- Q1 ----- --- +-----+ </pre>
2a	Bistable function block (reset dominant): RS(S, R1, Q1)	
	<pre> +-----+ RS S Q1 ---BOOL R1 +-----+ </pre>	<pre> +-----+ R1 -----0 & --- Q1 +-----+ S ----- >=1 --- Q1 ----- --- +-----+ </pre>
2b	Bistable function block (reset dominant) with long input names: RS(SET, RESET1, Q1)	
	<pre> +-----+ RS SET Q1 ---BOOL R1 +-----+ </pre>	<pre> +-----+ RESET1 -----0 & --- Q1 +-----+ SET ----- >=1 --- Q1 ----- --- +-----+ </pre>
^a The initial state of the output variable Q1 shall be the normal default value of zero for Boolean variables.		

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

No.	Description/Graphical form	Definition (ST language)
1	Rising edge detector: R_TRIG (CLK, Q)	
	<pre> +-----+ R_TRIG BOOL -- CLK Q -- BOOL +-----+ </pre>	<pre> 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 </pre>
2	Falling edge detector: F_TRIG (CLK, Q)	
	<pre> +-----+ F_TRIG BOOL -- CLK Q -- BOOL +-----+ </pre>	<pre> 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 </pre>
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.		

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

No.	Description/Graphical form	Function block body (ST language)
	Up-Counter	
1a	CTU_INT (CU, R, PV, Q, CV) or CTU(..)	
	<pre> +-----+ CTU BOOL--->CU Q ---BOOL BOOL--- R INT--- PV CV ---INT +-----+ </pre> and also: <pre> +-----+ CTU_INT BOOL--->CU Q ---BOOL BOOL--- R INT--- PV CV ---INT +-----+ </pre>	<pre> 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); </pre>
1b	CTU_DINT PV, CV: DINT	see 1a
1c	CTU_LINT PV, CV: LINT	see 1a
1d	CTU_UDINT PV, CV: UDINT	see 1a
1e	CTU_ULINT (CD, LD, PV, CV) PV, CV: ULINT	see 1a
	Down-counters	
2a	CTD_INT (CD, LD, PV, Q, CV) or CTD	

No.	Description/Graphical form	Function block body (ST language)
	<pre> +-----+ CTD BOOL--->CD Q ---BOOL BOOL--- LD INT--- PV CV ---INT +-----+ and also: +-----+ CTD_INT BOOL--->CD Q ---BOOL BOOL--- LD INT--- PV CV ---INT +-----+ </pre>	<pre> VAR_INPUT CU: BOOL R_EDGE; ... // The edge is evaluated internally by the data type R_EDGE IF LD THEN CV:= PV; ELSIF CD AND (CV > PVmin) THEN CV:= CV-1; END_IF; Q:= (CV <= 0); </pre>
2b	CTD_DINT PV, CV: DINT	See 2a
2c	CTD_LINT PV, CV: LINT	
2d	CTD_UDINT PV, CV: UDINT	See 2a
2e	CTD_ULINT PV, CV: ULINT	See 2a
Up-down counters		
3a	CTUD_INT(CD, LD, PV, Q, CV) or CTUD(..)	
	<pre> +-----+ CTUD BOOL--->CU QU ---BOOL BOOL--->CD QD ---BOOL BOOL--- R BOOL--- LD INT--- PV CV ---INT +-----+ and also: +-----+ CTUD_INT BOOL--->CU QU ---BOOL BOOL--->CD QD ---BOOL BOOL--- R BOOL--- LD INT--- PV CV ---INT +-----+ </pre>	<pre> VAR_INPUT CU, CD: BOOL R_EDGE; ... // Edge is evaluated internally by the data type R_EDGE 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>
3b	CTUD_DINT PV, CV: DINT	See 3a
3c	CTUD_LINT PV, CV: LINT	See 3a
3d	CTUD_UDINT PV, CV: UDINT	See 3a
3e	CTUD_ULINT PV, CV: ULINT	See 3a
NOTE The numerical values of the limit variables PVmin and PVmax are Implementer specific.		

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.

Figure 15 below shows the timing diagrams of the standard timer function blocks.

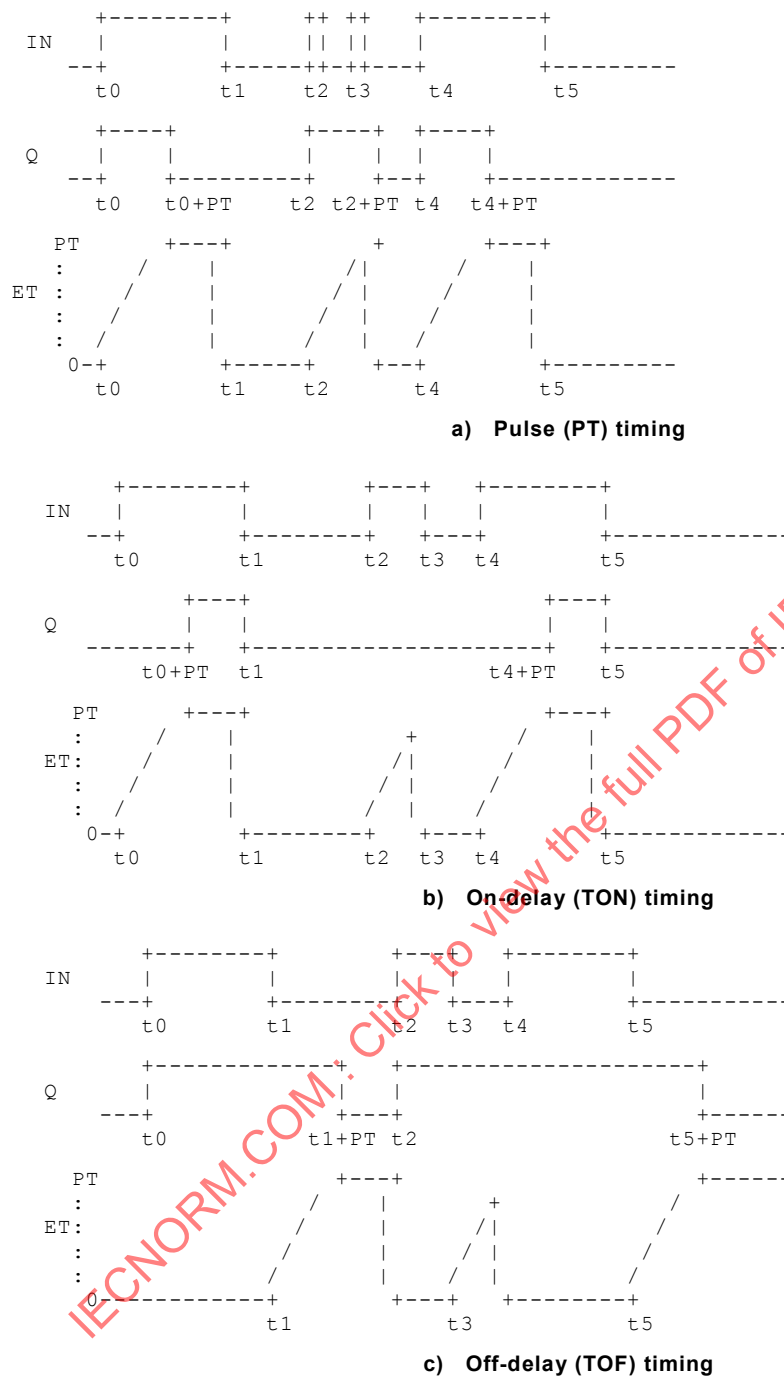


Figure 15 – Standard timer function blocks – timing diagrams (Rules)

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 <code>PROGRAM ... END_PROGRAM</code>	<code>PROGRAM myPrg ... END_PROGRAM</code>
2a	Declaration of inputs <code>VAR_INPUT ... END_VAR</code>	<code>VAR_INPUT IN: BOOL; T1: TIME; END_VAR</code>
2b	Declaration of outputs <code>VAR_OUTPUT ... END_VAR</code>	<code>VAR_OUTPUT OUT: BOOL; ET_OFF: TIME; END_VAR</code>
2c	Declaration of in-outs <code>VAR_IN_OUT ... END_VAR</code>	<code>VAR_IN_OUT A: INT; END_VAR</code>
2d	Declaration of temporary variables <code>VAR_TEMP ... END_VAR</code>	<code>VAR_TEMP I: INT; END_VAR</code>
2e	Declaration of static variables <code>VAR ... END_VAR</code>	<code>VAR B: REAL; END_VAR</code>
2f	Declaration of external variables <code>VAR_EXTERNAL ... END_VAR</code>	<code>VAR_EXTERNAL B: REAL; END_VAR</code> Corresponding to <code>VAR_GLOBAL B: REAL</code>
2g	Declaration of external variables <code>VAR_EXTERNAL CONSTANT ... END_VAR</code>	<code>VAR_EXTERNAL CONSTANT B: REAL; END_VAR</code> Corresponding to <code>VAR_GLOBAL B: REAL</code>
3a	Initialization of inputs	<code>VAR_INPUT MN: INT:= 0;</code>
3b	Initialization of outputs	<code>VAR_OUTPUT RES: INT:= 1;</code>
3c	Initialization of static variables	<code>VAR B: REAL:= 12.1;</code>
3d	Initialization of temporary variables	<code>VAR_TEMP I: INT:= 1;</code>
4a	Declaration of <code>RETAIN</code> qualifier on input variables	<code>VAR_INPUT RETAIN X: REAL; END_VAR</code>
4b	Declaration of <code>RETAIN</code> qualifier on output variables	<code>VAR_OUTPUT RETAIN X: REAL; END_VAR</code>
4c	Declaration of <code>NON_RETAIN</code> qualifier on input variables	<code>VAR_INPUT NON_RETAIN X: REAL; END_VAR</code>

No.	Description	Example
4d	Declaration of <code>NON_RETAIN</code> qualifier on output variables	<code>VAR_OUTPUT NON_RETAIN X: REAL; END_VAR</code>
4e	Declaration of <code>RETAIN</code> qualifier on static variables	<code>VAR RETAIN X: REAL; END_VAR</code>
4f	Declaration of <code>NON_RETAIN</code> qualifier on static variables	<code>VAR NON_RETAIN X: REAL; END_VAR</code>
5a	Declaration of <code>RETAIN</code> qualifier on local FB instances	<code>VAR RETAIN TMR1: TON; END_VAR</code>
5b	Declaration of <code>NON_RETAIN</code> qualifier on local FB instances	<code>VAR NON_RETAIN TMR1: TON; END_VAR</code>
6a	Textual declaration of - rising edge inputs	<pre> 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 </pre>
6b	Textual declaration of - falling edge inputs (textual)	See above
7a	Graphical declaration of - rising edge inputs (>)	<pre> PROGRAM (* External interface *) +-----+ AND_EDGE +-----+ (* FB body *) +-----+ & X-- --Z Y-- +-----+ END_PROGRAM </pre>
7b	Graphical declaration of - falling edge inputs (<)	See above
8a	<code>VAR_GLOBAL...END_VAR</code> declaration within a PROGRAM	<code>VAR_GLOBAL z1: BYTE; END_VAR</code>
8b	<code>VAR_GLOBAL CONSTANT</code> declarations within PROGRAM type declarations	<code>VAR_GLOBAL CONSTANT z2: BYTE; END_VAR</code>
9	<code>VAR_ACCESS...END_VAR</code> declaration within a PROGRAM	<pre> VAR_ACCESS ABLE: STATION_1.%IX1.1: BOOL READ_ONLY; BAKER: STATION_1.P1.x2: UINT READ_WRITE; END_VAR </pre>
NOTE The features 2a to 7b are equivalent to the same features of Table 40 for function blocks.		

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

Class (= type of a class)

Object (= instance of a class)

PLC languages of the standard

Type of a function block and 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.

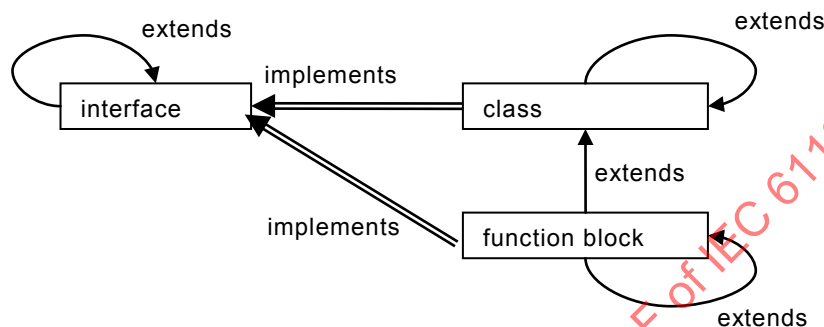


Figure 16 – Overview of inheritance and interface implementation

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	<code>CLASS ... END_CLASS</code>	Class definition
1a	<code>FINAL</code> specifier	Class cannot be used as a base class
	Adapted from function block	
2a	Declaration of variables <code>VAR ... END_VAR</code>	<code>VAR B: REAL; END_VAR</code>
2b	Initialization of variables	<code>VAR B: REAL:= 12.1; END_VAR</code>
3a	<code>RETAIN</code> qualifier on internal variables	<code>VAR RETAIN X: REAL; END_VAR</code>
3b	<code>NON_RETAIN</code> qualifier on internal variables	<code>VAR NON_RETAIN X: REAL; END_VAR</code>
4a	<code>VAR_EXTERNAL</code> declarations within class type declarations	For equivalent example see Table 40
4b	<code>VAR_EXTERNAL CONSTANT</code> declarations within class type declarations	For equivalent example see Table 40
	Methods and specifiers	
5	<code>METHOD...END_METHOD</code>	Method definition
5a	<code>PUBLIC</code> specifier	Method may be called from anywhere
5b	<code>PRIVATE</code> specifier	Method may only be called from inside the defining POU
5c	<code>INTERNAL</code> specifier	Method may only be called from inside the same namespace
5d	<code>PROTECTED</code> specifier	Method may only be called from inside the defining POU and its derivations (default)
5e	<code>FINAL</code> specifier	Method shall not be overridden
	Inheritance	- these features are the same as in Table 53 feature inheritance
6	<code>EXTENDS</code>	Class inherits from class (NOTE: no inheritance from FB)
7	<code>OVERRIDE</code>	Method overrides base method – see dynamic name binding
8	<code>ABSTRACT</code>	Abstract class – at least one method is abstract Abstract method – this method is abstract
	Access reference	
9a	<code>THIS</code>	Reference to own methods
9b	<code>SUPER</code>	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	<pre> VAR MyCounter1: CCounter; END_VAR </pre>
2	Declaration of class instance with initialization of its public variables	<pre> VAR MyCounter2: CCounter:= (m_iUpperLimit:=20000; m_iLowerLimit:=-20000); END_VAR </pre>

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`.

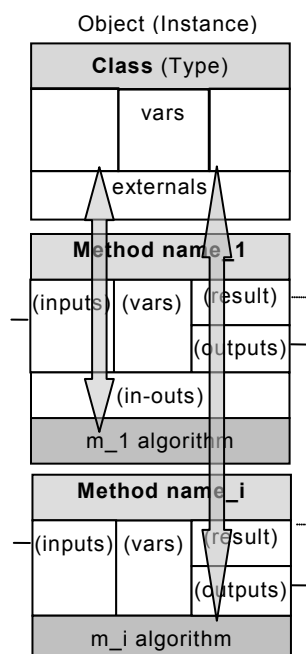
NOTE 1 Overloading of methods is not in the scope of this part of IEC 61131.

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.

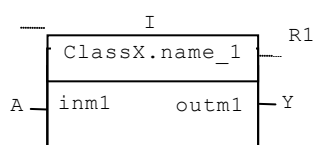
(Temporary parameters are parenthesized.)

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;
    Max: UINT := 1000;
  END_VAR

  METHOD PUBLIC UP: UINT
    VAR_INPUT INC: UINT; END_VAR
    VAR_OUTPUT QU: BOOL; END_VAR

    IF CV <= Max - INC
      THEN CV := CV + INC;
           QU := FALSE;
    ELSE QU := TRUE;
    END_IF
    UP := CV;
  END_METHOD

  METHOD PUBLIC UP5: UINT
    VAR_OUTPUT QU: BOOL; END_VAR
    UP5 := THIS.UP(INC := 5, QU => QU);
  END_METHOD
END_CLASS
```

```
// Current value of counter

// Method for count up by inc
// Increment
// Upper limit detection

// Count up of current value

// Upper limit reached

// Result of method

// Count up by 5
// Upper limit reached
// Internal method call
```

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.	<code>A:= COUNTER.UP(EN:= TRUE, INC:= B, START:= 1, ENO=> %MX1, QU => C);</code>
1b	Incomplete formal call (textual only) Shall be used if EN/ENO is not necessary in calls.	<code>A:= COUNTER.UP(INC:= B, QU => C);</code> START variable will have the default value 0 (zero).
2	Non-formal call (textual only) (fix order and complete)	<code>A:= COUNTER.UP(B, 1, C);</code> 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.

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013

EXAMPLE Internal and external method call

```

VAR
  CT:    COUNTER;
  LIMIT: BOOL;
  VALUE: UINT;
END_VAR

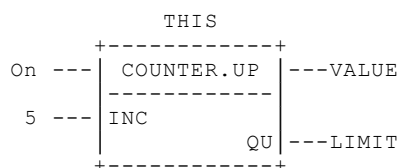
```

1) In Structured Text (ST)**a) Internal call of a method:**

```
VALUE := THIS.UP (INC := 5, QU => LIMIT);
```

b) External call of a method:

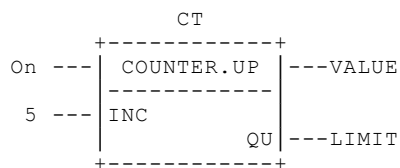
```
VALUE := CT.UP (INC := 5, QU => LIMIT);
```

2) In Function Block Diagram (FBD)**a) Internal call of a method**

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

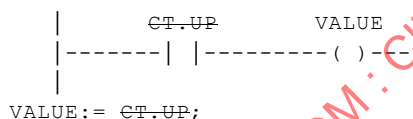
b) External call of a method:

CT 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

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.

Class inheritance
using **EXTENDS**

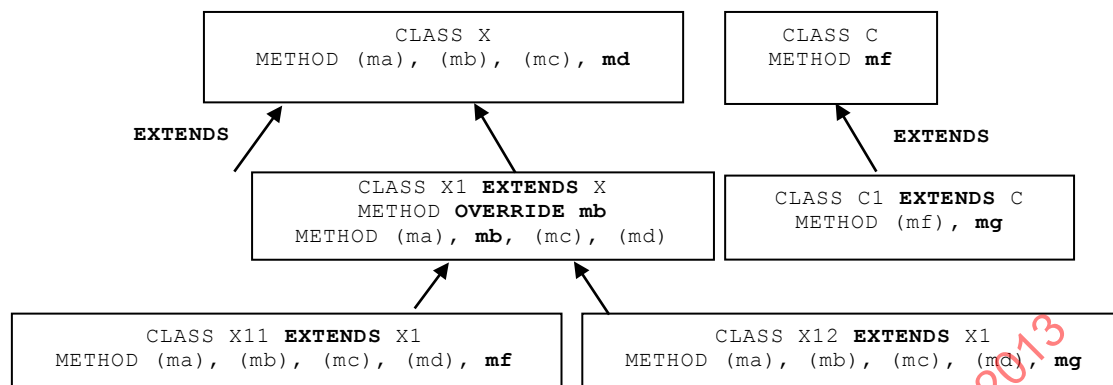


Figure 17 – Inheritance of classes (Illustration)

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

```

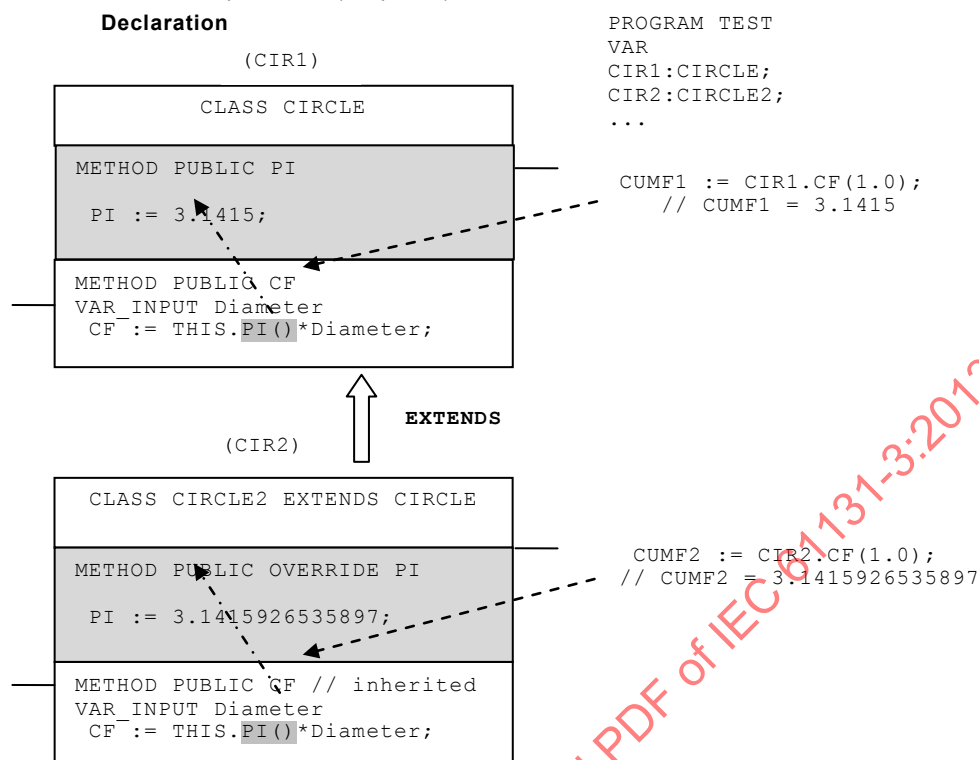
In this example the class `CIRCLE` contains an internal call of its method `PI` with low accuracy to calculate the circumference (CF) of a circle.

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)



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.

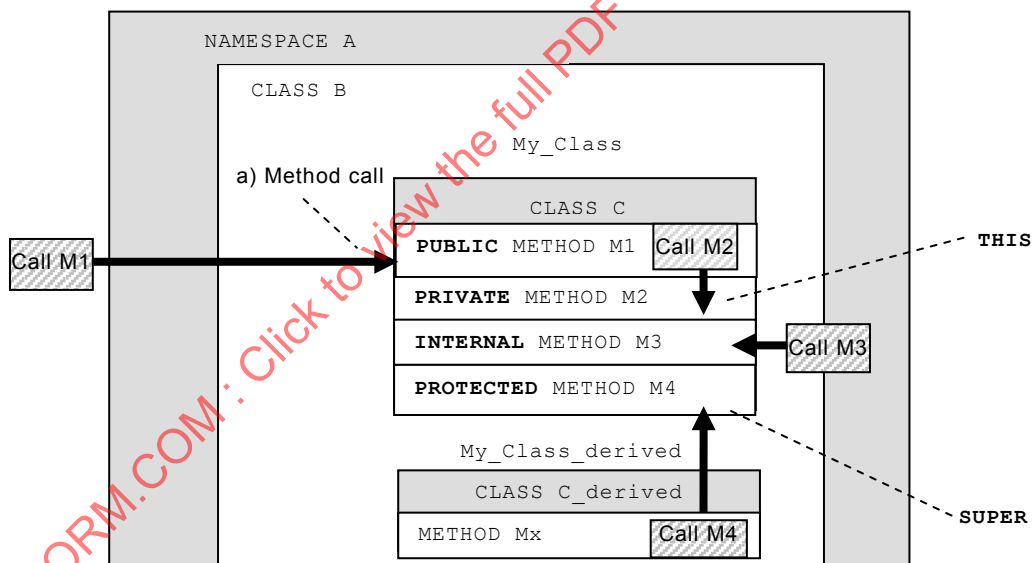
The access to method prototypes is implicitly always `PUBLIC`; therefore no access specifier is used on method prototypes.

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	<code>INTERFACE ...</code> <code>END_INTERFACE</code>	Interface definition
	Methods and specifiers	
2	<code>METHOD ...</code> <code>END_METHOD</code>	Method definition
	Inheritance	
3	<code>EXTENDS</code>	Interface inherits from interface
	Usage of interface	
4a	<code>IMPLEMENTS interface</code>	Implements an interface in a class declaration
4b	<code>IMPLEMENTS multi-interfaces</code>	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.

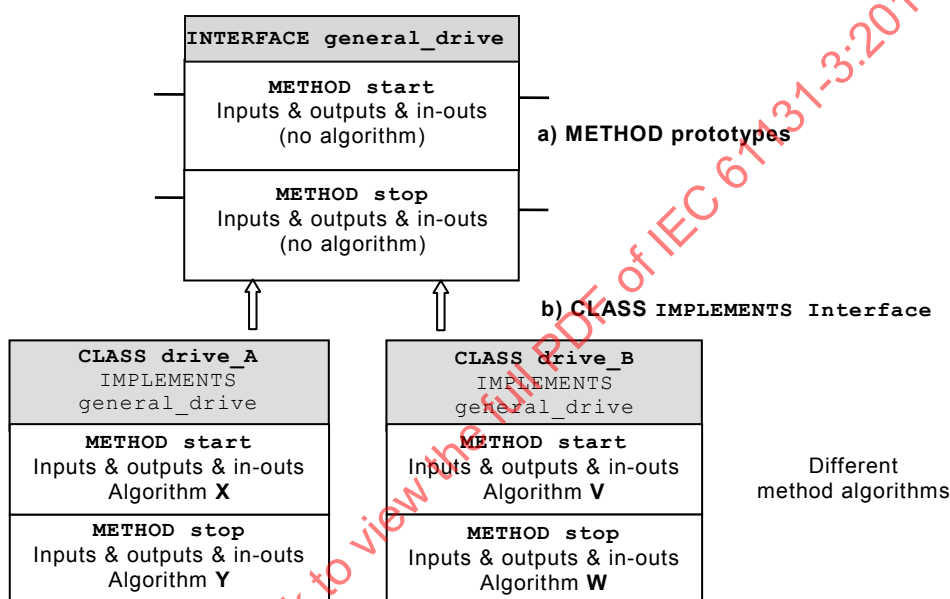


Figure 18 – Interface with derived classes (Illustration)

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
        My_Room:          LIGHTROOM;    // Instantiations
        My_Room_Ctrl:      ROOM_CTRL;    // See LIGHTROOM IMPLEMENTS ROOM
    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:

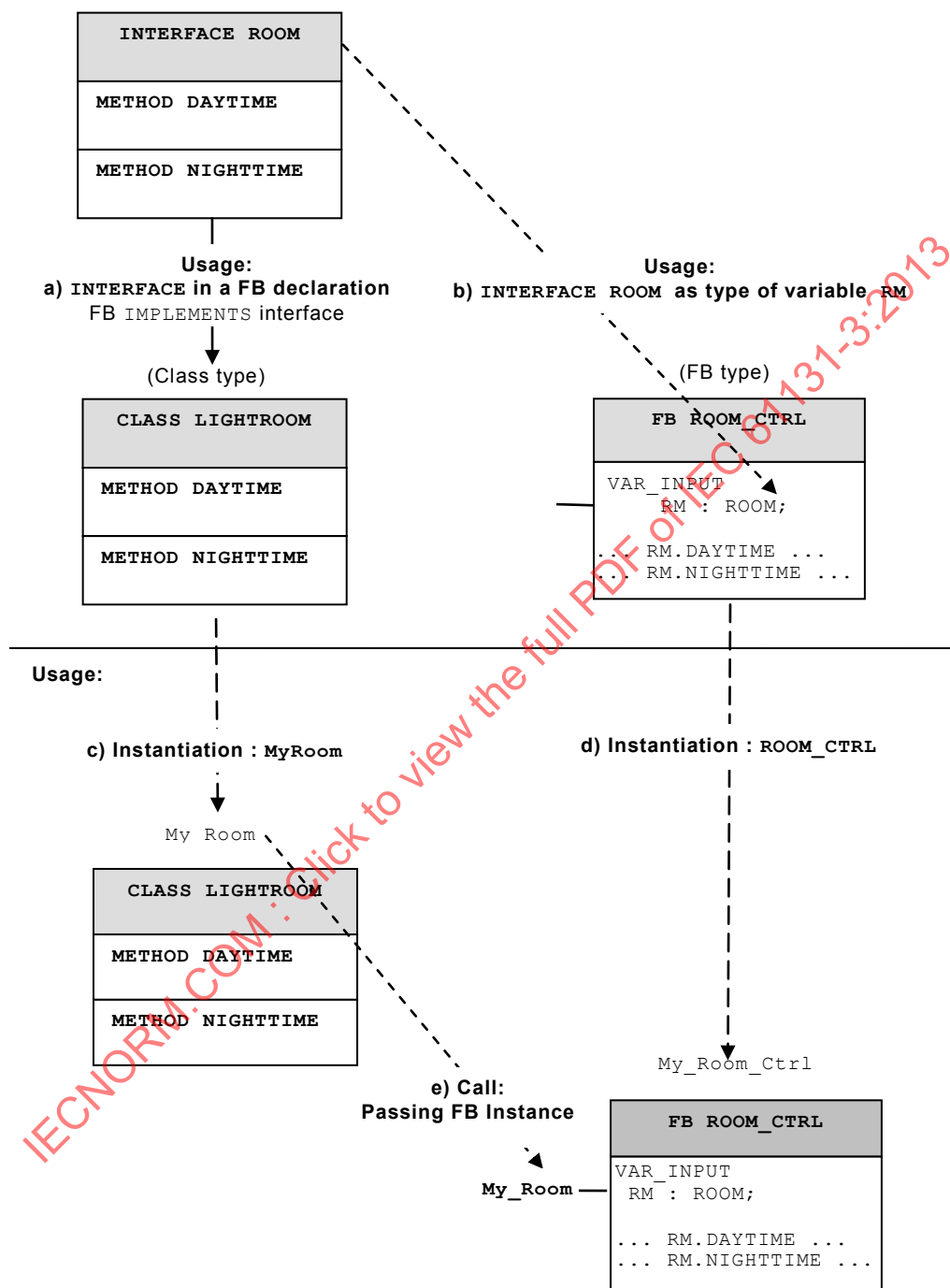
Program B instantiates the class `My_Room` and the function block `My_Room_Ctrl`

and calls the function block `My_Room_Ctrl` with passing the class `My_Room` to input variable `RM` of type interface `ROOM`.

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013

EXAMPLE 2 Illustration of the relationship of Example 1 above.

Declaration:



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.

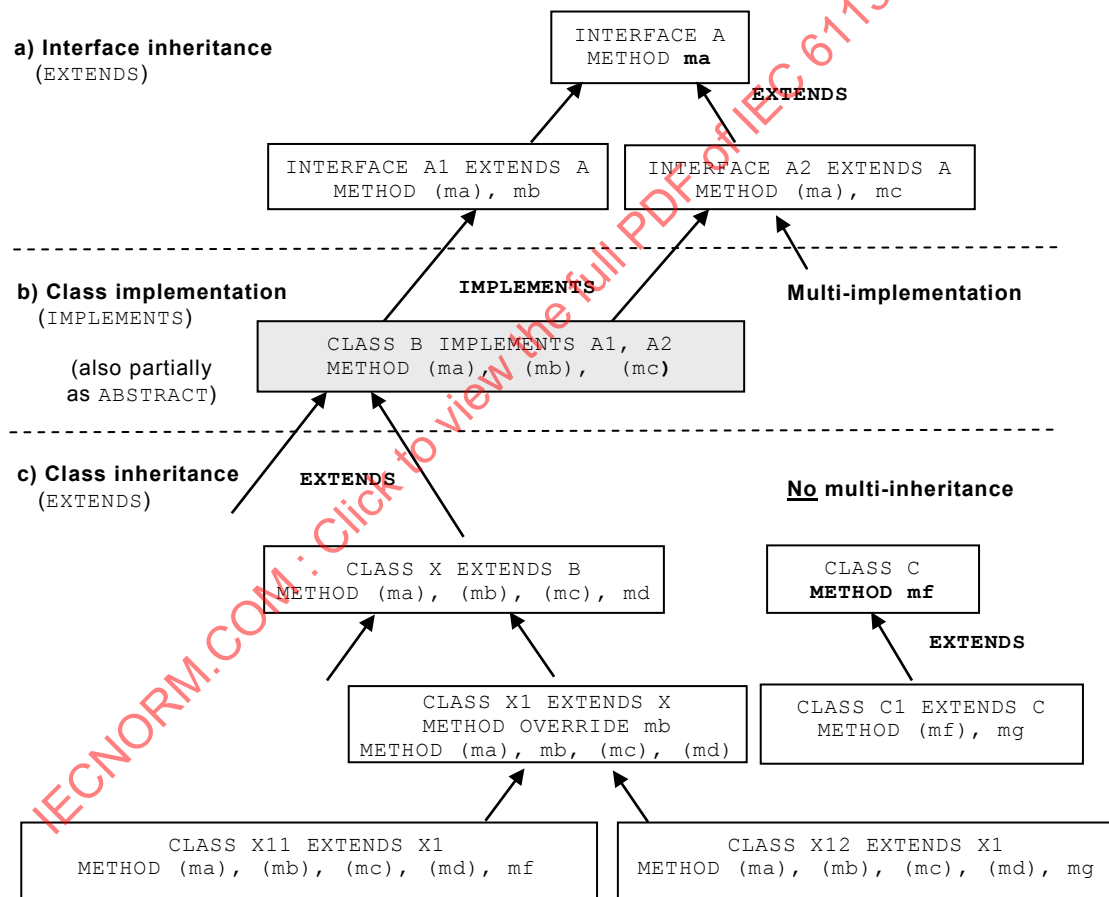


Illustration of the hierarchy of inheritance

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

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013

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 block.
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.

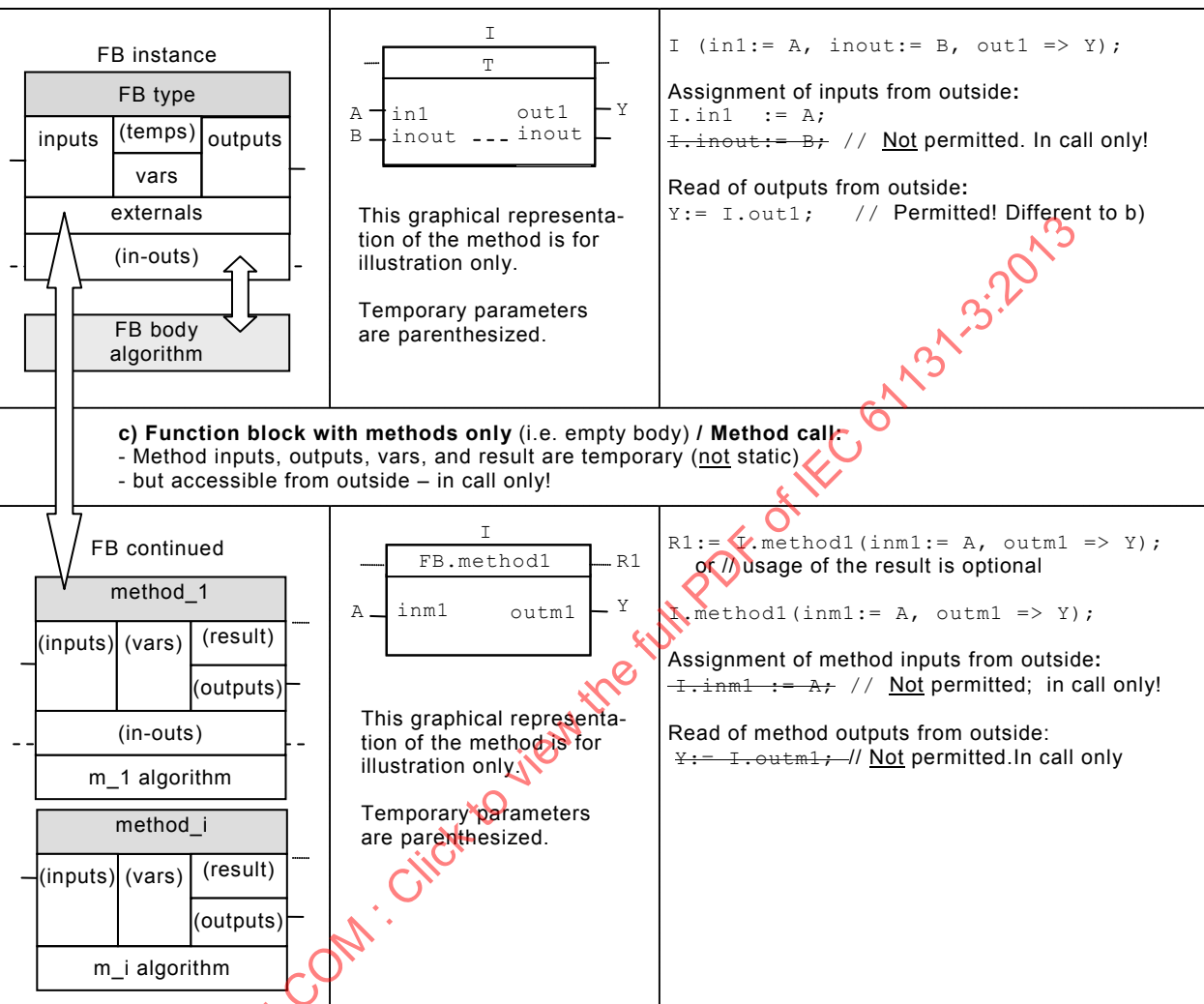
In this case this function block has an empty function block body implemented. The elements of the function block and the call of a method are shown in the example in Figure 20 b)

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.



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()` :

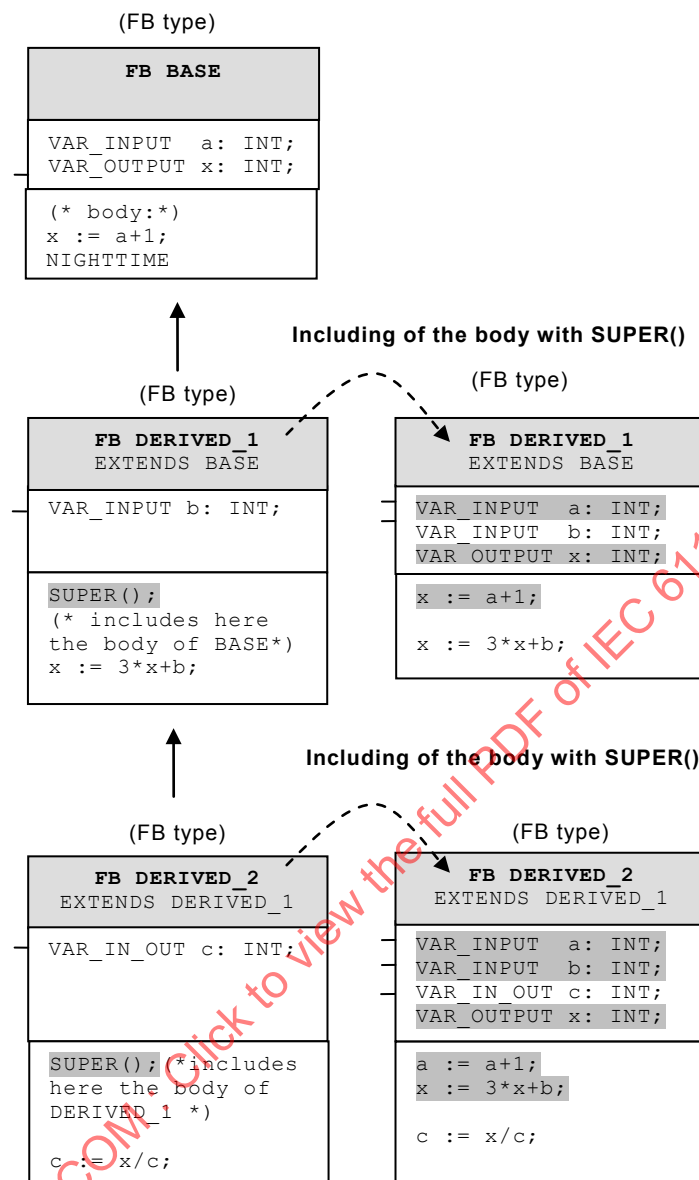


Figure 21 – Inheritance of function block body with `SUPER()` (Example)

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

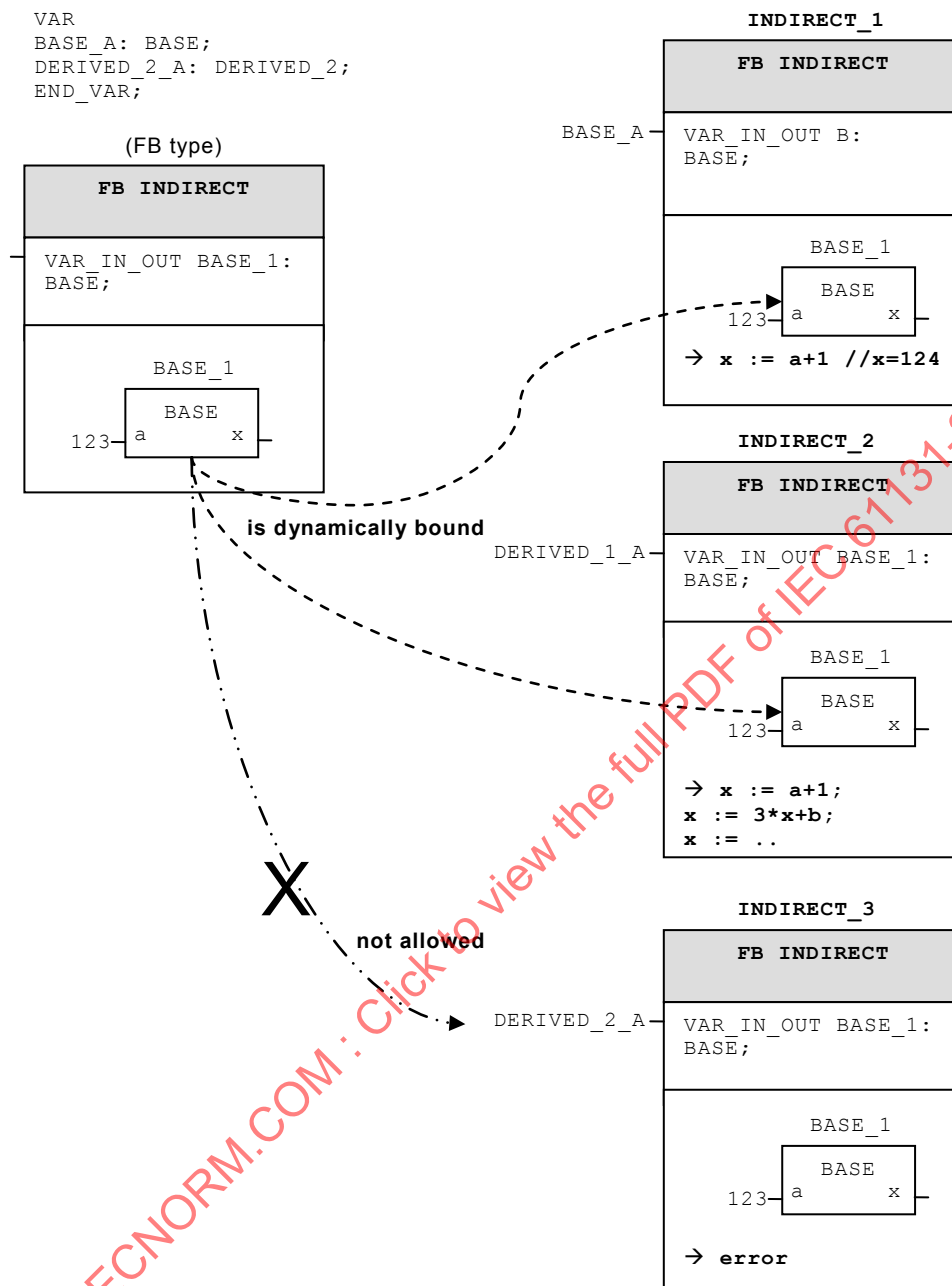
Since an interface cannot be instantiated, only derived types may be assigned to an interface reference. Thus, any call of a method via an interface reference is a case of dynamic binding.

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



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

No.	Description	Representation
1a	Step – graphical form with directed links	<pre> +-----+ *** +-----+ </pre>
1b	Initial step – graphical form with directed link	<pre> +=====+ *** +=====+ </pre>
2a	Step – textual form without directed links	<pre> STEP ***: (* Step body *) END_STEP </pre>

No.	Description	Representation
2b	Initial step – textual form without directed links	<pre> INITIAL_STEP ***: (* Step body *) END_STEP </pre>
3a ^a	Step flag – general form <code>***.X = BOOL#1</code> when <code>***</code> is active, <code>BOOL#0</code> otherwise	<code>***.X</code>
3b ^a	Step flag – direct connection of Boolean variable <code>***.X</code> to right side of step	<pre> +-----+ *** ---- +-----+ </pre>
4 ^a	Step elapsed time – general form <code>***.T = a variable of type TIME</code>	<code>***.T</code>
NOTE 1 The upper directed link to an initial step is not present if it has no predecessors. NOTE 2 <code>***</code> = 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: <pre> S4.X:= 1; (* ERROR *) S4.T:= t#100ms; (* ERROR *) </pre>		

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.

- By placing the appropriate Boolean expression in the ST language physically or logically adjacent to the vertical directed link.
- By a ladder diagram network in the LD language physically or logically adjacent to the vertical directed link.
- By a network in the FBD language defined in 8.3, physically or logically adjacent to the vertical directed link.
- By a LD or FBD network whose output intersects the vertical directed link via a connector.
- 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

No.	Description	Example
1 ^a	Transition condition physically or logically adjacent to the transition using ST language	<pre> +-----+ STEP7 +-----+ + bvar1 & bvar2 +-----+ STEP8 +-----+ </pre>
2 ^a	Transition condition physically or logically adjacent to the transition using LD language	<pre> +-----+ STEP7 +-----+ + bvar1 & bvar2 +-----+ STEP8 +-----+ </pre>

No.	Description	Example
3 ^a	Transition condition physically or logically adjacent to the transition using FBD language	<pre> +-----+ STEP7 +-----+ & +-----+ bvar1 --- bvar2 --- +-----+ STEP8 +-----+ </pre>
4 ^a	Use of connector	<pre> +-----+ STEP7 +-----+ +-----+ STEP8 +-----+ </pre> >TRANX>-----
5 ^a	Transition condition: Using LD language	<pre> bvar1 bvar2 +--- ----- ----->TRANX> </pre>
6 ^a	Transition condition: Using FBD language	<pre> & +-----+ bvar1 --- bvar2 --- +-----+ ----->TRANX> </pre>
7 ^b	Textual equivalent of feature 1 using ST language	<pre> STEP STEP7: END_STEP TRANSITION FROM STEP7 TO STEP8 := bvar1 & bvar2; END_TRANSITION STEP STEP8: END_STEP </pre>
8 ^b	Textual equivalent of feature 1 using IL language	<pre> STEP STEP7: END_STEP TRANSITION FROM STEP7 TO STEP 8: LD bvar1 AND bvar2 END_TRANSITION STEP STEP8: END_STEP </pre>
9 ^a	Use of transition name	<pre> +-----+ STEP7 +-----+ + TRAN7 TO STEP8 +-----+ STEP8 +-----+ </pre>
10 ^a	Transition condition using LD language	<pre> TRANSITION TRAN78 FROM STEP7 TO STEP8: bvar1 bvar2 TRAN78 +--- ----- ----- () ---+ END_TRANSITION </pre>
11 ^a	Transition condition using FBD language	<pre> TRANSITION TRAN78 FROM STEP7 TO STEP8: & +-----+ bvar1 --- bvar2 --- +-----+ -----END_TRANSITION </pre>

No.	Description	Example
12 ^b	Transition condition using IL language	<pre> TRANSITION TRAN78 FROM STEP7 TO STEP8: LD bvar1 AND bvar2 END_TRANSITION </pre>
13 ^b	Transition condition using ST language	<pre> TRANSITION TRAN78 FROM STEP7 TO STEP8 := bvar1 & bvar2; END_TRANSITION </pre>
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.	
2 ^l	Graphical declaration in LD language	<pre> +-----+ ACTION_4 +-----+ bvar1 bvar2 S8.X bOut1 +---+ -----+ -----+ -----+ +-----+ +-----+ bvar2 +---+ EN ENO C-- LT -----+ (S)-----+ D-- +-----+ +-----+ </pre>

2s	Inclusion of SFC elements in action	<pre> +-----+ OPEN_VALVE_1 +-----+ ... +-----+ VALVE_1_READY +-----+ STEP8.X +-----+ +-----+ VALVE_1_OPENING -- N VALVE_1_FWD +-----+ +-----+ ... +-----+ </pre>
2f	Graphical declaration in FBD language	<pre> +-----+ ACTION_4 +-----+ +---+ bvar1-- & bvar2-- -- bOut1 S8.X----- +---+ FF28 +---+ SR +-----+ Q1 - bOut2 C-- LT -- S1 D-- +-----+ +---+ +-----+ </pre>
3s	Textual declaration in ST language	<pre> ACTION ACTION_4: bOut1:= bvar1 & bvar2 & S8.X; FF28(S1:= (C<D)); bOut2:= FF28.Q; END_ACTION </pre>
3i	Textual declaration in IL language	<pre> 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 </pre>
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

No.	Description	Example
1	Action block physically or logically adjacent to the step	<pre> +-----+ +-----+-----+-----+ S8 -- L ACTION_1 DN1 +-----+ t#10s +-----+-----+-----+ + DN1 </pre>
2	Concatenated action blocks physically or logically adjacent to the step	<pre> +-----+ +-----+-----+-----+ S8 -- L ACTION_1 DN1 +-----+ t#10s +-----+-----+-----+ +DN1 P ACTION_2 +-----+-----+-----+ N ACTION_3 +-----+-----+-----+ </pre>
3	Textual step body	<pre> STEP S8: ACTION_1 (L,t#10s,DN1); ACTION_2 (P); ACTION_3 (N); END_STEP </pre>
4 ^a	Action block "d" field	<pre> +-----+-----+-----+-----+ ---- N ACTION_4 ---- +-----+-----+-----+-----+ bOut1:= bvar1 & bvar2 & S8.X; FF28 (S1:= (C<D)); bOut2:= FF28.Q; +-----+-----+-----+-----+ </pre>
When feature 4 is used, the corresponding action name cannot be used in any other action block.		

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

No.	Description	Graphical form/example
1 ^a	"a": Qualifier as per 6.7.4.5	<pre> +-----+-----+-----+ --- "a" "b" "c" --- +-----+-----+-----+ "d" +-----+-----+-----+ </pre>
2	"b": Action name	
3 ^b	"c": Boolean "indicator" variables (deprecated)	
	"d": Action using:	
4i	IL language	
4s	ST language	
4l	LD language	
4f	FBD language	
5l	Use of action blocks LD	<pre> S8.X bIn1 +---+-----+-----+ OK1 +--- --- --- N ACT1 DN1 ---()---+ +---+-----+-----+ </pre>
5f	Use of action blocks in FBD	<pre> +---+ +---+-----+-----+ S8.X --- & --- N ACT1 DN1 ---OK1 bIn1 --- +---+-----+-----+ +---+ </pre>
Field "a" can be omitted when the qualifier is "N".		
Field "c" can be omitted when no indicator variable is used.		

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	Non-stored	N
3	overriding Reset	R
4	Set (Stored)	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`.

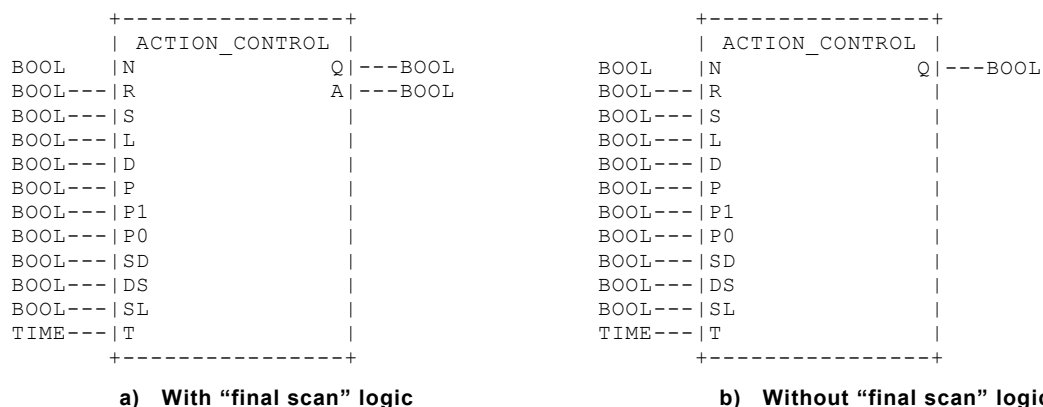
NOTE 4 Access to the functional equivalent of the `Q` or `A` outputs of an `ACTION_CONTROL` function block from outside of the associated action is an Implementer specific feature.

- 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.

