DEUTSCHE NORM ***Entwurf*** **Dezember 2009**

DIN IEC 61131-3

ICS 35.080

Einsprüche bis 2010-02-28
Vorgesehen als Ersatz für
DIN EN 61131-3:2003-12 und
DIN EN 61131-3 Beiblatt 1:2005-04

Entwurf

Speicherprogrammierbare Steuerungen – Teil 3: Programmiersprachen; Englische Fassung (IEC 65B/725/CD:2009)

Programmable Controllers –
Part 3: Programming languages;
English version (IEC 65B/725/CD:2009)

Anwendungswarnvermerk

Dieser Norm-Entwurf mit Erscheinungsdatum 2009-12-14 wird der Öffentlichkeit zur Prüfung und Stellungnahme vorgelegt.

Weil die beabsichtigte Norm von der vorliegenden Fassung abweichen kann, ist die Anwendung dieses Entwurfes besonders zu vereinbaren.

Stellungnahmen werden erbeten

– vorzugsweise als Datei per E-Mail an dke@din.de in Form einer Tabelle. Die Vorlage dieser Tabelle kann im Internet unter www.dke.de/stellungnahme abgerufen werden;

– oder in Papierform an die DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE (Hausanschrift: Stresemannallee 15, 60596 Frankfurt am Main).

Die Empfänger dieses Norm-Entwurfs werden gebeten, mit ihren Kommentaren jegliche relevante Patentrechte, die sie kennen, mitzuteilen und unterstützende Dokumentationen zur Verfügung zu stellen.

Gesamtumfang 185 Seiten

DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE

Preisgruppe 41
www.din.de
www.beuth.de
1564342

Nationales Vorwort

Die englische Originalfassung des internationalen Dokuments IEC 65B/725/CD:2009 „Programmable Controllers – Part 3: Programming languages“ (CD, en: Committee Draft) ist unverändert in diesen Norm-Entwurf übernommen worden.

Das internationale Dokument wurde vom SC 65B „Devices“ der Internationalen Elektrotechnischen Kommission (IEC) erarbeitet und den nationalen Komitees zur Stellungnahme vorgelegt.

Die IEC und das Europäische Komitee für Elektrotechnische Normung (CENELEC) haben vereinbart, dass ein auf IEC-Ebene erarbeiteter Entwurf für eine Internationale Norm zeitgleich (parallel) bei IEC und CENELEC zur Umfrage (CDV-Stadium) und Abstimmung als FDIS (en: Final Draft International Standard) bzw. Schluss-Entwurf für eine Europäische Norm gestellt wird, um eine Beschleunigung und Straffung der Normungsarbeit zu erreichen. Dokumente, die bei CENELEC als Europäische Norm angenommen und ratifiziert werden, sind unverändert als Deutsche Normen zu übernehmen.

Es ist vorgesehen, auch bei der entsprechenden zukünftigen Deutschen Norm auf die deutsche Sprachfassung zu verzichten und diese in der englischsprachigen Fassung zu veröffentlichen.

Da der Abstimmungszeitraum für einen FDIS bzw. Schluss-Entwurf prEN nur 2 Monate beträgt, und dann keine sachlichen Stellungnahmen mehr abgegeben werden können, sondern nur noch eine „JA/NEIN“-Entscheidung möglich ist, wobei eine „NEIN“-Entscheidung fundiert begründet werden muss, wird bereits der CD als DIN-Norm-Entwurf veröffentlicht, um die Stellungnahmen aus der Öffentlichkeit frühzeitig berücksichtigen zu können.

Für diesen vorliegenden Norm-Entwurf ist das nationale Arbeitsgremium K 962 „SPS“ der DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE (www.dke.de) zuständig.

Da sich die Benutzer des vorliegenden Norm-Entwurfs der englischen Sprache als Fachsprache bedienen, wird die Englische Fassung der EN 61131-3 veröffentlicht. Für die meisten der verwendeten Begriffe existieren keine gebräuchlichen deutschen Benennungen, da sich die deutschen Anwender in der Regel ebenfalls der englischen Benennungen bedienen. Dieser Norm-Entwurf steht nicht in unmittelbarem Zusammenhang mit Rechtsvorschriften und ist nicht als Sicherheitsnorm anzusehen.

Das Präsidium des DIN hat mit Präsidialbeschluss 1/2004 festgelegt, dass DIN-Normen, deren Inhalt sich auf internationale Arbeitsergebnisse der Informationsverarbeitung gründet, unter bestimmten Bedingungen allein in englischer Sprache veröffentlicht werden dürfen. Diese Bedingungen sind für die vorliegende Norm erfüllt.

Für den Fall einer undatierten Verweisung im normativen Text (Verweisung auf eine Norm ohne Angabe des Ausgabedatums und ohne Hinweis auf eine Abschnittsnummer, eine Tabelle, ein Bild usw.) bezieht sich die Verweisung auf die jeweils neueste gültige Ausgabe der in Bezug genommenen Norm.

Für den Fall einer datierten Verweisung im normativen Text bezieht sich die Verweisung immer auf die in Bezug genommene Ausgabe der Norm.

Der Zusammenhang der zitierten Normen mit den entsprechenden Deutschen Normen ergibt sich, soweit ein Zusammenhang besteht, grundsätzlich über die Nummer der entsprechenden IEC-Publikation. Beispiel: IEC 60068 ist als EN 60068 als Europäische Norm durch CENELEC übernommen und als DIN EN 60068 ins Deutsche Normenwerk aufgenommen.

Änderungen

Gegenüber DIN EN 61131-3:2003-12 und DIN EN 61131-3 Bbl 1:2005-04 wurden folgende Änderungen vorgenommen:

a) Einführung objektorientierter Techniken;

b) siehe Angaben auf Seite 2 und 3 des IEC-Schriftstücks.

Changes to 2nd Edition:

1. Error corrections and editorial changes (numbering of clauses, tables, figures): All done in the first **CD/65B/672.**

2. The following table gives an overview of the major changes of the 3rd Edition.

3. **All essential changes to 2nd Edition are in blue writing.**

[Editor’s note: - Notes like this marked yellow are temporarly for this 2CD only]

Overview of major changes:

[Editor’s note: This list is not complete; clause numbering is not yet correct.]

Clause (*link*)	**Title**	**Changes**
all	FB	**Clar**ify: Always distinguish instance or type
3	Terms and definitions	New: Some definitions
6.1.5	Comments	New: Single line comment. // text
6.2.1	Numeric literals …	New: **INT**#16#7FFF = decimal value 32767
6.3.3	Generic data type	Corr: Fig. 4 – Hierarchy of ..
6.3.4.2 et seq.	**Derived data type – Decl.** - Initialization - Usage	New: STRUCTURE, ARRAY with **- explicit layout of memory and endieness** using **AT** - Keyword **OVERLAP**. - **Packed** ARRAYS
6.4.2.3	Partial Access of ANY_BIT	e.g. <variable name> .X0 to <variable name> .X7
6.4.2.5	Variable-length array	New: ARRAY [* , * ,*] OF INT; Std FBs for upper und lower bound
6.4.4.1	Declaration – Type assignment	Del: Declaration of directly represented variables: ~~AT %IW6.2 : WORD;~~ see Table 20
0	Initial value assignment	New: Constant expression: 2*pi/2
6.5.2.1	Function - General	Clar: Results: VAR_EXTERNAL, VAR_IN_OUT New: Keyword VOID
6.5.2.5.2	Typed overloading	Corr: WORD_TO_INT vs. TO_INT
Et seq.	**Type conversion**	New: Table 28 – **Implicit** and explicit conversion
	Explicitly typed or overloaded .	New: Examples in tables
	Implicit type conversion	New/Clar: Example
6.5.2.6.2	Type conversion function	New: TRUNC vs. TRUNC_**
6.5.3.1	Function block - General	Clar: .. New: Error if no value specified for parameters in-out and function block instance
6.5.3.4	Function block - Declaration	New: Table 40#12 - Function block result and examples
6.5.3.5.2	Standard function blocks – Bistable elements	New: additionally long name SR SET1, RESET, … vs. S1, R, …
6.5.3.5.4	Counters	Del: long input names: LOAD, RESET
6.5.4	**Object Oriented extentions of FB concept**	Editors note: Hybrid FB (with and without OO) as option !
Et seq.	Methods – General, declaration	New: method = procedure (like a function) for a FB (type)
	Interfaces	New: Method prototypes, IMPLEMENTS, Representation, polymorphism, ABSTRACT, …
	Inheritance	New:
	THIS/SUPER	New:
6.6.5	SFC – Evaluation rules	Corr: Simult. seq. – divergence and convergence
6.8	**Namespaces**	New: NAMESPACE, INTERNAL ACCESS, PUBLIC ACCESS
8.2.6	Instruction List (IL)	New: FB call – Counter RESET (Long name)
7.3.2	Structured Text (ST)	New: Statements: “;“, CONTINUE (see also example)
8.2.4	Ladder Diagram (LD)	New: Contacts for Compare (typed and overloaded)
Annex B	**Formal spec** of language elements	New: various new elements:
Ex Annex F	Examples	Del: Move to **next** edition of part 8 - Guidelines
Ex Annex H	Interoperability with IEC 61499 devices	Deleted

CONTENTS

Figures

Tables

INTERNATIONAL ELECTROTECHNICAL COMMISSION

____

PROGRAMMABLE CONTROLLERS –

Part 3: Programming languages

IEC 61131-3, Ed. 3

FOREWORD

1) The IEC (International Electrotechnical Commission) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of the 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, the IEC publishes International Standards. 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. The 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 the 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 National Committees.

3) The documents produced have the form of recommendations for international use and are published in the form of standards, technical specifications, technical reports or guides and they are accepted by the National Committees in that sense.

4) In order to promote international unification, IEC National Committees undertake to apply IEC International Standards transparently to the maximum extent possible in their national and regional standards. Any divergence between the IEC Standard and the corresponding national or regional standard shall be clearly indicated in the latter.

5) The IEC provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with one of its standards.

6) Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights. The 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: Devices, of IEC technical committee 65: Industrial-process measurement and control.

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

FDIS	Report on voting
--	--

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

This International Standard has been reproduced without significant modification to its original contents or drafting.

The committee has decided that the contents of this publication will remain unchanged until 2013. At this date, the publication will be

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

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). These consist of two textual languages, IL (Instruction List) and ST (Structured Text), and two graphical languages, LD (Ladder Diagram) and FBD (Function Block Diagram).

Sequential Function Chart (SFC) elements are 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.

The programming language elements defined in this Part may be used in an interactive programming environment. The specification of such environments is beyond the scope of this standard; however, such an environment shall be capable of producing textual or graphic program documentation in the formats specified in this standard.

The material in this Part is arranged in “bottom-up” fashion, that is, simpler language elements are presented first, in order to minimize forward references in the text. The remainder of this subclause provides an overview of the material presented in this part and incorporates some general requirements.

This Part of IEC 61131-3 does not include any provisions for the harmonization with IEC 61499 - Function Blocks.

2 Normative references

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

IEC 60050 (all parts): *International Electrotechnical Vocabulary (IEV)*

IEC 60559:1989, *Binary floating-point arithmetic for microprocessors systems*

IEC 60617-12:1997, *Graphical symbols for diagrams – Part 12: Binary logic elements*

IEC 60617-13:1993, *Graphical symbols for diagrams – Part 13: Analogue elements*

IEC 60848:2002, *GRAFCET specification language for sequential function charts*

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

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

ISO/AFNOR: 1989, *Dictionary of computer science – The standardised vocabulary*

ISO/IEC 10646-1:1993, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane*

3 Terms and definitions

For the purposes of this part of IEC 61131, the following definitions apply. Definitions applying to all parts of IEC 61131 are given in Part 1.

NOTE 1 Terms defined in this subclause are *italicized* where they appear in the bodies of definitions.

NOTE 2 The notation "(ISO)" following a definition indicates that the definition is taken from the ISO/AFNOR Dictionary of computer science.

NOTE 3 The ISO/AFNOR Dictionary of computer science and the IEC 60050 should be consulted for terms not defined in this standard.

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, as specified in 6.6.4.1

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 as defined in 6.6.4.7

3.5
aggregate
structured collection of data objects forming a *data type*. (ISO)

3.6
argument
synonymous with *input variable*, *output variable* or *in-out variable*

3.7
array
aggregate that consists of data objects, with identical attributes, each of which may be uniquely referenced by *subscripting*. (ISO)

3.8
assignment
mechanism to give a value to a variable or to an *aggregate*. (ISO)

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
body
that portion of a *program organization unit* which specifies the operations to be performed on the declared *operands* of the program organization unit when its execution is calle*d*

3.14
character string
aggregate that consists of an ordered sequence of characters

3.15
comment
language construct for the inclusion of text in a program and having no impact on the execution of the program. (ISO)

3.16
compile
to translate a *program organization unit* or *data type* specification into its machine language equivalent or an intermediate form

3.17
configuration
language element corresponding to a *programmable controller system* as defined in IEC 61131-1

3.18
constant
language element which declares a data element with a fixed value

3.19
counter function block
function block which accumulates a value for the number of changes sensed at one or more specified *inputs*

3.20
data type
set of values together with a set of permitted operations. (ISO)

3.21
date and time
date within the year and the time of day represented as a single language element

3.22
declaration
mechanism for establishing the definition of a *language element*

NOTE A declaration normally involves attaching an identifier to the language element, and allocating attributes such as data types and algorithms to it.

3.23
delimiter
character or combination of characters used to separate program *language elements*

3.24
derived function block type
function block type created by *inheritance* from another function block type

3.25

direct representation

means of representing a variable in a programmable controller program from which a manufacturer-specified correspondence to a physical or *logical location* may be determined directly

3.26

double word

data element containing 32 bits

3.27

dynamic binding

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

3.28

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

3.29

execution control element

language element which controls the flow of program execution

3.30

falling edge

change from 1 to 0 of a Boolean variable

3.31

function (procedure)

program organization unit which, when executed, yields exactly one data element and possibly additional *output variables* (which may be multi-valued, for example, an *array* or *structure*), and whose call can be used in textual languages as an *operand* in an expression

3.32

function block instance (function block)

instance of a *function block type*

3.33

function block type

programmable controller programming *language element* consisting of:
1) the definition of a data structure partitioned into *input, output*, and *internal variables*; and
2 a) either a set of operations to be performed upon the elements of the data structure when an *instance* of the function block type is called, or
2b) a set of *methods*.

3.34

function block diagram

network in which the *nodes* are *function block instances*, graphically represented *functions* (procedures), *variables*, *literals*, and *labels*

3.35

generic data type

data type which represents more than one type of data, as specified in 6.3.2

3.36

global scope

scope of a declaration applying to all *program organization units* within a *resource* or *configuration*

3.37
global variable
variable whose *scope* is *global*

3.38
hierarchical addressing
direct representation of a data element as a member of a physical or logical hierarchy, for example, a point within a module which is contained in a rack, which in turn is contained in a cubicle, etc

3.39
identifier
combination of letters, numbers, and underline characters, as specified in 6.1.2, which begins with a letter or underline and which names a *language element*

3.40
in-out variable
variable which is used to supply an argument to a *program organization unit* and which is additionally used to return the result(s) of the *evaluation* of a *program organization unit*

3.41
initial value
value assigned to a variable at system start-up

3.42
inheritance
creation of a new function block type by using an existing function block type

NOTE The new function block type contains the same variables and methods than the existing function block type, unless a method is overridden by the new function block type. The new function block type may contain additional variables and methods.

3.43
input variable (input)
variable which is used to supply an argument to a *program organization unit*

3.44
instance
individual, named copy of the data structure associated with a *function block type* or *program type*, which persists from one call of the associated operations to the next

3.45
instance name
identifier associated with a specific *instance*

3.46
instantiation
the creation of an *instance*

3.47
integer literal
literal which directly represents a value of type SINT, INT, DINT, LINT, BOOL, BYTE, WORD, DWORD, or LWORD, as defined in 6.3.2

3.48
interface
language element containing a set of VAR CONSTANT-declarations and a set of *method prototypes*.

3.49
keyword
lexical unit that characterizes a *language element*, for example, “IF”

3.50
label
language construction naming an instruction, network, or group of networks, and including an *identifier*

3.51
language element
any item identified by a symbol on the left-hand side of a production rule in the formal specification given in annex B of this standard

3.52
literal
lexical unit that directly represents a value. (ISO)

3.53
local scope
the *scope* of a *declaration* or *label* applying only to the *program organization unit* in which the declaration or label appears

3.54
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

3.55
long real
real number represented in a *long word*

3.56
long word
64-bit data element.

3.57
memory (user data storage)
functional unit to which the user program can store data and from which it can retrieve the stored data

3.58
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 the variables of the function block type

3.59
method prototype
language element containing only the external interface of a method

NOTE i.e.: a method prototype contains VAR_INPUT, VAR_OUTPUT and VAR_IN_OUT variables and the return value but no local variables and no operations.

3.60
named element
element of a *structure* which is named by its associated *identifier*

3.61
network
arrangement of nodes and interconnecting branches

3.62
off-delay (on-delay) timer function block
function block which delays the *falling (rising) edge* of a Boolean *input* by a specified duration

3.63
operation
lanquage element that performs an elementary piece of a program organisation unit or method

NOTE An operation is represented in Instruction List (IL) as instruction, in Structured Text (ST) as statement, in Ladder Diagram (LD) and Function Block Diagram (FBD) as graphical symbol like contact.

3.64
operand
language element on which an operation is performed

3.65
operator
symbol that represents the action to be performed in an operation

3.66
override
method of a derived function block type with the same name, the same variables (VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT) and the same return value as a method of the base (parent) function block type

3.67
output variable (output)
variable which is used to return the result(s) of the *evaluation* of a *program organization unit*

3.68
overloaded
with respect to an operation or *function*, capable of operating on data of different types, as specified in 6.5.2.5

3.69
power flow
symbolic flow of electrical power in a ladder diagram, used to denote the progression of a logic solving algorithm

3.70
pragma
language construct for the inclusion of text in a program organization unit which may affect the preparation of the program for execution

3.71
program (verb)
to design, write, and test user programs

3.72
program organization unit (POU)
function, function block, or program

NOTE This term may refer to either a type or an instance.

3.73
real literal
literal representing data of type REAL or LREAL.

3.74
resource
language element corresponding to a “signal processing function” and its “man-machine interface” and “sensor and actuator interface functions”, if any, as defined in IEC 61131-1

3.75
retentive data
data stored in such a way that its value remains unchanged after a power down / power up sequence

3.76
return
language construction within a *program organization unit* designating an end to the execution sequences in the unit

3.77
rising edge
change from 0 to 1 of a Boolean variable

3.78
scope
that portion of a *language element* within which a *declaration* or *label* applies

3.79
semantics
relationships between the symbolic elements of a programming language and their meanings, interpretation and use.

3.80
semigraphic representation
representation of graphic information by the use of a limited set of characters

3.81
single data element
data element consisting of a single value

3.82
single-element variable
variable which represents a single data element

3.83
step
situation in which the behaviour 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

3.84
structured data type
aggregate data type which has been declared using a `STRUCT` or `FUNCTION_BLOCK` declaration

3.85
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

3.86
symbolic representation
use of *identifiers* to name variables

3.87
task
execution control element providing for periodic or triggered execution of a group of associated *program organization units*

3.88
time literal
literal representing data of type `TIME`, `DATE`, `TIME_OF_DAY`, or `DATE_AND_TIME`

3.89
transition
condition whereby control passes from one or more predecessor *steps* to one or more successor steps along a directed link

3.90
unsigned integer
integer literal not containing a leading plus (+) or minus (-) sign

3.91
variable
software entity that may take different values, one at a time
[ISO 2382]

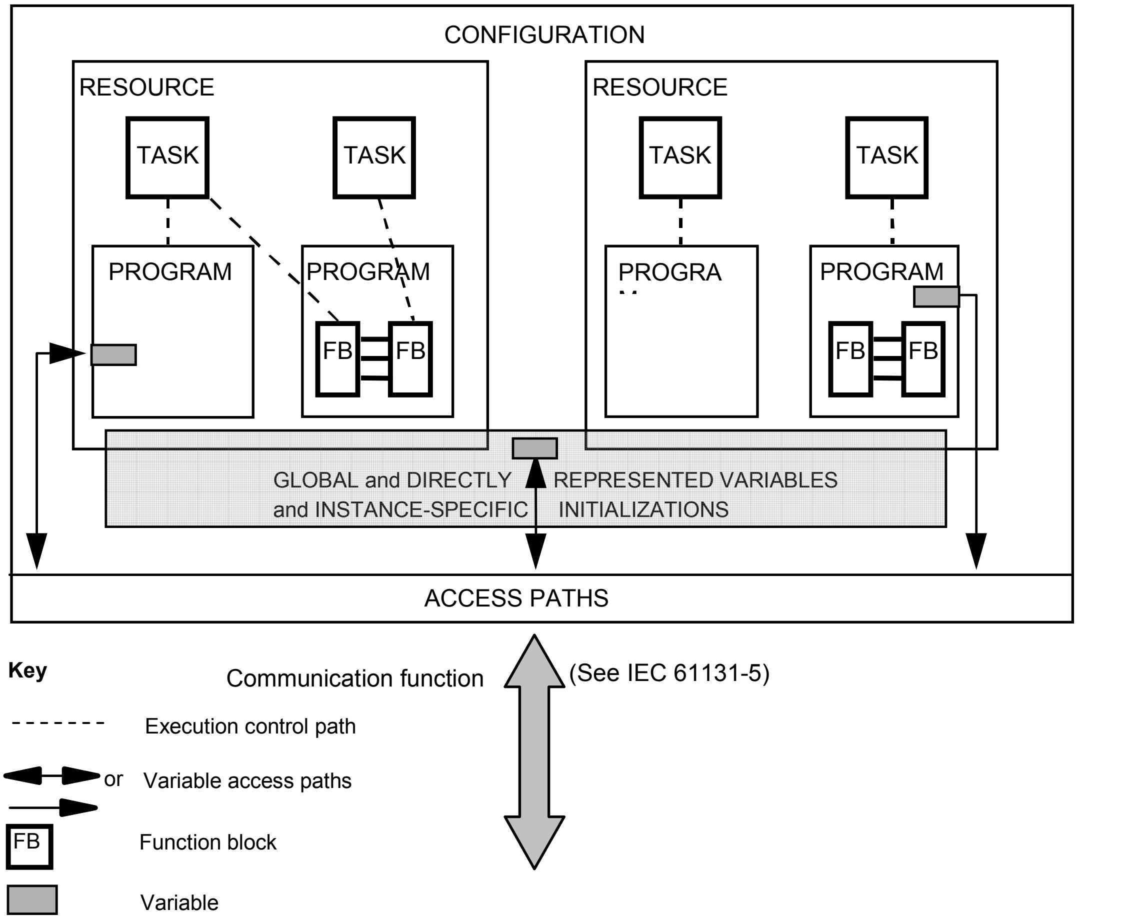
NOTE The values of a variable are usually restricted to a certain *data type*.

3.92
wired `OR`
construction for achieving the Boolean `OR` function in the LD language by connecting together the right ends of horizontal connectives with vertical connectives

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*; 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.



IEC 2468/02

NOTE 1 This figure 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.

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* according to the rules given in 6.4.3, 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 6.7.3, 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 6.7.2.

The mapping of the language elements defined in this subclause onto communication objects is defined in IEC 61131-5.

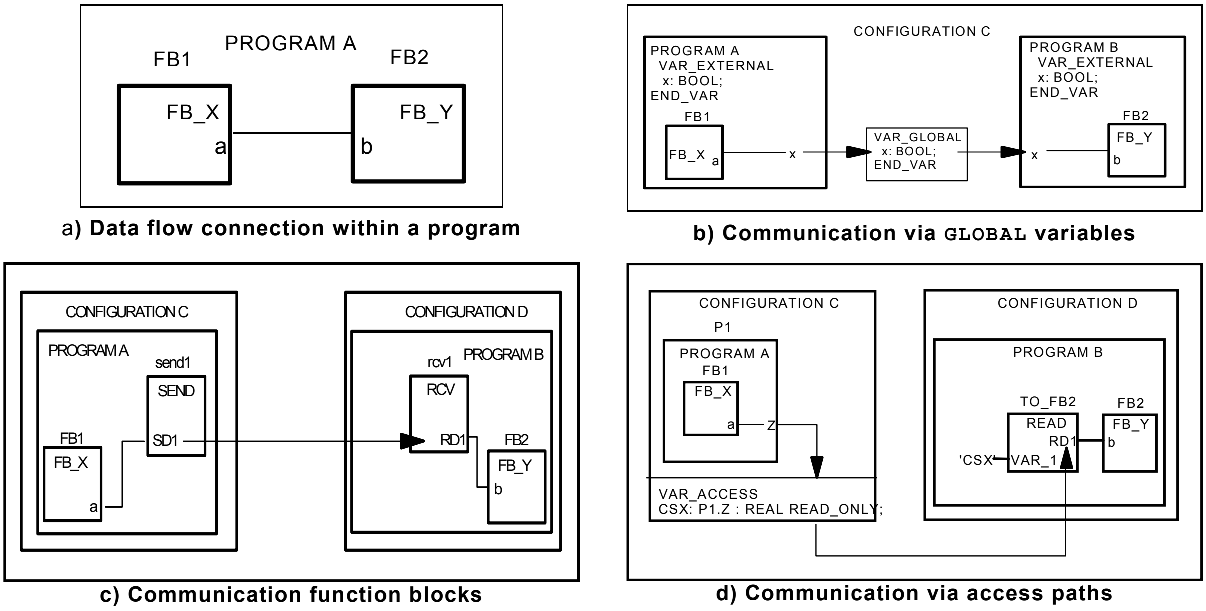
4.2 Communication model

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

As shown in Figure 2 a), 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 2 b). These variables shall be declared as `GLOBAL` in the configuration, and as EXTERNAL in the programs, as specified in 6.4.4.

As illustrated in Figure 2 c), 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 and described in 6.5.3.5.6. In addition, programmable controllers or non-programmable controller systems can transfer data which is made available by *access paths,* as illustrated in Figure 2 d), using the mechanisms defined in IEC 61131-5.



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

b) **Communication via `GLOBAL` variables**

c) **Communication function blocks**

d) **Communication via access paths**

NOTE 1 This figure 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 this figure. See 6.5.3.5.6 and IEC 61131-5.

NOTE 4 As specified in 6.7, *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

The elements of programmable controller programming languages, and the subclauses in which they appear in this Part, are classified as follows:

Data types Variables (6.4)

Program organization units (6.5)
    Functions (6.5.2)
    Function blocks (6.5.3)
        Methods (6.5.4.2)
        Interfaces (6.5.4.3
    Programs (6.5.5)
Sequential Function Chart (SFC) elements (4.3)
Configuration elements (6.7)
    Global variables (6.7.2)
    Resources (6.7.2)
    Access paths (6.7.2)
    Tasks (6.7.3)

[Editor’s note: Methods and Interfaces are new elements. TBD.]

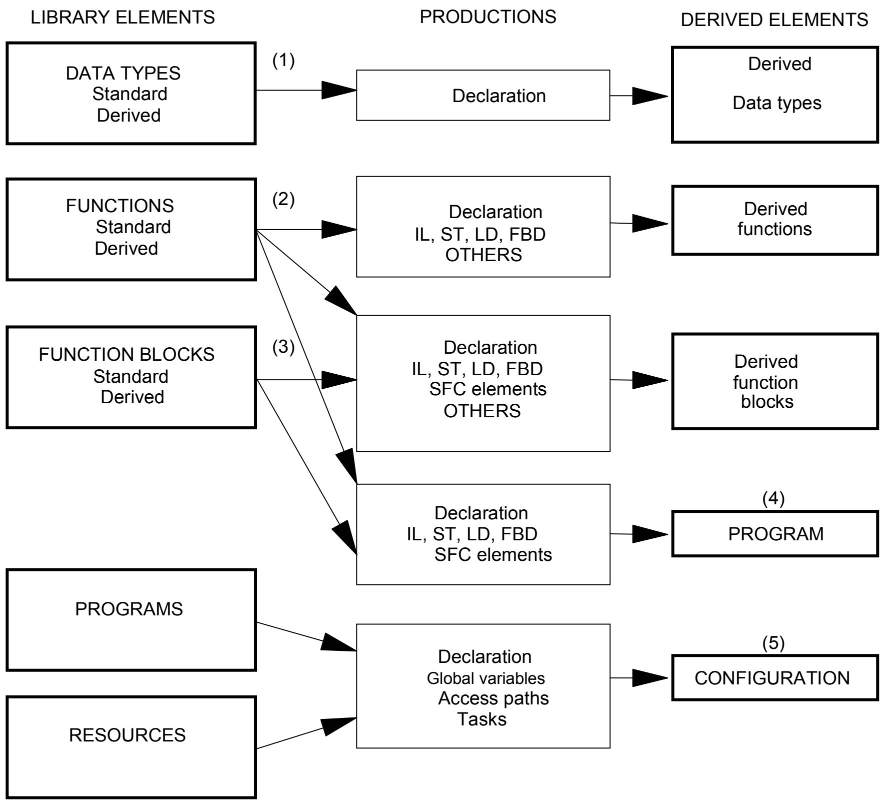
As shown in Figure 3, the combination of these elements shall obey the following rules:

1. Derived data types shall be declared as specified in 6.3.4, using the standard data types specified in 6.3.2 and 6.3.3 and any previously derived data types.
2. Derived *functions* can be declared as specified in 6.5.2, using standard or derived data types, the standard functions defined in 6.5.2.6, and any previously derived functions. This declaration shall use the mechanisms defined for the IL, ST, LD or FBD language.
3. Derived function block types can be declared as specified in 6.5.3.5, using standard or derived data types and functions, the standard function block types defined in 6.5.3.5, and any previously derived function block types. This declaration shall use the mechanisms defined for the IL, ST, LD, or FBD language, and can include Sequential Function Chart (SFC) elements as defined in 6.6.
4. A program shall be declared using standard or derived data types, functions, and function blocks. This declaration shall use the mechanisms defined for the IL, ST, LD, or FBD language, and can include Sequential Function Chart (SFC) elements as defined in 6.6.
5. Programs can be combined into configurations using the elements defined in 6.7.2, that is, global variables, resources, tasks, and access paths.

Reference to “previously derived” data types, functions, and function blocks in the above rules is intended to imply that once such a derived element has been declared, its definition is available, for example, in a “library” of derived elements, for use in further derivations. Therefore, the declaration of a derived element type shall not be contained within the declaration of another derived element type.

A programming language other than one of those defined in this standard may be used in the declaration of a *function* or *function block type*. The means by which a user program written in one of the languages defined in this standard calls the execution of, and accesses the data associated with, such a derived function or an instance of such a derived function block type shall be as defined in this standard.

LD - Ladder Diagram (8.2), FBD - Function Block Diagram (8.3),IL - Instruction List (7.2), ST - Structured Text (7.3), OTHERS - Other programming languages.


LIBRARY ELEMENTS
PRODUCTIONS
DERIVED ELEMENTS
DATA TYPES Standard Derived
(1)
Declaration
Derived Data types
FUNCTIONS Standard Derived
(2)
Declaration IL, ST, LD, FBD OTHERS
Derived functions
FUNCTION BLOCKS Standard Derived
(3)
Declaration IL, ST, LD, FBD SFC elements OTHERS
Derived function blocks
Declaration IL, ST, LD, FBD SFC elements
(4)
PROGRAM
PROGRAMS
RESOURCES
Declaration Global variables Access paths Tasks
(5)
CONFIGURATION

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 System compliance

A programmable controller system, as defined in IEC 61131-1, which claims to comply, wholly or partially, with the requirements of this Part of IEC 61131 shall do so only as described below.

A compliance statement shall be included in the documentation accompanying the system, or shall be produced by the system itself. The form of the compliance statement shall be:

> “This system complies with the requirements of IEC 61131-3, for the following language features:”,

followed by a set of compliance tables in the following format:

Table title

Table No.	Feature No.	Features description
…	…	…

Table and feature numbers and descriptions are to be taken from the tables given in the relevant subclauses of this part of IEC 61131. Table titles are to be taken from the following table.

Table title	For features in:
Common elements	Clause 6

Common textual elements	Subclause 7.1
IL language elements	Subclause 7.2
ST language elements	Subclause 7.3
Common graphical elements	Subclause 8.1
LD language elements	Subclause 8.2
FBD language elements	Subclause 8.3

For the purposes of determining compliance, some tables shall not be considered tables of features.

A programmable controller system complying with the requirements of this standard with respect to a language defined in this standard:

a) shall not require the inclusion of substitute or additional language elements in order to accomplish any of the features specified in this standard, **unless** such elements are identified and treated as noted in rules e) and f) below;

b) shall be accompanied by a document that specifies the values of all implementation dependencies as listed in Annex D;

c) shall be able to determine whether or not a user's language element violates any requirement of this standard, where such a violation is not designated as an error in annex E, and report the result of this determination to the user. In the case where the system does not examine the whole program organization unit, the user shall be notified that the determination is incomplete whenever no violations have been detected in the portion of the program organization unit examined;

d) shall treat each user violation that is designated as an **error** in Annex E in at least one of the following ways:

   1) there shall be a statement in an accompanying document that the error is not reported;

   2) the system shall report during preparation of the program for execution that an occurrence of that error is possible;

   3) the system shall report the error during preparation of the program for execution;

   4) the system shall report the error during execution of the program and initiate appropriate system- or user-defined error handling procedures;

   and if any violations that are designated as errors are treated in the manner described in d)1) above, then a note referencing each such treatment shall appear in a separate section of the accompanying document;

e) shall be accompanied by a document that separately describes any features accepted by the system that are prohibited or not specified in this standard. Such features shall be described as being "**extensions to the <language> language as defined in IEC 61131-3**";

f) shall be able to process in a manner similar to that specified for errors any use of any such **extension**;

g) shall be able to process in a manner similar to that specified for errors any use of one of implementation dependencies specified in Annex D;

h) shall not use any of the standard data type, function or function block type names defined in this standard for manufacturer-defined features whose functionality differs from that described in this standard, unless such features are identified and treated as noted in rules e) and f) above;

i) shall be accompanied by a document defining, in the form specified in Annex A, the format of all textual language elements supported by the system;

j) shall be capable of reading and writing files containing any of the language elements defined as alternatives in the production library_element_declaration in Annex B, in the syntax defined in requirement i) above, encoded according to the “ISO-646 IRV” given as table 1 - Row 00 of ISO/IEC 10646-1;

k) shall be accompanied by a document describing the processing by the system of errors, extensions and implementation dependencies as defined in items c) through f) above.

The phrase “be able to” is used in this subclause to permit the implementation of a software switch with which the user may control the reporting of errors.

In cases where compilation or program entry is aborted due to some limitation of tables, etc., an incomplete determination of the kind “no violations were detected, but the examination is incomplete” will satisfy the requirements of this subclause.

5.2 Program compliance

A programmable controller program complying with the requirements of IEC 61131-3:

a) shall use only those features specified in this standard for the particular language used;

b) shall not use any features identified as extensions to the language;

c) shall not rely on any particular interpretation of **implementation dependencies**.

The results produced by a complying program shall be the same when processed by any complying system which supports the features used by the program, such results are influenced by program execution timing, the use of **implementation dependencies** (as listed in Annex D) in the program, and the execution of error handling procedures.

6 Common elements

6.1 Use of printed characters

6.1.1 Character set

Textual languages and textual elements of graphic languages shall be represented in terms of the “ISO-646 IRV” given as table 1 - Row 00 of ISO/IEC 10646-1.

The use of characters from additional character sets, for example, the “Latin-1 Supplement” given as table 2 - Row 00 of ISO/IEC 10646-1, is a typical extension of this standard. The encoding of such characters shall be consistent with ISO/IEC 10646-1.

The **required character set** consists of all the characters in columns 002 through 007 of the “ISO-646 IRV” as defined above, except for lower-case letters

Table 1 - Character set features

No.	Description
1	Lower case characters[a]
2a	Number sign (`#`) OR
2b	Pound sign (`£`)
3a	Dollar sign (`$`) OR
3b	Currency sign (`¤`)
4a	Vertical bar (`|`) OR
4b	Exclamation mark (!)

[a] When lower-case letters (#1) 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 6.2.2, and variables of type STRING and WSTRING as defined in 6.3.

6.1.2 Identifiers

An *identifier* is a string of letters, digits, and underline characters which shall begin with a letter or underline character.

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

Underlines 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 underlines 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 **implementation dependency**.

Identifier features and examples are shown in Table 2.

Table 2 - Identifier features

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

6.1.3 Keywords

Keywords are unique combinations of characters utilized as individual syntactic elements as defined in Annex B. All keywords used in this standard are listed in Annex C. Keywords shall not contain imbedded spaces. The case of characters shall not be significant in keywords; for instance, the keywords FOR and for are syntactically equivalent. The keywords listed in Annex C shall not be used for any other purpose, for example, variable names or extensions as defined in 6.4.

NOTE National standards organizations can publish tables of translations of the keywords given in annex C.

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 as described in 6.4, or delimiter combinations (for example, for comments as defined in 6.1.5. “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 10646-1.

6.1.5 Comments

There are two different kinds of user comments.

Multi-line comments shall be delimited at the beginning and end by the special character combinations (* and *), respectively, as shown in feature 1 of Table 3.

Single line comments start with the character combination // and end at the next following new line as shown in Table 3 feature 2.

Comments shall be permitted anywhere in the program where spaces are allowed, except within character string literals as defined in 6.2.2. Comments shall have no syntactic or semantic significance in any of the languages defined in this standard.

Nested comments use corresponding pairs of (*, *), e.g. (* (* NESTED *) *).

In single-line comments the special character combinations (* and *) have no special meaning, and in multi-line comments the special character combination // has no special meaning.

Table 3 - Comment feature

No.	Feature description	Example

1	Multi-line Comments	`(****************************`
		`       A framed comment`
		`****************************)`

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

3	Nested comment	`(* (* NESTED *) *)`

6.1.6 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 **implementation dependencies**. Pragmas shall be permitted anywhere in the program where spaces are allowed, except within character string literals as defined in 6.2.2.

NOTE Curly brackets inside a *comment* have no semantic meaning; comments inside curly brackets may or may not have semantic meaning depending on the implementation.

Table 4 - Pragma feature

No.	Feature description	Examples

1	Pragmas	`{VERSION 3.1}`
		`{AUTHOR JHC}`
		`{x := 256, y := 384}`

6.2 External representation of data

6.2.1 Numeric literals and bit string literals

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

There are two classes 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 underline characters (_) inserted between the digits of a numeric literal shall not be significant. No other use of underline 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 -).

Integer 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 positive integers.

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.	Feature description	Examples
1	Integer literals	`-12   0   123_456   +986`
2	Real literals	`-12.0   0.0   0.4560   3.14159_26`

3	Real literals with exponents	`-1.34E-12 or -1.34e-12`
		`1.0E+6 or 1.0e+6`
		`1.234E6 or 1.234e6`

4	Base 2 literals	`2#1111_1111` (255 decimal) `2#1110_0000` (224 decimal)
5	Base 8 literals	`8#377` (255 decimal) `8#340` (224 decimal)
6	Base 16 literals	`16#FF or 16#ff` (255 decimal) `16#E0 or 16#e0` (224 decimal)
7	Boolean zero and one	`0   1`
8	Boolean `FALSE` and `TRUE`	`FALSE   TRUE`
9	Typed literals	`INT#16#7FFF` (INT representation of the decimal value 32767) `INT#16#FFFF` (not allowed representation of the decimal value -1) `WORD#16#AFFE` (WORD representation of the hexadecimal value AFFE) `WORD#1234` (WORD representation of the decimal value 1234 = 16#4D2) `UINT#16#89AF` (UINT representation of the hexadecimal value `89AF`) `BOOL#0   BOOL#1   BOOL#FALSE   BOOL#TRUE`

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

6.2.2 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 from Row 00 of the ISO/IEC 10646-1 character set 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 10646-1 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.

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 literal features

No.	Example	Explanation
1	Single-byte characters or character strings	
	`''`	Empty string (length zero)
	`'A'`	String of length one or character CHAR containing the single character A
	`' '`	String of length one or character CHAR containing the “space” character
	`'$''`	String of length one or character CHAR containing the “single quote” character
	`'"'`	String of length one or character CHAR containing the “double quote” character
	`'$R$L'`	String of length two containing CR and LF characters
	`'$0A'`	String of length one or character CHAR containing the LF character
	`'$$1.00'`	String of length five which would print as “$1.00”
	`'ÄË'` `'$C4$CB'`	Equivalent strings of length two
2	Double-byte characters or character strings	
	`""`	Empty string (length zero)
	`"A"`	String of length one or character WCHAR containing the single character A
	`" "`	String of length one or character WCHAR containing the “space” character
	`"'"`	String of length one or character WCHAR containing the “single quote” character
	`"$""`	String of length one or character WCHAR containing the “double quote” character
	`"$R$L"`	String of length two containing CR and LF characters
	`"$$1.00"`	String of length five which would print as “$1.00”
	`"ÄË"` `"$00C4$00CB"`	Equivalent strings of length two
3	Single-byte typed characters or string literals	
	`STRING#'OK'` `CHAR#’X’`	String of length two containing two single-byte characters Character CHAR containing the single-byte character X
4	Double-byte typed string literals	
	`WSTRING#”OK”` `WCHAR#”X”`	String of length two containing two double-byte characters Character WCHAR containing the single-byte character X

NOTE If a particular implementation supports feature 4 but not feature 2, the implementor may specify **implementation-dependent** syntax and semantics for the use of the double-quote character.

Table 7 - Two-character combinations in character strings

No.	Combination	Interpretation when printed
1	`$$`	Dollar sign
2	`$'`	Single quote
3	`$L or $l`	Line feed
4	`$N or $n`	Newline
5	`$P or $p`	Form feed (page)
6	`$R or $r`	Carriage return
7	`$T or $t`	Tab
8	`$"`	Double quote

NOTE 1 The “newline” character provides an implementation-independent means of defining the end of a line of data for both physical and file I/O; 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.

6.2.3 Time literals

6.2.3.1 General

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.

Duration and time of day literals shall be delimited on the left by the keywords defined in 6.2.3.

6.2.3.2 Duration

Duration data shall be delimited on the left by the keyword `T#` or `TIME#`. The representation of duration data in terms of days, hours, minutes, seconds, and milliseconds, 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 underline 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 lowercase letters.

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

Table 8 - Duration literal features

No.	Feature description	Examples
	Duration literals without underlines	
1a	short prefix	`T#14ms` `T#-14ms` `T#14.7s` `T#14.7m` `T#14.7h` `t#14.7d` `t#25h15m` `t#5d14h12m18s3.5ms`
1b	long prefix	`TIME#14ms` `TIME#-14ms` `time#14.7s`
	Duration literals with underlines	
2a	short prefix	`t#25h_15m` `t#5d_14h_12m_18s_3.5ms`
2b	long prefix	`TIME#25h_15m` `time#5d_14h_12m_18s_3.5ms`

6.2.3.3 Time of day and date

Prefix keywords for time of day and date literals shall be as shown in Table 9. As illustrated in Table 10, representation of time-of-day and date information shall be as specified by the syntax given in Annex B.

Table 9 - Date and time of day literals

No.	Feature description	Prefix Keyword
1	Date literals (long prefix)	`DATE#`
2	Date literals (short prefix)	`D#`
3	Time of day literals (long prefix)	`TIME_OF_DAY#`

4	Time of day literals (short prefix)	`TOD#`
5	Date and time literals (long prefix)	`DATE_AND_TIME#`
6	Date and time literals (short prefix)	`DT#`

Table 10 - Examples of date and time of day literals

Long prefix notation	Short prefix notation
`DATE#1984-06-25` `date#1984-06-25`	`D#1984-06-25` `d#1984-06-25`
`TIME_OF_DAY#15:36:55.36` `time_of_day#15:36:55.36`	`TOD#15:36:55.36` `tod#15:36:55.36`
`DATE_AND_TIME#1984-06-25-15:36:55.36` `date_and_time#1984-06-25-15:36:55.36`	`DT#1984-06-25-15:36:55.36` `dt#1984-06-25-15:36:55.36`

6.3 Data types

6.3.1 General

A number of elementary (pre-defined) data types are recognized by this standard. Additionally, generic data types are defined for use in the definition of overloaded functions. A mechanism for the user or manufacturer to specify additional data types is also defined.

