

Programmable controllers

Part 3. Programming languages

The European Standard EN 61131-3 : 1993 has the status of a
British Standard

Automates programmables
Partie 3. Langues de programmation

Speicherprogrammierbare Steuerungen
Teil 3. Programmiersprachen

Cooperating organizations

The European Committee for Electrotechnical Standardization (CENELEC), under whose supervision this European Standard was prepared, comprises the national committees of the following countries:

Austria	Italy
Belgium	Luxembourg
Denmark	Netherlands
Finland	Norway
France	Portugal
Germany	Spain
Greece	Sweden
Iceland	Switzerland
Ireland	United Kingdom

This British Standard, having been prepared under the direction of the Industrial-process Measurement and Control Standards Policy Committee, was published under the authority of the Standards Board and comes into effect on 15 November 1993

© BSI 1993

The following BSI references relate to the work on this standard:
Committee reference PCL/1
Draft for comment 87/22187 DC

ISBN 0 580 22564 X

Amendments issued since publication

Amd. No.	Date	Text affected

Contents

	Page
Cooperating organizations	Inside front cover
National foreword	i
Foreword	2
Text of EN 61131-3	3
National annex NA (informative) Committees responsible	Inside back cover
National annex NB (informative) Cross-references	Inside back cover

National foreword

This British Standard has been prepared under the direction of the Industrial-process Measurement and Control Standards Policy Committee and is the English language version of EN 61131-3 : 1993, *Programmable controllers — Part 3 : Programming Languages*, published by the European Committee for Electrotechnical Standardization (CENELEC).

It is identical with IEC 1131-3 : 1993 published by the International Electrotechnical Commission (IEC). In adopting IEC 1131-3 : 1993 as EN 61131-3 : 1993 CENELEC added normative annex ZA.

No British Standard is superseded.

BS EN 61131 consists of the following Parts under the general title *Programmable controllers*.

Part 1. *General information*¹⁾

Part 2. *Equipment requirements and tests*¹⁾

Part 3 : 1993 *Programming languages*

Part 4 : 1993 *User guidelines*¹⁾

It is envisaged that there will be a further Part on messaging service specification.

Annexes A, B, C, D and E form an integral part of this Part of BS EN 61131. Annexes F, G and H are for information only.

An IEC type 2 technical report (TR) will provide 'pre-standardization' guidance for the implementation and application of the programming language defined in this Part of BS EN 61131, including such issues as operating system/program interaction and requirements for programming support environments.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

¹⁾ In preparation.

UDC 681.5 : 62-55 : 681.3.04

Descriptors: Industrial-process, process control, programmable controller, programming language

English version

Programmable controllers
Part 3 : Programming languages
(IEC 1131-3 : 1993)

Automates programmables
Partie 3 : Langages de programmation
(CEI 1131-3 : 1993)

Speicherprogrammierbare Steuerungen
Teil 3 : Programmiersprachen
(IEC 1131-3 : 1993)

This European Standard was approved by CENELEC on 1993-07-06. CENELEC members are bound to comply with the CEN/CENELEC Internal Regulations which stipulate the conditions for giving this European Standard the status of a national standard without any alteration.

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

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

CENELEC members are the national electrotechnical committees of Austria, Belgium, Denmark, Finland, France, Germany, Greece, Iceland, Ireland, Italy, Luxembourg, Netherlands, Norway, Portugal, Spain, Sweden, Switzerland and United Kingdom.

CENELEC

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

Central Secretariat: rue de Stassart 35, B-1050 Brussels

© 1993 Copyright reserved to CENELEC members

Ref. No. EN 61131-3 : 1993 E

Foreword

The text of document 65B(CO)85, as prepared by sub-committee 65B : Devices, of IEC technical committee 65 : Industrial-process measurement and control, was submitted to the IEC-CENELEC parallel vote in February 1992.

The reference document was approved by CENELEC as EN 61131-3 on 6 July 1993.