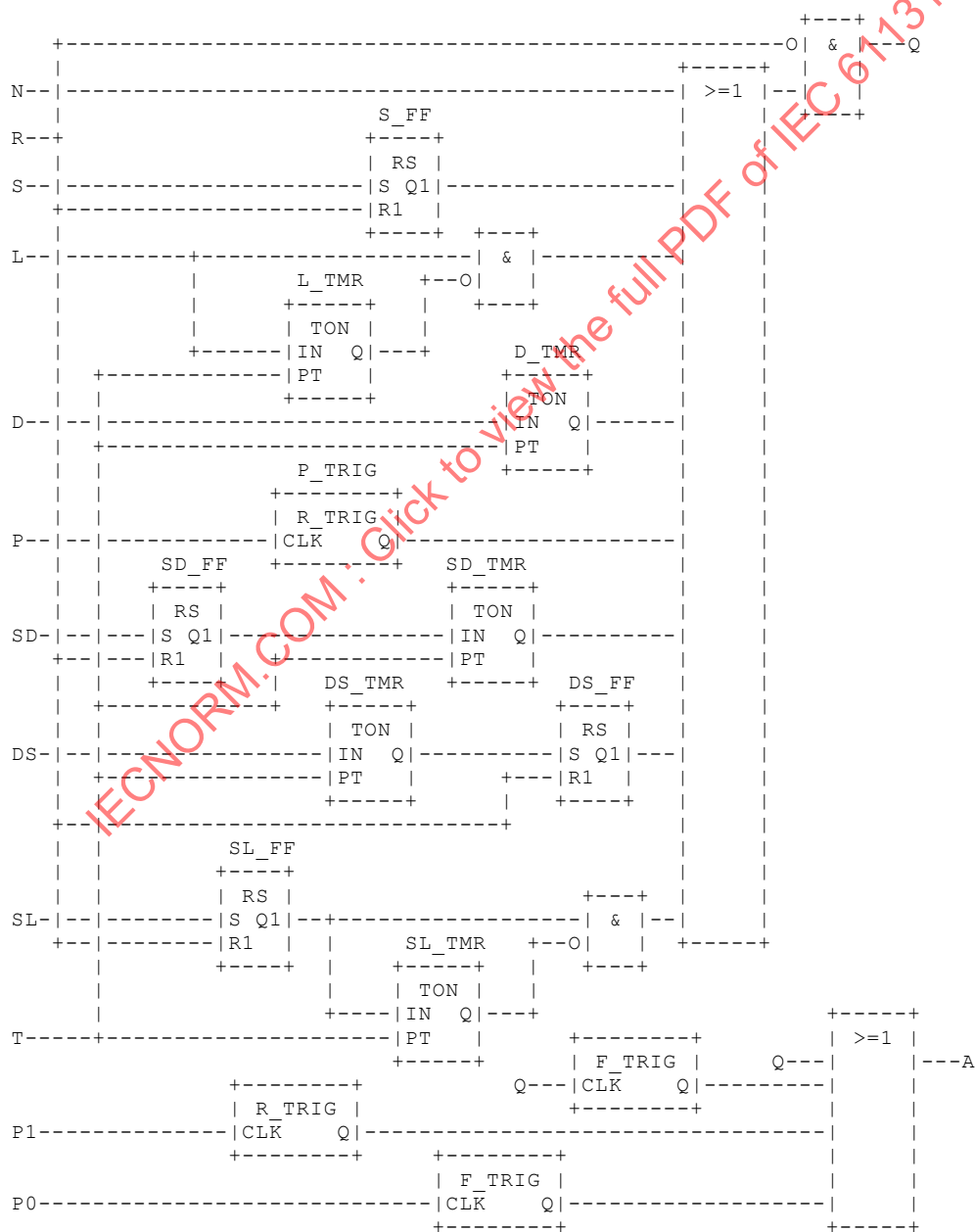


a) With “final scan” logic

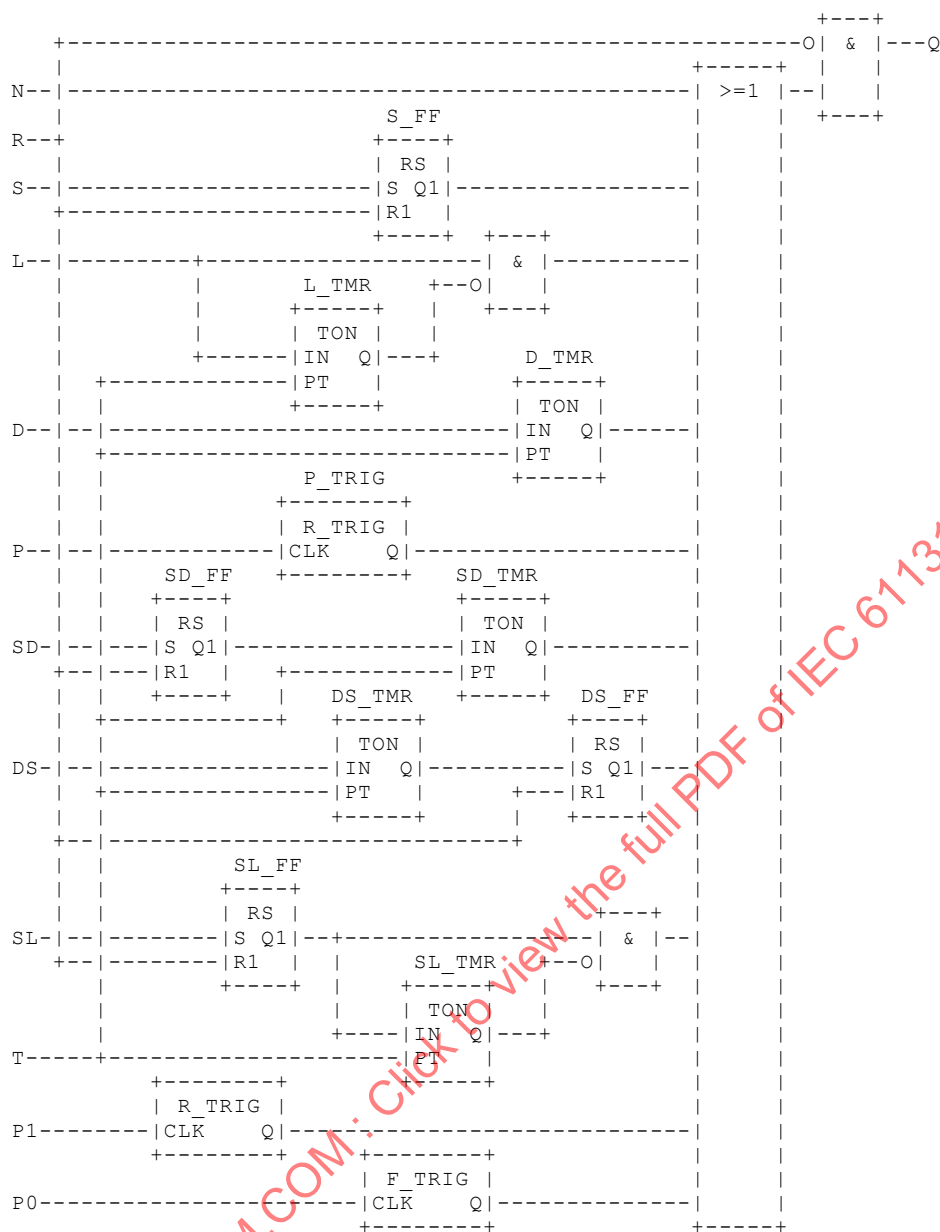
b) Without “final scan” logic

NOTE These interfaces are not visible to the user.

Figure 22 – ACTION_CONTROL function block – External interface (Summary)



a) Body with “final scan” logic



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-----+-----+
S27.X-----R1-----RUNUP_MONITOR_SL_TMR +--O| & |--RUNUP_MONITOR
+-----+ +-----+
| 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.

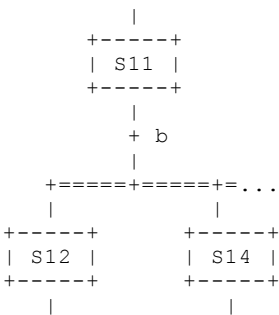
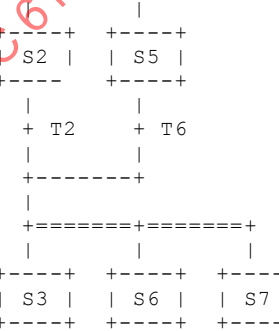
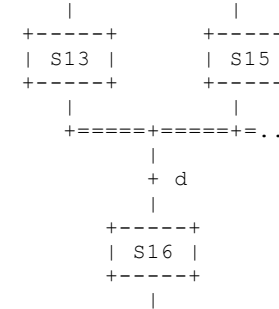
The application of the rules given in this subclause cannot prevent the formulation of “unsafe” SFCs, such as the one shown in Figure 26 a), which may exhibit uncontrolled proliferation of tokens. Likewise, the application of these rules cannot prevent the formulation of “unreachable” SFCs, such as the 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.

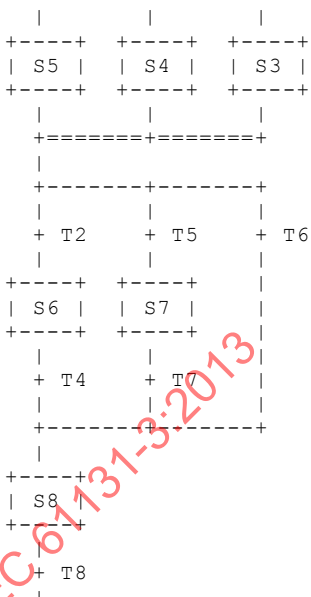
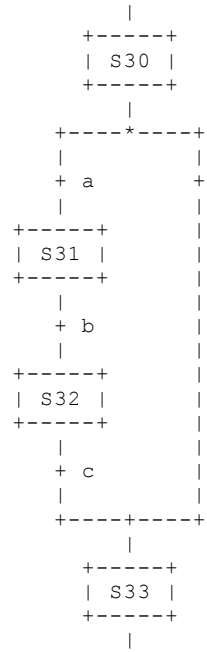
The maximum allowed widths of the “divergence” and “convergence” constructs in Table 61 are Implementer specific.

Table 61 – Sequence evolution – graphical

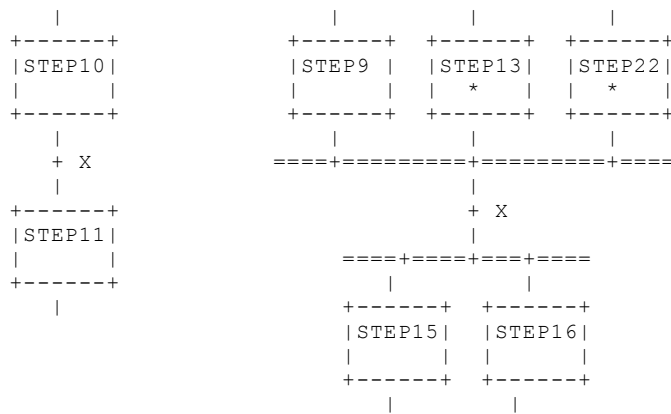
No.	Description	Explanation	Example
1	Single sequence	The alternation step-transition is repeated in series.	<pre> +-----+ S3 +-----+ + c +-----+ S4 +-----+ </pre> An evolution from step S3 to step S4 takes place if and only if step S3 is in the active state and the transition condition c is TRUE
2a	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.	<pre> +-----+ S5 +-----+ +-----*-----+... + e + f +-----+ +-----+ S6 S8 +-----+ +-----+ </pre> 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

No.	Description	Explanation	Example
2b	Divergence of sequence with numbered branches	The asterisk (" * "), followed by numbered branches, indicates a user-defined priority of transition evaluation, with the lowest-numbered branch having the highest priority.	<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.
2c	Divergence of sequence with mutual exclusion	The connection (" + ") of the branch indicates that the user shall assure that transition conditions are mutually exclusive.	<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.
3	Convergence of sequence	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.	<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.

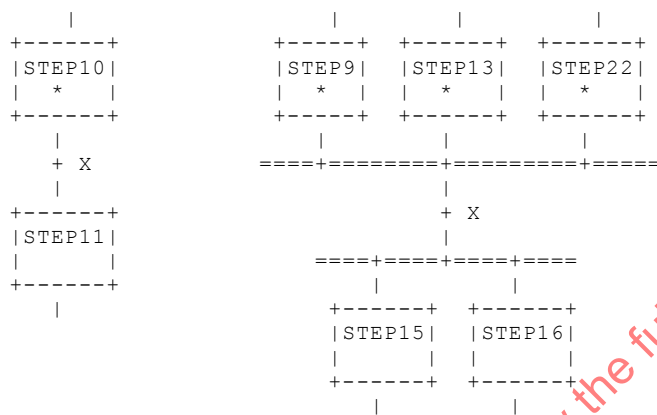
No.	Description	Explanation	Example
4a	Simultaneous divergence after a single transition	The double horizontal line of synchronization can be preceded by a single transition condition.	 An evolution takes place from S11 to S12, S14, ..., if S11 is active and the transition condition b associated to the common transition is TRUE. After the simultaneous activation of S12, S14, etc., the evolution of each sequence proceeds independently.
4b	Simultaneous divergence after conversion	The double horizontal line of synchronization can be preceded by a sequence selection convergence.	 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.
4c	Simultaneous convergence before one transition	Double lines of simultaneous convergence can be followed by a single transition.	 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.

No.	Description	Explanation	Example
4d	Simultaneous convergence before a sequence selection	Double lines of simultaneous convergence can be followed by a sequence selection divergence.	 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.

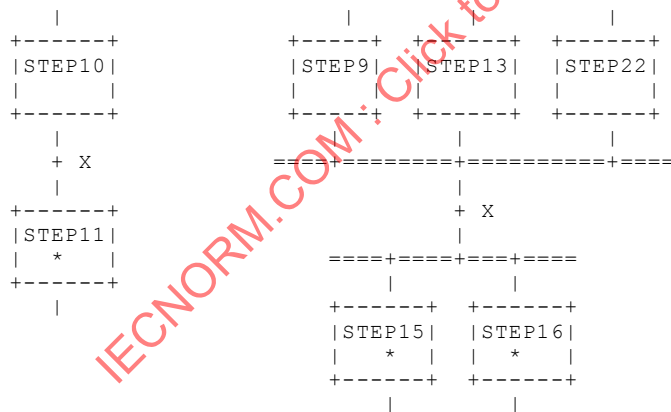
No.	Description	Explanation	Example
6a, b, c	Sequence loop	A “sequence loop” is a special case of sequence selection (feature 2) in which one or more of the branches return to a preceding step. Features 6a, 6b, and 6c correspond to the representation options given in features 2a, 2b, and 2c, respectively.	<pre> +-----+ S30 +-----+ + a +-----+ S31 +-----+ + b +-----+ S32 +-----+ + c + d +-----+ +-----+ S33 +-----+ </pre> (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.
7	Directional arrows	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.	<pre> +-----+ S30 +-----+ + a +-----+ S31 +-----+ + b +-----+ S32 +-----+ *-----+ + c + d +-----+ +-----+ S33 +-----+ </pre>



a) Transition not enabled (NOTE 2)



b) Transition enabled but not cleared (X = 0)

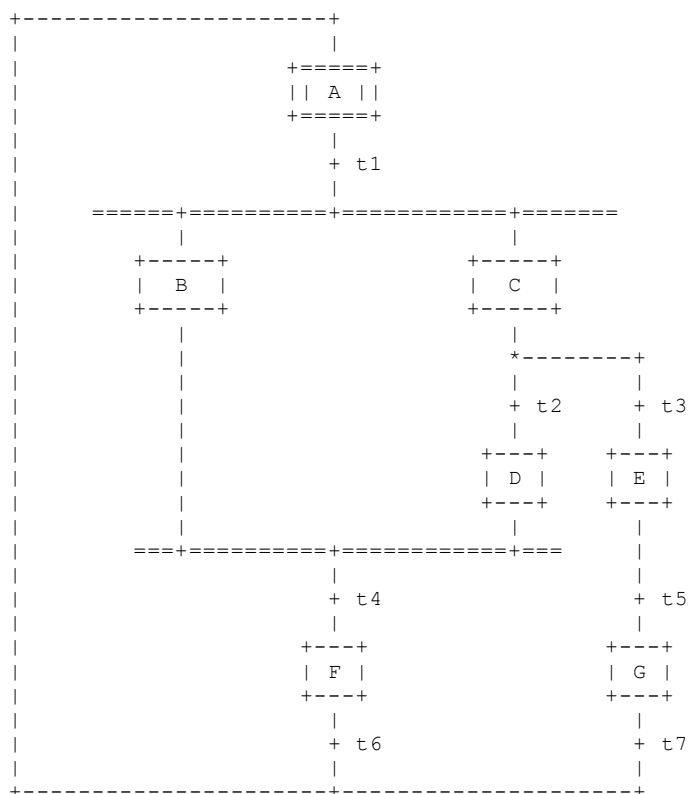


c) Transition cleared (X = 1)

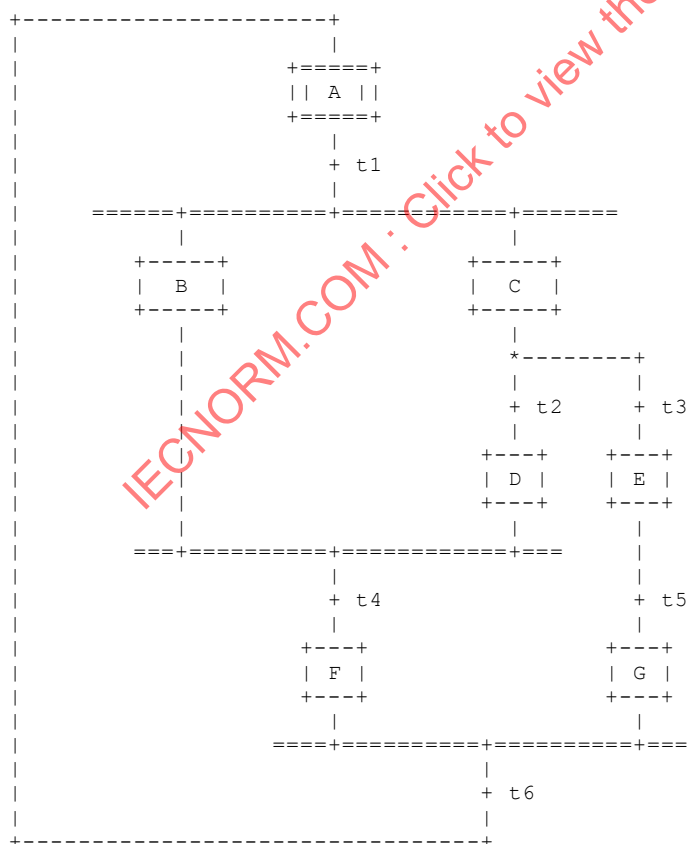
NOTE 1 In this figure, the active state of a step is indicated by the presence of an asterisk (*) in the corresponding block. This notation is used for illustration only, and is not a required language feature.

NOTE 2 In a), the value of the Boolean variable X may be either TRUE or FALSE.

Figure 25 – SFC evolution (Rules)



a) SFC error: an “unsafe” SFC



b) SFC error: an “unreachable” SFC

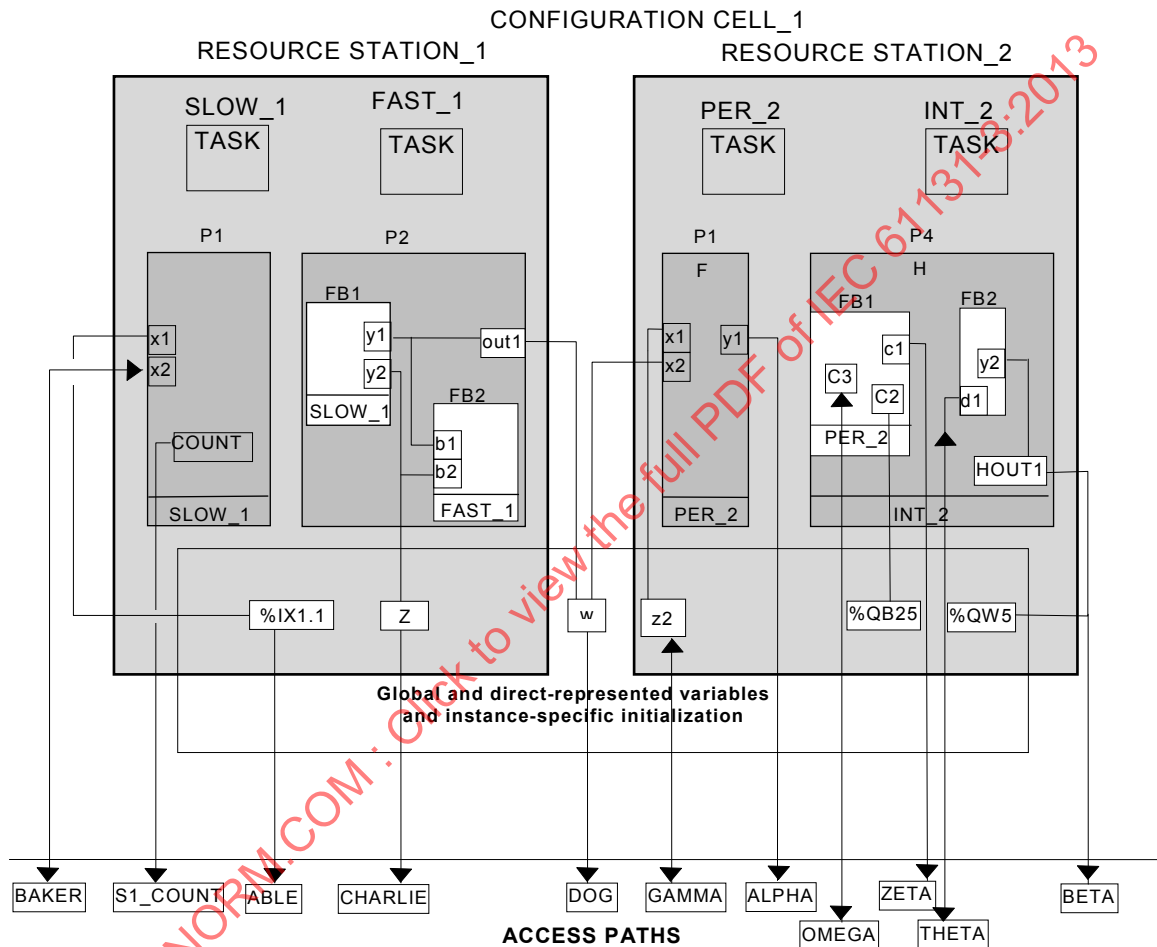
Figure 26 – SFC errors (Example)

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	<code>CONFIGURATION...END_CONFIGURATION</code>
2	<code>VAR_GLOBAL...END_VAR</code> within <code>CONFIGURATION</code>
3	<code>RESOURCE...ON ...END_RESOURCE</code>
4	<code>VAR_GLOBAL...END_VAR</code> within <code>RESOURCE</code>
5a	Periodic <code>TASK</code>
5b	Non-periodic <code>TASK</code>
6a	<code>WITH</code> for <code>PROGRAM</code> to <code>TASK</code> association
6b	<code>WITH</code> for <code>FUNCTION_BLOCK</code> to <code>TASK</code> association
6c	<code>PROGRAM</code> with no <code>TASK</code> 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

No.	Description	Examples
1a	Textual declaration of periodic TASK	(feature 5a of Table 62)
1b	Textual declaration of non-periodic TASK	(feature 5b of Table 62)
	Graphical representation of TASKs (general form)	<pre> TASKNAME +-----+ TASK +-----+ SINGLE +-----+ INTERVAL +-----+ PRIORITY +-----+ </pre>
2a	Graphical representation of periodic TASKs (with INTERVAL)	<pre> SLOW_1 FAST_1 +-----+ +-----+ TASK TASK +-----+ +-----+ SINGLE SINGLE +-----+ +-----+ INTERVAL INTERVAL +-----+ +-----+ PRIORITY PRIORITY +-----+ +-----+ </pre>
2b	Graphical representation of non-periodic TASK (with SINGLE)	<pre> INT_2 +-----+ TASK +-----+ SINGLE +-----+ INTERVAL +-----+ PRIORITY +-----+ </pre>
3a	Textual association with PROGRAMS	(feature 6a of Table 62)
3b	Textual association with function blocks	(feature 6b of Table 62)
4a	Graphical association with PROGRAMS	<pre> RESOURCE STATION_2 P1 P4 +-----+ +-----+ F H +-----+ +-----+ PER_2 INT_2 +-----+ +-----+ END_RESOURCE </pre>
4b	Graphical association with function blocks within PROGRAMS	<pre> P2 +-----+ G +-----+ FB1 FB2 +-----+ +-----+ A B +-----+ +-----+ SLOW_1 FAST_1 +-----+ +-----+ +-----+ END_RESOURCE </pre>
5a	Non-preemptive scheduling	See Figure 28
5b	Preemptive scheduling	See Figure 28

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.

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

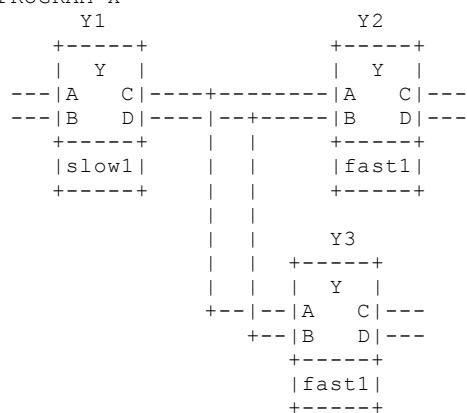
6	P2	
10	P2.FB2@1	P2
12	P2	
16	P2	(P2 restarts)
20	P2.FB2@1	P1@2, P2.FB1@2, P2
- RESOURCE STATION_2 as configured in Figure 28 - Execution times: P1 = 30 ms, P4 = 5 ms, P4.FB1 = 10 ms (NOTE 2) - INT_2 is triggered at t = 25, 50, 90,... ms - STATION_2 starts at t = 0		
Schedule		
t(ms)	Executing	Waiting
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.		

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013

EXAMPLES 2 Task associations to function block instances

RESOURCE R1

PROGRAM X



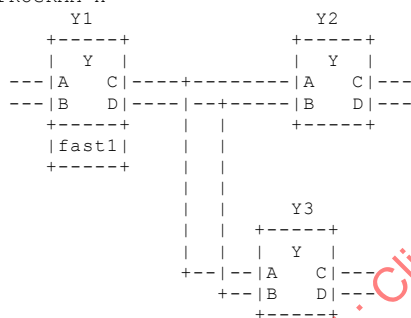
END_PROGRAM

a) Function blocks with explicit task associations



P1

PROGRAM X



END_PROGRAM

slow1

b) Function blocks with implicit task associations

RESOURCE R1



```

P1
PROGRAM X
    Y1
    +-----+
    |  Y  |
    ---|A  C|-----+-----|A  C|---
    ---|B  D|-----+-----|B  D|---
    +-----+
    |fast1|
    +-----+

    Y2
    +-----+
    |  Y  |
    ---|A  C|-----+-----|A  C|---
    ---|B  D|-----+-----|B  D|---
    +-----+
    |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*)
```

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013

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 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.

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013

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)	<pre> NAMESPACE name declaration(s) declaration(s) END_NAMESPACE All containing elements are accessible according to their access specifiers.</pre>
1b	Internal namespace (with <code>INTERNAL</code> specifier)	<pre> 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.</pre>
2	Nested namespaces	See Example 2
3	Variable access specifier <code>INTERNAL</code>	<pre> CLASS C1 VAR INTERNAL myInternalVar: INT; END_VAR VAR PUBLIC myPublicVar: INT; END_VAR END_CLASS</pre>
4	Method access specifier <code>INTERNAL</code>	<pre> CLASS C2 METHOD INTERNAL myInternalMethod: INT; ... END_METHOD METHOD PUBLIC myPublicMethod: INT; ... END_METHOD END_CLASS</pre>
5	Language element with access specifier <code>INTERNAL</code> : User-defined data types using keyword <code>TYPE</code> Functions Function block types Classes Interfaces	<pre> 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</pre>

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	<pre> 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*) </pre>
2	Nested namespace declaration by fully qualified name	<pre> NAMESPACE Standard.Timers.HighResolution FUNCTION PUBLIC TimeResolution: DWORD // ...declaration and operations here END_FUNCTION END_NAMESPACE (*Standard.Timers.HighResolution*) </pre>
3	Mixed lexically nested namespace and namespace nested by fully qualified name	<pre> NAMESPACE Standard.Timers NAMESPACE HighResolution FUNCTION PUBLIC TimeLimit: DWORD // ...declaration and operations here END_FUNCTION END_NAMESPACE (*HighResolution*) END_NAMESPACE (*Standard.Timers*) </pre>
NOTE Multiple namespace declarations with the same fully qualified name contribute to the same namespace. In the examples of this Table the functions <code>TimeTick</code>, <code>TimeResolution</code>, and <code>TimeLimit</code> are members of the same namespace <code>Standard.Timers.HighResolution</code> 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	<pre> USING Standard.Timers; FUNCTION PUBLIC TimeTick: DWORD VAR Ton1: TON; END_VAR // ...declaration and operations here END_FUNCTION </pre>
2	USING in other namespace	<pre> NAMESPACE Standard.Timers.HighResolution USING Counters; FUNCTION PUBLIC TimeResolution: DWORD // ...declaration and operations here END_FUNCTION END_NAMESPACE (*Standard.Timers.HighResolution*) </pre>
3	USING in POUs • Functions • Function block types • Classes • Methods • Interfaces	<pre> 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 </pre>

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.	Description [^]	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: <=
19	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. Addition-

al 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	<pre>CAL C10(%IX10, FALSE, A, OUT, B) CAL CMD_TMR(%IX5, T#300ms, OUT, ELAPSED)</pre>
1b	Function block call with formal parameter list	<pre>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)</pre>
2	Function block call with load/store of standard input parameters	<pre>LD A ADD 5 ST C10.PV LD %IX10 ST C10.CU CAL C10 // FB instance name LD C10.CV // current result</pre>
3a	Function call with formal parameter list	<pre>LIMIT(// Function name EN := COND, IN := B, MN := 1, MX := 5, ENO => TEMPL) ST A // Current result new</pre>
3b	Function call with non-formal parameter list	<pre>LD 1 // set current result LIMIT B, 5 // and use it as IN ST A // new current result</pre>
4a	Method call with formal parameter list	<pre>FB_INST.M1(// Method name EN := COND, IN := B, MN := 1, MX := 5, ENO => TEMPL) ST A // Current result new</pre>
4b	Method call with non-formal parameter list	<pre>LD 1 // set current result FB_INST.M1 B, 5 // and use it as IN ST A // new current result</pre>

No.	Description	Example (NOTE)
	NOTE A declaration such as <pre> VAR C10 : CTU; CMD_TMR : TON; A, B : INT; ELAPSED : TIME; OUT, ERR, TEMPL, COND : BOOL; END_VAR </pre> 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
 $A \leq B$, 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;
2a ^b	Call 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	<pre>IF ... THEN ... ELSIF ... THEN ... ELSE ...END_IF</pre>	<pre>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;</pre>
5	<pre>CASE ... OF ... ELSE ... END_CASE</pre>	<pre>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);</pre>
Iteration		
6	<pre>FOR ... TO ... BY ... DO ... END_FOR</pre>	<pre>J:= 101; FOR I:= 1 TO 100 BY 2 DO IF WORDS[I] = 'KEY' THEN J:= I; EXIT; END_IF; END_FOR;</pre>
7	<pre>WHILE ... DO ... END_WHILE</pre>	<pre>J:= 1; WHILE J <= 100 & WORDS[J] <> 'KEY' DO J:= J+2; END_WHILE;</pre>
8	<pre>REPEAT ... UNTIL ... END_REPEAT</pre>	<pre>J:= -1; REPEAT J:= J+2; UNTIL J = 101 OR WORDS[J] = 'KEY' END_REPEAT;</pre>
9 ^a	CONTINUE	<pre>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;</pre>
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

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013

```

      +-----+
      | x      | myFct |
      |-----| IN    |
      +-----+

```

Uses an operand:
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

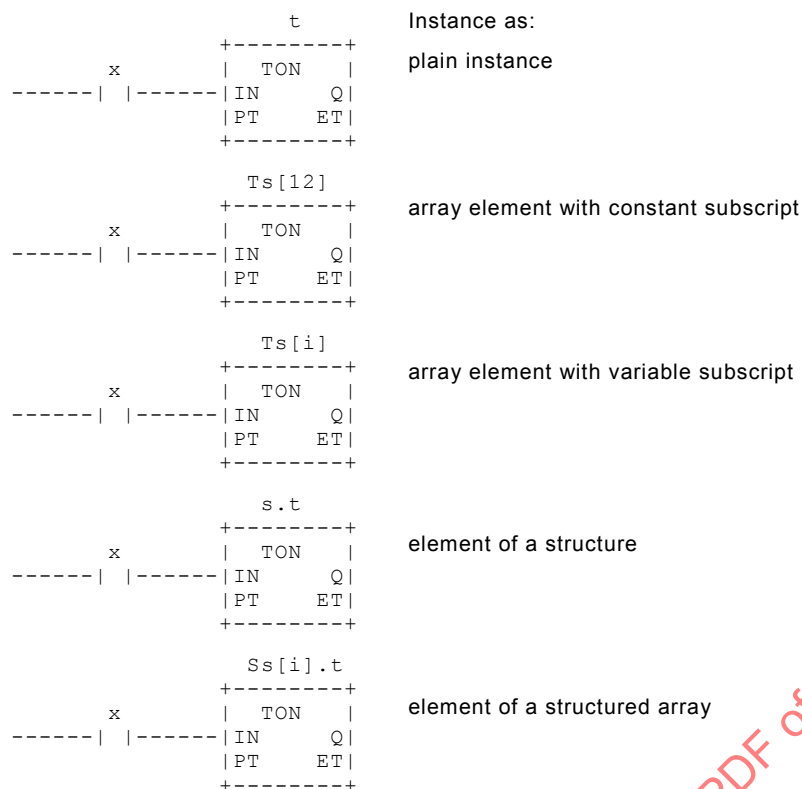
```

      +-----+
      | Ss[2].t  | myFct2 |
      |-----| aTON   |
      +-----+

```

as an element of a structured array

c) Representation of an instance as parameter



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

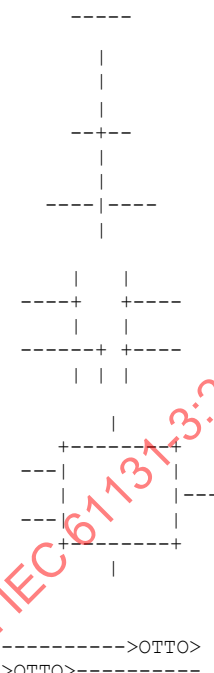
Horizontal/vertical connection (node)

Line crossings without connection (no node)

Connected and non-connected corners (nodes)

Blocks with connecting lines

Connectors and continuation



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

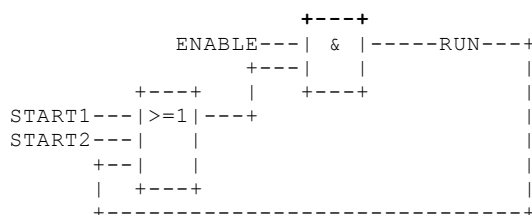
A feedback path is said to exist in a network when the output of a function or function block is used as the input to a function or function block which precedes it in the network; the associated variable is called a feedback variable.

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.

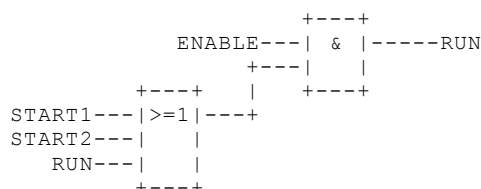
Feedback paths can be utilized in the graphic languages defined, subject to the following rules:

- 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.

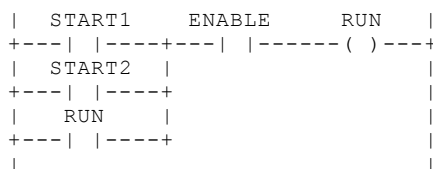
EXAMPLE Feedback path



a) Explicit loop



b) Implicit loop



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

No.	Description	Explanation
Unconditional jump		
1a	FBD language	1---->>LABELA
1b	LD language	<pre> +---->>LABELA </pre>
Conditional jump		
2a	FBD language	Example: jump condition, jump target <pre> X---->>LABELB +---+ bvar0--- & ---->>NEXT bvar50-- +---+ NEXT: +---+ bvar5--- >=1 ---bOut0 bvar60-- +---+ </pre>
2b	LD language	Example: jump condition, jump target <pre> X +-- ---->>LABELB bvar0 bvar50 +--- ----- ---->>NEXT NEXT: bvar5 bOut0 +--- -----+---()---+ bvar60 +--- -----+ </pre>
Conditional return		
3a	LD language	<pre> X +-- ----<RETURN> </pre>
3b	FBD language	X---<RETURN>
Unconditional return		
4	LD language	<pre> +---<RETURN> </pre>

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

No.	Description	Symbol
1	Left power rail (with attached horizontal link)	<pre> +--- </pre>
2	Right power rail (with attached horizontal link)	<pre> --- + </pre>
3	Horizontal link	-----
4	Vertical link (with attached horizontal links)	<pre> ----+---- ----+ +---- </pre>

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 modi-

fy the value of the associated Boolean variable. Standard contact symbols are given in Table 75.

Table 75 – Contacts

No.	Description	Explanation, Symbol
Static contacts		
1	Normally open contact	*** -- -- 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.
2	Normally closed contact	*** -- / -- 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.
Transition-sensing contacts		
3	Positive transition-sensing contact	*** -- 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.
4	Negative transition-sensing contact	*** -- 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.
5a	Compare contact (typed)	<operand 1> --- <cmp> --- DT --- <operand 2> 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: intvalue1 --- > --- Int intvalue2 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

No.	Description	Explanation, Symbol
5b	Compare contact, (overloaded)	$\begin{array}{c} \text{<operand 1>} \\ \\ \text{--- <cmp> ---} \\ \\ \text{<operand 2>} \end{array}$ 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: $\begin{array}{c} \text{value1} \\ \\ \text{--- <> ---} \\ \\ \text{value2} \end{array}$ 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.

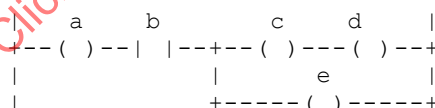


Table 76 – Coils

No.	Description	Explanation, Symbol
	Momentary coils	
1	Coil	$\begin{array}{c} *** \\ \text{--- () ---} \end{array}$ The state of the left link is copied to the associated Boolean variable and to the right link.
2	Negated coil	$\begin{array}{c} *** \\ \text{--- (/) ---} \end{array}$ 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:

- 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.
- 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

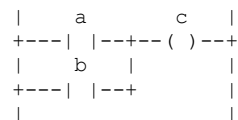
Subclause 8.3 defines FBD, a graphic language for the programming of programmable controllers which is consistent, as far as possible, with IEC 60617-12. Where conflicts exist between this standard and IEC 60617-12, the provisions of this standard shall apply for the programming of programmable controllers in the FBD language.

8.3.2 Combination of elements

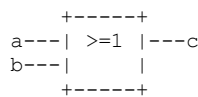
Elements of the FBD language shall be interconnected by signal flow lines following the conventions of 8.1.4.

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) “Wired-OR” in LD language



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.

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013

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' ( 'I' Unsigned_Int 'I' )? | 'WSTRING' ( 'I' Unsigned_Int 'I' )? | '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' 'I' Subrange ( ':' Subrange '*' ) 'J' '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_Spec       : Global_Var_Spec_Init | Loc_Var_Spec | 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_Partially_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;
                  ( Resource_Name ':' )? Global_Var_Name ( ':' Struct_Elem_Name )?;
Global_Var_Access : Identifier;
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 POU's 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*

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013

SOMMAIRE

AVANT-PROPOS	235
1 Domaine d'application	237
2 Références normatives	237
3 Termes et définitions	237
4 Modèles architecturaux	246
4.1 Modèle logiciel	246
4.2 Modèle de communication	248
4.3 Modèle de programmation	249
5 Conformité	251
5.1 Généralités	251
5.2 Tableaux de caractéristiques	252
5.3 Déclaration de conformité de l'Intégrateur	252
6 Eléments communs	254
6.1 Utilisation des caractères d'impression	254
6.1.1 Jeu de caractères	254
6.1.2 Identificateurs	254
6.1.3 Mots-clés	255
6.1.4 Utilisation de l'espace blanc	255
6.1.5 Commentaires	255
6.2 Pragma	256
6.3 Littéraux – représentation externe de données	256
6.3.1 Généralités	256
6.3.2 Littéraux numériques et littéraux de chaîne	256
6.3.3 Littéraux de chaîne de caractères	258
6.3.4 Littéraux de durée	260
6.3.5 Littéraux de date et heure	261
6.4 Types de données	262
6.4.1 Généralités	262
6.4.2 Types de données élémentaires (BOOL, INT, REAL, STRING, etc.)	262
6.4.3 Types de données génériques	264
6.4.4 Types de données définis par l'utilisateur	265
6.5 Variables	279
6.5.1 Déclaration et initialisation de variables	279
6.5.2 Sections de variables	281
6.5.3 Variables <code>ARRAY</code> de longueur variable	283
6.5.4 Variables constantes	285
6.5.5 Variables directement représentées (%)	286
6.5.6 Variables persistantes (RETAIN, NON_RETAIN)	288
6.6 Unités d'organisation de programme (POU)	290
6.6.1 Caractéristiques communes pour les POU	290
6.6.2 Fonctions	302
6.6.3 Blocs fonctionnels	332
6.6.4 Programmes	352
6.6.5 Classes	353

6.6.6	Interface	372
6.6.7	Caractéristiques orientées objet pour les blocs fonctionnels	381
6.6.8	Polymorphisme	387
6.7	Eléments d'un diagramme fonctionnel séquentiel (SFC)	390
6.7.1	Généralités	390
6.7.2	Étapes	391
6.7.3	Transitions	392
6.7.4	Actions	395
6.7.5	Règles d'évolution	404
6.8	Eléments de configuration	412
6.8.1	Généralités	412
6.8.2	Tâches	416
6.9	Espaces de noms	422
6.9.1	Généralités	422
6.9.2	Déclaration	422
6.9.3	Utilisation	427
6.9.4	Directive d'espace de noms USING	428
7	Langages textuels	430
7.1	Eléments communs	430
7.2	Liste d'instructions (IL)	430
7.2.1	Généralités	430
7.2.2	Instructions	430
7.2.3	Opérateurs, modificateurs et opérandes	431
7.2.4	Fonctions et blocs fonctionnels	433
7.3	Texte structuré (ST)	436
7.3.1	Généralités	436
7.3.2	Expressions	436
7.3.3	Enoncés	438
8	Langages graphiques	444
8.1	Eléments communs	444
8.1.1	Généralités	444
8.1.2	Représentation de variables et d'instances	444
8.1.3	Représentation de traits et de blocs	446
8.1.4	Sens du flux dans les réseaux	447
8.1.5	Évaluation des réseaux	448
8.1.6	Eléments de contrôle d'exécution	449
8.2	Diagramme à contacts (LD)	450
8.2.1	Généralités	450
8.2.2	Rails de puissance	451
8.2.3	Eléments de liaison et états	451
8.2.4	Contacts	451
8.2.5	Bobines	453
8.2.6	Fonctions et blocs fonctionnels	454
8.2.7	Ordre d'évaluation des réseaux	454
8.3	Diagramme de bloc fonctionnel (FBD)	454
8.3.1	Généralités	454
8.3.2	Combinaison d'éléments	454
8.3.3	Ordre d'évaluation des réseaux	455
Annexe A (normative)	Spécification formelle des éléments de langage	456

Annexe B (informative) Liste des modifications et extensions majeures de la troisième édition	463
Bibliographie.....	464
Figure 1 – Modèle logiciel	247
Figure 2 – Modèle de communication	249
Figure 3 – Combinaison d'éléments de langage pour automate programmable	251
Figure 4 – Déclaration de conformité de l'Intégrateur (exemple).....	253
Figure 5 – Hiérarchie des types de données génériques	265
Figure 6 – Initialisation par des littéraux et des expressions constantes (règles).....	266
Figure 7 – Mots-clés pour une déclaration de variable (résumé)	282
Figure 8 – Utilisation de <code>VAR_GLOBAL</code> , <code>VAR_EXTERNAL</code> et <code>CONSTANT</code> (règles).....	283
Figure 9 – Conditions associées à la valeur initiale d'une variable (règles).....	289
Figure 10 – Représentation formelle et informelle d'appel (exemples).....	295
Figure 11 – Règles de conversion d'un type de données – implicite et/ou explicite (résumé).....	299
Figure 12 – Conversions de type implicites prises en charge	300
Figure 13 – Utilisation des paramètres d'entrée et de sortie de bloc fonctionnel (règles).....	343
Figure 14 – Utilisation des paramètres d'entrée et de sortie de bloc fonctionnel (illustration des règles)	344
Figure 15 – Blocs fonctionnels normalisés minuteur – diagrammes temporels (règles).....	351
Figure 16 – Présentation de la mise en œuvre d'héritage et d'interface.....	354
Figure 17 – Héritage de classes (illustration)	363
Figure 18 – Interface avec classes dérivées (illustration)	373
Figure 19 – Héritage d'interface et de classe (illustration)	378
Figure 20 – Bloc fonctionnel avec corps et méthodes facultatifs (illustration)	384
Figure 21 – Héritage de corps de bloc fonctionnel avec <code>SUPER()</code> (exemple)	386
Figure 22 – Bloc fonctionnel <code>ACTION_CONTROL</code> – Interface externe (résumé).....	400
Figure 23 – Corps de bloc fonctionnel <code>ACTION_CONTROL</code> (résumé)	402
Figure 24 – Contrôle d'action (exemple).....	404
Figure 25 – Evolution d'un SFC (règles).....	410
Figure 26 – Erreurs d'un SFC (exemple)	411
Figure 27 – Configuration (exemple)	413
Figure 28 – Déclaration de <code>CONFIGURATION</code> et de <code>RESOURCE</code> (exemple)	416
Figure 29 – Accessibilité à l'aide des espaces de noms (règles)	424
Figure 30 – Eléments textuels communs (résumé)	430
Tableau 1 – Jeu de caractères.....	254
Tableau 2 – Identificateurs.....	254
Tableau 3 – Commentaires	256
Tableau 4 – Pragma	256
Tableau 5 – Littéraux numériques	258
Tableau 6 – Littéraux de chaîne de caractères.....	259
Tableau 7 – Combinaisons de deux caractères dans les chaînes de caractères.....	260

Tableau 8 – Littéraux de durée	261
Tableau 9 – Littéraux de date et heure	261
Tableau 10 – Types de données élémentaires	262
Tableau 11 – Déclaration des types de données définis par l'utilisateur et initialisation	266
Tableau 12 – Opérations sur les références	278
Tableau 13 – Déclaration de variables	280
Tableau 14 – Initialisation de variables	281
Tableau 15 – Variables <code>ARRAY</code> de longueur variable	284
Tableau 16 – Variables directement représentées	286
Tableau 17 – Accès partiel aux variables <code>ANY_BIT</code>	292
Tableau 18 – Contrôle de l'exécution en utilisant graphiquement <code>EN</code> et <code>ENO</code>	297
Tableau 19 – Déclaration de fonction	304
Tableau 20 – Appel d'une fonction	306
Tableau 21 – Fonctions typées et en surcharge	308
Tableau 22 – Fonction de conversion de type de données	311
Tableau 23 – Conversion de type de données des types de données numériques	312
Tableau 24 – Conversion de type de données des types de données binaires	315
Tableau 25 – Conversion de type de données des types binaires et numériques	316
Tableau 26 – Conversion de type de données des types date et heure	319
Tableau 27 – Conversion de type de données des types caractère	319
Tableau 28 – Fonctions numériques et arithmétiques	320
Tableau 29 – Fonctions arithmétiques	321
Tableau 30 – Fonctions de décalage de bit	322
Tableau 31 – Fonctions booléennes au niveau du bit	322
Tableau 32 – Fonctions de sélection ^d	323
Tableau 33 – Fonctions de comparaison	324
Tableau 34 – Fonctions de chaîne de caractères	326
Tableau 35 – Fonctions numériques des types de données de temps et de durée	327
Tableau 36 – Fonctions additionnelles des types de données de temps <code>CONCAT</code> et <code>SPLIT</code>	328
Tableau 37 – Fonctions de conversion de boutisme	331
Tableau 38 – Fonctions des types de données énumérés	332
Tableau 39 – Fonctions de validation	332
Tableau 40 – Déclaration du type de bloc fonctionnel	334
Tableau 41 – Déclaration d'instance de bloc fonctionnel	338
Tableau 42 – Appel de bloc fonctionnel	340
Tableau 43 – Blocs fonctionnels normalisés bistables ^a	347
Tableau 44 – Blocs fonctionnels normalisés de détection de front	348
Tableau 45 – Blocs fonctionnels normalisés compteur	348
Tableau 46 – Blocs fonctionnels normalisés minuteur	350
Tableau 47 – Déclaration de programme	352
Tableau 48 – Classe	355
Tableau 49 – Déclaration d'instance de classe	357

Tableau 50 – Appel textuel de méthodes – Liste des paramètres formels et informels	361
Tableau 51 – Interface	372
Tableau 52 – Tentative d'affectation	381
Tableau 53 – Bloc fonctionnel orienté objet.....	382
Tableau 54 – Etape d'un SFC	392
Tableau 55 – Transition et condition de transition d'un SFC.....	394
Tableau 56 – Déclaration des actions d'un SFC	396
Tableau 57 – Association étape/action	397
Tableau 58 – Bloc d'action.....	398
Tableau 59 – Qualificateurs d'action	399
Tableau 60 – Caractéristiques de contrôle d'action	404
Tableau 61 – Evolution de séquence – graphique	405
Tableau 62 – Déclaration de configuration et de ressource	415
Tableau 63 – Tâche	418
Tableau 64 – Espace de noms.....	426
Tableau 65 – Options de déclaration d'espace de nom imbriqué.....	427
Tableau 66 – Directive d'espace de noms USING.....	429
Tableau 67 – Expression entre parenthèses du langage IL	432
Tableau 68 – Opérateurs de liste d'instructions.....	432
Tableau 69 – Appels du langage IL.....	434
Tableau 70 – Opérateurs normalisés de bloc fonctionnel du langage IL	436
Tableau 71 – Opérateurs du langage ST.....	438
Tableau 72 – Enoncés en langage ST.....	439
Tableau 73 – Eléments de contrôle d'exécution graphiques	450
Tableau 74 – Rails de puissance et éléments de liaison	451
Tableau 75 – Contacts	452
Tableau 76 – Bobines	453

COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

AUTOMATES PROGRAMMABLES –

Partie 3: Langages de programmation

AVANT-PROPOS

- 1) La Commission Electrotechnique Internationale (CEI) est une organisation mondiale de normalisation composée de l'ensemble des comités électrotechniques nationaux (Comités nationaux de la CEI). La CEI a pour objet de favoriser la coopération internationale pour toutes les questions de normalisation dans les domaines de l'électricité et de l'électronique. A cet effet, la CEI – entre autres activités – publie des Normes internationales, des Spécifications techniques, des Rapports techniques, des Spécifications accessibles au public (PAS) et des Guides (ci-après dénommés "Publication(s) de la CEI"). Leur élaboration est confiée à des comités d'études, aux travaux desquels tout Comité national intéressé par le sujet traité peut participer. Les organisations internationales, gouvernementales et non gouvernementales, en liaison avec la CEI, participent également aux travaux. La CEI collabore étroitement avec l'Organisation Internationale de Normalisation (ISO), selon des conditions fixées par accord entre les deux organisations.
- 2) Les décisions ou accords officiels de la CEI concernant les questions techniques représentent, dans la mesure du possible, un accord international sur les sujets étudiés, étant donné que les Comités nationaux de la CEI intéressés sont représentés dans chaque comité d'études.
- 3) Les Publications de la CEI se présentent sous la forme de recommandations internationales et sont agréées comme telles par les Comités nationaux de la CEI. Tous les efforts raisonnables sont entrepris afin que la CEI s'assure de l'exactitude du contenu technique de ses publications, la CEI ne peut pas être tenue responsable de l'éventuelle mauvaise utilisation ou interprétation qui en est faite par un quelconque utilisateur final.
- 4) Dans le but d'encourager l'uniformité internationale, les Comités nationaux de la CEI s'engagent, dans toute la mesure possible, à appliquer de façon transparente les Publications de la CEI dans leurs publications nationales et régionales. Toutes divergences entre toutes Publications de la CEI et toutes publications nationales ou régionales correspondantes doivent être indiquées en termes clairs dans ces dernières.
- 5) La CEI elle-même ne fournit aucune attestation de conformité. Des organismes de certification indépendants fournissent des services d'évaluation de conformité et, dans certains secteurs, accèdent aux marques de conformité de la CEI. La CEI n'est responsable d'aucun des services effectués par les organismes de certification indépendants.
- 6) Tous les utilisateurs doivent s'assurer qu'ils sont en possession de la dernière édition de cette publication.
- 7) Aucune responsabilité ne doit être imputée à la CEI, à ses administrateurs, employés, auxiliaires ou mandataires, y compris ses experts particuliers et les membres de ses comités d'études et des Comités nationaux de la CEI, pour tout préjudice causé en cas de dommages corporels et matériels, ou de tout autre dommage de quelque nature que ce soit, directe ou indirecte, ou pour supporter les coûts (y compris les frais de justice) et les dépenses découlant de la publication ou de l'utilisation de cette Publication de la CEI ou de toute autre Publication de la CEI, ou au crédit qui lui est accordé.
- 8) L'attention est attirée sur les références normatives citées dans cette publication. L'utilisation de publications référencées est obligatoire pour une application correcte de la présente publication.
- 9) L'attention est attirée sur le fait que certains des éléments de la présente Publication de la CEI peuvent faire l'objet de droits de brevet. La CEI ne saurait être tenue pour responsable de ne pas avoir identifié de tels droits de brevets et de ne pas avoir signalé leur existence.

La Norme internationale CEI 61131-3 a été établie par le sous-comité 65B: Equipements de mesure et de contrôle-commande, du comité d'études 65 de la CEI: Mesure, commande et automation dans les processus industriels.

Cette troisième édition de la CEI 61131-3 annule et remplace la deuxième édition publiée en 2003. Cette édition constitue une révision technique.

La présente édition inclut les modifications techniques majeures suivantes par rapport à l'édition précédente:

Cette troisième édition est une extension compatible de la deuxième édition. Les principales extensions concernent de nouveaux types de données et de nouvelles fonctions de conversion, des références, des espaces de noms et des classes de caractéristiques orientées objet, et des blocs fonctionnels. Voir Annexe B.

Le texte de cette norme est issu des documents suivants:

FDIS	Rapport de vote
65B/858/FDIS	65B/863/RVD

Le rapport de vote indiqué dans le tableau ci-dessus donne toute information sur le vote ayant abouti à l'approbation de cette norme.

Cette publication a été rédigée selon les Directives ISO/CEI, Partie 2.

Une liste de toutes les parties de la série CEI 61131, publiée sous le titre général *Automates programmables*, peut être consultée sur le site web de la CEI.

Le comité a décidé que le contenu de cette publication ne sera pas modifié avant la date de stabilité indiquée sur le site web de la CEI sous "<http://webstore.iec.ch>" dans les données relatives à la publication recherchée. A cette date, la publication sera

- reconduite,
- supprimée,
- remplacée par une édition révisée, ou
- amendée.

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013

AUTOMATES PROGRAMMABLES –

Partie 3: Langages de programmation

1 Domaine d'application

La présente partie de la CEI 61131 spécifie la syntaxe et la sémantique des langages de programmation utilisés pour les automates programmables tels que définis dans la Partie 1 de la CEI 61131.

Les fonctions d'entrée de programme, d'essai, de surveillance, de système d'exploitation, etc. du système sont spécifiées dans la Partie 1 de la CEI 61131.

La présente partie de la CEI 61131 spécifie la syntaxe et la sémantique d'une suite unifiée de langages de programmation utilisés pour les automates programmables (AP). Cette suite est constituée de deux langages textuels, liste d'instructions (IL, Instruction List) et texte structuré (ST, Structured Text), et de deux langages graphiques, diagramme à contacts (LD, Ladder Diagram) et diagramme de bloc fonctionnel (FBD, Function Block Diagram).

Un autre ensemble d'éléments graphiques et textuels équivalents appelé "diagramme fonctionnel séquentiel" (SFC, Sequential Function Chart) est défini pour structurer l'organisation interne des programmes pour automate programmable et des blocs fonctionnels. En outre, des éléments de configuration qui prennent en charge l'installation des programmes pour automate programmable dans des systèmes d'automate programmable sont définis.

De plus, des caractéristiques sont définies pour faciliter la communication entre les automates programmables et les autres composants des systèmes automatisés.

2 Références normatives

Les documents suivants sont cités en référence de manière normative, en intégralité ou en partie, dans le présent document et sont indispensables pour son application. Pour les références datées, seule l'édition citée s'applique. Pour les références non datées, la dernière édition du document de référence s'applique (y compris les éventuels amendements).

CEI 61131-1, *Automates programmables – Partie 1: Informations générales*

CEI 61131-5, *Automates programmables – Partie 5: Communications*

ISO/CEI 10646:2012, *Technologies de l'information – Jeu universel de caractères codés (JUC)*

ISO/CEI/IEEE 60559, *Information technology – Microprocessor Systems – Floating-Point arithmetic* (disponible en anglais seulement)

3 Termes et définitions

Pour les besoins du présent document, les termes et définitions donnés dans la CEI 61131-1 ainsi que les suivants s'appliquent.