6.3.2 Elementary data types

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 11.

Table 11 - Elementary data types

No.	Keyword	Data type	N [a]
1	`BOOL`	Boolean	1 [h]
2	`SINT`	Short integer	8 [c]
3	`INT`	Integer	16 [c]
4	`DINT`	Double integer	32 [c]
5	`LINT`	Long integer	64 [c]
6	`USINT`	Unsigned short integer	8 [d]
7	`UINT`	Unsigned integer	16 [d]
8	`UDINT`	Unsigned double integer	32 [d]
9	`ULINT`	Unsigned long integer	64 [d]
10	`REAL`	Real numbers	32 [e]
11	`LREAL`	Long reals	64 [f]
12	`TIME`	Duration	-- [b]
13	`DATE`	Date (only)	-- [b]
14	`TIME_OF_DAY or TOD`	Time of day (only)	-- [b]
15	`DATE_AND_TIME or DT`	Date and time of Day	-- [b]
16	`STRING`	Variable-length single-byte character string	8 [i,g]
16a	`CHAR`	Single-byte character	8[g]
17	`BYTE`	Bit string of length 8	8 [j,g]

18	WORD	Bit string of length 16	16[j,g]
19	DWORD	Bit string of length 32	32[j,g]
20	LWORD	Bit string of length 64	64[j,g]
21	WSTRING	Variable-length double-byte character string	16[i,g]
21a	WCHAR	Double-byte character	16[g]

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

[b] The range of values and precision of representation in these data types is **implementation-dependent**.

[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.

[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.

[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.

6.3.3 Generic data types

In addition to the data types shown in Table 11, the hierarchy of generic data types shown in Figure 4 can be used in the specification of inputs and outputs of standard functions and function blocks (see 6.5.2.5). Generic data types are identified by the prefix “ANY”. The use of generic data types is subject to the following rules:

a) Generic data types shall not be used in user-declared program organization units.

b) The generic type of a *subrange* derived type (Table 12, feature 3) shall be ANY_INT.

c) The generic type of a *directly derived* type (Table 12, feature 1) shall be the same as the generic type of the elementary type from which it is derived.

d) The generic type of all other derived types defined in Table 12 shall be ANY_DERIVED.

```
ANY
    ANY_DERIVED
    ANY_ELEMENTARY
                  ANY_MAGNITUDE
                               ANY_NUM
                                      ANY_REAL
                                              REAL, LREAL
                                      ANY_INT
                                              SINT, INT, DINT, LINT,
                                              USINT, UINT, UDINT, ULINT
                               TIME
                  ANY_BIT
                               BOOL, BYTE, WORD, DWORD, LWORD
                  ANY_STRING
                               STRING, WSTRING,
                               CHAR, WCHAR
```

```
        ANY_DATE
                      DATE_AND_TIME, DATE, TIME_OF_DAY
```

Figure 4 - Hierarchy of generic data types

6.3.4 Derived data types

6.3.4.1 General

This subclause defines the requirements for the declaration, initialization and usage of derived (i.e., user- or manufacturer-specified) data types.

6.3.4.2 Declaration

Derived data types can be declared using the `TYPE...END_TYPE` textual construction shown in Table 12. These derived data types can then be used, in addition to the elementary data types defined in 6.3.2, in variable declarations as defined in 6.4.4.

- **Enumerated Data Type**

  An *enumerated* data type declaration specifies that the value of any data element of that type can only take on one of the values given in the associated list of identifiers, as illustrated in Table 12. The enumeration list defines an ordered set of enumerated values, starting with the first identifier of the list, and ending with the last. Different enumerated data types may use the same identifiers for enumerated values. The maximum allowed number of enumerated values is an **implementation dependency.**

  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 '#' sign, similar to typed literals defined in 6.2. Such a prefix shall not be used inside an enumeration list. It is an **error** if sufficient information is not provided in an enumerated literal to determine its value unambiguously.

- **Subrange**

  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 12. It is an **error** if the value of a subrange type falls outside the specified range of values.

- **Array**

  An ARRAY declaration 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). Thus, any element of type ANALOG_16_INPUT_CONFIGURATION as shown in Table 12 contains (among other elements) sufficient storage for 16 CHANNEL elements of type ANALOG_CHANNEL_CONFIGURATION. Mechanisms for access to array elements are defined in 6.4.1. The maximum number of array subscripts, maximum array size and maximum range of subscript values are **implementation dependencies**.

- **Structure**

  A STRUCT declaration specifies that data elements of that type shall contain sub-elements of specified types which can be accessed by the specified names. For instance, an element of data type ANALOG_CHANNEL_CONFIGURATION as declared in Table 12 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 **implementation dependencies**.

  Structures with explicit layout (keyword LAYOUT_EXPLICIT) shall define explicitly the memory layout and endianess of its components using the AT keyword. The offset value given in the AT clause specifies the byte or word offset of the begin of the memory location of this component relative to the begin of the memory location of the structure.

For components of the elementary data type BOOL the AT clause shall specify additionally a bit offset. The subrange allowed for bit offsets is 0..7 for using a byte offset and 0..15 for using a word offset.

It is allowed that the specified memory locations overlap arbitrarily if the keyword OVERLAP is given. Otherwise overlapping is not allowed.

Arrays contained in a structure with explicit layout shall be packed i.e. they shall not include unused memory locations.

NOTE Arrays of elements of data type BOOL may end at a memory location with a bit offset unequal 0.

Structures contained in a structure with explicit layout shall also have explicit layout.

Structures with explicit layout shall specify the endianess of the elementary data types using the keywords BIG_ENDIAN or LITTLE_ENDIAN. The endianess specifies order of the memory location of the bytes of an elementary data type.

If the keyword BIG_ENDIAN is given the data values are placed in the memory locations beginning with the highest value byte first and the lowest value byte last. If the keyword LITTLE_ENDIAN is given the data values are placed in the memory locations beginning with the lowest value byte first and the highest value byte last. Independently of the endianess the bit offset 0 addresses the lowest value bit of a data type.

EXAMPLE

TYPE L : ULINT := 16#1122_3344_5566_7788; END_TYPE; has the memory location

for big endian:	16#11, 16#22, 16#33, 16#44, 16#55, 16#66, 16#77, 16#88
for little endian:	16#88, 16#77, 16#66, 16#55, 16#44, 16#33, 16#22, 16#11 .

Table 12 - Data type declaration features – Using keyword TYPE

No.	Feature/textual example

1	Direct derivation from elementary types, e.g.:
	TYPE
	RU_REAL : REAL;
	END_TYPE

2	Enumerated data types, e.g.: TYPE ANALOG_SIGNAL_TYPE : (SINGLE_ENDED, DIFFERENTIAL) ; END_TYPE
3[a]	Subrange data types, e.g.: TYPE ANALOG_DATA : INT (-4095..4095) ; END_TYPE
4	Array data types, e.g.: TYPE ANALOG_16_INPUT_DATA : ARRAY [1..16] OF ANALOG_DATA ; END_TYPE
5	Structured data types, e.g.: TYPE ANALOG_CHANNEL_CONFIGURATION : STRUCT RANGE : ANALOG_SIGNAL_RANGE ; MIN_SCALE : ANALOG_DATA ; MAX_SCALE : ANALOG_DATA ; END_STRUCT ; ANALOG_16_INPUT_CONFIGURATION : STRUCT SIGNAL_TYPE : ANALOG_SIGNAL_TYPE ; FILTER_PARAMETER : SINT (0..99) ; CHANNEL : ARRAY [1..16] OF ANALOG_CHANNEL_CONFIGURATION ; END_STRUCT ; END_TYPE

No.	Feature/textual example

6	Structured data type with explicit layout
	`TYPE`
	` A_buffer: ARRAY [0..150] OF BYTE;`
	` Com_data: STRUCT LAYOUT_EXPLICIT BIG_ENDIAN OVERLAP`
	`   buffer    AT %B0         : A_buffer;`
	`   head      AT %B0         : INT;`
	`   length    AT %B2         : USINT;`
	`   data1     AT %B3         : ARRAY [1..50] of INT;`
	`   data2     AT %B3         : ARRAY [1..20] of REAL;`
	`   END_STRUCT;`
	`END_TYPE;`

NOTE For examples of the use of these types in variable declarations.	
[a] This usage is **deprecated** and may not appear in future Editions of IEC 61131-3.	

6.3.4.3 Initialization

In Table 12 and Table 14 the initialization along with the declaration is shown.

- **Enumeration**

The default initial value of an *enumerated* data type shall be the first identifier in the associated enumeration list, or a value specified by the assignment operator. For instance, as shown in Table 12 feature 2 and Table 14 feature 2, the default initial values of elements of data types ANALOG_SIGNAL_TYPE and ANALOG_SIGNAL_RANGE are SINGLE_ENDED and UNIPOLAR_1_5V, respectively.

- **Subrange**

For data types with *subranges*, the default initial values shall be the first (lower) limit of the subrange, unless otherwise specified by an assignment operator. For instance, as declared in table 12, the default initial value of elements of type ANALOG_DATA is -4095, while the default initial value for the FILTER_PARAMETER sub-element of elements of type ANALOG_16_ INPUT_CONFIGURATION is zero. In contrast, the default initial value of elements of type ANALOG_DATAZ as declared in table 14 is zero.

- **Structure**

Structures with explicit layout shall be initialized according to the declared layout. For initialization of a structure with overlapping components the initialization specified in a textually succeeding is prior to a textually preceding one.

The default maximum length of elements of type STRING and WSTRING shall be an **implementation-dependent** value unless specified otherwise by a parenthesized maximum length (which shall not exceed the implementation-dependent default value) in the associated declaration.

EXAMPLE
If type STR10 is declared by

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

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

The maximum allowed length of STRING and WSTRING variables is an **implementation dependency**.

For other derived data types, the default initial values, unless specified otherwise by the use of the assignment operator := in the TYPE declaration, shall be the default initial values of the underlying elementary data types as defined in Table 13. Further examples of the use of the assignment operator for initialization are given in 6.4.3.

1222 **Table 13 - Default initial values of elementary data types**

Data type(s)	Initial value
BOOL, SINT, INT, DINT, LINT	0
USINT, UINT, UDINT, ULINT	0
BYTE, WORD, DWORD, LWORD	0
REAL, LREAL	0.0
TIME	T#0S
DATE	D#0001-01-01
TIME_OF_DAY	TOD#00:00:00
DATE_AND_TIME	DT#0001-01-01-00:00:00
STRING	'' (the empty string)
WSTRING	"" (the empty string)
CHAR	'$00'
WCHAR	"$0000"

1223

Table 14 - Data type initial value declaration features

No.	Feature/textual example

1	Initialization of directly derived types, e.g.:
	`TYPE`
	`  FREQ : REAL := 50.0 ;`
	`END_TYPE`

2	Initialization of enumerated data types, e.g.:
	`TYPE`
	`  ANALOG_SIGNAL_RANGE :`
	`    (BIPOLAR_10V,         (* -10 to +10 VDC  *)`
	`     UNIPOLAR_10V,        (*   0 to +10 VDC  *)`
	`     UNIPOLAR_1_5V,       (* + 1 to + 5 VDC  *)`
	`     UNIPOLAR_0_5V,       (*   0 to + 5 VDC  *)`
	`     UNIPOLAR_4_20_MA,    (* + 4 to +20 mADC *)`
	`     UNIPOLAR_0_20_MA     (*   0 to +20 mADC *)`
	`    ) := UNIPOLAR_1_5V;`
	`END_TYPE`

3	Initialization of subrange data types, e.g.:
	`TYPE`
	`  ANALOG_DATAZ : INT (-4095 .. 4095) := 0 ; (* Initialized to zero *)`
	`END_TYPE`

4	Initialization of array data types, e.g.:
	`TYPE ANALOG_16_INPUT_DATAI :`
	`  ARRAY [1..16] OF ANALOG_DATAZ := [8(-4095), 8(4095)];`
	`END_TYPE`

5	Initialization of structured data type elements, e.g.:
	`TYPE ANALOG_CHANNEL_CONFIGURATIONI :`
	`    STRUCT`
	`      RANGE : ANALOG_SIGNAL_RANGE ;`
	`      MIN_SCALE : ANALOG_DATA := -4095 ; (* ANALOG_DATA is defined *)`
	`      MAX_SCALE : ANALOG_DATA :=  4095 ; (* in Table 12, feature 3 *)`
	`    END_STRUCT;`
	`END_TYPE`

6	Initialization of derived structured data types, e.g.:
	`TYPE`
	`  ANALOG_CHANNEL_CONFIGZ :`
	`    ANALOG_CHANNEL_CONFIGURATIONI := (MIN_SCALE := 0, MAX_SCALE := 4000);`
	`END_TYPE`

7	Structured data type with explicit layout
	`TYPE`
	`A_buffer: ARRAY [0..150] OF BYTE := 151(16#01);`
	`Com_data:     STRUCT LAYOUT_EXPLICIT BIG_ENDIAN OVERLAP`
	`  buffer      AT %B0        : A_buffer := 151(16#22);`
	`                                   // overrides initialization of Abuffer`
	`  head        AT %B0        : INT := 19;  // overrides initialization of buffer`
	`  length      AT %B2        : USINT := 5; // overrides initialization of buffer`
	`  data1       AT %B3        : ARRAY [1..50] of INT;`
	`  data2       AT %B3        : ARRAY [1..20] of REAL;`
	`END_STRUCT;`
	`END_TYPE;`

6.3.4.4 Usage of TYPE

The usage of variables which are declared to be of derived data types shall conform to the following rules:

a) A single-element variable as defined in 6.4.2, of a derived type, can be used anywhere that a variable of its base (parent's) type can be used, for example variables of the types RU_REAL and FREQ as shown in Table 12 and Table 14 can be used anywhere that a variable of type REAL could be used, and variables of type ANALOG_DATA can be used anywhere that a variable of type INT could be used.

This rule can be applied recursively. For example, given the declarations below, the variable R3 of type R2 can be used anywhere a variable of type REAL can be used:

```
    TYPE R1 : REAL := 1.0; END_TYPE
    TYPE R2 : R1; END_TYPE
    VAR  R3 : R2; END_VAR
```

b) An element of a multi-element variable, as defined in 6.4.2, can be used anywhere the base (parent) type can be used, for example, given the declaration of ANALOG_16_INPUT_DATA in Table 12 and the declaration

```
     VAR INS : ANALOG_16_INPUT_DATA; END_VAR
```

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

Similarly, given the definition of Com_data in Table 12 and the declarations

```
    VAR telegram : Com_data; END_VAR
```

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

This rule can also be applied recursively, for example, given the declarations of ANALOG_16_INPUT_CONFIGURATION, ANALOG_CHANNEL_CONFIGURATION and ANALOG_DATA in Table 12 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 Variables

6.4.1 General

In contrast to the external representations of data described in 6.2, *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. A variable can be declared to be one of the elementary types defined in 6.3.2, or one of the derived types which are declared as defined in 6.3.4.2.

6.4.2 Representation

6.4.2.1 General

A *single-element variable* is defined as a variable which represents a single data element of one of the elementary types defined in 6.3.2; a derived enumeration or subrange type as defined in 6.3.4.2; or a derived type whose “parentage”, as defined recursively in 6.3.4.4, is traceable to an elementary, enumeration or subrange type. Subclause 6.4.2 specifies the means of representing such variables *symbolically,* or alternatively in a manner which *directly* represents the association of the data element with physical or logical locations in the programmable controller’s input, output, or memory structure.

NOTE The use of *directly represented variables* in the bodies of *functions, function block types* 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.

Subclause 6.4.2.3 specifies the means of representing *multi-element variables*, i.e., *arrays* and *structures.*

6.4.2.2 Single-element variables

Identifiers, as defined in 6.1.2, shall be used for symbolic representation of variables.

Direct representation of a single-element variable shall be provided by a special symbol formed by the concatenation of the percent sign “%” (character code 037 decimal in table 1 – Row 00 of ISO/IEC 10646-1), a *location prefix* and a *size prefix* from Table 15, and one or more unsigned integers, separated by periods (the “full stop” character, “.”).

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 one or several unsigned integers in the concatenation to indicate that the direct representation is not yet fully specified. The percent sign and the location prefix `I`, `Q` or `M` from Table 15 shall always be present in the direct representation.

In both cases, 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 as defined in 6.7 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.

EXAMPLES

`%QX75` and `%Q75`	Output bit 75
`%IW215`	Input word location 215
`%QB7`	Output byte location 7
`%MD48`	Double word at memory location 48
`%IW2.5.7.1`	See explanation below
`%Q*`	Output at a not yet specified location

The manufacturer 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. When a 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. For instance, the variable `%IW2.5.7.1` may represent the first "channel" (word) of the seventh "module" in the fifth "rack" of the second "I/O bus" of a programmable controller system.

NOTE The use of hierarchical addressing to permit a program in one programmable controller system to access data in another programmable controller shall be considered a language **extension**.

The use of directly represented variables is permitted in *function blocks* as defined in 6.5.3, *programs* as defined in 6.5.5, and in *configurations* and *resources* as defined in 6.7. The maximum number of levels of hierarchical addressing is an **implementation dependency**.

Table 15 - Location and size prefix features for directly represented variables

No.	Prefix	Meaning	Default data type
1	`I`	Input location	
2	`Q`	Output location	
3	`M`	Memory location	
4	`X`	Single bit size	BOOL
5	`None`	Single bit size	BOOL
6	`B`	Byte (8 bits) size	BYTE
7	`W`	Word (16 bits) size	WORD
8	`D`	Double word (32 bits) size	DWORD
9	`L`	Long (quad) word (64 bits) size	LWORD
10	*	Use of an asterisk to indicate a not yet specified location (NOTE 2)	
NOTE 1 National standards organizations can publish tables of translations of these prefixes. NOTE 2 Use of feature 10 in this table requires feature 11 of Table 57 and vice versa.			

6.4.2.3 Partial access of ANY_BIT variables

For variables of the data type ANY_BIT (`BYTE`, `WORD`, `DWORD`, `LWORD`) as defined in 6.3.3 a partial access of a bit, byte, word and double word of the variable is defined in Table 16. To address the part of the variable the *size prefix* as defined for directly represented variables in Table 15 feature 4 to 9 (X, B, W, D, L) is used in combination with a constant integer number (0 to max) for the address within the variable. 0 refers to the least significant and max refers to the most significant part.

EXAMPLE:

```
    Var                                                                  *
     Bo : BOOL;
     By : BYTE;
     Wo : WORD;
     Do : DLWORD;
     Lo : LWORD;
    END_VAR;

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

     By := Wo.B1;  // byte 1 of Wo;
     By := Do.B7;  // byte 7 of Wo;       .
```

Table 16 - Partial access of ANY_BIT variables

No.	Data type	Access to	Syntax
1a	BYTE	bits	`<variable name>.X0 to` `<variable name>.X7` (Note)
1b	WORD	bits	`<variable name>.X0 to` `<variable name>.X15` (Note)
1c	DWORD	bits	`<variable name>.X0 to` `<variable name>.X31` (Note)
1d	LWORD	bits	`<variable name>.X0 to` `<variable name>.X63` (Note)

2a	WORD	bytes	`<variable name>.B0 to`
			`<variable name>.B1`

2b	DWORD	bytes	`<variable name>.B0 to`
			`<variable name>.B3`

2c	LWORD	bytes	`<variable name>.B0 to`
			`<variable name>.B7`

3a	DWORD	words	`<variable name>.W0 to`
			`<variable name>.W1`

3b	LWORD	words	`<variable name>.W0 to`
			`<variable name>.W3`

4	LWORD	dwords	`<variable name>.D0 to`
			`<variable name>.D1`

Note The bit access prefix X may be omitted according Table 15 feature 5. Example: By.X7 is equivalent to By.7.			

[Editor's note: Syntax in annex B to be done.]

An alternative to the partial access defined above is possible with an array using the (computable) array index as defined in 6.4.4.1.

6.4.2.4 Multi-element variables

The *multi-element variable* types defined in this standard are *arrays* and *structures*.

- **Array**

  An *array* is a collection of data elements of the same data type referenced by one or more *subscripts* enclosed in brackets and separated by commas. In the ST language defined in 7.3, a subscript shall be an expression yielding a value corresponding to one of the sub-types of generic type ANY_INT as defined in Figure 4. The form of subscripts in the IL language defined in 7.2, and the graphic languages defined in 8, is restricted to *single-element variables* or *integer literals*.

EXAMPLE 1

A usage of array variables in the ST language could be:

```
    OUTARY[%MB6,SYM] := INARY[0] + INARY[7] - INARY[%MB6] * %IW62;
```

It shall be an **error** if the value of a subscript is outside the range specified in the declaration of the variable.

▪ Structure

A *structured variable* is a variable which is declared to be of a type which has previously been specified to be a *data structure*, i.e., a data type consisting of a collection of named elements.

An element of a structured variable 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 component names to access the particular data element within the data structure.

EXAMPLE 2

If the variable MODULE_5_CONFIG has been declared to be of type ANALOG_16_INPUT_CONFIGURATION as shown in table 12, the following statements in the ST language defined in 7.3 would cause the value SINGLE_ENDED to be assigned to the element SIGNAL_TYPE of the variable MODULE_5_CONFIG, while the value BIPOLAR_10V would be assigned to the RANGE sub-element of the fifth CHANNEL element of MODULE_5_CONFIG:

```
    MODULE_5_CONFIG.SIGNAL_TYPE := SINGLE_ENDED;
    MODULE_5_CONFIG.CHANNEL[5].RANGE := BIPOLAR_10V;
```

6.4.2.5 Variable-length arrays

Variable-length arrays can only be used as input, output or in-out parameters**.** 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.

EXAMPLE 1

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

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

To handle the variable-length arrays the following standard functions shall be provided:

Table 17 - Variable-length array features

No.	Feature description	Examples
Variable-length array declaration		

1	ARRAY [*, *, ...]	VAR_IN_OUT
		  A: ARRAY [*, *] OF INT;
		END_VAR;

Standard functions for variable-length arrays		
	Graphical	**Usage example in ST**

2	Get lower bound of an array:
	`          +------------+`
	`          ! LOWERBOUND !`
	`ARRAY ----! ARR        !--- DINT`
	`UINT -----! DIM        !`
	`          +----------- +`
		Get lower bound of the 2nd dimention of the array A::
		low2 := LOWERBOUND (A, 2);

3	Get upper bound of an array:
	`          +------------+`
	`          ! UPPERBOUND !`
	`ARRAY ----! ARR        !--- DINT`
	`UINT -----! DIM        !`
	`          +----------- +`
		Get upper bound of the 2nd dimention of the array A::
		up2 := UPPERBOUND (A, 2);

EXAMPLE 2

```
A1: ARRAY [1..10] OF INT := 10(1);
A2: ARRAY [1..20, -2..2] OF INT := 20(5(1)):

LOWERBOUND (A1, 1)   →  1
UPPERBOUND (A1, 1)   → 10
LOWERBOUND (A2, 1)   →  1
UPPERBOUND (A2, 1)   → 20
LOWERBOUND (A2, 2)   → -2
UPPERBOUND (A2, 2)   →  2
```

EXAMPLE 3 Array Summation

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

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

SUM (A1) → 10
SUM (A2[1]) → 5
```

EXAMPLE 4 Matrix Multiplication

```
FUNCTION MATRIX_MUL: VOID;
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:= LOWERBOUND(A,1) TO UPPERBOUND(A,1)
  FOR j:= LOWERBOUND(B,2) TO UPPERBOUND(B,2)
    S:= 0;
    FOR k:= LOWERBOUND(A,2) TO UPPERBOUND(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..3, 1..4] OF INT;
END_VAR

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

6.4.3 Initialization

When a configuration element (*resource* or *configuration*) is "started" as defined in 6.7, each of the variables associated with the configuration element and its *programs* can take on one of the following initial values:

- the value the variable had when the configuration element was "stopped" (a retained value);
- a user-specified initial value;
- the default initial value for the variable's associated data type.

The user can declare that a variable is to be *retentive* by using the RETAIN qualifier specified in Table 18, when this feature is supported by the implementation.

The initial value of a variable upon starting of its associated configuration element shall be determined according to the following rules:

1) If the starting operation is a "warm restart" as defined in IEC 61131-1, the initial values of *retentive* variables shall be their *retained* values as defined above.
2) If the operation is a "cold restart as defined in IEC 61131-1, the initial values of retentive variables shall be the user-specified initial values or the default value for the associated data type of any variable for which no initial value is specified by the user.
3) Non-retained variables shall be initialized to the user-specified initial values, or to the default value for the associated data type of any variable for which no initial value is specified by the user.
4) Variables which represent *inputs* of the *programmable controller system* as defined in IEC 61131-1 shall be initialized in an **implementation-dependent** manner.

6.4.4 Declaration

Each declaration of a program organization unit type (i.e., each declaration of a *program*, *function*, or *function block*) shall contain at its beginning at least one *declaration part* which specifies the types (and, if necessary, the physical or logical location) of the variables used in the organization unit. This declaration part shall have the textual form of one of the keywords VAR, VAR_INPUT, or VAR_OUTPUT as defined in Table 18, followed in the case of VAR by zero or one occurrence of the qualifiers RETAIN,NON_RETAIN or the qualifier CONSTANT, and in the case of VAR_INPUT or VAR_OUTPUT by zero or one occurrence of the qualifier RETAIN or NON_RETAIN, followed by one or more declarations separated by semicolons and terminated by the keyword END_VAR. When a programmable controller supports the declaration by the user of initial values for variables, this declaration shall be accomplished in the declaration part(s) as defined in this subclause.

Table 18 - Variable declaration keywords

Keyword	Variable usage
VAR	Internal to organization unit
VAR_INPUT	Externally supplied, not modifiable within organization unit
VAR_OUTPUT	Supplied by organization unit to external entities
VAR_IN_OUT	Supplied by external entities – can be modified within organization unit

VAR_EXTERNAL	Supplied by configuration via VAR_GLOBAL (6.7.2) Can be modified within organization unit
VAR_GLOBAL	Global variable declaration (6.7.2)
VAR_ACCESS	Access path declaration (6.7.2)
VAR_TEMP	Temporary storage for variables in function blocks and programs (6.4.4)
VAR_CONFIG	Instance-specific initialization and location assignment.
RETAIN[b, c, d, e]	Retentive variables (see preceding text)
NON_RETAIN[b,c,d, e]	Non-retentive variables (see preceding text)
CONSTANT[a]	Constant (variable cannot be modified)
AT	Location assignment

NOTE 1 The usage of these keywords is a feature of the program organization unit or configuration element in which they are used.

NOTE 2 Examples of the use of VAR_IN_OUT variables are given in Figure 13 b) and Figure 14.

[a] The CONSTANT qualifier shall not be used in the declaration of *function block instances* as described in 6.5.3.2.

[b] The RETAIN and NON_RETAIN qualifiers may be used for *variables* declared in VAR, VAR_INPUT, VAR_OUTPUT, and VAR_GLOBAL blocks but not in VAR_IN_OUT blocks and not for individual elements of structures.

[c] Usage of RETAIN and NON_RETAIN for *function block* and *program instances* is allowed. The effect is that all members of the instance are treated as RETAIN or NON_RETAIN, except if:

- the member is explicitly declared as RETAIN or NON_RETAIN in the function block or program type definition;
- the member itself is a *function block*.

[d] Usage of RETAIN and NON_RETAIN for *instances* of structured data types is allowed. The effect is that all structure members, also those of nested structures, are treated as RETAIN or NON_RETAIN.

[e] Both RETAIN and NON_RETAIN are features. If a variable is neither explicitly declared as RETAIN nor as NON_RETAIN the “warm start” behaviour of the variable is **implementation dependent**.

Within *function blocks* and *programs*, variables can be declared in a VAR_TEMP...END_VAR construction. These variables are allocated and initialized at each call of an *instance* of the program organization unit, and do not persist between calls.

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 explicit argument passing via variables which have been declared as *inputs* or *outputs* of those units.

The one exception to this rule is the case of variables which have been declared to be *global*, as defined in 6.7.2. 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 block;
- 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 an **implementation dependency**.

Table 19 - Usages of VAR_GLOBAL, VAR_EXTERNAL and CONSTANT declarations

Declaration in containing element	Declaration in contained element	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` construct in a contained element may lead to unanticipated behaviours, for instance, when the value of an external variable is modified by another contained element in the same containing element.		

6.4.4.1 Type assignment

As shown in Table 20, the `VAR...END_VAR` construction shall be used to specify data types and retentivity for directly represented variables. This construction shall also be used to specify data types, retentivity, and (where necessary, in *programs* and `VAR_GLOBAL` declarations only) the physical or logical location of symbolically represented single- or multi-element variables. The usage of the `VAR_INPUT`, `VAR_OUTPUT`, and `VAR_IN_OUT` constructions**.**

The assignment of a physical or logical address to a symbolically represented variable shall be accomplished by the use of the `AT` keyword. Where no such assignment is made, automatic allocation of the variable to an appropriate location in the programmable controller memory shall be provided.

The asterisk notation (Table 15, feature 10) can be used in address assignments inside programs and function block types to denote not yet fully specified locations for directly represented variables.

Table 20 - Variable type assignment features – AT keyword

No.	Feature/examples	
1[a]	**Declaration of locations of symbolic variables – AT keyword**	

	`VAR_GLOBAL`
	`LIM_SW_S5 AT %IX27 : BOOL;`
	`END_VAR`
		Assigns input bit 27 to the Boolean variable `LIM_SW_5` (NOTE 2)

	`VAR`
	`CONV_START AT %QX25 : BOOL;`
	`END_VAR`
		Assigns output bit 25 to the Boolean variable `CONV_START`

	`TEMPERATURE AT %IW28: INT;`	Assigns input word 28 to the integer variable `TEMPERATURE` (NOTE 2)

	`VAR`
	`C2 AT %Q* : BYTE;`
	`END_VAR`
		Assigns not yet located output byte to bitstring variable `C2` of length 8 bits

2[a]	**Array location assignment – AT keyword**	

	`VAR`
	`INARY AT %IW6 :`
	`ARRAY [0..9] OF INT;`
	`END_VAR`
		Declares an array of 10 integers to be allocated to contiguous input locations starting at `%IW6` (note 2)

No.	Feature/examples	
3	**Automatic memory allocation of symbolic variables**	

	`VAR`
	`CONDITION_RED : BOOL;`
	`IBOUNCE : WORD;`
	`MYDUB : DWORD;`
	`AWORD, BWORD, CWORD : INT;`
	`MYSTR: STRING[10];`
	`END_VAR`
		Allocates a memory bit to the Boolean variable `CONDITION_RED`.
		Allocates a memory word to the 16-bit string variable `IBOUNCE`.
		Allocates a double memory word to the 32-bit-string variable `MYDUB`.
		Allocates 3 separate memory words for the integer variables `AWORD`, `BWORD`, and `CWORD`
		Allocates memory to contain a string with a maximum length of 10 characters. After initialization, the string has length 0 and contains the empty string `''`.

4	**Array declaration**	

	`VAR`
	`THREE :`
	`ARRAY[1..5,1..10,1..8]OF INT;`
	`END_VAR`
		Allocates 400 memory words (5 · 10 · 8) for a three-dimensional array of integers

5	**Retentive array declaration**	

	`VAR`
	`RETAIN RTBT :`
	`ARRAY[1..2, 1..3] OF INT;`
	`END_VAR`
		Declares retentive array `RTBT` with “cold restart” initial values of `0` for all elements

6	**Declaration of structured variables**	

	`VAR`
	`MODULE_8_CONFIG :`
	`ANALOG_16_INPUT_CONFIGURATION;`
	`END_VAR`
		Declaration of a variable of derived data type (see Table 12)

7[b]	**Declaration of enumerated variables**	

	`VAR`
	`Y : (Red, Yellow, Green);`
	`END_VAR`
		Declaration of an enumerated variable

8[c,d]	**Declaration of subrange variables**	

	`VAR`
	`Z : SINT(5..95);`
	`END_VAR`
		Declaration of a subrange variable

9	`VAR`
	`com1 : Com_data;`
	`END_VAR`
		Declaration of a variable of a structure with explicit layout (see Table 14)

NOTE Initialization of system inputs is **implementation-dependent**;		

[a] If directly represented variables are explicitly located (%), features 1 to 4 can only be used in `PROGRAM` and `VAR_GLOBAL` declarations. If the asterisk notation of feature 10 in Table 15 is used to indicate instance specific location assignment of a partly specified directly represented variable, features 1 and 2 can not be used, and features 3 and 4 can only be used in declarations of internal variables of function blocks and programs.

[b] This declaration shall be interpreted to mean that the values of the declared variable are restricted to the specified enumerated values.

[c] This declaration shall be interpreted to mean that the values of the declared variable are restricted to the specified subrange including the declared limit values.

[d] This usage is **deprecated** and may not appear in future Editions of IEC 61131-3.

EXAMPLE

An alternative to a partial bit access to an ANY-BIT variable defined in 6.4.2.3 is:

```
TYPE
  X  : WORD;
END_TYPE

VAR
```

```
  XA  AT X [ ARRAY [0..15] of BOOL;
  B  : BOOL;
  I   : INT;
END_VAR;

I   := 2;
B := XA [i];
```

6.4.4.2 Initial value assignment

The `VAR...END_VAR` construction can be used as shown in Table 21 to specify initial values of directly represented variables or symbolically represented single- or multi-element variables.

Initial values can also be specified by using the instance-specific initialization feature provided by the `VAR_CONFIG...END_VAR` construct described in 6.7.2 (Table 57, feature 11). Instance -specific initial values always override type-specific initial values.

NOTE The usage of the `VAR_INPUT`, `VAR_OUTPUT`, and `VAR_IN_OUT` constructions is defined in 6.7.

Initial values cannot be given in `VAR_EXTERNAL` declarations.

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

Parentheses can be used as a repetition factor in array initialization lists.

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

EXAMPLE 2 See Table 21 feature 7.

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.

When a variable is declared to be of a derived, structured data type as defined in 6.3.4.2, initial values for the elements of the variable can be declared in a parenthesized list following the data type identifier, as shown in Table 21. 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 data type declaration.

When a variable is declared to be a *function block instance*, as defined in 6.5.3.4, initial values for the inputs, outputs and public variables of the function block can be declared in a parenthesized list following the assignment operator that follows the function block type identifier as shown in Table 21. Elements for which initial values are not listed shall have the default initial values declared for those elements in the function block declaration.

Table 21 - Variable initial value assignment features

No.	Feature/examples	
1 [a]	Initialization of directly represented variables	

	`VAR`
	` AT %QX5.1  : BOOL  := 1;`
	` AT %MW6    : INT   := 8 ;`
	`END_VAR`
		Boolean type, initial value = `1`
		Initializes a memory word to integer `8`.

2 [a]	Initialization of directly represented retentive variables	

	`VAR RETAIN`
	` AT %QW5 : WORD := 16#FF00;`
	`END_VAR`
		At cold restart, the 8 leftmost bits of the 16-bit string at output word 5 are to be initialized to 1 and the 8 rightmost bits to 0.

3 [a]	Location and initial value assignment to symbolic variables	

	`VAR`
	` VALVE_POS AT %QW28 : INT := 100;`
	`END_VAR`
		Assigns output word 28 to the integer variable `VALVE_POS` with an initial value of 100.

No.	Feature/examples	
4 [a]	Array location assignment and initialization	

	VAR
	 OUTARY AT %QW6 :
	 ARRAY[0..9] OF INT := [10(1)];
	END_VAR
		Declares an array of 10 integers to be allocated to contiguous output locations starting at %QW6, each with an initial value of 1.

5	Initialization of symbolic variables	

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

6	Array initialization	

	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];
	END_VAR
		Allocates 8 memory bits to contain initial values
		BITS[0] := 1, BITS[1] := 1,...,
		BITS[6] := 0, BITS[7] := 0.
		Allocates a 2-by-3 integer array TBT with initial values
		TBT[1,1] :=9, TBT[1,2] :=8,
		TBT[1,3] :=10, TBT[2,1] :=10,
		TBT[2,2] :=10, TBT[2,3] :=6.

7	Retentive array declaration and initialization	

	VAR RETAIN
	 RTBT :
	  ARRAY[1..2,1..3] OF INT
	   := [9,8,3(10)];
	END_VAR
		Declares retentive array RTBT with “cold restart” initial values of:
		RTBT[1,1] := 9, RTBT[1,2] := 8,
		RTBT[1,3] := 10, RTBT[2,1] := 10,
		RTBT[2,2] := 10, RTBT[2,3] := 0.

8	Initialization of structured variables	

	VAR
	 MODULE_8_CONFIG :
	  ANALOG_16_INPUT_CONFIGURATION :=
	  (SIGNAL_TYPE := DIFFERENTIAL,
	  CHANNEL :=
	   [4((RANGE := UNIPOLAR_1_5V)),
	     (RANGE := BIPOLAR_10_V,
	     MIN_SCALE := 0,
	     MAX_SCALE := 500)]);
	END_VAR
		Initialization of a variable of derived data type (see table 12)
		This example illustrates the declaration of a non-default initial value for the fifth element of the CHANNEL array of the variable MODULE_8_CONFIG.

9	Initialization of constants	

	VAR
	 CONSTANT PI : REAL := 3.141592;
	END_VAR	

10	Initialization of function block instances	

	VAR
	 TempLoop :
	  PID :=
	   (PropBand := 2.5,
	   Integral := T#5s);
	END_VAR
		Allocates initial values to inputs and outputs of a function block instance.

[a] If directly represented variables are explicitly located (%), features1 to 4 can only be used in PROGRAM and VAR_GLOBAL declarations.

Where constants may be used for initialization, also constant expressions may be used.

- **Constant expression**

Constant expressions consist of literals, enumeration values, constant variables, and operators to connect them to an expression. The constant variables are variables which are defined inside a variable section which contains the keyword CONSTANT. The rules defined for expressions in 7.3.1 shall apply.

EXAMPLE Constant expression

PropBand := 2.5 *Pi/2;   // Pi := 3.14159.

6.5 Program organization units

6.5.1 General

The program organization units defined in this part of IEC 61131 are the *function, function block*, and *program*. These program organization units can be delivered by the manufacturer, or programmed by the user by the means defined in this part of the standard.

Program organization units shall not be *recursive*; that is, the call of a program organization unit shall not cause the call of another program organization unit of the same type.

The information necessary to determine execution times of program organization units may consist of one or more **implementation dependencies.**

6.5.2 Functions

6.5.2.1 General

For the purposes of programmable controller programming languages, a *function* is defined as a program organization unit (POE) which, when executed, yields no (VOID) or exactly one data element, which is considered to be the function result, and arbitrarily many additional output elements (`VAR_OUTPUT` and `VAR_IN_OUT`).

If a function result exists, it can be multi-valued as any data element, i.e. it can be an array or structure and the call of a function can be used in textual languages as an operand in an expression.

The keyword VOID indicates that the function has no function result. Then in textual languages it can not be used as an operand in an expression.