The following dates were fixed:

- latest date of publication
of an identical national
standard (dop) 1994-07-01
- latest date of withdrawal
of conflicting national
standards (dow) 1994-07-01

Annexes designated 'normative' are part of the body of the standard. Annexes designated 'informative' are given only for information. In this standard, annexes A, B, C, D, E and ZA are normative and annexes F, G and H are informative.

CONTENTS

Clause	Page
1 General	8
1.1 Scope	8
1.2 Normative references	8
1.3 Definitions	9
1.4 Overview and general requirements	14
1.4.1 Software model	14
1.4.2 Communication model	16
1.4.3 Programming model	18
1.5 Compliance	20
1.5.1 Programmable controller systems	21
1.5.2 Programs	22
2 Common elements	23
2.1 Use of printed characters	23
2.1.1 Character set	23
2.1.2 Identifiers	24
2.1.3 Keywords	25
2.1.4 Use of spaces	25
2.1.5 Comments	25
2.2 External representation of data	25
2.2.1 Numeric literals	25
2.2.2 Character string literals	26
2.2.3 Time literals	27
2.3 Data types	29
2.3.1 Elementary data types	29
2.3.2 Generic data types	30
2.3.3 Derived data types	31
2.4 Variables	35
2.4.1 Representation	35
2.4.2 Initialization	37
2.4.3 Declaration	38
2.5 Program organization units	43
2.5.1 Functions	43
2.5.2 Function blocks	60
2.5.3 Programs	75

Clause	Page
2.6 Sequential Function Chart (SFC) elements	76
2.6.1 General	76
2.6.2 Steps	76
2.6.3 Transition	78
2.6.4 Actions	82
2.6.5 Rules of evolution	91
2.6.6 Compatibility of SFC elements	100
2.6.7 Compliance requirements	100
2.7 Configuration elements	101
2.7.1 Configurations, resources, and access paths	102
2.7.2 Tasks	104
3 Textual languages	114
3.1 Common elements	114
3.2 Language IL (Instruction List)	114
3.2.1 Instructions	114
3.2.2 Operators, modifiers and operands	115
3.2.3 Functions and function blocks	116
3.3 Language ST (Structured Text)	117
3.3.1 Expressions	117
3.3.2 Statements	120
4 Graphic languages	123
4.1 Common elements	123
4.1.1 Representation of lines and blocks	123
4.1.2 Direction of flow in networks	124
4.1.3 Evaluation of networks	125
4.1.4 Execution control elements	127
4.2 Language LD (Ladder Diagram)	129
4.2.1 Power rails	129
4.2.2 Link elements and states	129
4.2.3 Contacts	130
4.2.4 Coils	130
4.2.5 Functions and function blocks	130
4.2.6 Order of network evaluation	130
4.3 Language FBD (Function Block Diagram)	133
4.3.1 General	133
4.3.2 Combination of elements	133
4.3.3 Order of network evaluation	133
Annexes	
A Specification method for textual languages	134
A.1 Syntax	134
A.2 Semantics	136

Annexes	Page
B Formal specifications of language elements	137
B.0 Programming model	137
B.1 Common elements	137
B.2 Language IL (Instruction List)	147
B.3 Language ST (Structured Text)	148
C Delimiters and keywords	150
D Implementation-dependent parameters	153
E Error conditions	155
F Examples	156
F.1 Function WEIGH	156
F.2 Function block CMD_MONITOR	157
F.3 Function block FWD_REV_MON	160
F.4 Function block STACK_INT	165
F.5 Function block MIX_2_BRIX	170
F.6 Analog signal processing	173
F.7 Program GRAVEL	182
F.8 Program AGV	190
G Index	194
H Software compliance testing	197
ZA Other international publications quoted in this standard with the references of the relevant European publications.....	198
Tables	
1 Character set features	24
2 Identifier features	24
3 Comment feature	25
4 Numeric literals	26
5 Character string literal feature	27
6 Two-character combinations in character strings	27
7 Duration literal features	28
8 Date and time of day literals	28
9 Examples of date and time of day literals	29
10 Elementary data types	30
11 Hierarchy of generic data types	31
12 Data type declaration features	33
13 Default initial values	33
14 Data type initial value declaration features	34
15 Location and size prefix features for directly represented variables	36
16 Variable declaration keywords	39
17 Variable type assignment features	40
18 Variable initial value assignment features	42
19 Graphical negation of Boolean signals	44
20 Use of EN input and ENO output	46
21 Typed and overloaded functions	48
22 Type conversion function features	51
23 Standard functions of one numeric variable	52
24 Standard arithmetic functions	53
25 Standard bit shift functions	54