3.1

temps absolu

combinaison d'informations d'heure et de date

3.2

chemin d'accès

association d'un nom symbolique à une variable dans le cadre d'une communication ouverte

3.3

action

variable booléenne ou collection d'opérations à effectuer, conjointement avec une structure de commande associée

3.4

bloc d'action

élément de langage graphique qui utilise une variable d'entrée booléenne pour déterminer la valeur d'une variable de sortie booléenne ou la condition d'activation d'une action, conformément à une structure de commande prédéterminée

3.5

agrégat

collection structurée d'objets de données formant un type de données

[SOURCE: ISO/AFNOR:1989]

3.6

tableau

agrégat constitué d'objets de données ayant des attributs identiques, dont chacun peut être référencé de manière univoque par indigage

[SOURCE: ISO/AFNOR:1989]

3.7

affectation

mécanisme permettant d'attribuer une valeur à une variable ou à un agrégat

[SOURCE: ISO/AFNOR:1989]

3.8

type de base

type de données, type de bloc fonctionnel ou classe à partir desquels d'autres types sont hérités/dérivés

3.9

nombre en base

nombre représenté dans une base spécifiée autre que dix

3.10

décimal codé binaire

BCD

codage de nombres décimaux dans lequel chaque chiffre est représenté par sa propre séquence binaire

3.11

bloc fonctionnel bistable

bloc fonctionnel ayant deux états stables contrôlés par une ou plusieurs entrées

3.12**chaîne de bits**

élément de données constitué d'un ou plusieurs bits

3.13**littéral de chaîne de bits**

littéral qui représente directement une valeur de chaîne de bits de type de données BOOL, BYTE, WORD, DWORD ou LWORD

3.14**corps**

ensemble d'opérations de l'unité d'organisation de programme

3.15**appel**

construction de langage provoquant l'exécution d'une fonction, d'un bloc fonctionnel ou d'une méthode

3.16**chaîne de caractères**

agrégat constitué d'une séquence ordonnée de caractères

3.17**littéral de chaîne de caractères**

littéral qui représente directement une valeur de caractère ou de chaîne de caractères de type de données CHAR, WCHAR, STRING ou WSTRING

3.18**classe**

unité d'organisation de programme constituée:

- de la définition d'une structure de données;
- d'un ensemble de méthodes à effectuer sur la structure de données

3.19**commentaire**

construction de langage permettant d'insérer dans un programme des textes quelconques, sans incidence sur l'exécution du programme

[SOURCE: ISO/AFNOR:1989]

3.20**configuration**

élément de langage correspondant à un système d'automate programmable

3.21**constante**

élément de langage qui déclare un élément de données avec une valeur fixe

3.22**bloc fonctionnel compteur**

bloc fonctionnel qui accumule une valeur correspondant au nombre de modifications détectées à une ou plusieurs entrées spécifiées

3.23**type de données**

ensemble de valeurs associé à un ensemble d'opérations permises

[SOURCE: ISO/AFNOR:1989]

3.24

date et heure

date de l'année et heure du jour représentées sous la forme d'un élément de langage unique

3.25

déclaration

mécanisme permettant d'établir la définition d'un élément de langage

3.26

délimiteur

caractère ou combinaison de caractères utilisés pour séparer des éléments de langage de programme

3.27

classe dérivée

classe créée par héritage d'une autre classe

Note 1 à l'article: La classe dérivée est également appelée "classe étendue" ou "classe enfant".

3.28

type de données dérivé

type de données créé à l'aide d'un autre type de données

3.29

type de bloc fonctionnel dérivé

type de bloc fonctionnel créé par héritage d'un autre type de bloc fonctionnel

3.30

représentation directe

moyen de représentation d'une variable dans un programme pour automate programmable à partir duquel une correspondance spécifiée par mise en œuvre à un emplacement physique ou logique peut être directement déterminée

3.31

double mot

élément de données contenant 32 bits

3.32

liaison dynamique

situation dans laquelle l'instance d'un appel de méthode est extraite pendant l'exécution en fonction du type réel d'une instance ou d'une interface

3.33

évaluation

processus d'établissement d'une valeur pour une expression ou une fonction, ou pour les sorties d'un réseau ou d'une instance de bloc fonctionnel, pendant l'exécution d'un programme

3.34

élément de contrôle d'exécution

élément de langage qui contrôle le flux d'exécution d'un programme

3.35

front descendant

passage de 1 à 0 d'une variable booléenne

3.36

fonction

élément de langage qui, lorsqu'il est exécuté, produit généralement un résultat sur des éléments de données et, éventuellement, des variables de sortie additionnelles

3.37**instance de bloc fonctionnel**

instance d'un type de bloc fonctionnel

3.38**type de bloc fonctionnel**

élément de langage constitué:

- de la définition d'une structure de données divisée en variables d'entrée, de sortie et internes; et
- d'un ensemble d'opérations ou de méthodes à effectuer sur les éléments de la structure de données lorsqu'une instance du type de bloc fonctionnel est appelée

3.39**diagramme de bloc fonctionnel**

réseau dans lequel les nœuds sont des instances de bloc fonctionnel, des fonctions ou appels de méthode représentés graphiquement, des variables, des littéraux et des étiquettes

3.40**type de données générique**

type de données qui représente plusieurs types de données

3.41**variable globale**

variable dont la portée est globale

3.42**adressage hiérarchique**

représentation directe d'un élément de données en tant que membre d'une hiérarchie physique ou logique

EXEMPLE Un point dans un module qui est contenu dans une baie, qui est elle-même contenue dans une armoire, etc.

3.43**identificateur**

combinaison de lettres, de chiffres et de caractères de soulignement qui commence par une lettre ou un caractère de soulignement, et qui nomme un élément de langage

3.44**mise en œuvre**

version produit d'un automate programmable ou d'un outil de programmation et de débogage fourni par l'intégrateur

3.45**Intégrateur**

fabricant de l'automate programmable ou de l'outil de programmation et de débogage fourni à l'utilisateur pour programmer une application pour automate programmable

3.46**héritage**

création d'une nouvelle classe, d'un nouveau type de bloc fonctionnel ou d'une nouvelle interface respectivement sur la base d'une classe, d'un type de bloc fonctionnel ou d'une interface existants

3.47**valeur initiale**

valeur affectée à une variable au démarrage du système

3.48

variable d'entrée-sortie

variable utilisée pour fournir une valeur à une unité d'organisation de programme et pour retourner une valeur depuis l'unité d'organisation de programme

3.49

variable d'entrée

variable qui est utilisée pour fournir une valeur à une unité d'organisation de programme autre qu'une classe

3.50

instance

copie individuelle nommée de la structure de données associée à un type de bloc fonctionnel, à une classe ou à un type de programme, qui conserve ses valeurs d'un appel des opérations associées au suivant

3.51

nom d'instance

identificateur associé à une instance spécifique

3.52

instanciation

création d'une instance

3.53

entier

nombre entier qui peut contenir des valeurs positives, nulles et négatives

3.54

littéral entier

littéral qui représente directement une valeur entière

3.55

interface

élément de langage dans le contexte de la programmation orientée objet contenant un ensemble de prototypes de méthode

3.56

mot-clé

unité lexicale qui caractérise un élément de langage

3.57

étiquette

construction de langage nommant une instruction, un réseau ou un groupe de réseaux et comprenant un identificateur

3.58

élément de langage

tout élément identifié par un symbole sur le côté gauche d'une règle de production dans la spécification formelle

3.59

littéral

unité lexicale qui représente directement une valeur

[SOURCE: ISO/AFNOR:1989]

3.60**emplacement logique**

emplacement d'une variable hiérarchiquement adressée dans un schéma qui peut comporter ou non une relation quelconque avec la structure physique des entrées, des sorties et de la mémoire de l'automate programmable

3.61**réel long**

nombre réel représenté par un mot long

3.62**mot long**

élément de données de 64 bits

3.63**méthode**

élément de langage similaire à une fonction qui peut être définie uniquement dans la portée d'un type de bloc fonctionnel et ayant un accès implicite à des variables statiques de l'instance de bloc fonctionnel ou de classe

3.64**prototype de méthode**

élément de langage contenant uniquement la signature d'une méthode

3.65**élément nommé**

élément d'une structure qui est nommé par l'identificateur qui lui est associé

3.66**réseau**

agencement de nœuds et de branches interconnectées

3.67**littéral numérique**

littéral qui représente directement une valeur numérique, c'est-à-dire un littéral entier ou un littéral réel

3.68**opération**

élément de langage qui représente une fonctionnalité élémentaire appartenant à une unité d'organisation de programme ou à une méthode

3.69**opérande**

élément de langage sur lequel une opération est effectuée

3.70**opérateur**

symbole qui représente l'action à effectuer dans une opération

3.71**override**

mot-clé utilisé avec une méthode dans une classe ou un type de bloc fonctionnel dérivés pour une méthode ayant la même signature qu'une méthode de la classe ou du type de bloc fonctionnel de base à l'aide d'un nouveau corps de méthode

3.72

variable de sortie

variable qui est utilisée pour retourner une valeur depuis l'unité d'organisation de programme, sauf s'il s'agit d'une classe

3.73

paramètre

variable qui est utilisée pour fournir une valeur à une unité d'organisation de programme (sous forme de paramètre d'entrée ou d'entrée-sortie) ou variable qui est utilisée pour retourner une valeur depuis une unité d'organisation de programme (sous forme de paramètre de sortie ou d'entrée-sortie)

3.74

référence

donnée définie par l'utilisateur contenant l'adresse de localisation d'une variable ou d'une instance d'un bloc fonctionnel d'un type spécifié

3.75

flux de puissance

flux symbolique d'énergie électrique dans un diagramme à contacts, utilisé pour indiquer la progression d'un algorithme de résolution logique

3.76

pragma

construction de langage permettant d'insérer, dans une unité d'organisation de programme, des textes quelconques pouvant affecter la préparation du programme à exécuter

3.77

programmer

concevoir, écrire et soumettre à l'essai des programmes utilisateur

3.78

unité d'organisation de programme

fonction, bloc fonctionnel, classe ou programme

3.79

littéral réel

littéral représentant directement une valeur de type `REAL` ou `LREAL`

3.80

ressource

élément de langage correspondant à une "fonction de traitement de signal", et à ses "interface homme-machine" et "fonctions d'interface de capteur et d'actionneur", le cas échéant

3.81

résultat

valeur qui est retournée en sortie d'une unité d'organisation de programme

3.82

retour

construction de langage dans une unité d'organisation de programme, indiquant la fin des séquences d'exécution dans l'unité

3.83

front montant

passage de 0 à 1 d'une variable booléenne

3.84**portée**

ensemble d'unités d'organisation de programme dans lequel une déclaration ou une étiquette s'applique

3.85**sémantique**

relations entre les éléments symboliques d'un langage de programmation et leurs définitions, interprétation et utilisation

3.86**représentation semi-graphique**

représentation d'informations graphiques à l'aide d'un ensemble limité de caractères

3.87**signature**

ensemble d'informations définissant de façon non ambiguë l'identité de l'interface des paramètres d'un objet `METHOD`, constitué de son nom et des noms, types et ordre de l'ensemble de ses paramètres (c'est-à-dire, entrées, sorties, variables d'entrée-sortie et type de résultat)

3.88**variable d'élément unique**

variable qui représente un élément de données unique

3.89**variable statique**

variable dont la valeur est stockée d'un appel au suivant

3.90**étape**

situation dans laquelle le comportement d'une unité d'organisation de programme en ce qui concerne ses entrées et sorties suit un ensemble de règles définies par les actions associées de l'étape

3.91**type de données structure**

type de données d'agrégat qui a été déclaré à l'aide d'une déclaration `STRUCT` ou `FUNCTION_BLOCK`

3.92**indixage**

mécanisme permettant de référencer un élément de tableau au moyen d'une référence de tableau et d'une ou plusieurs expressions qui, lorsqu'elles sont évaluées, indiquent la position de l'élément

3.93**tâche**

élément de contrôle d'exécution permettant l'exécution périodique ou déclenchée d'un groupe d'unités d'organisation de programme associées

3.94**littéral temporel**

littéral représentant des données de type `TIME`, `DATE`, `TIME_OF_DAY` ou `DATE_AND_TIME`

3.95**transition**

condition telle que le contrôle passe d'une ou plusieurs étapes précédentes à une ou plusieurs étapes suivantes selon une liaison dirigée

3.96

entier non signé

nombre entier qui peut contenir des valeurs positives et nulles

3.97

littéral entier non signé

littéral entier ne contenant pas de signe plus (+) ou moins (-) en tête

3.98

type de données défini par l'utilisateur

type de données défini par l'utilisateur

EXEMPLE Énumération, tableau ou structure.

3.99

variable

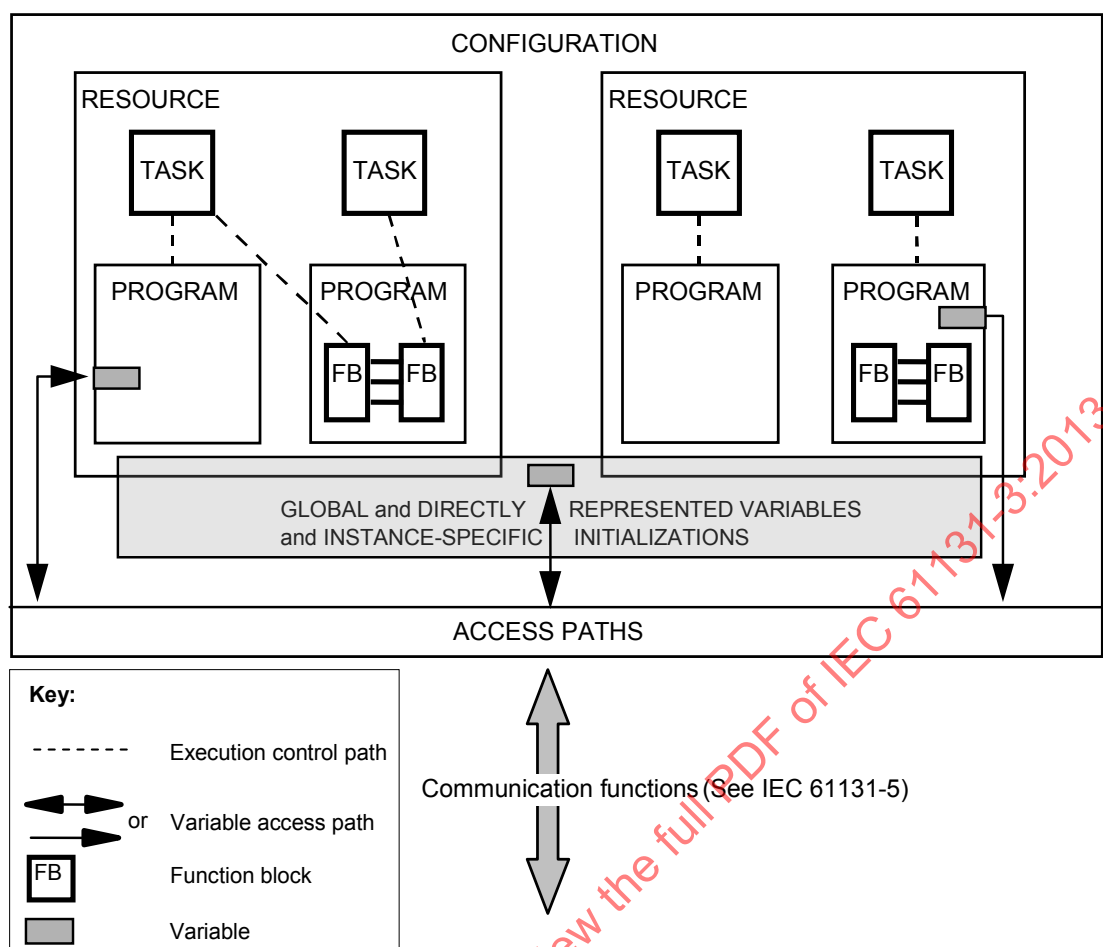
entité logicielle qui peut prendre tour à tour différentes valeurs

4 Modèles architecturaux

4.1 Modèle logiciel

Les éléments de langage de haut niveau basiques et leurs relations mutuelles sont décrits à la Figure 1.

Ces éléments sont des éléments programmés à l'aide des langages définis dans la présente norme, c'est-à-dire des programmes, et des types, des classes, des fonctions et des éléments de configuration de bloc fonctionnel, à savoir des configurations, ressources, tâches, variables globales, chemins d'accès et initialisations spécifiques à une instance, qui permettent l'installation de programmes pour automate programmable dans des systèmes d'automate programmable.



NOTE 1 La Figure 1 est donnée à titre d'illustration uniquement. La représentation graphique n'est pas normative.

NOTE 2 Dans une configuration avec une ressource unique, la ressource peut ne pas être explicitement représentée.

Légende

Anglais	Français
RESOURCE	RESSOURCE
TASK	TACHE
PROGRAM	PROGRAMME
GLOBAL and DIRECTLY REPRESENTED VARIABLES and INSTANCE-SPECIFIC INITIALIZATIONS	VARIABLES GLOBALES et DIRECTEMENT REPRESENTÉES, et INITIALISATIONS SPECIFIQUES A UNE INSTANCE
ACCESS PATHS	CHEMINS D'ACCES
Key:	Légende:
Execution control path	Chemin de contrôle d'exécution
or	ou
Variable access path	Chemin d'accès de variable
Function block	Bloc fonctionnel
Communication functions (See IEC 61131-5)	Fonctions de communication (voir CEI 61131-5)

Figure 1 – Modèle logiciel

Une configuration est l'élément de langage qui correspond à un système d'automate programmable tel que défini dans la CEI 61131-1. Une ressource correspond à une "fonction de

traitement de signal", et ses fonctions "interface homme-machine" et "interface de capteur et d'actionneur" (le cas échéant) comme défini dans la CEI 61131-1.

Une configuration contient une ou plusieurs ressources, dont chacune contient un ou plusieurs programmes exécutés sous le contrôle de zéro ou plusieurs tâches.

Un programme peut contenir zéro ou plusieurs instances de bloc fonctionnel ou d'autres éléments de langage comme défini dans la présente partie de la CEI 61131.

Une tâche est en mesure de provoquer, par exemple de façon périodique, l'exécution d'un ensemble de programmes et d'instances de bloc fonctionnel.

Des configurations et des ressources peuvent être initiées et arrêtées par l'intermédiaire des fonctions "interface opérateur", "programmation, essai et surveillance" ou "système d'exploitation" définies dans la CEI 61131-1. Le démarrage d'une configuration doit provoquer l'initialisation de ses variables globales, puis le démarrage de toutes ses ressources. Le démarrage d'une ressource doit provoquer l'initialisation de toutes ses variables, puis l'activation de toutes ses tâches. L'arrêt d'une ressource doit provoquer la désactivation de toutes ses tâches tandis que l'arrêt d'une configuration doit provoquer l'arrêt de toutes ses ressources.

Des mécanismes permettant de contrôler les tâches sont définis en 6.8.2, tandis que des mécanismes permettant de démarrer et arrêter des configurations et des ressources par l'intermédiaire de fonctions de communication sont définis dans la CEI 61131-5.

Les programmes, ressources, variables globales, chemins d'accès (et privilèges d'accès correspondants) et configurations peuvent être chargés ou supprimés par la "fonction de communication" définie dans la CEI 61131-1. Le chargement ou la suppression d'une configuration ou d'une ressource doivent être équivalents au chargement ou à la suppression de tous les éléments qu'elle contient.

Les chemins d'accès et les privilèges d'accès correspondants sont définis dans la présente norme.

Le mapping des éléments de langage avec les objets de communication doit être tel que défini dans la CEI 61131-5.

4.2 Modèle de communication

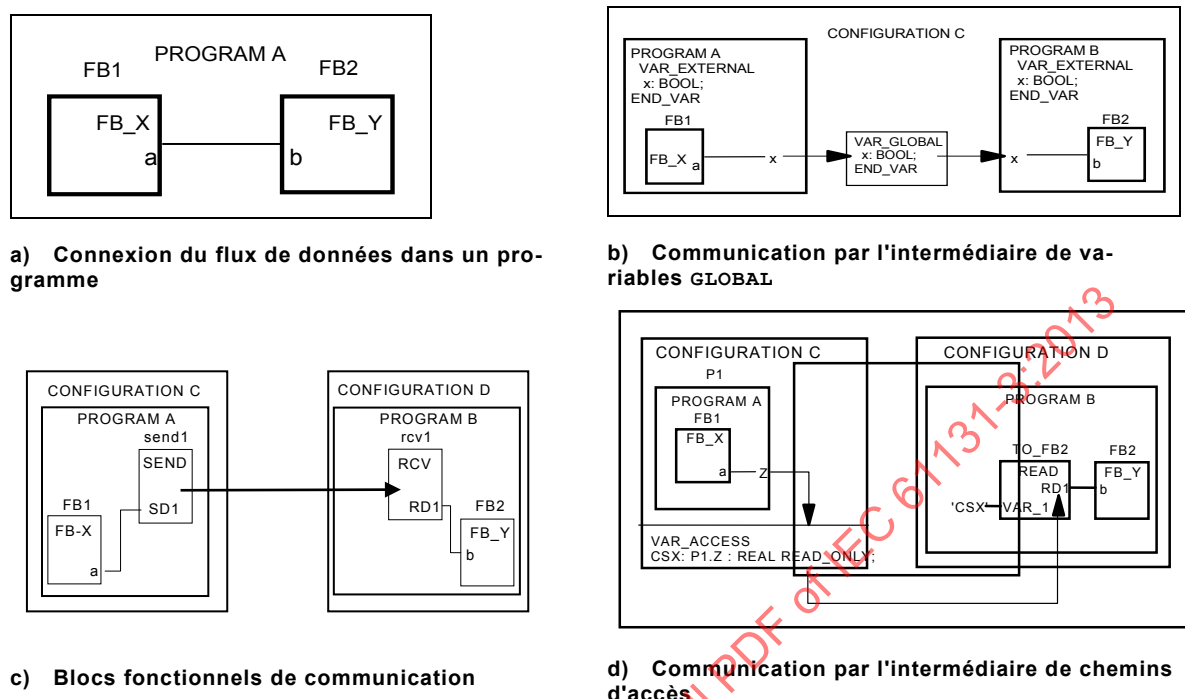
La Figure 2 illustre les façons selon lesquelles les valeurs de variable peuvent être échangées entre des éléments logiciels.

Comme décrit en Figure 2a), les valeurs de variable d'un programme peuvent être directement échangées en reliant la sortie d'un élément de programme à l'entrée d'un autre. Cette connexion est décrite explicitement dans les langages graphiques et implicitement dans les langages textuels.

Des valeurs de variable peuvent être échangées entre des programmes de la même configuration par l'intermédiaire de variables globales telles que la variable `x` décrite en Figure 2b). Ces variables doivent être déclarées `GLOBAL` dans la configuration et `EXTERNAL` dans les programmes.

Comme décrit en Figure 2c), les valeurs de variable peuvent être échangées entre différentes parties d'un programme, entre des programmes de la même configuration ou de configurations différentes, ou entre un programme pour automate programmable et un système d'automate non programmable, à l'aide des blocs fonctionnels de communication définis dans la CEI 61131-5.

De plus, des automates programmables ou des systèmes d'automate non programmable peuvent transférer des données qui sont accessibles par l'intermédiaire de chemins d'accès, comme décrit en Figure 2d), à l'aide des mécanismes définis dans la CEI 61131-5.



NOTE 1 La Figure 2 est donnée à titre d'illustration uniquement. La représentation graphique n'est pas normative.

NOTE 2 Dans ces exemples, les configurations C et D sont considérées comme ayant une seule ressource chacune.

NOTE 3 Les détails des blocs fonctionnels de communication ne sont pas décrits à la Figure 2.

NOTE 4 Des chemins d'accès peuvent être déclarés sur des variables directement représentées, des variables globales, ou des variables d'entrée, de sortie, ou internes de programmes ou d'instances de bloc fonctionnel.

NOTE 5 La CEI 61131-5 spécifie les moyens par lesquels des systèmes AP et non-AP peuvent utiliser des chemins d'accès pour la lecture et l'écriture de variables.

Figure 2 – Modèle de communication

4.3 Modèle de programmation

La Figure 3 donne le résumé des éléments de langage pour automate programmable. La combinaison de ces éléments doit respecter les règles suivantes:

1. Les types de données doivent être déclarés à l'aide des types de données normalisés et de tout type de données précédemment défini.
2. Des fonctions peuvent être déclarées à l'aide de types de données normalisés ou définis par l'utilisateur, des fonctions normalisées et de toute fonction précédemment définie.

Cette déclaration doit utiliser les mécanismes définis pour les langages IL, ST, LD ou FBD.

3. Des types de bloc fonctionnel peuvent être déclarés à l'aide de types de données normalisés ou définis par l'utilisateur, de fonctions, de types de bloc fonctionnel normalisés et de tout type de bloc fonctionnel précédemment défini.

Ces déclarations doivent utiliser les mécanismes définis pour les langages IL, ST, LD ou FBD et peuvent comprendre des éléments SFC.

Des types ou des classes de bloc fonctionnel orientés objet, qui utilisent des méthodes et des interfaces, peuvent éventuellement être définis.

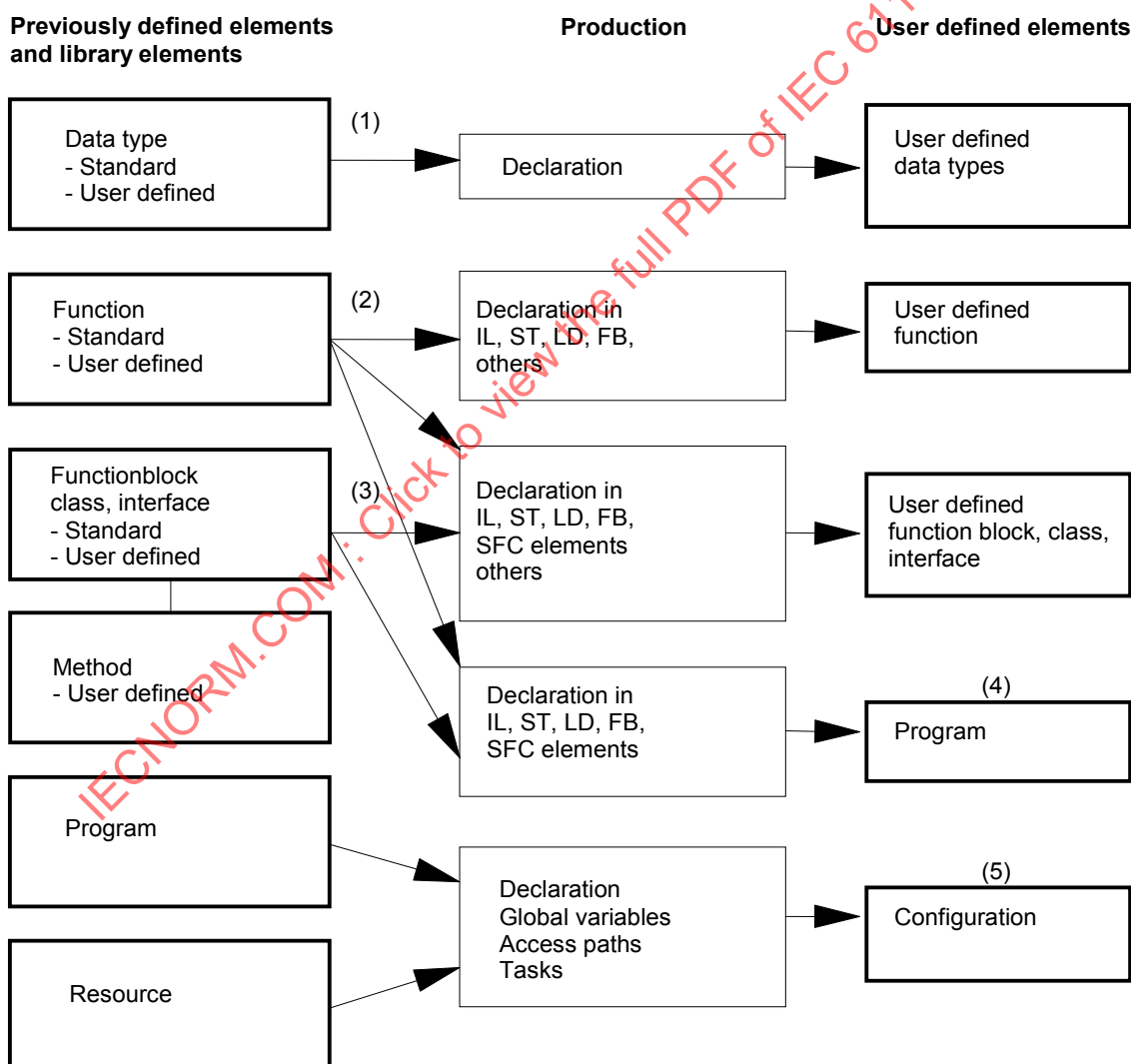
4. Un programme doit être déclaré à l'aide de types de données, de fonctions, de blocs fonctionnels et de classes normalisés ou définis par l'utilisateur.

Cette déclaration doit utiliser les mécanismes définis pour les langages IL, ST, LD ou FBD et peut comprendre des éléments SFC.

5. Des programmes peuvent être combinés dans des configurations à l'aide des éléments, c'est-à-dire des variables globales, des ressources, des tâches et des chemins d'accès.

Toute référence à des types de données, fonctions et blocs fonctionnels "précédemment définis" dans les règles ci-dessus indique qu'une fois qu'un tel élément précédemment défini a été déclaré, sa définition est disponible, par exemple, dans une "bibliothèque" d'éléments précédemment définis, pour utilisation dans d'autres définitions.

Un langage de programmation autre que l'un de ceux définis dans la présente norme peut être utilisé pour programmer une fonction, un type de bloc fonctionnel et des méthodes.



LD: Diagramme à contacts
FBD: Diagramme de bloc fonctionnel
IL: Liste d'instructions
ST: Texte structuré
Autres: Autres langages de programmation

NOTE 1 Les chiffres entre parenthèses (1) à (5) désignent les alinéas correspondants 1) à 5) ci-dessus.

NOTE 2 Les types de données sont utilisés dans toutes les présentations. Par souci de clarté, les liens correspondants sont omis sur cette figure.

Légende

Anglais	Français
Previously defined elements and library elements	Éléments précédemment définis et éléments de bibliothèque
User defined elements	Éléments définis par l'utilisateur
Data type	Type de données
Standard	Normalisé
User defined	Défini par l'utilisateur
Declaration	Déclaration
User defined data types	Types de données définis par l'utilisateur
Function	Fonction
Declaration in IL, ST, LD, FB, others	Déclaration IL, ST, LD, FB, autres
User defined function	Fonction définie par l'utilisateur
Function block, class, interface	Bloc fonctionnel, classe, interface
Declaration in IL, ST, LD, FB, SFC elements, others	Déclaration IL, ST, LD, FB, éléments SFC, autres
User defined function block, class, interface	Bloc fonctionnel, classe, interface définis par l'utilisateur
Method	Méthode
Declaration in IL, ST, LD, FB, SFC elements	Déclaration IL, ST, LD, FB, éléments SFC
Program	Programme
Resource	Ressource
Global variables	Variables globales
Access paths	Chemins d'accès
Tasks	Tâches

Figure 3 – Combinaison d'éléments de langage pour automate programmable

5 Conformité

5.1 Généralités

Un outil de programmation et de débogage (PADT, Programming And Debugging Tool) pour automate programmable, tel que défini dans la CEI 61131-1, qui satisfait, totalement ou partiellement, aux exigences de la présente partie de la CEI 61131 doit présenter cette conformité exclusivement comme décrit ci-dessous.

- Il doit produire un sous-ensemble des caractéristiques et produire la déclaration de conformité correspondante de l'Intégrateur comme défini ci-dessous.
- Il ne doit pas nécessiter l'insertion d'éléments de langage de substitution ou additionnels afin d'exécuter l'une quelconque des caractéristiques.
- Il doit produire un document qui spécifie toutes les extensions spécifiques de l'Intégrateur. Il s'agit des caractéristiques acceptées par le système qui sont prohibées ou non spécifiées.
- Il doit produire un document qui spécifie toutes les dépendances spécifiques de l'Intégrateur. Cela inclut les dépendances de mise en œuvre et les paramètres limitatifs tels que la

longueur, le nombre, la taille ou la plage de valeurs maximum qui n'y sont pas explicitement spécifiés.

- e) Il doit produire un document qui spécifie toutes les erreurs qui sont détectables et rapportées par la mise en œuvre. Cela inclut les erreurs et les erreurs détectables pendant la construction du programme à exécuter et son exécution.

NOTE Les erreurs survenant au cours de l'exécution du programme ne sont que partiellement spécifiées dans la présente partie de la CEI 61131.

- f) Il ne doit utiliser aucun des noms normalisés de types de données et de noms de fonction ou de bloc fonctionnel définis dans la présente norme pour les caractéristiques définies par mise en œuvre dont les fonctionnalités diffèrent de celles décrites dans la présente partie de la CEI 61131.

5.2 Tableaux de caractéristiques

Tous les tableaux de la présente partie de la CEI 61131 sont utilisés de la même façon à des fins bien précises. La première colonne contient le "numéro de caractéristique" et la deuxième donne la "description de la caractéristique"; les colonnes suivantes peuvent comporter des exemples ou des informations complémentaires. Cette structure tabulaire est utilisée dans la déclaration de conformité de l'Intégrateur.

5.3 Déclaration de conformité de l'Intégrateur

L'Intégrateur peut définir un sous-ensemble cohérent de caractéristiques répertoriées dans les tableaux de caractéristiques et doit déclarer le sous-ensemble produit dans la "déclaration de conformité de l'Intégrateur".

La déclaration de conformité de l'Intégrateur doit être incluse dans la documentation jointe au système ou être produite par le système lui-même.

Le format de la déclaration de conformité de l'Intégrateur doit contenir les informations suivantes; un exemple est décrit à la Figure 4:

- les informations générales comprenant le nom et l'adresse de l'Intégrateur, le nom et la version du produit, le type et la version de l'automate, et la date de réalisation;
 - pour chaque caractéristique mise en œuvre, le numéro du tableau de la caractéristique correspondant, le numéro de la caractéristique et le langage de programmation applicable.
- Il peut éventuellement contenir le titre et le sous-titre du tableau de la caractéristique, la description de la caractéristique, des exemples, des notes de l'Intégrateur, etc.

Les tableaux et les caractéristiques qui ne sont pas mis en œuvre peuvent être omis.

CEI 61131-3 "Langages de programmation pour automate programmable"						
Intégrateur: Nom de l'entreprise, adresse, etc. Produit: Nom du produit, version, etc. Sous-ensemble spécifique du type d'auto- mate, etc. Date: 2012-05-01						
Ce produit satisfait aux exigences de la norme pour les caractéristiques de langage suivantes:						
N° de caractéristique	Numéro et titre du tableau / Description de la caractéristique	Conformité de mise en œuvre dans le langage (✓)				Note de l'Intégrateur
		LD	FB D	ST	IL	
	Tableau 1 – Jeu de caractères					
1	ISO/CEI 10646:2012, Technologies de l'information – Jeu universel de caractères codés (JUC)	✓	✓	✓	✓	
2a	Caractères minuscules a: a, b, c, ...	✓	✓	✓		Pas de "ß, ü, ä, ö"
2b	Signe dièse: # Voir Tableau 5	✓				
2c	Signe dollar: \$ Voir Tableau 6		✓			
	Tableau 2 – Identificateurs					
1	Lettres majuscules et chiffres: IW215					
2	Lettres majuscules et minuscules, chiffres, caractère de soulignement incorporé					
3	Lettres majuscules et minuscules, chiffres, caractère de soulignement en tête ou incorporé					
	Tableau 3 – Commentaires					
1	Commentaire d'une ligne //...					
2a	Commentaire de plusieurs lignes (* ... *)					
2b	Commentaire de plusieurs lignes /* ... */					
3a	Commentaire imbriqué (* ..(* .. *) ..*)					
3b	Commentaire imbriqué /* .. /* .. */ .. */					
	Tableau 4 – Pragma					
1	Pragma avec accolades { ... }					
	Tableau 5 – Littéraux numériques					
1	Littéral entier: -12					
2	Littéral réel: -12.0					
3	Littéraux réels avec exposant: -1.34E-12					
4	Littéral binaire: 2#1111_1111					
5	Littéral octal: 8#377					
6	Littéral hexadécimal: 16#FF					
7	Zéro et un booléens					
8	FALSE (FAUX) et TRUE (VRAI) booléens					
9	Littéral typé: INT#-123					
	Etc.					

Figure 4 – Déclaration de conformité de l'Intégrateur (exemple)

6 Éléments communs

6.1 Utilisation des caractères d'impression

6.1.1 Jeu de caractères

Le Tableau 1 décrit le jeu de caractères des langages textuels et des éléments textuels des langages graphiques. Les caractères sont représentés selon les termes de l'ISO/CEI 10646.

Tableau 1 – Jeu de caractères

N°	Description
1	ISO/CEI 10646
2a	Caractères minuscules ^a : a, b, c
2b	Signe dièse: # Voir Tableau 5
2c	Signe dollar: \$ Voir Tableau 6
^a Lorsque des lettres minuscules sont prises en charge, la casse des lettres ne doit pas être significative dans les éléments de langage, sauf dans les commentaires tels que définis en 6.1.5, les littéraux de chaîne tels que définis en 6.3.3 et les variables de type STRING et WSTRING telles que définies en 6.3.3.	

6.1.2 Identificateurs

Un identificateur est une chaîne de lettres, de chiffres et de caractères de soulignement qui doit commencer par une lettre ou un caractère de soulignement.

La casse des lettres ne doit pas être significative dans les identificateurs; par exemple, les identificateurs `abcd`, `ABCD` et `abCd` doivent être interprétés de façon identique.

Le caractère de soulignement doit être significatif dans les identificateurs; par exemple, `A_BCD` et `AB_CD` doivent être interprétés comme des identificateurs différents. Les soulignements en tête ou incorporés multiples ne sont pas autorisés; par exemple, les séquences de caractères `__LIM_SW5` et `LIM__SW5` ne sont pas des identificateurs valides. Les caractères de soulignement de fin ne sont pas autorisés; par exemple, la séquence de caractères `LIM_SW5_` n'est pas un identificateur valide.

Au moins six caractères d'unicité doivent être pris en charge dans tous les systèmes qui prennent en charge l'utilisation d'identificateurs. Par exemple, `ABCDE1` doit être interprété comme étant différent d'`ABCDE2` dans tous ces systèmes. Le nombre maximal de caractères autorisés dans un identificateur est une dépendance spécifique de l'Intégrateur.

Les caractéristiques des identificateurs et des exemples d'identificateurs sont décrits dans le Tableau 2.

Tableau 2 – Identificateurs

N°	Description	Exemples
1	Lettres majuscules et chiffres: IW215	IW215 IW215Z QX75 IDENT
2	Lettres majuscules et minuscules, chiffres, caractère de soulignement incorporé	Tous les éléments ci-dessus plus: LIM_SW_5 LimSw5 abcd ab_Cd
3	Lettres majuscules et minuscules, chiffres, caractère de soulignement en tête ou incorporé	Tous les éléments ci-dessus plus: _MAIN _12V7

6.1.3 Mots-clés

Les mots-clés sont des combinaisons uniques de caractères utilisées comme éléments syntaxiques individuels. Les mots-clés ne doivent pas contenir d'espaces incorporés. La casse des caractères ne doit pas être significative dans des mots-clés. Par exemple, les mots-clés `FOR` et `for` sont équivalents sur le plan syntaxique. Les mots-clés ne doivent être utilisés pour aucune autre application, par exemple, des noms ou des extensions de variable.

6.1.4 Utilisation de l'espace blanc

L'utilisateur doit être autorisé à insérer un ou plusieurs caractères "espace blanc" n'importe où dans le texte des programmes pour automate programmable, sauf dans les mots-clés, littéraux, valeurs énumérées, identificateurs, variables directement représentées ou combinaisons de délimiteurs (par exemple, pour des commentaires). L'"espace blanc" est défini comme étant le caractère `SPACE` ayant la valeur décimale codée 32, ainsi que des caractères non imprimables tels que le caractère de tabulation, de saut de ligne manuel, etc., pour lesquels aucun codage n'est décrit dans la CEI/ISO 10646.

6.1.5 Commentaires

Il existe différents types de commentaires d'utilisateur; ils sont répertoriés dans le Tableau 3:

1. Les commentaires d'une ligne commencent par la combinaison de caractères `//` et se terminent au saut de ligne, saut de ligne manuel, saut de page ou retour chariot suivant.

Dans les commentaires d'une ligne, les combinaisons de caractères spéciaux (`*` et `*`) ou `/*` et `*/` n'ont aucune signification particulière.

2. Les commentaires de plusieurs lignes doivent être délimités au début et à la fin par les combinaisons de caractères spéciaux (`*` et `*`), respectivement.

Un autre commentaire de plusieurs lignes peut être produit à l'aide des combinaisons de caractères spéciaux `/*` et `*/`.

Dans les commentaires de plusieurs lignes, la combinaison de caractères spéciaux `//` n'a aucune signification particulière.

Les commentaires doivent être permis partout dans le programme où des espaces sont autorisés, sauf dans les littéraux de chaîne de caractères.

Les commentaires ne doivent avoir aucune signification syntaxique ou sémantique dans aucun des langages définis dans la présente norme. Ils sont traités comme un espace blanc.

Les commentaires imbriqués utilisent

- des paires correspondantes de (`*`, `*`), par exemple, (`*` ... (`*` IMBRIQUE `*`) ... `*`) ou
- des paires correspondantes de `/*`, `*/`, par exemple, `/*` ... `/*` IMBRIQUE `*/` ... `*/`.

Tableau 3 – Commentaires

N°	Description	Exemples
1	Commentaire d'une ligne avec // ...	X:= 13; // commentaire d'une ligne // un commentaire d'une ligne peut commencer à // la première position de caractère.
2a	Commentaire de plusieurs lignes avec (* ... *)	(* commentaire *) ***** Commentaire encadré de trois lignes *****
2b	Commentaire de plusieurs lignes avec /* ... */	/* commentaire d'une ou plusieurs lignes */
3a	Commentaire imbriqué avec (* .. (* .. *) .. *)	(* (* IMBRIQUE *) *)
3b	Commentaire imbriqué avec /* .. /* .. */ .. */	/* /* IMBRIQUE */ /*

6.2 Pragma

Comme décrit dans le Tableau 4, les pragmas doivent être délimités respectivement au début et à la fin par des accolades { et }. La syntaxe et la sémantique de constructions de pragma particulières sont spécifiques de l'Intégrateur. Les pragmas doivent être permis partout dans le programme où des espaces sont autorisés, sauf dans les littéraux de chaîne de caractères.

Tableau 4 – Pragma

N°	Description	Exemples
1	Pragma avec accolades { ... }	{VERSION 2.0} {AUTEUR JHC} {X:= 256, y:= 384}

6.3 Littéraux – représentation externe de données

6.3.1 Généralités

Les représentations externes de données dans les différents langages de programmation pour automate programmable doivent être constituées de littéraux numériques, de littéraux de chaîne de caractères et de littéraux temporels.

La nécessité de fournir des représentations externes pour deux types distincts de données temporelles est reconnue:

- des données de durée pour mesurer ou contrôler le temps écoulé d'un événement de contrôle,
- et des données d'heure qui peuvent également comprendre des informations de date pour synchroniser le début ou la fin d'un événement de contrôle avec une référence de temps absolu.

6.3.2 Littéraux numériques et littéraux de chaîne

Il existe deux types de littéraux numériques: les littéraux entiers et les littéraux réels. Un littéral numérique est défini comme étant un nombre décimal ou un nombre en base. Le nombre maximal de chiffres pour chaque type de littéral numérique doit être suffisant pour exprimer la

plage entière et la précision des valeurs de tous les types de données qui sont représentés par le littéral dans une mise en œuvre donnée.

Les caractères de soulignement uniques "_" insérés entre les chiffres d'un littéral numérique ne doivent pas être significatifs. Aucune autre utilisation de caractères de soulignement dans les littéraux numériques n'est autorisée.

Les littéraux décimaux doivent être représentés dans la notation décimale conventionnelle. Les littéraux réels doivent se distinguer par la présence d'un point décimal. Un exposant indique la puissance entière de dix par laquelle le nombre précédent doit être multiplié pour obtenir la valeur représentée. Les littéraux décimaux et leurs exposants peuvent être précédés d'un signe "+" ou "-".

Les littéraux peuvent également être représentés en base 2, 8 ou 16. La base doit être en notation décimale. Pour la base 16, un ensemble étendu de chiffres constitués des lettres A à F doit être utilisé, avec la signification conventionnelle de la décimale 10 à 15, respectivement. Les nombres en base ne doivent pas être précédés d'un signe "+" ou "-". Ils sont interprétés comme des littéraux de chaîne de bits.

Les littéraux numériques qui représentent un entier positif peuvent être utilisés en tant que littéraux de chaîne de bits.

Les données booléennes doivent être représentées par des littéraux entiers avec la valeur zéro (0) ou un (1), ou les mots-clés `FALSE` ou `TRUE`, respectivement.

Les caractéristiques des littéraux numériques et des exemples de littéraux numériques sont décrits dans le Tableau 5.

Le type de données d'un littéral booléen ou numérique peut être spécifié en ajoutant un préfixe de type au littéral, constitué du nom d'un type de données élémentaire et du signe "#". Pour obtenir des exemples, voir la caractéristique 9 du Tableau 5.

Tableau 5 – Littéraux numériques

N°	Description	Exemples	Explication
1	Littéral entier	-12, 0, 123_4, +986	
2	Littéral réel	0.0, 0.4560, 3.14159_26	
3	Littéraux réels avec exposant	-1.34E-12, -1.34e-12 1.0E+6, 1.0e+6 1.234E6, 1.234e6	
4	Littéral binaire	2#1111_1111 2#1110_0000	Littéral en base 2 255 décimal 224 décimal
5	Littéraux octaux	8#377 8#340	Littéral en base 8 255 décimal 224 décimal
6	Littéral hexadécimal	16#FF ou 16#ff 16#E0 ou 16#e0	Littéral en base 16 255 décimal 224 décimal
7	Zéro et un booléens	0 ou 1	
8	FALSE (FAUX) et TRUE (VRAI) booléens	FALSE TRUE	
9	Littéral typé	INT#-123	INT représentation de la valeur décimale -123
		INT#16#7FFF	INT représentation de la valeur décimale 32767
		WORD#16#AFF	WORD représentation de la valeur hexadécimale 0AFF
		WORD#1234	WORD représentation de la valeur décimale 1234=16#4D2
		UINT#16#89AF	UINT représentation de la valeur hexadécimale 89AF
		CHAR#16#41	CHAR représentation de "A"
		BOOL#0	
		BOOL#1	
		BOOL#FALSE	
	BOOL#TRUE		

NOTE 1 Les mots-clés FALSE et TRUE correspondent aux valeurs booléennes 0 et 1, respectivement.

NOTE 2 La caractéristique 5 "Littéraux octaux" est déconseillée et peut ne pas être incluse dans la prochaine édition de la présente partie de la CEI 61131.

6.3.3 Littéraux de chaîne de caractères

Les littéraux de chaîne de caractères comprennent des caractères codés sur un octet ou deux octets.

- Un littéral de chaîne de caractères à un octet est une séquence de zéro ou plusieurs caractères préfixée et terminée par le caractère de guillemet simple ('). Dans les chaînes de caractères à un octet, la combinaison de trois caractères composée du signe dollar (\$) suivi de deux chiffres hexadécimaux doit être interprétée comme étant la représentation hexadécimale du code de caractère à huit bits, comme décrit dans la caractéristique 1 du Tableau 6.
- Un littéral de chaîne de caractères à deux octets est une séquence de zéro ou plusieurs caractères du jeu de caractères de l'ISO/CEI 10646 préfixée et terminée par le caractère de guillemet double ("). Dans les chaînes de caractères à deux octets, la combinaison de cinq caractères composée du signe dollar "\$" suivi de quatre chiffres hexadécimaux doit être interprétée comme étant la représentation hexadécimale du code de caractère à seize bits, comme décrit dans la caractéristique 2 du Tableau 6.

NOTE Relation entre l'ISO/CEI 10646 et l'Unicode:

Bien que les codes de caractère et les formes codantes soient synchronisés entre l'Unicode et l'ISO/CEI 10646, la norme Unicode impose des contraintes additionnelles sur les mises en œuvre afin d'assurer qu'elles traitent uniformément les caractères entre les différentes plateformes et applications. A cette fin, elle comprend un ensemble étendu de spécifications fonctionnelles de caractères, données de caractère et algorithmes, et une documentation d'appui substantielle qui ne sont pas présents dans l'ISO/CEI 10646.

Les combinaisons de deux caractères commençant par le signe dollar doivent être interprétées comme décrit dans le Tableau 7 lorsqu'elles sont présentes dans des chaînes de caractères.

Tableau 6 – Littéraux de chaîne de caractères

N°	Description	Exemples
	Caractères ou chaînes de caractères à un octet avec ' '	
1a	Chaîne vide (longueur 0)	' '
1b	Chaîne de longueur 1 ou caractère CHAR contenant un seul caractère	'A'
1c	Chaîne de longueur 1 ou caractère CHAR contenant le caractère "espace"	' '
1d	Chaîne de longueur 1 ou caractère CHAR contenant le caractère "guillemet simple"	'\$' '
1e	Chaîne de longueur 1 ou caractère CHAR contenant le caractère "guillemet double"	'"'
1f	Prise en charge des combinaisons de deux caractères du Tableau 7	'\$R\$L'
1g	Prise en charge d'une représentation de caractère avec "\$" et deux caractères hexadécimaux	'\$0A'
	Caractères ou chaînes de caractères à deux octets avec " " (NOTE)	
2a	Chaîne vide (longueur 0)	" "
2b	Chaîne de longueur 1 ou caractère WCHAR contenant un seul caractère	"A"
2c	Chaîne de longueur 1 ou caractère WCHAR contenant le caractère "espace"	" "
2d	Chaîne de longueur 1 ou caractère WCHAR contenant le caractère "guillemet simple"	"' "
2e	Chaîne de longueur 1 ou caractère WCHAR contenant le caractère "guillemet double"	"\$"
2f	Prise en charge des combinaisons de deux caractères du Tableau 7	"\$R\$L"
2h	Prise en charge d'une représentation de caractère avec "\$" et quatre caractères hexadécimaux	"\$00C4"
	Caractères ou littéraux de chaîne typés à un octet avec #	
3a	Chaîne typée	STRING# 'OK'
3b	Caractère typé	CHAR# 'X'
	Littéraux de chaîne typés à deux octets avec # (NOTE)	
4a	Chaîne typée à deux octets (à l'aide du caractère "guillemet double")	WSTRING# "OK"

N°	Description	Exemples
4b	Caractère typé à deux octets (à l'aide du caractère "guillemet double")	WCHAR#"X"
4c	Chaîne typée à deux octets (à l'aide du caractère "guillemet simple")	WSTRING#'OK'
4d	Caractère typé à deux octets (à l'aide du caractère "guillemet simple")	WCHAR#'X'

NOTE Si une mise en œuvre particulière prend en charge la caractéristique 4 mais pas la caractéristique 2, l'Intégrateur peut spécifier une syntaxe et une sémantique spécifiques de l'Intégrateur pour l'utilisation du caractère de guillemet double.

Tableau 7 – Combinaisons de deux caractères dans les chaînes de caractères

N°	Description	Combinaisons
1	Signe dollar	\$\$
2	Guillemet simple	\$'
3	Saut de ligne	\$L ou \$l
4	Saut de ligne manuel	\$N ou \$n
5	Saut de page	\$P ou \$p
6	Retour chariot	\$R ou \$r
7	Tabulation	\$T ou \$t
8	Guillemet double	\$"

NOTE 1 Le caractère "saut de ligne manuel" constitue un moyen indépendant de la mise en œuvre pour définir la fin d'une ligne de données. Pendant l'impression, il a pour effet de terminer une ligne de données et de reprendre l'impression au début de la ligne suivante.

NOTE 2 La combinaison \$' est valide uniquement à l'intérieur des littéraux de chaîne à guillemets simples.

NOTE 3 La combinaison \$" est valide uniquement à l'intérieur des littéraux de chaîne à guillemets doubles.

6.3.4 Littéraux de durée

Les données de durée doivent être délimitées sur la gauche par le mot-clé T#, TIME# ou LTIME#. La représentation des données de durée en jours, heures, minutes, secondes et fractions de secondes, ou une combinaison quelconque de celles-ci, doit être prise en charge comme décrit dans le Tableau 8. L'unité de temps la moins significative peut être écrite en notation réelle sans exposant.

Les unités des littéraux de durée peuvent être séparées par des caractères de soulignement.

Le "dépassement" de l'unité la plus significative d'un littéral de durée est permis; par exemple, la notation T#25h_15m est autorisée.

Les unités de temps, par exemple, les secondes, les millisecondes, etc., peuvent être représentées en lettres majuscules ou minuscules.

Comme décrit dans le Tableau 8, des valeurs positives et négatives sont autorisées pour les durées.

Tableau 8 – Littéraux de durée

N°	Description	Exemples
	Abréviations de durée	
1a	d	Jour
1b	h	Heure
1c	m	Minute
1d	s	Seconde
1e	ms	Milliseconde
1f	us (μ non disponible)	Microseconde
1g	ns	Nanoseconde
	Littéraux de durée sans caractère de soulignement	
2a	préfixe court	T#14ms T#-14ms LT#14.7s T#14.7m T#14.7h t#14.7d t#25h15m lt#5d14h12m18s3.5ms t#12h4m34ms230us400ns
2b	préfixe long	TIME#14ms TIME#-14ms time#14.7s
	Littéraux de durée avec caractère de soulignement	
3a	préfixe court	t#25h_15m t#5d_14h_12m_18s_3.5ms LTIME#5m_30s_500ms_100.1us
3b	préfixe long	TIME#25h_15m ltime#5d_14h_12m_18s_3.5ms LTIME#34s_345ns

6.3.5 Littéraux de date et heure

Les mots-clés préfixes des littéraux de date et heure doivent être comme décrit dans le Tableau 9.

Tableau 9 – Littéraux de date et heure

N°	Description	Exemples
1a	Littéral de date (préfixe long)	DATE#1984-06-25, date#2010-09-22
1b	Littéral de date (préfixe court)	D#1984-06-25
2a	Littéral de date long (préfixe long)	LDATE#2012-02-29
2b	Littéral de date long (préfixe court)	LD#1984-06-25
3a	Littéral d'heure (préfixe long)	TIME_OF_DAY#15:36:55.36
3b	Littéral d'heure (préfixe court)	TOD#15:36:55.36
4a	Littéral d'heure long (préfixe court)	LTOD#15:36:55.36
4b	Littéral d'heure long (préfixe long)	LTIME_OF_DAY#15:36:55.36
5a	Littéral de date et heure (préfixe long)	DATE_AND_TIME#1984-06-25-15:36:55.360227400
5b	Littéral de date et heure (préfixe court)	DT#1984-06-25-15:36:55.360_227_400
6a	Littéral de date et heure long (préfixe long)	LDATE_AND_TIME#1984-06-25-15:36:55.360_227_400
6b	Littéral de date et heure long (préfixe court)	LDT#1984-06-25-15:36:55.360_227_400

6.4 Types de données

6.4.1 Généralités

Un type de données est une classification qui définit, pour les littéraux et les variables, les valeurs possibles, les opérations qui peuvent être effectuées et la façon dont les valeurs sont stockées.

6.4.2 Types de données élémentaires (BOOL, INT, REAL, STRING, etc.)

6.4.2.1 Spécification des types de données élémentaires

Un ensemble de types de données élémentaires (prédéfinis) est spécifié par la présente norme.

Les types de données élémentaires, mot-clé pour chaque type de données, nombre de bits par élément de données, et plage de valeurs pour chaque type de données élémentaire doivent être tels que décrits dans le Tableau 10.