EXAMPLE

The `SIN` and `COS` functions could be used as shown in Figure 5

```
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) Structured Text (ST) language - see 7.3

```
       +-----+      +------+     +------+
X ---+-| COS |--+  -|EN ENO|-----|EN ENO|--- V
     | |     |  |   |      |     |      |
     | +-----+  +---| DIV  |-----| ADD  |--- Z
     |              |      |     |      |
     | +-----+      |      |   +-|      |
Y -+---| SIN |------|      |   | +------+
   | | |     |      +------+   |
   | | +-----+                 |
   | |                         |
   | | +-----+      +------+   |
   | +-| SIN |--+  -|EN ENO|-  |
   |   |     |  |   |      |   |
   |   +-----+  +- -| MUL  |---+
   |                |      |
   |   +-----+      |      |
   +---| COS |------|      |
       |     |      +------+
       +-----+
```

b) Function Block Diagram (FBD) language - see 8.3

NOTE This figure shows two different representations of the same functionality. It is not required to support any automatic transformation between the two forms of representation.

Figure 5 - Examples of function usage

Functions shall contain no internal state information, i.e., call of a function with the same arguments (input variables `VAR_INPUT` and in-out variables `VAR_IN_OUT`) and the same values of external variables VAR_EXTERNAL shall always yield the same result values of its output variables `VAR_OUTPUT`, in-out variables `VAR_IN_OUT`, external variables `VAR_EXTERNAL` and its function result if any.

Any function type that has already been declared can be used in the declaration of another program organization unit, as shown in Figure 3.

6.5.2.2 Representation

Functions and their call can be represented either graphically or textually.

In the textual languages defined in 7 of this Part, the call of functions shall be according to the following rules:

1. Input argument assignment shall follow the rules given in Table 24 a).
2. Assignments of output variables of the function shall be either empty or to variables.
3. Assignments to `VAR_IN_OUT` arguments shall be variables.
4. Assignments to `VAR_INPUT` arguments may be empty as in feature 1 of Table 24 a), constants, variables or function calls. In the latter case, the function result is used as the actual argument.

In the graphic languages defined in Clause 7 of this Part, functions shall be represented as graphic blocks according to the following rules:

a) The form of the block shall be rectangular or square.

b) The size and proportions of the block may vary depending on the number of inputs and other information to be displayed.

c) The direction of processing through the block shall be from left to right (input variables on the left and output variables on the right).

d) The function name or symbol, as specified below, shall be located inside the block.

e) Provision shall be made for input and output variable names appearing at the inside left and right sides of the block respectively when the block represents:

- one of the standard functions defined in 6.5.2.6, when the given graphical form includes the variable names; or
- any additional function declared as specified in 6.5.2.4.

  This usage is subject to the following provisions:

  - Where no names are given for input variables in 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.
  - The default names described above may, but need not appear at the inside left-hand side of the graphic representation.

f) An additional input EN and/or output ENO as specified in 6.5.2.3 may be used. If present, they shall be shown at the uppermost positions at the left and right side of the block, respectively.

g) 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 function is used for the assignment of its output value as specified in 6.5.3.4, no output variable name shall be shown at the right side of the block.

h) Argument connections (including function result) shall be shown by signal flow lines.

i) 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 defined in 6.1.1, this shall be represented by the upper case alphabetic “O”, as shown in Table 22.

j) 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.

k) Function results and function outputs (VAR_OUTPUT) can be connected to a variable, used as input to other function block instances or functions, or can be left unconnected.

l) It shall be an error if any VAR_IN_OUT variable of any function block call or function 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 assigned using the “:=” operator in a textual call, to a variable declared (without the CONSTANT qualifier) in a VAR_IN_OUT, VAR, VAR_OUT, or VAR_EXTERNAL block of the containing program organization unit, or to a “properly mapped” VAR_IN_OUT of another contained function block instance or function call.

m) A “**properly mapped**” (as shown in rule l) above) VAR_IN_OUT variable of a function block instance or a function call can be connected graphically at the right, or assigned using the “:=” operator in a textual assignment statement, to a variable declared in a VAR, VAR_OUT 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.

Table 22 - Graphical negation of Boolean signals

No.	Feature[a, b]	Representation

1	Negated input	`    +---+`
		`---O|   |----`
		`    +---+`

2	Negated output	`    +---+`
		`----|   |O---`
		`    +---+`

[a] If either of these features is supported for *functions*, it shall also be supported for *function blocks* as defined in 6.5.3, and vice versa.

[b] The use of these constructs is forbidden for in-out variables.

EXAMPLE

Table 23 illustrates both the graphical and equivalent textual use of functions, including the use of a standard function (`ADD`) with no defined formal argument names; a standard function (`SHL`) with defined formal argument names; the same function with additional use of `EN` input and negated `ENO` output; and a user-defined function (`INC`) with defined formal argument names.

Table 23 - Usage of formal argument names

Example	Explanation

`    +-----+`
`
	ADD |`
	`B---|     |---A`
	`C---|     |`
	`D---|     |`
	`    +-----+`
		Graphical use of `ADD` function
		(FBD language)
		(No formal variable names)

`A := ADD(B,C,D);`	Textual use of `ADD` function (ST language)

`    +-----+`
`
	SHL |`
	`B---|IN   |---A`
	`C---|N    |`
	`    +-----+`
		Graphical use of `SHL` function
		(FBD language)
		(Formal argument names)

`A := SHL(IN := B,N := C);`	Textual use of `SHL` function (ST language)

`        +---------+`
`
	SHL |`
	`ENABLE--|EN    ENO|O--NO_ERR`
	`    B---|IN       |---A`
	`    C---|N        |`
	`        +---------+`
		Graphical use of `SHL` function
		(FBD language)
		(Formal argument names; use of `EN` input and negated `ENO` output)

`A := SHL(     EN  := ENABLE,`
`               IN  := B,`
`               N   := C,`
`               NOT ENO => NO_ERR);`
	Textual use of `SHL` function
	(ST language)

`    +-----+`
`
	INC |`
	`
		|---A`
		`X---|V---V|---X`
		`    +-----+`
			Graphical use of user-defined
			`INC` function
			(FBD language)
			(Formal argument names for `VAR_IN_OUT`)

`A := INC(V := X) ;`	Textual use of `INC` function (ST language)

Features for the textual call of functions are defined in Table 24. The textual call of a function shall consist of the function name followed by a list of arguments. In the ST language defined in 7.3, the arguments shall be separated by commas and this list shall be delimited on the left and right by parentheses.

In feature 1 of Table 24 (formal call), the argument list has the form of a set of assignments of actual values to the formal argument names (formal argument 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 ordering of arguments in the list shall not be significant. In feature 1 of Table 24, any variable not assigned a value in the list shall have the default value, if any, assigned in the function specification, or the default value for the associated data type.

In feature 2 of Table 24 (non-formal call), the argument list shall contain exactly the same number of arguments, in exactly the same order and of the same data types as given in the function definition, except the execution control arguments EN and ENO.

Table 24 - Textual call of functions for formal and non-formal argument list

No.	Feature				Example
	Invocation type	Variable assignment	Variable order	Number of variables	In Structured Text (ST) language
1	formal	yes	any	any	A := LIMIT (EN := COND, IN := B, MX := 5, ENO => TEMPL);
2[a]	non-formal	no	fixed	fixed	A := LIMIT (1, B, 5);

NOTE 1 In the example given in feature 1, the MN variable will have the default value 0 (zero).

NOTE 2 The example given in feature 2 is semantically equivalent to the following call with formal variable assignments (feature 1):

```
A := LIMIT (EN := TRUE, MN := 1, IN := B, MX := 5);
```

[a] Feature 2 is **required** for call of any of the standard functions without formal names for one or more input variables, but feature 1 shall be used if EN/ENO is necessary in function calls.

6.5.2.3 Execution control using EN and ENO

As shown in table 20, an additional Boolean EN (Enable) input or ENO (Enable Out) output, or both, can be provided by the manufacturer or 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 function shall be controlled according to the following rules:

1. If the value of EN is FALSE (0) when the function is called, the operations defined by the function body shall not be executed and the value of ENO shall be reset to FALSE (0) by the programmable controller system.
2. Otherwise, the value of ENO shall be set to TRUE (1) by the programmable controller system, and the operations defined by the function body shall be executed. These operations can include the assignment of a Boolean value to ENO.
3. If any of the errors defined in the table in Annex E for subclauses of 6.5.2.6 occurs during the execution of one of the standard functions, the ENO output of that function shall be reset to FALSE (0) by the programmable controller system, or the manufacturer shall specify other disposition of such an **error** according to the provisions of 5.1.
4. If the ENO output is evaluated to FALSE(0), the values of all function outputs (VAR_OUTPUT, VAR_IN_OUT and function result) shall be considered to be **implementation-dependent**.

NOTE It is a consequence of these rules that the ENO output of a function must be explicitly examined by the calling entity if necessary to account for possible error conditions.

Table 25 - Use of EN input and ENO output

<table>
<tr><th>No.</th><th>Feature</th><th>Example[a]</th></tr>
<tr><td>1</td><td>Use of EN and ENO
Shown in LD (Ladder Diagram)</td><td><pre>
          +--------+ |
| ADD_EN | + | ADD_OK |
+---| |---|EN   ENO|---( )---+
A---		---C
B---		
          +--------+ |
</pre></td></tr>
<tr><td>2</td><td>Usage without EN and ENO
Shown in FBD (Function Block Diagram)</td><td><pre>
    +-----+
A---| + |---C
B---| |
    +-----+
</pre></td></tr>
<tr><td>3</td><td>Usage with EN and without ENO
Shown in FBD (Function Block Diagram)</td><td><pre>
         +-----+
ADD_EN---|EN |
     A---| + |---C
     B---| |
         +-----+
</pre></td></tr>
<tr><td>4</td><td>Usage without EN and with ENO
Shown in FBD (Function Block Diagram)</td><td><pre>
      +-------+
      | ENO|--- ADD_OK
A ---| + |--- C
B ---| |
      +-------+
</pre></td></tr>
<tr><td colspan="3">[a] The graphical languages chosen for demonstrating the features above are given only as examples. An implementer may specify that a feature of this table is supported in all languages, or in a particular language. When a feature is supported in a particular language, an appropriate suffix shall be employed, namely f, l, i, or s for FBD, LD, IL or ST languages. For instance, the first example given above could be for feature 1 or 1l; the second example could be for feature 2 or 2f, etc.</td></tr>
</table>

6.5.2.4 Declaration

Features for the textual and graphical declaration of functions are listed in Table 26.

As illustrated in Figure 6, the textual declaration of a function shall consist of the following elements:

1. The keyword FUNCTION, followed by an identifier specifying the name of the function being declared
2. If a function result is available a colon ‘:‘, and the data type of the value to be returned by the function block or if no function block result is available nothing or the keyword ‘VOID’;
3. A VAR_INPUT...END_VAR construct specifying the names and types of the function's input variables;
4. VAR_IN_OUT...END_VAR and VAR_OUTPUT...END_VAR constructs if required, specifying the names and types of the function's in-out and output variables;
5. A VAR_EXTERNAL...END_VAR construct, if required, specifying the names and types of the function's external variables;
6. A VAR...END_VAR construct, if required, specifying the names and types of the function's internal variables;
7. A *function body*, written in one of the languages defined in this standard, or another programming language, which specifies the operations to be performed upon the variable(s) in order to assign values dependent on the function's semantics to its in-out, output or external variables and in the case that a function result exists to a variable with the same name as the function, which represents the function result to be returned by the function (function result);
8. The terminating keyword END_FUNCTION.

If the generic data types given in Figure 4 are used in the declaration of standard function variables, then the rules for inferring the actual types of the arguments of such functions shall be part of the function definition.

The variable initialization constructs defined in 6.4.3 can be used for the declaration of default values of function inputs and initial values of their internal and output variables.

The values of variables which are passed to the function via a `VAR_IN_OUT` construct can be modified from within the function.

Table 26 - Function features

No.	Description	Example
1	In-out variable declaration (textual)	`VAR_IN_OUT A: INT; END_VAR`
2	In-out variable declaration (graphical)	See Figure 6 b)
3	Graphical connection of in-out variable to different variables (graphical)	See Figure 6 d)

The graphic declaration of a function shall consist of the following elements:

1. The bracketing keywords `FUNCTION...END_FUNCTION` or a graphical equivalent.
2. A graphic specification of the function name and the names, types and possibly initial values of the function's result and variables (input, output and in-out).
3. A specification of the names, types and possibly initial values of the internal variables used in the function, for example, using the `VAR...END_VAR` construct.
4. A function body as defined above.

The maximum number of function specifications allowed in a particular *resource* is an **implementation dependency**.

EXAMPLE see Figure 6.

```
FUNCTION SIMPLE_FUN : REAL
 VAR_INPUT
  A, B : REAL;          (* External interface specification *)
  C :    REAL := 1.0;
 END_VAR

 VAR_IN_OUT COUNT        : INT; END_VAR
  VAR COUNTP1      : INT; END_VAR
  COUNTP1     := ADD(COUNT, 1);  (*Function body specification *)
  COUNT       := COUNTP1;
  SIMPLE_FUN  := A*B/C;
END_FUNCTION
```

a) Textual declaration in ST language

```
 FUNCTION
* External interface specification *)
             +-------------+
             | SIMPLE_FUN  |
     REAL----|A            |----REAL
     REAL----|B            |
     REAL----|C            |
     INT-----|COUNT---COUNT|----INT
             +-------------+
* Function body specification *)
            +---+
            |ADD|---         +----+
     COUNT--|   |---COUNTP1--| := |---COUNT
         1--|   |            +----+
            +---+    +---+
                 A---| * |   +---+
                 B---|   |---| / |---SIMPLE_FUN
                     +---+   |   |
                 C-----------|   |
                             +---+
 END FUNCTION
```

b) Graphical declaration in FBD language

```
VAR X, Y, Z, RESULT : REAL;
VAR COUNT1, COUNT2   : INT;
 …
RESULT := Simple_FUN (A:= X, B:=Y, C:=Z, COUNT:= COUNT1);
COUNT2 := COUNT1;
              ...
```

c) Usage of a function in ST language

```
         +-------------+
         | SIMPLE_FUN  |
    X----|A            |----RESULT
    Y----|B            |
    Z----|C            |
COUNT1---|COUNT---COUNT|----COUNT2
         +-------------+
```

d) Usage of a function in FBD language

```
VAR_GLOBAL DataArray: ARRAY [0..1000] OF INT; END_VAR

FUNCTION SPECIAL_FUN : VOID
VAR_INPUT
  FirstIndex :    INT;
  LastIndex  :    INT END_VAR
VAR_OUTPUT
  Sum : INT END_VAR
VAR_EXTERNAL DataArray: ARRAY [0..1000] OF INT; END_VAR
VAR i: INT END_VAR

for I := FirstIndex to LastIndex do
  Sum := Sum+DataArray[i]
end_for
END_FUNCTION
```

e) Textual declaration of a function with no function result and an external variable

```
        +-----------------+
        |   SPECIAL_FUN   |
10-----|FirstIndex     Sum|-----Result
20-----|LastIndex         |
        +-----------------+
```

f) Usage of a function with no function result and an external variable

NOTE 1 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.

NOTE 2 The effect of the functional call in d) is identical to that in c).

Figure 6 - Examples of function declarations and usage

6.5.2.5 Typing, overloading, and type conversion

6.5.2.5.1 Generic Overloading

A standard function, function block type, operator, or instruction is said to be *overloaded* when it can operate on input data elements of various types within a generic type designator as defined in 6.3.3.

EXAMPLE 1

> An overloaded addition function on generic type `ANY_NUM` can operate on data of types `LREAL`, `REAL`, `DINT`, `INT`, and `SINT`.

When a programmable controller system supports an overloaded standard function, function block type, operator, or instruction, this standard function, function block type, operator, or instruction shall apply to all data types of the given generic type which are supported by that system.

EXAMPLE 2

> If a programmable controller system supports the overloaded function `ADD` and the data types `SINT`, `INT`, and `REAL`, then the system supports the `ADD` function on inputs of type `SINT`, `INT`, and `REAL`.

6.5.2.5.2 Typed overloading

When a function which normally represents an overloaded operator is to be typed, i.e., the types of its inputs and outputs are restricted to a particular elementary or derived data type. This shall be done by appending an “underline” character followed by the required type, as shown in Table 27.

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 “underline” character.

Table 27 - Typed and overloaded functions

No.	Feature	Example

1a
 	
1b
	Overloaded functions	`+-----+`
		`| ADD |`
		`ANY_NUM-----|     |----ANY_NUM`
		`ANY_NUM-----|     |`
		`.   -----|     |`
		`.   -----|     |`
		`ANY_NUM-----|     |`
		`+-----+`
		
		`+----------+`
		`ANY_ELEMENTARY -----|  TO_INT  |----INT`
		`+----------+`

2a [a]
 	
2b [a]
	Typed functions	`+---------+`
		`| ADD_INT |`
		`INT-----|         |----INT`
		`INT-----|         |`
		`.  -----|         |`
		`.  -----|         |`
		`INT-----|         |`
		`+---------+`
		
		`+---------------+`
		`WORD -----|  WORD_TO_INT  |----INT`
		`+---------------+`

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

6.5.2.5.3 Type conversion

Explicit (overloaded or typed) conversion and implicit type conversion can be used to adapt data types for the use in expressions and as parameters.

The following table shows which type conversions shall be supported implicitly and explicitly.

Table 28 - Type Conversion

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

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

Legend

▨	no data type conversion necessary
-	no implicit or explicit data type conversion defined by this standard, the implementation may support additional data type conversions
i	Also implicit data type conversion additional to explicit type conversion allowed
e	Explicit data type conversion programmed by the user (as a standard functionality) necessary because of loss of accuracy, mismatch in the range or possible implementation dependent behaviour.

6.5.2.5.4 Explicitly typed or overloaded type conversion

When a programmable controller system requires explicit type conversion for overloaded functions then all input and output variables must be of the same type. More specifically, when the type of the result of a standard function defined in 6.5.2.6 is generic, the actual types of all input variables of the same generic type shall be of the same type as the actual type of the function value in a given call of the function. If necessary, the type conversion functions can be used to meet this requirement.

Explicit type conversion shall keep the value and accuracy of the source data type if the value fits into the target data type. The manufacturer shall define the result if the target data type cannot provide the same value as the source data type.

<table>
<tr><td>VAR
A : INT ;
B : INT ;
C : INT ;
END_VAR</td><td>+---+
A---| + |---C
B---| |
+---+
C := A+B;</td><td></td></tr>
<tr><td colspan="2">NOTE Type conversion is not required in the example shown above.</td><td></td></tr>
<tr><td>VAR
A : INT ;
B : REAL ;
C : REAL;
END_VAR</td><td>+-----------+   +---+
A---|INT_TO_REAL|---| + |---C
+-----------+ | |
B------------------| |
+---+
C := INT_TO_REAL(A)+B;</td><td>+--------+   +---+
A---|TO_REAL|---|ADD|---C
+--------+ | |
B---------------| |
+---+
C := TO_REAL(A) + B;</td></tr>
<tr><td>VAR
A : INT ;
B : INT ;
C : REAL;
END_VAR</td><td>+---+   +-----------+
A----| + |---|INT_TO_REAL|---C
B----| | +-----------+
+---+
C := INT_TO_REAL(A+B);</td><td>+---+   +--------+
A---|ADD|---|TO_REAL|---C
B---| | +--------+
+---+
C := TO_REAL(A+B);</td></tr>
<tr><td><b>a) Type declaration
(ST language)</b></td><td colspan="2"><b>b) Usage (FBD language and ST language)</b></td></tr>
</table>

Figure 7 - Typed and overloaded functions (Example)

<table>
<tr><td>VAR
A : INT ;
B : INT ;
C : INT ;
END_VAR</td><td>+---------+
A---| ADD_INT |---C
B---| |
+---------+
C := ADD_INT(A,B);</td></tr>
<tr><td colspan="2">NOTE Type conversion is not required in the example shown above.</td></tr>
<tr><td>VAR
A : INT ;
B : REAL ;
C : REAL;
END_VAR</td><td>+-----------+   +----------+
A---|INT_TO_REAL|---| ADD_REAL |---C
+-----------+ | |
B-------------------| |
+----------+
C := ADD_REAL(INT_TO_REAL(A),B);</td></tr>
<tr><td>VAR
A : INT ;
B : INT ;
C : REAL;
END_VAR</td><td>+---------+   +-----------+
A---| ADD_INT |---|INT_TO_REAL|---C
| | +-----------+
B---| |
+---------+
C := INT_TO_REAL(ADD_INT(A,B));</td></tr>
<tr><td><b>a) Type declaration
(ST language)</b></td><td><b>b) Usage (FBD language and ST language)</b></td></tr>
</table>

Figure 8 - Explicit typed type conversion functions with typed functions Example)

6.5.2.5.5 Implicit and explicit type conversion

When a programmable controller system supports implicit type conversion for overloaded functions then input and output variables can be of different types.

Implicit type conversion shall keep the value and accuracy of the data types. Otherwise the user can use explicit type conversion. Offering this to the user reassures him that his program will work as expected while saving him time in programming along with some screen real estate.

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

a) Type declaration

Explicit Type Conversion

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

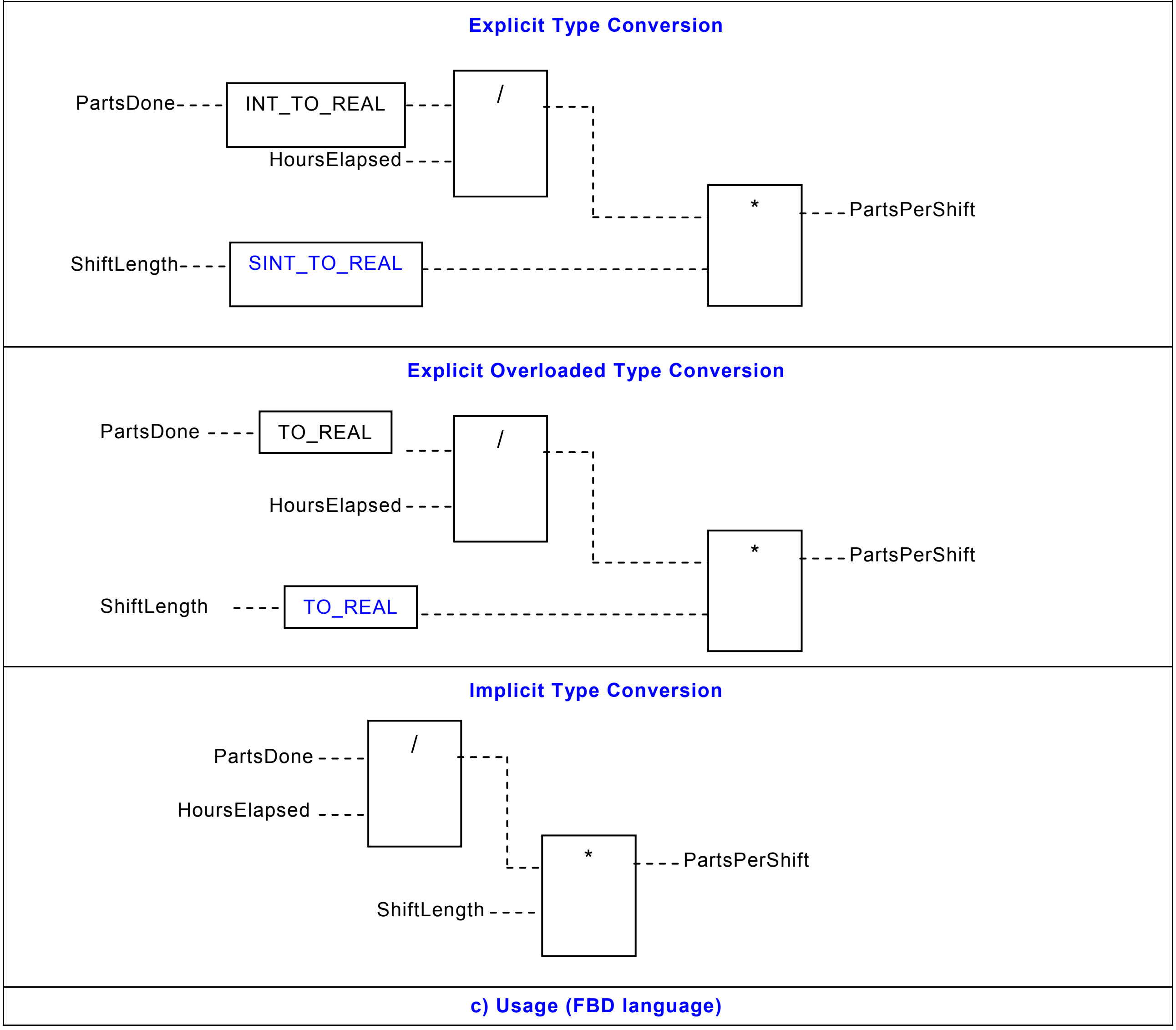
Explicit Overloaded Type Conversion

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

Implicit Type Conversion

```
PartsRatePerHr := PartsDone / HoursElapsed;
PartsPerShift  := ShiftLength * PartsRatePerHr;
```

b) Usage (ST language)



c) Usage (FBD language)

Figure 9 - Explicit vs. Implicite type conversion

6.5.2.6 Standard functions

6.5.2.6.1 General

Definitions of functions common to all programmable controller programming languages are given in this subclause. Where graphical representations of standard functions are shown in this subclause, equivalent textual declarations may be written as specified in 6.5.2.6.

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 as its output the sum of all its inputs. The maximum number of inputs of an extensible function is an **implementation dependency**. 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;`

6.5.2.6.2 Type conversion functions

As shown in Table 29, 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 **implementation dependencies**.

Table 29 - Type conversion function features

No.		Graphical form	Usage example

1 [a, b, e]	Typed	`+-----------+`
		`* ---| *_TO_**  |--- **`
		`+-----------+`
		(*) - Input data type, e.g., INT
		(**) - Output data type, e.g., REAL
		(*_TO_**) - Function name, e.g., INT_TO_REAL
			`A := INT_TO_REAL(B) ;`

	Overloaded	`+-----------+`
		`* ---|   TO_**  |--- **`
		`+-----------+`
		(*) - Input data type, e.g., INT
		(**) - Output data type, e.g., REAL
		(TO_**) - Function name, e.g., TO_REAL
			`A := TO_REAL(B) ;`

2 [c]	“old” over-loaded	`+-----------+`
		`ANY_REAL---|   TRUNC   |---ANY_INT`
		`+-----------+`
			Deprecated
			(from 2nd edition)

	Typed	`+-----------+`
		`ANY_REAL---|*_TRUNC_**|---ANY_INT`
		`+-----------+`
		(*) - Input data type ANY_REAL
		(**) - Output data type ANY_INT
		(*_TRUNC_**) - Function name, e.g., REAL_TRUNC_INT
			`A := REAL_TRUNC_INT(B) ;`

	Overloaded	`+-----------+`
		`ANY_REAL---| TRUNC_**  |---ANY_INT`
		`+-----------+`
			`A := TRUNC_INT(B) ;`

3 [d]	Typed	`+------------+`
		`* ---|*_BCD_TO_**|--- **`
		`+------------+`
			`A := WORD_BCD_TO_INT(B);`

	Over-loaded	`+-------------+`
		`* ----|  BCD_TO_**   |--- **`
		`+-------------+`
			`A := BCD_TO_INT(B);`

4 [d]	Typed	`+-------------+`
		`** ----|  **_BCD_*   |--- *`
		`+-------------+`
			`A := INT_TO_BCD_WORD(B);`

	Overloaded	`+-------------+`
		`* ----|  TO_BCD_**   |--- **`
		`+-------------+`
			`A := TO_BCD_WORD(B);`

NOTE 1 Usage examples are given in the ST language.

NOTE 2 Conversion of bitstring data type shall be done in that way that from
(a) a short data type to a longer one leading zeros shall be added or
(b) long data type to a shorter one this is implementation dependent.

[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

[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.

[c] The function TRUNC_* shall be 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.

[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 6.2.2, in the character set defined in 6.1.1.

6.5.2.6.3 Numerical 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 30. 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 31. 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 **implementation 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 30 - Standard functions of one numeric variable

<table>
<tr><th colspan="3">Graphical form</th><th>Usage example in ST</th></tr>
<tr><td colspan="3">`+---------+`
`* ---|   **    |--- *`
`+---------+`
(*) - Input/Output (I/O) type
(**) - Function name</td><td>`A := SIN(B);`
(ST language)</td></tr>
<tr><th>No.</th><th>Function name</th><th>I/O type</th><th>Description</th></tr>
<tr><td colspan="4">General functions</td></tr>
<tr><td>1</td><td>ABS</td><td>ANY_NUM</td><td>Absolute value</td></tr>
<tr><td>2</td><td>SQRT</td><td>ANY_REAL</td><td>Square root</td></tr>
<tr><td colspan="4">Logarithmic functions</td></tr>
<tr><td>3</td><td>LN</td><td>ANY_REAL</td><td>Natural logarithm</td></tr>
<tr><td>4</td><td>LOG</td><td>ANY_REAL</td><td>Logarithm base 10</td></tr>
<tr><td>5</td><td>EXP</td><td>ANY_REAL</td><td>Natural exponential</td></tr>
<tr><td colspan="4">Trigonometric functions</td></tr>
<tr><td>6</td><td>SIN</td><td>ANY_REAL</td><td>Sine of input in radians</td></tr>
<tr><td>7</td><td>COS</td><td>ANY_REAL</td><td>Cosine in radians</td></tr>
<tr><td>8</td><td>TAN</td><td>ANY_REAL</td><td>Tangent in radians</td></tr>
<tr><td>9</td><td>ASIN</td><td>ANY_REAL</td><td>Principal arc sine</td></tr>
<tr><td>10</td><td>ACOS</td><td>ANY_REAL</td><td>Principal arc cosine</td></tr>
<tr><td>11</td><td>ATAN</td><td>ANY_REAL</td><td>Principal arc tangent</td></tr>
</table>

Table 31 - Standard arithmetic functions

Graphical form	Usage example in ST

+-----+
ANY_NUM ---| *** |--- ANY_NUM
ANY_NUM ---| |
. ---| |
. ---| |
ANY_NUM ---| |
+-----+
(***) - Name or Symbol
	A := ADD(B,C,D) ;
	or
	A := B+C+D ;

No. [a,b]	Name	Symbol	Description
Extensible arithmetic functions			
1 [c]	ADD	+	OUT := IN1 + IN2 +... + INn
2	MUL	*	OUT := IN1 * IN2 *... * INn
Non-extensible arithmetic functions			
3 [c]	SUB	-	OUT := IN1 - IN2
4 [d]	DIV	/	OUT := IN1 / IN2
5 [e]	MOD		OUT := IN1 modulo IN2
6 [f]	EXPT	**	Exponentiation: OUT := $\mathrm{IN1}^{\mathrm{IN2}}$
7 [g]	MOVE	:=	OUT := IN

NOTE 1 Non-blank entries in the **Symbol** column are suitable for use as operators in textual languages, as shown in Table 60 and Table 63.

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.

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

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

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

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

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

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

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

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

6.5.2.6.4 Bit string functions

The standard graphical representation, function names and descriptions of shift functions for a single bit-string variable shall be as defined in Table 32. 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 bit-wise Boolean functions shall be as defined in Table 33. These functions shall be extensible, except for NOT, and overloaded on all bit-string types, and can be typed.

Table 32 - Standard bit shift functions

Graphical form	Usage example [a]

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

No.	Name	Description
1	SHL	`OUT := IN` left-shifted by `N` bits, zero-filled on right
2	SHR	`OUT := IN` right-shifted by `N` bits, zero-filled on left
3	ROR	`OUT := IN` right-rotated by `N` bits, circular
4	ROL	`OUT := IN` left-rotated by `N` bits, circular

NOTE 1 The notation `OUT` refers to the function output.

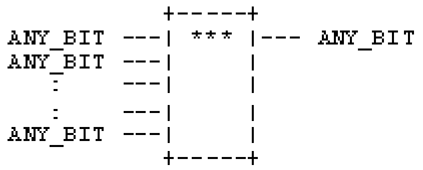
NOTE 2 Examples: `IN = 11001, N = 3`

```
      SHL(11001, 3) = 01000
      SHR(11001, 3) = 00011
      ROR(11001, 3) = 00111
      ROL(11001, 3) = 01110
```

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

Table 33 - Standard bitwise Boolean functions

Graphical form			Usage examples (NOTE 5)


(***) - Name or symbol
			A := AND(B,C,D) ;
			or
			A := B & C & D ;

No. [a,b]	Name	Symbol	Description (NOTE 3)
1	AND	& (NOTE 1)	OUT := IN1 & IN2 &... & INn
2	OR	>=1 (NOTE 2)	OUT := IN1 OR IN2 OR... OR INn
3	XOR	=2k+1 (NOTE 2)	OUT := IN1 XOR IN2 XOR... XOR INn
4	NOT		OUT := NOT IN1 (NOTE 4)

NOTE 1 This symbol is suitable for use as an operator in textual languages, as shown in Table 60 and Table 63.

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 as shown in Table 22.

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

[a] When the ***named*** representation of a function is supported, this shall be indicated by the suffix “n” in the compliance statement. For example, “5n” represents the notation “AND”.

[b] When the ***symbolic*** representation of a function is supported, this shall be indicated by the suffix “s” in the compliance statement. For example, “5s” represents the notation “&”.

1830 6.5.2.6.5 Selection and comparison functions

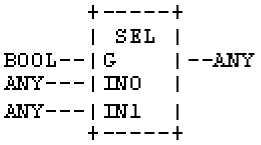
Selection and comparison functions shall be overloaded on all data types. The standard graphical representations, function names and descriptions of selection functions shall be as shown in Table 34.

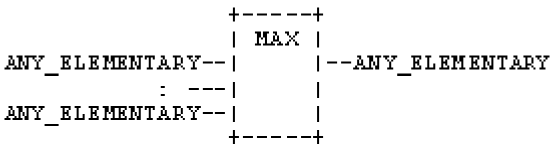
The standard graphical representation, function names and symbols, and descriptions of comparison functions shall be as defined in Table 35. All comparison functions (except NE) shall be extensible.

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.

1841 Table 34 - Standard selection functions[d]

No.	Graphical form	Explanation/example

1		Binary selection [c]:
		OUT := IN0 if G = 0
		OUT := IN1 if G = 1
		EXAMPLE:
		A := SEL (G:=0, IN0:=X, IN1:=5);

2		Extensible maximum function:
		OUT := MAX (IN1, IN2,..., INn);
		EXAMPLE:
		A := MAX(B, C ,D);

3	`              +-----+`
	`              | MIN |`
	`ANY_ELEMENTARY--|     |--ANY_ELEMENTARY`
	`      :     ---|     |`
	`ANY_ELEMENTARY--|     |`
	`              +-----+`
		Extensible minimum function:
		`OUT := MIN (IN1, IN2,..., Nn)`
		EXAMPLE:
		`A := MIN(B, C, D);`

4	`               +-------+`
	`               | LIMIT |`
	`ANY_ELEMENTARY--|MN     |--ANY_ELEMENTARY`
	`ANY_ELEMENTARY--|IN     |`
	`ANY_ELEMENTARY--|MX     |`
	`               +-------+`
		Limiter:
		`OUT := MIN(MAX(IN,MN),MX);`
		EXAMPLE:
		`A := LIMIT(IN := B, MN:=0,`
		`      MX := 5);`

5[e]	`         +-----+`
	`         | MUX |`
	`ANY_INT--|K    |----ANY`
	`ANY------|     |`
	` : ------|     |`
	`ANY------|     |`
	`         +-----+`
		Extensible multiplexer [a, b, c]:
		Select one of `N` inputs depending on input `K`
		EXAMPLE:
		`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 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.

[b] 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.

[c] It is allowed, but not required, that the manufacturer support selection among variables of *derived data types,* as defined in 6.3.3, in order to claim compliance with this feature.

[d] It is an **error** if the inputs and the outputs to one of these functions are not all of the same actual data type, with the exception of the `G` input of the `SEL` function and the `K` input of the `MUX` function.

[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 35 - Standard comparison functions

<table>
<tr><th colspan="3">Graphical form</th><th>Usage examples</th></tr>
<tr><td colspan="3">   +-----+
ANY_ELEMENTARY --| *** |--- BOOL
       :   --| |
ANY_ELEMENTARY --| |
                +-----+
(***) - Name or Symbol</td><td>A := GT(B,C,D);
or
A := (B>C) & (C>D);</td></tr>
<tr><th>No.</th><th>Name [a]</th><th>Symbol [b]</th><th>Description</th></tr>
<tr><td>1</td><td>GT</td><td>></td><td>Decreasing sequence:
OUT := (IN1>IN2)& (IN2>IN3) &... & (INn-1 > INn)</td></tr>
<tr><td>2</td><td>GE</td><td>>=</td><td>Monotonic sequence:
OUT := (IN1>=IN2)& (IN2>=IN3)&... & (INn-1 >= INn)</td></tr>
<tr><td>3</td><td>EQ</td><td>=</td><td>Equality:
OUT := (IN1=IN2)& (IN2=IN3) &... & (INn-1 = INn)</td></tr>
<tr><td>4</td><td>LE</td><td><=</td><td>Monotonic sequence:
OUT := (IN1<=IN2)& (IN2<=IN3)&... & (INn-1 <= INn)</td></tr>
<tr><td>5</td><td>LT</td><td><</td><td>Increasing sequence:
OUT := (IN1<IN2)& (IN2<IN3) &... & (INn-1 < INn)</td></tr>
<tr><td>6</td><td>NE</td><td><></td><td>Inequality (non-extensible):
OUT := (IN1<>IN2)</td></tr>
</table>

NOTE 1 The notations IN1, IN2,..., INn refer to the inputs in top-to-bottom order; OUT refers to the output.

NOTE 2 All the symbols shown in this table are suitable for use as operators in textual languages, as shown in Table 60 and Table 63.

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 as shown in Table 69

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

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

6.5.2.6.6 Character string functions

All the functions defined in 6.5.2.6.5 shall be applicable to character strings. 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 defined in 6.1.1.

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 36. 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 36 is less than zero;
- 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 implementation-dependent maximum string length.

Table 36 - Standard character string functions

No.	Graphical form	Explanation/example

1	`            +-----+`
	`ANY_STRING--| LEN |--ANY_INT`
	`            +-----+`
		String length function
		EXAMPLE
		`A := LEN('ASTRING');`
		is equivalent to `A := 7;`

2	`            +------+`
	`            | LEFT |`
	`ANY_STRING--|IN    |--ANY_STRING`
	`ANY_INT-----|L     |`
	`            +------+`
		Leftmost `L` characters of `IN`
		EXAMPLE
		`A := LEFT(IN:='ASTR', L:=3);`
		is equivalent to `A := 'AST';`

3	`            +-------+`
	`            | RIGHT |`
	`ANY_STRING--|IN     |--ANY_STRING`
	`ANY_INT-----|L      |`
	`            +-------+`
		Rightmost `L` characters of `IN`
		EXAMPLE
		`A := RIGHT(IN:='ASTR', L:=3);`
		is equivalent to `A := 'STR';`

4	`            +-------+`
	`            |  MID  |`
	`ANY_STRING--|IN     |--ANY_STRING`
	`ANY_INT-----|L      |`
	`ANY_INT-----|P      |`
	`            +-------+`
		`L` characters of `IN`,
		beginning at the `P`-th
		EXAMPLE
		`A := MID(IN:='ASTR', L:=2, P:=2);`
		is equivalent to `A := 'ST';`

5	`             +--------+`
	`             | CONCAT |`
	`ANY_STRING---|        |--ANY_STRING`
	`      :   ---|        |`
	`ANY_STRING---|        |`
	`             +--------+`
		Extensible concatenation
		EXAMPLE
		`A := CONCAT('AB','CD','E');`
		is equivalent to `A := 'ABCDE';`

6	`            +--------+`
	`            | INSERT |`
	`ANY_STRING--|IN1     |--ANY_STRING`
	`ANY_STRING--|IN2     |`
	`ANY_INT-----|P       |`
	`            +--------+`
		Insert `IN2` into `IN1` after the
		`P`-th character position
		EXAMPLE
		`A:=INSERT(IN1:='ABC', IN2:='XY', P=2);`
		is equivalent to `A := 'ABXYC' ;`

7	`            +--------+`
	`            | DELETE |`
	`ANY_STRING--|IN      |--ANY_STRING`
	`ANY_INT-----|L       |`
	`ANY_INT-----|P       |`
	`            +--------+`
		Delete `L` characters of `IN`, beginning
		at the `P`-th character position
		EXAMPLE
		`A := DELETE(IN:='ABXYC', L:=2, P:=3);`
		is equivalent to `A := 'ABC';`

8	`            +---------+`
	`            | REPLACE |`
	`ANY_STRING--|IN1      |--ANY_STRING`
	`ANY_STRING--|IN2      |`
	`ANY_INT-----|L        |`
	`ANY_INT-----|P        |`
	`            +---------+`
		Replace `L` characters of `IN1` by `IN2`,
		starting at the `P`-th character position
		EXAMPLE
		`A := REPLACE(IN1:='ABCDE', IN2:='X',`
		`L:=2, P:=3);`
		is equivalent to `A := 'ABXE';`

9	`            +--------+`
	`            |  FIND  |`
	`ANY_STRING--|IN1     |--ANY_INT`
	`ANY_STRING--|IN2     |`
	`            +--------+`
		Find the character position of the beginning of the first occurrence of `IN2` in `IN1`. If no occurrence of `IN2` is found, then `OUT := 0`.
		EXAMPLE
		`A := FIND(IN1:='ABCBC', IN2:='BC');`
		is equivalent to `A := 2;`

NOTE The examples in this table are given in the Structured Text (ST) language.		

6.5.2.6.7 Functions of time data types

In addition to the comparison and selection functions, the combinations of input and output time data types shown in Table 37 shall be allowed with the associated functions.

It shall be an **error** if the result of evaluating one of these functions exceeds the **implementation-dependent** range of values for the output data type.

Table 37 - Functions of time data types

No.	Name	Symbol	IN1	IN2	OUT
Numeric and concatenation functions					
1a[c,d]	ADD	+	TIME	TIME	TIME
1b[c,d]	ADD_TIME	+	TIME	TIME	TIME
2a	ADD[b]	+[b]	TIME_OF_DAY	TIME	TIME_OF_DAY
2b	ADD_TOD_TIME	+[b]	TIME_OF_DAY	TIME	TIME_OF_DAY
3a	ADD	+[b]	DATE_AND_TIME	TIME	DATE_AND_TIME
3b	ADD_DT_TIME	+[b]	DATE_AND_TIME	TIME	DATE_AND_TIME
4a[c,d]	SUB	-	TIME	TIME	TIME
4b[c,d]	SUB_TIME	-	TIME	TIME	TIME
5a	SUB[b]	-[b]	DATE	DATE	TIME
5b	SUB_DATE_DATE	-[b]	DATE	DATE	TIME
6a	SUB[b]	-[b]	TIME_OF_DAY	TIME	TIME_OF_DAY
6b	SUB_TOD_TIME	-[b]	TIME_OF_DAY	TIME	TIME_OF_DAY
7a	SUB[b]	-[b]	TIME_OF_DAY	TIME_OF_DAY	TIME
7b	SUB_TOD_TOD	-[b]	TIME_OF_DAY	TIME_OF_DAY	TIME
8a	SUB[b]	-[b]	DATE_AND_TIME	TIME	DATE_AND_TIME
8b	SUB_DT_TIME	-[b]	DATE_AND_TIME	TIME	DATE_AND_TIME
9a	SUB[b]	-[b]	DATE_AND_TIME	DATE_AND_TIME	TIME
9b	SUB_DT_DT	-[b]	DATE_AND_TIME	DATE_AND_TIME	TIME
10a	MUL[b]	*[b]	TIME	ANY_NUM	TIME
10b	MULTIME	*[b]	TIME	ANY_NUM	TIME
11a	DIV[b]	/[b]	TIME	ANY_NUM	TIME
11b	DIVTIME	/[b]	TIME	ANY_NUM	TIME
12	CONCAT_DATE_TOD		DATE	TIME_OF_DAY	DATE_AND_TIME
Type conversion functions					
13[a]	DT_TO_TOD				
14[a]	DT_TO_DATE				

NOTE 1 Non-blank entries in the **Symbol** column are suitable for use as operators in textual languages, as shown in tables 52 and 55.

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

NOTE 3 It is possible to type the functions MULTIME and DIVTIME, e.g., the operands of MULTIME_REAL would be of type TIME and REAL, respectively.

NOTE 4 The effects of conversion between time data types and types STRING and WSTRING are defined in footnote (e) Table 29.

NOTE 5 The effects of type conversions between time data types and other data types not defined in this table are implementation-**dependent**.

[a] The type conversion functions shall have the effect of “extracting” the appropriate data,
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 statements

```
        X := DT#1986-04-28-08:40:00 ;
        W := DATE#1986-04-28 ;
        Y := TIME_OF_DAY#08:40:00;
```

[b] This usage is **deprecated** and will not be included in future editions of this standard.

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

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

6.5.2.6.8 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 as defined in 6.3.2.

Table 38 - Functions of enumerated data types

No.	Name	Symbol	Feature No. in tables 27 and 28
1	SEL		1
2	MUX		4
3[a]	EQ	=	7
4[a]	NE	<>	10
NOTE The provisions of NOTES 1 and 2 of Table 35 apply to this table.			
[a] The provisions of footnotes a and b of Table 35 apply to this feature.			

6.5.3 Function blocks

6.5.3.1 General

For the purposes of programmable controller programming languages, a *function block* is a program organization unit (POU) which, when executed, yields no or exactly one data element, which is considered to be the *function block* ***result*** (like a function), and one or more values which are considered to be the function block outputs.

[Editor’s Note: Tbd: Syntax for optional FB result. (is a new feature)]

Multiple, named *instances* (copies) of a function block type can be created. Each instance shall have an associated identifier (the *instance name*), and a data structure containing its, if existing, function block result, its output and internal variables, and, depending on the **implementation**, values of or references to its input and in-out variables. All the values of the function block result, the output variables and the necessary internal variables of this data structure shall persist from one execution of the function block instance to the next; therefore, call of a function block instance with the same arguments (input variables) need not always yield the same output values.

Only the input and output variables and the function block *result* shall be accessible outside of an instance of a function block, i.e., the function block's internal variables shall be hidden from the user of the function block.

Execution of the operations of a function block instance shall be called as defined in 7 for textual languages (IL and ST), according to the rules of network evaluation given in 8 for graphic languages (LD and FBD), or under the control of sequential function chart (SFC) elements.

Any function block type which has already been declared can be used in the declaration of another function block type or program type as shown in Figure 3.

The scope of an instance of a function block shall be local to the program organization unit in which it is instantiated, unless it is declared to be global in a VAR_GLOBAL block as defined in 6.7.2.

As illustrated in 6.5.3.4, the instance name of a function block instance can be used as the input to a function or 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, as defined in 6.4.4.

The maximum number of function block types and instantiations for a given *resource* are **implementation dependencies**.

Object oriented extensions to the function block concept are specified as optional features in 6.5.4

6.5.3.2 Representation

As illustrated in Figure 10, an instance of a function block can be created *textually*, by declaring a data element using the declared function block type in a VAR...END_VAR construct, identically to the use of a structured data type, as defined in 6.4.4.

As further illustrated in Figure 10, an instance of a function block can be created *graphically*, by using a graphic representation of the function block, with the function block type name inside the block, and the instance name above the block, following the rules for representation of functions given in 6.5.2.2.

As shown in Figure 10, input and output variables of an instance of a function block can be represented as elements of structured data types as defined in 6.3.3.

If either of the two graphical negation features defined in Table 22 is supported for function blocks, it shall also be supported for functions as defined in 6.5.2, and vice versa.