Tables	Page
26 Standard bitwise Boolean functions	55
27 Standard selection functions	56
28 Standard comparison functions	57
29 Standard character string functions	58
30 Functions of time data types	59
31 Functions of enumerated data type	59
32 Examples of function block I/O parameter usage	61
33 Function block declaration features	64
34 Standard bistable function blocks	70
35 Standard edge detection function blocks	71
36 Standard counter function blocks	72
37 Standard timer function blocks	73
38 Standard timer function blocks – timing diagrams	74
39 Program declaration features	75
40 Step features	78
41 Transitions and transition conditions	80
42 Declaration of actions	83
43 Step/action association	85
44 Action block features	86
45 Action qualifiers	87
46 Sequence evolution	93
47 Compatible SFC features	100
48 SFC minimal compliance requirements	100
49 Configuration and resource declaration features	103
50 Task features	107
51 Examples of instruction fields	114
52 Instruction List (IL) operators	116
53 Function block invocation features for IL language	117
54 Standard function block input operators for IL language	117
55 Operators of the ST language	119
56 ST language statements	120
57 Representation of lines and blocks	125
58 Graphic execution control elements	128
59 Power rails	129
60 Link elements	130
61 Contacts	131
62 Coils	132

Figures

1 Software model	15
2 Communication model	17
3 Combination of programmable controller language elements	20
4 Examples of function usage	43
5 Use of formal parameter names	45
6 Examples of function declarations	47
7 Examples of explicit type conversion with overloaded functions	49
8 Examples of explicit type conversion with typed functions	50
9 Function block instantiation example	61
10 Examples of function block declarations	63
11 Graphical use of function block names as variables	65

Figures	Page
12 Examples of use of input/output variables	68
13 Semaphore usage example	69
14 ACTION_CONTROL function block – External interface	88
15 ACTION_CONTROL function block body	89
16 Action control example	90
17 SFC evolution rules	97
18 SFC errors	98
19 Configuration example	101
20 Examples of CONFIGURATION and RESSOURCE declaration features	104
21 Synchronization of function blocks	111
22 EXIT statement example	122
23 Feedback path example	127
24 Boolean OR examples	133

PROGRAMMABLE CONTROLLERS –

Part 3: Programming languages

1 General

1.1 Scope

This part of IEC 1131 applies to the printed and displayed representation, using characters of the ISO/IEC 646 character set, of the programming languages to be used for Programmable Controllers as defined in Part 1 of IEC 1131. Graphic and semigraphic representation of the language elements which are defined in this part is allowed, but is not defined in this part.

The functions of program entry, testing, monitoring, operating system, etc., are specified in Part 1 of IEC 1131.

1.2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of IEC 1131. At the time of publication, the editions indicated were valid. All normative documents are subject to revision, and parties to agreements based on this part of IEC 1131 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

IEC 50: *International Electrotechnical Vocabulary (IEV)*

IEC 559: 1989, *Binary floating-point arithmetic for microprocessor systems*

IEC 617-12: 1991, *Graphical symbols for diagrams – Part 12: Binary logic elements*

IEC 617-13: 1978, *Graphical symbols for diagrams – Part 13: Analogue elements*

IEC 848: 1988, *Preparation of function charts for control systems*

ISO/AFNOR: 1989, *Dictionary of computer science, ISBN 2-12-486911-6*

ISO/IEC 646: 1991, *Information technology – ISO 7-bit coded character set for information processing interchange*

ISO 8601: 1988, *Data elements and interchange formats – Information interchange – Representations of dates and times*

ISO 7185: 1990, *Information technology – Programming languages – Pascal*

ISO 7498: 1984, *Information processing systems – Open systems interconnection – Basic reference model*

1.3 Definitions

For the purposes of this part of IEC 1131, the following definitions apply. Definitions applying to all parts of IEC 1131 are given in part 1.