Tableau 10 – Types de données élémentaires

N°	Description	Mot-clé	Valeur initiale par défaut	N (bits) ^a
1	Booléen	BOOL	0, FALSE	1 ^h
2	Entier court	SINT	0	8 ^c
3	Entier	INT	0	16 ^c
4	Entier double	DINT	0	32 ^c
5	Entier long	LINT	0	64 ^c
6	Entier court non signé	USINT	0	8 ^d
7	Entier non signé	UINT	0	16 ^d
8	Entier double non signé	UDINT	0	32 ^d
9	Entier long non signé	ULINT	0	64 ^d
10	Nombres réels	REAL	0.0	32 ^e
11	Réels longs	LREAL	0.0	64 ^f
12a	Durée	TIME	T#0s	-- ^b
12b	Durée	LTIME	LTIME#0s	64 ^{m, q}
13a	Date (uniquement)	DATE	NOTE	-- ^b
13b	Date longue (uniquement)	LDATE	LDATE#1970-01-01	64 ⁿ
14a	Heure (uniquement)	TIME_OF_DAY ou TOD	TOD#00:00:00	-- ^b
14b	Heure (uniquement)	LTIME_OF_DAY ou LTOD	LTOD#00:00:00	64 ^{o, q}
15a	Date et heure	DATE_AND_TIME ou DT	NOTE	-- ^b
15b	Date et heure	LDATE_AND_TIME ou LDT	LDT#1970-01-01-00:00:00	64 ^{p, q}
16a	Chaîne de caractères à un octet de longueur variable	STRING	' ' (vide)	8 ^{i, g, k, l}
16b	Chaîne de caractères à deux octets de longueur variable	WSTRING	" " (vide)	16 ^{i, g, k, l}
17a	Caractère à un octet	CHAR	'\$00'	8 ^{g, l}
17b	Caractère à deux octets	WCHAR	"\$0000"	16 ^{g, l}
18	Chaîne de bits de longueur 8	BYTE	16#00	8 ^{j, g}
19	Chaîne de bits de longueur 16	WORD	16#0000	16 ^{j, g}

N°	Description	Mot-clé	Valeur initiale par défaut	N (bits) ^a
20	Chaîne de bits de longueur 32	DWORD	16#0000_0000	32 ^j , 9
21	Chaîne de bits de longueur 64	LWORD	16#0000_0000_0000_0000	64 ^j , 9

NOTE Dépend de l'Intégrateur en raison de la date particulière de début qui est différente de 0001-01-01.

^a Les entrées de cette colonne doivent être interprétées comme spécifié dans les notes de bas de tableau.

^b La plage de valeurs et la précision de la représentation dans ces types de données sont spécifiques de l'Intégrateur.

^c La plage de valeurs associée aux variables de ce type de données s'étend de $-(2^{N-1})$ à $(2^{N-1}) - 1$.

^d La plage de valeurs associée aux variables de ce type de données s'étend de 0 à $(2^N) - 1$.

^e La plage de valeurs associée aux variables de ce type de données doit être telle que définie dans la CEI 60559 pour le format de virgule flottante de largeur simple basique. Les résultats des instructions arithmétiques avec des valeurs dénormalisées, l'infini ou des valeurs non numériques sont spécifiques de l'Intégrateur.

^f La plage de valeurs associée aux variables de ce type de données doit être telle que définie dans la CEI 60559 pour le format de virgule flottante de largeur double basique. Les résultats des instructions arithmétiques avec des valeurs dénormalisées, l'infini ou des valeurs non numériques sont spécifiques de l'Intégrateur.

^g Aucune plage de valeurs numériques ne s'applique à ce type de données.

^h Les valeurs possibles des variables de ce type de données doivent être 0 et 1; ces valeurs correspondent aux mots-clés FALSE et TRUE, respectivement.

ⁱ La valeur N indique le nombre de bits/caractère pour ce type de données.

^j La valeur N indique le nombre de bits dans la chaîne de bits pour ce type de données.

^k La longueur maximale autorisée pour les variables STRING et WSTRING est spécifique de l'Intégrateur.

^l Le codage de caractères utilisé pour CHAR, STRING, WCHAR et WSTRING est ISO/CEI 10646 (voir 6.3.3).

^m Le type de données LTIME est un entier signé de 64 bits exprimant des nanosecondes.

ⁿ Le type de données LDATE est un entier signé de 64 bits exprimant des nanosecondes, avec la date de début 1970-01-01.

^o Le type de données LDT est un entier signé de 64 bits exprimant des nanosecondes, avec la date de début 1970-01-01-00:00:00.

^p Le type de données LTOD est un entier signé de 64 bits exprimant des nanosecondes, avec l'heure de début à minuit et TOD#00:00:00.

^q La précision de mise à jour des valeurs de ce format horaire est spécifique de l'Intégrateur, c'est-à-dire que la valeur est donnée en nanosecondes, mais qu'elle peut être mise à jour toutes les microsecondes ou toutes les millisecondes.

6.4.2.2 Types de données de chaîne élémentaires (STRING, WSTRING)

Les longueurs maximales prises en charge pour les éléments de type STRING et WSTRING doivent être des valeurs spécifiques de l'Intégrateur et définir la longueur maximale d'un élément STRING et WSTRING qui est prise en charge par l'outil de programmation et de débogage.

La longueur maximale explicite est spécifiée par une longueur maximale entre parenthèses (qui ne doit pas dépasser la valeur maximale prise en charge spécifique de l'Intégrateur) dans la déclaration associée.

L'accès aux caractères uniques d'une chaîne utilisant des éléments du type de données CHAR ou WCHAR doit être pris en charge à l'aide de crochets et de la position du caractère dans la chaîne, en commençant par la position 1.

L'accès aux chaînes de caractères à deux octets à l'aide de caractères à un octet et l'accès aux chaînes de caractères à un octet à l'aide de caractères à deux octets doivent être considérés comme des erreurs.

EXEMPLE 1 STRING, WSTRING et CHAR, WCHAR

a) Déclaration

VAR

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

END_VAR

b) Utilisation de STRING et CHAR

```
Char1:= String1[2];           //est équivalent à Char1:= 'B';
String1[3]:= Char1;           //donne String1:= 'ABBD '
String1[4]:= 'B';             //donne String1:= 'ABBB'
String1[1]:= String1[4];       //donne String1:= 'BBBB'
String2:= String1[2]; (*donne String2:= 'B'
                       si la conversion implicite CHAR_TO_STRING a été mise en œuvre*)
```

c) Utilisation de WSTRING et WCHAR

```
WChar1:= aWStrings[1][2];      //est équivalent à WChar1:= '6';
aWStrings[1][3]:=WChar1;       //donne aWStrings[1]:= "5668"
aWStrings[1][4]:= "6"; //donne aWStrings[1]:= "5666"
aWStrings[1][1]:= aWStrings[1][4]; //donne String1:= "6666"
aWStrings[0]:= aWStrings[1][4]; (* donne aWStrings[0]:= "6";
                                  si la conversion implicite WCHAR_TO_WSTRING a été mise en œuvre *)
```

d) Fonctions équivalentes (voir 6.6.2.5.11)

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

e) Cas d'erreur

```
Char1:= String1[2]; //combinaison WCHAR, STRING
String1[2]:= String2;
//nécessite la conversion implicite STRING_TO_CHAR, qui n'est pas autorisée
```

NOTE Les types de données associés à des caractères uniques (CHAR et WCHAR) ne peuvent contenir qu'un seul caractère. Les chaînes peuvent contenir plusieurs caractères. Par conséquent, ces dernières peuvent nécessiter des informations de gestion additionnelles qui ne sont pas nécessaires pour les caractères uniques.

EXEMPLE 2

Si le type STR10 est déclaré par

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

la longueur maximale de STR10 est de 10 caractères, la valeur initiale par défaut est 'ABCDEF' et la longueur initiale des éléments de données de type STR10 est de 6 caractères.

6.4.3 Types de données génériques

En plus des types de données élémentaires décrits dans le Tableau 10, la hiérarchie des types de données génériques décrite à la Figure 5 peut être utilisée dans la spécification des entrées et sorties des fonctions et blocs fonctionnels normalisés. Les types de données génériques sont identifiés par le préfixe "ANY".

L'utilisation des types de données génériques est soumise aux règles suivantes:

1. Le type générique d'un type directement dérivé doit être identique au type générique du type élémentaire duquel il est dérivé.
2. Le type générique d'un type d'intervalle doit être ANY_INT.

3. Le type générique de tous les autres types dérivés définis dans le Tableau 11 doit être ANY_DERIVED.

L'utilisation des types de données génériques dans les unités d'organisation de programme déclarées par l'utilisateur ne relève pas du domaine d'application de la présente norme.

Types de données génériques	Types de données génériques	Groupes de types de données élémentaires
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 – Hiérarchie des types de données génériques

6.4.4 Types de données définis par l'utilisateur

6.4.4.1 Déclaration (TYPE)

6.4.4.1.1 Généralités

Les types de données définis par l'utilisateur ont pour vocation d'être utilisés dans la déclaration d'autres types de données et dans les déclarations de variable.

Un type défini par l'utilisateur peut être utilisé partout où un type de base peut être utilisé.

Les types de données définis par l'utilisateur sont déclarés à l'aide de la construction textuelle TYPE...END_TYPE.

Une déclaration de type est constituée des éléments suivants:

- le nom du type,
- un ":" (deux points),
- la déclaration du type lui-même telle que définie dans les articles suivants.

EXEMPLE Déclaration de type

```
TYPE
    myDatatype1: <déclaration de type de données avec initialisation facultative>;
END_TYPE
```

6.4.4.1.2 Initialisation

Les types de données définis par l'utilisateur peuvent être initialisés avec des valeurs définies par l'utilisateur. Cette initialisation est prioritaire sur la valeur initiale par défaut.

L'initialisation définie par l'utilisateur est conforme à la déclaration de type et commence par l'opérateur d'affectation " :=", suivi de la ou des valeurs initiales.

Des littéraux (par exemple, -123, 1.55, "abc") ou des expressions constantes (par exemple, 12*24) peuvent être utilisés. Les valeurs initiales utilisées doivent être d'un type compatible, c'est-à-dire du même type ou d'un type qui peut être converti à l'aide de la conversion de type implicite.

Les règles décrites à la Figure 6 doivent s'appliquer pour l'initialisation de types de données.

Type de données générique	Initialisé par le littéral	Résultat
ANY_UNSIGNED	Littéral entier non négatif ou expression constante non négative	Valeur entière non négative
ANY_SIGNED	Littéral entier ou expression constante	Valeur entière
ANY_REAL	Littéral numérique ou expression constante	Valeur numérique
ANY_BIT	Littéral entier non signé ou expression constante non signée	Valeur entière non signée
ANY_STRING	Littéral de chaîne	Valeur de chaîne
ANY_DATE	Littéral de date et heure	Valeur de date et heure
ANY_DURATION	Littéral de durée	Valeur de durée

Figure 6 – Initialisation par des littéraux et des expressions constantes (règles)

Le Tableau 11 définit les caractéristiques de la déclaration des types de données définis par l'utilisateur et de l'initialisation.

Tableau 11 – Déclaration des types de données définis par l'utilisateur et initialisation

N°	Description	Exemple	Explication
1a 1b	Types de données énumérés	<pre> 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 </pre>	Initialisation
2a 2b	Types de données avec valeurs nommées	<pre> TYPE Couleurs: DWORD (Rouge := 16#00FF0000, Vert:= 16#0000FF00, Bleu := 16#000000FF, Blanc:= Rouge OR Vert OR Bleu, Noir:= Rouge AND Vert AND Bleu) := Vert; END_TYPE </pre>	Initialisation
3a 3b	Types de données d'intervalle	<pre> TYPE ANALOG_DATA: INT(-4095 .. 4095) := 0; END_TYPE </pre>	

N°	Description	Exemple	Explication
4a 4b	Types de données de tableau	<pre> TYPE ANALOG_16_INPUT_DATA: ARRAY [1..16] OF ANALOG_DATA := [8(-4095), 8(4095)]; END_TYPE </pre>	ANALOG_DATA (voir ci-dessus) Initialisation
5a 5b	Types FB et classes en tant qu'éléments de tableau	<pre> TYPE TONs: ARRAY[1..50] OF TON := [50(PT:=T#100ms)]; END_TYPE </pre>	FB TON en tant qu'élément de tableau Initialisation
6a 6b	Type de données structuré	<pre> TYPE ANALOG_CHANNEL_CONFIGURATION: STRUCT RANGE: ANALOG_SIGNAL_RANGE; MIN_SCALE: ANALOG_DATA:= -4095; MAX_SCALE: ANALOG_DATA:= 4095; END_STRUCT; END_TYPE </pre>	ANALOG_SIGNAL_RANGE (voir ci-dessus)
7a 7b	Types FB et classes en tant qu'éléments de structure	<pre> TYPE Refroidisseur: STRUCT Temp: INT; Refroidissement: TOF:= (PT:=T#100ms); END_TYPE </pre>	FB TOF en tant qu'élément de structure
8a 8b	Type de données structuré avec adressage relatif AT	<pre> TYPE Com1_data: STRUCT tête AT %B0: INT; longueur AT %B2: USINT:= 26; drapeau1 AT %X3.0: BOOL; fin AT %B25: BYTE; END_STRUCT; END_TYPE </pre>	Présentation explicite sans chevauchement
9a	Type de données structuré avec adressage relatif AT et OVERLAP	<pre> TYPE Com2_data: STRUCT OVERLAP tête AT %B0: INT; longueur AT %B2: USINT; drapeau2 AT %X3.3: BOOL; donnée1 AT %B5: BYTE; donnée2 AT %B5: REAL; fin AT %B19: BYTE; END_STRUCT; END_TYPE </pre>	Présentation explicite avec chevauchement
10a 10b	Eléments directement représentés d'une structure - partiellement spécifiés à l'aide de " * "	<pre> TYPE HW_COMP: STRUCT; IN AT %I*: BOOL; OUT_VAR AT %Q*: WORD:= 200; ITNL_VAR: REAL:= 123.0; // non localisé END_STRUCT; END_TYPE </pre>	Affecte les composants d'une structure à des entrées et sorties pas encore localisées (voir NOTE 2)
11a 11b	Types de données directement dérivés	<pre> TYPE CNT: UINT; FREQ: REAL:= 50.0; ANALOG_CHANNEL_CONFIG: ANALOG_CHANNEL_CONFIGURATION := (MIN_SCALE:= 0, MAX_SCALE:= 4000); END_TYPE </pre>	Initialisation nouvelle initialisation
12	Initialisation à l'aide d'expressions constantes	<pre> TYPE PIx2: REAL:= 2 * 3.1416; END_TYPE </pre>	Utilise une expression constante

N°	Description	Exemple	Explication
	La déclaration du type de données est possible sans initialisation (caractéristique a) ou avec initialisation (caractéristique b). Si seule la caractéristique (a) est prise en charge, le type de données est initialisé avec la valeur initiale par défaut. Si la caractéristique (b) est prise en charge, le type de données doit être initialisé avec la valeur spécifiée ou la valeur initiale par défaut, si aucune valeur initiale n'est spécifiée.		
	Les variables avec des éléments directement représentés d'une structure, partiellement spécifiés à l'aide de " * ", peuvent ne pas être utilisées dans les sections VAR_INPUT ou VAR_IN_OUT.		

6.4.4.2 Type de données énuméré

6.4.4.2.1 Généralités

La déclaration d'un type de données énuméré spécifie que la valeur d'un élément de données quelconque de ce type ne peut prendre qu'une des valeurs spécifiées dans la liste d'identificateurs associée, comme décrit dans le Tableau 11.

La liste d'énumération définit un ensemble ordonné de valeurs énumérées, qui commence par le premier identificateur de la liste et se termine par le dernier.

Différents types de données énuméré peuvent utiliser les mêmes identificateurs pour des valeurs énumérées. Le nombre maximal autorisé de valeurs énumérées est spécifique de l'Intégrateur.

Pour qu'une identification unique soit possible dans un contexte particulier, les littéraux énumérés peuvent être qualifiés par un préfixe constitué du nom du type de données associé et du caractère dièse '#', de façon similaire aux littéraux typés. Ce préfixe ne doit pas être utilisé dans une liste d'énumération.

Une erreur est générée si des informations suffisantes ne sont pas fournies dans un littéral énuméré pour déterminer sa valeur de façon non ambiguë (voir l'exemple ci-dessous).

EXEMPLE Type de données énuméré

```

TYPE
  Feu_de_circulation: (Rouge, Orange, Vert);
  Couleurs_peinture: (Rouge, Jaune, Vert, Bleu):= Bleu;
END_TYPE

VAR
  Mon_feu_de_circulation: Feu_de_circulation:= Rouge;
END_VAR

IF Mon_Feu_de_circulation = Feu_de_circulation#Orange THEN ...           // OK
IF Mon_Feu_de_circulation = Feu_de_circulation#Rouge THEN ...           // OK
IF Mon_Feu_de_circulation = Orange THEN ...                             // OK - Orange est unique
IF Mon_Feu_de_circulation = Rouge THEN ...                             // ERREUR - Rouge n'est pas unique

```

6.4.4.2.2 Initialisation

La valeur initiale par défaut d'un type de données énuméré doit être le premier identificateur de la liste d'énumération associée.

L'utilisateur peut initialiser le type de données avec une valeur définie par l'utilisateur provenant de la liste de ses valeurs énumérées. Cette initialisation est prioritaire.

Comme décrit dans le Tableau 11 pour ANALOG_SIGNAL_RANGE, la valeur initiale par défaut définie par l'utilisateur du type de données énuméré est la valeur affectée UNIPOLAR_1_5V.

L'affectation de la valeur initiale du type de données définie par l'utilisateur est une caractéristique du Tableau 11.

6.4.4.3 Type de données avec valeurs nommées

6.4.4.3.1 Généralités

Un type de données énuméré avec valeurs nommées est associé au type de données d'énumération (lorsque les valeurs des identificateurs énumérés ne sont pas connues par l'utilisateur). La déclaration spécifie le type de données et affecte les valeurs des valeurs nommées, comme décrit dans le Tableau 11.

La déclaration des valeurs nommées ne limite pas l'utilisation de la plage des valeurs de variable de ces types de données. D'autres constantes peuvent être affectées ou être dérivées par des calculs.

Pour qu'une identification unique soit possible dans un contexte particulier, les valeurs nommées peuvent être qualifiées par un préfixe constitué du nom du type de données associé et du caractère dièse '#', de façon similaire aux littéraux typés.

Ce préfixe ne doit pas être utilisé dans une liste de déclaration. Une erreur est générée si des informations suffisantes ne sont pas fournies dans un littéral énuméré pour déterminer sa valeur de façon non ambiguë (voir l'exemple ci-dessous).

EXEMPLE Type de données avec valeurs nommées

```

TYPE
  Feu_de_circulation:  INT (Rouge:= 1, Orange := 2, Vert:= 3):= Vert;
  Couleurs_peinture:  INT (Rouge:= 1, Jaune:= 2, Vert:= 3, Bleu:= 4):= Bleu;
END_TYPE

VAR
  Mon_feu_de_circulation: Feu_de_circulation;
END_VAR

Mon_feu_de_circulation:= 27;           // Affectation à partir d'une constante
Mon_feu_de_circulation:= Orange + 1;  // Affectation à partir d'une expression
// Note: Cela n'est pas possible pour des valeurs énumérées
Mon_feu_de_circulation:= Feu_de_circulation#Rouge + 1;

IF Mon_feu_de_circulation = 123 THEN ...           // OK
IF Mon_feu_de_circulation = Feu_de_circulation#Orange THEN ... // OK
IF Mon_feu_de_circulation = Feu_de_circulation#Rouge THEN ...  // OK
IF Mon_feu_de_circulation = Orange THEN ...           // OK parce qu'Orange est unique
IF Mon_feu_de_circulation = Rouge THEN ...           // Erreur parce que Rouge n'est pas unique

```

6.4.4.3.2 Initialisation

La valeur par défaut d'un type de données avec valeurs nommées est le premier élément de données de la liste d'énumération. Dans l'exemple ci-dessus, cet élément est Rouge pour Feu_de_circulation.

L'utilisateur peut initialiser le type de données avec une valeur définie par l'utilisateur. L'initialisation n'est pas limitée aux valeurs nommées, n'importe quelle valeur appartenant à la plage du type de données de base peut être utilisée. Cette initialisation est prioritaire.

Dans l'exemple, la valeur initiale définie par l'utilisateur du type de données énuméré pour Feu_de_circulation est Vert.

L'affectation de la valeur initiale du type de données définie par l'utilisateur est une caractéristique du Tableau 11.

6.4.4.4 Type de données d'intervalle

6.4.4.4.1 Généralités

Une déclaration d'intervalle spécifie que la valeur d'un élément de données quelconque de ce type ne peut prendre que des valeurs comprises entre les limites supérieure et inférieure spécifiées incluses, comme décrit dans le Tableau 11.

Les limites d'un intervalle doivent être des littéraux ou des expressions constantes.

EXEMPLE

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

6.4.4.4.2 Initialisation

Les valeurs initiales par défaut associées aux types de données avec intervalle doivent être la première limite (inférieure) de l'intervalle.

L'utilisateur peut initialiser le type de données avec une valeur définie par l'utilisateur provenant de l'intervalle. Cette initialisation est prioritaire.

Par exemple, comme décrit dans le Tableau 11, la valeur initiale par défaut des éléments de type `ANALOG_DATA` est `-4095` tandis qu'avec l'initialisation explicite, la valeur initiale par défaut est zéro (comme déclaré).

6.4.4.5 Type de données de tableau

6.4.4.5.1 Généralités

La déclaration d'un type de données de tableau spécifie qu'une quantité suffisante de stockage de données doit être attribuée pour chaque élément de ce type pour que toutes les données pouvant être indexées par le ou les intervalles d'index spécifiés puissent être stockées, comme décrit dans le Tableau 11.

Un tableau est une collection d'éléments de données du même type de données. Des types de données élémentaires et définis par l'utilisateur, des types de bloc fonctionnel, et des classes peuvent être utilisés en tant que type d'un élément de tableau. Cette collection d'éléments de données est référencée par un ou plusieurs indices encadrés par des parenthèses et séparés par des virgules. Une erreur doit être générée si la valeur d'un indice est en dehors de la plage spécifiée dans la déclaration du tableau.

NOTE Cette erreur peut être détectée uniquement lors de l'exécution pour un index calculé.

Le nombre maximal d'indices d'un tableau, la taille maximale du tableau et la plage maximale des valeurs d'index sont spécifiques de l'Intégrateur.

Les limites du ou des intervalles d'index doivent être des littéraux ou des expressions constantes. Les tableaux de longueur variable sont définis en 6.5.3.

Dans le langage ST, un indice doit être une expression générant une valeur correspondant à un des sous-types du type générique `ANY_INT`.

La forme des indices dans le langage IL et les langages graphiques définis à l'Article 8 est limitée aux variables d'élément unique ou aux littéraux entiers.

EXEMPLE

a) Déclaration d'un tableau

```
VAR myANALOG_16: ARRAY [1..16] OF ANALOG_DATA
    := [8(-4095), 8(4095)];    // valeurs initiales définies par l'utilisateur
END_VAR
```

b) Une utilisation des variables de tableau dans le langage ST pourrait être:

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

6.4.4.5.2 Initialisation

La valeur initiale par défaut de chaque élément d'un tableau est la valeur initiale définie pour le type de données des éléments du tableau.

L'utilisateur peut initialiser un type de tableau avec une valeur définie par l'utilisateur. Cette initialisation est prioritaire.

La valeur initiale définie par l'utilisateur d'un tableau est affectée sous la forme d'une liste qui peut utiliser des parenthèses pour exprimer des répétitions.

Lors de l'initialisation des types de données d'un tableau, la variation de l'indice le plus à droite du tableau doit varier plus rapidement que le remplissage du tableau à partir de la liste des valeurs d'initialisation.

EXEMPLE Initialisation d'un tableau

```
A: ARRAY [0..5] OF INT:= [2(1, 2, 3)]
est équivalent à la séquence d'initialisation 1, 2, 3, 1, 2, 3.
```

Si le nombre de valeurs initiales spécifié dans la liste d'initialisation dépasse le nombre d'entrées du tableau, les valeurs initiales en excès (les plus à droite) doivent être ignorées. Si le nombre de valeurs initiales est inférieur au nombre d'entrées du tableau, les entrées résiduelles du tableau doivent être renseignées avec les valeurs initiales par défaut associées au type de données correspondant. Dans tous les cas, l'utilisateur doit être averti de cette condition pendant la préparation du programme à exécuter.

L'affectation de la valeur initiale du type de données définie par l'utilisateur est une caractéristique du Tableau 11.

6.4.4.6 Type de données structuré**6.4.4.6.1 Généralités**

La déclaration d'un type de données structuré (**STRUCT**) spécifie que ce type de données doit contenir une collection de sous-éléments des types spécifiés, à laquelle on puisse accéder par les noms spécifiés, comme décrit dans le Tableau 11.

Un élément d'un type de données structuré doit être représenté par plusieurs identificateurs ou des accès au tableau séparés par des points ".". Le premier identificateur représente le nom de l'élément structuré et les identificateurs suivants représentent la séquence de noms d'élément permettant d'accéder à l'élément de données particulier dans la structure de données. Des types de données élémentaires et définis par l'utilisateur, des types de bloc fonctionnel, et des classes peuvent être utilisés en tant que type d'un élément de structure.

Par exemple, un élément de type de données **ANALOG_CHANNEL_CONFIGURATION** tel que déclaré dans le Tableau 11 contiendra un sous-élément **RANGE** de type **ANALOG_SIGNAL_RANGE**, un sous-élément **MIN_SCALE** de type **ANALOG_DATA** et un élément **MAX_SCALE** de type **ANALOG_DATA**.

Le nombre maximal d'éléments de structure, la quantité maximale de données qui peuvent être contenues dans une structure, et le nombre maximal de niveaux imbriqués d'adressage d'élément de structure sont spécifiques de l'Intégrateur.

Deux variables structurées ne sont compatibles en termes d'affectation que si elles appartiennent au même type de données.

EXEMPLE Déclaration et utilisation d'un type de données structuré et d'une variable structurée

a) Déclaration d'un type de données structuré

```
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) Déclaration d'une variable structurée

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

c) Utilisation de variables structurées dans le langage ST:

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

6.4.4.6.2 Initialisation

Les valeurs par défaut des composants d'une structure sont définies par leurs types de données individuels.

L'utilisateur peut initialiser les composants de la structure avec des valeurs définies par l'utilisateur. Cette initialisation est prioritaire.

L'utilisateur peut également initialiser une structure précédemment définie en utilisant une liste d'affectations pour les composants de la structure. Cette initialisation a une priorité plus élevée que l'initialisation par défaut et l'initialisation des composants.

EXEMPLE Initialisation d'une structure

a) Déclaration avec initialisation d'un type de données structuré

```

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) Déclaration avec initialisation d'une variable structurée

```

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 Emplacement relatif des éléments des types de données structurés (AT)**6.4.4.7.1 Généralités**

Les emplacements (adresses) des éléments d'un type structuré peuvent être définis par rapport au début de la structure.

Dans ce cas, le nom de chaque composant de cette structure doit être suivi du mot-clé **AT** et d'un emplacement relatif. La déclaration peut contenir des espacements dans la configuration mémoire.

L'emplacement relatif est constitué d'un "%" (pour cent), du qualificateur d'emplacement et d'un emplacement de bit ou d'octet. Un emplacement d'octet est un littéral entier non signé indiquant le décalage d'octet. Un emplacement de bit est constitué d'un décalage d'octet, suivi d'un "." (point), et le décalage de bit est un littéral entier non signé compris entre 0 et 7. Les espaces blancs ne sont pas autorisés dans l'emplacement relatif.

Les composants de la structure ne doivent pas se chevaucher dans leur configuration mémoire, sauf si le mot-clé **OVERLAP** est spécifié dans la déclaration.

Le chevauchement de chaînes ne relève pas du domaine d'application de la présente norme.

NOTE Le nombre des décalages de bit commence à 0 pour le bit le plus à droite. Le nombre des décalages d'octet commence au début de la structure avec un décalage d'octet de 0.

EXEMPLE Emplacement relatif et chevauchement dans une structure

TYPE

```
Com1_data: STRUCT
  tête      AT %B0:      INT;           // à l'emplacement 0
  longueur  AT %B2:      USINT:= 26;    // à l'emplacement 2
  drapeau1  AT %X3.0:    BOOL;          // à l'emplacement 3.0
  fin       AT %B25:     BYTE;          // à 25, en laissant un espace
END_STRUCT;

Com2_data: STRUCT OVERLAP
  tête      AT %B0:      INT;           // à l'emplacement 0
  longueur  AT %B2:      USINT;         // à l'emplacement 2
  drapeau2  AT %X3.3:    BOOL;          // à l'emplacement 3.3
  donnée1   AT %B5:      BYTE;          // à l'emplacement 5, chevauchant
  donnée2   AT %B5:      REAL;          // aux emplacements 5 à 8
  fin       AT %B19:     BYTE;          // à 19, en laissant un espace
END_STRUCT;

Com_data: STRUCT OVERLAP // C1 et C2 se chevauchent
  C1 à %B0: Com1_data;
  C2 à %B0: Com2_data;
END_STRUCT;
END_TYPE
```

6.4.4.7.2 Initialisation

Les structures chevauchantes ne peuvent pas être initialisées explicitement.

6.4.4.8 Composants directement représentés d'une structure – partiellement spécifiés à l'aide de "*"

La notation avec astérisque "*" du Tableau 11 peut être utilisée pour désigner des emplacements non encore complètement spécifiés pour des composants directement représentés d'une structure.

EXEMPLE Affectation des composants d'une structure à des entrées et sorties non encore localisées.

TYPE

```
HW_COMP: STRUCT;
  IN      AT %I*: BOOL;
  VAL     AT %I*: DWORD;
  OUT     AT %Q*: BOOL;
  OUT_VAR AT %Q*: WORD;
  ITNL VAR: REAL; // non localisé
END_STRUCT;
END_TYPE
```

Dans le cas où un composant directement représenté d'une structure est utilisé dans une affectation d'emplacement dans la partie déclaration d'un programme, un type de bloc fonctionnel ou une classe, un astérisque "*" doit être utilisé à la place du préfixe de taille et du ou des entiers non signés de la concaténation pour indiquer que la représentation directe n'est pas encore complètement spécifiée.

L'utilisation de cette caractéristique nécessite que l'emplacement de la variable structurée ainsi déclarée soit complètement spécifié à l'intérieur de la construction VAR_CONFIG...END_VAR de la configuration pour chaque instance du type contenant.

Les variables de ce type ne doivent pas être utilisées dans une section VAR_INPUT, VAR_IN_OUT ou VAR_TEMP.

Une erreur est générée si l'une quelconque des spécifications complètes de la construction VAR_CONFIG...END_VAR est manquante pour une spécification d'adresse incomplète quelconque exprimée par la notation avec astérisque "*" dans une instance quelconque de programmes ou de types de bloc fonctionnel contenant ces spécifications incomplètes.

6.4.4.9 Type de données directement dérivé

6.4.4.9.1 Généralités

Un type de données défini par l'utilisateur peut être directement dérivé d'un type de données élémentaire ou d'un type de données précédemment défini par l'utilisateur.

Il peut être utilisé pour définir de nouvelles valeurs initiales spécifiques au type.

EXEMPLE Type de données directement dérivé

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

END_TYPE

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

6.4.4.9.2 Initialisation

La valeur initiale par défaut est la valeur initiale du type de données duquel le nouveau type de données est dérivé.

L'utilisateur peut initialiser le type de données avec une valeur définie par l'utilisateur. Cette initialisation est prioritaire.

La valeur initiale définie par l'utilisateur des éléments de structure peut être déclarée dans une liste entre parenthèses à la suite de l'identificateur du type de données. Les éléments dont les valeurs initiales ne sont pas énumérées dans la liste de valeurs initiales doivent avoir les valeurs initiales par défaut déclarées pour ces éléments dans la déclaration du type de données d'origine.

EXEMPLE 1 Types de données définis par l'utilisateur - utilisation

Compte tenu de la déclaration de ANALOG_16_INPUT_DATA dans le Tableau 11

et de la déclaration VAR INS: ANALOG_16_INPUT_DATA; END_VAR,

les variables INS[1] à INS[16] peuvent être utilisées partout où une variable de type INT pourrait être utilisée.

EXEMPLE 2

De même, compte tenu de la définition de Com_data dans le Tableau 11

et en outre de la déclaration VAR telegram: Com_data; END_VAR ,

la variable telegram.length peut être utilisée partout où une variable de type USINT pourrait être utilisée.

EXEMPLE 3

Cette règle peut également être appliquée de façon récursive:

Compte tenu des déclarations de ANALOG_16_INPUT_CONFIGURATION,
ANALOG_CHANNEL_CONFIGURATION et ANALOG_DATA dans le Tableau 11

et de la déclaration VAR CONF: ANALOG_16_INPUT_CONFIGURATION; END_VAR,

la variable CONF.CHANNEL[2].MIN_SCALE peut être utilisée partout où une variable de type INT pourrait être utilisée.

6.4.4.10 Références

6.4.4.10.1 Déclaration de référence

Une référence est une variable qui doit contenir uniquement une référence à une variable ou à une instance d'un bloc fonctionnel. Une référence peut avoir la valeur `NULL`, c'est-à-dire ne rien désigner.

Les références doivent être déclarées pour un type de données défini à l'aide du mot-clé `REF_TO` et d'un type de données: le type de données référence. Le type de données référence doit déjà être défini. Il peut s'agir d'un type de données élémentaire ou d'un type de données défini par l'utilisateur

NOTE Les références sans lien avec un type de données ne relèvent pas du domaine d'application de la présente partie de la série CEI 61131.

EXEMPLE 1

```

TYPE
  myArrayType:      ARRAY[0..999] OF INT;
  myRefArrType:     REF_TO myArrayType;           // Définition d'une référence
  myArrOfRefType:   ARRAY [0..12] OF myRefArrType; // Définition d'un tableau de référé-
  rences
END_TYPE

VAR
  myArray1:         myArrayType;
  myRefArr1:        myRefArrType;                 // Déclaration d'une référence
  myArrOfRef:       myArrOfRefType;               // Déclaration d'un tableau de référé-
  rences
END_VAR

```

La référence doit faire uniquement référence à des variables du type de données référence spécifié. Les références à des types de données qui sont directement dérivés sont traitées comme des alias de références au type de données de base. La dérivation directe peut être appliquée à plusieurs reprises.

EXEMPLE 2

```

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

```

`myRefType1` et `myRefType2` peuvent référencer des variables de type `ARRAY[0..999] OF INT` et des types de données dérivés.

Le type de données référence d'une référence peut également être un type de bloc fonctionnel ou une classe. Une référence à un type de base peut également désigner des instances dérivées de ce type de base.

EXEMPLE 3

```

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

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

```

Les référence de type `myRefF1` peuvent faire référence à des instances de classe `F1` et `F2`, ainsi qu'à des dérivations de ces deux classes. Lorsque les références de `myRefF2` ne peuvent pas faire référence à des instances de `F1`, seules les instances de `F2` et leurs dérivations peuvent être référencées car `F1` peut ne pas prendre en charge des méthodes et des variables de la classe étendue `F2`.

6.4.4.10.2 Initialisation de références

Les références peuvent être initialisées à l'aide de la valeur `NULL` (par défaut) ou de l'adresse d'une variable, d'une instance d'un bloc fonctionnel ou d'une classe déjà déclarés.

EXEMPLE

```
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 Opérations sur les références

L'opérateur `REF()` retourne une référence à la variable ou à l'instance spécifiée. Le type de données référence de la référence retournée est le type de données de la variable spécifiée. L'application de l'opérateur `REF()` à une variable temporaire (par exemple, variables d'une section `VAR_TEMP` quelconque et variables quelconques à l'intérieur des fonctions) n'est pas permise.

Une référence peut être affectée à une autre référence si le type de données référence est identique au type de données de base ou est un type de données de base de ce dernier.

Des références peuvent être affectées à des paramètres de fonctions, blocs fonctionnels et méthodes dans un appel si le type de données référence du paramètre est identique au type de données de base ou est un type de données de base de ce dernier. Les références ne doivent pas être utilisées en tant que variables d'entrée-sortie.

Si une référence est affectée à une référence du même type de données, cette dernière fait référence à la même variable. Dans ce contexte, un type de données directement dérivé est traité comme son type de données de base.

Si une référence est affectée à une référence du même type de bloc fonctionnel ou d'un type de bloc fonctionnel de base, cette référence fait référence à la même instance, mais est encore liée à son type de bloc fonctionnel, c'est-à-dire qu'elle peut utiliser uniquement les variables et les méthodes de son type de données référence.

Le déréférencement doit être effectué explicitement.

Une référence peut être déréférencée à l'aide du suffixe `"^"` (accent circonflexe).

Une référence déréférencée peut être utilisée de la même façon qu'une variable directement utilisée.

Le déréférencement d'une référence `NULL` est une erreur.

NOTE 1 Les contrôles possibles de références `NULL` peuvent être effectués au moment de la compilation, par le système d'exécution ou le programme d'application.

La construction `REF()` et l'opérateur de déréférencement `"^"` doivent être utilisés dans les langages graphiques dans la définition des opérandes.

NOTE 2 L'arithmétique de référence n'est pas recommandée et ne relève pas du domaine d'application de la présente partie de la CEI 61131.

EXEMPLE 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(myS1);
  myRefA1: REF_TO A1:= REF(myA1);
  myRefInt: REF_TO INT:= REF(myA1[1]);
END_VAR

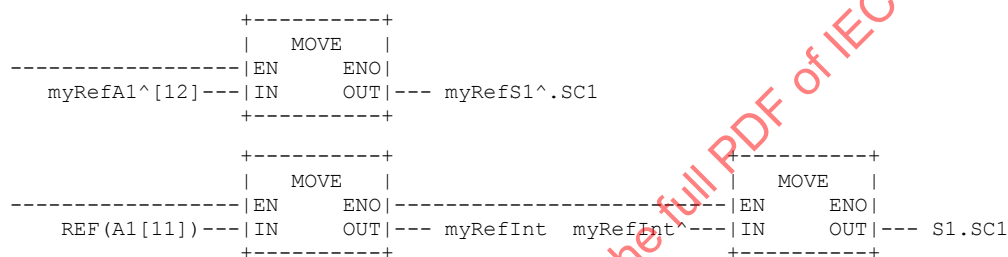
myRefS1^.SC1:= myRefA1^[12];    // dans ce cas, équivalent à S1.SC1:= A1[12];
myRefInt:= REF(A1[11]);

S1.SC1:= myRefInt^;              // affecte la valeur de A1[11] à S1.SC1

```

EXEMPLE 2

Représentation graphique des énoncés de l'Exemple 1



Le Tableau 12 définit les caractéristiques des opérations sur les références.

Tableau 12 – Opérations sur les références

N°	Description	Exemple
	Déclaration	
1	Déclaration d'un type référence	<pre> TYPE myRefType: REF_TO INT; END_TYPE </pre>
	Affectation et comparaison	
2a	Affectation d'une référence à une référence	<pre> <référence>:= <référence> myRefType1:= myRefType2; </pre>
2b	Affectation d'une référence à un paramètre de fonction, de bloc fonctionnel et de méthode	<pre> myFB (a:= myRefS1); </pre> Les types doivent être identiques!
2c	Comparaison à NULL	<pre> IF myInt = NULL THEN ... </pre>
	Référencement	
3a	REF(<variable>) Fournit la référence typée à la variable	<pre> myRefA1:= REF (A1); </pre>

N°	Description	Exemple
3b	REF(<instance de bloc fonctionnel> Fournit la référence typée à l'instance de bloc fonctionnel ou de classe	myRefFB1 := REF(myFB1)
	Déréférencement	
4	<référence>^ Fournit le contenu de la variable ou le contenu de l'instance pour laquelle la variable de référence contient la référence	myInt := myA1Ref^[12];

6.5 Variables

6.5.1 Déclaration et initialisation de variables

6.5.1.1 Généralités

Les variables permettent d'identifier des objets de données dont le contenu peut changer, par exemple, des données associées aux entrées, aux sorties ou à la mémoire de l'automate programmable.

Contrairement aux littéraux qui sont les représentations externes des données, les variables peuvent changer de valeur au cours du temps.

6.5.1.2 Déclaration

Les variables sont déclarées à l'intérieur d'une des sections de variables.

Une variable peut être déclarée à l'aide des éléments suivants:

- un type de données élémentaire, ou
- un type précédemment défini par l'utilisateur, ou
- un type référence, ou
- un type instantanément défini par l'utilisateur dans la déclaration de la variable.

Une variable peut être constituée de ce qui suit:

- une variable d'élément unique, c'est-à-dire une variable associée à l'un des types suivants:
 - un type élémentaire, ou
 - un type d'énumération ou d'intervalle défini par l'utilisateur, ou
 - un type défini par l'utilisateur dont la "parenté", définie de façon récursive, est traçable jusqu'à un type élémentaire, ou un type d'énumération ou d'intervalle;
- une variable d'éléments multiples, c'est-à-dire une variable qui représente un `ARRAY` ou un `STRUCT`;
- une référence, c'est-à-dire une variable qui désigne une autre variable ou instance de bloc fonctionnel.

Une déclaration de variable est constituée de ce qui suit:

- une liste des noms de variable qui sont déclarés,
- un ":" (deux points), et
- un type de données avec une initialisation spécifique à la variable facultative.

EXEMPLE

```

TYPE
myType: ARRAY [1..9] OF INT;      // type de données précédemment défini par l'uti-
lisateur
END_TYPE

VAR
  myVar1, myVar1a: INT;             // deux variables utilisant un type élémentaire
  myVar2: myType;                  // utilisant un type précédemment défini par
  l'utilisateur
  myVar3: ARRAY [1..8] OF REAL;    // utilisant un type instantanément défini par
  l'utilisateur
END_VAR

```

6.5.1.3 Initialisation de variables

La ou les valeurs initiales par défaut d'une variable doivent être constituées de ce qui suit:

1. la ou les valeurs initiales par défaut des types de données élémentaires sous-jacents tels que définis dans le Tableau 10,
2. la valeur `NULL`, si la variable est une référence,
3. la ou les valeurs définies par l'utilisateur du type de données assigné; cette valeur est facultativement spécifiée à l'aide de l'opérateur d'affectation ":@" dans la déclaration `TYPE` définie dans le Tableau 11,
4. la ou les valeurs définies par l'utilisateur de la variable; cette valeur est facultativement spécifiée à l'aide de l'opérateur d'affectation ":@" dans la déclaration `VAR` (Tableau 14).

Cette valeur définie par l'utilisateur peut être un littéral (par exemple, -123, 1.55, "abc") ou une expression constante (par exemple, 12*24).

Des valeurs initiales ne peuvent pas être spécifiées dans des déclarations `VAR_EXTERNAL`.

Des valeurs initiales peuvent également être spécifiées à l'aide de la caractéristique d'initialisation spécifique à une instance fournie par la construction `VAR_CONFIG...END_VAR`. Les valeurs initiales spécifiques à une instance supplantent toujours les valeurs initiales spécifiques à un type.

Tableau 13 – Déclaration de variables

N°	Description	Exemple	Explication
1	Variable avec type de données élémentaire	<pre> VAR MYBIT: BOOL; OKAY: STRING[10]; VALVE_POS AT %QW28: INT; END_VAR </pre>	Attribue un bit de mémoire à la variable booléenne <code>MYBIT</code>. Attribue de la mémoire pour recevoir une chaîne d'une longueur maximale de 10 caractères.
2	Variable avec type de données défini par l'utilisateur	<pre> VAR myVAR: myType; END_VAR </pre>	Déclaration d'une variable avec un type de données utilisateur.
3	Tableau	<pre> 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 </pre>	
4	Référence	<pre> VAR myRefInt: REF_TO INT; END_VAR </pre>	Déclaration d'une variable en tant que référence.

Tableau 14 – Initialisation de variables

N°	Description	Exemple	Explication
1	Initialisation d'une variable avec type de données élémentaire	<pre> VAR MYBIT: BOOL := 1; OKAY: STRING[10] := 'OK'; VALVE_POS AT %QW28: INT:= 100; END_VAR </pre>	Attribue un bit de mémoire à la variable booléenne <code>MYBIT</code> avec une valeur initiale de 1. Attribue de la mémoire pour recevoir une chaîne d'une longueur maximale de 10 caractères. Après l'initialisation, la chaîne a une longueur de 2 et contient la séquence de caractères à deux octets 'OK' (décimal 79 et 75, respectivement), dans un ordre approprié pour l'impression sous la forme d'une chaîne de caractères.
2	Initialisation d'une variable avec type de données défini par l'utilisateur	<pre> TYPE myType: ... END_TYPE VAR myVAR: myType:= ... // initialisation END_VAR </pre>	Déclaration d'un type de données utilisateur avec ou sans initialisation. Déclaration avec une initialisation précédente d'une variable avec un type de données utilisateur.
3	Tableau	<pre> 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(10)]; END_VAR </pre>	Attribue 8 bits de mémoire pour recevoir les valeurs initiales <code>BITS[0] := 1, BITS[1] := 1, ..., BITS[6] := 0, BITS[7] := 0.</code> Attribue un tableau d'entiers 2 x 3 TBT avec les valeurs initiales <code>TBT[1,1] := 9, TBT[1,2] := 8, TBT[1,3] := 10, TBT[2,1] := 10, TBT[2,2] := 10, TBT[2,3] := 6.</code>
4	Déclaration et initialisation de constantes	<pre> VAR CONSTANT PI: REAL:= 3.141592; PI2: REAL:= 2.0*PI; END_VAR </pre>	constante constante symbolique PI
5	Initialisation à l'aide d'expressions constantes	<pre> VAR PIx2: REAL:= 2.0 * 3.1416; END_VAR </pre>	Utilise une expression constante.
6	Initialisation d'une référence	<pre> VAR myRefInt: REF_TO INT := REF(myINT); END_VAR </pre>	Initialise <code>myRefInt</code> pour désigner la variable <code>myINT</code> .

6.5.2 Sections de variables

6.5.2.1 Généralités

Chaque déclaration d'une unité d'organisation de programme (POU, Program Organization Unit), qu'il s'agisse du bloc fonctionnel, de la fonction, du programme ou de la méthode, commence par zéro ou plusieurs parties de déclaration qui spécifient les noms, types (et, le cas échéant, l'emplacement physique ou logique et l'initialisation) des variables utilisées dans l'unité d'organisation.

La partie déclaration de la POU peut contenir différentes sections VAR suivant le type de la POU.

Les variables peuvent être déclarées dans les différentes constructions textuelles `VAR ... END_VAR` comprenant des qualificatifs tels que `RETAIN` ou `PUBLIC`, le cas échéant. Les qualificatifs des sections de variables sont résumés à la Figure 7.

Mot-clé Utilisation de la variable

Sections VAR: suivant le type de POU (voir pour une fonction, un bloc fonctionnel, un programme) ou la méthode

VAR	Interne à l'entité (fonction, bloc fonctionnel, etc.)
VAR_INPUT	Fourni de façon externe; non modifiable dans l'entité
VAR_OUTPUT	Fourni par l'entité à des entités externes
VAR_IN_OUT	Fourni par des entités externes; peut être modifié dans l'entité et fourni à une entité externe
VAR_EXTERNAL	Fourni par la configuration via VAR_GLOBAL
VAR_GLOBAL	Déclaration de la variable globale
VAR_ACCESS	Déclaration du chemin d'accès
VAR_TEMP	Stockage temporaire des variables dans des blocs fonctionnels, des méthodes et des programmes
VAR_CONFIG	Initialisation et affectation d'emplacement spécifiques à une instance
(END_VAR)	Termine les différentes sections VAR ci-dessus

Qualificateurs: peuvent suivre les mots-clés ci-dessus

RETAIN	Variables persistantes
NON_RETAIN	Variables non persistantes
PROTECTED	Accès uniquement depuis l'intérieur de l'entité elle-même et de ses dérivations (par défaut)
PUBLIC	Accès autorisé depuis toutes les entités
PRIVATE	Accès uniquement depuis l'entité elle-même
INTERNAL	Accès uniquement au sein du même espace de noms
CONSTANT ^a	Constante (la variable ne peut pas être modifiée)

NOTE L'utilisation de ces mots-clés est une caractéristique de l'unité d'organisation de programme ou de l'élément de configuration dans lesquels ceux-ci sont utilisés.

^a Les instances de bloc fonctionnel ne doivent pas être déclarées dans des sections de variables avec un qualificateur **CONSTANT**.

Figure 7 – Mots-clés pour une déclaration de variable (résumé)

- **VAR**

Les variables déclarées dans la section **VAR . . . END_VAR** persistent d'un appel de l'instance de programme ou de bloc fonctionnel à un autre.

Dans les fonctions, les variables déclarées dans cette section ne persistent pas d'un appel de la fonction à un autre.

- **VAR_TEMP**

Dans les unités d'organisation de programme, les variables peuvent être déclarées dans une section **VAR_TEMP . . . END_VAR**.

Pour les fonctions et les méthodes, les mots-clés **VAR** et **VAR_TEMP** sont équivalents.

Ces variables sont attribuées et initialisées avec une valeur par défaut spécifique du type à chaque appel et ne persistent pas entre les appels.

- **VAR_INPUT, VAR_OUTPUT et VAR_IN_OUT**

Les variables déclarées dans ces sections sont les paramètres formels des fonctions, types de bloc fonctionnel, programmes et méthodes.

- **VAR_GLOBAL et VAR_EXTERNAL**

Les variables déclarées dans une section **VAR_GLOBAL** peuvent être utilisées dans une autre POU à condition d'être redéclarées dans une section **VAR_EXTERNAL**.

La Figure 8 décrit l'utilisation de `VAR_GLOBAL`, `VAR_EXTERNAL` et `CONSTANT`.

Déclaration dans l'élément contenant la variable	Déclaration dans l'élément utilisant la variable	Autorisé?
<code>VAR_GLOBAL X</code>	<code>VAR_EXTERNAL CONSTANT X</code>	Oui
<code>VAR_GLOBAL X</code>	<code>VAR_EXTERNAL X</code>	Oui
<code>VAR_GLOBAL CONSTANT X</code>	<code>VAR_EXTERNAL CONSTANT X</code>	Oui
<code>VAR_GLOBAL CONSTANT X</code>	<code>VAR_EXTERNAL X</code>	Non

NOTE L'utilisation de la section `VAR_EXTERNAL` dans un élément contenu peut conduire à des comportements imprévus, par exemple, lorsque la valeur d'une variable externe est modifiée par un autre élément contenu dans le même élément contenant.

Figure 8 – Utilisation de `VAR_GLOBAL`, `VAR_EXTERNAL` et `CONSTANT` (règles)

- **`VAR_ACCESS`**

On peut accéder aux variables déclarées dans une section `VAR_ACCESS` à l'aide du chemin d'accès spécifié dans la déclaration.

- **`VAR_CONFIG`**

La construction `VAR_CONFIG...END_VAR` permet d'affecter des emplacements spécifiques d'une instance à des variables représentées symboliquement à l'aide de la notation avec astérisque "*" ou d'affecter des valeurs initiales spécifiques d'une instance à des variables représentées symboliquement, ou les deux.

6.5.2.2 Portée des déclarations

La portée (plage de validité) des déclarations contenues dans la partie déclaration doit être locale pour l'unité d'organisation de programme dans laquelle la partie déclaration est contenue, c'est-à-dire que les variables déclarées ne doivent pas être accessibles par d'autres unités d'organisation de programme, sauf s'il s'agit d'un paramètre explicite passant par des variables qui ont été déclarées en tant qu'entrées ou sorties de ces unités.

L'exception à cette règle est le cas de variables qui ont été déclarées comme étant globales. Ces variables sont accessibles uniquement par une unité d'organisation de programme via une déclaration `VAR_EXTERNAL`. Le type d'une variable déclarée dans un bloc `VAR_EXTERNAL` doit être en accord avec le type déclaré dans le bloc `VAR_GLOBAL` du programme, de la configuration ou de la ressource associés.

Une erreur doit être générée dans les cas suivants:

- si une unité d'organisation de programme quelconque tente de modifier la valeur d'une variable qui a été déclarée avec le qualificateur `CONSTANT` ou dans une section `VAR_INPUT`;
- si une variable déclarée en tant que `VAR_GLOBAL CONSTANT` dans un élément de configuration ou une unité d'organisation de programme ("élément contenant") est utilisée dans une déclaration `VAR_EXTERNAL` (sans le qualificateur `CONSTANT`) d'un élément quelconque contenu dans l'élément contenant comme décrit ci-dessous.

Le nombre maximal de variables autorisé dans un bloc de déclaration de variable est spécifique de l'Intégrateur.

6.5.3 Variables `ARRAY` de longueur variable

Les tableaux de longueur variable peuvent être utilisés uniquement

- en tant que variables d'entrée, de sortie ou d'entrée-sortie de fonctions et de méthodes,

- en tant que variables d'entrée-sortie de blocs fonctionnels.

La grandeur des dimensions d'un tableau de paramètre réel et formel doit être identique. Ces dimensions sont spécifiées à l'aide d'un astérisque en tant que spécification d'intervalle indéfinie pour les plages d'index.

Les tableaux de longueur variable permettent à des programmes, des fonctions, des blocs fonctionnels et des méthodes d'utiliser des tableaux avec différentes plages d'index.

Pour gérer les tableaux de longueur variable, les fonctions normalisées suivantes doivent être fournies (Tableau 15).

Tableau 15 – Variables ARRAY de longueur variable

N°	Description	Exemples
1	Déclaration à l'aide de * ARRAY [*, *, . . .] OF type de données	VAR_IN_OUT A: ARRAY [*, *] OF INT; END_VAR;
Fonctions normalisées LOWER_BOUND / UPPER_BOUND		
2a	Représentation graphique	Obtention de la limite inférieure d'un tableau: <pre> +-----+ ! LOWER_BOUND ! ARRAY ----! ARR !--- ANY_INT ANY_INT --! DIM ! +-----+</pre> Obtention de la limite supérieure d'un tableau: <pre> +-----+ ! UPPER_BOUND ! ARRAY ----! ARR !--- ANY_INT ANY_INT --! DIM ! +-----+</pre>
2b	Représentation textuelle	Obtention de la limite inférieure de la 2^{ème} dimension du tableau A: <pre>low2:= LOWER_BOUND (A, 2);</pre> Obtention de la limite supérieure de la 2^{ème} dimension du tableau A: <pre>up2:= UPPER_BOUND (A, 2);</pre>

EXEMPLE 1

```

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

A2: ARRAY [1..20, -2..2] OF INT:= [20(5(1))];
    // selon l'initialisation de tableau en 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)      → erreur
LOWER_BOUND  (A2, 3)      → erreur

```

EXEMPLE 2 Addition de tableau

```

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

```

EXEMPLE 3 Multiplication de tableau

```

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

// Utilisation:
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 Variables constantes

Les variables constantes sont des variables qui sont définies à l'intérieur d'une section de variables contenant le mot-clé `CONSTANT`. Les règles définies pour les expressions doivent s'appliquer.

EXEMPLE Variables constantes

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

6.5.5 Variables directement représentées (%)

6.5.5.1 Généralités

La représentation directe d'une variable d'élément unique doit être fournie par un symbole spécial formé par la concaténation de ce qui suit:

- un signe pour cent "%", et
- des préfixes d'emplacement I, Q ou M, et
- un préfixe de taille X (ou aucun), B, W, D ou L, et
- un ou plusieurs (voir l'adressage hiérarchique ci-dessous) entiers non signés, qui doivent être séparés par des points ".".

EXEMPLE

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

L'Intégrateur doit spécifier la correspondance entre la représentation directe d'une variable et l'emplacement physique ou logique de l'élément adressé en mémoire, entrée ou sortie.

NOTE L'utilisation de variables directement représentées dans les corps des fonctions, types de bloc fonctionnel, méthodes et types de programme limite la réutilisation de ces types d'unité d'organisation de programme, par exemple, entre des systèmes d'automate programmable dans lesquels des entrées et sorties physiques sont utilisées pour différentes applications.

L'utilisation de variables directement représentées est permise dans le corps des fonctions, blocs fonctionnels, programmes et méthodes, et dans les configurations et les ressources.

Le Tableau 16 définit les caractéristiques associées aux variables directement représentées.

L'utilisation de variables directement représentées dans le corps des POU et des méthodes est déconseillée.