```
           FF75
         +------+
         |  SR  |
%IX1---|S1   Q1|---%QX3
%IX2---|R       |
         +------+
```

```
VAR FF75: SR; END_VAR      (* Declaration   *)
 FF75(S1:=%IX1, R:=%IX2); (* call     *)
 %QX3 := FF75.Q1;          (* Assign Output *)
```

```
                  MyTon
                +-------+
   +----+       |  TON  |
a--| NE |---O|EN   ENO|--
b--|    | r--|IN     Q|O-out
   +----+  --|PT    ET|--
                +-------+
```

```
VAR a,b,r,out : BOOL;
    MyTon : TON; END_VAR
  MyTon(EN := NOT (a <> b),
        IN := r,
        NOT Q => out);
```

Function block with a result output

```
                  YourTon
     +-----+     +--------+
a---| NE  |---0|  TON   |---result
b---|     |     |EN   ENO|--
     +-----+ r--|IN     Q|0-out
                |PT    ET|--
                +--------+
```

a) Graphical (FBD language)

Function block with a result output

```
VAR
    a,b,r,out, result : BOOL;
    MyTon : TON;
END_VAR
result := MyTon(EN := NOT (a <> b),
                IN := r,
                NOT Q => out);
```

b) Textual (ST language)

Figure 10 - Function block instantiation examples

Assignment of a value to an output variable of a function block is not allowed except from within the function block. The assignment of a value to the input of a function block is permitted only as part of the call of the function block. Unassigned or unconnected inputs of a function block shall keep their initialized values or the values from the latest previous call, if any. Allowable usages of function block inputs and outputs are summarized in Table 39, using the function block FF75 of type SR shown in Figure 10. The examples are shown in the ST language.

It shall be an **error** if no value is specified for:

a) an in-out variable of a function block instance;

b) a function block instance used as an input variable of another function block instance.

Table 39 - Examples of function block I/O variable usage

Usage	Inside function block	Outside function block
Input read	`IF IN1 THEN...`	Not allowed (NOTES 1 and 2)
Input assignment	Not allowed (NOTE 1)	`FB_INST(IN1:=A, IN2:=B);`
Output read	`OUT := OUT AND NOT IN2;`	`C := FB_INST.OUT;`
Output assignment	`OUT := 1;`	Not Allowed (NOTE 1)
In-out read	`IF INOUT THEN...`	`IF FB1.INOUT THEN...`
In-out assignment	`INOUT := OUT OR IN1;` (NOTE 3)	`FB_INST(INOUT:=D);`

NOTE 1 Those usages listed as “not allowed” in this table could lead to implementation-dependent, unpredictable side effects.

NOTE 2 Reading and writing of input, output 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 As illustrated in 6.5.3.4, modification within the function block of a variable declared in a `VAR_IN_OUT` block is permitted.

6.5.3.3 Execution control using EN and ENO

As shown in Table 25 for functions, for function blocks an additional Boolean `EN` (Enable) input or `ENO` (Enable Out) output, or both, can also be provided by the manufacturer or 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 function block shall be controlled according to the following rules:

1. If the value of `EN` is `FALSE (0)` when the function block instance is called, the assignments of actual values to the function block inputs may or may not be made in an **imple-**

mentation-dependent fashion, the operations defined by the function block body shall not be executed and the value of ENO shall be reset to FALSE (0) by the programmable controller system.

2. Otherwise, the value of ENO shall be set to TRUE(1) by the programmable controller system, the assignments of actual values to the function block inputs shall be made and the operations defined by the function block body shall be executed. These operations can include the assignment of a Boolean value to ENO.

If the ENO output is evaluated to FALSE(0), the values of the function block outputs (VAR_OUTPUT) keep their states from the previous call.

NOTE It is a consequence of these rules that the ENO output of a function block must be explicitly examined by the calling entity if necessary to account for possible error conditions`.

EXAMPLE

Figure 11 illustrates the use of EN and ENO in association with the standard TP,TON and TOF blocks (represented by T**) defined in 6.5.3.5.5, and the CTU and CTD blocks (represented by CT*) defined in 6.5.3.5.4. In accordance with the above rules, a FALSE value of the EN input may be used to “freeze” the operation of the associated function block; that is, the output values do not change irrespective of changes in any of the other input values. When the EN input value becomes TRUE, normal operation of the function block may resume. The value of the ENO output is FALSE after each evaluation of the function block for which the EN input is FALSE. When EN is TRUE, a TRUE value of ENO reflects a normal evaluation of the block, and a FALSE value of ENO may be used to indicate an **implementation-dependent** error condition.

```
          +-------+                   +-------+
          |  T**  |                   |  CT*  |
 BOOL---|EN   ENO|---BOOL      BOOL---|EN   ENO|---BOOL
 BOOL---|IN     Q|---BOOL      BOOL--->CU     Q|---BOOL
 TIME---|PT    ET|---TIME      BOOL---|R     CV|---INT
          +-------+             INT---|PV      |
                                      +-------+
```

Figure 11 - Examples of usage of EN and ENO in function blocks

6.5.3.4 Declaration

As illustrated in Figure 11, a function block shall be declared textually or graphically in the same manner as defined for functions in 6.5.2.4, with the differences described below and summarized in Table 40:

1) The keyword FUNCTION_BLOCK, followed by an identifier specifying the name of the function being declared,

2) If a function block result is available a colon ‘:‘, and the data type of the value to be returned by the function block or if no function block result is available nothing or the keyword ‘VOID’;

3) A VAR_INPUT...END_VAR construct specifying the names and types of the function's input variables; In textual declarations, the R_EDGE and F_EDGE qualifiers can be used to indicate an edge-detection function on Boolean inputs. This shall cause the implicit declaration of a function block of type R_TRIG or F_TRIG, respectively, as defined in 6.5.3.5.3, to perform the required edge detection. For an example of this construction, see features 8a and 8b of Table 40 and the accompanying NOTE.

4) The construction illustrated in features 9a and 9b of Table 40 shall be used in graphical declarations for rising and falling edge detection. When the character set defined in 6.1.1 is used, the “greater than” (>) or “less than” (<) character shall be in line with the edge of the function block. When graphic or semigraphic representations are employed, the notation of IEC 60617-12 for dynamic inputs shall be used.

5) VAR_IN_OUT...END_VAR and VAR_OUTPUT...END_VAR constructs if required, specifying the names and types of the function's in-out and output variables;

6) EN/ENO inputs and outputs shall be declared and used as described in 6.5.2.3.

7) A VAR...END_VAR construct, if required, specifying the names and types of the function block's internal variables;

8) A VAR_TEMP...END_VAR construct, if required, specifying the names and types of the function block's internal variables;

9) A VAR_EXTERNAL...END_VAR construct, if required, specifying the names and types of the function block's temporary variables;

10) The RETAIN or NON_RETAIN qualifier defined in 6.4.4 can be used for internal and output variables of a function block, as shown in features 1, 2, and 3 in Table 40.

11) The asterisk notation (feature 10 in Table 15) can be used in the declaration of internal variables of a function block.

12) A *function block body*, written in one of the languages defined in this standard, or another programming language, which specifies the operations to be performed upon the variable(s) in order to assign values dependent on the function's semantics to its in-out, output or external variables and in the case that a function block result exists to a variable with the same name as the function block, which represents the function block result to be returned by the function block (function block result);

13) The values of variables which are passed to the function block via a VAR_EXTERNAL construct can be modified from within the function block, as shown in feature 10 of Table 40.

14) The output values of a 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, as shown in features 5, 6, and 7 Table 40.

15) 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, as shown in features 6 and 7 of Table 40.

16) If the generic data types given in Figure 4 are used in the 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. In textual calls of such function blocks assignments of the outputs to variables shall be made directly in the call statement (using the operator ‘=>‘).

As illustrated in Figure 14, only variables or function block instance names can be passed into a function block via the VAR_IN_OUT construct, i.e., function or function block outputs cannot be passed via this construction. This is to prevent the inadvertent modifications of such outputs. However, “cascading” of VAR_IN_OUT constructions is permitted, as illustrated in Figure 14 c).

```
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 in ST language

```
FUNCTION_BLOCK
(** External Interface **)
                 +----------------+
                 |    DEBOUNCE    |
          BOOL---|IN          OUT|---BOOL
          TIME---|DB_TIME   ET_OFF|---TIME
                 +----------------+
(** Function Block Body **)
               DB_ON        DB_FF
              +-----+      +-----+
              | TON |      | SR  |
 IN----+------|IN  Q|------|S1 Q1|---OUT
       |  +---|PT ET|   +--|R    |
       |  |   +-----+   |  +-----+
       |  |             |
       |  |    DB_OFF   |
       |  |   +-----+   |
       |  |   | TON |   |
       +--|--O|IN  Q|---+
 DB_TIME--+---|PT ET|---------------ET_OFF
              +-----+
 END_FUNCTION_BLOCK
```

b) Graphical declaration in FBD language

Figure 12 - Examples of function block declarations

The following table shows the function block declarations and the usage of the features.

Table 40 - Function block declaration and usage features

No.	Description	Example
1a	`RETAIN` qualifier on internal variables	`VAR RETAIN X : REAL; END_VAR`
1b	`NON_RETAIN` qualifier on internal variables	`VAR NON_RETAIN X : REAL; END_VAR`
2a	`RETAIN` qualifier on output variables	`VAR_OUTPUT RETAIN X : REAL; END_VAR`
2b	`RETAIN` qualifier on input variables	`VAR_INPUT RETAIN X : REAL; END_VAR`
2c	`NON_RETAIN` qualifier on output variables	`VAR_OUTPUT NON_RETAIN X : REAL; END_VAR`
2d	`NON_RETAIN` qualifier on input variables	`VAR_INPUT NON_RETAIN X : REAL; END_VAR`
3a	`RETAIN` qualifier on internal function blocks	`VAR RETAIN TMR1: TON; END_VAR`
3b	`NON_RETAIN` qualifier on internal function blocks	`VAR NON_RETAIN TMR1: TON; END_VAR`
4a	`VAR_IN_OUT` declaration (textual)	`VAR_IN_OUT A: INT; END_VAR`
4b	`VAR_IN_OUT` declaration and usage(graphical)	See Figure 14
4c	`VAR_IN_OUT` declaration with assignment to different variables (graphical)	See Figure 14 d

5a	Function block instance name as input (textual)	`VAR_INPUT I_TMR: TON; END_VAR`
		`EXPIRED := I_TMR.Q;   (* See NOTE 1 *)`

5b	Function block instance name as input (graphical)	See Figure 14 a

6a	Function block instance name as `VAR_IN_OUT` (textual)	`VAR_IN_OUT IO_TMR: TOF; END_VAR`
		`IO_TMR(IN:=A_VAR, PT:=T#10S);`
		`EXPIRED := IO_TMR.Q;   (*See NOTE 1 *)`

6b	Function block instance name as `VAR_IN_OUT` (graphical)	See Figure 14 b

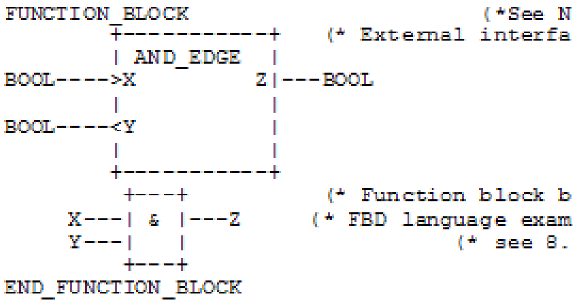
7a	Function block instance name as external variable (textual)	`VAR_EXTERNAL EX_TMR : TOF ;END_VAR`
		`EX_TMR(IN:=A_VAR, PT:=T#10S);`
		`EXPIRED := EX_TMR.Q;   (*See NOTE 1 *)`

7b	Function block instance name as external variable (graphical)	See Figure 14 c

No.	Description	Example

8a	Textual declaration of:
	- rising edge inputs
		`FUNCTION_BLOCK AND_EDGE           (*See NOTE 2 *)`
		`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`

8b	- falling edge inputs	

9a	Graphical declaration of:
	- rising edge inputs
		
		`FUNCTION_BLOCK                              (*See N`
		`       +------------+    (* External interfa`
		`       | AND_EDGE   |`
		`BOOL---->X          Z|---BOOL`
		`       |            |`
		`BOOL----<Y           |`
		`       |            |`
		`       +------------+`
		`        +---+            (* Function block b`
		`     X---| & |---Z      (* FBD language exam`
		`     Y---|   |                     (* see 8.`
		`        +---+`
		`END_FUNCTION_BLOCK`

9b	- falling edge inputs	
10a	`VAR_EXTERNAL` declarations within function block type declarations	
10b	`VAR_EXTERNAL CONSTANT` declarations within function block type declarations	
11	`VAR_TEMP` declarations (Table 18) within function block type declarations	

12a	Textual declaration of:
	`- function block result`
		`FUNCTION_BLOCK EDGES : BOOL`
		`VAR_INPUT X : BOOL; END_VAR`
		`VAR`
		`    X_TRG  : R_TRIG;`
		`    Y_TRIG : F_TRIG;`
		`END_VAR`
		` EDGES := X_TRIG.Q OR Y_TRIG.Q;`
		`END_FUNCTION_BLOCK`

12b	Graphical declaration of:
	`- function block result`
		`FUNCTION_BLOCK`
		`       +-----------+`
		`       |   EDGES   |`
		` BOOL --|X          |---BOOL`
		`       +-----------+`
		`VAR`
		`    X_TRG  : R_TRIG;`
		`    Y_TRIG : F_TRIG;`
		`END_VAR`
		`               +----+`
		`               | OR |`
		`     X_TRIG.Q---|    |---EDGES`
		`     Y_TRIG.Q---|    |`
		`               +----+`
		`END_FUNCTION_BLOCK`

12c	Textual usage of function block result	

	`VAR`
	`      Bool1  : BOOL;`
	`      Bool2  : BOOL;`
	`      EDGES1 : EDGES;`
	`      EDGES2 : EDGES;`
	`END_VAR`	

	Direct use in an expression	`IF (EDGES1(Bool1) OR EDGES2(Bool2))`
		`THEN   …`
		`END_IF;`

No.	Description	Example

	Explicit use	`EDGES1(Bool1);`
		`EDGES2(Bool1);`
		`IF (EDGES1.EDGES OR EDGES2.EDGES)`
		` THEN  …`
		`END_IF;`

NOTE 1 It is assumed in these examples that the variables `EXPIRED` and `A_VAR` have been declared of type `BOOL`.

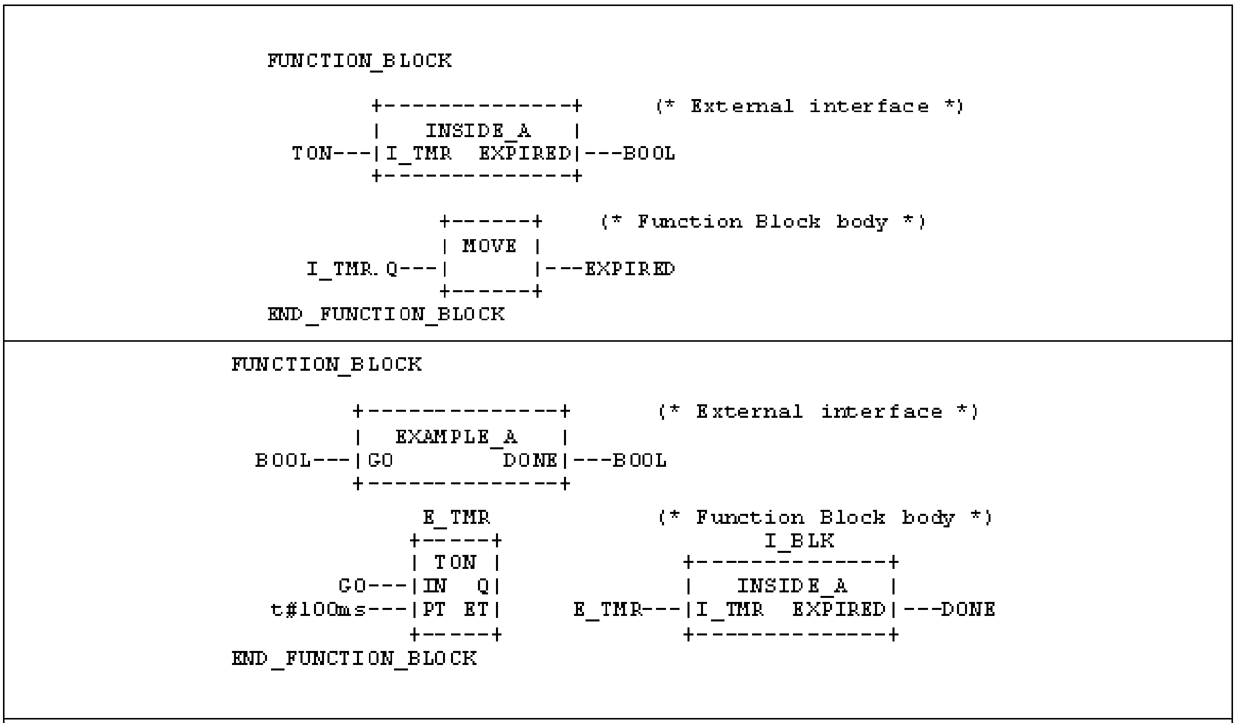
NOTE 2 The declaration of function block `AND_EDGE` in the above examples is equivalent to:

```
FUNCTION_BLOCK AND_EDGE
  VAR_INPUT
    X : BOOL;
    Y : BOOL;
  END_VAR
  VAR
    X_TRG : 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 42 for the definition of the edge detection function blocks `R_TRIG` and `F_TRIG`.

The following figurs shows the graphical use of function block names.



```
FUNCTION_BLOCK

         +--------------+      (* External interface *)
         |   INSIDE_A   |
   TON---|I_TMR  EXPIRED|---BOOL
         +--------------+

              +-------+     (* Function Block body *)
              | MOVE  |
     I_TMR.Q---|       |---EXPIRED
              +-------+
END_FUNCTION_BLOCK
```

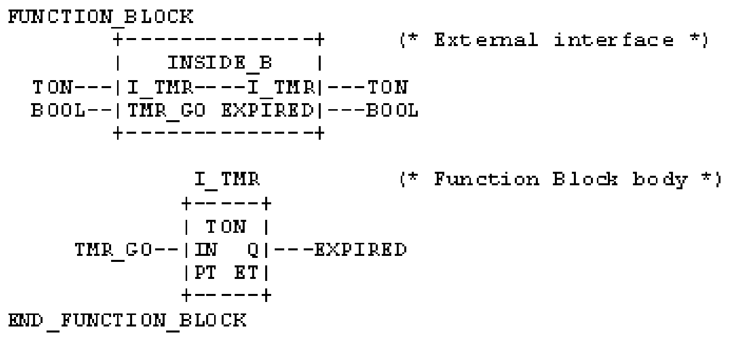
```
FUNCTION_BLOCK

         +--------------+       (* External interface *)
         |  EXAMPLE_A   |
   BOOL---|GO         DONE|---BOOL
         +--------------+

             E_TMR              (* Function Block body *)
            +-----+                  I_BLK
            | TON |             +--------------+
      GO---|IN   Q|             |   INSIDE_A   |
 t#100ms---|PT  ET|      E_TMR---|I_TMR  EXPIRED|---DONE
            +-----+             +--------------+
END_FUNCTION_BLOCK
```

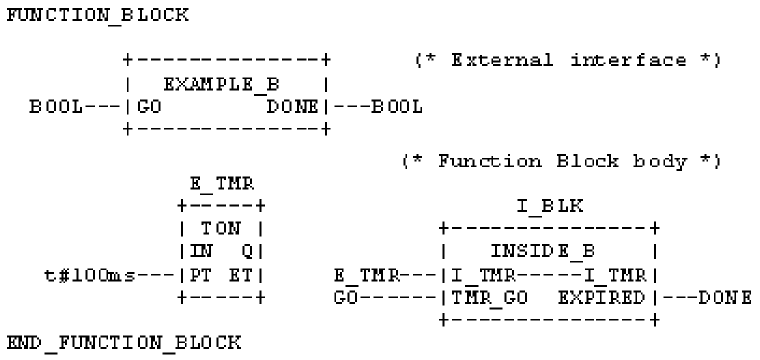
See Table 40, feature 5b (NOTE 1)

a) - Function block name as an input variable



```
FUNCTION_BLOCK
        +--------------+      (* External interface *)
        |   INSIDE_B   |
  TON---|I_TMR----I_TMR|---TON
  BOOL--|TMR_GO EXPIRED|---BOOL
        +--------------+

              I_TMR            (* Function Block body *)
             +-----+
             | TON |
     TMR_GO--|IN   Q|---EXPIRED
             |PT  ET|
             +-----+
END_FUNCTION_BLOCK
```

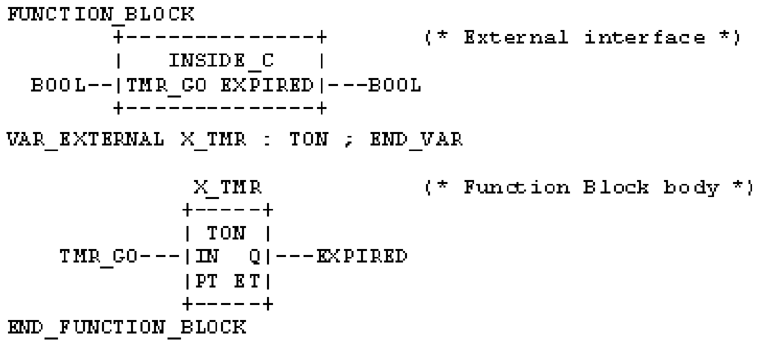


```
FUNCTION_BLOCK

         +--------------+      (* External interface *)
         |  EXAMPLE_B   |
  BOOL---|GO        DONE|---BOOL
         +--------------+
                              (* Function Block body *)
              E_TMR
             +-----+                  I_BLK
             | TON |             +---------------+
             |IN  Q|             |   INSIDE_B    |
   t#100ms---|PT  ET|    E_TMR---|I_TMR-----I_TMR|
             +-----+     GO------|TMR_GO  EXPIRED|---DONE
                                 +---------------+
END_FUNCTION_BLOCK
```

See Table 40, feature 6b

b) Function block name as an in-out variable



```
FUNCTION_BLOCK
        +--------------+       (* External interface *)
        |   INSIDE_C   |
  BOOL--|TMR_GO EXPIRED|---BOOL
        +--------------+
VAR_EXTERNAL X_TMR : TON ; END_VAR

              X_TMR            (* Function Block body *)
             +-----+
             | 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

                I_BLK            (* Program body *)
          +---------------+
          |    INSIDE_C    |
 GO------|TMR_GO   EXPIRED|---DONE
          +---------------+
END_PROGRAM
```

See Table 40, feature 7b (NOTE 2)

c) Function block name as an external variable

NOTE 1 I_TMR is here not represented graphically since this would imply call of I_TMR within INSIDE_A, which is forbidden by rules 4) and 5) of 6.5.3.4. See also feature 5 a) of Table 40.

NOTE 2 The PROGRAM declaration mechanism is defined in 2.5.3.

Figure 13 - Graphical use of function block names

The following figure shows the declaration and usage of in-out variables in function blocks.

```
          +--------+
          | ACCUM  |
   INT---|A-----A|---INT
   INT---|X       |
          +--------+
            +---+
        A---| + |---A
        X---|   |
            +---+
```

```
FUNCTION_BLOCK ACCUM
   VAR_IN_OUT A : INT ; END_VAR
   VAR_INPUT X : INT ; END_VAR
   A := A+X ;
END_FUNCTION_BLOCK
```

a) Graphical and textual declarations

```
                 ACC1
              +--------+
              | ACCUM  |
ACC-----------|A-----A|---ACC
      +---+   |        |
X1---| * |---|X       |
X2---|   |    +--------+
      +---+
```

A declaration such as

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

is assumed: the effect of execution is

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

b) Allowed usage

```
             ACC1                           ACC2
          +--------+                     +--------+
          | ACCUM  |                     | ACCUM  |
ACC-----------|A-----A|----------------|A-----A|---ACC
      +---+   |        |        +---+    |        |
X1---| * |---|X       |   X3---| * |---|X       |
X2---|   |    +--------+   X4---|   |    +--------+
      +---+                     +---+
```

Declarations as in **b)** are assumed for `ACC`, `X1, X2, X3`, and `X4`;

the effect of execution is

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

c) Allowed usage

```
                 ACC1
              +--------+
              | ACCUM  |
X3------------|A-----A|---X4
      +---+   |        |
X1---| * |---|X       |
X2---|   |    +--------+
      +---+
```

A declaration such as

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

is assumed: the effect of execution is

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

d) Allowed usage

```
                ACC1
      +---+   +--------+
X1---| * |   | ACCUM  |
X2---|   |---|A-----A|---ACC
      +---+   |        |
X3------------|X       |
              +--------+
```

NOT ALLOWED!!!
Connection to in-out variable `A` is not a variable or function block name (see preceding text)

e) Disallowed usage

Figure 14 - Declaration and usage of in-out variables in function blocks

6.5.3.5 Standard function blocks

6.5.3.5.1 General

Definitions of function blocks common to all programmable controller programming languages are given below.

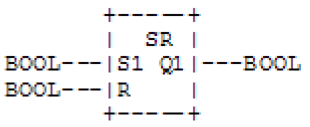
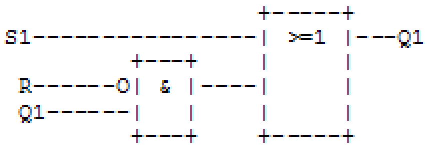
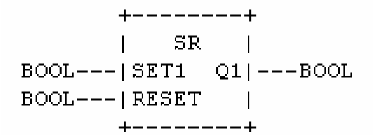
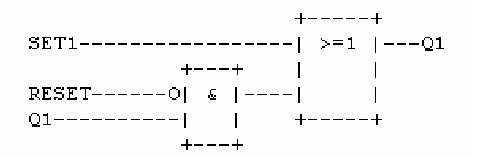
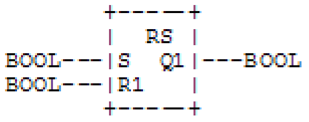
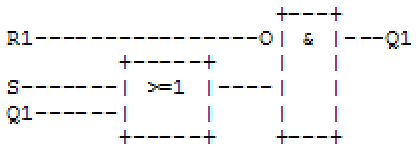
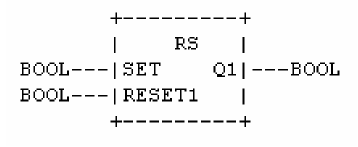
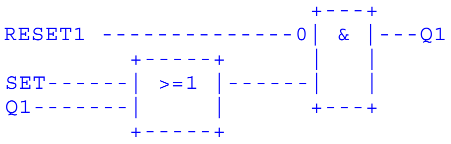
Where graphical declarations of standard function blocks are shown in this subclause, equivalent textual declarations, as specified in 6.5.3.4, can also be written, as for example in Table 42.

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.

6.5.3.5.2 Bistable elements

The graphical form and *function block body* of standard bistable elements are shown in Table 41. The notation for these elements is chosen to be as consistent as possible with symbols 12-09-01 and 12-09-02 of IEC 60617-12.

Table 41 - Standard bistable function blocks [a]

<table>
<tr><th>No.</th><th>Graphical form</th><th>Function block body</th></tr>
<tr><td rowspan="2">1a</td><td colspan="2"><b>Bistable function block (set dominant)</b></td></tr>
<tr><td>
<pre>
       +-----+
       | SR |
BOOL---|S1 Q1|---BOOL
BOOL---|R |
       +-----+
</pre></td><td>
<pre>
                  +-----+
S1----------------| >=1 |---Q1
        +---+ | |
 R-----O| & |-----| |
 Q1-----| | | |
        +---+   +-----+
</pre></td></tr>
<tr><td rowspan="2">1b</td><td colspan="2"><b>Bistable function block (set dominant) with long input names</b></td></tr>
<tr><td>
<pre>
       +---------+
       | SR |
BOOL---|SET1   Q1|---BOOL
BOOL---|RESET |
       +---------+
</pre></td><td>
<pre>
                    +-----+
SET1----------------| >=1 |---Q1
          +---+ | |
RESET----O| & |-----| |
Q1--------| | +-----+
          +---+
</pre></td></tr>
<tr><td rowspan="2">2a</td><td colspan="2"><b>Bistable function block (reset dominant)</b></td></tr>
<tr><td>
<pre>
       +-----+
       | RS |
BOOL---|S Q1|---BOOL
BOOL---|R1 |
       +-----+
</pre></td><td>
<pre>
                    +---+
R1-----------------O| & |---Q1
       +-----+ | |
S------| >=1 |------| |
Q1-----| | | |
       +-----+   +---+
</pre></td></tr>
<tr><td rowspan="2">2b</td><td colspan="2"><b>Bistable function block (reset dominant) with long input names</b></td></tr>
<tr><td>
<pre>
       +----------+
       | RS |
BOOL---|SET   Q1|---BOOL
BOOL---|RESET1 |
       +----------+
</pre></td><td>
<pre>
                        +---+
RESET1 ----------------0| & |---Q1
           +------+ | |
SET--------| >=1 |-----| |
Q1---------| | +---+
           +------+
</pre></td></tr>
<tr><td colspan="3">NOTE The function block body is specified in the Function Block Diagram (FBD) language defined in 8.3.</td></tr>
<tr><td colspan="3">[a] The initial state of the output variable Q1 shall be the normal default value of zero for Boolean variables.</td></tr>
</table>

6.5.3.5.3 Edge detection

The graphic representation of standard rising- and falling-edge detecting function blocks shall be as shown in Table 42. 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 42 - Standard edge detection function blocks

No.	Graphical form	Definition (ST language)
1	Rising edge detector	

	`+--------+`
	`     | R_TRIG |`
	`BOOL---|CLK    Q|---BOOL`
	`     +--------+`
		`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`

2	Falling edge detector	

	`+--------+`
	`     | F_TRIG |`
	`BOOL---|CLK    Q|---BOOL`
	`     +--------+`
		`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`

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” as described in 6.4.2. 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`.		

6.5.3.5.4 Counters

The graphic representations of standard counter function blocks, with the types of the associated inputs and outputs, shall be as shown in Table 43. The operation of these function blocks shall be as specified in the corresponding function block bodies.

Table 43 - Standard counter function blocks

No.	Graphical form	Function block body (ST language)
Up-counters		

1a	`+-----+`
	`       | CTU |`
	`BOOL--->CU   Q|---BOOL`
	`BOOL---|R     |`
	` INT---|PV CV|---INT`
	`       +-----+`
		`IF R`
		`THEN CV := 0;`
		`ELSIF CU AND (CV < PVmax)`
		`    THEN`
		`    CV := CV+1;`
		`END_IF;`
		` Q := (CV >= PV);`

1b	`+----------+`
	`       | CTU_DINT |`
	`BOOL--->CU        Q|---BOOL`
	`BOOL---|R          |`
	`DINT---|PV       CV|---DINT`
	`       +----------+`
		Same as 1a

1c	`+----------+`
	`       | CTU_LINT |`
	`BOOL--->CU        Q|---BOOL`
	`BOOL---|R          |`
	`LINT---|PV       CV|---LINT`
	`       +----------+`
		Same as 1a

<table>
<tr><th>No.</th><th>Graphical form</th><th>Function block body
(ST language)</th></tr>
<tr><td>1d</td><td>+-----------+
| CTU_UDINT |
BOOL--->CU Q|---BOOL
BOOL---|R |
UDINT---|PV CV|---UDINT
+-----------+</td><td>Same as 1a</td></tr>
<tr><td>1e</td><td>+-----------+
| CTU_ULINT |
BOOL--->CU Q|---BOOL
BOOL---|R |
ULINT---|PV CV|---ULINT
+-----------+</td><td>Same as 1a</td></tr>
<tr><td colspan="3"><b>Down-counters</b></td></tr>
<tr><td>2a</td><td>+-----+
| CTD |
BOOL--->CD Q|---BOOL
BOOL---|LD |
INT---|PV CV|---INT
+-----+</td><td>IF LD
THEN CV := PV;
ELSIF CD AND (CV > PVmin)
   THEN CV := CV-1;
END_IF ;
 Q := (CV &lt;= 0);</td></tr>
<tr><td>2b</td><td>+----------+
| CTD_DINT |
BOOL--->CD Q|---BOOL
BOOL---|LD |
DINT---|PV CV|---DINT
+----------+</td><td>Same as 2a</td></tr>
<tr><td>2c</td><td>+----------+
| CTD_LINT |
BOOL--->CD Q|---BOOL
BOOL---|LD |
LINT---|PV CV|---LINT
+----------+</td><td>Same as 2a</td></tr>
<tr><td>2d</td><td>+-----------+
| CTD_UDINT |
BOOL--->CD Q|---BOOL
BOOL---|LD |
UDINT---|PV CV|---UDINT
+-----------+</td><td>Same as 2a</td></tr>
<tr><td>2e</td><td>+-----------+
| CTD_ULINT |
BOOL--->CD Q|---BOOL
BOOL---|LD |
ULINT---|PV CV|---ULINT
+-----------+</td><td>Same as 2a</td></tr>
<tr><td colspan="3"><b>Up-down counters</b></td></tr>
<tr><td>3a</td><td>+------+
| CTUD |
BOOL--->CU QU|---BOOL
BOOL--->CD QD|---BOOL
BOOL---|R |
BOOL---|LD |
INT---|PV CV|---INT
+------+</td><td>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 &lt;= 0);</td></tr>
</table>

No.	Graphical form	Function block body (ST language)
3b	+-----------+ | CTUD_DINT | BOOL--->CU QU|---BOOL BOOL--->CD QD|---BOOL BOOL---|R | BOOL---|LD | DINT---|PV CV|---DINT +-----------+	Same as 3a
3c	+-----------+ | CTUD_LINT | BOOL--->CU QU|---BOOL BOOL--->CD QD|---BOOL BOOL---|R | BOOL---|LD | LINT---|PV CV|---LINT +-----------+	Same as 3a
3d	+------------+ | CTUD_UDINT | BOOL--->CU QU|---BOOL BOOL--->CD QD|---BOOL BOOL---|R | BOOL---|LD | ULINT---|PV CV|---ULINT +------------+	Same as 3a
3e	+------------+ | CTUD_ULINT | BOOL--->CU QU|---BOOL BOOL--->CD QD|---BOOL BOOL---|R | BOOL---|LD | ULINT---|PV CV|---ULINT +------------+	Same as 3a
NOTE The numerical values of the limit variables `PVmin` and `PVmax` are **implementation-dependent**.		

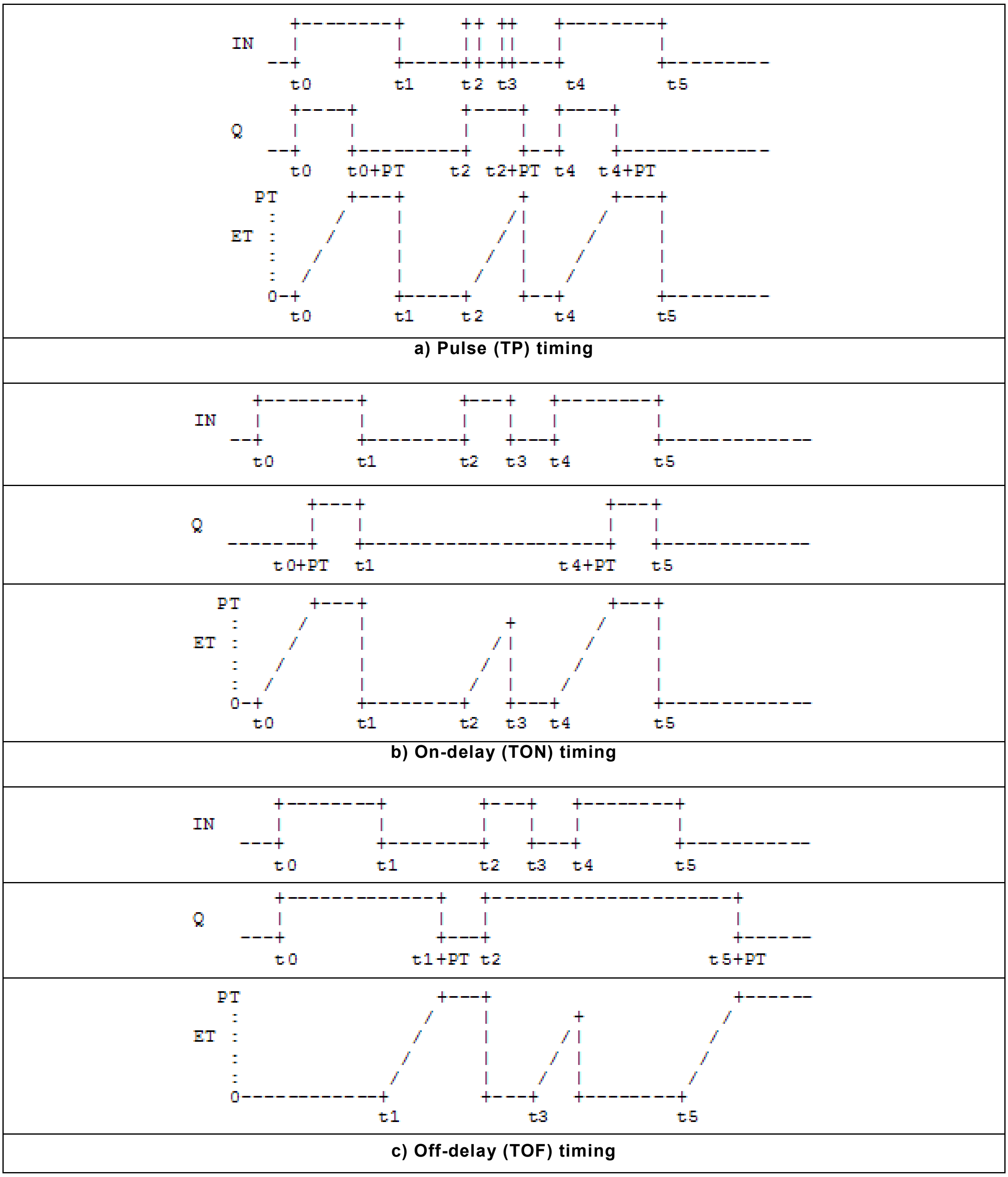
6.5.3.5.5 Timers

The graphic form for standard timer function blocks shall be as shown in Table 44. The operation of these function blocks shall be as defined in the timing diagrams given in Figure 15.

Table 44 - Standard timer function blocks

No.	Description	Graphical form
1	`*** is: TP (Pulse)`	+--------+ | *** | BOOL---|IN Q|---BOOL TIME---|PT ET|---TIME +--------+
2a	`TON (On-delay)`	
2b [a]	`T---0 (On-delay)`	
3a	`TOF (Off-delay)`	
3b [a]	`0---T (Off-delay)`	
NOTE The effect of a change in the value of the `PT` input during the timing operation, e.g., the setting of `PT` to `t#0s` to reset the operation of a `TP` instance, is an **implementation-dependent parameter**.		
[a] In textual languages, features 2b and 3b shall **not** be used.		

The following figure shows the timing diagrams of the standard timer function blocks.



a) Pulse (TP) timing

b) On-delay (TON) timing

c) Off-delay (TOF) timing

Figure 15 - Standard timer function blocks - timing diagrams (Example)

6.5.3.5.6 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.

6.5.4 Object oriented extensions to the function block concept

6.5.4.1 General

The function block concept of IEC 61131-3 2nd edition is extended to support the object oriented paradigm using the following concepts:

- Method
- Interface
- Inheritance

Which subset of the following features a particular **implementation** supports shall be stated by the manufacturer according the feature Table 45.

Table 45 - Features of the object oriented function blocks concept

No.	Keyword	DESCRIPTION	Clause

1a	`METHOD`
	`… END_METHOD`
		Method definition
		and call (call);
		i.e. `function_block_instance_name.method_name`
			6.5.4.2

1b	`PUBLIC,`
	`PRIVATE`
		Method access specifiers; i.e. `PUBLIC METHOD method_name`	6.5.4.2.4

2	`INTERNAL`	Method access specifier - `Namespace`	6.5.4.2.4
3	-	Function block body - **additional** to methods	6.5.4.1

4a
4b
	`INTERFACE,`
	`IMPLEMENTS`
		Interface and method prototype used
		a) with function block declaration – function block implements an interface
		b) as variable of type interface
			6.5.4.3
			6.5.4.3.3

5	`EXTENDS,`
	`THIS,`
	`SUPER,`
	`OVERRIDE`
		Function block inheritance - from another function block
		access to own FB
		access to base FB
		base methods to override - including “Name binding” in feature no. 8
			6.5.4.4.2

6	`EXTENDS`	Interface inheritance – from another interface	6.5.4.4.3
7	`PROTECTED`	Method access specifier - from inside of own and derived function block(s) only	6.5.4.2.4
 8a 8b		Name binding – see `OVERRIDE` a) Static override of methods b) Dynamic override of methods	6.5.4.4.4
9a	`ABSTRACT`	a) Abstract function block b) Abstract method	6.5.4.4.7

10	`NAMESPACE`
	`INTERNAL ACCESS`
	`… ACCESS_END`
	
	`PUBLIC ACCESS`
	`… ACCESS_END`
	`END_NAMESPACE`
		Applies **not only** to OOP language elements.
		Access areas.
			6.8

[Editor’s Note: All new language elements to be syntactically defined also in Annex B-Formal specification]

6.5.4.2 Methods in function blocks

6.5.4.2.1 General

For the purpose of the programmable controller languages the concept of *methods* well known in the object-oriented programming is adopted as 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. The construct corresponding to the syntactic element `function_block_body` in annex B.2.5.2 shall be used.

A function block with its methods and the call of a method is shown in the example in 6.5.4.2

When executed, a method may yield one or no data element, which is considered to be the method result, and additional output elements (`VAR_OUTPUT` and `VAR_IN_OUT`). As for any data element, the method result can be multi-valued, for example, an array or structure. Like the function result the call of a method yielding a result could be used as an operand in an expression.

Methods may be defined *instead* of the function block body or *additionally* to the function block body at is defined in 6.5.3.4. In the latter case the function block body shall be executed like a method.

NOTE The function block body in addition to the methods is permitted for compatibility reasons.

6.5.4.2.2 Method declaration

A method may be defined in any of the programming languages specified in this standard by using the keywords `METHOD` method_ name … `END_METHOD`. See also in the feature Table 45.

The name of a method shall be unique within the definition of the function block, this includes variable and method names. Two function blocks types may define methods with the same name.

The methods are declared within the scope of a function block type (after the function block body, if there is any). They have access to variables of the function block type (`VAR`, `VAR_INPUT`, `VAR_OUTPUT`, `VAR_EXTERNAL`), except variables within a `VAR_TEMP` or `VAR_IN_OUT` declaration.

Variables declared within the definition of a method shall **not** keep their state from one call to the next call, but shall be either assigned by the call (`VAR_INPUT`, `VAR_IN_OUT`) or initialized (`VAR_TEMP`, `VAR_OUTPUT`).

The methods may contribute to the internal state of the function block instance by declaring variables their own `VAR` … `END_VAR` section which is located in the function block instance data. Variables within this section keep their state from one call of the method of an FB instance to the next call of the method of the same instance.

The following example is for illustration only. The representation is not normative.

EXAMPLE see Figure 16

Illustration of the concept a function block with methods:

a) Function block type definition (drive)
   - FB Variables x, y, z
   - optional FB body and FB Input/Output variables
   - set of methods (start, stop) with each its result and variables and a algorithm
b) FB instantiation (my_drive)
c) Access from methods to FB variables
   - own variable (x), (y) and
   - common variables (z)
d) Method call: here the call from outside a Function block.

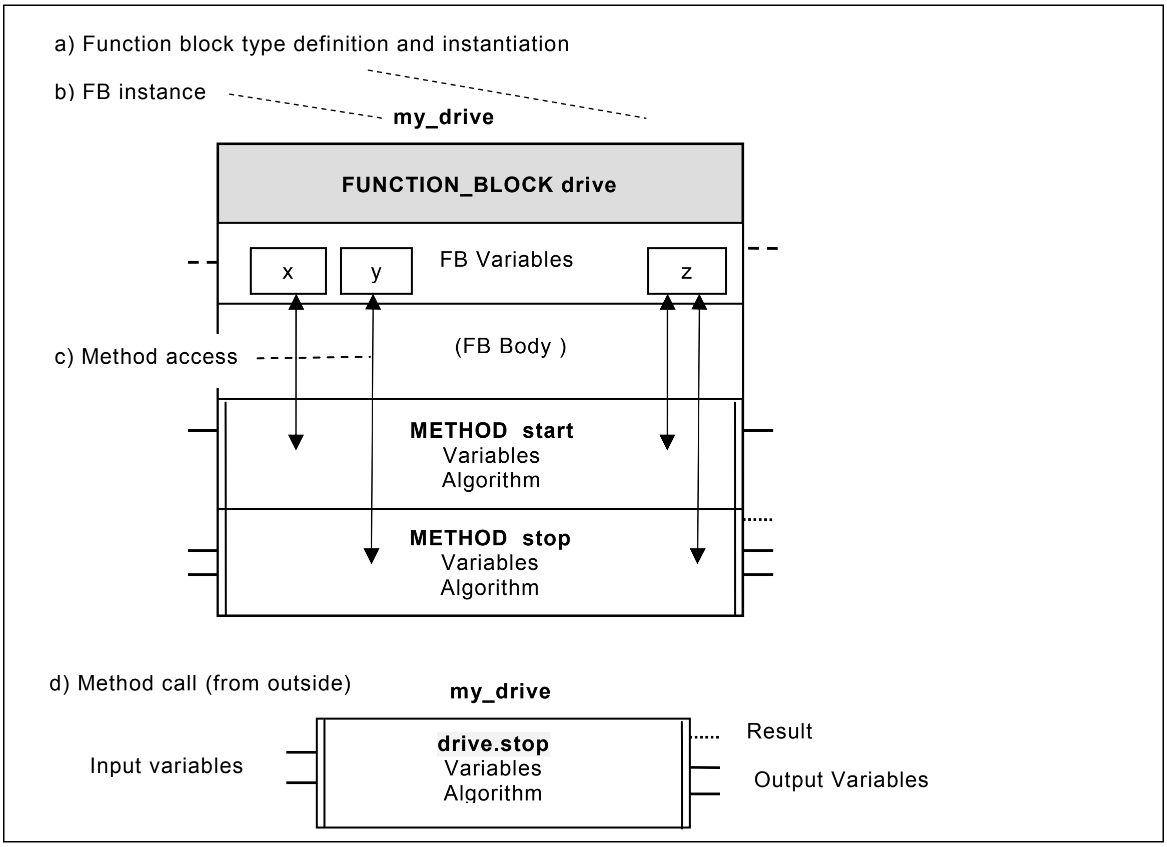


Figure 16 – Function block with methods declaration and method call (Example)

A method declaration is similar to a function declaration with the following differences:

- A method declaration is delimited by the keyword combination `METHOD` and `END_ METHOD`.
- A method declaration allows `VAR_TEMP` ... `END_VAR` declarations.
- A method declaration is placed after the body of the function block, if any or after the function block declarations.

  NOTE: The function block as defined in 6.5.3 has a body with operations, but this is optional if the function block has methods. If a function block has a body then it can be called.

- A method declaration may contain the additional keyword `OVERRIDE or ABSTRACT` as defined in 6.5.4.4.7
- A method declaration shall contain one of the following access specifiers: `PUBLIC`, `PRIVATE`, `INTERNAL`, and `PROTECTED` as defined in 6.5.4.2.4.

EXAMPLE see Figure 17

The example contains a Function Block COUNTER with two methods for counting up. Method UP1 shows how to call a method of the same function block.