NOTES

- 1 Terms defined in this subclause are italicized where they appear in the bodies of definitions.
- 2 The notation "(ISO)" following a definition indicates that the definition is taken from the ISO/AFNOR *Dictionary of computer science*.
- 3 The ISO/AFNOR *Dictionary of computer science* and the *International Electrotechnical Vocabulary* should be consulted for terms not defined in this standard.

1.3.1 **absolute time**: The combination of time of day and date information.

1.3.2 **access path**: The association of a symbolic name with a variable for the purpose of open communication.

1.3.3 **action**: A Boolean variable, or a collection of operations to be performed, together with an associated control structure, as specified in 2.6.4.

1.3.4 **action block**: A 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 2.6.4.5.

1.3.5 **aggregate**: A structured collection of data objects, forming a *data type*. (ISO)

1.3.6 **argument**: Synonymous with *input parameter* or *output parameter*.

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

1.3.8 **assignment**: A mechanism to give a value to a variable or to an *aggregate*. (ISO)

1.3.9 **based number**: A number represented in a specified base other than ten.

1.3.10 **bistable function block**: A *function block* with two stable states controlled by one or more inputs.

1.3.11 **bit string**: A data element consisting of one or more bits.

1.3.12 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 *invoked*.

1.3.13 call: A language construct for *invoking* the execution of a *function* or *function block*.

1.3.14 character string: An *aggregate* that consists of an ordered sequence of characters.

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

1.3.16 compile: To translate a *program organization unit* or *data type* specification into its machine language equivalent or an intermediate form.

1.3.17 configuration: A language element corresponding to a *programmable controller system* as defined in IEC 1131-1.

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

1.3.19 data type: A set of values together with a set of permitted operations. (ISO)

1.3.20 date and time: The date within the year and the time of day, represented according to ISO 8601.

1.3.21 declaration: The mechanism for establishing the definition of a *language element*. A declaration normally involves attaching an identifier to the language element, and allocating attributes such as *data types* and algorithms to it.

1.3.22 delimiter: A character or combination of characters used to separate program *language elements*.

1.3.23 direct representation: A 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.

1.3.24 double word: A data element containing 32 bits.

1.3.25 evaluation: The process of establishing a value for an expression or a *function*, or for the *outputs* of a network or *function block*, during program execution.

1.3.26 execution control element: A *language element* which controls the flow of program execution.

1.3.27 falling edge: The change from 1 to 0 of a Boolean variable.

1.3.28 function: A *program organization unit* which, when executed, yields exactly one data element (which may be multi-valued, e.g., an *array* or *structure*), and whose *invocation* can be used in textual languages as an *operand* in an expression.

1.3.29 function block instance (function block): An *instance* of a *function block type*.

1.3.30 function block type: A programmable controller programming *language element* consisting of: (i) the definition of a data structure partitioned into input, output, and internal variables; and (ii) a set of operations to be performed upon the elements of the data structure when an *instance* of the function block type is *invoked*.

1.3.31 function block diagram: One or more networks of graphically represented *functions*, *function blocks*, data elements, *labels*, and connective elements.

1.3.32 generic data type: A *data type* which represents more than one type of data, as specified in 2.3.2.

1.3.33 global scope: Scope of a declaration applying to all program organization units within a *resource* or *configuration*.

1.3.34 global variable: A variable whose *scope* is *global*.

1.3.35 hierarchical addressing: The *direct representation* of a data element as a member of a physical or logical hierarchy, e.g., a point within a module which is contained in a rack, which in turn is contained in a cubicle, etc.