Tableau 16 – Variables directement représentées

N°	Description	Exemple	Explication
	Emplacement (NOTE 1)		
1	Emplacement d'entrée I	%IW215	Mot d'entrée 215
2	Emplacement de sortie Q	%QB7	Octet de sortie 7
3	Emplacement de mémoire M	%MD48	Double mot à l'emplacement de mémoire 48
	Taille		
4a	Taille d'un seul bit X	%IX1	Type de données d'entrée BOOL
4b	Taille d'un seul bit Aucun	%I1	Type de données d'entrée BOOL
5	Taille d'un octet (8 bits) B	%IB2	Type de données d'entrée BYTE
6	Taille d'un mot (16 bits) W	%IW3	Type de données d'entrée WORD
7	Taille d'un double mot (32 bits) D	%ID4	Type de données d'entrée DWORD

N°	Description	Exemple	Explication
8	Taille d'un mot long (quad) (64 bits) $\mathbb{L}$	$\%IL5$	Type de données d'entrée $\mathbb{LWORD}$
	Adressage		
9	Adressage simple $\%IX1$	$\%IB0$	1 niveau
10	Adressage hiérarchique à l'aide de "." $\%QX7.5$	$\%QX7.5$ $\%MW1.7.9$	Défini par l'Intégrateur; par exemple, 2 niveaux, pages 0..7 3 niveaux, pages 1..16
11	Variables partiellement spécifiées à l'aide d'un astérisque "*" (NOTE 2)	$\%M^*$	

NOTE 1 Les organisations nationales de normalisation peuvent publier des tables pour la traduction de ces préfixes.

NOTE 2 L'utilisation d'un astérisque dans ce tableau requiert la caractéristique `VAR_CONFIG` et réciproquement.

6.5.5.2 Variables directement représentées – adressage hiérarchique

Lorsque la représentation directe simple (1 niveau) est étendue avec des champs entiers additionnels séparés par des points, elle doit être interprétée comme une adresse physique ou logique hiérarchique, le champ le plus à gauche représentant le niveau le plus élevé de la hiérarchie et les niveaux inférieurs apparaissant successivement sur la droite.

EXEMPLE Adresse hiérarchique

$\%IW2.5.7.1$

Par exemple, cette variable représente le premier "canal" (mot) du septième "module" dans la cinquième "baie" du deuxième "bus E/S" d'un système d'automate programmable. Le nombre maximal de niveaux d'adressage hiérarchique est spécifique de l'Intégrateur.

L'utilisation de l'adressage hiérarchique pour permettre à un programme dans un système d'automate programmable d'accéder à des données dans un autre automate programmable doit être considérée comme une extension du langage spécifique de l'Intégrateur.

6.5.5.3 Déclaration de variables directement représentées (AT)

Un nom symbolique et un type de données peuvent être attribués à la déclaration des variables directement représentées telles que définies dans le Tableau 16 (par exemple, $\%IW6$), à l'aide du mot-clé `AT`.

Une mémoire "absolue" peut être affectée à des variables avec des types de données définis par l'utilisateur (par exemple, un tableau), à l'aide de `AT`. L'emplacement de la variable définit l'adresse de début de l'emplacement de mémoire. Il n'est pas nécessaire que sa taille soit supérieure ou égale à la représentation directe spécifiée, c'est-à-dire que la mémoire vide et le chevauchement sont permis.

EXEMPLE Utilisation de la représentation directe.

```
VAR
  INP_0 AT %I0.0: BOOL;
  AT %IB12: REAL;
  PA_VAR AT %IB200: PA_VALUE;

  OUTARY AT %QW6: ARRAY[0..9] OF INT;
END_VAR
```

Nom et type d'une entrée

Nom et type défini par l'utilisateur d'un emplacement d'entrée commençant à $\%IB200$

Tableau de 10 entiers à attribuer à des emplacements de sortie contigus commençant à $\%QW6$

Pour tous les types de variable définis dans le Tableau 13, une allocation de mémoire explicite (définie par l'utilisateur) peut être déclarée à l'aide du mot-clé `AT` en combinaison avec les variables directement représentées (par exemple, `%MW10`).

Si cette caractéristique n'est pas prise en charge dans une ou plusieurs déclarations de variable, il convient de l'énoncer dans la déclaration de conformité de l'Intégrateur.

NOTE L'initialisation d'entrées du système (par exemple, `%IW10`) est spécifique de l'Intégrateur.

6.5.5.4 Variables directement représentées – partiellement spécifiées à l'aide de "*"

La notation avec astérisque "*" peut être utilisée dans des affectations d'adresse à l'intérieur des programmes et des types de bloc fonctionnel pour désigner des emplacements non encore complètement spécifiés pour des variables directement représentées.

EXEMPLE

VAR

C2 AT %Q*: BYTE;

END_VAR

Affecte un octet de sortie non encore complètement localisé à la variable de chaîne de bits C2 d'une longueur d'un octet.

Dans le cas où une variable directement représentée est utilisée dans une affectation d'emplacement d'une variable interne dans la partie déclaration d'un programme ou d'un type de bloc fonctionnel, un astérisque "*" doit être utilisé à la place du préfixe de taille et du ou des entiers non signés dans la concaténation pour indiquer que la représentation directe n'est pas encore complètement spécifiée.

Les variables de ce type ne doivent pas être utilisées dans les sections `VAR_INPUT` et `VAR_IN_OUT`.

L'utilisation de cette caractéristique nécessite que l'emplacement de la variable ainsi déclarée soit complètement spécifié à l'intérieur de la construction `VAR_CONFIG...END_VAR` de la configuration pour chaque instance du type contenant.

Une erreur est générée si l'une quelconque des spécifications complètes de la construction `VAR_CONFIG...END_VAR` est manquante pour une spécification d'adresse incomplète quelconque exprimée par la notation avec astérisque "*" dans une instance quelconque de programmes ou de types de bloc fonctionnel contenant ces spécifications incomplètes.

6.5.6 Variables persistantes (RETAIN, NON_RETAIN)

6.5.6.1 Généralités

Lorsqu'un élément de configuration (ressource ou configuration) est "démarré" ("redémarrage à chaud" ou "redémarrage à froid" selon la Partie 1 de la CEI 61131), chacune des variables associées à cet élément de configuration et à ses programmes a une valeur dépendant de l'opération de démarrage de l'élément de configuration et de la déclaration de la faculté de persistance de la variable.

La faculté de persistance peut déclarer toutes les variables contenues dans les sections de variables `VAR_INPUT`, `VAR_OUTPUT` et `VAR` des blocs fonctionnels et des programmes comme étant persistantes ou non persistantes, à l'aide du qualificateur `RETAIN` ou `NON_RETAIN` spécifié à la Figure 7. L'utilisation de ces mots-clés est une caractéristique facultative.

La Figure 9 ci-dessous présente les conditions associées à la valeur initiale d'une variable.

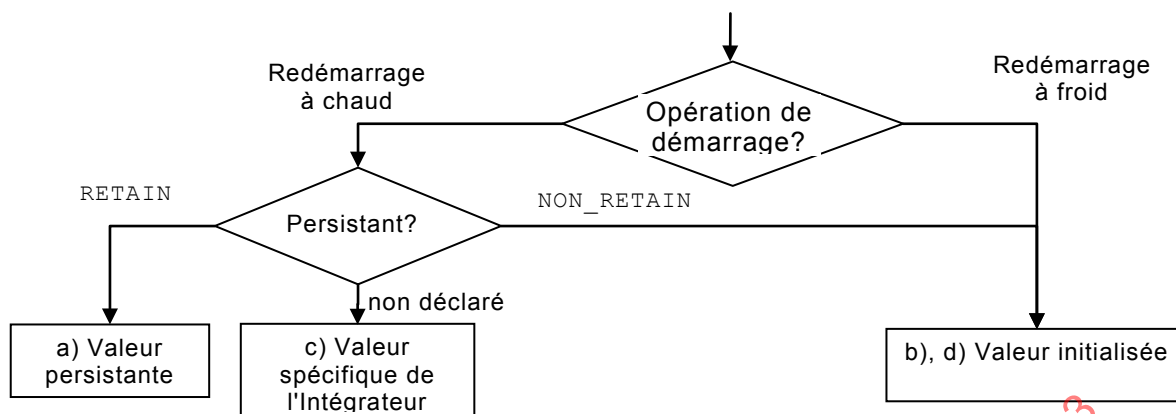


Figure 9 – Conditions associées à la valeur initiale d'une variable (règles)

1. Si l'opération de démarrage est un "redémarrage à chaud" tel que défini dans la CEI 61131-1, les valeurs initiales de toutes les variables d'une section de variables avec le qualificateur `RETAIN` doivent être les valeurs persistantes. Il s'agit des valeurs qu'avaient les variables lors de l'arrêt de la ressource ou de la configuration.
2. Si l'opération de démarrage est un "redémarrage à chaud", la valeur initiale de toutes les variables d'une section de variables avec le qualificateur `NON_RETAIN` doit être initialisée.
3. Si l'opération de démarrage est un "redémarrage à chaud" et qu'aucun qualificateur `RETAIN` ou `NON_RETAIN` n'est spécifié, les valeurs initiales sont spécifiques de l'Intégrateur.
4. Si l'opération de démarrage est un "redémarrage à froid", la valeur initiale de toutes les variables d'une section VAR avec le qualificateur `RETAIN` et `NON_RETAIN` doit être initialisée comme défini ci-dessous.

6.5.6.2 Initialisation

Les variables sont initialisées à l'aide des valeurs définies par l'utilisateur spécifiques à chaque variable.

Si aucune valeur n'est définie, la valeur initiale définie par l'utilisateur spécifique au type est utilisée. Si aucune n'est définie, la valeur initiale par défaut spécifique au type définie dans le Tableau 10 est utilisée.

Les règles supplémentaires suivantes s'appliquent:

- Les variables qui représentent des entrées du système d'automate programmable telles que définies dans la CEI 61131-1 doivent être initialisées d'une façon spécifique de l'Intégrateur.
- Les qualificateurs `RETAIN` et `NON_RETAIN` peuvent être utilisés pour des variables déclarées dans les sections statiques `VAR`, `VAR_INPUT`, `VAR_OUTPUT` et `VAR_GLOBAL`, mais pas dans la section `VAR_IN_OUT`.
- L'utilisation de `RETAIN` et `NON_RETAIN` est autorisée dans la déclaration des instances de bloc fonctionnel, de classe et de programme. Toutes les variables de l'instance sont alors traitées comme `RETAIN` ou `NON_RETAIN`, sauf si:
 - la variable est explicitement déclarée comme `RETAIN` ou `NON_RETAIN` dans la définition du type de bloc fonctionnel, de classe ou de programme;
 - la variable elle-même est un type de bloc fonctionnel ou une classe. Dans ce cas, la déclaration de persistance du type de bloc fonctionnel ou de la classe utilisés est appliquée.

L'utilisation de `RETAIN` et `NON_RETAIN` pour les instances de type de données structuré est autorisée. Tous les éléments de la structure, ainsi que ceux des structures imbriquées, sont alors traités comme `RETAIN` ou `NON_RETAIN`.

EXEMPLE

```
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 Unités d'organisation de programme (POU)

6.6.1 Caractéristiques communes pour les POU

6.6.1.1 Généralités

Les unités d'organisation de programme (POU) définies dans la présente partie de la CEI 61131 sont les fonctions, les blocs fonctionnels, les classes et les programmes. Les blocs fonctionnels et les classes peuvent contenir des méthodes.

Pour la modularité et la structuration, une POU contient une partie bien définie du programme. La POU a une interface définie avec des entrées et des sorties, et peut être appelée et exécutée plusieurs fois.

NOTE L'interface de paramètres susmentionnée n'est pas identique à celle définie dans le contexte de l'orientation objet.

Les POU et les méthodes peuvent être fournies par l'Intégrateur ou programmées par l'utilisateur.

Les POU qui ont déjà été déclarées peuvent être utilisées dans la déclaration d'autres POU comme décrit à la Figure 3.

L'appel récursif des POU et des méthodes est spécifique de l'Intégrateur.

Le nombre maximal de POU, de méthodes et d'instances pour une ressource donnée est spécifique de l'Intégrateur.

6.6.1.2 Affectation et expression

6.6.1.2.1 Généralités

Les constructions de langage affectation et expression sont utilisées dans les langages textuels et (partiellement) dans les langages graphiques.

6.6.1.2.2 Affectation

Une affectation est utilisée pour écrire la valeur d'un littéral, d'une expression constante, d'une variable ou d'une expression (voir ci-dessous) dans une autre variable. Cette dernière variable peut être d'un type de variable quelconque, tel que, par exemple, une variable d'entrée ou de sortie d'une fonction, une méthode, un bloc fonctionnel, etc.

Les variables du même type de données peuvent toujours être affectées. De plus, les règles suivantes s'appliquent:

- Une variable ou une constante de type `STRING` ou `WSTRING` peut être affectée à une autre variable de type `STRING` ou `WSTRING`, respectivement. Si la chaîne source est plus longue que la chaîne cible, le résultat est spécifique de l'Intégrateur.
- Une variable d'un type d'intervalle peut être utilisée partout où une variable de son type de base peut être utilisée. Une erreur est générée si la valeur d'un type d'intervalle est en dehors de la plage de valeurs spécifiée.
- Une variable d'un type dérivé peut être utilisée partout où une variable de son type de base peut être utilisée.
- Des règles additionnelles pour les tableaux peuvent être définies par l'Intégrateur.

Une conversion implicite et explicite du type de données peut être appliquée pour adapter le type de données de la source au type de données de la cible:

- a) sous forme textuelle (également partiellement applicable aux langages graphiques), l'opérateur d'affectation peut être

" := " ce qui signifie que la valeur de l'expression située sur le côté droit de l'opérateur est écrite dans la variable située sur le côté gauche de l'opérateur, ou

" => " ce qui signifie que la valeur située sur le côté gauche de l'opérateur est écrite dans la variable située sur le côté droit de l'opérateur.

L'opérateur "=>" est utilisé uniquement dans la liste de paramètres des appels de fonction, méthode, bloc fonctionnel, etc. pour rendre un paramètre `VAR_OUTPUT` à l'appelant;

EXEMPLE

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

NOTE Pour l'affectation des types de données définis par l'utilisateur (`STRUCTURE`, `ARRAY`), voir le Tableau 72.

- b) sous forme graphique

l'affectation est visualisée sous la forme d'une ligne de connexion graphique d'une source à une cible, en principe de gauche à droite; par exemple, d'une sortie de bloc fonctionnel à une entrée de bloc fonctionnel, de l'"emplacement" graphique d'une variable/constante à une entrée de fonction ou d'une sortie de fonction à l'"emplacement" graphique d'une variable.

La fonction normalisée `MOVE` est une des représentations graphiques d'une affectation.

6.6.1.2.3 Expression

Une expression est une construction de langage qui est constituée d'une combinaison définie d'opérandes, tels que des littéraux, des variables, des appels de fonction ou des opérateurs (`+`, `-`, `*`, `/`) et produit une valeur qui peut être à valeurs multiples.

La conversion implicite et explicite du type de données peut être appliquée pour adapter les types de données d'une opération de l'expression.

- a) Sous la forme textuelle (également partiellement applicable dans les langages graphiques), l'expression est exécutée dans un ordre défini suivant la précedence spécifiée dans le langage.

EXEMPLE ... B + C / 2 * SIN(x) ...

- b) Sous la forme graphique, l'expression est visualisée sous la forme d'un réseau de blocs graphiques (blocs fonctionnels, fonctions, etc.) reliés par des lignes.

6.6.1.2.4 Expression constante

Une expression constante est une construction de langage qui est constituée d'une combinaison définie d'opérandes tels que des littéraux, des variables constantes, des valeurs énumérées et des opérateurs (`+`, `-`, `*`) et produit une valeur qui peut être à valeurs multiples.

6.6.1.3 Accès partiel aux variables ANY_BIT

Pour les variables du type de données ANY_BIT (BYTE, WORD, DWORD, LWORD), un accès partiel à un bit, un octet, un mot et un double mot de la variable est défini dans le Tableau 17.

Afin d'adresser la partie de la variable, le symbole "%" et le préfixe de taille défini pour les variables directement représentées dans le Tableau 16 (X, aucun, B, W, D, L) sont utilisés en combinaison avec un littéral entier (0 à max) pour l'adresse dans la variable. Le littéral 0 désigne la partie la moins significative et le littéral max la partie la plus significative. Le "%X" est facultatif dans le cas de l'accès aux bits.

EXEMPLE Accès partiel à ANY_BIT

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

Bo:= By.%X0; // bit 0 de By
Bo:= By.7;   // bit 7 de By; %X est la valeur par défaut et peut être omis.
Bo:= Lo.63   // bit 63 de Lo;

By:= Wo.%B1; // octet 1 de Wo;
By:= Do.%B3; // octet 3 de Do;
```

Tableau 17 – Accès partiel aux variables ANY_BIT

N°	Description	Type de données	Exemple et syntaxe (NOTE 2)
	Type de données - Accès à		myVAR_12.%X1; yourVAR1.%W3;
1a	BYTE - bit VB2.%X0	BOOL	<nom_variable>.%X0 à <nom_variable>.%X7
1b	WORD - bit VW3.%X15	BOOL	<nom_variable>.%X0 à <nom_variable>.%X15
1c	DWORD - bit	BOOL	<nom_variable>.%X0 à <nom_variable>.%X31
1d	LWORD - bit	BOOL	<nom_variable>.%X0 à <nom_variable>.%X63
2a	WORD - octet VW4.%B0	BYTE	<nom_variable>.%B0 à <nom_variable>.%B1
2b	DWORD - octet	BYTE	<nom_variable>.%B0 à <nom_variable>.%B3
2c	LWORD - octet	BYTE	<nom_variable>.%B0 à <nom_variable>.%B7
3a	DWORD - mot	WORD	<nom_variable>.%W0 à <nom_variable>.%W1
3b	LWORD - mot	WORD	<nom_variable>.%W0 à <nom_variable>.%W3
4	LWORD - dword VL5.%D1	DWORD	<nom_variable>.%D0 à <nom_variable>.%D1
Le préfixe d'accès au bit %X peut être omis conformément au Tableau 16; par exemple, By1.%X7 est équivalent à By1.7.			
L'accès partiel ne doit pas être utilisé avec une variable directe, par exemple, %IB10.			

6.6.1.4 Représentation et règles d'appel

6.6.1.4.1 Généralités

Un appel est utilisé pour exécuter une fonction, une instance de bloc fonctionnel, ou une méthode d'un bloc fonctionnel ou d'une classe. Comme décrit à la Figure 10, un appel peut être représenté sous forme textuelle ou graphique.

1. Si aucun nom n'est spécifié pour les variables d'entrée des fonctions normalisées, les noms par défaut `IN1`, `IN2`, ... doivent s'appliquer de haut en bas. Lorsqu'une fonction normalisée a une entrée non nommée unique, le nom par défaut `IN` doit s'appliquer.
2. Une erreur doit être générée si une variable `VAR_IN_OUT` quelconque d'un appel quelconque dans une POU n'est pas "correctement mappée". Une variable `VAR_IN_OUT` est "correctement mappée" si elle remplit l'une des conditions suivantes:
 - elle est reliée graphiquement sur la gauche, ou
 - elle est affectée à l'aide de l'opérateur ":@" dans un appel textuel, à une variable déclarée (sans le qualificateur `CONSTANT`) dans un bloc `VAR_IN_OUT`, `VAR`, `VAR_TEMP`, `VAR_OUTPUT` ou `VAR_EXTERNAL` de l'unité d'organisation de programme contenant, ou à une variable `VAR_IN_OUT` "correctement mappée" d'un autre appel contenu.
3. Une variable `VAR_IN_OUT` "correctement mappée" (comme décrit dans la règle ci-dessus) d'un appel peut
 - être reliée graphiquement sur la droite, ou
 - être affectée à l'aide de l'opérateur ":@" dans une instruction d'affectation textuelle, à une variable déclarée dans un bloc `VAR`, `VAR_OUTPUT` ou `VAR_EXTERNAL` de l'unité d'organisation de programme contenant.

Une erreur doit être générée si une telle connexion conduit à une valeur ambiguë de la variable ainsi reliée.
4. Le nom d'une instance de bloc fonctionnel peut être utilisé en entrée s'il est déclaré en tant que `VAR_INPUT` ou `VAR_IN_OUT`.
L'instance peut être utilisée à l'intérieur de l'entité appelée de la façon suivante:
 - si elle est déclarée en tant que `VAR_INPUT`, les variables du bloc fonctionnel peuvent uniquement être lues,
 - si elle est déclarée en tant que `VAR_IN_OUT`, les variables du bloc fonctionnel peuvent être lues et écrites, et le bloc fonctionnel peut être appelé.

6.6.1.4.2 Langages textuels

Les caractéristiques associées à l'appel textuel sont définies dans le Tableau 20. L'appel textuel doit être constitué du nom de l'entité appelée suivi d'une liste de paramètres.

Dans le langage ST, les paramètres doivent être séparés par des virgules, et cette liste doit être délimitée sur la gauche et la droite par des parenthèses.

La liste de paramètres d'un appel doit fournir les valeurs réelles et peut les affecter aux noms de paramètre formel correspondants (le cas échéant):

- Appel formel
La liste des paramètres a la forme d'un ensemble d'affectations de valeurs réelles pour les noms de paramètre formel (liste des paramètres formels), c'est-à-dire:
 - a) les affectations de valeurs aux variables d'entrée et d'entrée-sortie à l'aide de l'opérateur ":@", et
 - b) les affectations des valeurs de variable de sortie aux variables à l'aide de l'opérateur "=>".

La liste des paramètres formels peut être complète ou incomplète. Toute variable à laquelle aucune valeur n'est affectée dans la liste doit avoir la valeur initiale, le cas échéant, affectée dans la déclaration de l'entité appelée ou la valeur par défaut pour le type de données associé.

L'ordre des paramètres dans la liste ne doit pas être significatif.

Les paramètres de contrôle d'exécution `EN` et `ENO` peuvent être utilisés.

EXEMPLE 1

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

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

- Appel informel

La liste des paramètres doit contenir exactement le même nombre de paramètres, apparaissant exactement dans le même ordre et appartenant aux mêmes types de données que ceux spécifiés dans la définition de la fonction, à l'exception des paramètres de contrôle d'exécution `EN` et `ENO`.

EXEMPLE 2

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

Cet appel est équivalent à l'appel complet de l'exemple ci-dessus, mais sans `EN/ENO`.

6.6.1.4.3 Langages graphiques

Dans les langages graphiques, l'appel de fonctions doit être représenté sous la forme de blocs graphiques conformément aux règles suivantes:

1. La forme du bloc doit être rectangulaire.
2. La taille et les proportions du bloc peuvent varier suivant le nombre d'entrées et d'autres informations à afficher.
3. Le sens de traitement dans le bloc doit être de gauche à droite (paramètres d'entrée sur la gauche et paramètres de sortie sur la droite).
4. Le nom ou le symbole de l'entité appelée, comme spécifié ci-dessous, doit être situé à l'intérieur du bloc.
5. Des dispositions doivent être prises pour que les noms des variables d'entrée et de sortie apparaissent respectivement sur les côtés intérieurs gauche et droit du bloc.
6. Une entrée `EN` et/ou une sortie `ENO` additionnelles peuvent être utilisées. En cas d'utilisation, celles-ci doivent figurer dans les positions supérieures, respectivement sur la gauche et la droite, du bloc.
7. Le résultat de la fonction doit figurer en position supérieure sur le côté droit du bloc, sauf s'il existe une sortie `ENO`, auquel cas le résultat de la fonction doit être présenté à la position suivante, au-dessous de la sortie `ENO`. Etant donné que le nom de l'entité appelée est utilisé pour l'affectation de sa valeur de sortie, aucun nom de variable de sortie ne doit figurer sur le côté droit du bloc, c'est-à-dire pour le résultat de la fonction.
8. Les connexions de paramètres (y compris le résultat de la fonction) doivent être représentées par des lignes de flux de signal.
9. L'inversion de signaux booléens doit être représentée par un cercle ouvert placé juste à l'extérieur de l'intersection des lignes d'entrée ou de sortie avec le bloc. Dans le jeu de caractères, elle peut être représentée par la lettre majuscule "O", comme décrit dans le Tableau 20. L'inversion est effectuée à l'extérieur de la POU.
10. Toutes les entrées et sorties (y compris le résultat de la fonction) d'une fonction représentée graphiquement doivent être représentées par un simple trait à l'extérieur du côté correspondant du bloc, même si l'élément de données peut être une variable d'éléments multiples.
11. Les résultats et les sorties (`VAR_OUTPUT`) peuvent être reliés à une variable, utilisée en entrée pour d'autres appels, ou laissés déconnectés.

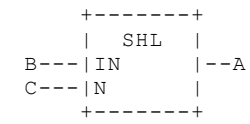
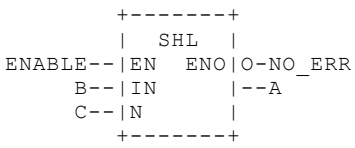
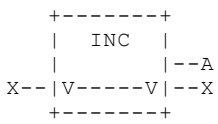
Exemple graphique (FBD)	Exemple textuel (ST)	Explication
a) 	A:= ADD(B,C,D); //Fonction ou A:= B + C + D; // Opérateurs	Liste des paramètres informels (B, C, D)
b) 	A:= SHL(IN:= B, N:= C);	Noms de paramètre formel IN, N
c) 	A:= SHL(EN:= ENABLE, IN:= B, N := C, NOT ENO => NO_ERR);	Noms de paramètre formel Utilisation de l'entrée EN et de la sortie ENO inversée
d) 	A:= INC(V:= X);	Fonction INC définie par l'utilisateur Noms de paramètre formel V pour VAR_IN_OUT
Ces exemples illustrent l'utilisation graphique et l'utilisation textuelle équivalente, comprenant l'utilisation d'une fonction normalisée (ADD) sans noms de paramètre formel définis, d'une fonction normalisée (SHL) avec des noms de paramètre formel définis, de la même fonction avec l'utilisation additionnelle de l'entrée EN et de la sortie ENO inversée, et d'une fonction définie par l'utilisateur (INC) avec des noms de paramètre formel définis.		

Figure 10 – Représentation formelle et informelle d'appel (exemples)

6.6.1.5 Contrôle d'exécution (EN, ENO)

Comme décrit dans le Tableau 18, une entrée EN (Activer) et/ou une sortie ENO (Activer sortie) booléennes additionnelles peuvent être fournies par l'Intégrateur ou l'utilisateur conformément aux déclarations.

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

Lorsque ces variables sont utilisées, l'exécution des opérations définies par la POU doit être contrôlée selon les règles suivantes:

1. Si la valeur de EN est FALSE, la POU ne doit pas être exécutée. De plus, ENO doit être réinitialisé à FALSE. L'Intégrateur doit spécifier le comportement de façon détaillée dans ce cas; voir les exemples ci-dessous.
2. Sinon, si la valeur de EN est TRUE, ENO est défini à TRUE et la mise en œuvre de la POU doit être exécutée. La POU peut assigner une valeur booléenne à ENO conformément au résultat de l'exécution.
3. Si une erreur quelconque survient pendant l'exécution de l'une des POU, la sortie ENO de cette POU doit être réinitialisée à FALSE (0) par le système d'automate programmable ou l'Intégrateur doit spécifier un autre comportement lors d'une telle erreur.
4. Si la sortie ENO est évaluée à FALSE (0), les valeurs de toutes les sorties de POU (VAR_OUTPUT, VAR_IN_OUT et résultat de la fonction) sont spécifiques de l'Intégrateur.
5. L'entrée EN doit uniquement être définie comme une valeur réelle faisant partie de l'appel d'une POU.

6. La sortie `ENO` doit uniquement être transférée à une variable faisant partie de l'appel d'une POU.
7. La sortie `ENO` doit uniquement être définie à l'intérieur de sa POU.
8. L'utilisation des paramètres `EN/ENO` dans la fonction `REF()` pour obtenir une référence à `EN/ENO` est une erreur.

Un comportement différent de l'exécution de POU normale peut être mis en œuvre si `EN` est `FALSE`. Ceci doit être spécifié par l'Intégrateur. Voir les exemples ci-dessous.

EXEMPLE 1 Mise en œuvre interne

L'entrée `EN` est évaluée à l'intérieur de la POU.

Si `EN` est `FALSE`, `ENO` est défini à `False` et la POU revient immédiatement ou effectue un sous-ensemble d'opérations suivant cette situation.

Tous les paramètres d'entrée et d'entrée-sortie spécifiés sont évalués et définis dans l'instance de la POU (sauf pour les fonctions).

La validité des paramètres d'entrée-sortie est contrôlée.

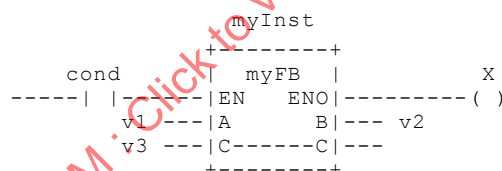
EXEMPLE 2 Mise en œuvre externe

L'entrée `EN` est évaluée à l'extérieur de la POU. Si `EN` est `False`, seul `ENO` est défini à `False` et la POU n'est pas appelée.

Les paramètres d'entrée et d'entrée-sortie ne sont pas évalués et ne sont pas définis dans l'instance de la POU. La validité des paramètres d'entrée-sortie n'est pas contrôlée.

L'entrée `EN` n'est pas affectée à l'extérieur de la POU séparément de l'appel.

La figure et les exemples ci-dessous décrivent l'utilisation avec et sans `EN/ENO`:



EXEMPLE 3 Mise en œuvre interne

```
myInst (EN:= cond, A:= v1, C:= v3, B=> v2, ENO=> X);
où le corps de myInst commence en principe par
```

```
IF NOT EN THEN...           // effectuer un sous-ensemble d'opérations
                           // suivant la situation
ENO:= 0; RETURN; END_IF;
```

EXEMPLE 4 Mise en œuvre externe

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

Le Tableau 18 décrit les caractéristiques associées à l'appel de POU sans et avec `EN/ENO`.

Tableau 18 – Contrôle de l'exécution en utilisant graphiquement EN et ENO

N°	Description ^a	Exemple ^b
1	Utilisation sans EN et ENO	Représenté pour une fonction en FBD et ST <pre> +-----+ A--- + ---C B--- +-----+ </pre> C := ADD(IN1:= A, IN2:= B);
2	Utilisation de EN uniquement (sans ENO)	Représenté pour une fonction en FBD et ST <pre> +-----+ ADD_EN--- EN A--- + ---C B--- +-----+ </pre> C := ADD(EN:= ADD_EN. IN1:= A, IN2:= B);
3	Utilisation de ENO uniquement (sans EN)	Représenté pour une fonction en FBD et ST <pre> +-----+ ENO ---ADD_OK A--- + ---C B--- +-----+ </pre> C := ADD(IN1:= A, IN2:= B, ENO=> ADD_OK);
4	Utilisation de EN et ENO	Représenté pour une fonction en LD et ST <pre> +-----+ ADD_EN + ADD_OK +--- --- EN ENO () ---+ A--- ---C B--- +-----+ </pre> C := ADD(EN:= ADD_EN, IN1:= a, IN2:= IN2, EN => ADD_OK);

^a L'Intégrateur doit spécifier les langages dans lesquels la caractéristique est prise en charge. Il peut ainsi être interdit d'utiliser EN et/ou ENO dans une mise en œuvre.

^b Les langages choisis pour illustrer les caractéristiques ci-dessus sont spécifiés à titre d'exemple uniquement.

6.6.1.6 Conversion de type de données

La conversion de type de données est utilisée pour adapter les types de données en vue d'une utilisation dans des expressions, des affectations et des affectations de paramètre.

La représentation et l'interprétation des informations stockées dans une variable sont dépendantes du type de données déclaré de la variable. Il existe deux cas dans lesquels la conversion de type est utilisée.

- Dans une affectation d'une valeur de données d'une variable à une autre variable d'un type de données différent.

Cela est applicable avec les opérateurs d'affectation " := " et " => " et avec l'affectation de variables déclarées en tant que paramètres, c'est-à-dire les entrées, sorties, etc. de fonctions, blocs fonctionnels, méthodes et programmes. La Figure 11 décrit les règles de conversion d'un type de données source en type de données cible.

```

EXEMPLE 1      A:= B;                // Affectation de variable
                  FB1 (x:= z, v => W);  // Affectation de paramètre

```

- Dans une expression (voir 7.3.2 pour le langage ST) constituée d'opérateurs tels que "+", d'opérandes tels que des littéraux et des variables ayant des types de données identiques ou différents.

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

- La conversion de type de données explicite est effectuée à l'aide des fonctions de conversion.
- La conversion de type de données implicite possède les règles d'application suivantes:
 - 1) Elle doit conserver la valeur et la précision des types de données.
 - 2) Elle peut être appliquée pour des fonctions typées.
 - 3) Elle peut être appliquée pour les affectations d'une expression à une variable.

EXEMPLE 3

```
myUDInt:= myUInt1 * myUInt2;
/* La multiplication a un résultat UINT
   qui est ensuite implicitement converti en UDINT lors de l'affectation */
```

- 4) Elle peut être appliquée pour l'affectation d'un paramètre d'entrée.
- 5) Elle peut être appliquée pour l'affectation d'un paramètre de sortie.
- 6) Elle ne doit pas être appliquée pour l'affectation à des paramètres d'entrée-sortie.
- 7) Elle peut être appliquée de sorte que les opérandes et les résultats d'une opération ou d'une fonction en surcharge aient le même type de données.

EXEMPLE 4

```
myUDInt:= myUInt1 * myUDInt2;
// myUInt1 est implicitement converti en UDINT, la multiplication a un résultat UDINT
```

- 8) L'Intégrateur doit définir les règles pour les littéraux non typés.

NOTE L'utilisateur peut utiliser des littéraux typés afin d'éviter toute ambiguïté.

EXEMPLE 5

```
IF myWord = NOT (0) THEN ...; // Comparaison ambiguë à 16#FFF, 16#0001, 16#00FF, etc.
IF myWord = NOT (WORD#0) THEN ...; // Comparaison ambiguë à 16#FFFF
```

La Figure 11 décrit les deux alternatives de conversion "implicite" et "explicite" du type de données source en type de données cible.

Type de données source		Type de données cible																									
		réel		entier				non signé				bit				date et heure						caractère					
		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
réel	LREAL		e	e	e	e	e	e	e	e	e	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	REAL	i		e	e	e	e	e	e	e	-	e	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
entier	LINT	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	-	-	-	-	-	-	-	-	-	-	-	-	
	INT	i	i	i	i		e	e	e	e	e	e	e	e	-	-	-	-	-	-	-	-	-	-	-	-	
	SINT	i	i	i	i	i		e	e	e	e	e	e	e	-	-	-	-	-	-	-	-	-	-	-	-	
non signé	ULINT	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	-	-	-	-	-	-	-	-	-	-	-	-	
	UINT	i	i	i	i	e	e	i	i		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	-	-	-	-	-	-	-	-	-	-	-	-	
	DWORD	-	e	e	e	e	e	e	e	e	i		e	e	-	-	-	-	-	-	-	-	-	-	-	-	
	WORD	-	-	e	e	e	e	e	e	e	i	i		e	-	-	-	-	-	-	-	-	-	-	e	-	
	BYTE	-	-	e	e	e	e	e	e	e	i	i	i		-	-	-	-	-	-	-	-	-	-	-	e	
	BOOL	-	-	e	e	e	e	e	e	e	i	i	i	i		-	-	-	-	-	-	-	-	-	-	-	
date et heure	LTIME	-	-	-	-	-	-	-	-	-	-	-	-	-	-		e	-	-	-	-	-	-	-	-	-	
	TIME	-	-	-	-	-	-	-	-	-	-	-	-	-	-	i		-	-	-	-	-	-	-	-	-	
	LDT	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		e	e	e	e	e	-	-	-	
	DT	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	i		e	e	e	e	-	-	-	
	LDATE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		e	-	-	-	-	-	
	DATE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	i		-	-	-	-	-	
	LTOD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		e	-	-	-	
	TOD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	i		-	-	-	
caractère	WSTRING	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		e	-	-
	STRING (NOTE)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	e		-	e
	WCHAR	-	-	-	-	-	-	-	-	-	-	e	e	e	-	-	-	-	-	-	-	-	-	i	-		e
	CHAR (NOTE)	-	-	-	-	-	-	-	-	-	e	e	e	e	e	-	-	-	-	-	-	-	-	-	i	e	

Légende

Aucune conversion de type de données n'est nécessaire.

- Aucune conversion implicite ou explicite de type de données n'est définie par la présente norme. La mise en œuvre peut prendre en charge des conversions additionnelles de type de données spécifiques de l'Intégrateur.
- i Conversion implicite du type de données; cependant, une conversion de type explicite est également autorisée.
- e Une conversion explicite du type de données appliquée par l'utilisateur (fonctions de conversion normalisées) peut être utilisée pour accepter une perte de précision ou une incohérence de plage, ou pour permettre un comportement dépendant de l'Intégrateur possible.

NOTE Les conversions de STRING en WSTRING et de CHAR en WCHAR ne sont pas implicites, afin d'éviter les conflits avec le jeu de caractères utilisé.

Figure 11 – Règles de conversion d'un type de données – implicite et/ou explicite (résumé)

La Figure 12 ci-dessous décrit les conversions de type de données qui sont prises en charge par la conversion de type implicite. Les flèches présentent les chemins de conversions possibles; par exemple, **BOOL** peut être converti en **BYTE**, **BYTE** peut être converti en **WORD**, etc.

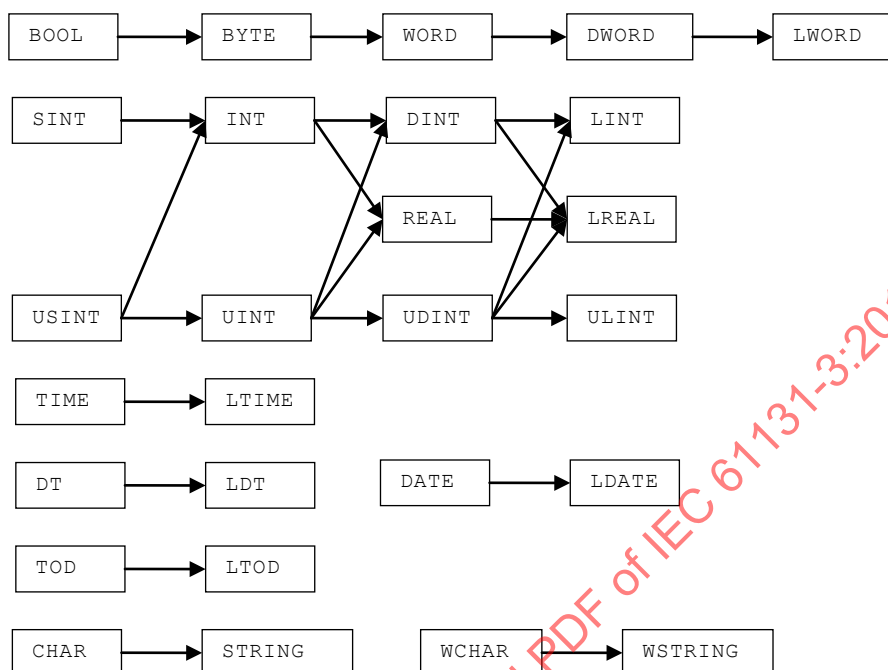


Figure 12 – Conversions de type implicites prises en charge

L'exemple ci-dessous donne des exemples de conversions de type de données.

EXEMPLE 6 Conversion de type explicite / implicite

1) Déclaration de type

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

2) Utilisation dans le langage ST

a) Conversion de type explicite

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

b) Conversion de type en surcharge explicite

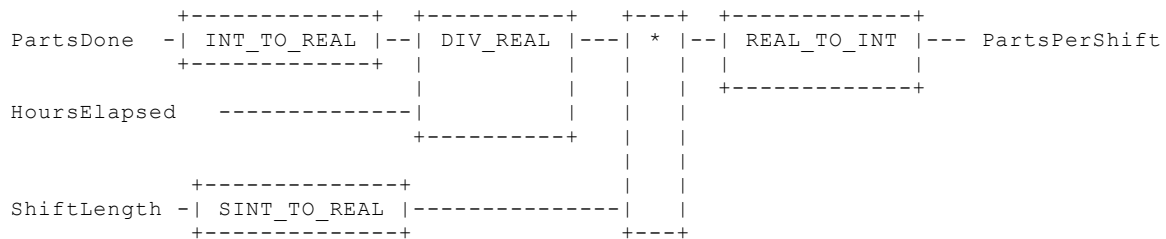
```
PartsRatePerHr := TO_REAL(PartsDone) / HoursElapsed;
PartsPerShift := TO_INT(TO_REAL(ShiftLength) * PartsRatePerHr);
```

c) Conversion de type implicite

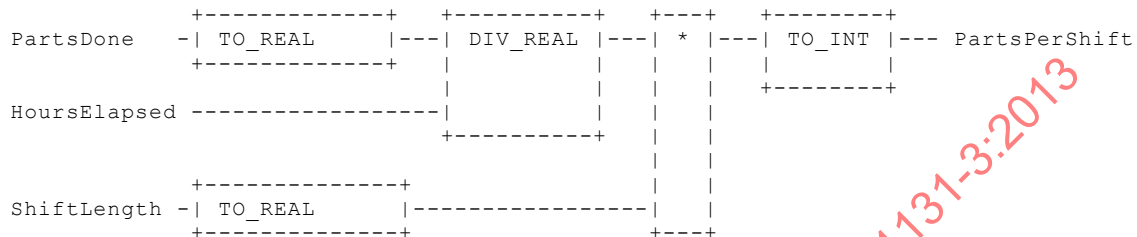
```
PartsRatePerHr := PartsDone / HoursElapsed;
PartsPerShift := TO_INT(ShiftLength * PartsRatePerHr);
```

3) Utilisation dans le langage FBD

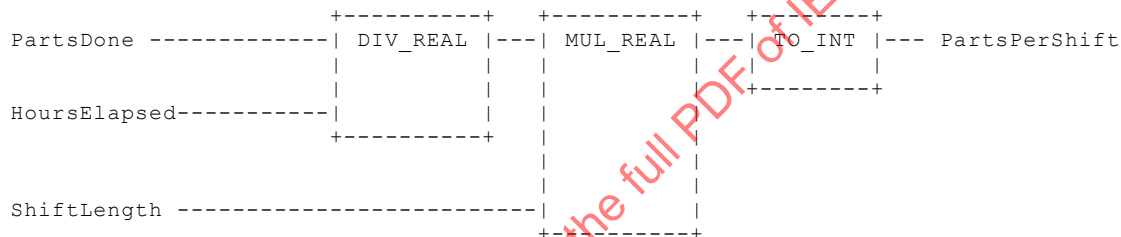
a) Conversion de type explicite



b) Conversion de type en surcharge explicite



c) Conversion de type implicite avec fonctions typées



6.6.1.7 Surcharge

6.6.1.7.1 Généralités

Un élément de langage est dit en surcharge lorsqu'il peut agir sur des éléments de données d'entrée de différents types dans un type de données générique (par exemple, ANY_NUM, ANY_INT).

Les éléments de langage normalisés suivants, qui sont fournis par le fabricant, peuvent comporter une caractéristique spéciale de surcharge générique:

- Fonctions normalisées

Il s'agit de fonctions normalisées en surcharge (par exemple, ADD, MUL) et de fonctions de conversion normalisées en surcharge (par exemple, TO_REAL, TO_INT).

- Méthodes normalisées

La présente partie de la CEI 61131 ne définit pas de méthodes normalisées dans des classes ou des types de bloc fonctionnel normalisés. L'Intégrateur peut cependant en fournir.

- Blocs fonctionnels normalisés

La présente partie de la CEI 61131 ne définit pas de blocs fonctionnels normalisés, si ce n'est des blocs simples tels que des compteurs.

Cependant, d'autres parties de la CEI 61131 peuvent en définir et l'Intégrateur peut en fournir.

- Classes normalisées

La présente partie de la CEI 61131 ne définit pas de classes normalisées. Cependant, d'autres parties de la CEI 61131 peuvent en définir et l'Intégrateur peut en fournir.

- Opérateurs

Il s'agit par exemple de "+" et "*" dans le langage ST, et de `ADD`, `MUL` dans le langage IL.

6.6.1.7.2 Conversion de type de données

Lorsqu'un système d'automate programmable prend en charge un élément de langage en surcharge, cet élément de langage doit s'appliquer à tous les types de données adaptés du type générique spécifié qui sont pris en charge par ce système.

Les types de données adaptés pour chaque élément de langage sont définis dans les tableaux de caractéristiques associés. Les exemples ci-dessous en décrivent les détails:

EXEMPLE 1

La présente norme définit, pour la fonction `ADD`, le type de données générique `ANY_NUM` pour un nombre d'entrées du même genre et une sortie de résultats.
L'Intégrateur spécifie, pour ce type de données générique `ANY_NUM` du système d'automate programmable, les types de données élémentaires associés `REAL` et `INT`.

EXEMPLE 2

La présente norme définit, pour la fonction de décalage de bit `LEFT`, le type de données générique `ANY_BIT` pour une entrée, et la sortie de résultats et le type de données générique `ANY_INT` pour une autre entrée.
L'Intégrateur spécifie, pour ces deux types de données génériques du système d'automate programmable: `ANY_BIT`, qui représente, par exemple, les types de données élémentaires `BYTE` et `WORD`;
`ANY_INT` qui représente, par exemple, les types de données élémentaires `INT` et `LINT`.

Un élément de langage en surcharge doit agir sur les types de données élémentaires définis selon les règles suivantes:

- Les types de données des entrées et les sorties/le résultat doivent être du même type; cette règle est applicable aux entrées et aux sorties/au résultat du même genre.
L'expression "du même genre" signifie que les paramètres, les opérandes et le résultat sont utilisés de la même manière comme entrées d'une addition ou d'une multiplication. Les combinaisons plus complexes doivent être spécifiques de l'Intégrateur.
- Si les types de données des entrées et des sorties du même genre ne possèdent pas le même type, la conversion dans l'élément de langage est spécifique de l'Intégrateur.
- La conversion de type implicite d'une expression et de l'affectation suit la séquence d'évaluation de l'expression. Voir l'exemple ci-dessous.
- Le type de données de la variable utilisée pour stocker le résultat de la fonction en surcharge n'influence pas le type de données du résultat de la fonction ou opération.

NOTE L'utilisateur peut explicitement spécifier le type de résultat de l'opération à l'aide de fonctions typées.

EXEMPLE 3

```
int3 := int1 + int2 (* L'addition est effectuée comme une opération d'entier *)
dint1:= int1 + int2; (* L'addition est effectuée comme une opération d'entier, puis le résultat est converti
                    en DINT et affecté à dint1 *)
dint1:= dint2 + int3; (* int3 est converti en DINT, l'addition est effectuée comme une addition de DINT *)
```

6.6.2 Fonctions

6.6.2.1 Généralités

Une fonction est une unité d'organisation de programme (POU) qui ne stocke pas son état, c'est-à-dire les entrées, les variables internes et les sorties/le résultat.

Les caractéristiques communes des POU s'appliquent aux fonctions, sauf indication contraire.

L'exécution d'une fonction

- fournit généralement un résultat temporaire qui peut être un élément d'une donnée, ou un tableau ou une structure à valeurs multiples,

- produit éventuellement une ou plusieurs variables de sortie qui peuvent être à valeurs multiples,
- peut modifier la valeur de la ou des variables d'entrée-sortie et `VAR_EXTERNAL`.

Une fonction avec résultat peut être appelée dans une expression ou en tant qu'énoncé.

Une fonction sans résultat ne doit pas être appelée à l'intérieur d'une expression.

6.6.2.2 Déclaration de fonction

La déclaration d'une fonction doit être constituée des éléments ci-dessous tels que définis dans le Tableau 19. Ces caractéristiques sont déclarées de manière similaire à ce qui est décrit pour les blocs fonctionnels.

Les règles de déclaration de fonction ci-dessous doivent être appliquées comme spécifié dans le Tableau 19:

1. Les déclarations commencent par le mot-clé `FUNCTION` suivi d'un identificateur spécifiant le nom de la fonction.
2. Si un résultat est disponible, deux points ":" suivis du type de données de la valeur à retourner par la fonction doivent être spécifiés ou, si aucun résultat de la fonction n'est disponible, les deux points et le type de données doivent être omis.
3. Les constructions avec `VAR_INPUT`, `VAR_OUTPUT` et `VAR_IN_OUT`, si nécessaire, spécifient les noms et les types de données des paramètres de la fonction.
4. Les valeurs des variables qui sont transmises à la fonction par l'intermédiaire d'une construction `VAR_EXTERNAL` peuvent être modifiées depuis l'intérieur du bloc fonctionnel.
5. Les valeurs des constantes qui sont transmises à la fonction par l'intermédiaire d'une construction `VAR_EXTERNAL CONSTANT` ne peuvent pas être modifiées depuis l'intérieur de la fonction.
6. Les valeurs des variables qui sont transmises à la fonction par l'intermédiaire d'une construction `VAR_IN_OUT` peuvent être modifiées depuis l'intérieur de la fonction.
7. Les tableaux de longueur variable peuvent être utilisés comme `VAR_INPUT`, `VAR_OUTPUT` et `VAR_IN_OUT`.
8. Les variables d'entrée, de sortie et temporaires peuvent être initialisées.
9. Les entrées et sorties `EN/ENO` peuvent être utilisées comme décrit.
10. Une construction `VAR...END_VAR` et `VAR_TEMP...END_VAR`, si nécessaire, spécifient les noms et les types des variables temporaires internes. Contrairement aux blocs fonctionnels, les variables déclarées dans la section `VAR` ne sont pas stockées.
11. Si les types de données génériques (par exemple, `ANY_INT`) sont utilisés dans la déclaration des variables de la fonction normalisée, les règles régissant l'utilisation des types réels des paramètres de ces fonctions doivent faire partie de la définition de la fonction.
12. Les constructions d'initialisation des variables peuvent être utilisées pour la déclaration des valeurs initiales d'entrées de la fonction et les valeurs initiales de leurs variables internes et de sortie.
13. Le mot-clé `END_FUNCTION` termine la déclaration.

Tableau 19 – Déclaration de fonction

N°	Description	Exemple
1a	Sans résultat FUNCTION ... END_FUNCTION	FUNCTION myFC ... END_FUNCTION
1b	Avec résultat FUNCTION <nom>: <type de données> END_FUNCTION	FUNCTION myFC: INT ... END_FUNCTION
2a	Entrées VAR_INPUT...END_VAR	VAR_INPUT IN: BOOL; T1: TIME; END_VAR
2b	Sorties VAR_OUTPUT...END_VAR	VAR_OUTPUT OUT: BOOL; ET_OFF: TIME; END_VAR
2c	Entrées-sorties VAR_IN_OUT...END_VAR	VAR_IN_OUT A: INT; END_VAR
2d	Variables temporaires VAR_TEMP...END_VAR	VAR_TEMP I: INT; END_VAR
2e	Variables temporaires VAR...END_VAR	VAR B: REAL; END_VAR Pour des raisons de compatibilité, il existe une différence par rapport aux blocs fonctionnels: les VAR sont statiques dans les blocs fonctionnels (stockées)!
2f	Variables externes VAR_EXTERNAL...END_VAR	VAR_EXTERNAL B: REAL; END_VAR Correspondant à VAR_GLOBAL B: REAL...
2g	Constantes externes VAR_EXTERNAL CONSTANT...END_VAR	VAR_EXTERNAL CONSTANT B: REAL; END_VAR Correspondant à VAR_GLOBAL B: REAL
3a	Initialisation des entrées	VAR_INPUT MN: INT:= 0;
3b	Initialisation des sorties	VAR_OUTPUT RES: INT:= 1;
3c	Initialisation des variables temporaires	VAR I: INT:= 1;
--	Entrées et sorties EN/ENO	Défini dans le Tableau 18

EXEMPLE

```
// Spécification de l'interface des paramètres
FUNCTION SIMPLE_FUN: REAL
VAR_INPUT
  A, B: REAL;
  C: REAL:= 1.0;
END_VAR
VAR_IN_OUT COUNT: INT;
END_VAR

// Spécification de l'interface des paramètres
FUNCTION
+-----+
| SIMPLE_FUN |
REAL----|A|-----REAL
REAL----|B|-----
REAL----|C|-----
INT----|COUNT---COUNT|----INT
+-----+

// Spécification du corps de la fonction
+---+
|ADD|---+
COUNT--| |---COUNTPl--|:=|---COUNT
1--| |---+
+---+
A---| * |---+
B---| |---+---SIMPLE_FUN
+---+
C-----| |---+
+---+

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

a) Déclaration et corps d'une fonction (ST et FBD) – NOTE

```
VAR_GLOBAL dataArray: ARRAY [0..100]
OF INT; END_VAR // Interface externe

FUNCTION SPECIAL_FUN // pas de résultat de fonction, mais une sortie Sum
VAR_INPUT
  FirstIndex: INT;
  LastIndex: INT;
END_VAR
VAR_OUTPUT
  Sum: INT;
END_VAR
VAR_EXTERNAL dataArray:
  ARRAY [0..100] OF INT;
END_VAR

VAR I: INT; Sum: INT:= 0; END_VAR
FOR i:= FirstIndex TO LastIndex
DO Sum:= Sum + dataArray[i];
END_FOR // Corps de la fonction – non représenté graphiquement

END_FUNCTION
```

b) Déclaration et corps d'une fonction (sans résultat de fonction – avec sortie Var)

NOTE Dans a), une valeur par défaut définie de 1.0 est attribuée à la variable d'entrée afin d'éviter une erreur de "division par zéro" si l'entrée n'est pas spécifiée lorsque la fonction est appelée, par exemple, si une entrée graphique de la fonction n'est pas reliée.

6.6.2.3 Appel d'une fonction

L'appel d'une fonction peut être représenté sous forme textuelle ou graphique.

Etant donné que les variables d'entrée, les variables de sortie et le résultat d'une fonction ne sont pas stockés, l'affectation des entrées, et l'accès aux sorties et au résultat doivent être immédiats lors de l'appel de la fonction.

Si un tableau de longueur variable est utilisé en tant que paramètre, le paramètre doit être relié à la variable statique.

Une fonction ne doit pas contenir d'informations d'état interne, c'est-à-dire:

- elle ne stocke aucun des éléments d'entrée, internes (temporaires) et de sortie d'un appel au suivant;

- l'appel d'une fonction avec les mêmes paramètres (VAR_INPUT et VAR_IN_OUT) et les mêmes valeurs de VAR_EXTERNAL produira toujours la même valeur sur ses variables de sortie, ses variables d'entrée-sortie, ses variables externes et le résultat de sa fonction, le cas échéant.

NOTE 1 Certaines fonctions, généralement fournies en tant que fonctions système par l'Intégrateur, peuvent produire différentes valeurs; par exemple, TIME(), RANDOM().

Tableau 20 – Appel d'une fonction

N°	Description	Exemple
1a	Appel formel complet (textuel uniquement) NOTE 1 Est utilisé si EN/ENO est nécessaire dans les appels.	A:= LIMIT(EN:= COND, IN:= B, MN:= 0, MX:= 5, ENO => TEMPL);
1b	Appel formel incomplet (textuel uniquement) NOTE 2 Est utilisé si EN/ENO n'est pas nécessaire dans les appels.	A:= LIMIT(IN:= B, MX:= 5); NOTE 3 La variable MN aura la valeur par défaut 0 (zéro).
2	Appel informel (textuel uniquement) (corriger l'ordre et terminer) NOTE 4 Est utilisé pour l'appel de fonctions normalisées sans noms formels.	A:= LIMIT(B, 0, 5); NOTE 5 Cet appel est équivalent à 1a, mais sans EN/ENO.
3	Fonction sans résultat de fonction	FUNCTION myFun // pas de déclaration de type VAR_INPUT x: INT; END_VAR; VAR_OUTPUT y: REAL; END_VAR; myFun(150, var); // Appel
4	Représentation graphique	+-----+ FUN a -- EN ENO -- b -- IN1 -- résultat c -- IN2 Q1 --sortie Q2 +-----+
5	Utilisation de l'entrée et de la sortie booléennes inversées en représentation graphique	+-----+ FUN a -o EN ENO -- b -- IN1 -- résultat c -- IN2 Q1 o- sortie Q2 +-----+ NOTE 6 L'utilisation de ces constructions est interdite pour les variables d'entrée-sortie.
6	Utilisation graphique de VAR_IN_OUT	+-----+ myFC1 a -- In1 Out1 -- d b -- Inout--Inout -- c +-----+

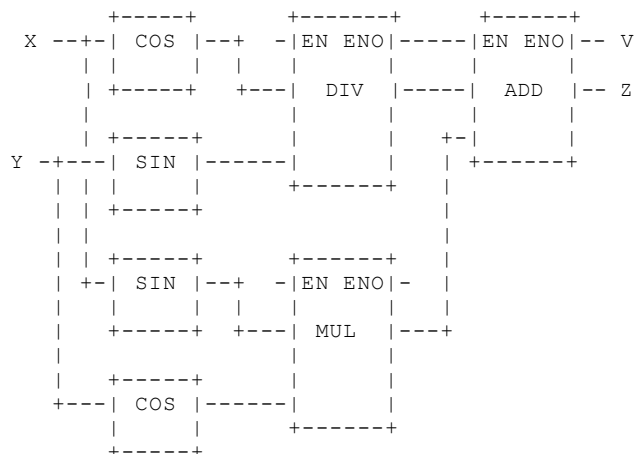
EXEMPLE Appel d'une fonction**Appel**

```

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);

```

**a) Appel des fonctions normalisées avec résultat et EN/ENO****Déclaration**

```

FUNCTION My_function                                     // pas de type, pas de résultat
  VAR_INPUT In1: REAL; END_VAR
  VAR_OUTPUT Out1, Out2: REAL; END_VAR
  VAR_TEMP Tmp1: REAL; END_VAR                          // VAR_TEMP autorisé
  VAR_EXTERNAL Ext: BOOL; END_VAR

  // Corps de la fonction
END_FUNCTION

```

Appel textuel et graphique

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

**b) Déclaration et appel d'une fonction sans résultat mais avec des variables de sortie**

Appel textuel et graphique

```
myFC1 (In1 := a, Inout := b, Out1 => Tmp1);    // Utilisation d'une variable tempo-
raire
d:= myFC2 (In1 := Tmp1, Inout:= b);           // b stockée dans inout; affectation à
c
c:= b;                                         // b affecté à c
```

```

+-----+      +-----+
| myFC1 |      | myFC2 |
a --|In1   Out1|-----|In1   |-- d    // Résultat
b --|Inout--Inout|-----|Inout--Inout|-- c    // Affectation à c
+-----+      +-----+
```

c) Appel d'une fonction avec représentation graphique des variables d'entrée-sortie

Appel textuel et graphique

```
My_Function (In1:= a, Out1+Out2=> d);          // non permis dans ST
My_Function (In1:= a, Out1 => Tmp1, Out2 => Tmp2);
d:= Tmp1 + Tmp2;
```

```

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

d) Appel d'une fonction sans résultat mais avec l'expression de variables de sortie

NOTE 2 Ces exemples décrivent deux représentations différentes de la même fonctionnalité. Il n'est pas nécessaire de prendre en charge une transformation automatique entre les deux formes de représentation.