```
FUNCTION_BLOCK COUNTER
  VAR
    CV : UINT;                      // current value of counter
  END_VAR

  PUBLIC METHOD UP : UINT           // method for count up by inc
  VAR_INPUT
       INC : UINT;                  // the increment
  END_VAR
  VAR_OUTPUT
       QU : BOOL;                   // upper limit detection
  END_VAR
  IF CV <= Max - INC                // max e.g. 10000
       THEN CV := CV + INC;         // count up of current value
       ELSE QU := TRUE;             / upper limit reached
  END_IF
  UP := CV;                         // result of method
  END_METHOD


  PUBLIC METHOD UP1 : UINT          // count up by 1
  VAR_OUTPUT
    QU: BOOL;                       // upper limit reached
  END_VAR
  UP1 := UP (INC := 1, QU => QU);  // internal method call
  END_METHOD
       // no body!
END_FUNCTION_BLOCK
```

Figure 17 – Function block with methods and method call (Example)

6.5.4.2.3 Method call

The methods can be called as shown in Figure 16 and Figure 17 in textual languages and in graphical languages.

In all languages representations there are two different cases of call (invocation) of a method as shown in Figure 18:

a) Internal call: an call of method of the *same* function block instance.

b) External call: an call of a method of an instance of *another* function block.

1. In the textual representation

- the internal call is similar to the function call: … method_name(arguments)
  It is also possible to use the keyword THIS as defined in 6.5.4.4.5

- the external call: … function_block_instance_name.method_name(arguments).

2. In the graphical representation

- the internal call shows only the method name inside the graphical block.
  It is also possible to use the keyword THIS as defined in 6.5.4.4.5 above the block.

- the outside call shows the method_name preceded by the function_block_type_name and "." inside the graphical block. The instance name shall be located above the block.

EXAMPLE see Figure 18
Method usage in Structured Text and Function Block Diagram:

```
VAR
       CT            : COUNTER; //see Function Block in Example Figure 17
       LIMITATION    : BOOL;
       VALUE         : UINT;
END_VAR
```

1. In Structured Text (ST) - see 7.3

a) Internal call of a method:

```
            VALUE := UP   (INC := 1, QU => LIMITATION);
```

b) External call of a method:

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

2. In Function Block Diagram (FBD) - see 8.3.

a) Internal call of a method:

```
       +--------------+
       |      UP      |---VALUE   // Method UP returns the result as “UP”
   1---|INC           |
       |            QU|---LIMITATION
       +--------------+
```

b) External call of a method:

```
              UP
       +--------------+
       |  COUNTER.UP  |
   1---|INC         UP|---VALUE   // Method UP returns the result as “CT.UP”
       |            QU|---LIMITATION
       +--------------+
```

Figure 18 –Internal and external method call (Example)

6.5.4.2.4 Access specifiers of method (`Public, Private, Internal, Protected`)

The accessibility of a method is defined by using one of the following *access specifiers* before the keyword `METHOD;` e.g. `PUBLIC METHOD Start.`

- `PUBLIC:` Indication for methods that are accessible at any place where the function block type can be used.
- `PRIVATE:` Indication for methods that are only accessible from inside the function block type itself.

  Note By specifying `PRIVATE` access to the function block body an call of the function block type itself is not possible from outside.

If *namespace* is implemented as defined in 6.8 a further access specifier is applicable

- `INTERNAL:` Indication for methods that are only accessible from within the *namespace* as specified in 6.8, in which the function block type is declared.

If *inheritance* is implemented a further access specifier is applicable

- `PROTECTED:` Indication for methods that are only accessible from inside a function block type and from inside all derived function block types.

All improper uses shall be treated as an **error**.

The accessibility of the (optional) *body* of a function block type shall also be defined with these *access specifiers*. Therefore the keyword shall be inserted before the keyword `FUNCTION_BLOCK`. For compatibility reasons, when no specifier is present, the access is `PUBLIC`.

EXAMPLE see Figure 19

Illustration of the accessibility (call) of methods defined in FB C:

a) Access specifiers: PUBLIC, PRIVAT, INTERNAL, PROTECTED
- PUBLIC M1 accessible by call M1 from inside FB B (also FB C)
- PRIVAT M2 accessible by call M2 from inside FB C only
- INTERNAL M3 accessible by call M3 from inside NAMESPACE A (also FB B , FB C)
- PROTECTED M4 accessible by call M4 from inside FB C_derived (also FB C)

b) Method calls inside/outside:
- M2 is called from inside FB C. - e.g. with keyword THIS.
- M1, M3 and M4 are FB C called from outside FB C – e.g. with keyword SUPER for M4.

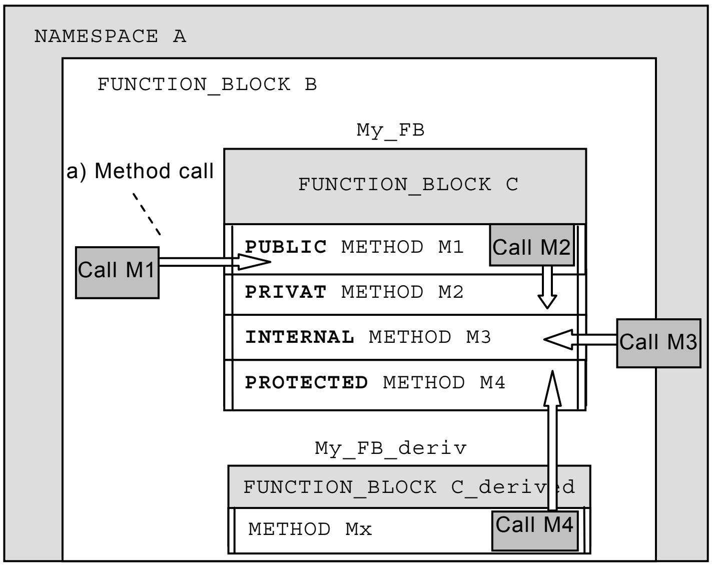


Figure 19 – Method accessibility (Example)

6.5.4.3 Interface

6.5.4.3.1 General

The keyword `INTERFACE` is used to declare a collection of *method prototypes* as defined in 6.5.4.3.2.

An `INTERFACE` is a “contract” between a function block and its caller(s) with following purposes:

- provides for separation of the interface specification from its implementation(s).
- allows for multiple implementations behind the common interface specification.
- allows abstraction across multiple function blocks.

The interface specification may be used in two ways as defined in 6.5.4.3.3 :

a) in a function block declaration.
This specifies what methods the function block shall implement; e.g. reuse of the interface specification.

b) as a type of a variable.
Variables whose *type* is *interface* are references to instances of function blocks and shall be assigned before usage to a valid function block instance. Otherwise it shall be an **error**.

Note To avoid a runtime error the programming tool could provides a default “dummy” method. Another way is to check in advance if it is assigned.

The interface can not be instantiated.

6.5.4.3.2 Method prototype

A method prototype is a restricted method declaration for the use with interface. It contains only the method name, VAR_INPUT, VAR_OUTPUT and VAR_IN_OUT variables and the method result. A method prototype does *not* contain any operations (code).

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

EXAMPLE see Figure 20

Illustration of INTERFACE general_drive with
a) method prototypes (no algorithm)
b) FB drive_A and FB B drive_B IMPLEMENTS the INTERFACE.
These FBs have methods with different algorithms.

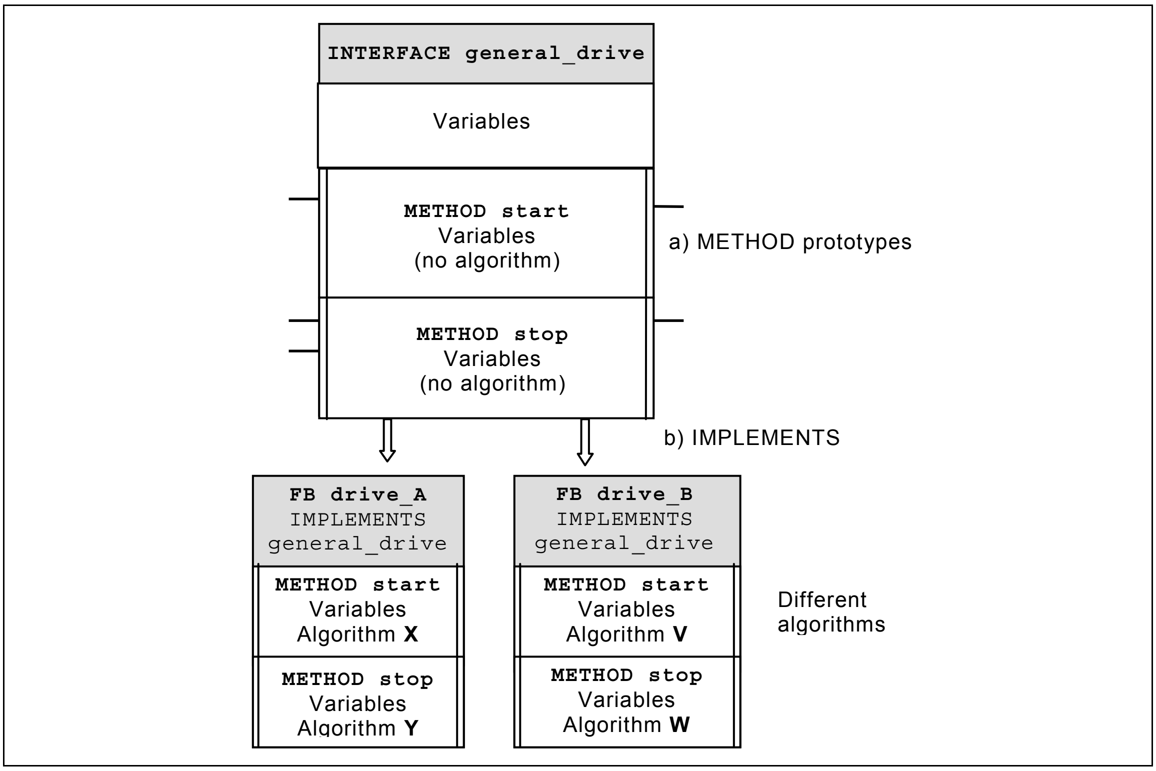


Figure 20 – Interface with derived Function blocks (Example)

6.5.4.3.3 Usage of interface (IMPLEMENTS)

As defined 6.5.4.3.1 an INTERFACE may be used in two ways:

a) In a function block declaration:

In this case the function block implements one or more INTERFACE(s) by using the keyword IMPLEMENTS as shown as examples in Figure 20, Figure 21and Figure 24**.**

The function block shall implement all methods specified by the *method prototype(s)* as defined in 6.5.4.3.2 that are contained in the INTERFACE specification. That means the function block contains all methods including their operations as defined in 6.5.4.2.1.

Note The implementation of a method prototype may have additional local variables (VAR).

The following situations shall be treated as an **error** according to the provisions of 5.1 d):

1. If a function block type does not implement all methods defined in the interface.

2. If a function block type implements a method with the same name as defined in the interface but with another set (or order) of VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT-variables or with another method result.

3. If a function block type implements a method with the same name as defined in the interface but not with the access specifier PUBLIC.

EXAMPLE 1 A function block implements an interface.

Declaration

```
INTERFACE ROOM
   METHOD DAYTIME   : VOID; // called during daytime
   END_METHOD
   METHOD NIGHTTIME : VOID; // called during nighttime
   END_METHOD
END_INTERFACE

FUNCTION_BLOCK LIGHTROOM IMPLEMENTS ROOM
VAR
    LIGHT : BOOL;
END_VAR

PUBLIC METHOD DAYTIME : VOID
    LIGHT := FALSE;
END_METHOD

PUBLIC METHOD NIGHTTIME : VOID
    LIGHT := TRUE;
END_METHOD

END_FUNCTION_BLOCK

---------------
```

Usage (by an external method call)

```
PROGRAM A
VAR  MyRoom : LIGHTROOM; END_VAR:

  ... IF MyRoom.DAYTIME THEN ..

END_PROGRAM
```

Figure 21 – Function block implements an interface (Example)

b) The interface may be used as a type of a variable*:*

In this case this variable is a *reference* to an instance of a function block implementing this interface. The variable shall be assigned to an instance of a function block before it can be used.

A variable of a type INTERFACE may be assigned to the following values:

1. an instance of a function block implementing the interface.

2. an instance of a function block which is derived from a function block type implementing the interface.

3. another variable of the same type.

4. the special invalid reference NULL. This is also the initial value of the variable, if not declared otherwise.

A variable of a type of an interface may be compared for equality with NULL. So the variable **shall** be tested to be a valid reference before calling a method of the interface.

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.

EXAMPLE 2 An interface with two methods and a function block using this methods.

Declaration

```
INTERFACE ROOM               // same as Example 1
   METHOD DAYTIME    : VOID; // called during daytime
   END_METHOD
   METHOD NIGHTTIME : VOID; // called during nighttime
   END_METHOD
END_INTERFACE

FUNCTION_BLOCK ROOM_CTRL
  VAR_INPUT
    RM: ROOM;        // interface ROOM as type of an (input) variable !
  END_VAR

  VAR_EXTERNAL
    Actual_TOD : TOD;       // global time definition
  END_VAR

IF (RM = NULL) THEN  // Important: test valid reference!
   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
```

Figure 22 – Function block type with calls of the methods of an interface (Example)

EXAMPLE 3: A program passing a specific instance to a variable of type interface

```
PROGRAM B
VAR
  MyRoom      : LIGHTROOM;    // Figure 21 - FB LIGHTROOM implements ROOM with methods
  MyRoomCtrl  : ROOM_CTRL;    // Figure 23 - FB ROOM_CTRL calls
END_VAR

    MyRoomCtrl(RM := MyRoom);
END_PROGRAM
```

Figure 23 –Passing of function block instance (Example)

6.5.4.4 Inheritance

6.5.4.4.1 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

a) new function block types which include methods as defined in 6.5.4.2 or

b) new interfaces as defined in 6.5.4.3 using function block or interfaces that have already been defined.

This is possible in a hierarchical tree like it is illustrated in Figure 24.

The new function blocks and interfaces are called *derived (child)* function block and *derived* (*child)* interface respectively. The already existing predecessors are called *base* (*parent)* function block and *base (parent)* interface respectively.

When derived elements inherit only from *one* base element it is known as *single inheritance*.

NOTE In this standard only the *single* inheritance is defined. *Multiple* inheritance with more than one base element is **not** defined by this standard.

6.5.4.4.2 Function block inheritance (`EXTENDS, OVERRIDE`)

A function block type may be derived from an existing function block type (the base function block type) using the keyword `EXTENDS`.

The following rules apply to inheritance:

1. The derived function block type inherits all methods and all variables from its base function bock type..

   Note 1 That means, it contains all inputs, outputs and local variables and all methods without explicitly declaring it. An exception to the inheritance of methods is described in rule 2.

   Note 2 The function block type used as a base function block, may itself be a derived function block type. Then it passes on to its heir also the methods and variables it inherited.

   Note 3 If the base function block type changes its definition, all derived function block types (and their heirs) have also this changed functionality.

   Note 4 The derived function block type may have variables and methods in addition to its base function block and thus create new functionality.

2. In order to *override* base methods the following rules apply:

   a) The method that overrides shall be declared with the additional keyword `OVERRIDE` following the keyword `METHOD`.

   b) The method that overrides shall have the same *signature* (i.e.: the same name, access specifier, inputs, outputs and return type) within the scope of the derived function block type.

      Note The newly declared method replaces the method declaration of the base function block.

EXAMPLE 1 (see Figure 24)

Illustration of the hierarchy of inheritance:

a) Interface inheritance as defined in 6.5.4.4.3:
   Using the keyword EXTENDS for the derived interface

b) FB implementation of an interface using the keyword IMPLEMENTS as defined in 6.5.4.3.3

c) Function Block inheritance as defined in 6.5.4.4.2:
   Using keyword
   - EXTENTS for the derived FB and
   - OVERRIDE for overriding a base method

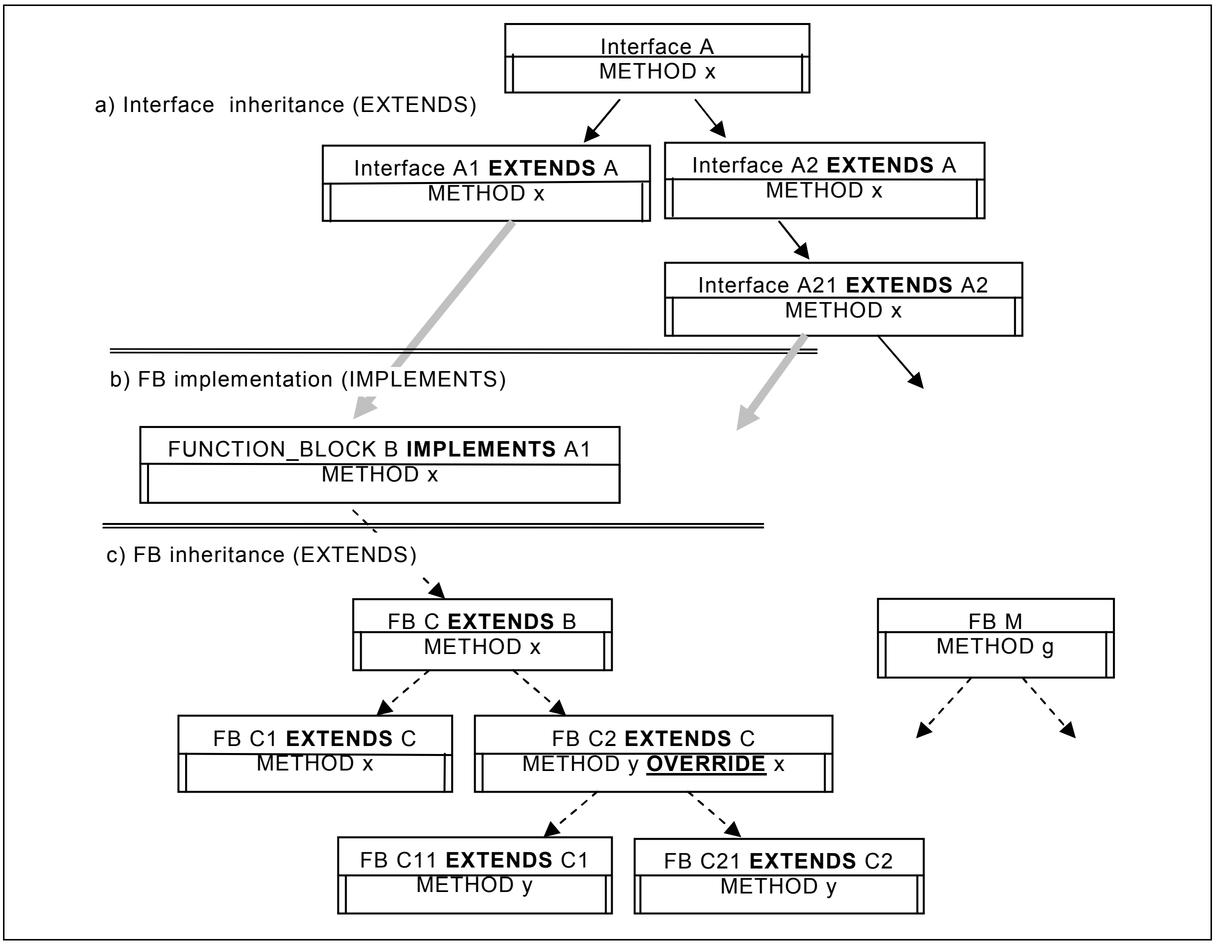


Figure 24 – Interface inheritance and function block inheritance (Example)

The following situations shall be treated as an **error** according to the provisions 5.1d):

1. The derived function block type defines a variable with the name of a variable or method already contained in its base function block type, whether defined or inherited.
2. The derived function block type defines a method with the name of a method already contained in its base function block, but:
   - the derived function block type has no access to the parent's method (`PRIVATE` or `INTERNAL` in a different *namespace*) or
   - the new method does not have the same signature (i.e.: the same name, *access specifier*, inputs, outputs and return type) or
   - the new method has a different *access specifier* or
   - the new method does not have the keyword `OVERRIDE`.
3. The derived function block defines a method with the name of a variable already contained in its parent function block.
4. A function block is its own base function block, whether directly or indirectly.

EXAMPLE see Figure 25

A function block that extends the function block LIGHTROOM

```
FUNCTION_BLOCK LIGHT2ROOM EXTENDS LIGHTROOM      // see example Figure 21
VAR
    LIGHT2 : BOOL;                               // second light
END_VAR

PUBLIC OVERRIDE METHOD DAYTIME : VOID
    LIGHT := FALSE;                       // access to parent's variable
    LIGHT2 := FALSE;                      // specific implementation
END_METHOD

PUBLIC OVERRIDE METHOD NIGHTTIME : VOID
    LIGHT := TRUE;                        // access to parent's variable
    LIGHT2 := TRUE;                       // specific implementation
END_METHOD

END_FUNCTION_BLOCK
```

Figure 25 –Inheritance and override (Example)

6.5.4.4.3 Interface inheritance (EXTENDS)

An interface may be derived from an existing interface (the base interface) using the keyword EXTENDS. See example in 6.5.4.4.2 The following rules shall apply:

1. The derived (child) interface inherits all *method prototypes* defined in 6.5.4.3.2 from its base (parent) interface.

   The interface used as a base interface, may itself be a derived interface. Then it passes on to its descendants (derived) also the methods and variables it inherited.

   If the base interface changes its definition, all derived interfaces (and their descendants) have also this changed functionality.

2. The derived interface may have method prototypes in addition to its base interface and thus create new functionality.

The implementation of interfaces shall be according to following rules:.

1. A function block type inherits all interfaces from its base function block type and may also override methods implementing method-prototypes.

2. The implementation of a method-prototype may also be inherited from the base function block type, even if the base function block type does not implement the interface.

The following situations shall be treated as an **error** according to the provisions of 5.1 d):

1. If a function block type or its base types do not implement all methods defined in the interface.

2. If a function block type or its base type implement a method with the same name as defined in the interface but with another set (or order) of VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT-variables or with another method result.

The following situations shall be treated as **errors**:

1. An interface defines a additional method prototype (like in rule 2) with the name of a method prototype of one of its base interfaces.
2. Two or more parent interfaces contain method prototypes with the same name.

6.5.4.4.4 Name Binding

Name binding is the association of a method name with a method implementation. The binding of a name before the program runs is called *static* binding**.** A binding performed when the program runs is *dynamic* binding.

In case of an internal method call as defined 6.5.4.2.3, the overriding feature causes a difference between the static and dynamic form of name binding:

1. *Static* binding associates the method name to the method implementation of the function block type which, respectively, makes the internal method call or contains the method making the internal method call.
2. *Dynamic* binding associates the method name to the method implementation of the actual type of the function block instance.

A particular implementation supporting the overriding feature shall state in the feature Table 45 whether it resolves internal method calls using static or dynamic binding.

EXAMPLE

Overriding with effect on dynamic binding

In the following example the function block type `CIRCLE` contains an internal call of its method `PI` with low accuracy to calculate the circumference of a circle. The derived function block type `CIRCLE2` overrides this definition with a more accurate definition of `PI`.

In case of static binding the call `PI()` refers to `CIRCLE.PI`. In case of dynamic binding the call `PI()` refer either to `CIRCLE.PI` or to `CIRCLE2.PI`, according to the type of the instance on which the call of `CIRCUMFERENCE` was performed.

In case of static binding, `CUMF1` and `CUMF2` have the same value. In case of dynamic binding, `CUMF2` is more accurate than `CUMF1`.

```
FUNCTION_BLOCK CIRCLE

PUBLIC METHOD PI : LREAL
   PI := 3.1415;
END_METHOD

PUBLIC METHOD CIRCUMFERENCE : LREAL
VAR_INPUT
   DIAMETER : LREAL;
END_VAR
   CIRCUMFERENCE := PI() * DIAMETER; // internal call of PI
END_METHOD
END_FUNCTION_BLOCK

FUNCTION_BLOCK CIRCLE2 EXTENDS CIRCLE
PUBLIC OVERRIDE METHOD PI : LREAL
   PI := 3.1415926535897;
END_METHOD
END_FUNCTION_BLOCK

PROGRAM TEST
VAR
  CIR1 : CIRCLE;
  CIR2 : CIRCLE2;
  CUMF1 : LREAL;
  CUMF2 : LREAL;
  DYNAMIC : BOOL;
END_VAR
  CUMF1 := CIR1.CIRCUMFERENCE(1.0);
  CUMF2 := CIR2.CIRCUMFERENCE(1.0);
  DYNAMIC := CUMF1 <> CUMF2;
END_PROGRAM
```

Figure 26 – Dynamic vs. static Name binding (Example)

6.5.4.4.5 Access reference (`THIS`/`SUPER`)

With the keyword “`THIS`”, a reference to the own instance can be accessed in the scope of a function block. This reference may be passed to a variable of the type of an `INTERFACE` according to the rules given in 6.5.4.3.3.

EXAMPLE 1:

```
FUNCTION_BLOCK DARKROOM IMPLEMENTS ROOM   // ROOM see example Figure 21
VAR_EXTERNAL
   RoomCtrl : ROOM_CTRL;
END_VAR

PUBLIC METHOD DAYTIME : VOID
END_METHOD
PUBLIC METHOD NIGHTTIME : VOID
END_METHOD

// function block body
  RoomCtrl(RM := THIS);  // call room ctrl with own instance
END_FUNCTION_BLOCK
```

Figure 27 – Usage of `THIS` (Example)

With the keyword “`SUPER`”, the base class implementation of a method can be called. Thus, no dynamic binding takes place, but the base function block’s method is called despite the actual instance of the function block.

EXAMPLE 2: Alternative implementation of LIGHT2ROOM

```
FUNCTION_BLOCK LIGHT2ROOM EXTENDS LIGHTROOM // LIGHTROOM see example Figure 21
VAR
    LIGHT2 : BOOL;                       // second light
END_VAR

PUBLIC OVERRIDE METHOD DAYTIME : VOID
    SUPER.DAYTIME();                     // access to parent variable
    LIGHT2 := TRUE;                      // specific implementation
END_METHOD

PUBLIC OVERRIDE METHOD NIGHTTIME : VOID
    SUPER.NIGHTTIME();                   // access to parent variable
    LIGHT2 := FALSE;                     // specific implementation
END_METHOD
END_FUNCTION_BLOCK
```

Figure 28 – Usage of `SUPER` (Example)

6.5.4.4.6 Polymorphism

Polymorphism in object oriented programming is the ability of one type to appear as and be used like another type. This means that the first type derives from the second type or the first type implements an interface that represents second type.

NOTE Since a variable of interface type might refer to different instances of different derived function block types with completely different implementations of the called method, it is necessary to use dynamic binding.

EXAMPLE Polymorphism.

```
PROGRAM
VAR
  MyRoom1     : LIGHTROOM;   // see example Figure 21
  MyRoom2     : LIGHT2ROOM;  // see example Figure 28
  MyRoomCtrl  : ROOM_CTRL;   // see example Figure 22
END_VAR

 MyRoomCtrl(RM := MyRoom1); // calls in MyRoomCtrl will call methods of LIGHTROOM
 MyRoomCtrl(RM := MyRoom2); // calls in MyRoomCtrl will call methods of LIGHT2ROOM
END_PROGRAM
```

Figure 29 – Polymorphism (Example)

6.5.4.4.7 ABSTRACT function block and method

The ABSTRACT modifier may be used with function blocks or with single methods.

- **Abstract function block**

  The use the ABSTRACT modifier in a Function Block declaration indicates that a function block is intended to be a *base* type of other Function Blocks to be used for inheritance as explained in 6.5.4.4.1.

  The abstract function block has the following features:

  - An abstract function block cannot be instantiated.
  - An abstract function block shall only contain abstract methods.

  A (non-abstract) function block derived from an abstract function block shall include actual implementations of all inherited abstract methods.

- **Abstract method**

  If one or more methods are marked as ABSTRACT

  a) in a function block declaration, or

  b) in an abstract function block

  then they shall be implemented by function blocks that derive from the abstract function block.

6.5.5 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 programmable controller system."

Subclause 4.1 of this Part describes the place of programs in the overall software model of a programmable controller; subclause 4.2 describes the means available for inter- and intra-program communication; and subclause 4.3 describes the overall process of program development.

The declaration and usage of *programs* is identical to that of *function blocks* as defined in 6.5.3, with the additional features shown in Table 46 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. The format and usage of this declaration shall be as described in 6.7.2 and in IEC 61131-5.
3. *Programs* can only be instantiated within *resources*, as defined in 6.7.2, while *function blocks* can only be instantiated within *programs* or other *function blocks*.
4. A program can contain location assignments as described in 6.4.4 in the declarations of its global and internal variables. Location assignments with not fully specified direct representation as described in 6.4.2 and 6.4.4 can only be used in the declaration of internal variables of a program.

The declaration and use of programs are illustrated in Figure 35.

Limitations on the size of programs in a particular *resource* are **implementation dependencies**.

Table 46 - Program declaration features

No.	DESCRIPTION
1a to 9b	Same as features 1a to 9b, respectively, of Table 40
10	Formal input and output variables
11 to 14	Same as features 1 to 4, respectively, of Table 20
15 to 17	Same as features 1 to 3, respectively, of Table 21
18	Feature number not used
19	Use of directly represented variables
20	VAR_GLOBAL...END_VAR declaration within a PROGRAM
21	VAR_ACCESS...END_VAR declaration within a PROGRAM
22a	VAR_EXTERNAL declarations within PROGRAM type declarations
22b	VAR_EXTERNAL CONSTANT declarations within PROGRAM type declarations
23	VAR_GLOBAL CONSTANT declarations within PROGRAM type declarations
24	VAR_TEMP declarations within PROGRAM type declarations

6.6 Sequential Function Chart (SFC) elements

6.6.1 General

This subclause 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 this subclause 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 only 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.

6.6.2 Steps

A *step* represents a situation in which the behaviour 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 47, a step shall be represented graphically by a block containing a *step name* in the form of an identifier as defined in 6.1.2, 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` construction defined in 6.6.3.

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 47. 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 47.

Similarly, the elapsed time, `***.T`, since initiation of a step can be represented by a structure element of type `TIME`, as shown in Table 47. 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 47.

For system initialization as defined in 6.4.3, 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, as in feature 3 of Table 40, 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 as defined in 6.4.3.

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 47 - Step features

No.	REPRESENTATION	DESCRIPTION

1	`|`
	`+-----+`
	`| *** |`
	`+-----+`
	`|`
		Step - graphical form
		with directed links
		"***" = step name

	`|`
	`+=======+`
	`|| *** ||`
	`||     ||`
	`+=======+`
	`|`
		Initial step - graphical form with directed links
		"***" = name of initial step

2	`STEP *** :`
	`   (* Step body *)`
	`END_STEP`
		Step - textual form
		without directed links
		"***" = step name

	`INITIAL_STEP *** :`
	`   (* Step body *)`
	`END_STEP`
		Initial step - textual form
		without directed links
		"***" = name of initial step

3a [a]	`***.X`	Step flag - general form "***" = step name `***.X` = `BOOL#1` when `***` is active, `BOOL#0` otherwise

3b [a]	`|`
	`+-----+`
	`| *** |----`
	`+-----+`
	`|`
		Step flag - direct connection
		of Boolean variable `***.X` to
		right side of step "***"

4 [a]	`***.T`	Step elapsed time - general form "***" = step name `***.T` = a variable of type `TIME`

NOTE The upper directed link to an initial step is not present if it has no predecessors.

[a] When feature 3a, 3b, or 4 is supported, it shall be an error if the user program attempts to modify the associated variable. For example, if S4 is a step name, then the following statements would be errors in the ST language defined in 7.3:

```
S4.X := 1 ; (* ERROR *)
S4.T := t#100ms ; (* ERROR *)
```

6.6.3 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 48:

1. By placing the appropriate Boolean expression in the ST language defined in 7.3 physically or logically adjacent to the vertical directed link.
2. By a ladder diagram network in the LD language defined in 8.2, physically or logically adjacent to the vertical directed link.
3. By a network in the FBD language defined in 8.3, physically or logically adjacent to the vertical directed link.
4. By a LD or FBD network whose output intersects the vertical directed link via a *connector* as defined in 8.1.2.
5. 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`.
6. By a `TRANSITION...END_TRANSITION` construct using the IL language defined in 7.2. 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.

7. 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** in the sense of 5.1 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 are **implementation dependencies**.

Table 48 - Transitions and transition conditions

<table>
<tr><th>No.</th><th>Example</th><th>Description</th></tr>
<tr><td>1[a]</td><td> |
+-----+
|STEP7|
+-----+
   |
   + %IX2.4 & %IX2.3
   |
+-----+
|STEP8|
+-----+
   |</td><td>Predecessor step
Transition condition physically or logically adjacent to the transition using ST language
Successor step</td></tr>
<tr><td>2[a]</td><td> |
+-----+
|STEP7|
+-----+
   | %IX2.4   %IX2.3 |
   +---||-----||------+
   | |
+-----+
|STEP8|
+-----+
   |</td><td>Predecessor step
Transition condition physically or logically adjacent to the transition using LD language
Successor step</td></tr>
<tr><td>3[a]</td><td> |
               +-----+
               |STEP7|
               +-----+
      +-------+ |
%IX2.4--| & |-----+
%IX2.3--| | |
      +-------+ +-----+
                |STEP8|
                +-----+
                   |</td><td>Predecessor step
Transition condition physically or logically adjacent to the transition using FBD language
Successor step</td></tr>
<tr><td>4[a]</td><td> |
       +-----+
       |STEP7|
       +-----+
          |
>TRANX>----+
          |
       +-----+
       |STEP8|
       +-----+
          |</td><td>Use of connector:
predecessor step
transition connector
successor step</td></tr>
<tr><td>4[a]
4 [b]</td><td>| %IX2.4   %IX2.3
+---||-----||---->TRANX>
|
         +-------+
         | & |
%IX2.4---| |-->TRANX>
%IX2.3---| |
         +-------+</td><td>Transition condition:
Using LD language
Using FBD language</td></tr>
</table>

No.	Example	Description
5 [b]	STEP STEP7: END_STEP TRANSITION FROM STEP7 TO STEP8 := %IX2.4 & %IX2.3 ; END_TRANSITION STEP STEP8: END_STEP	Textual equivalent of feature1 using ST language

6 [b]	STEP STEP7: END_STEP
	TRANSITION FROM STEP7 TO STEP 8:
	  LD %IX2.4
	  AND %IX2.3
	END_TRANSITION
	STEP STEP8: END_STEP
		Textual equivalent of feature 1 using IL language

7[a]	| +------+ |STEP7| +------+ | + TRAN78 | +------+ |STEP8| +------+ |	Use of transition name: predecessor step transition name successor step
7[a]	TRANSITION TRAN78 FROM STEP7 TO STEP8: | | | %IX2.4 %IX2.3 TRAN78 | +---||-----||------( )---+ | | END_TRANSITION	Transition condition using LD language

7 [b]	TRANSITION TRAN78 FROM STEP7 TO STEP8:
	          +--------+
	          | & |
	%IX2.4---| |--TRAN78
	%IX2.3---| |
	          +--------+
	END_TRANSITION
		Transition condition using FBD language

7c	TRANSITION TRAN78 FROM STEP7 TO STEP8:
	      LD %IX2.4
	      AND %IX2.3
	END_TRANSITION
		Transition condition using IL language

7[d]	TRANSITION TRAN78 FROM STEP7 TO STEP8
	  := %IX2.4 & %IX2.3 ;
	END_TRANSITION
		Transition condition using ST language

[a] If feature 1 of Table 47 is supported, then one or more of features 1, 2, 3, 4, or 7 of this table shall be supported.

[b] If feature 2 of Table 47 is supported, then feature 5 or 6 of this table, or both, shall be supported.

6.6.4 Actions

6.6.4.1 General

An action can be a Boolean variable, a collection of *instructions* in the IL language defined in 8.2, a collection of *statements* in the ST language defined in 7.3, a collection of *rungs* in the LD language defined in 8.2, a collection of *networks* in the FBD language defined in 8.2.6, or a *sequential function chart* (SFC) organized .

Actions shall be declared via one or more of the mechanisms defined in 6.6.4.2, and shall be associated with steps via textual *step bodies* or graphical *action blocks*, as defined in 6.6.4.3. The details of action block representation are defined in 6.6.4.5. Control of actions shall be expressed by *action qualifiers* as defined in 6.6.4.6.

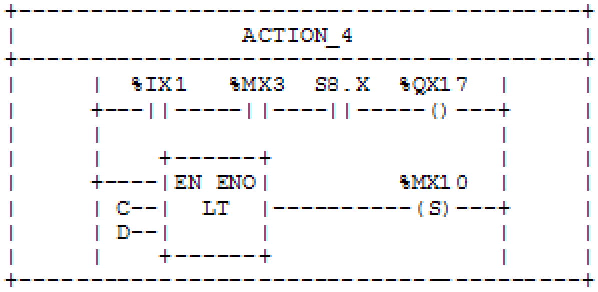
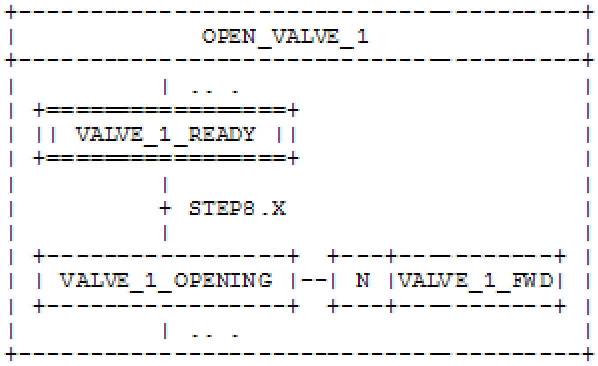
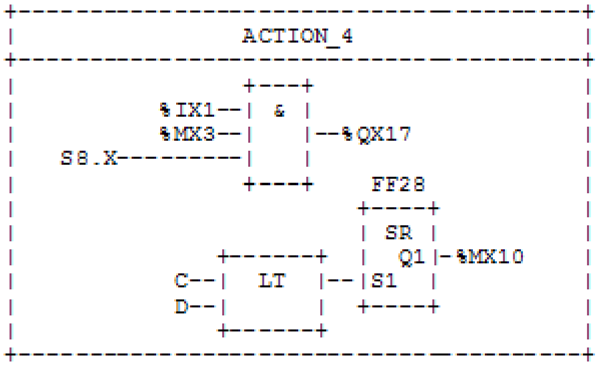
It shall be an **error** if the value of a Boolean variable used as the name of an action is modified in any manner other than as the name of one or more actions in the same SFC.

A programmable controller implementation which supports SFC elements shall provide one or more of the mechanisms defined in Table 49 for the declaration of actions. The *scope* of the declaration of an action shall be *local* to the program organization unit containing the declaration.

6.6.4.2 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 49 - Declaration of actions ^{a,b}

No.	Example	Feature
1	Any Boolean variable declared in a `VAR` or `VAR_OUTPUT` block, or their graphical equivalents, can be an action.	
2l		Graphical declaration in LD language
2s		Inclusion of SFC elements in action
2f		Graphical declaration in FBD language

3s	`ACTION ACTION_4:`
	`  %QX17 := %IX1 & %MX3 & S8.X ;`
	`  FF28(S1 := (C<D));`
	`  %MX10 := FF28.Q;`
	`END_ACTION`
		Textual declaration in ST language

No.	Example	Feature

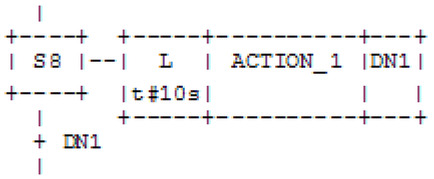
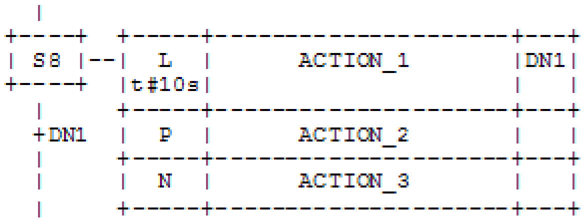
3i	`ACTION ACTION_4:`
	`LD S8.X`
	`AND %IX1`
	`AND %MX3`
	`ST %QX17`
	`LD C`
	`LT D`
	`S1 FF28`
	`LD FF28.Q`
	`ST %MX10`
	`END_ACTION`
		Textual declaration in IL language

NOTE The step flag `S8.X` is used in these examples to obtain the desired result such that, when `S8` is deactivated, `%QX17 := 0`.		
[a] If feature 1 of Table 47 is supported, then one or more of the features in this table, or feature 4 of Table 50, shall be supported. [b] If feature 2 of Table 47 is supported, then one or more of features 1, 3s, or 3i of this table shall be supported.		

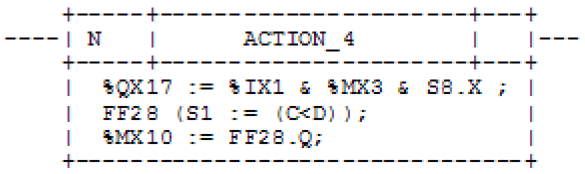
6.6.4.3 Association with steps

A programmable controller implementation which supports SFC elements shall provide one or more of the mechanisms defined in Table 50 for the association of actions with steps. The maximum number of action blocks per step is an **implementation dependency.**.

Table 50 - Step/action association

No.	Example	Feature
1		Action block physically or logically adjacent to the step
2	 S8 L ACTION_1 DN1 t#10s DN1 P ACTION_2 N ACTION_3	Concatenated action blocks physically or logically adjacent to the step

3	`STEP S8:`
	`ACTION_1(L,t#10s,DN1) ;`
	`ACTION_2(P) ;`
	`ACTION_3(N) ;`
	`END_STEP`
		Textual step body

4 [a]	 `%QX17 := %IX1 & %MX3 & S8.X ;`
	`FF28 (S1 := (C<D));`
	`%MX10 := FF28.Q;`
		Action block
		"d" field

[a] When feature 4 is used, the corresponding action name cannot be used in any other action block.		

6.6.4.4

6.6.4.5 Action blocks

As shown in Table 51, an *action block* is a graphical element for the combination of a Boolean variable with one of the *action qualifiers* specified in 6.6.4.6 to produce an enabling condition, according to the rules given in 6.6.4.7, for an associated action.

The action block provides a means of optionally specifying Boolean “indicator” variables, indicated by the “c” field in Table 51, 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 51, 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.

As well as being associated with a step, an action block can be used as a graphical element in the LD or FBD languages specified in clause 8. In this case, signal or power flow through an action block shall follow the rules specified in 8.1.3.

Table 51 - Action block features

<table>
<tr><th>No.</th><th>Feature</th><th>Graphical form/example</th></tr>
<tr><td>1 [a]</td><td>"a" : Qualifier as per 6.6.4.6</td><td rowspan="8"><pre>
    +-----+--------------+-----+
 ---| "a" | "b" | "c" |---
    +-----+--------------+-----+
    | "d" |
    | |
    +--------------------------+
</pre></td></tr>
<tr><td>2</td><td>"b" : Action name</td></tr>
<tr><td>3 [b]</td><td>"c" : Boolean "indicator" variables</td></tr>
<tr><td colspan="2"><b>"d" : Action using:</b></td></tr>
<tr><td>4</td><td>- IL language</td></tr>
<tr><td>5</td><td>- ST language</td></tr>
<tr><td>6</td><td>- LD language</td></tr>
<tr><td>7</td><td>- FBD language</td></tr>
<tr><td>8</td><td>Use of action blocks in ladder diagrams</td><td><pre>
| S8.X   %IX7.5   +---+------+---+ OK1 |
+--| |----| |-----| N | ACT1 |DN1|--( )--+
| +---+------+---+ |
</pre></td></tr>
<tr><td>9</td><td>Use of action blocks in function block diagrams</td><td><pre>
         +---+   +---+------+-----+
  S8.X---| & |------| N | ACT1 | DN1 |---OK1
%IX7.5---| | +---+------+-----+
         +---+
</pre></td></tr>
<tr><td colspan="3">[a] Field “a” can be omitted when the qualifier is “N”.
[b] Field “c” can be omitted when no indicator variable is used.</td></tr>
</table>

6.6.4.6 Action qualifiers

Associated with each step/action association defined in 6.6.4.3, or each occurrence of an action block as defined in 6.6.4.5, shall be an *action qualifier*. The value of this qualifier shall be one of the values listed in Table 52. In addition, the qualifiers L, D, SD, DS, and SL shall have an associated duration of type TIME.

The control of actions using these qualifiers is defined in 6.6.4.7.

Table 52 - Action qualifiers

No.	Qualifier	Explanation
1	None	Non-stored (null qualifier)
2	N	**N**on-stored
3	R	overriding **R**eset
4	S	**S**et (**S**tored)
5	L	time **L**imited
6	D	time **D**elayed

7	P	**P**ulse
8	SD	**S**tored and time **D**elayed
9	DS	**D**elayed and **S**tored
10	SL	**S**tored and time **L**imited
11	P1	**P**ulse (rising edge)
12	P0	**P**ulse (falling edge)

6.6.4.7 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 30 and Figure 31. If the action is declared as a Boolean variable, as defined in 6.6.4.2, 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, as defined in 6.6.4.2, 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 manufacturer may opt for a simpler implementation as shown in Figure 31 b). In this case, if the action is declared as a collection of statements or networks, as defined in 6.6.4.2, 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 manufacturer shall specify which one of the features given in Table 53 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 **implementation-dependent** feature.

b) A Boolean input to the ACTION_CONTROL block for an action shall be said to have an *association* with a step as defined in 6.6.4.2, or with an action block as defined in 6.6.4.5, 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** in the sense of 5.1 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 32. 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.