1.3.36 Identifier: A combination of letters, numbers, and underline characters, as specified in 2.1.2, which begins with a letter or underline and which names a *language element*.

1.3.37 Initial value: The value assigned to a variable at system start-up.

1.3.38 Input parameter (Input): A parameter which is used to supply an argument to a *program organization unit*.

1.3.39 Instance: An individual, named copy of the data structure associated with a *function block type* or *program type*, which persists from one *invocation* of the associated operations to the next.

1.3.40 Instance name: An *identifier* associated with a specific *instance*.

1.3.41 Instantiation: The creation of an *instance*.

1.3.42 Integer literal: A *literal* which directly represents a value of type SINT, INT, DINT, LINT, BOOL, BYTE, WORD, DWORD, or LWORD, as defined in 2.3.1.

1.3.43 Invocation: The process of initiating the execution of the operations specified in a *program organization unit*.

1.3.44 keyword: A lexical unit that characterizes a *language element*, e.g., "IF".

1.3.45 label: A language construction naming an instruction, network, or group of networks, and including an *identifier*.

1.3.46 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 part of IEC 1131.

1.3.47 literal: A lexical unit that directly represents a value. (ISO)

1.3.48 local scope: The *scope* of a *declaration* or *label* applying only to the *program organization unit* in which the declaration or label appears.

1.3.49 logical location: The 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.

1.3.50 long real: A real number represented in a *long word*.

1.3.51 long word: A 64-bit data element.

1.3.52 memory (user data storage): A functional unit to which the user program can store data and from which it can retrieve the stored data.

1.3.53 named element: An element of a *structure* which is named by its associated *identifier*.

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

1.3.55 operand: A *language element* on which an operation is performed.

1.3.56 operator: A symbol that represents the action to be performed in an operation.

1.3.57 output parameter (output): A *parameter* which is used to return the result(s) of the *evaluation* of a *program organization unit*.

1.3.58 overloaded: With respect to an operation or *function*, capable of operating on data of different types, as specified in 2.5.1.4.

1.3.59 power flow: The symbolic flow of electrical power in a ladder diagram, used to denote the progression of a logic solving algorithm.

1.3.60 program (verb): To design, write, and test user programs.

1.3.61 program organization unit: A *function*, *function block*, or *program*.

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

1.3.62 real literal: A *literal* representing data of type REAL or LREAL.

1.3.63 resource: A *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 1131-1.

1.3.64 retentive data: Data stored in such a way that its value remains unchanged after a power down / power up sequence.

1.3.65 return: A language construction within a *program organization unit* designating an end to the execution sequences in the unit.

1.3.66 rising edge: The change from 0 to 1 of a Boolean variable.

1.3.67 scope: That portion of a *language element* within which a *declaration* or *label* applies.

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

1.3.69 semigraphic representation: Representation of graphic information by the use of a limited set of characters.

1.3.70 single data element: A data element consisting of a single value.

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

1.3.72 structured data type: An *aggregate* data type which has been declared using a STRUCT or FUNCTION_BLOCK declaration.

1.3.73 subscripting: A 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.

1.3.74 symbolic representation: The use of *identifiers* to name variables.

1.3.75 task: An *execution control element* providing for periodic or triggered execution of a group of associated *program organization units*.

1.3.76 time literal: A *literal* representing data of type TIME, DATE, TIME_OF_DAY, or DATE_AND_TIME.

1.3.77 transition: The condition whereby control passes from one or more predecessor *steps* to one or more successor steps along a directed link.

1.3.78 unsigned integer: An *integer literal* not containing a leading plus (+) or minus (-) sign.

1.3.79 wired OR: A construction for achieving the Boolean OR function in the LD language by connecting together the right ends of horizontal connectives with vertical connectives.

1.4 Overview and general requirements

This part of IEC 1131 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.