6.6.2.4 Fonctions typées et en surcharge

Une fonction qui représente normalement un opérateur en surcharge doit être typée. Cela doit être effectué en ajoutant le caractère "_" (caractère de soulignement) suivi du type requis, comme décrit dans le Tableau 21. La fonction typée est exécutée en utilisant le type comme type de données pour ses entrées et sorties. Une conversion de type implicite ou explicite peut s'appliquer.

Une fonction de conversion en surcharge de la forme TO_XXX ou TRUNC_XXX, où XXX est le type de sortie élémentaire typé, peut être typée; il suffit pour cela de la faire précéder des données élémentaires requises et de la faire suivre d'un caractère de soulignement.

Tableau 21 – Fonctions typées et en surcharge

N°	Description	Exemple
1a	Fonction en surcharge ADD (ANY_Num à ANY_Num)	<pre> +-----+ ADD ANY_NUM -- -- ANY_NUM ANY_NUM -- . . ANY_NUM -- +-----+</pre>
1b	Conversion d'entrées ANY_ELEMENT TO_INT	<pre> +-----+ ANY_ELEMENTARY--- TO_INT ----INT +-----+</pre>

N°	Description	Exemple
2a ^a	Fonctions typées ADD_INT	<pre> +-----+ ADD_INT INT -- -- INT INT -- . -- . -- INT -- +-----+ </pre>
2b ^a	Conversion WORD_TO_INT	<pre> +-----+ WORD---- WORD_TO_INT ---INT +-----+ </pre>
NOTE La surcharge de fonctions ou de types bloc fonctionnel non normalisés ne relève pas du domaine d'application de la présente norme.		
^a Si la caractéristique 2 est prise en charge, l'Intégrateur fournit une table supplémentaire indiquant les fonctions qui sont en surcharge et celles qui sont typées dans la mise en œuvre.		

EXEMPLE 1 Fonctions typées et en surcharge

```

VAR
  A: INT;
  B: INT;
  C: INT;
END_VAR
C := A+B;

```

```

+----+
A --| + |-- C
B --|   |
+----+

```

NOTE 1 La conversion de type n'est pas requise dans l'exemple ci-dessus.

```

VAR
  A: INT;
  B: REAL;
  C: REAL;
END_VAR
C := INT_TO_REAL(A) + B;

```

```

+-----+ +----+
A --|INT_TO_REAL|--| + |-- C
+-----+ +----+
B -----|   |
+-----+ +----+
C := INT_TO_REAL(A) + B;

```

```

+-----+ +----+
A ---|TO_REAL|---|ADD|---C
+-----+ +----+
B -----|   |
+-----+ +----+
C := TO_REAL(A) + B;

```

```

VAR
  A: INT;
  B: INT;
  C: REAL;
END_VAR
C := INT_TO_REAL(A+B);

```

```

+----+ +-----+
A --| + |--|INT_TO_REAL|-- C
B --|   | +-----+
+----+ +-----+
C := INT_TO_REAL(A+B);

```

```

+----+ +-----+
A ---|ADD|---|TO_REAL|-- C
B ---|   | +-----+
+----+ +-----+
C := TO_REAL(A+B);

```

a) Déclaration de type (ST)

b) Utilisation (FBD et ST)

EXEMPLE 2 Conversion de type explicite et implicite avec fonctions typées

```

VAR
  A: INT;
  B: INT;
  C: INT;
END_VAR
C := ADD_INT(A, B);

```

```

+-----+
A---| ADD_INT |---C
B---|   |
+-----+

```

NOTE 2 La conversion de type n'est pas requise dans l'exemple ci-dessus.

```
VAR
  A: INT;
  B: REAL;
  C: REAL;
END_VAR
```

Conversion de type explicite

```

+-----+ +-----+
A--| INT_TO_REAL |--| ADD_REAL |-- C
+-----+ +-----+
B--|               |
+-----+ +-----+
```

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

```
VAR
  A: INT;
  B: REAL;
  C: REAL;
END_VAR
```

Conversion de type implicite

```

+-----+
A -----| ADD_REAL |-- C
+-----+
B -----|
+-----+
```

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

```
VAR
  A: INT;
  B: INT;
  C: REAL;
END_VAR
```

Conversion de type explicite

```

+-----+ +-----+
A --| ADD_INT |--| INT_TO_REAL |-- C
+-----+ +-----+
B --|
+-----+
```

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

```
VAR
  A: INT;
  B: INT;
  C: REAL;
END_VAR
```

Conversion de type implicite

```

+-----+
A --| ADD_INT |-- C
+-----+
B --|
+-----+
```

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

a) Déclaration de type (ST)

b) Utilisation (FBD et ST)

6.6.2.5 Fonctions normalisées

6.6.2.5.1 Généralités

Une fonction normalisée spécifiée dans le présent paragraphe comme étant extensible est autorisée à avoir deux entrées ou plus auxquelles l'opération indiquée doit être appliquée. Par exemple, une addition extensible doit produire à sa sortie la somme de toutes ses entrées. Le nombre maximal d'entrées d'une fonction extensible est spécifique de l'Intégrateur. Le nombre réel d'entrées effectives dans un appel formel d'une fonction extensible est déterminé par le nom d'entrée formel occupant la position la plus élevée dans la séquence des noms de variable.

EXEMPLE 1

L'énoncé `X:= ADD(Y1, Y2, Y3);`
est équivalent à `X:= ADD(IN1:= Y1, IN2:= Y2, IN3:= Y3);`

EXEMPLE 2

L'énoncé `I:= MUX_INT(K:=3, IN0:= 1, IN2:= 2, IN4:= 3);`
est équivalent à `I:= 0;`

6.6.2.5.2 Fonctions de conversion de type de données

Comme décrit dans le Tableau 22, les fonctions de conversion de type doit avoir la forme `*_TO_*`, où `""` est le type de la variable d'entrée `IN` et `""` le type de la variable de sortie `OUT`, par exemple, `INT_TO_REAL`. Les effets des conversions de type sur la précision et les types d'erreur qui peuvent survenir pendant l'exécution d'opérations de conversion de type sont spécifiques de l'Intégrateur.

Tableau 22 – Fonction de conversion de type de données

N°	Description	Forme graphique	Exemple d'utilisation
1a	Conversion typée input_TO_output	<pre> +-----+ B --- * _TO_ ** --- A +-----+ </pre> (*) - Type de données d'entrée, par exemple, INT (**) - Type de données de sortie, par exemple, REAL	A := INT_TO_REAL (B) ;
1b ^{a,b,e}	Conversion en surcharge TO_output	<pre> +-----+ B --- TO ** --- A +-----+ </pre> - Type de données d'entrée, par exemple, INT (**) - Type de données de sortie, par exemple, REAL	A := TO_REAL (B) ;
2a ^c	Troncature en surcharge "ancienne" TRUNC	<pre> +-----+ ANY_REAL --- TRUNC --- ANY_INT +-----+ </pre>	Déconseillée
2b ^c	Troncature typée input_TRUNC_output	<pre> +-----+ ANY_REAL --- * _TRUNC ** --- ANY_INT +-----+ </pre>	A := REAL_TRUNC_INT (B) ;
2c ^c	Troncature en surcharge TRUNC_output	<pre> +-----+ ANY_REAL --- TRUNC ** --- ANY_INT +-----+ </pre>	A := TRUNC_INT (B) ;
3a ^d	Typé input_BCD_TO_output	<pre> +-----+ * --- * _BCD_TO_ ** --- ** +-----+ </pre>	A := WORD_BCD_TO_INT (B) ;
3b ^d	En surcharge BCD_TO_output	<pre> +-----+ * --- BCD_TO ** --- ** +-----+ </pre>	A := BCD_TO_INT (B) ;
4a ^d	Typé input_TO_BCD_output	<pre> +-----+ ** --- ** _TO_BCD_ * --- * +-----+ </pre>	A := INT_TO_BCD_WORD (B) ;
4b ^d	En surcharge TO_BCD_output	<pre> +-----+ * --- TO_BCD ** --- ** +-----+ </pre>	A := TO_BCD_WORD (B) ;

NOTE Les exemples d'utilisation sont présentés en langage ST.

^a Une déclaration de conformité à la caractéristique 1 du présent tableau doit comprendre une liste des conversions de type spécifiques prises en charge et une déclaration des effets de l'exécution de chaque conversion.

^b La conversion du type REAL ou LREAL en SINT, INT, DINT ou LINT doit effectuer un arrondi conformément à la convention de la CEI 60559, selon laquelle si les deux entiers les plus proches sont également proches, le résultat doit être l'entier pair le plus proche, par exemple:

REAL_TO_INT (1.6) est équivalent à 2
 REAL_TO_INT (-1.6) est équivalent à -2

REAL_TO_INT (1.5) est équivalent à 2
 REAL_TO_INT (-1.5) est équivalent à -2

REAL_TO_INT (1.4) est équivalent à 1
 REAL_TO_INT (-1.4) est équivalent à -1

REAL_TO_INT (2.5) est équivalent à 2
 REAL_TO_INT (-2.5) est équivalent à -2.

^c La fonction TRUNC_* est utilisée pour la troncature vers zéro d'un élément REAL ou LREAL de manière à produire une variable d'un des types entiers, par exemple:

TRUNC_INT (1.6) est équivalent à INT#1
 TRUNC_INT (-1.6) est équivalent à INT#-1

TRUNC_SINT (1.4) est équivalent à SINT#1
 TRUNC_SINT (-1.4) est équivalent à SINT#-1.

N°	Description	Forme graphique	Exemple d'utilisation
d	Les fonctions de conversion <code>*_BCD_TO_*</code> et <code>**_TO_BCD_*</code> doivent effectuer des conversions entre des variables de type <code>BYTE</code> , <code>WORD</code> , <code>DWORD</code> et <code>LWORD</code> , et des variables de type <code>USINT</code> , <code>UINT</code> , <code>UDINT</code> et <code>ULINT</code> (représentées par <code>""</code> et <code>****</code> , respectivement) lorsque les variables de chaîne de bits correspondantes contiennent des données codées au format BCD. Par exemple, la valeur de <code>USINT_TO_BCD_BYTE(25)</code> serait <code>2#0010_0101</code> et la valeur de <code>WORD_BCD_TO_UINT (2#0011_0110_1001)</code> serait <code>369</code> .		
e	Lorsqu'une entrée ou une sortie d'une fonction de conversion de type est de type <code>STRING</code> ou <code>WSTRING</code> , les données de chaîne de caractères doivent être conformes à la représentation externe des données correspondantes, comme spécifié en 0, dans le jeu de caractères défini en 6.1.1.		

6.6.2.5.3 Conversion de type de données des types de données numériques

La conversion des types de données numériques utilise les règles suivantes:

1. Le type de données source est étendu au type de données le plus grand de la catégorie de type de données contenant sa valeur.
2. Le résultat est ensuite converti vers le type de données le plus grand de la catégorie de type de données à laquelle appartient le type de données cible.
3. Le résultat est ensuite converti vers le type de données cible.

Si la valeur de la variable source ne correspond pas au type de données cible, c'est-à-dire que la plage de valeurs est trop faible, la valeur de la variable cible est spécifique de l'Intégrateur.

NOTE La mise en œuvre de la fonction de conversion peut utiliser une procédure plus efficace.

EXEMPLE

`X:= REAL_TO_INT (70_000.4)`

1. La valeur `REAL (70_000.4)` est convertie en valeur `LREAL (70_000.400_000..)`.
2. La valeur `LREAL (70_000.4000_000..)` est convertie en valeur `LINT (70_000)`. Elle est ici arrondie sous la forme d'un entier.
3. La valeur `LINT (70_000)` est convertie en valeur `INT`. Elle est ici spécifique de l'Intégrateur car `INT` peut contenir au maximum `65.536`.

Cela donne une variable du type de données cible qui contient la même valeur que la variable source, si le type de données cible peut contenir cette valeur. Lors de la conversion d'un nombre à virgule flottante, les règles normales d'arrondi sont appliquées, c'est-à-dire arrondi à l'entier le plus proche et, en cas d'ambiguïté, à l'entier pair le plus proche.

Le type de données `BOOL` utilisé en tant que type de données source est traité comme un type de données entier non signé qui ne peut contenir que les valeurs 0 et 1.

Le Tableau 23 décrit les fonctions de conversion avec des détails de conversion résultant des règles ci-dessus.

Tableau 23 – Conversion de type de données des types de données numériques

N°	Fonction de conversion	Détails de la conversion
1	<code>LREAL _TO_ REAL</code>	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
2	<code>LREAL _TO_ LINT</code>	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
3	<code>LREAL _TO_ DINT</code>	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur

N°	Fonction de conversion	Détails de la conversion
4	LREAL _TO_ INT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
5	LREAL _TO_ SINT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
6	LREAL _TO_ ULINT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
7	LREAL _TO_ UDINT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
8	LREAL _TO_ UINT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
9	LREAL _TO_ USINT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
10	REAL _TO_ LREAL	Conversion avec maintien de la valeur
11	REAL _TO_ LINT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
12	REAL _TO_ DINT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
13	REAL _TO_ INT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
14	REAL _TO_ SINT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
15	REAL _TO_ ULINT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
16	REAL _TO_ UDINT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
17	REAL _TO_ UINT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
18	REAL _TO_ USINT	Conversion avec arrondi. Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
19	LINT _TO_ LREAL	Conversion avec perte potentielle de précision
20	LINT _TO_ REAL	Conversion avec perte potentielle de précision
21	LINT _TO_ DINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
22	LINT _TO_ INT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
23	LINT _TO_ SINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
24	LINT _TO_ ULINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
25	LINT _TO_ UDINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
26	LINT _TO_ UINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
27	LINT _TO_ USINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
28	DINT _TO_ LREAL	Conversion avec maintien de la valeur
29	DINT _TO_ REAL	Conversion avec perte potentielle de précision
30	DINT _TO_ LINT	Conversion avec maintien de la valeur
31	DINT _TO_ INT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
32	DINT _TO_ SINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
33	DINT _TO_ ULINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
34	DINT _TO_ UDINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
35	DINT _TO_ UINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
36	DINT _TO_ USINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
37	INT _TO_ LREAL	Conversion avec maintien de la valeur
38	INT _TO_ REAL	Conversion avec maintien de la valeur
39	INT _TO_ LINT	Conversion avec maintien de la valeur
40	INT _TO_ DINT	Conversion avec maintien de la valeur

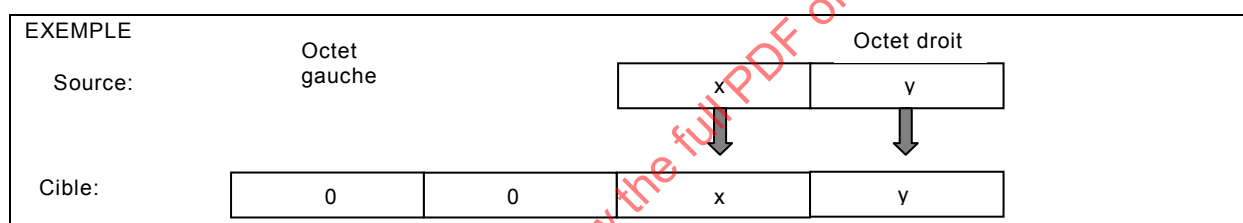
N°	Fonction de conversion	Détails de la conversion
41	INT _TO_ SINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
42	INT _TO_ ULINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
43	INT _TO_ UDINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
44	INT _TO_ UINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
45	INT _TO_ USINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
46	SINT _TO_ LREAL	Conversion avec maintien de la valeur
47	SINT _TO_ REAL	Conversion avec maintien de la valeur
48	SINT _TO_ LINT	Conversion avec maintien de la valeur
49	SINT _TO_ DINT	Conversion avec maintien de la valeur
50	SINT _TO_ INT	Conversion avec maintien de la valeur
51	SINT _TO_ ULINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
52	SINT _TO_ UDINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
53	SINT _TO_ UINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
54	SINT _TO_ USINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
55	ULINT _TO_ LREAL	Conversion avec perte potentielle de précision
56	ULINT _TO_ REAL	Conversion avec perte potentielle de précision
57	ULINT _TO_ LINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
58	ULINT _TO_ DINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
59	ULINT _TO_ INT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
60	ULINT _TO_ SINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
61	ULINT _TO_ UDINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
62	ULINT _TO_ UINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
63	ULINT _TO_ USINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
64	UDINT _TO_ LREAL	Conversion avec maintien de la valeur
65	UDINT _TO_ REAL	Conversion avec perte potentielle de précision
66	UDINT _TO_ LINT	Conversion avec maintien de la valeur
67	UDINT _TO_ DINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
68	UDINT _TO_ INT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
69	UDINT _TO_ SINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
70	UDINT _TO_ ULINT	Conversion avec maintien de la valeur
71	UDINT _TO_ UINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
72	UDINT _TO_ USINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
73	UINT _TO_ LREAL	Conversion avec maintien de la valeur
74	UINT _TO_ REAL	Conversion avec maintien de la valeur
75	UINT _TO_ LINT	Conversion avec maintien de la valeur
76	UINT _TO_ DINT	Conversion avec maintien de la valeur
77	UINT _TO_ INT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
78	UINT _TO_ SINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
79	UINT _TO_ ULINT	Conversion avec maintien de la valeur
80	UINT _TO_ UDINT	Conversion avec maintien de la valeur
81	UINT _TO_ USINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
82	USINT _TO_ LREAL	Conversion avec maintien de la valeur
83	USINT _TO_ REAL	Conversion avec maintien de la valeur
84	USINT _TO_ LINT	Conversion avec maintien de la valeur
85	USINT _TO_ DINT	Conversion avec maintien de la valeur

N°	Fonction de conversion	Détails de la conversion
86	USINT _TO_ INT	Conversion avec maintien de la valeur
87	USINT _TO_ SINT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur
88	USINT _TO_ ULINT	Conversion avec maintien de la valeur
89	USINT _TO_ UDINT	Conversion avec maintien de la valeur
90	USINT _TO_ UINT	Conversion avec maintien de la valeur

6.6.2.5.4 Conversion de type de données des types de données binaires

Cette conversion de type de données utilise les règles suivantes:

1. La conversion de type de données est effectuée comme un transfert binaire.
2. Si le type de données source est plus petit que le type de données cible, la valeur source est stockée dans les octets les plus à droite de la variable cible et les octets les plus à gauche sont définis à zéro.
3. Si le type de données source est plus grand que le type de données cible, seuls les octets les plus à droite de la variable source sont stockés dans le type de données cible.



Le Tableau 24 décrit les fonctions de conversion avec des détails de conversion résultant des règles ci-dessus.

Tableau 24 – Conversion de type de données des types de données binaires

N°	Fonction de conversion		Détails de la conversion
1	LWORD	_TO_ DWORD	Transfert binaire des octets les plus à droite dans la cible
2	LWORD	_TO_ WORD	Transfert binaire des octets les plus à droite dans la cible
3	LWORD	_TO_ BYTE	Transfert binaire des octets les plus à droite dans la cible
4	LWORD	_TO_ BOOL	Transfert binaire du bit le plus à droite dans la cible
5	DWORD	_TO_ LWORD	Transfert binaire dans les octets les plus à droite de la cible. Les octets les plus à gauche de la cible sont définis à zéro
6	DWORD	_TO_ WORD	Transfert binaire des octets les plus à droite dans la cible
7	DWORD	_TO_ BYTE	Transfert binaire des octets les plus à droite dans la cible
8	DWORD	_TO_ BOOL	Transfert binaire du bit le plus à droite dans la cible
9	WORD	_TO_ LWORD	Transfert binaire dans les octets les plus à droite de la cible. Les octets les plus à gauche de la cible sont définis à zéro
10	WORD	_TO_ DWORD	Transfert binaire dans les octets les plus à droite de la cible. Les octets les plus à gauche de la cible sont définis à zéro
11	WORD	_TO_ BYTE	Transfert binaire des octets les plus à droite dans la cible
12	WORD	_TO_ BOOL	Transfert binaire du bit le plus à droite dans la cible
13	BYTE	_TO_ LWORD	Transfert binaire dans les octets les plus à droite de la cible. Les octets les plus à gauche de la cible sont définis à zéro
14	BYTE	_TO_ DWORD	Transfert binaire dans les octets les plus à droite de la cible. Les octets les plus à gauche de la cible sont définis à zéro

N°	Fonction de conversion			Détails de la conversion
15	BYTE	_TO_	WORD	Transfert binaire dans les octets les plus à droite de la cible. Les octets les plus à gauche de la cible sont définis à zéro
16	BYTE	_TO_	BOOL	Transfert binaire du bit le plus à droite dans la cible
17	BYTE	_TO_	CHAR	Transfert binaire
18	BOOL	_TO_	LWORD	Donne la valeur 16#0 ou 16#1
19	BOOL	_TO_	DWORD	Donne la valeur 16#0 ou 16#1
20	BOOL	_TO_	WORD	Donne la valeur 16#0 ou 16#1
21	BOOL	_TO_	BYTE	Donne la valeur 16#0 ou 16#1
22	CHAR	_TO_	BYTE	Transfert binaire
23	CHAR	_TO_	WORD	Transfert binaire dans les octets les plus à droite de la cible. Les octets les plus à gauche de la cible sont définis à zéro
24	CHAR	_TO_	DWORD	Transfert binaire dans les octets les plus à droite de la cible. Les octets les plus à gauche de la cible sont définis à zéro
25	CHAR	_TO_	LWORD	Transfert binaire dans les octets les plus à droite de la cible. Les octets les plus à gauche de la cible sont définis à zéro
26	WCHAR	_TO_	WORD	Transfert binaire
27	WCHAR	_TO_	DWORD	Transfert binaire dans les octets les plus à droite de la cible. Les octets les plus à gauche de la cible sont définis à zéro
28	WCHAR	_TO_	LWORD	Transfert binaire dans les octets les plus à droite de la cible. Les octets les plus à gauche de la cible sont définis à zéro

6.6.2.5.5 Conversion de type de données d'un type binaire en type numérique

Cette conversion de type de données utilise les règles suivantes:

1. La conversion de type de données est effectuée comme un transfert binaire.
2. Si le type de données source est plus petit que le type de données cible, la valeur source est stockée dans les octets les plus à droite de la variable cible et les octets les plus à gauche sont définis à zéro.

EXEMPLE 1 X: SINT:= 18; W: WORD; W:= SINT_TO_WORD(X); et W obtient 16#0012.

3. Si le type de données source est plus grand que le type de données cible, seuls les octets les plus à droite de la variable source sont stockés dans le type de données cible.

EXEMPLE 2 W: WORD:= 16#1234; X: SINT; X:= W; et X obtient 54 (=16#34).

Le Tableau 25 décrit les fonctions de conversion avec des détails de conversion résultant des règles ci-dessus.

Tableau 25 – Conversion de type de données des types binaires et numériques

N°	Fonction de conversion			Détails de la conversion
1	LWORD	_TO_	LREAL	Transfert binaire
2	DWORD	_TO_	REAL	Transfert binaire
3	LWORD	_TO_	LINT	Transfert binaire
4	LWORD	_TO_	DINT	Transfert binaire des octets les plus à droite dans la cible
5	LWORD	_TO_	INT	Transfert binaire des octets les plus à droite dans la cible
6	LWORD	_TO_	SINT	Transfert binaire de l'octet le plus à droite dans la cible
7	LWORD	_TO_	ULINT	Transfert binaire
8	LWORD	_TO_	UDINT	Transfert binaire des octets les plus à droite dans la cible

N°	Fonction de conversion			Détails de la conversion
9	LWORD	_TO_	UINT	Transfert binaire des octets les plus à droite dans la cible
10	LWORD	_TO_	USINT	Transfert binaire de l'octet le plus à droite dans la cible
11	DWORD	_TO_	LINT	Transfert binaire dans les octets les plus à droite de la cible
12	DWORD	_TO_	DINT	Transfert binaire
13	DWORD	_TO_	INT	Transfert binaire des octets les plus à droite dans la cible
14	DWORD	_TO_	SINT	Transfert binaire de l'octet le plus à droite dans la cible
15	DWORD	_TO_	ULINT	Transfert binaire dans les octets les plus à droite de la cible
16	DWORD	_TO_	UDINT	Transfert binaire
17	DWORD	_TO_	UINT	Transfert binaire des octets les plus à droite dans la cible
18	DWORD	_TO_	USINT	Transfert binaire de l'octet le plus à droite dans la cible
19	WORD	_TO_	LINT	Transfert binaire dans les octets les plus à droite de la cible
20	WORD	_TO_	DINT	Transfert binaire dans les octets les plus à droite de la cible
21	WORD	_TO_	INT	Transfert binaire
22	WORD	_TO_	SINT	Transfert binaire de l'octet le plus à droite dans la cible
23	WORD	_TO_	ULINT	Transfert binaire dans les octets les plus à droite de la cible
24	WORD	_TO_	UDINT	Transfert binaire dans les octets les plus à droite de la cible
25	WORD	_TO_	UINT	Transfert binaire
26	WORD	_TO_	USINT	Transfert binaire de l'octet le plus à droite dans la cible
27	BYTE	_TO_	LINT	Transfert binaire dans les octets les plus à droite de la cible
28	BYTE	_TO_	DINT	Transfert binaire dans les octets les plus à droite de la cible
29	BYTE	_TO_	INT	Transfert binaire dans les octets les plus à droite de la cible
30	BYTE	_TO_	SINT	Transfert binaire
31	BYTE	_TO_	ULINT	Transfert binaire dans les octets les plus à droite de la cible
32	BYTE	_TO_	UDINT	Transfert binaire dans les octets les plus à droite de la cible
33	BYTE	_TO_	UINT	Transfert binaire dans les octets les plus à droite de la cible
34	BYTE	_TO_	USINT	Transfert binaire
35	BOOL	_TO_	LINT	Donne la valeur 0 ou 1
36	BOOL	_TO_	DINT	Donne la valeur 0 ou 1
37	BOOL	_TO_	INT	Donne la valeur 0 ou 1
38	BOOL	_TO_	SINT	Donne la valeur 0 ou 1
39	BOOL	_TO_	ULINT	Donne la valeur 0 ou 1
40	BOOL	_TO_	UDINT	Donne la valeur 0 ou 1
41	BOOL	_TO_	UINT	Donne la valeur 0 ou 1
42	BOOL	_TO_	USINT	Donne la valeur 0 ou 1
43	LREAL	_TO_	LWORD	Transfert binaire
44	REAL	_TO_	DWORD	Transfert binaire
45	LINT	_TO_	LWORD	Transfert binaire
46	LINT	_TO_	DWORD	Transfert binaire des octets les plus à droite dans la cible
47	LINT	_TO_	WORD	Transfert binaire des octets les plus à droite dans la cible
48	LINT	_TO_	BYTE	Transfert binaire de l'octet le plus à droite dans la cible
49	DINT	_TO_	LWORD	Transfert binaire dans les octets les plus à droite de la cible; reste = 0

N°	Fonction de conversion			Détails de la conversion
50	DINT	_TO_	DWORD	Transfert binaire
51	DINT	_TO_	WORD	Transfert binaire des octets les plus à droite dans la cible
52	DINT	_TO_	BYTE	Transfert binaire de l'octet le plus à droite dans la cible
53	INT	_TO_	LWORD	Transfert binaire dans les octets les plus à droite de la cible; reste = 0
54	INT	_TO_	DWORD	Transfert binaire dans les octets les plus à droite de la cible; reste = 0
55	INT	_TO_	WORD	Transfert binaire
56	INT	_TO_	BYTE	Transfert binaire de l'octet le plus à droite dans la cible
57	SINT	_TO_	LWORD	Transfert binaire dans les octets les plus à droite de la cible; reste = 0
58	SINT	_TO_	DWORD	Transfert binaire dans les octets les plus à droite de la cible; reste = 0
59	SINT	_TO_	WORD	Transfert binaire
60	SINT	_TO_	BYTE	Transfert binaire
61	ULINT	_TO_	LWORD	Transfert binaire
62	ULINT	_TO_	DWORD	Transfert binaire des octets les plus à droite dans la cible
63	ULINT	_TO_	WORD	Transfert binaire des octets les plus à droite dans la cible
64	ULINT	_TO_	BYTE	Transfert binaire de l'octet le plus à droite dans la cible
65	UDINT	_TO_	LWORD	Transfert binaire dans les octets les plus à droite de la cible; reste = 0
66	UDINT	_TO_	DWORD	Transfert binaire
67	UDINT	_TO_	WORD	Transfert binaire des octets les plus à droite dans la cible
68	UDINT	_TO_	BYTE	Transfert binaire de l'octet le plus à droite dans la cible
69	UINT	_TO_	LWORD	Transfert binaire dans les octets les plus à droite de la cible; reste = 0
70	UINT	_TO_	DWORD	Transfert binaire dans les octets les plus à droite de la cible; reste = 0
71	UINT	_TO_	WORD	Transfert binaire
72	UINT	_TO_	BYTE	Transfert binaire de l'octet le plus à droite dans la cible
73	USINT	_TO_	LWORD	Transfert binaire dans les octets les plus à droite de la cible; reste = 0
74	USINT	_TO_	DWORD	Transfert binaire dans les octets les plus à droite de la cible; reste = 0
75	USINT	_TO_	WORD	Transfert binaire
76	USINT	_TO_	BYTE	Transfert binaire

6.6.2.5.6 Conversion de type de données des types date et heure

Le Tableau 26 décrit la conversion de type de données des types date et heure.

Tableau 26 – Conversion de type de données des types date et heure

N°	Fonction de conversion			Détails de la conversion
1	LTIME	_TO_	TIME	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur et une perte de précision peut se produire.
2	TIME	_TO_	LTIME	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur et une perte de précision peut se produire.
3	LDT	_TO_	DT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur et une perte de précision peut se produire.
4	LDT	_TO_	DATE	Convertit uniquement la date contenue. Une erreur de plage de valeurs donne un résultat spécifique de l'Intégrateur.
5	LDT	_TO_	LTOD	Convertit uniquement l'heure contenue.
6	LDT	_TO_	TOD	Convertit uniquement l'heure contenue. Une perte de précision peut se produire.
7	DT	_TO_	LDT	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur et une perte de précision peut se produire.
8	DT	_TO_	DATE	Convertit uniquement la date contenue. Une erreur de plage de valeurs donne un résultat spécifique de l'Intégrateur.
9	DT	_TO_	LTOD	Convertit uniquement l'heure contenue. Une erreur de plage de valeurs donne un résultat spécifique de l'Intégrateur.
10	DT	_TO_	TOD	Convertit uniquement l'heure contenue. Une erreur de plage de valeurs donne un résultat spécifique de l'Intégrateur.
11	LTOD	_TO_	TOD	Conversion avec maintien de la valeur
12	TOD	_TO_	LTOD	Les erreurs de plage de valeurs donnent un résultat spécifique de l'Intégrateur et une perte de précision peut se produire.

6.6.2.5.7 Conversion de type de données des types caractère

Le Tableau 27 décrit la conversion de type de données des types caractère.

Tableau 27 – Conversion de type de données des types caractère

N°	Fonction de conversion			Détails de la conversion
1	WSTRIN G	_TO_	STRING	Les caractères qui sont pris en charge par l'Intégrateur avec le type de données STRING sont convertis. Les autres sont convertis en éléments dépendants de l'Intégrateur.
2	WSTRIN G	_TO_	WCHAR	Le premier caractère de la chaîne est transféré. Si la chaîne est vide, la variable cible est indéfinie.
3	STRING	_TO_	WSTRING	Convertit les caractères de la chaîne comme défini par l'Intégrateur en caractères ISO/CEI 10646 (UTF-16) appropriés.
4	STRING	_TO_	CHAR	Le premier caractère de la chaîne est transféré. Si la chaîne est vide, la variable cible est indéfinie.
5	WCHAR	_TO_	WSTRING	Produit une chaîne d'une taille réelle d'un caractère.
6	WCHAR	_TO_	CHAR	Les caractères qui sont pris en charge par l'Intégrateur avec le type de données CHAR sont convertis. Les autres sont convertis en éléments spécifiques de l'Intégrateur.
7	CHAR	_TO_	STRING	Produit une chaîne d'une taille réelle d'un caractère.

N°	Fonction de conversion			Détails de la conversion
8	CHAR	_TO_	WCHAR	Convertit un caractère tel que défini par l'Intégrateur en caractère UTF-16 approprié.

6.6.2.5.8 Fonctions numériques et arithmétiques

La représentation graphique normalisée, les noms de fonction, les types de variable d'entrée et de sortie, et les descriptions des fonctions d'une variable numérique unique doivent être tels que définis dans le Tableau 28. Ces fonctions doivent être en surcharge sur les types génériques définis et peuvent être typées. Pour ces fonctions, les types d'entrée et de sortie doivent être identiques.

La représentation graphique normalisée, les noms et symboles de fonction, et les descriptions des fonctions arithmétiques de deux variables ou plus doivent être tels que décrits dans le Tableau 29. Ces fonctions doivent être en surcharge sur tous les types numériques et peuvent être typées.

La précision des fonctions numériques doit être exprimée en termes d'une ou plusieurs dépendances spécifiques de l'Intégrateur.

Une erreur est générée si le résultat de l'évaluation d'une de ces fonctions dépasse la plage de valeurs spécifiée pour le type de données de la sortie de la fonction ou si une division par zéro est tentée.

Tableau 28 – Fonctions numériques et arithmétiques

N°	Description (nom de fonction)	Type d'E/S	Explication
	Forme graphique <pre> +-----+ * -- ** -- * +-----+ </pre> (*) - Type d'entrée-sortie (E/S) (**) - Nom de fonction		Exemple d'utilisation dans ST <pre> A:= SIN(B) ; (langage ST) </pre>
	Fonctions générales		
1	ABS (x)	ANY_NUM	Valeur absolue
2	SQRT (x)	ANY_REAL	Racine carrée
	Fonctions logarithmiques		
3	LN (x)	ANY_REAL	Logarithme naturel
4	LOG (x)	ANY_REAL	Logarithme en base 10
5	EXP (x)	ANY_REAL	Exponentiel naturel
	Fonctions trigonométriques		
6	SIN (x)	ANY_REAL	Sinus de l'entrée en radians
7	COS (x)	ANY_REAL	Cosinus en radians
8	TAN (x)	ANY_REAL	Tangente en radians
9	ASIN (x)	ANY_REAL	Arc-sinus principal
10	ACOS (x)	ANY_REAL	Arc-cosinus principal
11	ATAN (x)	ANY_REAL	Arc-tangente principal

12	<pre> +-----+ ATAN2 ----- ANY_REAL-- Y --ANY_REAL ANY_REAL-- X ----- </pre>	ANY_REAL	Angle entre la partie positive de l'axe x d'un plan et le point défini sur celui-ci par les coordonnées (x, y). L'angle est positif pour les angles antihoraires (demi-plan supérieur, y > 0) et négatif pour les angles horaires (demi-plan inférieur, y < 0).
----	--	----------	---

Tableau 29 – Fonctions arithmétiques

N° a,b	Description	Nom	Symbole (opérateur)	Explication
	Forme graphique			Exemple d'utilisation dans ST
	<pre> +-----+ ANY_NUM -- *** -- ANY_NUM ANY_NUM -- . -- . -- ANY_NUM -- ----- </pre> (***) - Nom ou symbole			en tant qu'appel de fonction A := ADD (B, C, D); ou en tant qu'opérateur (symbole) A := B + C + D;
	Fonctions arithmétiques extensibles			
1 ^c	Addition	ADD	+	OUT := IN1 + IN2 + ... + INn
2	Multiplication	MUL	*	OUT := IN1 * IN2 * ... * INn
	Fonctions arithmétiques non extensibles			
3 ^c	Soustraction	SUB	-	OUT := IN1 - IN2
4 ^d	Division	DIV	/	OUT := IN1 / IN2
5 ^e	Modulo	MOD		OUT := IN1 modulo IN2
6 ^f	Exponentiation	EXPT	**	OUT := IN1 ^{IN2}
7 ^g	Déplacement	MOVE	:=	OUT := IN
NOTE 1 Les entrées non vides de la colonne Symbole sont adaptées à une utilisation en tant qu'opérateurs dans les langages textuels. NOTE 2 Les notations IN1, IN2, ..., INn désignent les entrées dans l'ordre de haut en bas; OUT désigne la sortie. NOTE 3 Les exemples d'utilisation et les descriptions sont présentés en langage ST.				
a	Lorsque la représentation d'une fonction est prise en charge avec un nom, cela est indiqué par le suffixe "n" dans la déclaration de conformité. Par exemple, "1n" représente la notation "ADD".			
b	Lorsque la représentation d'une fonction est prise en charge avec un symbole, cela est indiqué par le suffixe "s" dans la déclaration de conformité. Par exemple, "1s" représente la notation "+".			
c	Le type générique des entrées et sorties de ces fonctions est ANY_MAGNITUDE.			
d	Le résultat d'une division d'entiers doit être un entier du même type avec troncature vers zéro; par exemple, 7/3 = 2 et (-7)/3 = -2.			
e	IN1 et IN2 doivent être du type générique ANY_INT pour cette fonction. Le résultat de l'évaluation de cette fonction MOD doit être l'équivalent de l'exécution des énoncés suivants dans ST:			
	<pre> IF (IN2 = 0) THEN OUT:=0; ELSE OUT:=IN1 - (IN1/IN2)*IN2; END_IF </pre>			
f	IN1 doit être de type ANY_REAL et IN2 de type ANY_NUM pour cette fonction EXPT. La sortie doit être du même type que IN1.			
g	La fonction MOVE a exactement une entrée (IN) de type ANY et une sortie (OUT) de type ANY.			

6.6.2.5.9 Fonctions booléennes de chaîne de bits et au niveau du bit

La représentation graphique normalisée, les noms de fonction et les descriptions des fonctions de décalage d'une variable de chaîne de bits unique doivent être tels que définis dans le Tableau 30. Ces fonctions doivent être en surcharge sur tous les types de chaîne de bits et peuvent être typées.

La représentation graphique normalisée, les noms et symboles de fonction, et les descriptions des fonctions booléennes au niveau du bit doivent être tels que définis dans le

Tableau 31. Ces fonctions doivent être extensibles, sauf pour NOT, et en surcharge sur tous les types de chaîne de bits, et peuvent être typées.

Tableau 30 – Fonctions de décalage de bit

N°	Description	Nom	Explication
	Forme graphique		Exemple d'utilisation ^a
	<pre> +-----+ *** ANY_BIT -- IN -- ANY_BIT ANY_INT -- N +-----+ </pre> (***) - Nom de fonction		<pre> A := SHL (IN:=B, N:=5); </pre> (langage ST)
1	Décalage à gauche	SHL	OUT:= IN décalé à gauche de N bits, rempli par des zéros à droite
2	Décalage à droite	SHR	OUT:= IN décalé à droite de N bits, rempli par des zéros à gauche
3	Rotation à gauche	ROL	OUT:= IN déplacé de manière circulaire à gauche de N bits
4	Rotation à droite	ROR	OUT:= IN déplacé de manière circulaire à droite de N bits
NOTE 1 La notation OUT désigne la sortie de la fonction. EXEMPLE IN:= 2#0001_1001 de 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 de type BOOL (un bit) n'a pas de sens.			
^a Une erreur est générée si la valeur de l'entrée N est inférieure à zéro.			

Tableau 31 – Fonctions booléennes au niveau du bit

N° a,b	Description	Nom	Symbole	Explication (NOTE 3)
	Forme graphique			Exemples d'utilisation (NOTE 5)

	<pre> +-----+ ANY_BIT -- *** -- ANY_BIT ANY_BIT -- : : ANY_BIT -- +-----+ </pre> (***) - Nom ou symbole			<pre> A:= AND(B, C, D); ou A:= B & C & D; </pre>
1	Et	AND	& (NOTE 1)	OUT:= IN1 & IN2 &... & INn
2	Ou	OR	>=1 (NOTE 2)	OUT:= IN1 OR IN2 OR... OR INn
3	Ou exclusif	XOR	=2k+1 (NOTE 2)	OUT:= IN1 XOR IN2 XOR... XOR INn
4	Non	NOT		OUT:= NOT IN1 (NOTE 4)
NOTE 1 Ce symbole est adapté à une utilisation en tant qu'opérateur dans les langages textuels, comme décrit dans le Tableau 68 et le Tableau 71. NOTE 2 Ce symbole n'est pas adapté à une utilisation en tant qu'opérateur dans les langages textuels. NOTE 3 Les notations IN1, IN2, ..., INn désignent les entrées dans l'ordre de haut en bas; OUT désigne la sortie. NOTE 4 L'inversion graphique des signaux de type BOOL peut également être effectuée. NOTE 5 Les exemples d'utilisation et les descriptions sont présentés en langage ST. ^a Lorsque la représentation d'une fonction est prise en charge avec un nom, cela doit être indiqué par le suffixe "n" dans la déclaration de conformité. Par exemple, "1n" représente la notation "AND". ^b Lorsque la représentation d'une fonction est prise en charge avec un symbole, cela doit être indiqué par le suffixe "s" dans la déclaration de conformité. Par exemple, "1s" représente la notation "&".				

6.6.2.5.10 Fonctions de sélection et de comparaison

Les fonctions de sélection et de comparaison doivent être en surcharge sur tous les types de données. Les représentations graphiques normalisées, les noms de fonction et les descriptions des fonctions de sélection doivent être tels que définis dans le Tableau 32.

La représentation graphique normalisée, les noms et symboles de fonction, et les descriptions des fonctions de comparaison doivent être tels que définis dans le Tableau 33. Toutes les fonctions de comparaison (sauf NE) doivent être extensibles.

Les comparaisons des données de chaîne de bits doivent être effectuées au niveau du bit, du bit le plus à gauche au bit le plus à droite, et les chaînes de bits les plus courtes doivent être considérées comme étant remplies sur la gauche par des zéros lorsqu'elles sont comparées à des chaînes de bits plus longues; en d'autres termes, la comparaison des variables de chaîne de bits doit avoir le même résultat que la comparaison de variables entières non signées.

Tableau 32 – Fonctions de sélection ^d

N°	Description	Nom	Forme graphique	Explication/Exemple
1	Déplacement ^{a, d} (affectation)	MOVE	<pre> +-----+ MOVE ANY -- -- ANY +-----+ </pre>	OUT:= IN
2	Sélection binaire ^d	SEL	<pre> +-----+ SEL BOOL -- G -- ANY ANY -- IN0 ANY -- IN1 +-----+ </pre>	OUT:= IN0 si G = 0 OUT:= IN1 si G = 1 EXEMPLE 1 A:= SEL (G := 0, IN0:= X, IN1:= 5);

3	Fonction maximale extensible	MAX	<pre> +-----+ MAX ANY_ELEMENTARY -- -- ANY_ELEMENTARY : -- ANY_ELEMENTARY -- +-----+ </pre>	OUT:= MAX(IN1, IN2, ..., INn); EXEMPLE 2 A:= MAX(B, C, D);
4	Fonction minimale extensible	MIN	<pre> +-----+ MIN ANY_ELEMENTARY -- -- ANY_ELEMENTARY : -- ANY_ELEMENTARY -- +-----+ </pre>	OUT:= MIN(IN1, IN2, ..., INn) EXEMPLE 3 A:= MIN(B, C, D);
5	Limiteur	LIMIT	<pre> +-----+ LIMIT ANY_ELEMENTARY -- MN -- ANY_ELEMENTARY ANY_ELEMENTARY -- IN -- ANY_ELEMENTARY ANY_ELEMENTARY -- MX -- ANY_ELEMENTARY +-----+ </pre>	OUT:= MIN (MAX(IN, MN), MX); EXEMPLE 4 A:= LIMIT(IN:= B, MN:= 0, MX:= 5);
6	Multi- plexeur ^{b, c, d, e} extensible	MUX	<pre> +-----+ MUX ANY_ELEMENTARY -- K -- ANY_ELEMENTARY ANY_ELEMENTARY -- -- ANY_ELEMENTARY ANY_ELEMENTARY -- -- ANY_ELEMENTARY +-----+ </pre>	a, b, c: La sélection d'une des N entrées en fonction de l'entrée K EXEMPLE 5 A:= MUX(0, B, C, D); aurait le même effet que A:= B;

NOTE 1 Les notations IN1, IN2, ..., INn désignent les entrées dans l'ordre de haut en bas; OUT désigne la sortie.

NOTE 2 Les exemples d'utilisation et les descriptions sont présentés en langage ST.

a La fonction MOVE a exactement une entrée IN de type ANY et une sortie OUT de type ANY.

b Les entrées non nommées dans la fonction MUX doivent avoir les noms par défaut IN0, IN1, ..., INn-1 dans l'ordre de haut en bas, où n est le nombre total de ces entrées. Ces noms peuvent, si nécessaire, être présentés dans la représentation graphique.

c La fonction MUX peut être typée sous la forme MUX_*_**, où * est le type de l'entrée K, et ** le type des autres entrées et de la sortie.

d L'Intégrateur peut, si nécessaire, prendre en charge la sélection parmi les variables des types de données définis par l'utilisateur, pour revendiquer la conformité à cette caractéristique.

e Une erreur est générée si la valeur réelle de l'entrée K de la fonction MUX n'est pas dans la plage {0 ... n-1}.

Tableau 33 – Fonctions de comparaison

N°	Description	Nom ^a	Symbole ^b	Explication (pour 2 opérandes extensibles ou plus)
	Forme graphique			Exemples d'utilisation
	<pre> +-----+ ANY_ELEMENTARY -- *** -- BOOL : -- ANY_ELEMENTARY -- +-----+ </pre> (***) Nom ou symbole			A:= GT(B, C, D); // Nom de fonction ou A:= (B>C) & (C>D); // Symbole
1	Séquence décroissante	GT	>	OUT:= (IN1>IN2) & (IN2>IN3) & .. & (INn-1 > INn)

2	Séquence monotone	GE	>=	OUT:= (IN1>=IN2) & (IN2>=IN3) &... & (INn-1 >= INn)
3	Egalité	EQ	=	OUT:= (IN1=IN2) & (IN2=IN3) &... & (INn-1 = INn)
4	Séquence monotone	LE	<=	OUT:= (IN1<=IN2) & (IN2<=IN3) &... & (INn-1 <= INn)
5	Séquence croissante	LT	<	OUT:= (IN1<IN2) & (IN2<IN3) &... & (INn-1 < INn)
6	Inégalité	NE	<>	OUT:= (IN1<>IN2) (non extensible)

NOTE 1 Les notations IN1, IN2, ..., INn désignent les entrées dans l'ordre de haut en bas; OUT désigne la sortie.

NOTE 2 Tous les symboles décrits dans le présent tableau sont adaptés à une utilisation en tant qu'opérateurs dans les langages textuels.

NOTE 3 Les exemples d'utilisation et les descriptions sont présentés en langage ST.

NOTE 4 Les fonctions de comparaison normalisées peuvent être définies de façon dépendante du langage, par exemple, à contacts.

^a Lorsque la représentation d'une fonction est prise en charge avec un nom, cela doit être indiqué par le suffixe "n" dans la déclaration de conformité. Par exemple, "1n" représente la notation "ST".

^b Lorsque la représentation d'une fonction est prise en charge avec un symbole, cela doit être indiqué par le suffixe "s" dans la déclaration de conformité. Par exemple, "1s" représente la notation ">".

6.6.2.5.11 Fonctions de chaîne de caractères

Le Tableau 33 doit être applicable aux chaînes de caractères. Une variable de type de données CHAR ou WCHAR peut être utilisée à la place d'une chaîne de caractère unique.

Pour la comparaison de deux chaînes de longueurs différentes, la chaîne la plus courte doit être considérée comme étant étendue sur la droite jusqu'à la longueur de la chaîne la plus longue par des caractères de valeur zéro. La comparaison doit être effectuée de gauche à droite, sur la base de la valeur numérique des codes de caractère dans le jeu de caractères.

EXEMPLE

La chaîne de caractères "Z" est plus longue que la chaîne de caractères 'AZ' ('Z' > 'A') et 'AZ' est plus longue que 'ABC' ('A' = 'A' et 'Z' > 'B').

Les représentations graphiques normalisées, les noms de fonction et les descriptions de fonctions additionnelles des chaînes de caractères doivent être tels que décrits dans le Tableau 34. Pour ces opérations, les positions des caractères dans la chaîne doivent être considérées comme étant numérotées 1, 2, ..., L, en commençant par la position du caractère le plus à gauche, où L est la longueur de la chaîne.

Une erreur doit être générée si:

- la valeur réelle d'une entrée quelconque désignée par ANY_INT dans le Tableau 34 est inférieure à zéro;
- l'évaluation de la fonction conduit à une tentative (1) d'accès à une position de caractère inexistante dans une chaîne ou (2) de production d'une chaîne plus longue que la longueur de chaîne maximale spécifique de l'intégrateur;
- les arguments de type de données STRING ou CHAR et les arguments de type de données WSTRING ou WCHAR sont combinés dans la même fonction.

Tableau 34 – Fonctions de chaîne de caractères

N°	Description	Forme graphique	Exemple
1	Longueur de chaîne	<pre> +-----+ ANY_STRING-- LEN -- ANY_INT +-----+ </pre>	Longueur de chaîne A:= LEN('ASTRING'); est équivalent à A:= 7;
2	Gauche	<pre> +-----+ ANY_STRING-- LEN -- ANY_INT +-----+ </pre>	L caractères les plus à gauche de IN A:= LEFT(IN:='ASTR', L:=3); est équivalent à A:= 'AST';
3	Droite	<pre> +-----+ RIGHT ANY_STRING-- IN -- ANY_STRING ANY_INT -- L +-----+ </pre>	L caractères les plus à droite de IN A:= RIGHT(IN:='ASTR', L:=3); est équivalent à A:= 'STR';
4	Centre	<pre> +-----+ MID ANY_STRING-- IN -- ANY_STRING ANY_INT -- L ANY_INT -- P +-----+ </pre>	L caractères de IN, à partir de la Pème position de caractère A:= MID(IN:='ASTR', L:=2, P:=2); est équivalent à A:= 'ST';
5	Concaténation extensible	<pre> +-----+ CONCAT ANY_CHARS-- -- ANY_STRING : -- ANY_CHARS-- +-----+ </pre>	Concaténation extensible A:= CONCAT('AB','CD','E'); est équivalent à A:= 'ABCDE';
6	Insérer	<pre> +-----+ INSERT ANY_STRING-- IN1 -- ANY_STRING ANY_CHARS -- IN2 ANY_INT---- P +-----+ </pre>	Insertion de IN2 dans IN1 après la Pème position de caractère A:= INSERT(IN1:='ABC', IN2:='XY', P=2); est équivalent à A:= 'ABXYC';
7	Supprimer	<pre> +-----+ DELETE ANY_STRING-- IN -- ANY_STRING ANY_INT -- L ANY_INT -- P +-----+ </pre>	L caractères de IN, à partir de la Pème position de caractère A:= DELETE(IN:='ABXYC', L:=2, P:=3); est équivalent à A:= 'ABC';
8	Remplacer	<pre> +-----+ REPLACE ANY_STRING-- IN1 -- ANY_STRING ANY_CHARS -- IN2 ANY_INT -- L ANY_INT -- P +-----+ </pre>	Remplacement de L caractères de IN1 par IN2, à partir de la Pème position de caractère A:= REPLACE(IN1:='ABCDE', IN2:='X', L:=2, P:=3); est équivalent à A:= 'ABXE';
9	Rechercher	<pre> +-----+ FIND ANY_STRING-- IN1 -- ANY_INT ANY_CHARS -- IN2 +-----+ </pre>	Recherche de la position de caractère du début de la première occurrence de IN2 dans IN1. Si aucune occurrence de IN2 n'est trouvée, OUT:= 0. A:= FIND(IN1:='ABCBC', IN2:='BC'); est équivalent à A:= 2;

NOTE 1 Les exemples du présent tableau sont présentés en langage ST.

NOTE 2 Toutes les entrées de CONCAT sont ANY_CHARS, c'est-à-dire qu'elles peuvent aussi être de type CHAR ou WCHAR.

NOTE 3 L'entrée IN2 des fonctions INSERT, REPLACE, FIND est ANY_CHARS, c'est-à-dire qu'elle peut aussi être de type CHAR ou WCHAR.

6.6.2.5.12 Fonctions de date et de durée

En plus des fonctions de comparaison et de sélection, les combinaisons de types de données de temps et de durée d'entrée et de sortie décrites dans le Tableau 35 doivent être autorisées avec les fonctions associées.

Une erreur doit être générée si le résultat de l'évaluation d'une de ces fonctions dépasse la plage de valeurs spécifique de l'Intégrateur pour le type de données de sortie.

Tableau 35 – Fonctions numériques des types de données de temps et de durée

N°	Description (nom de fonction)	Sym- bole	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
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 Ces fonctions normalisées prennent en charge la surcharge mais uniquement dans les deux ensembles de types de données (TIME, DT, DATE, TOD) et (LTIME, LDT, DATE, LTOD).