```
          +-----------------+
          | ACTION_CONTROL |
BOOL---|N                Q|---BOOL
BOOL---|R                A|---BOOL
BOOL---|S                 |
BOOL---|L                 |
BOOL---|D                 |
BOOL---|P                 |
BOOL---|P1                |
BOOL---|P0                |
BOOL---|SD                |
BOOL---|DS                |
BOOL---|SL                |
TIME---|T                 |
          +-----------------+
```

a) With "final scan" logic (see Figure 31 a)

```
          +-----------------+
          | ACTION_CONTROL |
BOOL---|N                Q|---BOOL
BOOL---|R                 |
BOOL---|S                 |
BOOL---|L                 |
BOOL---|D                 |
BOOL---|P                 |
BOOL---|P1                |
BOOL---|P0                |
BOOL---|SD                |
BOOL---|DS                |
BOOL---|SL                |
TIME---|T                 |
          +-----------------+
```

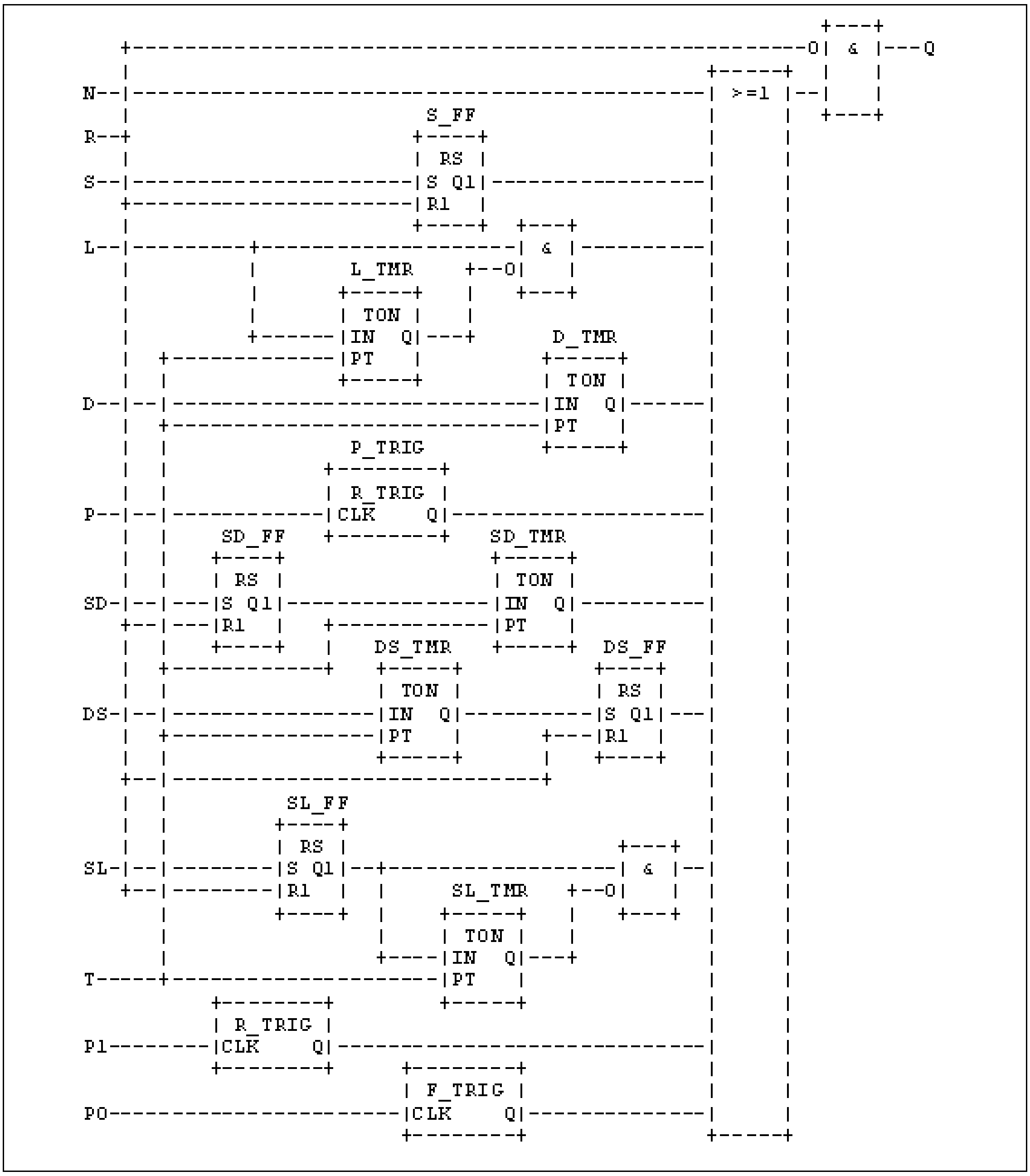
b) Without "final scan" logic (see Figure 31 b)

NOTE These interfaces are not visible to the user.

Figure 30 - ACTION_CONTROL function block - External interface

```
                                                          +---+
   +-----------------------------------------------------O| & |---Q
   |                                             +-----+  |   |
N--|---------------------------------------------| >=1 |--|   |
   |                       S_FF                  |     |  +---+
R--+                      +----+                 |     |
   |                      | RS |                 |     |
S--|----------------------|S Q1|-----------------|     |
   +----------------------|R1  |                 |     |
   |                      +----+  +---+          |     |
L--|---------+--------------------| & |----------|     |
   |         |       L_TMR    +--O|   |          |     |
   |         |      +-----+   |   +---+          |     |
   |         |      | TON |   |                  |     |
   |         +------|IN  Q|---+      D_TMR       |     |
   |  +-------------|PT   |         +-----+      |     |
   |  |             +-----+         | TON |      |     |
D--|--|-----------------------------|IN  Q|------|     |
   |  +-----------------------------|PT   |      |     |
   |  |              P_TRIG         +-----+      |     |
   |  |            +--------+                    |     |
   |  |            | R_TRIG |                    |     |
P--|--|------------|CLK    Q|--------------------|     |
   |  |    SD_FF   +--------+   SD_TMR           |     |
   |  |   +----+                +-----+          |     |
   |  |   | RS |                | TON |          |     |
SD-|--|---|S Q1|----------------|IN  Q|----------|     |
   +--|---|R1  |   +------------|PT   |          |     |
   |  |   +----+   |   DS_TMR   +-----+  DS_FF   |     |
   |  +------------+   +-----+          +----+   |     |
   |  |                | TON |          | RS |   |     |
DS-|--|----------------|IN  Q|----------|S Q1|---|     |
   |  +----------------|PT   |      +---|R1  |   |     |
   |  |                +-----+      |   +----+   |     |
   +--|-----------------------------+            |     |
   |  |         SL_FF                            |     |
   |  |        +----+                            |     |
   |  |        | RS |                     +---+  |     |
SL-|--|--------|S Q1|--+------------------| & |--|     |
   +--|--------|R1  |  |     SL_TMR   +--O|   |  +-----+
      |        +----+  |    +-----+   |   +---+
      |                |    | TON |   |
      |                +----|IN  Q|---+                     +-----+
T-----+---------------------|PT   |      +--------+         | >=1 |
                            +-----+      | F_TRIG |     Q---|     |---A
                +--------+           Q---|CLK    Q|---------|     |
                | R_TRIG |               +--------+         |     |
P1--------------|CLK    Q|----------------------------------|     |
                +--------+     +--------+                   |     |
                               | F_TRIG |                   |     |
P0-----------------------------|CLK    Q|-------------------|     |
                               +--------+                   +-----+
```

a) Body with “final scan” logic

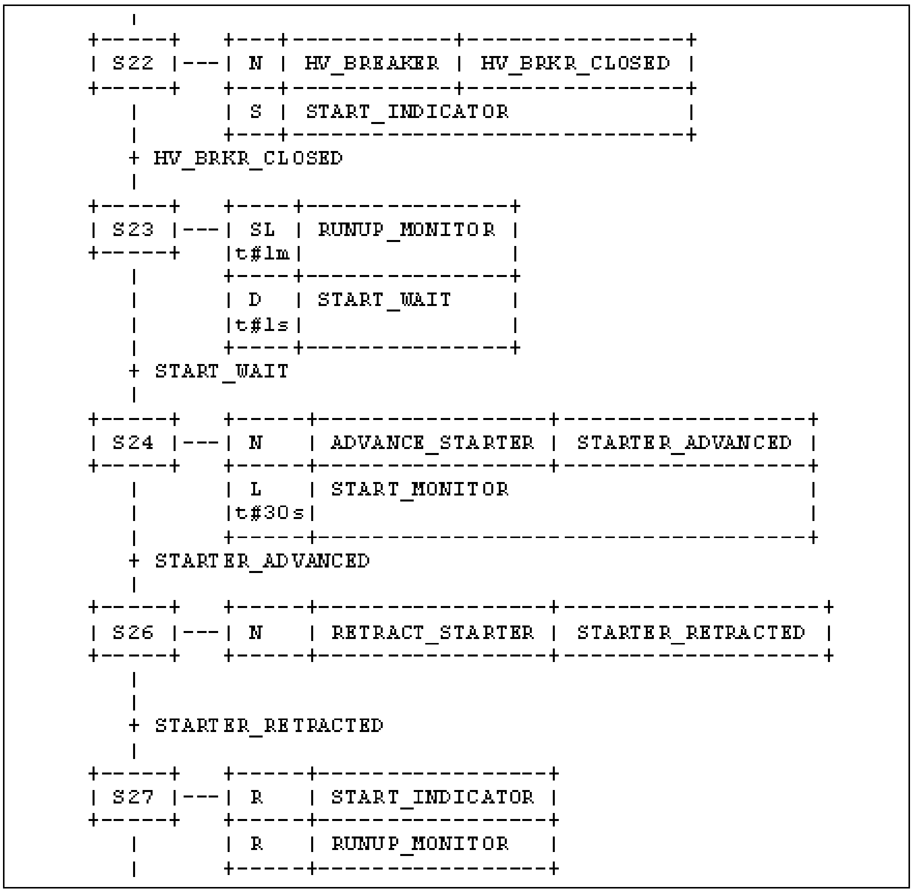


b) Body without “final scan” logic

NOTE 1 Instances of these function block types are not visible to the user.

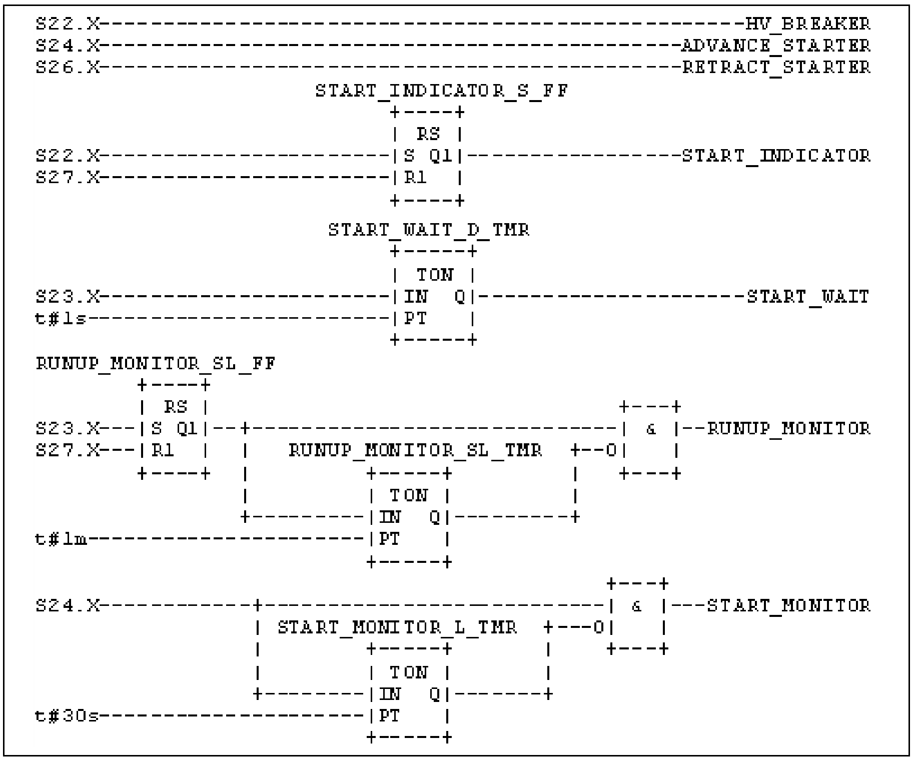
NOTE 2 The external interfaces of these function block types are given in Figure 30.

Figure 31 - ACTION_CONTROL function block body



```
      |
   +-----+   +---+------------+----------------+
   | S22 |---| N | HV_BREAKER | HV_BRKR_CLOSED |
   +-----+   +---+------------+----------------+
      |      | S | START_INDICATOR             |
      |      +---+-----------------------------+
      + HV_BRKR_CLOSED
      |
   +-----+   +----+---------------+
   | S23 |---| SL | RUNUP_MONITOR |
   +-----+   |t#1m|               |
      |      +----+---------------+
      |      | D  | START_WAIT    |
      |      |t#1s|               |
      |      +----+---------------+
      + START_WAIT
      |
   +-----+   +-----+-----------------+------------------+
   | S24 |---| N   | ADVANCE_STARTER | STARTER_ADVANCED |
   +-----+   +-----+-----------------+------------------+
      |      | L   | START_MONITOR                      |
      |      |t#30s|                                    |
      |      +-----+------------------------------------+
      + STARTER_ADVANCED
      |
   +-----+   +-----+-----------------+-------------------+
   | S26 |---| N   | RETRACT_STARTER | STARTER_RETRACTED |
   +-----+   +-----+-----------------+-------------------+
      |
      |
      + STARTER_RETRACTED
      |
   +-----+   +-----+-----------------+
   | S27 |---| R   | START_INDICATOR |
   +-----+   +-----+-----------------+
      |      | R   | RUNUP_MONITOR   |
      |      +-----+-----------------+
```

a) SFC representation



```
S22.X--------------------------------------------------HV_BREAKER
S24.X---------------------------------------------ADVANCE_STARTER
S26.X---------------------------------------------RETRACT_STARTER
                      START_INDICATOR_S_FF
                           +----+
                           | RS |
S22.X----------------------|S Q1|-----------------START_INDICATOR
S27.X----------------------|R1  |
                           +----+
                      START_WAIT_D_TMR
                           +-----+
                           | TON |
S23.X----------------------|IN  Q|---------------------START_WAIT
t#1s-----------------------|PT   |
                           +-----+
RUNUP_MONITOR_SL_FF
        +----+
        | RS |                                 +---+
S23.X---|S Q1|--+------------------------------| & |--RUNUP_MONITOR
S27.X---|R1  |  |   RUNUP_MONITOR_SL_TMR  +--O|   |
        +----+  |        +-----+          |    +---+
                |        | TON |          |
                +--------|IN  Q|----------+
t#1m---------------------|PT   |
                         +-----+
                                            +---+
S24.X------------+--------------------------| & |---START_MONITOR
                 | START_MONITOR_L_TMR  +---O|   |
                 |       +-----+        |    +---+
                 |       | TON |        |
                 +-------|IN  Q|--------+
t#30s--------------------|PT   |
                         +-----+
```

b) Functional equivalent

NOTE The complete SFC network and its associated declarations are not shown in this example.

Figure 32 - Action control example

Table 53 - Action control features [a]

No.	Description
1	per Figure 30 a) and Figure 31 a)
2	per Figure 30 b) and Figure 31 b)

[a] These two features are mutually exclusive, i.e., only one of the two shall be supported in a given SFC **implementation**.

6.6.5 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 clearing 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 54, are simultaneously true. The user may make provisions to avoid this error as shown in features 2b and 2c of Table 54.

Table 51 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 54.

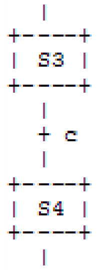
Testing of the successor transition condition(s) of an active step shall not be performed until the effects of the step activation have propagated throughout the program organization unit in which the step is declared.

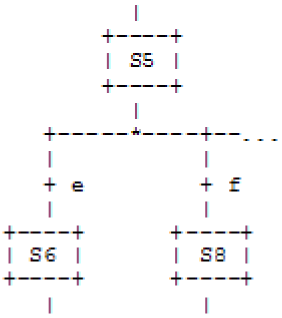
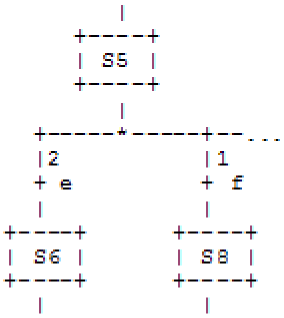
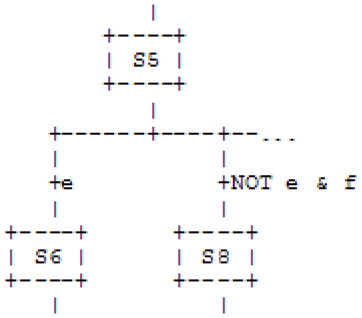
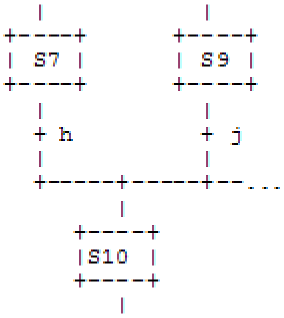
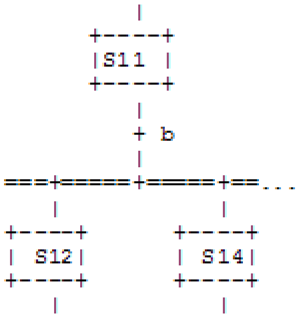
Figure 33 illustrates the application of these rules. In this figure, the active state of a step is indicated by the presence of an asterisk (*) in the corresponding block. This notation is used for illustration only, and is not a required language feature.

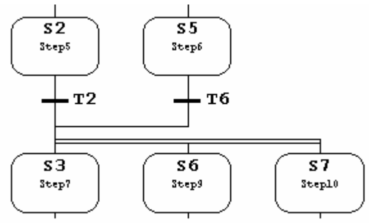
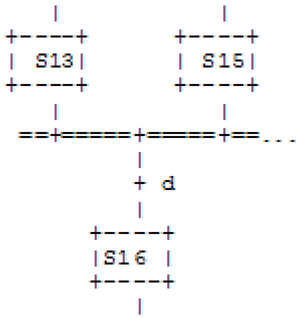
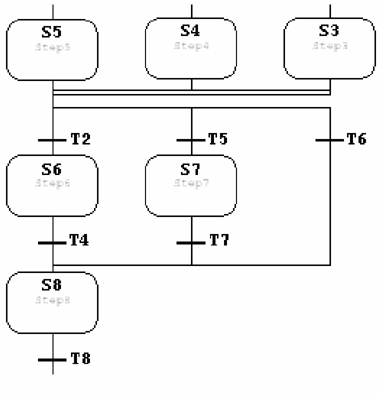
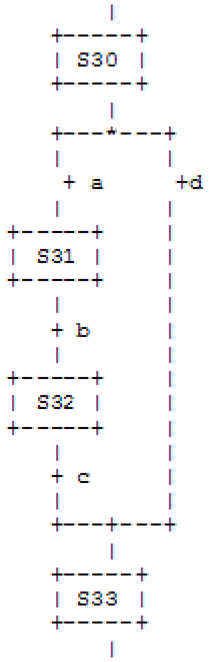
The application of the rules given in this subclause cannot prevent the formulation of “unsafe” SFCs, such as the one shown in Figure 34 a), which may exhibit uncontrolled proliferation of tokens. Likewise, the application of these rules cannot prevent the formulation of “unreachable” SFCs, such as the one shown in Figure 34 b), which may exhibit “locked up” behavior. The programmable controller system shall treat the existence of such conditions as **errors** as defined in 5.1.

The maximum allowed widths of the “divergence” and “convergence” constructs in Table 54 are **implementation dependencies.**

Table 54 - Sequence evolution

No.	Example	Rule
1		**Single sequence**: The alternation step-transition is repeated in series. EXAMPLE 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`.

No.	Example	Rule
2a		**Divergence of sequence selection:** 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. EXAMPLE 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.
2b		**Divergence of sequence selection:** The asterisk (" * "), followed by *numbered* branches, indicates a user-defined priority of transition evaluation, with the lowest-numbered branch having the highest priority. EXAMPLE 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 selection:** The connection (" + ") of the branch indicates that the user must assure that transition conditions are mutually exclusive. EXAMPLE 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 selection:** 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. EXAMPLE 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.
4a		**Simultaneous sequences – divergence:** The double horizontal line of synchronisation can be preceded by a single transition condition EXAMPLE 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.

No.	Example	Rule
4b	 S2 Step5 S5 Step6 T2 T6 S3 Step7 S6 Step9 S7 Step10	**Simultaneous sequences – divergence:** The double horizontal line of synchronisation can be preceded by **a sequence selection convergence** EXAMPLE 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 sequences – convergence:** **Double lines of simultaneous convergence can be followed by a single transition** EXAMPLE 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.
4d	 S5 Step5 S4 Step4 S3 Step3 T2 T5 T6 S6 Step6 S7 Step7 T4 T7 S8 Step8 T8	**Simultaneous sequences – convergence:** **Double lines of simultaneous convergence can be followed by a sequence selection divergence** EXAMPLE 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 5b 5c		**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. EXAMPLE (feature 5a shown) An evolution takes place from S30 to S33 if “a” is FALSE and d is TRUE, that is, the sequence (S31, S32) will be skipped.

<table>
<tr><th>No.</th><th>Example</th><th>Rule</th></tr>
<tr><td>6a
6b
6c</td><td><pre>
    |
+------+
| S30 |
+------+
    |
    + a
    |
    +----------+
    | |
+------+ |
| S31 | |
+------+ |
    | |
    + b |
    | |
+------+ |
| S32 | |
+------+ |
    | |
    *-----+ |
    | | |
    + c   + d |
    | | |
+------+ +----+
| S33 |
+------+
    |
</pre></td><td><b>Sequence loop:</b>
A “sequence loop” is a special case of sequence selection (feature 2) in which one or more of the branches returns to a preceding step. Features 6a, 6b, and 6c correspond to the representation options given in features 2a, 2b, and 2c, respectively.
EXAMPLE (feature 6a shown)
An evolution takes place from S32 to S31 if “c” is false and “d” is TRUE, that is, the sequence (S31, S32) will be repeated.</td></tr>
<tr><td>7</td><td><pre>
   |
+-----+
| S30 |
+-----+
   |
   + a
   |
   +----<----+
   | |
+-----+ |
| S31 | |
+-----+ |
   | |
   + b |
   | |
+-----+ |
| S32 | |
+-----+ |
   | |
   *-----+ |
   | | |
   + c   + d |
   | | |
+-----+ +->-+
| S33 |
+-----+
   |
</pre></td><td><b>Directional arrows:</b>
When necessary for clarity, the “less than” (<) character of the character set defined in 6.1.1 can be used to indicate right-to-left control flow, and the “greater than” (>) character to represent left-to-right control flow. When this feature is used, the corresponding character shall be located between two “-” characters, that is, in the character sequence “-<-” or “->-” as shown in the accompanying example.</td></tr>
</table>

```
   |                  |        |          |
+------+           +-----+  +-------+  +------+
|STEP10|           |STEP9|  |STEP13|   |STEP22|
|      |           |     |  |   *  |   |  *   |
+------+           +-----+  +-------+  +------+
   |                  |        |          |
   + X             ====+========+=========+====
   |                           |
+------+                       + X
|STEP11|                       |
|      |                  ====+====+===+====
+------+                      |        |
   |                      +-------+  +------+
                          |STEP15|   |STEP16|
                          |      |   |      |
                          +-------+  +------+
                              |         |
```

a) Transition not enabled (NOTE 2)

```
   |                  |        |          |
+------+           +-----+  +-------+  +-------+
|STEP10|           |STEP9|  |STEP13|   |STEP22|
|  *   |           |  *  |  |   *  |   |  *   |
+------+           +-----+  +-------+  +-------+
   |                  |        |          |
   + X             ===+========+=========+====
   |                           |
+------+                       + X
|STEP11|                       |
|      |                  ====+====+====+====
+------+                      |         |
   |                      +------+  +------+
                          |STEP15|  |STEP16|
                          |      |  |      |
                          +------+  +------+
                              |         |
```

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

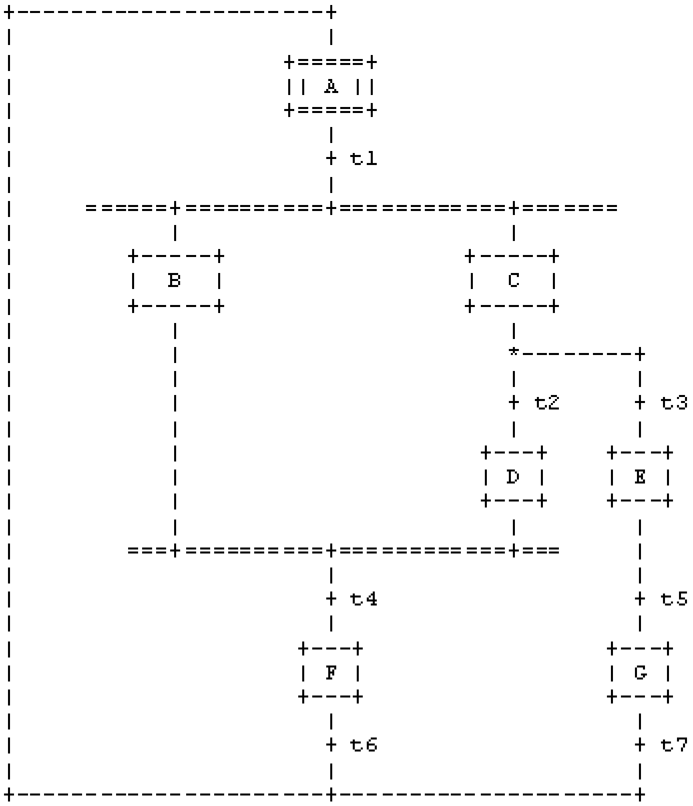
```
   |                  |        |          |
+------+           +-----+  +-------+  +-------+
|STEP10|           |STEP9|  |STEP13|   |STEP22|
|      |           |     |  |      |   |      |
+------+           +-----+  +-------+  +-------+
   |                  |        |          |
   + X             ====+========+=========+====
   |                           |
+------+                       + X
|STEP11|                       |
|  *   |                  ====+====+===+====
+------+                      |        |
   |                      +------+  +------+
                          |STEP15|  |STEP16|
                          |   *  |  |  *   |
                          +------+  +------+
                              |        |
```

c) Transition cleared (X = 1)

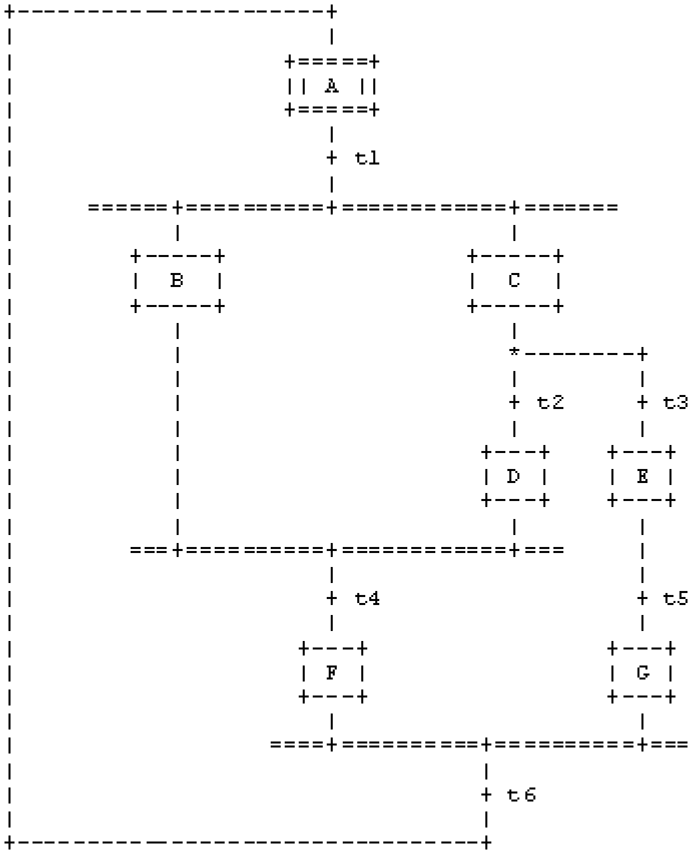
NOTE 1 In this figure, the active state of a step is indicated by the presence of an asterisk (*) in the corresponding block. This notation is used for illustration only, and is not a required language feature.

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

Figure 33 - Examples of SFC evolution rules



a) SFC error: an “unsafe” SFC



b) SFC error: an “unreachable” SFC

Figure 34 - Examples of SFC errors

6.6.6 Compatibility of SFC elements

SFCs can be represented graphically or textually, utilizing the elements defined above. Table 55 summarizes for convenience those elements which are mutually compatible for graphical and textual representation, respectively.

Table 55 - Compatible SFC features

Table	Graphical representation	Textual representation
Table 47	1, 3a, 3b, 4	2, 3a, 4
Table 48	1, 2, 3, 4, 4a, 4b, 7, 7a, 7b	5, 6, 7c, 7d
Table 49	1, 2l, 2s, 2f	3s, 3i
Table 50	1, 2, 4	3
Table 51	1 to 9	--
Table 52	1 to 10	1 to 10 (textual equivalent)
Table 53	1 to 7	1 to 6
Table 65	All	--

6.6.7 SFC compliance requirements

In order to claim compliance with the requirements of 5, the elements shown in Table 56 shall be supported and the compatibility requirements defined in 6.6.6 shall be fulfilled.

Table 56 - SFC minimal compliance requirements

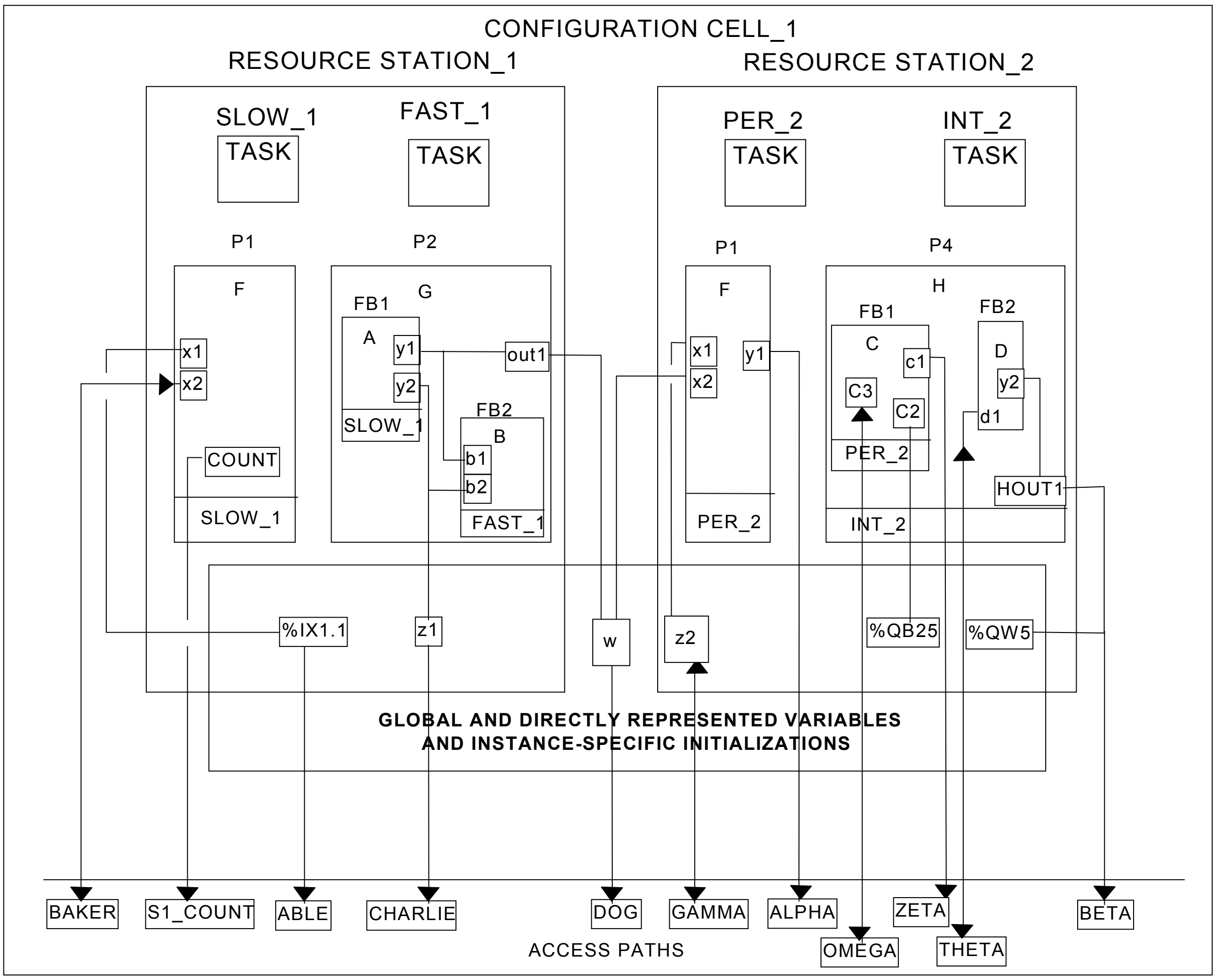
Table	Graphical representation	Textual representation
Table 47	1	2
Table 48	1 or 2 or 3 or (4 and (4a or 4b)) or (7 and (7a or 7b or 7c or 7d))	5 or 6
Table 49	1 or 2l or 2f	1 or 3s or 3i
Table 50	1 or 2 or 4	3
Table 51	1 or 2	1 or 2
Table 52	1 and (2a or 2b or 2c) and 3 and 4	Same (textual equivalent)
Table 53	(1 or 2) and (3 or 4) and (5 or 6) and (7 or 8) and (9 or 10) and (11 or 12)	Not required

6.7 Configuration elements

6.7.1 General

As described in 4.1, 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 subclause.

A graphic example of a simple configuration is shown in Figure 35 a). Skeleton declarations for the corresponding function blocks and programs are given in Figure 35 b). This figure serves as a reference point for the examples of configuration elements given in the remainder of subclause 6.7.



a) Graphical representation

```
FUNCTION_BLOCK A
  VAR_OUTPUT
    y1 : UINT ;
    y2 : BYTE ;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK B
  VAR_INPUT
    b1 : UINT ;
    b2 : 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 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 35 - Configuration example

6.7.2 Configurations, resources, and access paths

Table 57 enumerates the language features for declaration of *configurations, resources, global variables, access paths* and instance specific initializations. Partial enumeration of TASK declaration features is also given; additional information on *tasks* is provided in 6.7.3. The formal syntax for these features is given in B.2.7. Figure 35 provides examples of these features, corresponding to the example configuration shown in Figure 35 a) and the supporting declarations in Figure 35 b).

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 manufacturer shall supply an **implementation-dependent** *resource library* of such functions, 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 The RESOURCE...ON...END_RESOURCE construction is not required in a *configuration* with a single *resource*. See the production single_resource_declaration in B.2.7 for the syntax to be used in this case.

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 57.

The 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 as defined in 6.4.1, 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*), 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 The effect of using READ_WRITE access to function block output variables is **implementation-dependent.**

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 described in 6.4.1, 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 object to be located or initialized is concatenated at the end of the chain. All names in the concatenation shall be separated by dots. The location assignment or the initial value assignment follows the syntax and the semantics described in 6.4.

Instance specific initial values provided by the VAR_CONFIG...END_VAR construction always override 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 57 - Configuration and resource declaration features

No.	Description
1	CONFIGURATION...END_CONFIGURATION construction
2	VAR_GLOBAL...END_VAR construction within CONFIGURATION
3	RESOURCE...ON...END_RESOURCE construction
4	VAR_GLOBAL...END_VAR construction within RESOURCE
5a	Periodic TASK construction (see NOTE 1)
5b	Non-periodic TASK construction (see NOTE 1)
6a	WITH construction for PROGRAM to TASK association (see NOTE 1)
6b	WITH construction for Function Block to TASK association (see NOTE 1)
6c	PROGRAM declaration with no TASK association (see NOTE 1)
7	Declaration of directly represented variables in VAR_GLOBAL (see NOTE 2)
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 construction
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
11	VAR_CONFIG...END_VAR construction[a]
12a	VAR_GLOBAL CONSTANT in RESOURCE declarations
12b	VAR_GLOBAL CONSTANT in CONFIGURATION declarations
13a	VAR_EXTERNAL in RESOURCE declarations
13b	VAR_EXTERNAL CONSTANT in RESOURCE declarations

NOTE 1 See 6.7.3 for further descriptions of TASK features.

NOTE 2 See 6.4.2 for further descriptions of related features.