EXEMPLE

Les énoncés en langage ST

```
X:= DT#1986-04-28-08:40:00;
Y:= DT_TO_TOD(X);
W:= DT_TO_DATE(X);
```

ont le même résultat que l'énoncé avec des données "extraites".

```
X:= DT#1986-04-28-08:40:00;
Y:= TIME_OF_DAY#08:40:00;
W:= DATE#1986-04-28;
```

Des fonctions de concaténation et de séparation, comme indiqué au Tableau 36, sont définies pour gérer la date et l'heure. De plus, une fonction permettant d'extraire le jour de la semaine est définie.

Une erreur doit être générée si le résultat de l'évaluation d'une de ces fonctions dépasse la plage de valeurs spécifique de l'Intégrateur pour le type de données de sortie.

Tableau 36 – Fonctions additionnelles des types de données de temps CONCAT et SPLIT

N°	Description	Forme graphique	Exemple
Concaténation des types de données de temps			
1a	CONCAT_DATE_TOD	<pre> +-----+ CONCAT_DATE_TOD DATE -- DATE --DT TOD -- TOD +-----+</pre>	Concaténation d'une date: <pre> VAR myD: DATE; END_VAR myD:= CONCAT_DATE_TOD (D#2010-03-12, TOD#12:30:00);</pre>
1b	CONCAT_DATE_LTOD	<pre> +-----+ CONCAT_DATE_LTOD DATE -- DATE --LDT LTOD -- LTOD +-----+</pre>	Concaténation d'une date et heure: <pre> VAR myD: DATE; END_VAR myD:= CONCAT_DATE_LTOD (D#2010-03-12, TOD#12:30:12.1223452);</pre>
2	CONCAT_DATE	<pre> +-----+ CONCAT_DATE ANY_INT -- YEAR --DATE ANY_INT -- MONTH ANY_INT -- DAY +-----+</pre>	Concaténation d'une date et heure: <pre> VAR myD: DATE; END_VAR myD:= CONCAT_DATE (2010,3,12);</pre>
3a	CONCAT_TOD	<pre> +-----+ CONCAT_TOD ANY_INT -- HOUR --TOD ANY_INT -- MINUTE ANY_INT -- SECOND ANY_INT -- MILLISECOND +-----+</pre>	Concaténation d'une heure: <pre> VAR myTOD: TOD; END_VAR myTD:= CONCAT_TOD (16,33,12,0);</pre>
3b	CONCAT_LTOD	<pre> +-----+ CONCAT_LTOD ANY_INT -- HOUR --LTOD ANY_INT -- MINUTE ANY_INT -- SECOND ANY_INT -- MILLISECOND +-----+</pre>	Concaténation d'une heure: <pre> VAR myTOD: LTOD; END_VAR myTD:= CONCAT_TOD (16,33,12,0);</pre>

N°	Description	Forme graphique	Exemple
4a	CONCAT_DT	<pre> +-----+ CONCAT_DT +-----+ ANY_INT -- YEAR --DT ANY_INT -- MONTH ANY_INT -- DAY ANY_INT -- HOUR ANY_INT -- MINUTE ANY_INT -- SECOND ANY_INT -- MILLISECOND +-----+ </pre>	Concaténation d'une heure: <pre> VAR myDT: DT; Day: USINT; END_VAR Day := 17; myDT:= CONCAT_DT (2010,3,Day,12,33,12,0); </pre>
4b	CONCAT_LDT	<pre> +-----+ CONCAT_LDT +-----+ ANY_INT -- YEAR --LDT ANY_INT -- MONTH ANY_INT -- DAY ANY_INT -- HOUR ANY_INT -- MINUTE ANY_INT -- SECOND ANY_INT -- MILLISECOND +-----+ </pre>	Concaténation d'une heure: <pre> VAR myDT: LDT; Day: USINT; END_VAR Day := 17; myDT:= CONCAT_LDT (2010,3,Day,12,33,12,0); </pre>
Séparation des types de données de temps			
5	SPLIT_DATE	<pre> +-----+ SPLIT_DATE +-----+ DATE-- IN YEAR -- ANY_INT MONTH -- ANY_INT DAY -- ANY_INT +-----+ </pre> Voir NOTE 2	Séparation d'une date: <pre> VAR myD: DATE:= DATE#2010-03-10; myYear: UINT; myMonth, myDay: USINT; END_VAR SPLIT_DATE (myD,myYear,myMonth,myDay); </pre>
6a	SPLIT_TOD	<pre> +-----+ SPLIT_TOD +-----+ TOD-- IN HOUR -- ANY_INT MINUTE -- ANY_INT SECOND -- ANY_INT MILLISECOND -- ANY_INT +-----+ </pre> Voir NOTE 2	Séparation d'une heure: <pre> VAR myTOD: TOD:= TOD#14:12:03; myHour, myMin, mySec: USINT; myMilliSec: UINT; END_VAR SPLIT_TOD(myTOD, myHour, myMin, mySec,myMilliSec); </pre>
6b	SPLIT_LTOD	<pre> +-----+ SPLIT_LTOD +-----+ LTOD-- IN HOUR -- ANY_INT MINUTE -- ANY_INT SECOND -- ANY_INT MILLISECOND -- ANY_INT +-----+ </pre> Voir NOTE 2	Séparation d'une heure: <pre> VAR myTOD: LTOD:= TOD#14:12:03; myHour, myMin, mySec: USINT; myMilliSec: UINT; END_VAR SPLIT_TOD(myTOD, myHour, myMin, mySec,myMilliSec); </pre>
7a	SPLIT_DT	<pre> +-----+ 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 +-----+ </pre> Voir NOTE 2	Séparation d'une date: <pre> 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); </pre>

N°	Description	Forme graphique	Exemple
7b	SPLIT_LDT	<pre> +-----+ 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 +-----+</pre> Voir NOTE 2	Séparation d'une date: <pre> 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);</pre>
Extraction du jour de la semaine			
8	DAY_OF_WEEK	<pre> +-----+ DAY_OF_WEEK DATE-- IN -- ANY_INT +-----+</pre> Voir NOTE 2	Extraction du jour de la semaine: <pre> VAR myD: DATE:= DATE#2010-03-10; myDoW: USINT; END_VAR myDoW:= DAY_OF_WEEK(myD);</pre>
La fonction DAY_OF_WEEK retourne 0 pour dimanche, 1 pour lundi, ..., et 6 pour samedi.			
NOTE 1 Le type de données d'entrée YEAR est un type d'au moins 16 bits pour pouvoir prendre en charge une valeur d'année valide.			
NOTE 2 L'Intégrateur spécifie les types de données fournis pour les sorties ANY_INT.			
NOTE 3 L'Intégrateur peut définir des entrées ou sorties additionnelles en fonction de la précision prise en charge, par exemple, des microsecondes et des nanosecondes.			

6.6.2.5.13 Fonctions de conversion de boutisme ("endianess")

Les fonctions de conversion de boutisme convertissent vers et depuis le boutisme utilisé en interne spécifique de l'Intégrateur de l'automate programmable depuis et vers le boutisme demandé.

Le boutisme est l'organisation des octets dans une variable ou un type de données longs.

Les valeurs des données au format grand-boutiste ("big endian") sont placées dans les emplacements de mémoire en commençant par l'octet le plus à gauche pour terminer par l'octet le plus à droite.

Les valeurs des données au format petit-boutiste ("little endian") sont placées dans les emplacements de mémoire en commençant par l'octet le plus à droite pour terminer par l'octet le plus à gauche.

Indépendamment du boutisme, le décalage de bit 0 adresse le bit le plus à droite d'un type de données.

L'utilisation de l'accès partiel avec le nombre inférieur retourne la partie de valeur plus faible indépendamment du boutisme spécifié.

EXEMPLE 1 Boutisme

```
TYPE D: DWORD:= 16#1234_5678; END_TYPE;
```

Configuration mémoire

```

pour un grand-boutiste: 16#12, 16#34, 16#56, 16#78
pour un petit-boutiste: 16#78, 16#56, 16#34, 16#12.

```

EXEMPLE 2 Boutisme

```
TYPE L: ULINT:= 16#1234_5678_9ABC_DEF0; END_TYPE;
```

Configuration mémoire

pour un grand-boutiste: 16#12, 16#34, 16#56, 16#78, 16#9A, 16#BC, 16#DE, 16#F0
pour un petit-boutiste: 16#F0, 16#DE, 16#BC, 16#9A, 16#78, 16#56, 16#34, 16#12.

Les types de données suivants doivent être pris en charge en tant qu'entrées ou sorties des fonctions de conversion de boutisme:

- ANY_INT avec une taille supérieure ou égale à 16 bits
- ANY_BIT avec une taille supérieure ou égale à 16 bits
- ANY_REAL
- WCHAR
- TIME
- tableaux de ces types de données
- structures contenant des composants de ces types de données

Les autres types de données ne sont pas convertis mais peuvent être contenus dans les structures à convertir.

Le Tableau 37 décrit les fonctions de conversion de boutisme.

Tableau 37 – Fonctions de conversion de boutisme

N°	Description	Forme graphique	Forme textuelle
1	TO_BIG_ENDIAN	<pre> +-----+ TO_BIG_ENDIAN ANY -- IN --ANY +-----+</pre>	Conversion au format de données grand-boutiste A:= TO_BIG_ENDIAN(B);
2	TO_LITTLE_ENDIAN	<pre> +-----+ TO_LITTLE_ENDIAN ANY -- IN --ANY +-----+</pre>	Conversion au format de données petit-boutiste B:= TO_LITTLE_ENDIAN(A);
3	BIG_ENDIAN_TO	<pre> +-----+ FROM_BIG_ENDIAN ANY -- IN --ANY +-----+</pre>	Conversion depuis le format de données grand-boutiste A:= FROM_BIG_ENDIAN(B);
4	LITTLE_ENDIAN_TO	<pre> +-----+ FROM_LITTLE_ENDIAN ANY -- IN --ANY +-----+</pre>	Conversion depuis le format de données petit-boutiste A:= FROM_LITTLE_ENDIAN(B);
Les types de données sur les côtés entrée et sortie doivent être identiques. NOTE Dans le cas où la variable est déjà au format de données demandé, la fonction ne modifie pas la représentation des données.			

6.6.2.5.14 Fonctions des types de données énumérés

Les fonctions de sélection et de comparaison répertoriées dans le Tableau 38 peuvent être appliquées aux entrées qui sont d'un type de données énuméré.

Tableau 38 – Fonctions des types de données énumérés

N°	Description/ Nom de fonction	Symbole	Caractéristique n° x du Tableau y
1	SEL		Caractéristique 2, Tableau 32
2	MUX		Caractéristique 6, Tableau 32
3 ^a	EQ	=	Caractéristique 3, Tableau 33
4 ^a	NE	<>	Caractéristique 6, Tableau 33
NOTE Les dispositions des Notes 1 et 2 du Tableau 33 s'appliquent à ce tableau.			
^a Les dispositions des notes de bas de page a et b du Tableau 33 s'appliquent à cette caractéristique.			

6.6.2.5.15 Fonctions de validation

Les fonctions de validation vérifient si le paramètre d'entrée spécifié contient une valeur valide.

La fonction en surcharge `IS_VALID` est définie pour les types de données `REAL` et `LREAL`. Dans le cas où le nombre réel est un non-numérique (NaN, Not-a-Number) ou l'infini (+Inf, -Inf), le résultat de la fonction de validation est `FALSE`.

L'Intégrateur peut prendre en charge des types de données additionnels avec la fonction de validation `IS_VALID`. Le résultat de ces extensions est spécifique de l'Intégrateur.

La fonction en surcharge `IS_VALID_BCD` est définie pour les types de données `BYTE`, `WORD`, `DWORD` et `LWORD`. Dans le cas où la valeur n'est pas conforme à la définition de BCD, le résultat de la fonction de validation est `FALSE`.

Le Tableau 39 présente la liste des caractéristiques des fonctions de validation.

Tableau 39 – Fonctions de validation

N°	Fonction	Forme graphique	Exemple
1	<code>IS_VALID</code>	<pre> +-----+ . IS_VALID ANY_REAL-- IN --BOOL +-----+</pre>	Validité d'un <code>REAL</code> VAR R: REAL; END_VAR IF IS_VALID(R) THEN ...
2	<code>IS_VALID_BCD</code>	<pre> +-----+ IS_VALID_BCD -ANY_BIT-- IN --BOOL +-----+</pre>	Essai de validité pour un mot BCD VAR W: WORD; END_VAR IF IS_VALID_BCD(W) THEN ...

6.6.3 Blocs fonctionnels

6.6.3.1 Généralités

Un bloc fonctionnel est une unité d'organisation de programme (POU) qui représente, pour la modularité et la structuration, une partie bien définie du programme.

Le concept de bloc fonctionnel comprend le type de bloc fonctionnel et l'instance de bloc fonctionnel.

- Le type de bloc fonctionnel est constitué:

- de la définition d'une structure de données divisée en variables d'entrée, de sortie et internes; et
 - d'un ensemble d'opérations à effectuer sur les éléments de la structure de données lorsqu'une instance du type de bloc fonctionnel est appelée.
 - Instance de bloc fonctionnel
 - Il s'agit d'une utilisation nommée multiple (instances) d'un type de bloc fonctionnel.
 - Chaque instance doit avoir un identificateur associé (le nom d'instance), et une structure de données contenant les variables statiques d'entrée, de sortie et internes.
 Les variables statiques doivent conserver leur valeur d'une exécution de l'instance de bloc fonctionnel à la suivante. Par conséquent, l'appel d'une instance de bloc fonctionnel avec les mêmes paramètres d'entrée peut ne pas toujours produire les mêmes valeurs de sortie.
- Les caractéristiques communes des POU s'appliquent aux blocs fonctionnels.
- Bloc fonctionnel orienté objet
 Le bloc fonctionnel peut être étendu par un ensemble de caractéristiques orientées objet.
 Le bloc fonctionnel orienté objet est également un super-ensemble de la classe.

6.6.3.2 Déclaration du type de bloc fonctionnel

Le type de bloc fonctionnel doit être déclaré de manière similaire à ce qui est décrit pour les fonctions.

Les caractéristiques de la déclaration du type de bloc fonctionnel sont définies dans le Tableau 40:

- 1) Le mot-clé `FUNCTION_BLOCK` suivi d'un identificateur spécifiant le nom du bloc fonctionnel déclaré.
- 2) Un ensemble d'opérations constituant le corps.
- 3) Le mot-clé de terminaison `END_FUNCTION_BLOCK` après le corps du bloc fonctionnel.
- 4) La construction avec `VAR_INPUT`, `VAR_OUTPUT` et `VAR_IN_OUT`, si nécessaire, spécifiant les noms et les types des variables.
- 5) Les valeurs des variables qui sont déclarées par l'intermédiaire d'une construction `VAR_EXTERNAL` peuvent être modifiées depuis l'intérieur du bloc fonctionnel.
- 6) Les valeurs des constantes qui sont déclarées par l'intermédiaire d'une construction `VAR_EXTERNAL CONSTANT` ne peuvent pas être modifiées depuis l'intérieur du bloc fonctionnel.
- 7) Les tableaux de longueur variable peuvent être utilisés en tant que `VAR_IN_OUT`.
- 8) Les variables d'entrée, de sortie et statiques peuvent être initialisées.
- 9) Les entrées et sorties `EN/ENO` doivent être déclarées de manière similaire aux variables d'entrée et de sortie.

Les caractéristiques suivantes sont spécifiques aux blocs fonctionnels (différents des fonctions):

- 10) Une construction `VAR...END_VAR` et `VAR_TEMP...END_VAR`, si nécessaire, spécifiant les noms et les types des variables internes du bloc fonctionnel.
 Contrairement aux fonctions, les variables déclarées dans la section `VAR` sont statiques.
- 11) Les variables de la section `VAR` (statique) peuvent être déclarées `PUBLIC` ou `PRIVATE`.
 Le spécificateur d'accès `PRIVATE` est défini par défaut. On peut accéder à une variable publique depuis l'extérieur du bloc fonctionnel en utilisant la syntaxe comme pour l'accès aux sorties de bloc fonctionnel.
- 12) Le qualificateur `RETAIN` ou `NON_RETAIN` peut être utilisé pour les variables d'entrée, de sortie et internes d'un bloc fonctionnel, comme décrit dans le Tableau 40.

- 13) Dans les déclarations textuelles, les qualificateurs `R_EDGE` et `F_EDGE` doivent être utilisés pour indiquer une fonction de détection de front sur des entrées booléennes. Cela doit provoquer la déclaration implicite d'un bloc fonctionnel de type `R_TRIG` ou `F_TRIG`, respectivement, dans ce bloc fonctionnel pour effectuer la détection de front requise. Pour un exemple de cette construction, voir le Tableau 40.
- 14) Dans les déclarations graphiques, la construction de détection des fronts descendants et montants représentée doit être utilisée. Lorsque le jeu de caractères est utilisé, le caractère "supérieur à" ">" ou "inférieur à" "<" doit correspondre au front du bloc fonctionnel.
- 15) La notation avec astérisque "*" telle que définie dans le Tableau 16 peut être utilisée dans la déclaration des variables internes d'un bloc fonctionnel.
- 16) Si les types de données génériques sont utilisés dans la déclaration de type des entrées et sorties de bloc fonctionnel normalisé, les règles de déduction des types réels des sorties de ces types de bloc fonctionnel doivent faire partie de la définition du type du bloc fonctionnel.
- 17) Des instances d'autres blocs fonctionnels, classes et blocs fonctionnels orientés objet peuvent être déclarées dans toutes les sections de variables, sauf la section `VAR_TEMP`.
- 18) Il convient qu'une instance de bloc fonctionnel déclarée à l'intérieur d'un type de bloc fonctionnel n'utilise pas le même nom qu'une fonction présentant la même portée pour les noms, afin d'éviter toute ambiguïté.

Tableau 40 – Déclaration du type de bloc fonctionnel

N°	Description	Exemple
1	Déclaration du type de bloc fonctionnel <code>FUNCTION_BLOCK ...</code> <code>END_FUNCTION_BLOCK</code>	<code>FUNCTION_BLOCK myFB ... END_FUNCTION_BLOCK</code>
2a	Déclaration des entrées <code>VAR_INPUT ... END_VAR</code>	<code>VAR_INPUT IN: BOOL; T1: TIME; END_VAR</code>
2b	Déclaration des sorties <code>VAR_OUTPUT ... END_VAR</code>	<code>VAR_OUTPUT OUT: BOOL; ET_OFF: TIME; END_VAR</code>
2c	Déclaration des entrées-sorties <code>VAR_IN_OUT ... END_VAR</code>	<code>VAR_IN_OUT A: INT; END_VAR</code>
2d	Déclaration des variables temporaires <code>VAR_TEMP ... END_VAR</code>	<code>VAR_TEMP I: INT; END_VAR</code>
2e	Déclaration des variables statiques <code>VAR ... END_VAR</code>	<code>VAR B: REAL; END_VAR</code>
2f	Déclaration des variables externes <code>VAR_EXTERNAL ... END_VAR</code>	<code>VAR_EXTERNAL B: REAL; END_VAR</code> Correspondant à <code>VAR_GLOBAL B: REAL</code>
2g	Déclaration des variables externes <code>VAR_EXTERNAL CONSTANT ... END_VAR</code>	<code>VAR_EXTERNAL CONSTANT B: REAL; END_VAR</code> Correspondant à <code>VAR_GLOBAL B: REAL</code>
3a	Initialisation des entrées	<code>VAR_INPUT MN: INT:= 0;</code>
3b	Initialisation des sorties	<code>VAR_OUTPUT RES: INT:= 1;</code>
3c	Initialisation des variables statiques	<code>VAR B: REAL:= 12.1;</code>
3d	Initialisation des variables temporaires	<code>VAR_TEMP I: INT:= 1;</code>
-	Entrées et sorties EN/ENO	Défini dans le Tableau 18
4a	Déclaration du qualificateur <code>RETAIN</code> pour les variables d'entrée	<code>VAR_INPUT RETAIN X: REAL; END_VAR</code>
4b	Déclaration du qualificateur <code>RETAIN</code> pour les variables de sortie	<code>VAR_OUTPUT RETAIN X: REAL; END_VAR</code>
4c	Déclaration du qualificateur <code>NON_RETAIN</code> pour les variables d'entrée	<code>VAR_INPUT NON_RETAIN X: REAL; END_VAR</code>

N°	Description	Exemple
4d	Déclaration du qualificateur NON_RETAIN pour les variables de sortie	VAR_OUTPUT NON_RETAIN X: REAL; END_VAR
4e	Déclaration du qualificateur RETAIN pour les variables statiques	VAR RETAIN X: REAL; END_VAR
4f	Déclaration du qualificateur NON_RETAIN pour les variables statiques	VAR NON_RETAIN X: REAL; END_VAR
5a	Déclaration du qualificateur RETAIN pour les instances de bloc fonctionnel locales	VAR RETAIN TMR1: TON; END_VAR
5b	Déclaration du qualificateur NON_RETAIN pour les instances de bloc fonctionnel locales	VAR NON_RETAIN TMR1: TON; END_VAR
6a	Déclaration textuelle - d'entrées avec front montant	<pre> 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; (* Exemple de langage ST *) END_FUNCTION_BLOCK </pre>
6b	- d'entrées avec front descendant (textuel)	Voir ci-dessus
7a	Déclaration graphique - d'entrées avec front montant (>)	<pre> FUNCTION_BLOCK (* Interface externe *) +-----+ AND_EDGE +-----+ (* Corps de bloc fonctionnel *) +-----+ +-----+ X-- & --Z Y-- +-----+ END_FUNCTION_BLOCK </pre>
7b	Déclaration graphique - d'entrées avec front descendant (<)	Voir ci-dessus
NOTE Les caractéristiques 1 à 3 du présent tableau sont équivalentes à des fonctions; voir le Tableau 19.		

Des exemples de déclarations du type de bloc fonctionnel sont décrits ci-dessous.

EXEMPLE 1 Déclaration du type de bloc fonctionnel

FUNCTION_BLOCK DEBOUNCE

(*** Interface externe ***)

```
VAR_INPUT
  IN: BOOL;                (* Défaut = 0 *)
  DB_TIME: TIME:= t#10ms;  (* Défaut = t#10ms *)
END_VAR
```

```
VAR_OUTPUT
  OUT: BOOL;                (* Défaut = 0 *)
  ET_OFF: TIME;             (* Défaut = t#0s *)
END_VAR
```

```
VAR DB_ON: TON;              (** Variables internes **)
  DB_OFF: TON;              (** et instances de FB **)
  DB_FF: SR;
END_VAR
```

```
(*** Corps de bloc fonctionnel ***)
  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) Déclaration textuelle (langage ST)

FUNCTION_BLOCK

(* Interface de paramètres externe *)

```
+-----+
| DEBOUNCE |
+-----+
| IN | OUT |
+-----+
| DB_TIME | ET_OFF |
+-----+
```

(* Corps de type de bloc fonctionnel *)

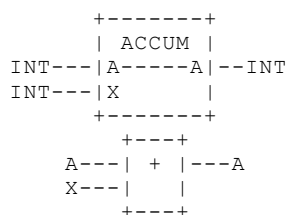
```
+-----+ +-----+
| DB_ON | | DB_FF |
+-----+ +-----+
| TON | | SR |
+-----+ +-----+
| IN | Q | | S1 | Q |
+-----+ +-----+
| PT | ET | | R |
+-----+ +-----+
| DB_OFF |
+-----+
| TON |
+-----+
| IN | Q |
+-----+
| DB_TIME | PT ET | ET_OFF |
+-----+
```

END_FUNCTION_BLOCK

b) Déclaration graphique (langage FBD)

L'exemple ci-dessous présente la déclaration et l'utilisation graphique des variables d'entrée-sortie dans les blocs fonctionnels, comme décrit dans le Tableau 40.

EXEMPLE 2

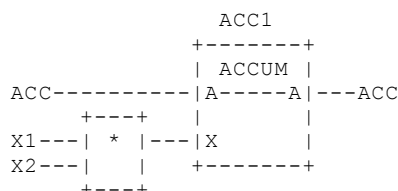


```

FUNCTION_BLOCK ACCUM
VAR_IN_OUT A: INT; END_VAR
VAR_INPUT X: INT; END_VAR
A:= A+X;
END_FUNCTION_BLOCK

```

a) Déclaration graphique et textuelle de type de bloc fonctionnel et de fonction



```

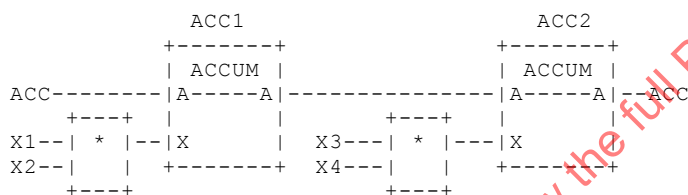
VAR
  ACC: INT;
  X1: INT;
  X2: INT;
END_VAR

```

Cette déclaration est supposée être l'effet de l'exécution:

```
ACC:= ACC+X1*X2;
```

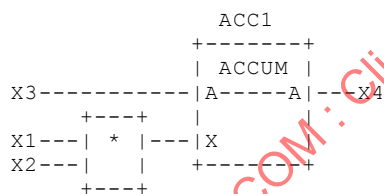
b) Utilisation autorisée de l'instance de bloc fonctionnel et de la fonction



Les déclarations décrites en b) sont supposées pour
ACC, X1, X2, X3, et X4;

l'effet de l'exécution est
ACC:= ACC+X1*X2+X3*X4;

c) Utilisation autorisée de l'instance de bloc fonctionnel



```

VAR
  X1: INT;
  X2: INT;
  X3: INT;
  X4: INT;
END_VAR

```

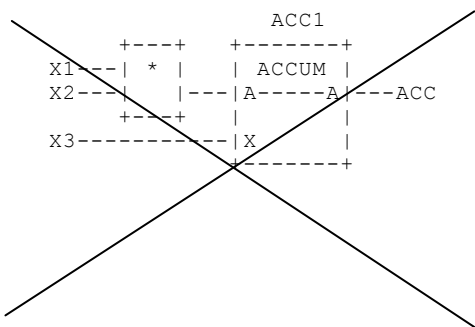
La déclaration est supposée être l'effet de l'exécution:

```

X3:= X3+X1*X2;
X4:= X3;

```

d) Utilisation autorisée de l'instance de bloc fonctionnel et de la fonction – avec affectation d'une sortie



NON AUTORISÉ!

La connexion à la variable d'entrée-sortie A n'est pas une variable ou un nom de bloc fonctionnel (voir le texte précédent).

e) Utilisation non autorisée de l'instance de bloc fonctionnel

L'exemple ci-dessous décrit le bloc fonctionnel AND_EDGE utilisé dans le Tableau 40.

EXEMPLE 3 Déclaration du type de bloc fonctionnel AND_EDGE

La déclaration du bloc fonctionnel AND_EDGE dans les exemples du Tableau 40 ci-dessus est équivalente à:

```
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
```

Voir le Tableau 44 pour obtenir la définition des blocs fonctionnels de détection de front R_TRIG et F_TRIG.

6.6.3.3 Déclaration d'instance de bloc fonctionnel

L'instance de bloc fonctionnel doit être déclarée de manière similaire à ce qui est décrit pour les variables structurées.

Lorsqu'une instance de bloc fonctionnel est déclarée, les valeurs initiales des entrées, des sorties ou des variables publiques de cette instance peuvent être déclarées dans une liste entre parenthèses à la suite de l'opérateur d'affectation suivant l'identificateur de type de bloc fonctionnel comme décrit dans le Tableau 41.

Les valeurs initiales par défaut des éléments dont les valeurs initiales ne sont pas répertoriées dans la liste d'initialisation décrite ci-dessus doivent être déclarées dans la déclaration du type de bloc fonctionnel.

Tableau 41 – Déclaration d'instance de bloc fonctionnel

N°	Description	Exemple
1	Déclaration d'une ou plusieurs instances de bloc fonctionnel	<pre>VAR FB_instance_1, FB_instance_2: my_FB_Type; T1, T2, T3: TON; END_VAR</pre>
2	Déclaration d'une instance de bloc fonctionnel avec initialisation de ses variables	<pre>VAR TempLoop: PID:= (PropBand:= 2.5, Integral:= T#5s); END_VAR</pre> Attribue des valeurs initiales aux entrées et sorties d'une instance de bloc fonctionnel.

6.6.3.4 Appel de bloc fonctionnel

6.6.3.4.1 Généralités

L'appel d'une instance d'un bloc fonctionnel peut être représenté sous forme textuelle ou graphique.

Les caractéristiques de l'appel de bloc fonctionnel (comprenant l'appel formel et l'appel informel) sont similaires à celles des fonctions avec les extensions suivantes:

- 1) L'appel textuel d'un bloc fonctionnel doit être constitué du nom d'instance du bloc fonctionnel suivi d'une liste de paramètres.

- 2) Dans la représentation graphique, le nom d'instance du bloc fonctionnel doit être situé au-dessus du bloc.
- 3) Les variables d'entrée et les variables de sortie d'une instance d'un bloc fonctionnel sont stockées et peuvent être représentées comme des éléments de types de données structurés. Par conséquent, l'affectation des entrées et l'accès aux sorties d'un bloc fonctionnel peuvent être
 - a) immédiats dans l'appel du bloc fonctionnel, ce qui constitue l'utilisation type; ou
 - b) séparés de l'appel. Ces affectations séparées doivent prendre effet lors de l'appel suivant du bloc fonctionnel.
 - c) Les entrées non affectées ou non reliées d'un bloc fonctionnel doivent conserver leurs valeurs initialisées ou les valeurs du dernier appel, le cas échéant.

Il est possible qu'aucun paramètre réel ne soit spécifié pour une variable d'entrée-sortie ou une instance de bloc fonctionnel utilisée en tant que variable d'entrée d'une autre instance de bloc fonctionnel. Cependant, l'instance doit être pourvue d'une valeur valide qui est stockée, par exemple, par l'initialisation ou l'appel précédent, avant d'être utilisée dans le bloc fonctionnel (corps) ou par une méthode. Sinon, il se produit une erreur d'exécution.

Des règles supplémentaires s'appliquent à l'appel de bloc fonctionnel:

- 4) Si une instance de bloc fonctionnel est appelée avec `EN=0`, l'Intégrateur doit spécifier si les variables d'entrée et d'entrée-sortie sont définies dans l'instance.
- 5) Le nom d'une instance de bloc fonctionnel peut être utilisé comme entrée d'une instance de bloc fonctionnel si elle est déclarée en tant que variable d'entrée dans une déclaration `VAR_INPUT` ou en tant que variable d'entrée-sortie d'une instance de bloc fonctionnel dans une déclaration `VAR_IN_OUT`.
- 6) On peut accéder aux valeurs de sortie d'une instance de bloc fonctionnel différente dont le nom est passé dans le bloc fonctionnel par l'intermédiaire d'une construction `VAR_INPUT`, `VAR_IN_OUT` ou `VAR_EXTERNAL`, depuis l'intérieur du bloc fonctionnel, mais on ne peut pas les modifier.
- 7) Un bloc fonctionnel dont le nom d'instance est passé dans le bloc fonctionnel par l'intermédiaire d'une construction `VAR_IN_OUT` ou `VAR_EXTERNAL` peut être appelé depuis l'intérieur du bloc fonctionnel.
- 8) Seuls les variables ou les noms d'instance de bloc fonctionnel peuvent être passés dans un bloc fonctionnel par l'intermédiaire de la construction `VAR_IN_OUT`.

Cette règle a pour but d'éviter toute modification fortuite de ces sorties. Cependant, la définition en "cascade" de constructions `VAR_IN_OUT` est permise.

Le Tableau 42 suivant contient les caractéristiques de l'appel de bloc fonctionnel.

Tableau 42 – Appel de bloc fonctionnel

N°	Description	Exemple
1	Appel formel complet (textuel uniquement) Est utilisé si EN/ENO est nécessaire dans les appels.	<pre> YourCTU(EN:= not B, CU:= r, PV:= c1, ENO=> next, Q => out, CV => c2); </pre>
2	Appel formel incomplet (textuel uniquement)	<pre> YourCTU(Q => out, CV => c2); </pre> La variable EN, CU, PV aura la valeur du dernier appel ou une valeur initiale, si elle n'a jamais été appelée auparavant.
3	Appel graphique	<pre> YourCTU +-----+ CTU B -- EN ENO -- next r -- CU Q -- out c1 -- PV CV -- c2 +-----+ </pre>
4	Appel graphique avec entrée et sortie booléennes inversées	<pre> YourCTU +-----+ CTU B -0 EN ENO -- next r -- CU Q 0- out c1 -- PV CV -- c2 +-----+ </pre> L'utilisation de ces constructions est interdite pour les variables d'entrée-sortie.
5a	Appel graphique avec utilisation de VAR_IN_OUT	
5b	Appel graphique avec affectation de VAR_IN_OUT à une variable	
6a	Appel textuel avec affectation d'entrée séparée FB_Instance.Input:= x;	<pre> YourTon.IN:= r; YourTon.PT:= t; YourTon(not Q => out); </pre>
6b	Appel graphique avec affectation d'entrée séparée	<pre> +-----+ r-- MOVE --YourCTU.CU +-----+ +-----+ c-- MOVE --YourCTU.PV +-----+ YourCTU +-----+ CTU 1-- EN ENO -- next -- CU Q 0- out -- PV CV -- +-----+ </pre>
7	Sortie textuelle lue après l'appel de bloc fonctionnel x:= FB_Instance.Output;	
8a	Sortie textuelle affectée dans l'appel de bloc fonctionnel	
8b	Sortie textuelle affectée dans l'appel de bloc fonctionnel avec inversion	
9a	Appel textuel avec un nom d'instance de bloc fonctionnel en entrée	<pre> VAR_INPUT I_TMR: TON; END_VAR EXPIRED:= I_TMR.Q; </pre> Il est supposé que les variables EXPIRED et A_VAR ont été déclarées de type BOOL dans cet exemple et dans les suivants.

9b	Appel graphique avec un nom d'instance de bloc fonctionnel en entrée	a
10a	Appel textuel avec un nom d'instance de bloc fonctionnel comme VAR_IN_OUT	VAR_IN_OUT IO_TMR: TOF; END_VAR IO_TMR (IN:=A_VAR, PT:= T#10S); EXPIRED:= IO_TMR.Q;
10b	Appel graphique avec un nom d'instance de bloc fonctionnel comme VAR_IN_OUT	
11a	Appel textuel avec un nom d'instance de bloc fonctionnel comme variable externe	VAR_EXTERNAL EX_TMR: TOF; END_VAR EX_TMR (IN:= A_VAR, PT:= T#10S); EXPIRED:= EX_TMR.Q;
11b	Appel graphique avec un nom d'instance de bloc fonctionnel comme variable externe	

IECNORM.COM : Click to view the full PDF of IEC 61131-3:2013

EXEMPLE Appel de bloc fonctionnel avec affectation de paramètre immédiate et séparée

```

YourCTU
+-----+
| CTU |
B -0|EN ENO|--
r --|CU Q|0-out
c --|PV CV|--
+-----+

YourCTU (EN:= not b,
CU:= r,
PV:= c,
not Q => out);

```

a) Appel de bloc fonctionnel avec affectation d'entrées immédiate (utilisation type)

```

+-----+
r--| MOVE |--YourCTU.CU
+-----+
+-----+
c--| MOVE |--YourCTU.PV
+-----+

YourCTU.CU:= r;
YourCTU.PV:= V;

YourCTU(not Q => out);

```

```

YourCTU
+-----+
| CTU |
--|EN ENO|--
--|CU Q|0-out
--|PV CV|--
+-----+

```

b) Appel de bloc fonctionnel avec affectation d'entrée séparée

```

YourCTU
+-----+
| CTU |
a--| NE |---0|EN ENO|--
b--| | r--|CU Q|0-out
+-----+
+-----+

VAR a, b, r, out: BOOL;
YourCTU: CTU; END_VAR

YourCTU (EN := NOT (a <> b),
CU := r,
NOT Q => out);

```

c) Appel de bloc fonctionnel avec accès immédiat à la sortie (utilisation type)

De plus, une inversion dans l'appel est permise

```

FF75
+-----+
| SR |
bIn1---|S1 Q1|---bOut3
bIn2---|R |
+-----+

VAR FF75: SR; END_VAR (* Déclaration *)
FF75(S1:= bIn1, (* appel *)
R:= bIn2);

bOut3:= FF75.Q1; (* Affectation de sortie *)

```

d) Appel de bloc fonctionnel avec affectation de sortie séparée textuelle (après appel)

```

TONs[12]
+-----+
| TON |
bIn1 --|IN Q|--
T#10ms --|PT ET|--
+-----+

TONs[i]
+-----+
| TON |
bIn1 --|IN Q|--
T#20ms --|PT ET|--
+-----+

VAR
TONs: array [0..100] OF TON;
i: INT;
END_VAR

TON[12](IN:= bIn1, PT:= T#10ms);

TON[i](IN:= bIn1, PT:= T#20ms);

```

e) Appel de bloc fonctionnel à l'aide d'un tableau d'instances

```

myCooler.Cooling
+-----+
| TOF |
bIn1 --|IN Q|--
T#30s --|PT ET|--
+-----+

TYPE
Cooler: STRUCT
Temp: INT;
Cooling: TOF;
END_STRUCT;
END_TYPE

VAR
myCooler: Cooler;
END_VAR

myCooler.Cooling(IN:= bIn1, PT:= T#30s);

```

f) Appel de bloc fonctionnel en utilisant une instance comme élément de structure

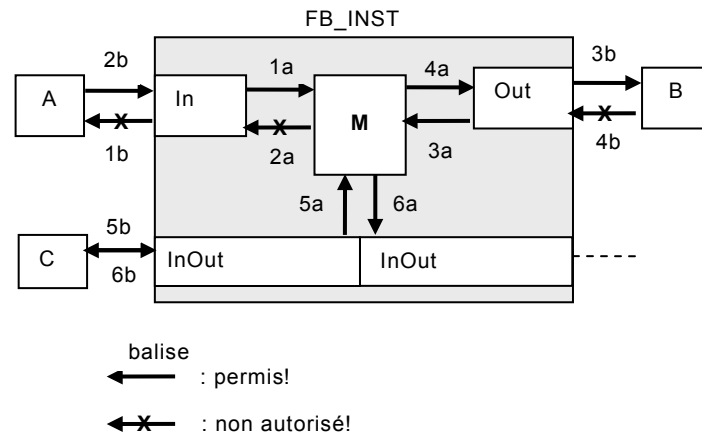
6.6.3.4.2 Utilisation des paramètres d'entrée et de sortie

La Figure 13 et la Figure 14 résument les règles d'utilisation des paramètres d'entrée et de sortie d'un bloc fonctionnel dans le contexte de l'appel de ce bloc fonctionnel. Cette affectation aux paramètres d'entrée et d'entrée-sortie doit prendre effet lors de l'appel suivant du bloc fonctionnel.

<pre> 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 </pre>		
Utilisation	a) A l'intérieur du bloc fonctionnel	b) A l'extérieur du bloc fonctionnel
1 Lecture d'une entrée	M := In;	A := In; Non autorisé (NOTES 1 et 2)
2 Affectation d'une entrée	In := M; Non autorisé (NOTE 1)	// Appel avec affectation de paramètre immédiate FB_INST(In := A); // Affectation séparée (NOTE 4) FB_INST.In := A;
3 Lecture d'une sortie	M := Out;	// Appel avec affectation de paramètre immédiate FB_INST(Out => B); // Affectation séparée B := FB_INST.Out;
4 Affectation d'une sortie	Out := M;	FB_INST.Out := B; Non autorisé (NOTE 1)
5 Lecture d'une entrée-sortie	M := In_out;	FB_INST(In_out => C); Non autorisé C := FB_INST.In_out; Non autorisé
6 Affectation d'une entrée-sortie	In_out := M; (NOTE 3)	// Appel avec affectation de paramètre immédiate FB_INST(In_out := C); FB_INST.In_out := C; Non autorisé
NOTE 1 Les utilisations répertoriées comme étant "non autorisées" dans le présent tableau pourraient conduire à des effets secondaires imprévisibles spécifiques de l'Intégrateur. NOTE 2 La lecture et l'écriture (affectation) des paramètres d'entrée et de sortie, et des variables internes d'un bloc fonctionnel peuvent être effectuées par la "fonction de communication", la "fonction d'interface opérateur" ou les "fonctions de programmation, essai et surveillance" définies dans la CEI 61131-1. NOTE 3 Une modification dans le bloc fonctionnel d'une variable déclarée dans un bloc VAR_IN_OUT est permise.		

Figure 13 – Utilisation des paramètres d'entrée et de sortie de bloc fonctionnel (règles)

L'utilisation des paramètres d'entrée et de sortie définie par les règles de la Figure 13 est décrite à la Figure 14.



Les balises 1a, 1b, etc. correspondent aux règles de la Figure 13.

Figure 14 – Utilisation des paramètres d'entrée et de sortie de bloc fonctionnel (illustration des règles)

Les exemples ci-dessous décrivent l'utilisation graphique des noms de bloc fonctionnel en tant que paramètres et variables externes.

EXEMPLES Utilisation graphique des noms de bloc fonctionnel en tant que paramètres et variables externes

```

FUNCTION_BLOCK

(* Interface externe *)

+-----+
|  INSIDE_A  |
TON---| I_TMR  EXPIRED |---BOOL
+-----+

(* Corps de bloc fonctionnel *)

+-----+
|  MOVE  |
I_TMR.Q---|  |---EXPIRED
+-----+

END_FUNCTION_BLOCK

FUNCTION_BLOCK

(* Interface externe *)

+-----+
| EXAMPLE_A |
BOOL---| GO  DONE |---BOOL
+-----+

(* Corps de bloc fonctionnel *)

E_TMR
+-----+
|  TON  |
GO---| IN  Q |
t#100ms---| PT ET |
+-----+

I_BLK
+-----+
|  INSIDE_A  |
E_TMR---| I_TMR  EXPIRED |---DONE
+-----+

END_FUNCTION_BLOCK
    
```

a) Nom de bloc fonctionnel comme variable d'entrée (NOTE)

```

FUNCTION_BLOCK
(* Interface externe *)

+-----+
|   INSIDE_B   |
TON---| I_TMR---I_TMR|---TON
BOOL--| TMR_GO EXPIRED|---BOOL
+-----+

(* Corps de bloc fonctionnel *)

      I_TMR
      +-----+
      | TON |
TMR_GO---| IN  Q|---EXPIRED
          | PT ET|
          +-----+
END_FUNCTION_BLOCK

FUNCTION_BLOCK
(* Interface externe *)

+-----+
|   EXAMPLE_B   |
BOOL---| GO      DONE|---BOOL
+-----+

(* Corps de bloc fonctionnel *)

      E_TMR
      +-----+
      | TON |
t#100ms---| IN  Q|
          | PT ET|
          +-----+

      I_BLK
      +-----+
      |   INSIDE_B   |
E_TMR---| I_TMR---I_TMR|
GO-----| TMR_GO  EXPIRED|---DONE
          +-----+

END_FUNCTION_BLOCK

```

b) Nom de bloc fonctionnel comme variable d'entrée-sortie

```

FUNCTION_BLOCK
(* Interface externe *)

+-----+
|   INSIDE_C   |
BOOL--| TMR_GO EXPIRED|---
+-----+

VAR_EXTERNAL X_TMR: TON; END_VAR

(* Corps de bloc fonctionnel *)

      X_TMR
      +-----+
      | TON |
TMR_GO---| IN  Q|---EXPIRED
          | PT ET|
          +-----+
END_FUNCTION_BLOCK

```

```

PROGRAM
(* Interface externe *)

+-----+
|   EXAMPLE_C   |
+-----+
BOOL---|GO      DONE|---BOOL
+-----+

VAR_GLOBAL X_TMR: TON; END_VAR

(* Corps de programme *)
I_BLK
+-----+
|   INSIDE_C   |
+-----+
GO---|TMR_GO  EXPIRED|---DONE
+-----+