[a] This feature shall be supported if feature 10 in Table 15 is supported.

```
1      CONFIGURATION CELL_1
2        VAR_GLOBAL  w: UINT;  END_VAR
3        RESOURCE STATION_1 ON PROCESSOR_TYPE_1
4          VAR_GLOBAL  z1: BYTE;  END_VAR
5a         TASK SLOW_1(INTERVAL := t#20ms, PRIORITY := 2) ;
5a         TASK FAST_1(INTERVAL := t#10ms, PRIORITY := 1) ;
6a         PROGRAM P1 WITH SLOW_1 :
8a                      F(x1 := %IX1.1) ;
9b         PROGRAM P2 : G(OUT1 => w,
6b                        FB1 WITH SLOW_1,
6b                        FB2 WITH FAST_1)  ;
```

```
3      END_RESOURCE
3      RESOURCE STATION_2 ON PROCESSOR_TYPE_2
4        VAR_GLOBAL  z2      : BOOL ;
7                    AT %QW5 : INT  ;
4        END_VAR
5a       TASK PER_2(INTERVAL := t#50ms, PRIORITY := 2) ;
5b       TASK INT_2(SINGLE := z2,       PRIORITY := 1) ;
6a       PROGRAM P1 WITH PER_2 :
8b             F(x1 := z2, x2 := w)  ;
6a       PROGRAM P4 WITH INT_2 :
9a             H(HOUT1 => %QW5,
6b               FB1 WITH  PER_2);
3      END_RESOURCE
10a    VAR_ACCESS
10b      ABLE    : STATION_1.%IX1.1    : BOOL READ_ONLY  ;
10c      BAKER   : STATION_1.P1.x2     : UINT READ_WRITE ;
10d      CHARLIE : STATION_1.z1        : BYTE            ;
10e      DOG     : w                   : UINT READ_ONLY  ;
10f      ALPHA   : STATION_2.P1.y1     : BYTE READ_ONLY  ;
10f      BETA    : STATION_2.P4.HOUT1  : INT READ_ONLY   ;
10d      GAMMA   : STATION_2.z2        : BOOL READ_WRITE ;
10g      S1_COUNT : STATION_1.P1.COUNT : INT;
10h      THETA : STATION_2.P4.FB2.d1 : BOOL READ_WRITE;
10i      ZETA : STATION_2.P4.FB1.c1 : BOOL READ_ONLY;
10k      OMEGA : STATION_2.P4.FB1.C3 : INT READ_WRITE;
10a    END_VAR
11     VAR_CONFIG
         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
1    END_CONFIGURATION
```

NOTE 1 The numbers in the left-hand margin refer to the feature numbers in Table 57.

NOTE 2 Graphical and semigraphic representation of these features is allowed but is beyond the scope of this part of IEC 61131.

NOTE 3 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 36 - CONFIGURATION and RESOURCE declaration features (Example)

6.7.3 Tasks

For the purposes of this part of IEC 61131, 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 are **implementation dependencies**.

Tasks and their association with program organization units can be represented graphically or textually using the `WITH` construction, as shown in Table 58, as part of *resources* within *configurations*. A task is implicitly enabled or disabled by its associated resource according to the mechanisms defined in 4.1. 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 58, 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.

   NOTE 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 58, all program organization units meet their *deadlines*, that is, they all complete execution before being scheduled for re-execution. The manufacturer 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*, as defined in 4.1, 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.

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.

   EXAMPLE 1
   In the example represented by figure 21 a), when `Y2` is evaluated, the inputs `Y2.A` and `Y2.B` shall represent the outputs `Y1.C` and `Y1.D` from the same (not two different) evaluations of `Y1`.

   - 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.

   EXAMPLE 2
   In the example represented by Figure 37 b) and c), when `Y2` and `Y3` are evaluated in the normal course of

evaluating program P1, the inputs Y2.A and Y2.B shall be the results of the same evaluation of Y1 as the inputs Y3.A and Y3.B.

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** in the sense of 5.1 if a task fails to be scheduled or to meet its execution deadline because of excessive resource requirements or other task scheduling conflicts.

Table 58 - Task features

<table>
<tr><th>No.</th><th>Description/Examples</th></tr>
<tr><td>1a</td><td>Textual declaration of periodic TASK (feature 5a of Table 57)</td></tr>
<tr><td>1b</td><td>Textual declaration of non-periodic TASK (feature 5b of Table 57)</td></tr>
<tr><td colspan="2">Graphical representation of TASKs (general form)</td></tr>
<tr><td colspan="2"><pre>
       TASKNAME
       +---------+
       | TASK |
BOOL---|SINGLE |
TIME---|INTERVAL |
UINT---|PRIORITY |
       +---------+
</pre></td></tr>
<tr><td rowspan="2">2a</td><td>Graphical representation of periodic TASKs</td></tr>
<tr><td><pre>
          SLOW_1   FAST_1
         +---------+   +---------+
         | TASK | | TASK |
         |SINGLE | |SINGLE |
t#20ms---|INTERVAL | t#10ms---|INTERVAL |
     2---|PRIORITY | 1---|PRIORITY |
         +---------+   +---------+
</pre></td></tr>
<tr><td rowspan="2">2b</td><td>Graphical representation of non-periodic TASK</td></tr>
<tr><td><pre>
        INT_2
       +---------+
       | TASK |
%IX2---|SINGLE |
       |INTERVAL |
   1---|PRIORITY |
       +---------+
</pre></td></tr>
<tr><td>3a</td><td>Textual association with PROGRAMs (feature 6a of Table 57)</td></tr>
<tr><td>3b</td><td>Textual association with function blocks (feature 6b of Table 57)</td></tr>
<tr><td rowspan="2">4a</td><td>Graphical association with PROGRAMS</td></tr>
<tr><td><pre>
RESOURCE STATION_2
         P1   P4
     +--------+   +--------+
     | F | | H |
     | | | |
     | | | |
     +--------+   +--------+
     | PER_2 | | INT_2 |
     +--------+   +--------+
END_RESOURCE
</pre></td></tr>
<tr><td>4b</td><td>Graphical association with function blocks within PROGRAMS</td></tr>
</table>

No.	Description/Examples

```
RESOURCE STATION_1
   P2
   +---------------------------------------------------------+
   |                            G                            |
   |                                                         |
   |            FB1                      FB2                 |
   |          +-------+               +-------+              |
   |          |   A   |               |   B   |              |
   |          |       |               |       |              |
   |          |       |               |       |              |
   |          +-------+               +-------+              |
   |          |SLOW_1|                |FAST_1 |              |
   |          +-------+               +-------+              |
   +---------------------------------------------------------+
END_RESOURCE
```

No.	Description/Examples		
5a	Non-preemptive scheduling		
	EXAMPLE 1 - RESOURCE STATION_1 as configured in Figure 36 - Execution times: P1 = 2 ms; P2 = 8 ms - P2.FB1 = P2.FB2 = 2 ms (see NOTE 3) - 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
5a	Non-preemptive scheduling		
	EXAMPLE 2 - RESOURCE STATION_2 as configured in Figure 36 - Execution times: P1 = 30 ms, P4 = 5 ms, P4.FB1 = 10 ms (see OTE 4) - 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	

No.	Description/Examples		
	90	P4.FB1@2	P4@1
	95	P4@1	
	100	P1@2	P4.FB1@2
5b	Preemptive scheduling		
	EXAMPLE 3 - RESOURCE STATION_1 as configured in Figure 36 - Execution times: P1 = 2 ms; P2 = 8 ms; P2.FB1 = P2.FB2 = 2 ms (see NOTE 3) - 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
5b	Preemptive scheduling		
	EXAMPLE 4 - RESOURCE STATION_2 as configured in Figure 36 - Execution times: P1 = 30 ms, P4 = 5 ms, P4.FB1 = 10 ms (NOTE 4) - 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 Details of RESOURCE and PROGRAM declarations are not shown; see 6.7.2 and 6.7.3.

NOTE 2 The notation X@Y indicates that program organization unit X is scheduled or executing at priority Y.

NOTE 3 The execution times of P2.FB1 and P2.FB2 are not included in the execution time of P2.

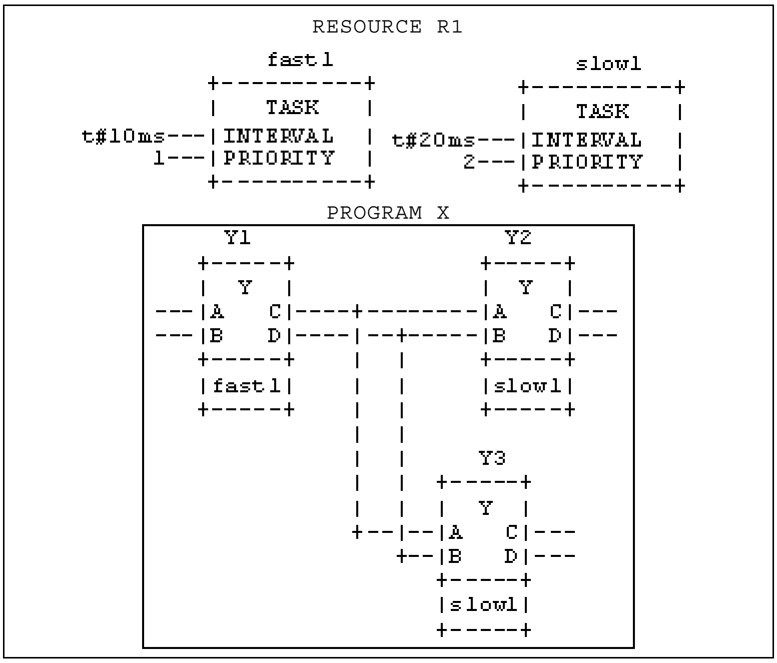
NOTE 4 The execution time of P4.FB1 is not included in the execution time of P4.

```
                RESOURCE R1
             fast1                 slow1
         +----------+          +----------+
         |   TASK   |          |   TASK   |
t#10ms---|INTERVAL  | t#20ms---|INTERVAL  |
     1---|PRIORITY  |      2---|PRIORITY  |
         +----------+          +----------+
                 PROGRAM X
           Y1                  Y2
         +-----+             +-----+
         |  Y  |             |  Y  |
      ---|A   C|----+--------|A   C|---
      ---|B   D|----|--+-----|B   D|---
         +-----+    |  |     +-----+
         |slow1|    |  |     |fast1|
         +-----+    |  |     +-----+
                    |  |
                    |  |    Y3
                    |  |  +-----+
                    |  |  |  Y  |
                    +--|--|A   C|---
                       +--|B   D|---
                          +-----+
                          |fast1|
                          +-----+
```

a) Function blocks with explicit task associations

```
                RESOURCE R1
             fast1                 slow1
         +----------+          +----------+
         |   TASK   |          |   TASK   |
t#10ms---|INTERVAL  | t#20ms---|INTERVAL  |
     1---|PRIORITY  |      2---|PRIORITY  |
         +----------+          +----------+
                 PROGRAM X
           Y1                  Y2
         +-----+             +-----+
         |  Y  |             |  Y  |
      ---|A   C|----+--------|A   C|---
      ---|B   D|----|--+-----|B   D|---
         +-----+    |  |     +-----+
         |fast1|    |  |
         +-----+    |  |
                    |  |
                    |  |    Y3
                    |  |  +-----+
                    |  |  |  Y  |
                    +--|--|A   C|---
                       +--|B   D|---
                          +-----+
                   slow1
```

b) Function blocks with implicit task associations



c) Explicit task associations equivalent to b)

NOTE The graphical representations in these figures are illustrative only and are not normative.

Figure 37 - Examples of task associations to function block instances

6.8 Namespaces

6.8.1 General

For the purposes of programmable controller programming languages, a *namespace* is a language element combining other language elements to a combined entity.

A name of a language element declared within a namespace may also be used within other namespaces or outside of any 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.

If the feature *namespace* is provided in a implementation this shall be defined in Table 46.

6.8.2 Declaration

A namespace declaration starts with the keyword NAMESPACE followed by the name of the namespace and ends with the keyword END_NAMESPACE. A namespace contains a sequence of *access areas*, each starting with one of the following keyword combinations:

- INTERNAL ACCESS for an access only within the namespace itself or
- PUBLIC ACCESS for an access also from outside the namespace

and ending with the keyword END_ACCESS.

An *access area* defines the access to the language elements it contains. An access area may contain the following language elements:

- Functions
- Function blocks

- Interfaces
- User defined data types
- Lists of global variables
- Namespaces

Methods of function blocks within an INTERNAL access area shall **not** have the method access specifier PUBLIC as defined in 6.5.4.2.4

EXAMPLE

Namespaces containing some function blocks.

```
NAMESPACE Standard
PUBLIC ACCESS

  NAMESPACE Timers

       INTERNAL ACCESS
         FUNCTION TimeTick: DWORD
          (*...declaration and operations deleted...*)
         END_FUNCTION
       END_ACCESS

       PUBLIC ACCESS
         FUNCTION_BLOCK TON
          (*... declaration and operations deleted...*)
         END_FUNCTION_BLOCK

         FUNCTION_BLOCK TOF
          (*... declaration and operations deleted...*)
         END_FUNCTION_BLOCK
       END_ACCESS

  END_NAMESPACE (*Timers*)

END_ACCESS
END_NAMESPACE (*Standard*)
```

6.8.3 Usage

Elements of a namespace within a PUBLIC ACCESS can be accessed from outside the namespace with the name of the namespace and a following “.”. This is not necessary from within the namespace but permitted.

Elements declared within an INTERNAL ACCESS can not be accessed from outside the namespace.

Elements in nested namespaces can only be accessed by naming all parent namespaces as shown in the example.

EXAMPLE
Usage of a Timer out of the Standard.Timers namespace

```
FUNCTION_BLOCK Uses_Timer
VAR
Ton1 : Standard . Timers . Ton;
       (* starts timer with rising edge, resets timer with falling edge *)
       bTest: BOOL;
END_VAR
Ton1(In := bTest, PT := t#5s);
END_FUNCTION_BLOCK
```

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) elements defined in 5 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 B.3. Subclause 7.3 defines the semantics of the ST language, whose syntax is given in B.4.

The textual elements specified in clause 6 shall be common to the textual languages (IL and ST) defined in this clause. In particular, the following program structuring elements 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
STEP...END_STEP
TRANSITION...END_TRANSITION
ACTION...END_ACTION
```

7.2 Instruction list (IL)

7.2.1 Instructions

As illustrated in Figure 38, 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 any of the data representations defined in 6.2 for literals, in 6.3 for enumerated values, and in 6.4 for variables.

The instruction can be preceded by an identifying *label* followed by a colon (:). Empty lines can be inserted between instructions.

```
LABEL   OPERATOR      OPERAND     COMMENT

START:   LD            %IX1        (* PUSH BUTTON    *)
         ANDN          %MX5        (* NOT INHIBITED *)
         ST            %QX2        (* FAN ON         *)
```

Figure 38 - Instruction fields (Example)

7.2.2 Operators, modifiers and operands

Standard operators with their allowed modifiers and operands shall be as listed in Table 59. The typing of operators shall conform to the conventions of 6.5.2.5.

Unless otherwise defined in Table 59, 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.

The modifier “N” indicates bitwise Boolean negation (one's complement) of the operand.

EXAMPLE 3 The instruction `ANDN %IX2` is interpreted as `result := result AND NOT %IX2`.

It shall be an **error** in the sense of 5.1 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 59, two equivalent forms of a parenthesized sequence of instructions are shown. Both features in Table 59 shall be interpreted as

```
result := result AND (%IX1 OR %IX2)
```

Table 59 - Parenthesized expression features for IL language

No.	DESCRIPTION/EXAMPLE
1	**Parenthesized expression beginning with explicit operator:**

	`AND(`
	`LD   %IX1    (NOTE 1)`
	`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 60 - Instruction list operators

No.	OPERATOR[a]	MODIFIERS (see NOTE)	SEMANTICS
1	LD	N	Set current result equal to operand
2	ST	N	Store current result to operand location
3	S[e]		Set operand to 1 if current result is Boolean 1
	R[e]		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
7a	NOT[d]		Logical negation (one's complement)
8	ADD	(	Addition
9	SUB	(	Subtraction
10	MUL	(	Multiplication
11	DIV	(	Division
11a	MOD	(	Modulo-division
12	GT	(	Comparison: >
13	GE	(	Comparison: >=
14	EQ	(	Comparison: =
15	NE	(	Comparison: <>
16	LE	(	Comparison: <=
17	LT	(	Comparison: <
18	JMP[b]	C, N	Jump to label
19	CAL[c]	C, N	Call function block (see Table 61)
20	RET[f]	C, N	Return from called function, function block or program
21	)		Evaluate deferred operation

NOTE See preceding text for explanation of modifiers and evaluation of expressions.

[a] Unless otherwise noted, these operators shall be either overloaded or typed as defined in 6.5.2.

[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 calle*d*.

[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.

7.2.3 Functions and function blocks

Functions as defined in 6.5.2 shall be called by placing the function name in the operator field. As shown in features 4 and 5 in Table 61 successful execution of a RET instruction or upon reaching the physical end of the function shall become the “current result” described in 7.2.2.

The argument list of functions (feature 4 in Table 61) is equivalent to feature 1 in Table 24. The rules and features defined in 6.5.2.2 and in Table 24 for function calls apply.

A non-formal input list of functions (feature 5 in Table 61) is equivalent to feature 2 in Table 24. The rules and features defined in 6.5.2.2 and Table 24 for function calls apply. In contrast to the examples given in Table 24 for ST language, the first argument is not contained in the non-formal input list in IL, but the current result shall be used as the first argument of the function. Additional arguments (starting with the 2nd), if required, shall be given in the operand field, separated by commas, in the order of their declaration.

Function blocks as defined in 6.5.3 can be called conditionally and unconditionally via the CAL (Call) operator listed in Table 60. As shown in features 1a, 1b, 2 and 3 of Table 57, this call can take one of four forms.

A formal argument list of a function block call (feature 1a in Table 61) is equivalent to feature 1 in Table 24. A non-formal argument list of a function block call (feature 1b in Table 61) is equivalent to feature 2 in Table 24. The rules and features defined in 6.5.2.2 and Table 24 for function calls applycorrespondingly, by replacing each occurrence of the term ‘function’ by the term ‘function block’ in these rules. All assignments in an argument list of a conditional function block call shall only be performed together with the call, if the condition is true.

Table 61 - Function block call and Function call features for IL language

No.	**DESCRIPTION/EXAMPLE** (NOTE 1)	
1a	CAL of function block with non-formal argument list:	
	CAL C10(%IX10, FALSE, A, OUT, B) CAL CMD_TMR(%IX5, T#300ms, OUT, ELAPSED)	
1b	CAL of function block with formal argument list:	

	CAL C10(
	CU := %IX10,
	R := FALSE,
	PV := A,
	Q => OUT
	CV => B)
	CAL CMD_TMR(
	IN := %IX5,
	PT := T#300ms,
	Q => OUT,
	ET => ELAPSED,
	ENO => ERR)
		(* alternate input names *)
		CAL C10(
		CU := %IX10,
		RESET := FALSE,
		PV := A,
		Q => OUT
		CV => B)
		CAL CMD_TMR(
		IN := %IX5,
		PT := T#300ms,
		Q => OUT,
		ET => ELAPSED,
		ENO => ERR)

2	CAL of function block with load/store of arguments (NOTE 2)	
	LD A ADD 5 ST C10. PV LD %IX10 ST C10. CU CAL C10	
3	Function call with formal argument list:	

	LIMIT(
	EN := COND,
	IN := B,
	MN := 1,
	MX := 5,
	ENO => TEMPL
	)
	ST A	

4	Function call with non-formal argument list:	
	LD 1 LIMIT B, 5 ST A	

No.	DESCRIPTION/EXAMPLE (NOTE 1)

NOTE 1 A declaration such as

```
VAR
  C10      : CTU;
  CMD_TMR : TON;
  A, B     : INT;
  ELAPSED : TIME;
  OUT, ERR, TEMPL, COND : BOOL;
END_VAR
```

is assumed in the above examples.

NOTE 2 This usage is an exception to the rule given in 6.5.3.2 that “The assignment of a value to the inputs of a function block is permitted only as part of the call of the function block.”

The input operators shown in Table 62 can be used in conjunction with feature 3 in Table 61. This method of call is equivalent to a CAL with an argument list, which contains only one variable with the name of the input operator. Arguments, 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 argument 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 62 - Standard function block input operators for IL language

No.	Operators	FB Type	Reference
4	S1,R	SR	6.5.3.5.2
5	S,R1	RS	6.5.3.5.2
6	CLK	R_TRIG	xx
8	CU,R,PV	CTU	6.5.3.5.4
9	CD,PV	CTD	6.5.3.5.4 (NOTE 1)
10	CU,CD,R,PV	CTUD	6.5.3.5.4 (NOTE 1)
11	IN,PT	TP	6.5.3.5.5
12	IN,PT	TON	6.5.3.5.5
13	IN,PT	TOF	6.5.3.5.5

NOTE 1 LD is not necessary as a Standard Function Block input operator, because the LD functionality is included in PV.

NOTE 2 The feature numbering in this table is such as to maintain consistency with the first edition of IEC 61131-3.

Arguments, 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 Expressions

In the ST language, the end of a textual line shall be treated the same as a space (SP) character, as defined in 6.4.2.

An *expression* is a construct which, when evaluated, yields a value corresponding to one of the data types defined in 6.3. The maximum allowed length of expressions is an **implementation dependency**..

Expressions are composed of operators and operands. An *operand* shall be a literal as defined in 6.2, an enumerated value as defined in 6.3.3, a variable as defined in 6.4, a function call as defined in 6.5.2, or another expression.

The *operators* of the ST language are summarized in Table 63. The evaluation of an expression consists of applying the operators to the operands in a sequence defined by the operator precedence shown in Table 63. The operator with highest precedence in an expression shall be applied first, followed by the operator of next lower precedence, etc., until evaluation is complete. Operators of equal precedence shall be applied as written in the expression from left to right.

EXAMPLE 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)
```

shall evaluate to -9, and

```
(A+B-C)*ABS(D)
```

shall evaluate to 0.

When an operator has two operands, the leftmost operand shall be evaluated first.

EXAMPLE 2

In the expression

```
SIN(A)*COS(B)
```

the expression SIN(A) shall be evaluated first, followed by COS(B), followed by evaluation of the product.

The following conditions in the execution of operators shall be treated as **errors** in the sense of 5.1:

1. An attempt is made to divide by zero.
2. Operands are not of the correct data type for the operation.
3. The result of a numerical operation exceeds the range of values for its data type.

Boolean expressions may be evaluated only to the extent necessary to determine the resultant value. For instance, if A<=B, then only the expression (A>B) would be evaluated to determine that the value of the expression

```
(A>B) & (C<D)
```

is Boolean zero.

Functions shall be called as elements of expressions consisting of the function name followed by a parenthesized list of arguments, as defined in 6.5.2.

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 in 6.5.2.5.

Table 63 - Operators of the ST language

No.	Operation[a]	Symbol	Precedence
1	Parenthesization	(expression)	HIGHEST
2	Function evaluation EXAMPLES	identifier(argument list) LN(A), MAX(X,Y), etc.	
4	Negation	-	
5	Complement	NOT	
3	Exponentiation[b]	**	
6	Multiply	*	
7	Divide	/	
8	Modulo	MOD	
9	Add	+	
10	Subtract	-	
11	Comparison	< , > , <= , >=	
12	Equality	=	
13	Inequality	<>	
14	Boolean AND	&	
15	Boolean AND	AND	
16	Boolean Exclusive OR	XOR	
17	Boolean OR	OR	LOWEST

NOTE The feature numbering in this table is such as to maintain consistency with the first edition of IEC 61131-3.

[a] The same restrictions apply to the operands of these operators as to the inputs of the corresponding functions defined in 6.5.2.6.

[b] The result of evaluating the expression A**B shall be the same as the result of evaluating the function EXPT(A,B) as defined in Table 26.

7.3.2 Statements

7.3.2.1 General

The statements of the ST language are summarized in Table 64. Statements shall be terminated by semicolons as specified in the syntax of B.4. The maximum allowed length of statements is an **implementation dependency**.

Table 64 - ST language statements

No.	Statement type/Reference	Examples
1	Assignment	`A := B;  CV := CV+1; C := SIN(X);`

2	Function block call and FB output usage (7.3.2.3)	`CMD_TMR(IN := %IX5, PT := T#300ms);`
		`A := CMD_TMR.Q ;`

3	RETURN	`RETURN;`

No.	Statement type/Reference	Examples

4	`IF …`
	`    THEN …`
	`      ELSIF …`
	`      THEN …`
	`    ELSE …`
	`END_IF`
		`D := B*B - 4.0*A*C;`
		`IF D < 0.0`
		`THEN NROOTS := 0;`
		`     ELSIF D = 0.0`
		`     THEN`
		`       NROOTS := 1;`
		`       X1 := - B/(2.0*A);`
		`     ELSE`
		`       NROOTS := 2 ;`
		`       X1 := (- B + SQRT(D))/(2.0*A);`
		`       X2 := (- B - SQRT(D))/(2.0*A);`
		`END_IF;`

5	`CASE … OF`
	`    …`
	`    ELSE …`
	`END_CASE`
		`TW := WORD_BCD_TO_INT(THUMBWHEEL);`
		`TW_ERROR := 0;`
		`CASE TW OF`
		`  1,5:  DISPLAY := OVEN_TEMP;`
		`  2:  DISPLAY := MOTOR_SPEED;`
		`  3:  DISPLAY := GROSS - TARE;`
		`  4,6..10: DISPLAY := STATUS(TW - 4);`
		`  ELSE DISPLAY := 0;`
		`       TW_ERROR := 1;`
		`END_CASE;`
		`QW100 := INT_TO_BCD(DISPLAY);`

6	`FOR .. TO.. BY … DO`
	`    …`
	`END_FOR`
		`J := 101 ;`
		`FOR I := 1 TO 100 BY 2 DO`
		`    IF WORDS[I] = 'KEY' THEN`
		`      J := I;`
		`      EXIT;`
		`    END_IF;`
		`END_FOR;`

7	`WHILE … DO`
	`    END_WHILE`
		`J := 1;`
		`WHILE J <= 100 & WORDS[J] <> 'KEY' DO`
		`  J := J+2;`
		`END_WHILE ;`

8	`REPEAT …`
	`      UNTIL …`
	`END_REPEAT`
		`J := -1;`
		`REPEAT`
		`  J := J+2;`
		`UNTIL J = 101 OR WORDS[J] = 'KEY'`
		`END_REPEAT;`

9 [a]	`EXIT`	`EXIT;` (see also in feature 6)
10	Empty Statement	`;`

11 [a]	`CONTINUE`	`J := 1;`
		`WHILE (J <= 100 AND WORDS[J] <> 'KEY') DO`
		`..IF (J MOD 3 = 0) THEN`
		`      CONTINUE;`
		`  END_IF;`
		`..(* Functionalities if j=1,2,4,5,7,8…*);`
		`   …`
		`   …`
		`END_WHILE ;`

[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**.

7.3.2.2 Assignment statements

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. However, if both A and B were of type ANALOG_CHANNEL_CONFIGURATION as described in Table 12, 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.

As illustrated in Figure 6, the assignment statement shall also be used to assign the value to be returned by a function, by placing the function name to the left of an assignment operator in the body of the function declaration. The value returned by the function shall be the result of the most recent evaluation of such an assignmen. It is an **error** to return from the evaluation of a function with an ENO value of TRUE, or with a non-existent ENO output, unless at least one such assignment has been made.

7.3.2.3 Function and function block control statements

Function and function block control statements consist of the mechanisms for calling function blocks and for returning control to the calling entity before the physical end of a function or function block.

Function evaluation shall be called as part of expression evaluation, as specified in 7.3.1.

Function blocks shall be called by a statement consisting of the name of the function block instance followed by a parenthesized list of arguments, as illustrated in Table 64. The rules and features defined in 6.5.2.2 and Table 24 for function calls apply correspondingly, by replacing each occurrence of the term ‘function’ by the term ‘function block’ in these rules.

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).

7.3.2.4 Selection statements

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 64.

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.

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

The maximum allowed number of selections in CASE statements is an **implementation dependency**.

7.3.2.5 Iteration statements

Iteration statements specify that the group of associated statements shall be executed repeatedly. The FOR statement is used if the number of iterations can be determined in advance; otherwise, the WHILE or REPEAT constructs are used.

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. For instance, after executing the statements shown in Figure 39, the value of the variable `SUM` shall be `15` if the value of the Boolean variable `FLAG` is `0`, and `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 ;
```

Figure 39 - EXIT statement (Example)

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`). For instance, after executing the statements shown in Figure 39a, the value of the variable `SUM` shall be `15` if the value of the Boolean variable `FLAG` is `0`, and `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 ;
```

Figure 40 - CONTINUE statement (Example)

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; this value defaults to 1. The iteration is terminated when the value of the control variable is outside the range specified by the TO construct.

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 initial value exceeds the final value. The value of the control variable after completion of the `FOR` loop is **implementation-dependent**.

An example of the usage of the `FOR` statement is given in feature 6 of Table 64. 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.

The `WHILE` statement causes the sequence of statements up to the `END_WHILE` keyword to be executed repeatedly until the associated Boolean expression is false. If the expression is initially false, then the group of statements is not executed at all. For instance, the `FOR...END_FOR` example given in Table 64 can be rewritten using the `WHILE...END_WHILE` construction shown in Table 64.

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 given in Table 64 can be rewritten using the `REPEAT...END_REPEAT` construction shown in Table 64.

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** in the sense of 5.1 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.

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 defined in 8.1.2 and 8.1.3 apply to both the graphic languages in this standard, that is, LD (Ladder Diagram) and FBD (Function Block Diagram), and to the graphic representation of sequential function chart (SFC) elements.

8.1.2 Representation of lines and blocks

The graphic language elements defined in this clause are drawn with line elements using characters from the character set defined in 6.1.1, or using graphic or semi-graphic elements, as shown in Table 65.

Lines can be extended by the use of *connectors* as shown in Table 65. 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 **implementation dependencies**.

Table 65 - Representation of lines and blocks

No.	Feature	Example
Horizontal lines		-----
1	ISO/IEC 10646-1 “minus” character	
2	Graphic or semi-graphic	
Vertical lines		|
3	ISO/IEC 10646-1 “vertical line” character	
4	Graphic or semi-graphic	

Horizontal/vertical connection		`  |`
		`--+--`
		`  |`

5	ISO/IEC 10646-1 “plus” character	
6	Graphic or semi-graphic	

Line crossings without connection		`    |`
		`----|----`
		`    |`

7	ISO/IEC 10646-1 characters	
8	Graphic or semi-graphic	

Connected and non-connected corners		`    |   |`
		`----+   +----`
		`    |   |`
		`----+-+ +----`
		`    | | |`

9	ISO/IEC 10646-1 characters	
10	Graphic or semi-graphic	

Blocks with connecting lines		`      |`
		`  +--------+`
		`---|        |`
		`   |        |---`
		`---|        |`
		`  +--------+`
		`      |`

11	ISO/IEC 10646-1 characters	
12	Graphic or semi-graphic	

Connectors and continuation		`---------->OTTO>`
		`>OTTO>----------`

13	ISO/IEC 10646-1	
14	Graphic or semi-graphic connectors	

8.1.3 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 defined in 8.2. 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 as defined in 6.1.2 or an unsigned decimal integer as defined in 6.2.1. The *scope* of a network and its label shall be *local* to the program organization unit in which the network is located. Examples of networks and network labels are shown in annex F.

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;
- Signal flow”, analogous to the flow of signals between elements of a signal processing system, typically used in function block diagrams;
- 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.

The appropriate conceptual quantity shall flow along lines between elements of a network according to the following rules:

1) Power flow in the LD language shall be from left to right.

2) 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.

3) 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).

8.1.4 Evaluation of networks

The order in which networks and their elements are evaluated is not necessarily the same as the order in which they are labelled 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:

1. No element of a network shall be evaluated until the states of all of its inputs have been evaluated.
2. The evaluation of a network element shall not be complete until the states of all of its outputs have been evaluated.
3. 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 defined in 8.1.5.
4. The order in which networks are evaluated shall conform to the provisions of 8.2.7 for the LD language and 8.3.3 for the FBD language.

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 in Figure 41. A feedback variable can also be an output element of a function block data structure as defined in 6.5.3.

Feedback paths can be utilized in the graphic languages defined in 8.2 and 8.3, subject to the following rules:

1. Explicit loops such as the one shown in Figure 41 a) shall only appear in the FBD language defined in 8.3.
2. It shall be possible for the user to utilize an **implementation-dependent** 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 Figure 41 b).
3. Feedback variables shall be initialized by one of the mechanisms defined in 6.4. 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.
4. 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.

```
                      +---+
             ENABLE---| & |-----RUN---+
                  +---|   |           |
         +----+    |   +---+           |
START1---|>=1 |---+                   |
START2---|    |                       |
      +--|    |                       |
      |  +----+                       |
      +-------------------------------+
```

a) Explicit loop

```
                      +---+
             ENABLE---| & |-----RUN
                  +---|   |
         +----+    |   +---+
START1---|>=1 |---+
START2---|    |
   RUN---|    |
         +----+
```

b) Implicit loop

```
|   START1     ENABLE      RUN    |
+---| |----+---| |------( )---+
|   START2   |                    |
+---| |----+                    |
|    RUN     |                    |
+---| |----+                    |
|                                 |
```

c) LD language equivalent

Figure 41 - Feedback path (Example)

8.1.5 Execution control elements

Transfer of program control in the LD and FBD languages shall be represented by the graphical elements shown in Table 66.

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 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 66. 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 66.

Table 66 - Graphic execution control elements

No.	Symbol/Example	Explanation
	Unconditional jump	
1	`1---->>LABELA`	FBD language

2	`|`
	`+---->>LABELA`
	`|`
		LD language

	Conditional jump	

3	`X---->>LABELB`
	`          +---+`
	`%IX20---| & |--->>NEXT`
	`%MX50---|   |`
	`          +---+`
	`NEXT:`
	`          +---+`
	`%IX25---|>=1|---%QX100`
	`%MX60---|   |`
	`          +---+`
		(FBD language)
		Example:
		jump condition
		
		jump target

<table>
<tr><th>No.</th><th>Symbol/Example</th><th>Explanation</th></tr>
<tr><td>4</td><td>| X
+-| |---->>LABELB
|
|
| %IX20   %MX50
+---| |-----| |--->>NEXT
|
|
NEXT:
| %IX25   %QX100 |
+----| |----+----( )---+
| %MX60 | |
+----| |----+ |
| |</td><td>LD language
Example:
jump condition
jump target</td></tr>
<tr><td colspan="3">Conditional return</td></tr>
<tr><td>5</td><td>| X
+--| |---<RETURN>
|</td><td>LD language</td></tr>
<tr><td>6</td><td>X---<RETURN></td><td>FBD language</td></tr>
<tr><td colspan="3">Unconditional return</td></tr>
<tr><td>7</td><td>END_FUNCTION
END_FUNCTION_BLOCK</td><td>from FUNCTION
from FUNCTION_BLOCK</td></tr>
<tr><td>8</td><td>|
+---<RETURN>
|</td><td>LD language</td></tr>
</table>

8.2 Ladder diagram (LD)

8.2.1 General

This subclause 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*.

8.2.2 Power rails

As shown in Table 67, 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.

Table 67 - Power rails

<table>
<tr><th>No.</th><th>Symbol</th><th>Description</th></tr>
<tr><td>1</td><td>|
+---
|</td><td>Left power rail
(with attached horizontal link)</td></tr>
<tr><td>2</td><td> |
---+
   |</td><td>Right power rail
(with attached horizontal link)</td></tr>
</table>

8.2.3 Link elements and states

As shown in Table 68, 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 68 - Link elements

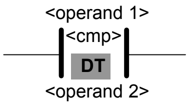
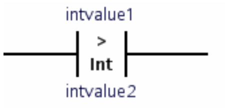
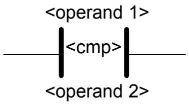
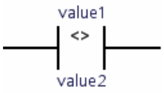
No.	Symbol	Description
1	-----------	Horizontal link
2	| ----+---- ----+ | +----	Vertical link (with attached horizontal links)

8.2.4 Contacts

A *contact* is an element which imparts a state to the horizontal link on its right side which is equal to the Boolean AND of the state of the horizontal link at its left side with an appropriate function of an associated Boolean input, output, or memory variable. A contact does not modify the value of the associated Boolean variable. Standard contact symbols are given in Table 69.

Table 69 - Contacts [a]

Static contacts		
No.	**Symbol**	**Description**
Normally open contact		
1a	*** --| |--	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.
1b	*** --! !--	semigraphic
Normally closed contact		
2a	*** --|/|--	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.
2b	*** --!/!--	semigraphic
Transition-sensing contacts		

3	*** --|P|--	Positive transition-sensing contact 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	*** --!P!--	semigrafic
5	*** --|N|--	Negative transition-sensing contact 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.
6	*** --!N!--	semigrafic
7		Compare contact (typed) 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. <cmp> may be substituted by one of the compare functions that are valid for the given data type (see.table 32). DT is the data type of both given operands. Example:  If the left link is ON and (intvalue1 > intvalue2) the right link switches to ON . Both intvalue1 and intvalue2 are of the data type INT
8		Compare contact (overloaded) The state of the right link is ON from one evaluation of this element to the next when the left link is ON and the <cmp> result of the operands 1 and 2 is true. The state of the right link shall be OFF otherwise. <cmp> may be substituted by one of the compare functions that are valid for the operands data type (see.table 32). Example:  If the left link is ON and (value1 <> value2) the right link switches to ON .
[a] As specified in 6.2, the exclamation mark “!” shall be used when a national character set does not support the vertical bar “|”.		

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 70.

EXAMPLE

In the rung shown below, the value of the Boolean output a is always TRUE, while the value of outputs c, d und e upon completion of an evaluation of the rung is equal to the value of the input b.

```
|    a      b        c     d     |
+--( )--| |--+--( )---( )--+
|            |        e      |
|            +-----( )-----+
```

Table 70 - Coils

No.	Symbol	Description
Momentary coils		

1	*** --( )--	Coil The state of the left link is copied to the associated Boolean variable and to the right link.
2	*** --(/)--	Negated coil The state of the left link is copied to the right link. The inverse of the state of the left link is copied to the associated Boolean variable, that is, if the state of the left link is `OFF`, then the state of the associated variable is `ON`, and vice versa.
Latched Coils		
3	*** --(S)--	SET (latch) coil 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	*** --(R)--	RESET (unlatch) coil 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	*** --(P)--	Positive transition-sensing coil 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	*** --(N)--	Negative transition-sensing coil 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 and function blocks in the LD language shall be as defined in 6.5.2 and 6.5.3, with the following exceptions:

1) 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.
2) 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 defined in 8.1.5.

8.3 Function Block Diagram (FBD)

8.3.1 General

This subclause 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.

The provisions of 6 and 8.1 shall apply to the construction and interpretation of programmable controller programs 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 8.1.2.

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 Figure 42.

```
|   a      c   |
+---||--+--()--+
|  b    |      |
+--||---+      |
|              |
```

a) “Wired-OR” in LD language

```
    +-----+
a---| >=1 |---c
b---|     |
    +-----+
```

b) Function in FBD language

Figure 42 - Boolean OR (Example)

8.3.3 Order of network evaluation

When a program organization unit written in the FBD language contains more than one network, the manufacturer shall provide **implementation-dependent** means by which the user may determine the order of execution of networks.

Annex A
(normative)
Specification method for textual languages

A.1 Syntax

A.1.1 Terminal symbols

A syntax is defined by a set of *terminal symbols* to be utilized for program specification; a set of *non-terminal symbols* defined in terms of the terminal symbols; and a set of *production rules* specifying those definitions.

The terminal symbols for textual programmable controller programs shall consist of combinations of the characters in the character set defined in 6.1.1.

For the purposes of this part, terminal textual symbols consist of the appropriate character string enclosed in paired single or double quotes.

EXAMPLE 1

A terminal symbol represented by the character string ABC can be represented by either `"ABC"` or `'ABC'`.

This allows the representation of strings containing either single or double quotes.

EXAMPLE 2

A terminal symbol consisting of the double quote itself would be represented by `'"'`.

A special terminal symbol utilized in this syntax is the end-of-line delimiter, which is represented by the unquoted character string EOL. This symbol shall normally consist of the “paragraph separator” character defined as hexadecimal code 2029 by ISO/IEC 10646-1.

A second special terminal symbol utilized in this syntax is the “null string”, that is, a string containing no characters. This is represented by the terminal symbol `NIL`.

The case of letters shall not be significant in terminal symbols.

A.1.2 Non-terminal symbols

Non-terminal textual symbols shall be represented by strings of lower-case letters, numbers, and the underline character (_), beginning with a lower-case letter.

EXAMPLE

The strings `nonterm1` and `non_term_2` are valid non-terminal symbols, while the strings `3nonterm` and `_nonterm4` are not.

A.1.3 Production rules

The production rules for textual programmable controller programming languages shall form an *extended grammar* in which each rule has 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:

1) The null string, `NIL`, is an extended structure.

2) A terminal symbol is an extended structure.

3) A non-terminal symbol is an extended structure.

4) 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`], *option*, meaning zero or one occurrence of `S`.

5) If `S1` and `S2` are extended structures, 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`.

6) Concatenation *precedes* alternation, that is,

`S1 | S2 S3` is equivalent to `S1 | (S2 S3)`,
and `S1 S2 | S3` is equivalent to `(S1 S2) | S3`.

A.2 Semantics

Programmable controller textual programming language semantics are defined in this part of IEC 61131 by appropriate natural language text, accompanying the production rules, which references the descriptions provided in the appropriate clauses. Standard options available to the user and manufacturer are specified in these semantics.

In some cases it is more convenient to embed semantic information in an extended structure. In such cases, this information is delimited by paired angle brackets, for example, `<semantic information>`.

Annex B
(normative)
Formal specifications of language elements

B.1 Programming model

The contents of this annex are normative in the sense that a compiler which is capable of recognizing all the syntax in this annex shall be capable of recognizing the syntax of any textual language implementation complying with this standard.

PRODUCTION RULES:

```
library_element_name ::= data_type_name | function_name
   | function_block_type_name | program_type_name
   | resource_type_name | configuration_name

library_element_declaration ::= data_type_declaration
   | function_declaration | function_block_declaration
   | program_declaration | configuration_declaration
```

SEMANTICS: These productions reflect the basic programming model defined in 4.3, where *declarations* are the basic mechanism for the production of named *library elements*. The syntax and semantics of the non-terminal symbols given above are defined in the subclauses listed below.

Non-terminal symbol	Syntax	Semantics
data_type_name data_type_declaration	B.2.3	6.3
function_name function_declaration	B.2.5.1	6.5.2
function_block_type_name function_block_declaration	B.2.5.2	xxx
program_type_name program_declaration	B.2.5.3	0
resource_type_name configuration_name configuration_declaration	B.2.7	6.7

B.2 Common elements

B.2.1 Letters, digits and identifiers

SEMANTICS:

The ellipsis <...> here indicates the ISO/IEC 10646-1 sequence of 26 letters.

Characters from national character sets can be used; however, international portability of the printed representation of programs cannot be guaranteed in this case.

PRODUCTION RULES:

```
letter          ::=  'A' | 'B' | <...> | 'Z' | 'a' | 'b' | <...> | 'z'
digit           ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8'
                     | '9'
octal_digit     ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
hex_digit       ::= digit | 'A'|'B'|'C'|'D'|'E'|'F'
identifier      := (letter | ('_' (letter | digit))) {['_'] (letter
                           | digit)}
```

B.2.2 Constants

B.2.2.1 General

SEMANTICS: The external representations of data described in 6.2 are designated as “constants” in this annex.

PRODUCTION RULE:

```
  constant ::= numeric_literal | character_string | time_literal
    | bit_string_literal | boolean_literal
```

B.2.2.2 Numeric literals

SEMANTICS: see 6.2.1.

PRODUCTION RULES:

```
numeric_literal    ::= integer_literal | real_literal

integer_literal    ::= [ integer_type_name '#' ](signed_integer
                         | binary_integer | octal_integer | hex_integer)

signed_integer     ::= ['+' |'-'] unsigned_integer

unsigned_integer   ::= digit {['_'] digit}

binary_integer     ::= '2#' bit {['_'] bit}

bit                ::= '1' | '0'

octal_integer      ::= '8#' octal_digit {['_'] octal_digit}

hex_integer        ::= '16#' hex_digit {['_'] hex_digit}

real_literal       ::= [ real_type_name '#' ]signed_integer  '.'
                         integer [exponent]

exponent           ::= ('E' | 'e') ['+'|'-'] integer

bit_string_literal ::= [ ('BYTE' | 'WORD' | 'DWORD' | 'LWORD') '#' ]
                         ( unsigned_integer | binary_integer
                         | octal_integer | hex_integer)

boolean_literal    ::= [ 'BOOL#' ] ( '1' | '0' | 'TRUE' | 'FALSE')
```

B.2.2.3 Character strings

SEMANTICS: see 6.2.2.

PRODUCTION RULES:

```
character_string   ::= single_byte_character_string
                         | double_byte_character_string

single_byte_character_string
                   ::= "'" {single_byte_character_representation} "'"


double_byte_character_string
                   ::= '"' {double_byte_character_representation} '"'

single_byte_character_representation
                   ::= common_character_representation | "$'" | '"'
                         | '$' hex_digit hex_digit

double_byte_character_representation
                   ::= common_character_representation |  '$"'
                         | "'"| '$' hex_digit hex_digit hex_digit hex_digit

common_character_representation
                   ::= <any printable character except '$', '"' or "'">
                         | '$$' | '$L' | '$N' | '$P' | '$R' | '$T'
                         | '$l' | '$n' | '$p' | '$r' | '$t'
```

B.2.2.4 Time literals

B.2.2.4.1 General

SEMANTICS: see 6.2.3.

PRODUCTION RULE:

```
time_literal ::= duration | time_of_day | date | date_and_time
```

B.2.2.4.2 Duration

SEMANTICS: see 6.2.3.2.

NOTE The semantics of impose additional constraints on the allowable values of `hours`, `minutes`, `seconds`, and `milliseconds`.

PRODUCTION RULES:

```
duration        ::= ('T' | 'TIME') '#' ['-'] interval
interval        ::= days | hours | minutes | seconds | milliseconds
days            ::= fixed_point ('d') | integer ('d') ['_'] hours
fixed_point     ::= integer [ '.' integer]
hours           ::= fixed_point ('h') | integer ('h') ['_'] minutes
minutes         ::= fixed_point ('m')  | integer ('m') ['_'] seconds
seconds         ::= fixed_point ('s') | integer ('s') ['_'] milliseconds
milliseconds    ::= fixed_point ('ms')
```

B.2.2.4.3 Time of day and date

SEMANTICS: see 6.2.3.2.

NOTE The semantics impose additional constraints on the allowable values of `day_hour`, `day_minute`, `day_second`, `year`, `month`, and `day`.

PRODUCTION RULES:

```
time_of_day     ::= ('TIME_OF_DAY' | 'TOD')  '#' daytime
daytime         ::= day_hour ':' day_minute ':' day_second
day_hour        ::= integer
day_minute      ::= integer
day_second      ::= fixed_point
date            ::= ('DATE' | 'D') '#' date_literal
date_literal    ::= year '-' month '-' day
year            ::= integer
month           ::= integer
day             ::= integer
date_and_time   ::= ('DATE_AND_TIME' | 'DT') '#' date_literal '-' daytime
```

B.2.3 Data types

B.2.3.1 General

SEMANTICS: see 6.3.

PRODUCTION RULES:

```
data_type_name          ::= non_generic_type_name | generic_type_name
non_generic_type_name   ::=  elementary_type_name | derived_type_name
```

B.2.3.2 Elementary data types

SEMANTICS: See 6.3.2.

PRODUCTION RULES:

```
elementary_type_name           ::= numeric_type_name | date_type_name
                                     | bit_string_type_name
                                     | 'STRING' | 'WSTRING' | 'TIME'

numeric_type_name              ::= integer_type_name | real_type_name

integer_type_name              ::= signed_integer_type_name
                                     | unsigned_integer_type_name

signed_integer_type_name       ::= 'SINT'  | 'INT' | 'DINT' | 'LINT'

unsigned_integer_type_name     ::= 'USINT' | 'UINT' | 'UDINT'  | 'ULINT'

real_type_name                 ::= 'REAL'  | 'LREAL'

date_type_name                 ::= 'DATE'  | 'TIME_OF_DAY' | 'TOD'
                                           | 'DATE_AND_TIME' | 'DT'

bit_string_type_name           ::= 'BOOL' | 'BYTE' | 'WORD' | 'DWORD' | 'LWORD'
```

B.2.3.3 Generic data types

SEMANTICS: see 6.3.2.

PRODUCTION RULE:

```
generic_type_name  ::= 'ANY' | 'ANY_DERIVED' | 'ANY_ELEMENTARY'
      | 'ANY_MAGNITUDE' | 'ANY_NUM' | 'ANY_REAL' | 'ANY_INT' | 'ANY_BIT'
      | 'ANY_STRING' | 'ANY_DATE'
```

B.2.3.4 Derived data types

SEMANTICS: see 6.3.3.

PRODUCTION RULES:

```
derived_type_name        ::= single_element_type_name | array_type_name
                               | structure_type_name | string_type_name

single_element_type_name ::= simple_type_name | subrange_type_name
                               | enumerated_type_name

simple_type_name         ::= identifier

subrange_type_name       ::= identifier

enumerated_type_name     ::= identifier

array_type_name          ::= identifier

structure_type_name      ::= identifier

data_type_declaration    ::= 'TYPE' type_declaration ';' {type_declaration ';'}
                               'END_TYPE'

type_declaration         ::= single_element_type_declaration
                               | array_type_declaration
                               | structure_type_declaration
                               | string_type_declaration

single_element_type_declaration ::= simple_type_declaration
   | subrange_type_declaration | enumerated_type_declaration

simple_type_declaration        ::= simple_type_name ':' simple_spec_init

simple_spec_init               ::= simple_specification [':=' constant]

simple_specification           ::= elementary_type_name | simple_type_name

subrange_type_declaration      ::= subrange_type_name ':' subrange_spec_init

subrange_spec_init             ::= subrange_specification [':=' signed_integer]
```

```
subrange_specification   ::= integer_type_name '(' subrange')'
                               | subrange_type_name

subrange                 ::= signed_integer '..' signed_integer

enumerated_type_declaration
                         ::= enumerated_type_name ':' enumerated_spec_init

enumerated_spec_init     ::= enumerated_specification
                               [':=' enumerated_value]

enumerated_specification ::= ('(' enumerated_value_spec
                               {',' enumerated_value_spec} ')')
                               | enumerated_type_name

enumerated_value_spec    ::= identifier

enumerated_value         ::= [enumerated_type_name '#'] identifier

array_type_declaration   ::= array_type_name ':' array_spec_init

array_spec_init          ::= array_specification
                               [':=' array_initialization]

array_specification      ::= array_type_name
                               | 'ARRAY' '[' subrange {',' subrange} ']'
                               'OF' non_generic_type_name

array_initialization     ::='[' array_initial_elements
                               {',' array_initial_elements} ']'

array_initial_elements   ::= array_initial_element
                               | integer '(' [array_initial_element] ')'

array_initial_element    ::= constant | enumerated_value
                               | structure_initialization
                               | array_initialization

structure_type_declaration
                         ::= structure_type_name ':'
                               structure_specification

structure_specification  ::= structure_declaration
                               | initialized_structure

initialized_structure    ::= structure_type_name
                               [':=' structure_initialization]

structure_declaration    ::= 'STRUCT' [structure_options]
                               structure_element_declaration ';'
                               {structure_element_declaration ';'}
                               'END_STRUCT'

structure_options        ::= 'LAYOUT_EXPLICIT' ( 'LITTLE_ENDIAN'
                                     | 'BIG_ENDIAN') ['OVERLAP']

structure_element_declaration
                         ::= structure_element_name
                               [relative_location]':'
                               (simple_spec_init | subrange_spec_init
                               | enumerated_spec_init
                               | array_spec_init
                               | initialized_structure)

structure_element_name   ::= identifier

relative_location        ::= 'AT' ( '%B'byte_location['.'bit8_location]
                               | '%W'word_location['.'bit16_location])

byte_location            ::= unsigned_integer

word_location            ::= unsigned_integer

bit8_location            ::= '0' .. '7'

bit16_location           ::= '0' .. '15'

structure_initialization ::= '(' structure_element_initialization
                               {','  structure_element_initialization} ')'
```

```
structure_element_initialization
                         ::= structure_element_name ':=' (constant
                               | enumerated_value
                               | array_initialization
                               | structure_initialization)

string_type_name         ::= identifier

string_type_declaration  ::= string_type_name ':' ('STRING'|'WSTRING')
                               ['[' integer ']'] [':=' character_string]
```

B.2.4 Variables

B.2.4.1 General

SEMANTICS: see 6.3.4.2.

PRODUCTION RULES:

```
variable           ::= direct_variable | symbolic_variable

symbolic_variable  ::= variable_name | multi_element_variable

variable_name      ::= identifier
```

B.2.4.2 Directly represented variables

SEMANTICS: see 6.4.2.2.

PRODUCTION RULES:

```
direct_variable    ::= '%' location_prefix size_prefix integer {'.' integer}

location_prefix    ::= 'I' | 'Q' | 'M'

size_prefix        ::= NIL | 'X' | 'B' | 'W' | 'D' | 'L'
```

B.2.4.3 Multi-element variables

SEMANTICS: see 6.4.2.3.

PRODUCTION RULES:

```
multi_element_variable   ::= array_variable | structured_variable

array_variable           ::= subscripted_variable subscript_list

subscripted_variable     ::= symbolic_variable

subscript_list           ::= '[' subscript {',' subscript} ']'

subscript                ::= expression

structured_variable      ::= record_variable '.' field_selector

record_variable          ::= symbolic_variable

field_selector           ::= identifier
```

B.2.4.4 Declaration and initialization

SEMANTICS: see 6.4.2 and 6.4.3. The non-terminal `function_block_type_name` is defined in B.2.5.2.

PRODUCTION RULES:

```
input_declarations ::= 'VAR_INPUT' ['RETAIN' | 'NON_RETAIN']
                         input_declaration ';'
                         {input_declaration ';'}
                         'END_VAR'

input_declaration  ::= var_init_decl | edge_declaration
                         | array_var_flex_decl
```

```
edge_declaration   ::= var1_list ':' 'BOOL' ('R_EDGE' | 'F_EDGE')
                         array_var_flex_decl
                   ::= var1_list ':' array_spec_flex

array_spec_flex    ::= 'ARRAY' '[' '*' {',' '*'} ']' 'OF'
                         non_generic_type_name

var_init_decl      ::= var1_init_decl | array_var_init_decl
                         | structured_var_init_decl
                         | fb_name_decl
                         | string_var_declaration

var1_init_decl     ::= var1_list ':' (simple_spec_init
                         | subrange_spec_init
                         | enumerated_spec_init)

var1_list          ::= variable_name {',' variable_name}

array_var_init_decl
                   ::= var1_list ':' array_spec_init

structured_var_init_decl
                   ::= var1_list ':' initialized_structure

fb_name_decl       ::= fb_name_list ':'
                         function_block_type_name
                         [ ':=' structure_initialization ]

fb_name_list       ::= fb_name {',' fb_name}

fb_name            ::= identifier

output_declarations
                   ::= 'VAR_OUTPUT' ['RETAIN' | 'NON_RETAIN']
                         output_declaration ';'
                         { output_declaration ';'}
                         'END_VAR'

output_declaration ::= var_init_decl | array_var_flex_decl

input_output_declarations

                   ::= 'VAR_IN_OUT'
                         var_declaration ';'
                         {var_declaration ';'}
                         'END_VAR'

var_declaration    ::=  temp_var_decl | fb_name_decl
                         | array_var_flex_decl

temp_var_decl      ::= var1_declaration | array_var_declaration
                         | structured_var_declaration
                         | string_var_declaration

var1_declaration   ::=  var1_list  ':' (simple_specification
                         | subrange_specification | enumerated_specification)

array_var_declaration
                   ::= var1_list ':' array_specification

structured_var_declaration
                   ::= var1_list ':' structure_type_name

var_declarations   ::= 'VAR' ['CONSTANT']
                         var_init_decl ';'
                         {(var_init_decl ';')}
                         'END_VAR'

retentive_var_declarations
                   ::= 'VAR' 'RETAIN'
                         var_init_decl ';' {var_init_decl ';'}
                         'END_VAR'

located_var_declarations
                   ::= 'VAR' ['CONSTANT' | 'RETAIN' | 'NON_RETAIN']
                          located_var_decl ';' {located_var_decl ';'}
                         'END_VAR'

located_var_decl   ::= [variable_name] location ':' located_var_spec_init
```

```
external_var_declarations
                   := 'VAR_EXTERNAL' ['CONSTANT']
                         external_declaration ';' {external_declaration ';'}
                         'END_VAR'

external_declaration
                   ::= global_var_name ':'
                         (simple_specification | subrange_specification
                         | enumerated_specification | array_specification
                         | structure_type_name | function_block_type_name)

global_var_name    ::= identifier

global_var_declarations
                   ::= 'VAR_GLOBAL' ['CONSTANT' | 'RETAIN']
                         global_var_decl ';' {global_var_decl ';'}
                         'END_VAR'

global_var_decl    ::= global_var_spec ':'
                         ( located_var_spec_init | function_block_type_name )

global_var_spec    ::= global_var_list | [global_var_name] location

located_var_spec_init
                   ::= simple_spec_init
                         | subrange_spec_init
                         | enumerated_spec_init
                         | array_spec_init
                         | initialized_structure
                         | single_byte_string_spec
                         | double_byte_string_spec

location           ::= 'AT' direct_variable

global_var_list    ::= global_var_name {',' global_var_name}

string_var_declaration
                   ::= single_byte_string_var_declaration
                         | double_byte_string_var_declaration

single_byte_string_var_declaration
                   ::= var1_list ':' single_byte_string_spec

single_byte_string_spec
                   ::= 'STRING' ['[' integer ']']
                         [':=' single_byte_character_string]

double_byte_string_var_declaration
                   ::= var1_list ':' double_byte_string_spec

double_byte_string_spec
                   ::= 'WSTRING' ['[' integer ']']
                         [':=' double_byte_character_string]

incompl_located_var_declarations
                   ::= 'VAR' ['RETAIN'|'NON_RETAIN']
                         incompl_located_var_decl ';
                         '{incompl_located_var_decl ';'}
                         'END_VAR'

incompl_located_var_decl
                   ::= variable_name incompl_location ':' var_spec

incompl_location   ::= 'AT' '%' ('I' | 'Q' | 'M') '*'

var_spec           ::= simple_specification
                         | subrange_specification
                         | enumerated_specification
                         | array_specification
                         | structure_type_name
                         | 'STRING' ['[' integer ']']
                         | 'WSTRING' ['[' integer ']']
```

B.2.5 Program organization units

B.2.5.1 Functions

SEMANTICS: see 6.5.2.

NOTE 1 This syntax does not reflect the fact that each function must have at least one input declaration.

NOTE 2 This syntax does not reflect the fact that edge declarations, function block references and calls are not allowed in function bodies.

NOTE 3 Ladder diagrams and function block diagrams are graphically represented. The non-terminals `instruction_list` and `statement_list` are defined in B.3.1 and B.4.2, respectively.

PRODUCTION RULES:

```
function_name            ::= standard_function_name | derived_function_name

standard_function_name   ::= <as defined in 6.5.2.6>

derived_function_name    ::= identifier

function_declaration     ::= 'FUNCTION' derived_function_name
                               [':' (elementary_type_name | derived_type_name
                               | 'VOID')]
                               { io_var_declarations
                               | function_var_decls }
                                 function_body
                               'END_FUNCTION'

io_var_declarations      ::= input_declarations | output_declarations
                               | input_output_declarations

function_var_decls       ::= external_var_declarations | var_declarations

function_body            ::= ladder_diagram
                               | function_block_diagram
                               | instruction_list
                               | statement_list
                               | <other languages>

var2_init_decl           ::= var1_init_decl
                               | array_var_init_decl
                               | structured_var_init_decl
                               | string_var_declaration
```

B.2.5.2 Function blocks

[Editor's note: Methods, ... TBD]

SEMANTICS: see 6.5.3. [and OO]

NOTE 1 Ladder diagrams and function block diagrams are graphically represented as defined in 8.

NOTE 2 The non-terminals `sequential_function_chart`, `instruction_list`, and `statement_list` are defined in B.2.6, B.3, and B.4.2, respectively.

PRODUCTION RULES:

```
function_block_type_name       ::= standard_function_block_name
                                     | derived_function_block_name

standard_function_block_name   ::= <as defined in 6.5.3.5>

derived_function_block_name    ::= identifier

function_block_declaration     ::= 'FUNCTION_BLOCK' derived_function_block_name
                                     [':' (elementary_type_name
                                     | derived_type_name | 'VOID')]
                                     { io_var_declarations
                                     | function_var_decls }
                                     { io_var_declarations
                                     | other_var_declarations }
                                     { function_block_body     }
                                     { method ´_declarations   }
                                     'END_FUNCTION_BLOCK'
```

```
other_var_declarations   ::= external_var_declarations | var_declarations
                               | retentive_var_declarations
                               | non_retentive_var_declarations
                               | temp_var_decls
                               | incompl_located_var_declarations

temp_var_decls           ::= 'VAR_TEMP'
                               temp_var_decl ';' {temp_var_decl ';'}
                               'END_VAR'

non_retentive_var_declarations
                         ::= 'VAR' 'NON_RETAIN'
                               var_init_decl ';' {var_init_decl ';'}
                               'END_VAR'

function_block_body      ::= sequential_function_chart
                         | ladder_diagram
                         | function_block_diagram
                         | instruction_list
                         | statement_list
                         | <other languages>
                         | empty body
```

B.2.5.3 Programs

SEMANTICS:

PRODUCTION RULES:

```
method_declarations      ::= 'METHOD'
                               declarations ....
                               function_body
                               'Method_END'

program_type_name        :: = identifier

program_declaration      ::= 'PROGRAM' program_type_name
                               { io_var_declarations | other_var_declarations
                               | located_var_declarations
                               | program_access_decls }
                                function_block_body
                               'END_PROGRAM'

program_access_decls     ::= 'VAR_ACCESS'
                               program_access_decl';' {program_access_decl';' }
                               'END_VAR'

program_access_decl      ::= access_name ':' symbolic_variable ':'
                               non_generic_type_name [direction]
```

B.2.6 Sequential function chart elements

SEMANTICS:. The use of function block diagram networks and ladder diagram rungs, denoted by the non-terminals `fbd_network` and `rung`, respectively, for the expression of transition conditions shall be as defined in 6.6.3.

NOTE 1 The non-terminals `simple_instruction_list` and `expression` are defined in B.3.1 and B.4.1, respectively.

NOTE 2 The term [`transition_name`] can only be used in the production for `transition` when feature #7 in Table 49 is supported. The resulting production is the textual equivalent of this feature.

PRODUCTION RULES:

```
sequential_function_chart
                   ::= sfc_network {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' | timed_qualifier ',' action_time

timed_qualifier    ::= 'L' | 'D' | 'SD' | 'DS' | 'SL'

action_time        ::= duration | variable_name

indicator_name     ::= variable_name

transition         ::= ‘TRANSITION’ [transition_name]
                         ['(' 'PRIORITY' ':=' integer ')']
                         'FROM' steps 'TO' steps transition_condition
                         'END_TRANSITION'

transition_name    ::= identifier

steps              ::= step_name
                         | '(' step_name ',' step_name {',' step_name} ')'

transition_condition
                   ::= ':' simple_instruction_list | ':=' expression ';'
                         | ':' (fbd_network | rung)

action             ::= 'ACTION' action_name ':'function_block_body 'END_ACTION'
```

B.2.7 Configuration elements

SEMANTICS: see 6.7.

NOTE This syntax does not reflect the fact that location assignments are only allowed for references to variables which are marked by the asterisk notation at type declaration level.

PRODUCTION RULES:

```
configuration_name ::= identifier

resource_type_name ::= identifier

configuration_declaration ::=
   'CONFIGURATION' configuration_name
     [global_var_declarations]
       (single_resource_declaration
        | (resource_declaration {resource_declaration}))
     [access_declarations]
     [instance_specific_initializations]
   'END_CONFIGURATION'

resource_declaration ::=
   'RESOURCE' resource_name 'ON' resource_type_name
      [global_var_declarations]
      single_resource_declaration
    'END_RESOURCE'

single_resource_declaration ::=
   {task_configuration ';'}
   program_configuration ';'
   {program_configuration ';'}

resource_name ::= identifier

access_declarations ::=
   'VAR_ACCESS'
    access_declaration ';'
    {access_declaration ';'}
   'END_VAR'

access_declaration ::= access_name ':' access_path ':' non_generic_type_name
   [direction]

access_path ::= [resource_name '.'] direct_variable
   | [resource_name '.'] [program_name '.']
       {fb_name'.'} symbolic_variable
```

```
global_var_reference ::=
   [resource_name '.'] global_var_name ['.' structure_element_name]
access_name        ::= identifier
program_output_reference ::= program_name '.' symbolic_variable
program_name       ::= identifier
direction          ::= 'READ_WRITE' | 'READ_ONLY'
task_configuration ::= 'TASK' task_name task_initialization
task_name          := identifier
task_initialization ::=
   '(' ['SINGLE' ':=' data_source ',']
       ['INTERVAL' ':=' data_source ',']
       'PRIORITY' ':=' integer ')'
data_source ::= constant_expression | global_var_reference
   | program_output_reference | direct_variable
program_configuration ::=
   'PROGRAM' [RETAIN | NON_RETAIN]
     program_name ['WITH' task_name] ':' program_type_name
     ['(' prog_conf_elements ')']
prog_conf_elements ::= prog_conf_element {',' prog_conf_element}
prog_conf_element ::= fb_task | prog_cnxn
fb_task      ::= fb_name 'WITH' task_name
prog_cnxn    ::= symbolic_variable ':=' prog_data_source
   | symbolic_variable '=>' data_sink
prog_data_source ::=
   constant_expression | enumerated_value | global_var_reference
   | direct_variable
data_sink    ::= global_var_reference | direct_variable
instance_specific_initializations ::=
   'VAR_CONFIG'
     instance_specific_init ';'
     {instance_specific_init ';'}
   'END_VAR'
instance_specific_init ::=
   resource_name '.' program_name '.' {fb_name '.'}
   ((variable_name [location] ':' located_var_spec_init) |
    (fb_name ':' function_block_type_name ':=' structure_initialization))
```

B.3 Language IL (Instruction List)

B.3.1 Instructions and operands

PRODUCTION RULES:

```
instruction_list ::= il_instruction {il_instruction}
il_instruction ::= [label':']
   [  il_simple_operation
    | il_expression
    | il_jump_operation
    | il_fb_call
    | il_formal_funct_call
    | il_return_operator ] EOL {EOL}
label ::= identifier
il_simple_operation ::= ( il_simple_operator [il_operand] )
   | ( function_name [il_operand_list] )
il_expression ::= il_expr_operator '(' [il_operand] EOL {EOL}
   [simple_instr_list] ')'
il_jump_operation ::= il_jump_operator label
```

```
il_fb_call ::= il_call_operator fb_name ['('
   (EOL {EOL} [ il_param_list ]) | [ il_operand_list ] ')']

il_formal_funct_call ::= function_name '(' EOL {EOL} [il_param_list] ')'

il_operand ::= constant_expression | variable | enumerated_value

il_operand_list ::= il_operand {',' il_operand}

simple_instr_list ::= il_simple_instruction {il_simple_instruction}

il_simple_instruction ::=
   (il_simple_operation | il_expression | il_formal_funct_call) EOL {EOL}

il_param_list ::= {il_param_instruction} il_param_last_instruction

il_param_instruction ::= (il_param_assignment | il_param_out_assignment) ','
   EOL {EOL}

il_param_last_instruction ::=
   ( il_param_assignment | il_param_out_assignment ) EOL {EOL}

il_param_assignment ::= il_assign_operator ( il_operand | ( '(' EOL {EOL} sim-
   ple_instr_list ')' ) )

il_param_out_assignment ::= il_assign_out_operator variable
```

B.3.2 Operators

SEMANTICS: see 7.2. This syntax does not reflect the possibility for typing IL operators as noted in Table 60.

PRODUCTION RULES:

```
il_simple_operator ::= 'LD' | 'LDN' | 'ST' | 'STN' | '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_assign_operator ::= variable_name':='

il_assign_out_operator ::= ['NOT'] variable_name'=>'

il_call_operator ::= 'CAL' | 'CALC' | 'CALCN'

il_return_operator ::= 'RET' | 'RETC' | 'RETCN'

il_jump_operator ::= 'JMP' | 'JMPC' | 'JMPCN'
```

B.4 Language ST (Structured Text)

B.4.1 Expressions

SEMANTICS: these definitions have been arranged to show a top-down derivation of expression structure. The precedence of operations is then implied by a “bottom-up” reading of the definitions of the various kinds of expressions. Further discussion of the semantics of these definitions is given in 7.3.2. See 6.5.2.2 for details of the semantics of function calls.

PRODUCTION RULES:

```
expression          ::= xor_expression {'OR' xor_expression}

xor_expression      ::= and_expression {'XOR' and_expression}

and_expression      ::= comparison {('&' | 'AND') comparison}

comparison          ::= equ_expression { ('=' | '<>') equ_expression}

equ_expression      ::= add_expression {comparison_operator add_expression}

comparison_operator
                    ::= '<' | '>' | '<=' | '>='

add_expression      ::= term {add_operator term}

add_operator        ::= '+' | '-'
```

```
term                 ::= power_expression  {multiply_operator power_expression}
multiply_operator    ::= '*' | '/' | 'MOD'
power_expression     ::= unary_expression {'**' unary_expression}
unary_expression     ::= [unary_operator] primary_expression
unary_operator       ::= '-' | 'NOT'
primary_expression
                     ::= constant | enumerated_value | variable
                     | '(' expression ')'
                     | funtion_name
                           '('[ param_assignment {',' param_assignment} '])'
primary_constant_expression
                     ::= constant | enumerated_value | variable
                                 | '(' expression ')'
```

B.4.2 Statements

B.4.2.1 General

SEMANTICS: see 6.7.3.

PRODUCTION RULE:

```
statement_list       ::= statement ';' {statement ';'}
statement            ::= NIL | assignment_statement
                           | subprogram_control_statement | selection_statement
                           | iteration_statement
```

B.4.2.2 Assignment statements

SEMANTICS: see 7.3.2.2.

PRODUCTION RULE:

```
assignment_statement ::= variable ':=' expression
```

B.4.2.3 Subprogram control statements

SEMANTICS: see 7.3.2.3.

PRODUCTION RULES:

```
subprogram_control_statement
                     ::= fb_invocation | 'RETURN'
fb_invocation        ::= fb_name '(' [param_assignment
                                 {',' param_assignment}] ')'
param_assignment     ::= ([variable_name ':='] expression)
                           | (['NOT'] variable_name '=>' variable)
```

B.4.2.4 Selection statements

SEMANTICS: see 7.3.2.4.

PRODUCTION RULES:

```
selection_statement ::= if_statement | case_statement
if_statement ::=
   'IF' expression 'THEN' statement_list
     {'ELSIF' expression 'THEN' statement_list}
     ['ELSE' statement_list]
   'END_IF'
```

```
case_statement     ::= 'CASE' expression 'OF'
                         case_element
                          {case_element}
                          ['ELSE' statement_list]
                         'END_CASE'

case_element       ::= case_list ':' statement_list

case_list          ::= case_list_element {',' case_list_element}

case_list_element  ::= subrange | signed_integer | enumerated_value
                         | identifier
```

B.4.2.5 Iteration statements

SEMANTICS: see 7.3.2.5.

PRODUCTION RULES:

```
iteration_statement      ::= for_statement | while_statement | repeat_statement
                               | exit_statement | continue_statement

for_statement            ::= 'FOR' control_variable ':=' for_list
                               'DO' statement_list 'END_FOR'

control_variable         ::= identifier

for_list                 ::= expression 'TO' expression ['BY' expression]

while_statement          ::= 'WHILE' expression 'DO' statement_list 'END_WHILE'

repeat_statement         ::= 'REPEAT' statement_list 'UNTIL' expression
   'END_REPEAT'

exit_statement           ::= 'EXIT'

continue_statement       ::= 'CONTINUE'
```

Annex C
(normative)
Delimiters and keywords

The usages of delimiters and keywords in this standard is summarized in tables C.1 and C.2. National standards organizations can publish tables of translations for the textual portions of the delimiters listed in table C.1 and the keywords listed in table C.2.

[Editor’s note: Subclause numbers and links not yet complete.]

Table C.1 - Delimiters

Delimiters	Subclause	Usage
Space	6.1.4	As specified in 6.1.4.
(*	6.1.5	Begin comment
*)		End comment
//		Single line comment
+	6.2.1	Leading sign of decimal literal
	7.3.2	Addition operator
-	6.2.1	Leading sign of decimal literal
	6.2.3.2	Year-month-day separator
	7.3.1	Subtraction, negation operator
	8.1.2	Horizontal line
#	6.2.1	Based number separator
	6.2.3	Time literal separator
.	6.2.1	Integer/fraction separator
	6.4.2.2	Hierarchical address separator
	6.4.2.3	Structure element separator
	6.5.3.2	Function block structure separator
e or E	6.2.1	Real exponent delimiter
'	6.2.2	Start and end of character string
$	6.2.2	Start of special character in strings
T#, D, H, M, S, MS, DATE#, D#,TOD#,DT#, TIME_OF_DAY#, DATE_AND_TIME#	6.2.3	Time literal delimiters
:	6.2.3.2	Time of day separator
	6.2.3.1	Type name/specification separator
	6.4.3	Variable/type separator
	6.6.2	Step name terminator
	6.7.2	RESOURCE name/type separator
	6.7.2	PROGRAM name/type separator
	6.7.2	Access name/path/type separator
	7.2.1	Instruction label terminator
	8.2.1	Network label terminator
:=	**6.3.3**	**Initialization operator**
	6.7.2	**Input connection operator**
	7.3.2.2	**Assignment operator**
()	6.3.4.2	Enumeration list delimiters

()	6.3.3.1	Subrange delimiters
[]	6.4.1.2	Array subscript delimiters
[]	6.4.3.1	String length delimiters
()	6.4.3.2	Multiple initialization
()	7.2.2	Instruction List modifier/operator
()	7.3.1	Function arguments
()	7.3.1	Subexpression hierarchy
()	7.3.2.3	Function block input list delimiters
,	6.3.3.1	Enumeration list separator
	6.3.3.2	Initial value separator
	6.4.1	Array subscript separator
	6.4.2	Declared variable separator
	6.5.3.2	Function block initial value separator
	6.5.3.2	Function block input list separator
	7.3.2.3	Operand list separator
	7.3.2.3	Function argument list separator
	3.3.2.3	CASE value list separator
;	2.3.3.1	Type declaration separator
	3.3	Statement separator
..	2.3.3.1	Subrange separator
	7.3.2.4	CASE range separator
%	2.4.1.1	Direct representation prefix
=>	6.7.2	Output connection operator
**, NOT, *, /, MOD, +, -, <, >, <= >=, =, <>, &, AND, XOR, OR	7.3.1	Infix operators
| or !	8.1.2	Vertical lines

4114

Table C.2 - Keywords

Keywords	Subclause
ACTION...END_ACTION	6.6.4.2
ARRAY...OF	6.3.3.1
AT	6.4.3
CASE...OF...ELSE...END_CASE	7.3.2.4
CONFIGURATION...END_CONFIGURATION	6.7.2
CONSTANT	6.4.3
Data type names	6.3
EN, ENO	6.5.2.3, 6.5.3.3a
EXIT	7.3.2.5
FALSE	6.2.1
F_EDGE	6.5.3.4
FOR...TO...BY...DO...END_FOR	7.3.2.5
FUNCTION...END_FUNCTION	6.5.2.4
Function names	2.5.1
FUNCTION_BLOCK...END_FUNCTION_BLOCK	2.5.2.2
Function Block names	6.5.2
METHOD...END_METHOD	
THIS, SUPER	6.5.4.4.5
INTERFACE...END_INTERFACE, IMPLEMENTS, EXTENDS	xxx
IF...THEN...ELSIF...ELSE...END_IF	7.3.2.4
INITIAL_STEP...END_STEP	6.6.2
NOT, MOD, AND, XOR, OR	7.3.1
PROGRAM...WITH...	6.7.2
PROGRAM...END_PROGRAM	6.5.4
R_EDGE	6.5.3.4
READ_ONLY, READ_WRITE	6.7.2
REPEAT...UNTIL...END_REPEAT	7.3.2.5
RESOURCE...ON...END_RESOURCE	6.7.2
RETAIN, NON_RETAIN	6.4.3
RETURN	7.3.2.3
STEP...END_STEP	6.6.2
STRUCT...END_STRUCT	6.3.3.1
TASK	6.7.3
TRANSITION...FROM...TO...END_TRANSITION	6.6.3
TRUE	6.2.1
TYPE...END_TYPE	6.3.3.1
VAR...END_VAR	6.4.3
VAR_INPUT...END_VAR	6.4.3
VAR_OUTPUT...END_VAR	6.4.3
VAR_IN_OUT...END_VAR	6.4.3

VAR_TEMP...END_VAR	6.4.3
VAR_EXTERNAL...END_VAR	2.4.3
VAR_ACCESS...END_VAR	6.7.2
VAR_CONFIG...END_VAR	6.7.2
VAR_GLOBAL...END_VAR	6.7.2
WHILE...DO...END_WHILE	7.3.2.5
WITH	6.5.3.4

Annex D
(normative)
Implementation dependencies

The **implementation dependencies** defined in this standard, and the primary reference clause for each, are listed in Table D.1.

NOTE Other **implementation dependencies** such as the accuracy, precision and repeatability of timing and execution control features may have significant effects on the portability of programs but are beyond the scope of this part of IEC 61131.

Editor's note: Subclause numbers and links not yet complete.]

Table D.1 - Implementation dependencies

Subclause	Parameters
6.1.2	Maximum length of identifiers
6.1.6	Syntax and semantics of pragmas
6.2.2	Syntax and semantics for the use of the double-quote character when a particular implementation supports feature #4 but not feature #2 of Table 6.
6.3.1	Range of values and precision of representation for variables of type `TIME`, `DATE`, `TIME_OF_DAY` and `DATE_AND_TIME` Precision of representation of seconds in types `TIME`, `TIME_OF_DAY` and `DATE_AND_TIME`
6.3.3.1	Maximum number of enumerated values Maximum number of array subscripts Maximum array size Maximum number of structure elements Maximum structure size Maximum range of subscript values Maximum number of levels of nested structures
6.3.3.2	Default maximum length of `STRING` and `WSTRING` variables Maximum allowed length of `STRING` and `WSTRING` variables
6.4.1.1	Maximum number of hierarchical levels Logical or physical mapping
6.4.2	Initialization of system inputs
6.4.3	Maximum number of variables per declaration Effect of using `AT` qualifier in declaration of function block instances Warm start behavior if variable is declared as neither `RETAIN` nor `NON_RETAIN`
6.5	Information to determine execution times of program organization units
6.5.2.3	Values of outputs when `ENO` is `FALSE`
2.5.2.4	Maximum number of function specifications
2.5.1.5	Maximum number of inputs of extensible functions
6.5.2.6.2	Effects of type conversions on accuracy Error conditions during type conversions
6.5.2.6.3	Accuracy of numerical functions
2.5.2.6.7	Effects of type conversions between time data types and other data types not defined in table 34
6.5.3	Maximum number of function block specifications and instantiations
6.5.3.3	Function block input variable assignment when `EN` is `FALSE`
6.5.3.5.4	`Pvmin`, `Pvmax` of counters
6.5.3.5.5	Effect of a change in the value of a `PT` input during a timing operation
6.5.4	Program size limitations
6.6.2	Precision of step elapsed time Maximum number of steps per SFC
6.6.3	Maximum number of transitions per SFC and per step

6.6.4.3	Maximum number of action blocks per step
6.6.4.6	Access to the functional equivalent of the Q or A outputs
6.6.5	Transition clearing time Maximum width of diverge/converge constructs
6.7.2	Contents of RESOURCE libraries
6.7.2	Effect of using READ_WRITE access to function block outputs
6.7.3	Maximum number of tasks Task interval resolution
7.3.1	Maximum length of expressions
7.3.2	Maximum length of statements
7.3.2.4	Maximum number of CASE selections
7.3.2.5	Value of control variable upon termination of FOR loop
8.1.1	Restrictions on network topology
8.1.4	Evaluation order of feedback loops

Annex E
(normative)
Error conditions

The error conditions defined in this standard, and the primary reference clause for each, are listed in table E.1. These errors may be detected during preparation of the program for execution or during execution of the program. The manufacturer shall specify the disposition of these errors according to the provisions of 5.1.

[Editor's note: Subclause numbers and links not yet complete.]

Table E.1 - Error conditions

Subclause	Error conditions
6.3.3.1	Ambiguous enumerated value
6.3.3.1	Value of a variable exceeds the specified subrange
6.4.1.1	Missing configuration of an incomplete address specification ("*" notation)
6.4.2.3	Invalid subscript value
6.4.2.4	Invalid modification of an ARRAY element
6.4.3	Attempt by a program organization unit to modify a variable which has been declared CONSTANT or VAR_INPUT
xxx	Usage of a non CONSTANT variable in a constant expression
6.4.3	Attempt by a program organization unit to modify a variable which has been declared CONSTANT
6.4.3	Declaration of a variable as VAR_GLOBAL CONSTANT in a containing element having a contained element in which the same variable is declared VAR_EXTERNAL without the CONSTANT qualifier.
6.5.2	Improper use of directly represented or external variables in functions
6.5.2.4	A VAR_IN_OUT variable is not "properly mapped"
6.5.2.4	Ambiguous value caused by a VAR_IN_OUT connection
6.5.2.6.2	Type conversion errors
6.5.2.6.3	Numerical result exceeds range for data type Division by zero
6.5.2.6.4	N input is less than zero in a bit-shift function
6.5.2.6.5	Mixed input data types to a selection function Selector (K) out of range for MUX function
6.5.2.6.6	Invalid character position specified Result exceeds maximum string length ANY_INT input is less than zero in a string function
6.5.2.6.7	Result exceeds range for data type
6.5.3.4	No value specified for a function block instance used as input variable
6.5.3.4	No value specified for an in-out variable
6.6.2	Zero or more than one initial steps in SFC network User program attempts to modify step state or time
6.6.3	Side effects in evaluation of transition condition
6.6.4.2	Modification of a Boolean action from outside its SFC
6.6.4.6	Action control contention error
6.6.5	Simultaneously true, non-prioritized transitions in a selection divergence Unsafe or unreachable SFC
6.7.2	Data type conflict in VAR_ACCESS
6.7.3	A task fails to be scheduled or to meet its execution deadline

7.2.2	Numerical result exceeds range for data type Current result and operand not of same data type
7.3.1	Division by zero Numerical result exceeds range for data type Invalid data type for operation
7.3.2.2	Return from function without value assigned
7.3.2.5	Iteration fails to terminate
8.1.1	Same identifier used as connector label and element name
8.1.4	Uninitialized feedback variable

Annex F
(informative)
Reference character set

NOTE 1 The contents of the most recent edition of “table 1Row 00: ISO-646 IRV” of ISO/IEC 10646-1 are normative for the purposes of this standard. The reference character set is reproduced here for information only.

NOTE 2 In variables of type STRING, the individual byte encodings of the characters in this reference character set are as given in table H.2. In variables of type WSTRING, the numerical equivalent of individual 16-bit word encodings are also as given in table H.2.

Table G.1 - Character representations

	First hexadecimal digit					
Second hexadecimal digit	**2**	**3**	**4**	**5**	**6**	**7**
0		0	@	P	`	p
1	!	1	A	Q	a	q
2	"	2	B	R	b	r
3	#	3	C	S	c	s
4	$	4	D	T	d	t
5	%	5	E	U	e	u
6	&	6	F	V	f	v
7	'	7	G	W	g	w
8	(	8	H	X	h	x
9	)	9	I	Y	i	y
A	*	:	J	Z	j	z
B	+	;	K	[	k	{
C	,	<	L	\	l	|
D	-	=	M	]	m	}
E	.	>	N	^	n	~
F	/	?	O	_	o	

Table G.2 - Character encodings

dec	hex	Name	dec	hex	Name
032	20	SPACE	080	50	LATIN CAPITAL LETTER P
033	21	EXCLAMATION MARK	081	51	LATIN CAPITAL LETTER Q
034	22	QUOTATION MARK	082	52	LATIN CAPITAL LETTER R
035	23	NUMBER SIGN	083	53	LATIN CAPITAL LETTER S
036	24	DOLLAR SIGN	084	54	LATIN CAPITAL LETTER T
037	25	PERCENT SIGN	085	55	LATIN CAPITAL LETTER U
038	26	AMPERSAND	086	56	LATIN CAPITAL LETTER V
039	27	APOSTROPHE	087	57	LATIN CAPITAL LETTER W
040	28	LEFT PARENTHESIS	088	58	LATIN CAPITAL LETTER X
041	29	RIGHT PARENTHESIS	089	59	LATIN CAPITAL LETTER Y
042	2A	ASTERISK	090	5A	LATIN CAPITAL LETTER Z
043	2B	PLUS SIGN	091	5B	LEFT SQUARE BRACKET
044	2C	COMMA	092	5C	REVERSE SOLIDUS
045	2D	HYPHEN-MINUS	093	5D	RIGHT SQUARE BRACKET
046	2E	FULL STOP	094	5E	CIRCUMFLEX ACCENT
047	2F	SOLIDUS	095	5F	LOW LINE
048	30	DIGIT ZERO	096	60	GRAVE ACCENT
049	31	DIGIT ONE	097	61	LATIN SMALL LETTER A
050	32	DIGIT TWO	098	62	LATIN SMALL LETTER B
051	33	DIGIT THREE	099	63	LATIN SMALL LETTER C
052	34	DIGIT FOUR	100	64	LATIN SMALL LETTER D
053	35	DIGIT FIVE	101	65	LATIN SMALL LETTER E
054	36	DIGIT SIX	102	66	LATIN SMALL LETTER F
055	37	DIGIT SEVEN	103	67	LATIN SMALL LETTER G
056	38	DIGIT EIGHT	104	68	LATIN SMALL LETTER H
057	39	DIGIT NINE	105	69	LATIN SMALL LETTER I
058	3A	COLON	106	6A	LATIN SMALL LETTER J
059	3B	SEMICOLON	107	6B	LATIN SMALL LETTER K
060	3C	LESS-THAN SIGN	108	6C	LATIN SMALL LETTER L
061	3D	EQUALS SIGN	109	6D	LATIN SMALL LETTER M
062	3E	GREATER-THAN SIGN	110	6E	LATIN SMALL LETTER N
063	3F	QUESTION MARK	111	6F	LATIN SMALL LETTER O
064	40	COMMERCIAL AT	112	70	LATIN SMALL LETTER P
065	41	LATIN CAPITAL LETTER A	113	71	LATIN SMALL LETTER Q
066	42	LATIN CAPITAL LETTER B	114	72	LATIN SMALL LETTER R
067	43	LATIN CAPITAL LETTER C	115	73	LATIN SMALL LETTER S
068	44	LATIN CAPITAL LETTER D	116	74	LATIN SMALL LETTER T
069	45	LATIN CAPITAL LETTER E	117	75	LATIN SMALL LETTER U
070	46	LATIN CAPITAL LETTER F	118	76	LATIN SMALL LETTER V
071	47	LATIN CAPITAL LETTER G	119	77	LATIN SMALL LETTER W
072	48	LATIN CAPITAL LETTER H	120	78	LATIN SMALL LETTER X
073	49	LATIN CAPITAL LETTER I	121	79	LATIN SMALL LETTER Y
074	4A	LATIN CAPITAL LETTER J	122	7A	LATIN SMALL LETTER Z
075	4B	LATIN CAPITAL LETTER K	123	7B	LEFT CURLY BRACKET
076	4C	LATIN CAPITAL LETTER L	124	7C	VERTICAL LINE
077	4D	LATIN CAPITAL LETTER M	125	7D	RIGHT CURLY BRACKET
078	4E	LATIN CAPITAL LETTER N	126	7E	TILDE
079	4F	LATIN CAPITAL LETTER O			

END