END_PROGRAM

```

c) Nom de bloc fonctionnel comme variable externe

NOTE I_TMR n'est pas représenté graphiquement ici étant donné que cela impliquerait l'appel de I_TMR dans INSIDE_A, ce qui est interdit selon les règles 3) et 4) de la Figure 13.

6.6.3.5 Blocs fonctionnels normalisés

6.6.3.5.1 Généralités

Les définitions des blocs fonctionnels normalisés communs à tous les langages de programmation pour automate programmable sont présentées ci-dessous. L'Intégrateur peut fournir des blocs fonctionnels normalisés additionnels.

Lorsque des déclarations graphiques de blocs fonctionnels normalisés sont décrites dans le présent paragraphe, des déclarations textuelles équivalentes peuvent également être spécifiées, comme par exemple dans le Tableau 44.

Les blocs fonctionnels normalisés peuvent être en surcharge, et avoir des entrées et des sorties extensibles. Les définitions de ces types de bloc fonctionnel doivent décrire chaque contrainte concernant le nombre et les types de données de ces entrées et sorties. L'utilisation de ces fonctionnalités dans des blocs fonctionnels non normalisés ne relève pas du domaine d'application de la présente norme.

6.6.3.5.2 Éléments bistables

La forme graphique et le corps de bloc fonctionnel des éléments normalisés bistables sont décrits dans le Tableau 43.

Tableau 43 – Blocs fonctionnels normalisés bistables^a

N°	Description/Forme graphique	Corps de bloc fonctionnel
1a	Bloc fonctionnel bistable (Set dominant): SR(S1, R, Q1)	
	<pre> +-----+ SR S1 Q1 ---BOOL R +-----+ </pre>	<pre> +-----+ S1 ----- >=1 --- Q1 +-----+ R ---O & --- Q1 ----- --- +-----+ </pre>
1b	Bloc fonctionnel bistable (Set dominant) avec noms d'entrée longs: SR(SET1, RESET, Q1)	
	<pre> +-----+ SR SET1 Q1 ---BOOL RESET +-----+ </pre>	<pre> +-----+ SET1 ----- >=1 --- Q1 +-----+ RESET -O & --- Q1 ----- --- +-----+ </pre>
2a	Bloc fonctionnel bistable (Reset dominant): RS(S, R1, Q1)	
	<pre> +-----+ RS S Q1 ---BOOL R1 +-----+ </pre>	<pre> +-----+ R1 -----O & --- Q1 +-----+ S --- >=1 --- Q1 ----- --- +-----+ </pre>
2b	Bloc fonctionnel bistable (Reset dominant) avec noms d'entrée longs: RS(SET, RESET1, Q1)	
	<pre> +-----+ RS SET Q1 ---BOOL R1 +-----+ </pre>	<pre> +-----+ RESET1 -----O & --- Q1 +-----+ SET --- >=1 --- Q1 ----- --- +-----+ </pre>
^a L'état initial de la variable de sortie Q1 doit être la valeur par défaut normale zéro pour les variables booléennes.		

6.6.3.5.3 Détection de front (R_TRIG et F_TRIG)

La représentation graphique des blocs fonctionnels normalisés de détection de front montant et descendant doit être telle que décrite dans le Tableau 44. Les comportements de ces blocs doivent être équivalents aux définitions présentées dans ce tableau. Ce comportement correspond aux règles suivantes:

1. La sortie Q d'un bloc fonctionnel R_TRIG doit conserver la valeur BOOL#1 d'une exécution du bloc fonctionnel à la suivante, conformément à la transition de 0 à 1 de l'entrée CLK, et doit retourner à 0 lors de l'exécution suivante.
2. La sortie Q d'un bloc fonctionnel F_TRIG doit conserver la valeur BOOL#1 d'une exécution du bloc fonctionnel à la suivante, conformément à la transition de 1 à 0 de l'entrée CLK, et doit retourner à 0 lors de l'exécution suivante.

Tableau 44 – Blocs fonctionnels normalisés de détection de front

N°	Description/Forme graphique	Définition (langage ST)
1	Détecteur de front montant: R_TRIG(CLK, Q)	
	<pre> +-----+ R_TRIG BOOL -- CLK Q -- BOOL +-----+ </pre>	<pre> 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 </pre>
2	Détecteur de front descendant: F_TRIG(CLK, Q)	
	<pre> +-----+ F_TRIG BOOL -- CLK Q -- BOOL +-----+ </pre>	<pre> 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 </pre>
NOTE Lorsque l'entrée CLK d'une instance du type R_TRIG est reliée à une valeur BOOL#1, sa sortie Q conserve la valeur BOOL#1 après sa première exécution après un "redémarrage à froid". La sortie Q conservera la valeur BOOL#0 après toutes les exécutions ultérieures. Il en va de même pour une instance F_TRIG dont l'entrée CLK est déconnectée ou reliée à une valeur FALSE.		

6.6.3.5.4 Compteurs

Les représentations graphiques des blocs fonctionnels normalisés compteur, avec les types des entrées et sorties associées, doivent être telles que décrites dans le Tableau 45. Le fonctionnement de ces blocs fonctionnels doit être tel que spécifié dans les corps des blocs fonctionnels correspondants.

Tableau 45 – Blocs fonctionnels normalisés compteur

N°	Description/Forme graphique	Corps de bloc fonctionnel (langage ST)
	Compteur croissant	
1a	CTU_INT(CU, R, PV, Q, CV) ou CTU(...)	
	<pre> +-----+ CTU BOOL-->CU Q ---BOOL BOOL--- R INT--- PV CV ---INT +-----+ </pre> et également: <pre> +-----+ CTU_INT BOOL-->CU Q ---BOOL BOOL--- R INT--- PV CV ---INT +-----+ </pre>	<pre> VAR_INPUT CU: BOOL R_EDGE; ... /* Le front est évalué en interne par le type de données R_EDGE */ IF R THEN CV:= 0; ELSIF CU AND (CV < PVmax) THEN CV:= CV+1; END_IF; Q:= (CV >= PV); </pre>
1b	CTU_DINT PV, CV: DINT	Voir 1a
1c	CTU_LINT PV, CV: LINT	Voir 1a
1d	CTU_UDINT PV, CV: UDINT	Voir 1a
1e	CTU_ULINT(CD, LD, PV, CV) PV, CV: ULINT	Voir 1a
	Compteurs décroissants	

N°	Description/Forme graphique	Corps de bloc fonctionnel (langage ST)
2a	CTD_INT(CD, LD, PV, Q, CV) ou CTD	
	<pre> +-----+ CTD +-----+ BOOL--->CD Q ---BOOL BOOL--- LD INT--- PV CV ---INT +-----+ et également: +-----+ CTD_INT +-----+ BOOL--->CD Q ---BOOL BOOL--- LD INT--- PV CV ---INT +-----+ </pre>	<pre> VAR_INPUT CU: BOOL R_EDGE; ... // Le front est évalué en interne par le type de données R_EDGE IF LD THEN CV:= PV; ELSIF CD AND (CV > PVmin) THEN CV:= CV-1; END_IF; Q:= (CV <= 0); </pre>
2b	CTD_DINT PV, CV: DINT	Voir 2a
2c	CTD_LINT PV, CV: LINT	
2d	CTD_UDINT PV, CV: UDINT	Voir 2a
2e	CTD_ULINT PV, CV: ULINT	Voir 2a
Compteurs croissants-décroissants		
3a	CTUD_INT(CD, LD, PV, Q, CV) ou CTUD(..)	
	<pre> +-----+ CTUD +-----+ BOOL--->CU QU ---BOOL BOOL--->CD QD ---BOOL BOOL--- R BOOL--- LD INT--- PV CV ---INT +-----+ et également: +-----+ CTUD_INT +-----+ BOOL--->CU QU ---BOOL BOOL--->CD QD ---BOOL BOOL--- R BOOL--- LD INT--- PV CV ---INT +-----+ </pre>	<pre> VAR_INPUT CU, CD: BOOL R_EDGE; ... // Le front est évalué en interne par le type de données R_EDGE 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>
3b	CTUD_DINT PV, CV: DINT	Voir 3a
3c	CTUD_LINT PV, CV: LINT	Voir 3a
3d	CTUD_UDINT PV, CV: UDINT	Voir 3a
3e	CTUD_ULINT PV, CV: ULINT	Voir 3a
NOTE Les valeurs numériques des variables limites PVmin et PVmax sont spécifiques de l'Intégrateur.		

6.6.3.5.5 Minuteurs

La forme graphique des blocs fonctionnels normalisés minuteur doit être telle que décrite dans le Tableau 46. Le fonctionnement de ces blocs fonctionnels doit être tel que défini dans les diagrammes temporels décrits à la Figure 15.

Les blocs fonctionnels normalisés minuteur peuvent être utilisés en surcharge avec TIME ou LTIME, ou le type de donnée de base du minuteur normalisé peut être spécifié en tant que TIME ou LTIME.

Tableau 46 – Blocs fonctionnels normalisés minuteur

N°	Description	Symbole	Forme graphique
1a	Impulsion, en surcharge TP	*** est: TP	<pre> +-----+ *** BOOL--- IN Q ---BOOL TIME--- PT ET ---TIME +-----+ PT voir NOTE IN: Entrée (démarrage) PT: Temps prédéfini Q: Sortie ET: Temps écoulé </pre>
1b	Impulsion utilisant TIME	TP_TIME	
1c	Impulsion utilisant LTIME	TP_LTIME	
2a	Délai d'activation, en surcharge TON	TON	
2b	Délai d'activation utilisant TIME	TON_TIME	
2c	Délai d'activation utilisant LTIME	TON_LTIME	
2d ^a	Délai d'activation, en surcharge (graphique)	T---0	
3a	Délai de désactivation, en surcharge TOF	TOF	
3b	Délai de désactivation utilisant TIME	TOF_TIME	
3c	Délai de désactivation utilisant LTIME	TOF_LTIME	
3d ^a	Délai de désactivation, en surcharge (graphique)	0---T	

NOTE L'effet d'une modification de la valeur de l'entrée PT lors de l'opération de temporisation, par exemple, la définition de PT à t#0s pour réinitialiser le fonctionnement d'une instance TP, est un paramètre spécifique de l'Intégrateur.

^a Dans les langages textuels, les caractéristiques 2b et 3b ne doivent pas être utilisées.

La Figure 15 ci-dessous présente les diagrammes temporels des blocs fonctionnels normalisés minuteur.

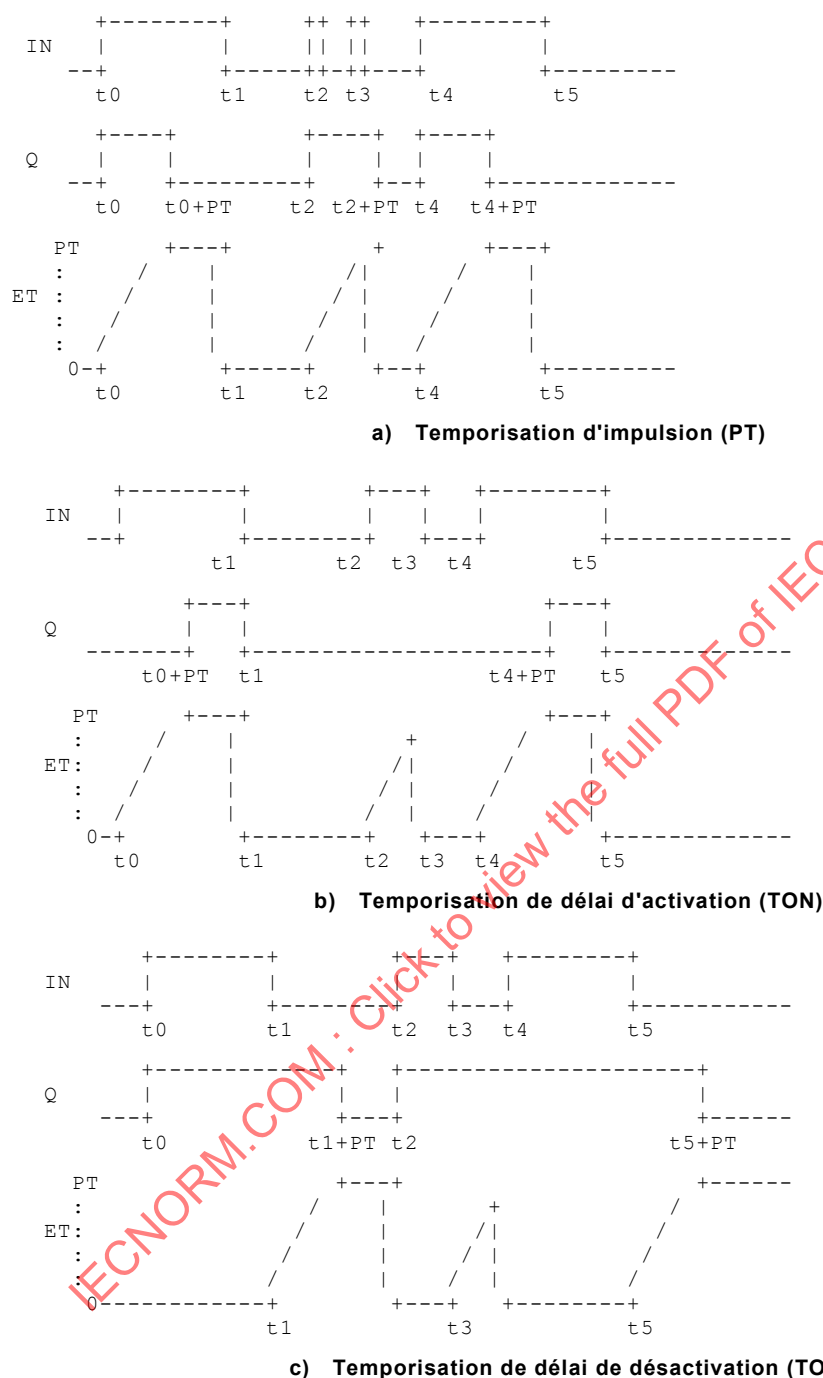


Figure 15 – Blocs fonctionnels normalisés minuteur – diagrammes temporels (règles)

6.6.3.5.6 Blocs fonctionnels de communication

Les blocs fonctionnels normalisés de communication pour automate programmable sont définis dans la CEI 61131-5. Ces blocs fonctionnels apportent une fonctionnalité de communication programmable telle que la vérification d'appareils, l'acquisition de données interrogées, l'acquisition de données programmées, le contrôle des paramètres, le contrôle des protections, le rapport d'alarme programmée ou la gestion et la protection des connexions.

6.6.4 Programmes

Un programme est défini dans la CEI 61131-1 comme étant un "ensemble logique de tous les éléments et constructions de langage de programmation nécessaires au traitement prévu des signaux requis pour le contrôle d'une machine ou d'un processus par une configuration d'AP".

La déclaration et l'utilisation des programmes sont identiques à celles des blocs fonctionnels avec les caractéristiques additionnelles décrites dans le Tableau 47 et les différences suivantes:

1. Les mots-clés limites pour les déclarations de programme doivent être PROGRAM...END_PROGRAM.
2. Un programme peut contenir une construction VAR_ACCESS...END_VAR, qui permet de spécifier des variables nommées auxquelles certains des services de communication spécifiés dans la CEI 61131-5 peuvent accéder. Un chemin d'accès associe chacune de ces variables à une variable d'entrée, de sortie ou interne du programme.
3. Les programmes peuvent être instanciés uniquement dans des ressources tandis que les blocs fonctionnels peuvent être instanciés uniquement dans des programmes ou d'autres blocs fonctionnels.
4. Un programme peut contenir des affectations d'emplacement dans les déclarations de ses variables globales et internes. Des affectations d'emplacement avec une représentation directe partiellement spécifiée peuvent être utilisées uniquement dans la déclaration des variables internes d'un programme.
5. Les caractéristiques d'orientation objet des programmes ne relèvent pas du domaine d'application de la présente partie de la CEI 61131.

Tableau 47 – Déclaration de programme

N°	Description	Exemple
1	Déclaration d'un programme PROGRAM ... END_PROGRAM	PROGRAM myPrg ... END_PROGRAM
2a	Déclaration des entrées VAR_INPUT ... END_VAR	VAR_INPUT IN: BOOL; T1: TIME; END_VAR
2b	Déclaration des sorties VAR_OUTPUT ... END_VAR	VAR_OUTPUT OUT: BOOL; ET_OFF: TIME; END_VAR
2c	Déclaration des entrées-sorties VAR_IN_OUT ... END_VAR	VAR_IN_OUT A: INT; END_VAR
2d	Déclaration des variables temporaires VAR_TEMP ... END_VAR	VAR_TEMP I: INT; END_VAR
2e	Déclaration des variables statiques VAR ... END_VAR	VAR B: REAL; END_VAR
2f	Déclaration des variables externes VAR_EXTERNAL ... END_VAR	VAR_EXTERNAL B: REAL; END_VAR Correspondant à VAR_GLOBAL B: REAL
2g	Déclaration des variables externes VAR_EXTERNAL CONSTANT ... END_VAR	VAR_EXTERNAL CONSTANT B: REAL; END_VAR Correspondant à VAR_GLOBAL B: REAL
3a	Initialisation des entrées	VAR_INPUT MN: INT:= 0;
3b	Initialisation des sorties	VAR_OUTPUT RES: INT:= 1;
3c	Initialisation des variables statiques	VAR B: REAL:= 12.1;
3d	Initialisation des variables temporaires	VAR_TEMP I: INT:= 1;
4a	Déclaration du qualificateur RETAIN pour les variables d'entrée	VAR_INPUT RETAIN X: REAL; END_VAR

N°	Description	Exemple
4b	Déclaration du qualificateur <code>RETAIN</code> pour les variables de sortie	<code>VAR_OUTPUT RETAIN X: REAL; END_VAR</code>
4c	Déclaration du qualificateur <code>NON_RETAIN</code> pour les variables d'entrée	<code>VAR_INPUT NON_RETAIN X: REAL; END_VAR</code>
4d	Déclaration du qualificateur <code>NON_RETAIN</code> pour les variables de sortie	<code>VAR_OUTPUT NON_RETAIN X: REAL; END_VAR</code>
4e	Déclaration du qualificateur <code>RETAIN</code> pour les variables statiques	<code>VAR RETAIN X: REAL; END_VAR</code>
4f	Déclaration du qualificateur <code>NON_RETAIN</code> pour les variables statiques	<code>VAR NON_RETAIN X: REAL; END_VAR</code>
5a	Déclaration du qualificateur <code>RETAIN</code> pour les instances de bloc fonctionnel locales	<code>VAR RETAIN TMR1: TON; END_VAR</code>
5b	Déclaration du qualificateur <code>NON_RETAIN</code> pour les instances de bloc fonctionnel locales	<code>VAR NON_RETAIN TMR1: TON; END_VAR</code>
6a	Déclaration textuelle - d'entrées avec front montant	<pre> 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; (* Exemple de langage ST *) END_PROGRAM </pre>
6b	Déclaration textuelle - d'entrées avec front descendant (textuel)	Voir ci-dessus
7a	Déclaration graphique - d'entrées avec front montant (>)	<pre> PROGRAM (* Interface externe *) +-----+ AND_EDGE +-----+ (* Corps de bloc fonctionnel *) +-----+ +-----+ X-- --Z Y-- +-----+ END_PROGRAM </pre>
7b	Déclaration graphique - d'entrées avec front descendant (<)	Voir ci-dessus
8a	Déclaration <code>VAR_GLOBAL...END_VAR</code> dans un PROGRAM	<code>VAR_GLOBAL z1: BYTE; END_VAR</code>
8b	Déclarations <code>VAR_GLOBAL CONSTANT</code> dans des déclarations de type PROGRAM	<code>VAR_GLOBAL CONSTANT z2: BYTE; END_VAR</code>
9	Déclaration <code>VAR_ACCESS...END_VAR</code> dans un PROGRAM	<pre> VAR_ACCESS ABLE: STATION_1.%IX1.1: BOOL READ_ONLY; BAKER: STATION_1.P1.x2: UINT READ_WRITE; END_VAR </pre>

NOTE Les caractéristiques 2a à 7b sont équivalentes aux mêmes caractéristiques du Tableau 40 pour des blocs fonctionnels.

6.6.5 Classes

6.6.5.1 Généralités

La classe élément de langage prend en charge le paradigme orienté objet et se caractérise par les concepts suivants:

- Définition d'une structure de données divisée en variables publiques et internes,
- Méthode à exécuter sur les éléments de la structure de données,
- Classes, constituées de méthodes (algorithmes) et de structures de données,
- Interface avec des prototypes de méthode et mise en œuvre d'interfaces,
- Héritage d'interfaces et de classes,
- Instanciation de classes.

NOTE Les termes "classe" et "objet" utilisés dans des langages de programmation informatique tels que C#, C++, Java ou UML correspondent aux termes "type" et "instance" utilisés dans les langages de programmation pour automate programmable de la présente norme. Cela apparaît ci-dessous.

Langages de programmation informatique: C#, C++, Java, UML **Langages pour automate programmable de la norme**

Classe (= type d'une classe)

Type d'un bloc fonctionnel et d'une classe

Objet (= instance d'une classe)

Instance d'un bloc fonctionnel et d'une classe

La Figure 16 ci-dessous décrit l'héritage de l'interface et des classes à l'aide des mécanismes de mise en œuvre et d'extension. Il est défini dans le présent 6.6.5.

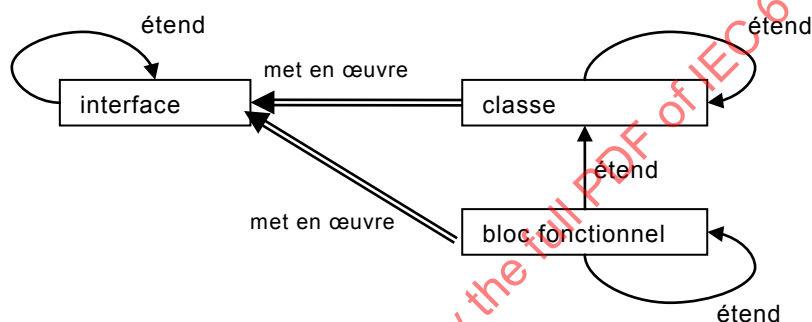


Figure 16 – Présentation de la mise en œuvre d'héritage et d'interface

Une classe est une POU conçue pour la programmation orientée objet. Une classe contient essentiellement des variables et des méthodes. Une classe doit être instanciée avant que ses méthodes ne puissent être appelées ou que ses variables ne puissent être accessibles.

6.6.5.2 Déclaration de classe

Les caractéristiques de la déclaration de classe sont définies dans le Tableau 48:

1. Le mot-clé `CLASS` suivi d'un identificateur spécifiant le nom de la classe déclarée.
2. Le mot-clé de terminaison `END_CLASS`.
3. Les valeurs des variables qui sont déclarées par l'intermédiaire d'une construction `VAR_EXTERNAL` peuvent être modifiées depuis l'intérieur de la classe.
4. Les valeurs des constantes qui sont déclarées par l'intermédiaire d'une construction `VAR_EXTERNAL CONSTANT` ne peuvent pas être modifiées depuis l'intérieur de la classe.
5. Une construction `VAR...END_VAR`, si nécessaire, spécifiant les noms et les types des variables de la classe.
6. Les variables peuvent être initialisées.
7. Les variables de la section `VAR` (statique) peuvent être déclarées comme `PUBLIC`. On peut accéder à une variable publique depuis l'extérieur de la classe en utilisant la même syntaxe que pour l'accès aux sorties de bloc fonctionnel.
8. Le qualificateur `RETAIN` ou `NON_RETAIN` peut être utilisé pour les variables internes d'une classe.

9. La notation avec astérisque "*" telle que définie dans le Tableau 16 peut être utilisée dans la déclaration des variables internes d'une classe.
10. Les variables peuvent être `PUBLIC`, `PRIVATE`, `INTERNAL` ou `PROTECTED`. Le spécificateur d'accès `PROTECTED` est défini par défaut.
11. Une classe peut prendre en charge l'héritage d'autres classes pour étendre une classe de base.
12. Une classe peut mettre en œuvre une ou plusieurs interfaces.
13. Les instances d'autres blocs fonctionnels, classes et blocs fonctionnels orientés objet peuvent être déclarées dans les sections de variables `VAR` et `VAR_EXTERNAL`.
14. Il convient qu'une instance de classe déclarée à l'intérieur d'une classe n'utilise pas le même nom qu'une fonction (présentant la même portée pour les noms) afin d'éviter toute ambiguïté.

La classe présente les différences suivantes par rapport au bloc fonctionnel:

- Les mots-clés `FUNCTION_BLOCK` et `END_FUNCTION_BLOCK` sont remplacés par `CLASS` et `END_CLASS`, respectivement.
- Les variables sont déclarées uniquement dans la section `VAR`. Les sections `VAR_INPUT`, `VAR_OUTPUT`, `VAR_IN_OUT`, et `VAR_TEMP` ne sont pas autorisées.
- Une classe n'a pas de corps. Une classe ne peut définir que des méthodes.
- L'appel d'une instance d'une classe n'est pas possible. Seules les méthodes d'une classe peuvent être appelées.

L'intégrateur des classes doit fournir un sous-ensemble intrinsèquement cohérent de caractéristiques définies dans le Tableau 48 ci-dessous.

Tableau 48 – Classe

N°	Description Mot-clé	Explication
1	<code>CLASS ... END_CLASS</code>	Définition d'une classe
1a	Spécificateur <code>FINAL</code>	Une classe ne peut pas être utilisée comme une classe de base.
	Adapté du bloc fonctionnel	
2a	Déclaration de variables <code>VAR ... END_VAR</code>	<code>VAR B: REAL; END_VAR</code>
2b	Initialisation de variables	<code>VAR B: REAL:= 12.1; END_VAR</code>
3a	Qualificateur <code>RETAIN</code> pour les variables internes	<code>VAR RETAIN X: REAL; END_VAR</code>
3b	Qualificateur <code>NON_RETAIN</code> pour les variables internes	<code>VAR NON_RETAIN X: REAL; END_VAR</code>
4a	Déclarations <code>VAR_EXTERNAL</code> dans des déclarations de type de classe	Pour un exemple équivalent, voir le Tableau 40.
4b	Déclarations <code>VAR_EXTERNAL</code> <code>CONSTANT</code> dans des déclara- tions de type de classe	Pour un exemple équivalent, voir le Tableau 40.
	Méthodes et spécificateurs	
5	<code>METHOD...END_METHOD</code>	Définition de méthode
5a	Spécificateur <code>PUBLIC</code>	Une méthode peut être appelée depuis n'importe quel emplacement.
5b	Spécificateur <code>PRIVATE</code>	Une méthode peut être appelée uniquement depuis l'intérieur de la POU la définissant.
5c	Spécificateur <code>INTERNAL</code>	Une méthode peut être appelée uniquement depuis l'intérieur du même espace de noms.

N°	Description Mot-clé	Explication
5d	Spécificateur <code>PROTECTED</code>	Une méthode peut être appelée uniquement depuis l'intérieur de la POU la définissant et de ses dérivations (par défaut).
5e	Spécificateur <code>FINAL</code>	Une méthode ne doit pas être supplantée.
	Héritage	- Ces caractéristiques sont identiques à celle de l'héritage dans le Tableau 53.
6	<code>EXTENDS</code>	La classe hérite d'une classe (NOTE: aucun héritage du bloc fonctionnel).
7	<code>OVERRIDE</code>	La méthode supplante la méthode de base – voir la liaison de nom dynamique.
8	<code>ABSTRACT</code>	Classe abstraite – au moins une méthode est abstraite. Méthode abstraite – cette méthode est abstraite.
	Référence d'accès	
9a	<code>THIS</code>	Référence aux méthodes propriétaires
9b	<code>SUPER</code>	Référence d'accès à la méthode de la classe de base
	Spécificateurs d'accès à une variable	
10a	Spécificateur <code>PUBLIC</code>	On peut accéder à la variable depuis n'importe quel emplacement.
10b	Spécificateur <code>PRIVATE</code>	On ne peut accéder à la variable que depuis l'intérieur de la POU la définissant.
10c	Spécificateur <code>INTERNAL</code>	On ne peut accéder à la variable que depuis l'intérieur du même espace de noms.
10d	Spécificateur <code>PROTECTED</code>	On ne peut accéder à la variable que depuis l'intérieur de la POU la définissant et de ses dérivations (par défaut).
	Polymorphisme	
11a	avec <code>VAR_IN_OUT</code>	Une instance d'une classe dérivée peut être affectée au <code>VAR_IN_OUT</code> d'une classe (de base).
11b	avec référence	L'adresse d'une instance d'une classe dérivée peut être affectée à une référence à une classe (de base).

L'exemple ci-dessous décrit les caractéristiques de la déclaration de classe et son utilisation.

EXEMPLE Déclaration de classe

```

Class CCounter
  VAR
    m_iCurrentValue: INT;          (* Par défaut = 0 *)
    m_bCountUp: BOOL:=TRUE;
  END_VAR
  VAR PUBLIC
    m_iUpperLimit: INT:=+10000;
    m_iLowerLimit: INT:=-10000;
  END_VAR

  METHOD Count (* Corps uniquement *)
    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 Déclaration d'instance de classe

Une instance de classe doit être déclarée de manière similaire à ce qui est défini pour les variables structurées.

Lorsqu'une instance de classe est déclarée, les valeurs initiales des variables publiques de cette instance peuvent être affectées dans une liste d'initialisation entre parenthèses à la suite de l'opérateur d'affectation suivant l'identificateur de classe comme décrit dans le Tableau 49.

Les éléments qui ne sont pas affectés dans la liste d'initialisation doivent avoir les valeurs initiales de la déclaration de classe.

Tableau 49 – Déclaration d'instance de classe

N°	Description	Exemple
1	Déclaration d'une ou plusieurs instances de classe avec initialisation par défaut	<pre> VAR MyCounter1: CCounter; END_VAR </pre>
2	Déclaration d'une instance de classe avec initialisation de ses variables publiques	<pre> VAR MyCounter2: CCounter:= (m_iUpperLimit:=20000; m_iLowerLimit:=-20000); END_VAR </pre>

6.6.5.4 Méthodes d'une classe**6.6.5.4.1 Généralités**

Dans le contexte des langages pour automate programmable, le concept de méthodes connu dans la programmation orientée objet est adopté comme un ensemble d'éléments de langage facultatifs défini dans la définition de classe.

Des méthodes peuvent être appliquées pour définir les opérations à effectuer sur les données d'une instance de classe.

6.6.5.4.2 Signature

Dans le contexte de la présente norme, le terme "signature" est défini à l'Article 3 comme un ensemble d'informations définissant de façon non ambiguë l'identité de l'interface des paramètres d'un objet `METHOD`.

Une signature est constituée des éléments suivants:

- un nom de méthode,
- un type de résultat,
- des noms de variable, des types de données et l'ordre de tous ses paramètres (entrées, sorties, variables d'entrée-sortie).

Les variables locales ne font pas partie de la signature. `VAR_EXTERNAL` et les variables constantes ne sont pas pertinentes pour la signature.

Les spécificateurs d'accès tels que `PUBLIC` ou `PRIVATE` ne sont pas pertinents pour la signature.

6.6.5.4.3 Déclaration et exécution de méthode

Une classe peut avoir un ensemble de méthodes.

La déclaration d'une méthode doit satisfaire aux règles suivantes:

1. Les méthodes sont déclarées dans la portée d'une classe.
2. Une méthode peut être définie dans l'un quelconque des langages de programmation spécifiés dans la présente norme.
3. Dans la déclaration textuelle, les méthodes sont répertoriées après la déclaration des variables de la classe.
4. Une méthode peut déclarer son propre `VAR_INPUT`, des variables temporaires internes `VAR` et `VAR_TEMP`, `VAR_OUTPUT`, `VAR_IN_OUT`, et un résultat de méthode.
Les mots-clés `VAR_TEMP` et `VAR` ont la même signification et sont tous deux permis pour les variables internes. (`VAR` est utilisé dans des fonctions).
5. La déclaration de méthode doit contenir un des spécificateurs d'accès suivants: `PUBLIC`, `PRIVATE`, `INTERNAL` et `PROTECTED`. Si aucun spécificateur d'accès n'est défini, la méthode sera par défaut `PROTECTED`.
6. La déclaration de méthode peut contenir le mot-clé additionnel `OVERRIDE` ou `ABSTRACT`.

NOTE 1 La surcharge de méthodes ne relève pas du domaine d'application de la présente partie de la CEI 61131.

L'exécution d'une méthode doit satisfaire aux règles suivantes:

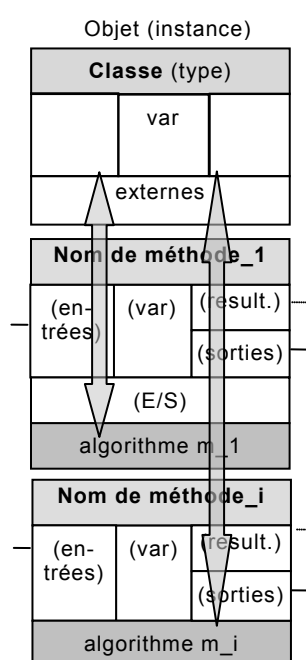
7. Lorsqu'elle est exécutée, une méthode peut lire ses entrées, et elle calcule ses sorties et son résultat à l'aide de ses variables temporaires.
8. Le résultat de méthode est affecté au nom de méthode.
9. L'ensemble des variables d'une méthode et le résultat sont temporaires (comme les variables d'une fonction), c'est-à-dire que les valeurs ne sont pas stockées d'une exécution de méthode à la suivante. Par conséquent, l'évaluation des variables de sortie d'une méthode est possible uniquement dans le contexte immédiat de l'appel de méthode.
10. Les noms de variable de chaque méthode et de la classe doivent être différents (uniques).

Les noms des variables locales de méthodes différentes peuvent être identiques.

11. Toutes les méthodes ont un accès en lecture/écriture aux variables statiques et externes déclarées dans la classe.
12. L'ensemble des variables et des résultats peut être à valeurs multiples, c'est-à-dire un tableau ou une structure. Comme défini pour les fonctions, le résultat de méthode peut être utilisé en tant qu'opérande dans une expression.
13. Lorsqu'elle est exécutée, une méthode peut utiliser d'autres méthodes définies dans cette classe. Les méthodes de cette instance de classe doivent être appelées à l'aide du mot-clé **THIS**.

L'exemple ci-dessous décrit la déclaration simplifiée d'une classe avec deux méthodes et l'appel de la méthode.

EXEMPLE 1



NOTE 2

Les algorithmes des méthodes ont accès à leurs propres données et aux données de la classe.

(Les paramètres temporaires sont entre parenthèses.)

Déclaration de la classe (type) avec des méthodes:

```

CLASS nom
  VAR variables var; END_VAR
  VAR_EXTERNAL variables externes; END_VAR

METHOD nom_1
  VAR_INPUT entrées; END_VAR
  VAR_OUTPUT sorties; END_VAR
END_METHOD

METHOD nom_i
  VAR_INPUT entrées; END_VAR
  VAR_OUTPUT sorties; END_VAR
END_METHOD
END_CLASS
  
```

NOTE 3

Cette représentation graphique de la méthode est donnée à titre d'illustration uniquement.

Appel d'une méthode:

a) Utilisation du résultat: (le résultat est facultatif)
`R1:= I.method1(inm1:= A, outm1 => Y);`

b) Utilisation de l'appel: (pas de résultat déclaré)
`I.method1(inm1:= A, outm1 => Y);`

Affectation des entrées de méthode depuis l'extérieur:

~~`I.inm1 := A;`~~ // Non permis;

Lecture des sorties de méthode depuis l'extérieur:
~~`Y:= I.outm1;`~~ // Non permis,

EXEMPLE 2

Classe **COUNTER** avec deux méthodes de comptage croissant. La méthode **UP5** décrit comment appeler une méthode de la même classe.

```

CLASS COUNTER
VAR
    CV: UINT;                                // Valeur actuelle du compteur
    Max: UINT:= 1000;
END_VAR
METHOD PUBLIC UP: UINT                        // Méthode pour un comptage croissant
VAR_INPUT INC: UINT; END_VAR                par inc
VAR_OUTPUT QU: BOOL; END_VAR               // Incrément
                                           // Détection de limite supérieure
    IF CV <= Max - INC
    THEN CV:= CV + INC;
        QU:= FALSE;
    ELSE QU:= TRUE;
    END_IF
    UP:= CV;                                // Limite supérieure atteinte
END_METHOD                                  // Résultat de la méthode

METHOD PUBLIC UP5: UINT                     // Comptage croissant de 5
VAR_OUTPUT QU: BOOL; END_VAR               // Limite supérieure atteinte
UP5:= THIS.UP(INC:= 5, QU => QU);          // Appel de méthode interne
END_METHOD
END_CLASS

```

6.6.5.4.4 Représentation d'un appel de méthode

Les méthodes peuvent être appelées dans des langages textuels (Tableau 50) et dans des langages graphiques.

Dans toutes les représentations de langage, il existe deux cas différents d'appels d'une méthode:

a) Un appel interne

d'une méthode d'une instance de la classe propriétaire. Le nom de la méthode doit être précédé de "THIS".

Cet appel peut être émis par une autre méthode.

b) Un appel externe

d'une méthode d'une instance d'une autre classe. Le nom de la méthode doit être précédé du nom d'instance et de ".".

Cet appel peut être émis par une méthode ou un corps de bloc fonctionnel où l'instance est déclarée.

NOTE La syntaxe suivante est utilisée:

- La syntaxe `A()` est utilisée pour appeler une fonction globale A.
- La syntaxe `THIS.A()` est utilisée pour appeler une méthode de l'instance elle-même.
- La syntaxe `I1.A()` est utilisée pour appeler une méthode A d'une autre instance I1.

6.6.5.4.5 Représentation textuelle d'un appel

Une méthode avec résultat doit être appelée en tant qu'opérande d'une expression.

Une méthode sans résultat ne doit pas être appelée à l'intérieur d'une expression.

La méthode peut être appelée de manière formelle ou informelle.

L'appel externe d'une méthode requiert en outre le nom de l'instance de classe externe.

EXEMPLE 1 ... `class_instance_name.method_name(parameters)`

L'appel interne d'une méthode utilise `THIS` à la place du nom d'instance.

EXEMPLE 2 ... `THIS.method_name (parameters)`

**Tableau 50 – Appel textuel de méthodes –
Liste des paramètres formels et informels**

N°	Description	Exemple
1a	Appel formel complet (textuel uniquement) Doit être utilisé si EN/ENO est nécessaire dans les appels.	<code>A:= COUNTER.UP(EN:= TRUE, INC:= B, START:= 1, ENO=> %MX1, QU => C);</code>
1b	Appel formel incomplet (textuel uniquement) Doit être utilisé si EN/ENO n'est pas nécessaire dans les appels.	<code>A:= COUNTER.UP(INC:= B, QU => C);</code> La variable <code>START</code> aura la valeur par défaut 0 (zéro).
2	Appel informel (textuel uniquement) (corriger l'ordre et terminer)	<code>A:= COUNTER.UP(B, 1, C);</code> Cet appel est équivalent à 1a, mais sans EN/ENO.

6.6.5.4.6 Représentation graphique

La représentation graphique d'un appel de méthode est similaire à la représentation d'une fonction ou d'un bloc fonctionnel. Il s'agit d'un bloc rectangulaire avec des entrées sur le côté gauche et des sorties sur le côté droit du bloc.

Les appels de méthode peuvent prendre en charge EN et ENO comme défini dans le Tableau 18.

- L'appel interne présente le nom de classe et le nom de méthode séparés par un point à l'intérieur d'un bloc.
Le mot-clé `THIS` doit être situé au-dessus du bloc.
- L'appel externe présente le nom de classe et le nom de méthode séparés par un point à l'intérieur d'un bloc.
Le nom d'instance de classe doit être situé au-dessus du bloc.

6.6.5.4.7 Erreur

L'utilisation d'une sortie de méthode indépendante de l'appel de méthode doit être traitée comme une erreur.

Voir l'exemple ci-dessous.

EXEMPLE Appel de méthode interne et externe

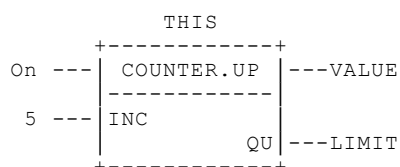
```
VAR
  CT:    COUNTER;
  LIMIT: BOOL;
  VALUE: UINT;
END_VAR
```

1) En texte structuré (ST)

- a) Appel interne d'une méthode:
`VALUE:= THIS.UP (INC:= 5, QU => LIMIT);`
- b) Appel externe d'une méthode:
`VALUE:= CT.UP (INC:= 5, QU => LIMIT);`

2) Dans un diagramme de bloc fonctionnel (FBD)

- a) Appel interne d'une méthode:

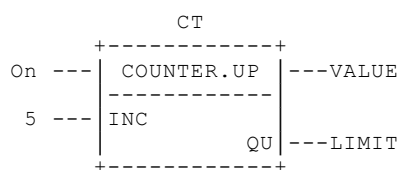


Appelé dans une classe par une autre méthode

THIS est obligatoire
La méthode UP retourne le résultat

La représentation graphique est donnée à titre d'illustration uniquement
La variable On active l'appel de méthode

- b) Appel externe d'une méthode:



CT est une instance de classe déclarée dans une autre classe ou un autre bloc fonctionnel.

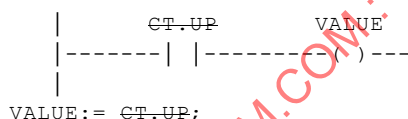
Appelé par une méthode ou un corps de bloc fonctionnel

La méthode UP retourne le résultat

La représentation graphique est donnée à titre d'illustration uniquement

La variable On active l'appel de méthode

3) Erreur: utilisation de la sortie de méthode sans appel graphique ni textuel



Cette évaluation de la sortie de méthode n'est PAS possible parce qu'une méthode ne stocke pas ses sorties d'une exécution à la suivante.

6.6.5.5 Héritage de classe (EXTENDS, SUPER, OVERRIDE, FINAL)

6.6.5.5.1 Généralités

Dans le contexte des langages pour automate programmable, le concept d'héritage défini dans la programmation orientée objet générale est ici adapté comme un moyen de créer de nouveaux éléments.

L'héritage de classes est décrit à la Figure 17. Une ou plusieurs classes peuvent être dérivées sur la base d'une classe existante. Cette opération peut être répétée plusieurs fois.

NOTE L'"héritage multiple" n'est pas pris en charge.

Une classe dérivée (enfant) étend généralement la classe de base (parent) par des méthodes additionnelles.

Le terme classe "de base" désigne tous les "ancêtres", c'est-à-dire la classe parent, ses classes parent, etc.

Héritage de classe à l'aide de EXTENDS

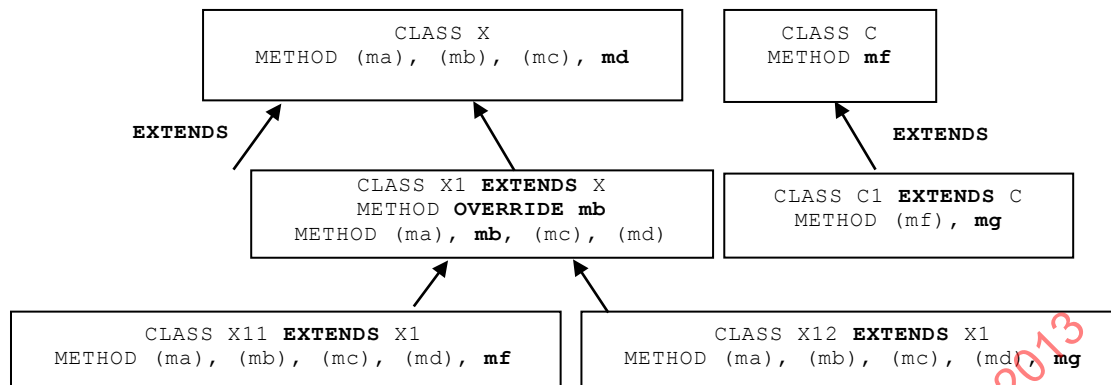


Figure 17 – Héritage de classes (illustration)

6.6.5.5.2 EXTENDS (extension) de classe

Une classe peut être dérivée d'une classe déjà existante (classe de base) à l'aide du mot-clé **EXTENDS**.

EXEMPLE `CLASS X1 EXTENDS X;`

Les règles suivantes doivent s'appliquer:

1. La classe dérivée hérite sans déclarations supplémentaires de toutes les méthodes (le cas échéant) de sa classe de base avec les exceptions suivantes:
 - Les méthodes **PRIVATE** ne sont pas héritées.
 - Les méthodes **INTERNAL** ne sont pas héritées à l'extérieur de l'espace de noms.
2. La classe dérivée hérite de toutes les variables (le cas échéant) de sa classe de base.
3. Une classe dérivée hérite d'une seule classe de base.

L'héritage multiple n'est pas pris en charge par la présente norme.

NOTE Une classe peut mettre en œuvre (à l'aide du mot-clé **IMPLEMENTS**) une ou plusieurs interfaces.

4. La classe dérivée peut étendre la classe de base, c'est-à-dire qu'elle peut avoir ses propres méthodes et variables en plus des méthodes et variables héritées de la classe de base, et ainsi créer une nouvelle fonctionnalité.
5. La classe utilisée en tant que classe de base peut elle-même être une classe dérivée. Elle transmet en outre à une classe dérivée les méthodes et les variables dont elle a hérité.

Cette opération peut être répétée plusieurs fois.

6. Si la définition de la classe de base est modifiée, la fonctionnalité de toutes les classes dérivées (et de leurs enfants) change aussi.

6.6.5.5.3 OVERRIDE (supplanter) une méthode

Une classe dérivée peut supplanter (remplacer) une ou plusieurs méthodes héritées en utilisant sa propre mise en œuvre de cette ou de ces méthodes. En ce qui concerne le remplacement des méthodes de base, les règles suivantes s'appliquent:

1. La méthode qui supprime une méthode héritée doit avoir la même signature (nom de méthode et variables) dans la portée de la classe dérivée.
2. La méthode qui supprime une méthode héritée doit avoir les caractéristiques suivantes:

- Le mot-clé `OVERRIDE` suit le mot-clé `METHOD`.
- La classe dérivée a accès à la méthode de base, qui est `PUBLIC`, `PROTECTED` ou `INTERNAL` dans le même espace de noms.
- La nouvelle méthode doit avoir les mêmes spécificateurs d'accès, mais le spécificateur de méthode `FINAL` peut être utilisé pour une méthode supplantée.

EXEMPLE `METHOD OVERRIDE mb;`

6.6.5.5.4 `FINAL` pour des classes et des méthodes

Une méthode avec le spécificateur `FINAL` ne doit pas être supplantée.

Une classe avec le spécificateur `FINAL` ne peut pas être une classe de base.

EXEMPLE 1 `METHOD FINAL mb;`

EXEMPLE 2 `CLASS FINAL c1;`

6.6.5.5.5 Erreurs pour `EXTENDS`, `SUPER`, `OVERRIDE`, `FINAL`

La situation suivante doit être traitée comme une erreur:

1. La classe dérivée définit une variable avec le nom d'une variable déjà contenue dans sa classe de base, qu'elle soit définie dans celle-ci ou héritée. Cette règle ne s'applique pas aux variables `PRIVATE`.
2. La classe dérivée définit une méthode avec le nom d'une variable déjà contenue dans sa classe de base.
3. La classe dérivée est dérivée de sa propre classe de base, directement ou indirectement, c'est-à-dire que la récursion n'est pas permise.
4. La classe définit une méthode avec le mot-clé `OVERRIDE` qui ne supprime pas une méthode d'une classe de base.

EXEMPLE Héritage et remplacement

Classe qui étend la classe 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                                // Deuxième éclairage

METHOD PUBLIC OVERRIDE DAYTIME
    LIGHT := FALSE;                                       // Accès à la mise en œuvre
    LIGHT2:= FALSE;                                       // spécifique de la variable du parent
END_METHOD

METHOD PUBLIC OVERRIDE NIGHTTIME
    LIGHT := TRUE;                                         // Accès à la mise en œuvre
    LIGHT2:= TRUE;                                         // spécifique de la variable du parent
END_METHOD

END_CLASS

```

6.6.5.6 Liaison de nom dynamique (OVERRIDE)

La liaison de nom est l'association d'un nom de méthode et d'une mise en œuvre de méthode. La liaison d'un nom (par exemple, par le compilateur) avant l'exécution du programme est appelée liaison statique ou "précoce". Une liaison établie pendant l'exécution du programme est appelée liaison dynamique ou "tardive".

Dans le cas d'un appel de méthode interne, la fonctionnalité de remplacement avec le mot-clé **OVERRIDE** provoque une différence entre la forme statique et la forme dynamique de la liaison de nom:

- **Liaison statique**
associe le nom de méthode à la mise en œuvre de méthode de la classe lors d'un appel de méthode interne ou contient la méthode effectuant l'appel de méthode interne.
- **Liaison dynamique**
associe le nom de méthode à la mise en œuvre de méthode du type réel de l'instance de classe.

EXEMPLE 1 Liaison de nom dynamique

Remplacement avec effet sur la liaison.

// Déclaration

```

CLASS CIRCLE

METHOD PUBLIC PI: LREAL          // La méthode produit un PI moins précis
    PI:= 3.1415;
END_METHOD

METHOD PUBLIC CF: LREAL          // La méthode produit la circonférence
    VAR_INPUT DIAMETER: LREAL; END_VAR
    CF:= THIS.PI() * DIAMETER;    // Appel interne de méthode PI
END_METHOD                      // à l'aide de la liaison dynamique de PI
END_CLASS

CLASS CIRCLE2 EXTENDS CIRCLE     // Classe avec remplacement de méthode PI
METHOD PUBLIC OVERRIDE PI: LREAL // La méthode produit un PI plus précis
    PI:= 3.1415926535897;
END_METHOD
END_CLASS

PROGRAM TEST
VAR
    CIR1:    CIRCLE;              // Instance de CIRCLE
    CIR2:    CIRCLE2;            // Instance de CIRCLE2
    CUMF1:    LREAL;
    CUMF2:    LREAL;
    DYNAMIC:  BOOL;
END_VAR

    CUMF1:= CIR1.CF(1.0);          // Appel de méthode CIR1
    CUMF2:= CIR2.CF(1.0);          // Appel de méthode CIR2
    DYNAMIC:= CUMF1 <> CUMF2;      // La liaison dynamique donne True
END_PROGRAM
    
```

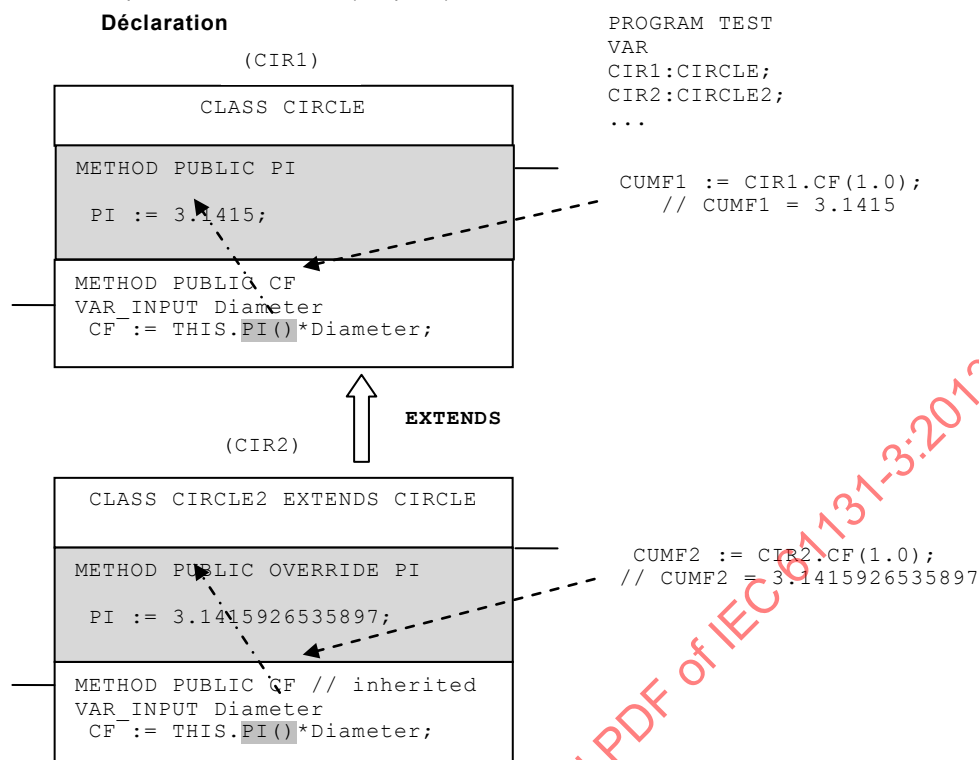
Dans cet exemple, la classe `CIRCLE` contient un appel interne de sa méthode `PI` avec une précision faible pour calculer la circonférence (`CF`) d'un cercle.

La classe dérivée `CIRCLE2` remplace cette méthode par une définition plus précise de `PI`.

L'appel de la méthode `PI()` désigne soit `CIRCLE.PI`, soit `CIRCLE2.PI` selon le type de l'instance sur laquelle l'appel de `CF` a été effectué. Ici, `CUMF2` est plus précis que `CUMF1`.

EXEMPLE 2

Illustration de l'exemple textuel ci-dessus (simplifié)



6.6.5.7 Appel de méthode de classe propriétaire et de base (THIS, SUPER)

6.6.5.7.1 Généralités

Pour accéder à une méthode définie à l'intérieur ou à l'extérieur de la classe propriétaire, les mots-clés **THIS** et **SUPER** sont disponibles.

6.6.5.7.2 THIS

THIS est une référence à l'instance de la classe propriétaire.

Avec le mot-clé **THIS**, une méthode de l'instance de la classe propriétaire peut être appelée depuis une autre méthode de cette instance de classe.

THIS peut être transmis à une variable du type **INTERFACE**.

Le mot-clé **THIS** ne peut pas être utilisé avec une autre instance; par exemple, l'expression `myInstance.THIS` n'est pas autorisée.

EXEMPLE Utilisation du mot-clé `THIS`.

Des exemples antérieurs sont copiés ici pour des raisons de commodité.

```

INTERFACE ROOM
    METHOD DAYTIME    END_METHOD          // Appelé pendant la journée
    METHOD NIGHTTIME  END_METHOD          // Appelé pendant la nuit
END_INTERFACE

FUNCTION_BLOCK ROOM_CTRL                //
    VAR_INPUT
        RM: ROOM;                      // Interface ROOM en tant que type de variable
    d'entrée
    END_VAR
    VAR_EXTERNAL
        Actual_TOD: TOD;                // Définition du temps global
    END_VAR
    IF (RM = NULL)                      // Important: référence valide d'essai!
    THEN RETURN;
    END_IF;
    IF Actual_TOD >= TOD#20:15 OR Actual_TOD <= TOD#6:00
    THEN RM.NIGHTTIME();                 // appel de méthode de RM
    ELSE RM.DAYTIME();
    END_IF;

END_FUNCTION_BLOCK

// Applique le mot-clé THIS pour affecter l'instance elle-même

CLASS DARKROOM IMPLEMENTS ROOM          // ROOM voir ci-dessus
VAR_EXTERNAL
    Ext_Room_Ctrl: ROOM_CTRL;           // ROOM_CTRL voir ci-dessus
END_VAR

METHOD PUBLIC DAYTIME;    END_METHOD
METHOD PUBLIC NIGHTTIME; END_METHOD

METHOD PUBLIC EXT_1
    Ext_Room_Ctrl(RM:= THIS);           // Appel de Ext_Room_Ctrl avec l'instance elle-
    même
END_METHOD
END_CLASS
    
```

6.6.5.7.3 SUPER

`SUPER` permet d'accéder à des méthodes de la mise en œuvre de classe de base.

Avec le mot-clé `SUPER`, une méthode valide dans l'instance de la classe de base (parent) peut être appelée. Par conséquent, la liaison de nom statique se produit.

Le mot-clé `SUPER` ne peut pas être utilisé avec une autre instance; par exemple, l'expression `myRoom.SUPER.DAYTIME()` n'est pas autorisée.

Le mot-clé `SUPER` ne peut pas être utilisé pour accéder à d'autres méthodes dérivées; par exemple, l'expression `SUPER.SUPER.aMethod` n'est pas prise en charge.

EXEMPLE Utilisation du mot-clé `SUPER` et du polymorphisme.

`LIGHT2ROOM` utilisant `SUPER` comme mise en œuvre alternative à l'exemple ci-dessus.
Des exemples antérieurs sont copiés ici pour des raisons de commodité.

```

INTERFACE ROOM
    METHOD DAYTIME    END_METHOD // Appelé pendant la journée
    METHOD NIGHTTIME  END_METHOD // Appelé pendant la nuit
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 en tant que type de variable
    END_VAR

    VAR_EXTERNAL
        Actual_TOD: TOD;                          // Définition du temps global
    END_VAR

    IF (RM = NULL)                                // Important: référence valide d'essai!
    THEN RETURN;
    END_IF;

    IF Actual_TOD >= TOD#20:15 OR
        Actual_TOD <= TOD#06:00
    THEN RM.NIGHTTIME();                          // Appel de méthode de RM (liaison dynamique) à
                                                // LIGHTROOM.NIGHTTIME
                                                // ou LIGHT2ROOM.NIGHTTIME)

    ELSE RM.DAYTIME();
    END_IF;
END_FUNCTION_BLOCK

// Applique le mot-clé SUPER pour appeler une méthode la classe de base
CLASS LIGHT2ROOM EXTENDS LIGHTROOM               // Voir ci-dessus
VAR LIGHT2: BOOL; END_VAR                        // Deuxième éclairage

METHOD PUBLIC OVERRIDE DAYTIME
    SUPER.DAYTIME();                             // Appel de méthode dans LIGHTROOM
    LIGHT2:= TRUE;
END_METHOD

METHOD PUBLIC OVERRIDE NIGHTTIME
    SUPER.NIGHTTIME()                             // Appel de méthode dans LIGHTROOM
    LIGHT2:= FALSE;
END_METHOD
END_CLASS

// Utilisation du polymorphisme et de la liaison dynamique
PROGRAM C
VAR
    MyRoom1: LIGHTROOM;                          // Voir ci-dessus
    MyRoom2: LIGHT2ROOM;                         // Voir ci-dessus
    My_Room_Ctrl: ROOM_CTRL;                     // Voir ci-dessus
END_VAR

    My_Room_Ctrl(RM:= MyRoom1);                  // Appels dans les méthodes d'appel My_Room_Ctrl de
LIGHTROOM
    My_Room_Ctrl(RM:= MyRoom2);                  // Appels dans les méthodes d'appel My_Room_Ctrl de
LIGHT2ROOM
END_PROGRAM

```

6.6.5.8 Classe ABSTRACT et méthode ABSTRACT

6.6.5.8.1 Généralités

Le modificateur **ABSTRACT** peut être utilisé avec des classes ou avec des méthodes uniques. L'Intégrateur doit déclarer la mise en œuvre de ces caractéristiques conformément au Tableau 48.

6.6.5.8.2 Classe abstraite

L'utilisation du modificateur `ABSTRACT` dans une déclaration de classe indique qu'une classe est destinée à être un type de base pour d'autres classes à utiliser pour l'héritage.

EXEMPLE `CLASS ABSTRACT A1`

La classe abstraite a les caractéristiques suivantes:

- Une classe abstraite ne peut pas être instanciée.
- Une classe abstraite doit contenir au moins une méthode abstraite.

Une classe (non abstraite) dérivée d'une classe abstraite doit comprendre des mises en œuvre réelles de toutes les méthodes abstraites héritées.

Une classe abstraite peut être utilisée en tant que type d'un paramètre d'entrée ou d'entrée-sortie.

6.6.5.8.3 Méthode abstraite

Toutes les méthodes d'une classe abstraite qui sont marquées comme étant `ABSTRACT` doivent être mises en œuvre par des classes dérivées de cette classe abstraite, si la classe dérivée elle-même n'est pas marquée comme étant `ABSTRACT`.

Les méthodes d'une classe qui sont héritées d'une interface doivent avoir le mot-clé `ABSTRACT` si elles ne sont pas encore mises en œuvre.

Le mot-clé `ABSTRACT` ne doit pas être utilisé en combinaison avec le mot-clé `OVERRIDE`.

Le mot-clé `ABSTRACT` ne peut être utilisé que sur les méthodes d'une classe abstraite.

EXEMPLE `METHOD PUBLIC ABSTRACT M1`

6.6.5.9 Spécificateurs d'accès aux méthodes (`PROTECTED`, `PUBLIC`, `PRIVATE`, `INTERNAL`)

Les emplacements à partir desquels l'appel de chaque méthode est permis doivent être définis. L'accessibilité d'une méthode est définie à l'aide d'un des spécificateurs d'accès ci-dessous, placé après le mot-clé `METHOD`.

- **`PROTECTED`**

Si l'héritage est mis en œuvre, le spécificateur d'accès `PROTECTED` est applicable. Il indique que les méthodes associées sont accessibles uniquement de l'intérieur d'une classe et de l'intérieur de toutes les classes dérivées.

`PROTECTED` est défini par défaut et peut être omis.

NOTE Si l'héritage n'est pas pris en charge, le spécificateur d'accès par défaut `PROTECTED` a le même effet que `PRIVATE`.

- **`PUBLIC`**

Le spécificateur d'accès `PUBLIC` indique que les méthodes associées sont accessibles partout où la classe peut être utilisée.

- **`PRIVATE`**

Le spécificateur d'accès `PRIVATE` indique que les méthodes associées sont accessibles uniquement depuis l'intérieur de la classe elle-même.

- **INTERNAL**

Si l'espace de noms est mis en œuvre, le spécificateur d'accès **INTERNAL** est applicable. Il indique que les méthodes associées sont accessibles uniquement depuis le **NAMESPACE** dans lequel la classe est déclarée.

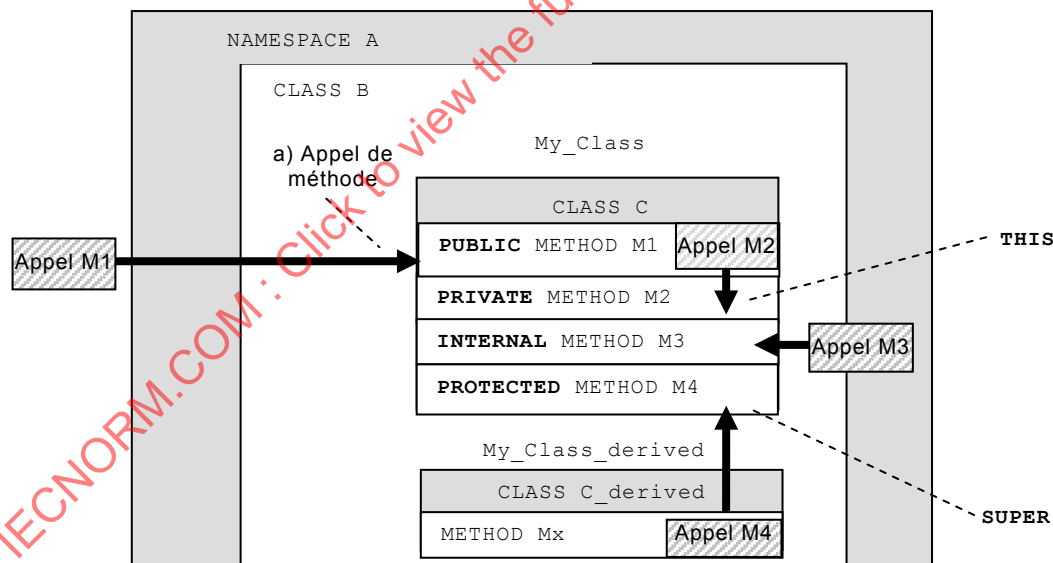
L'accès aux prototypes de méthode est implicitement toujours **PUBLIC**. Par conséquent, aucun spécificateur d'accès n'est utilisé sur les prototypes de méthode.

Toutes les utilisations incorrectes doivent être traitées comme des erreurs.

EXEMPLE Spécificateur d'accès pour les méthodes.

Illustration de l'accessibilité (appel) des méthodes définies dans la classe C:

- a) Spécificateurs d'accès: **PUBLIC**, **PRIVATE**, **INTERNAL**, **PROTECTED**
- **PUBLIC** M1 accessible par l'appel de M1 de l'intérieur de la classe B (également classe C)
 - **PRIVATE** M2 accessible par l'appel de M2 de l'intérieur de la classe C uniquement
 - **INTERNAL** M3 accessible par l'appel de M3 de l'intérieur de **NAMESPACE A** (également classe B, classe C)
 - **PROTECTED** M4 accessible par l'appel de M4 de l'intérieur de la classe **C_derived** (également classe C)
- b) Appels de méthode internes/externes:
- M2 est appelé de l'intérieur de la classe C – avec le mot-clé **THIS**.
 - M1, M3 et M4, de classe C, sont appelés de l'extérieur de la classe C – avec le mot-clé **SUPER** pour M4.



6.6.5.10 Spécificateurs d'accès aux variables (**PROTECTED**, **PUBLIC**, **PRIVATE**, **INTERNAL**)

Les emplacements à partir desquels l'accès aux variables de la section **VAR** est permis doivent être définis. L'accessibilité des variables est définie à l'aide d'un des spécificateurs d'accès ci-dessous, placé après le mot-clé **VAR**.

NOTE Les spécificateurs d'accès peuvent être combinés avec d'autres spécificateurs tels que **RETAIN** ou **CONSTANT** dans un ordre quelconque.

- **PROTECTED**

Si l'héritage est mis en œuvre, le spécificateur d'accès **PROTECTED** est applicable. Il indique que les variables associées sont accessibles uniquement de l'intérieur d'une classe

et de l'intérieur de toutes les classes dérivées. `PROTECTED` est défini par défaut et peut être omis.

Si l'héritage est mis en œuvre mais pas utilisé, `PROTECTED` a le même effet que `PRIVATE`.

- **PUBLIC**

Le spécificateur d'accès `PUBLIC` indique que les variables associées sont accessibles partout où la classe peut être utilisée.

- **PRIVATE**

Le spécificateur d'accès `PRIVATE` indique que les variables associées sont accessibles uniquement depuis l'intérieur de la classe elle-même.

Si l'héritage n'est pas mis en œuvre, `PRIVATE` est défini par défaut et peut être omis.

- **INTERNAL**

Si l'espace de noms est mis en œuvre, le spécificateur d'accès `INTERNAL` est applicable. Il indique que les variables associées sont accessibles uniquement depuis le `NAMESPACE` dans lequel la classe est déclarée.

Toutes les utilisations incorrectes doivent être traitées comme des erreurs.

6.6.6 Interface

6.6.6.1 Généralités

Dans la programmation orientée objet, le concept d'interface est introduit pour permettre la séparation de la spécification d'interface de sa mise en œuvre en tant que classe. Cela permet différentes mises en œuvre d'une spécification d'interface commune.

Une définition d'interface commence par le mot-clé `INTERFACE` suivi du nom d'interface et se termine par le mot-clé `END_INTERFACE` (voir Tableau 51).

L'interface peut contenir un ensemble de prototypes de méthode (implicitement publics).

6.6.6.2 Utilisation de l'interface

La spécification d'interface peut être utilisée de deux façons:

a) Dans une déclaration de classe.

Elle spécifie quelles méthodes la classe doit mettre en œuvre; par exemple, pour la réutilisation de la spécification d'interface comme indiqué à la Figure 18.

b) En tant que type de variable.

Les variables de type interface sont des références à des instances de classes et doivent être affectées avant utilisation. Les interfaces ne doivent pas être utilisées en tant que variables d'entrée-sortie.

Tableau 51 – Interface

N°	Description Mot-clé	Explication
1	<code>INTERFACE ...</code> <code>END_INTERFACE</code>	Définition d'interface
	Méthodes et spécificateurs	
2	<code>METHOD...END_METHOD</code>	Définition de méthode
	Héritage	
3	<code>EXTENDS</code>	L'interface hérite d'une interface