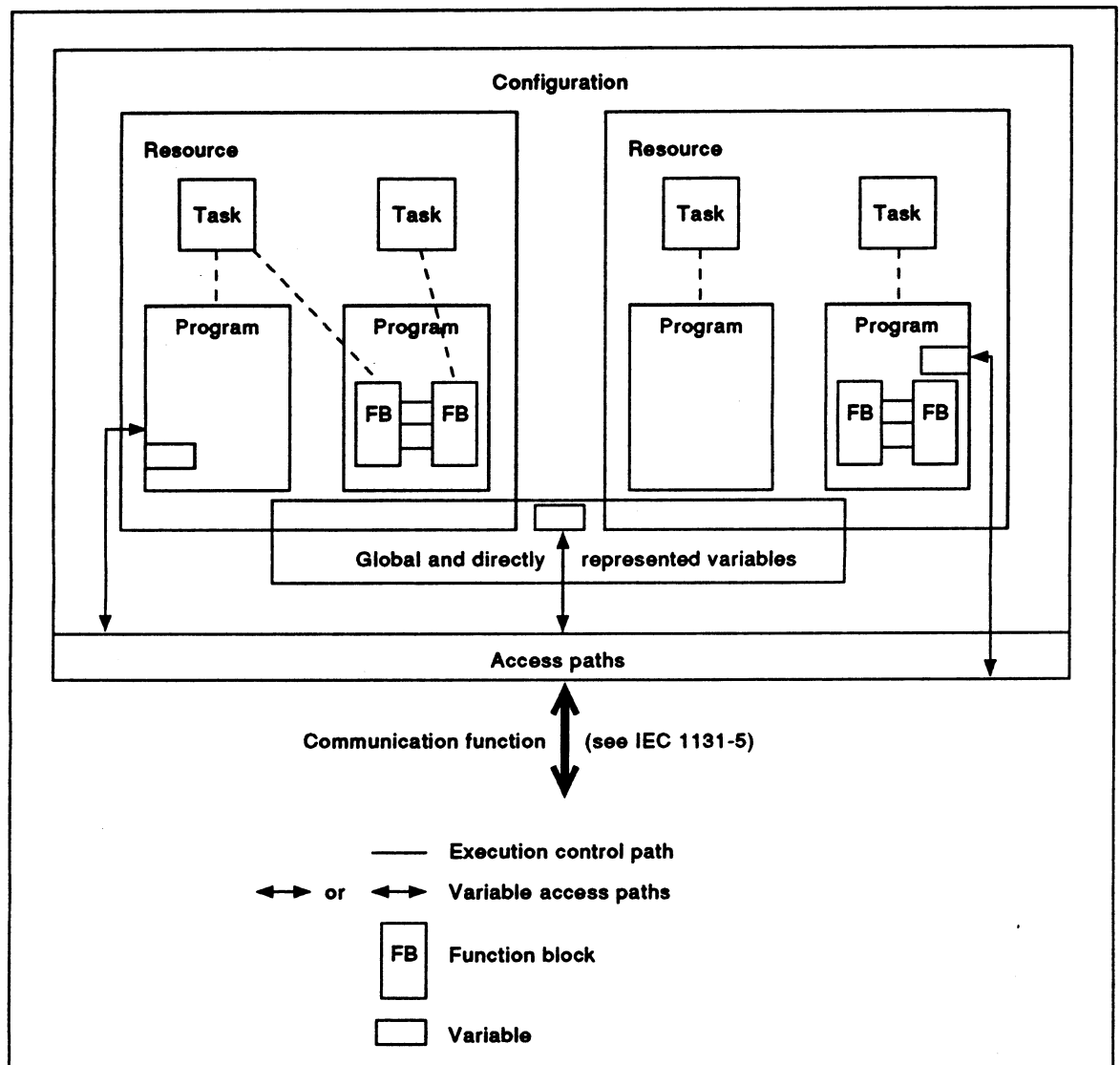
In addition, features are defined which facilitate communication among programmable controllers and other components of automated systems.

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 part; however, such an environment shall be capable of producing textual or graphic program documentation in the formats specified in this part.

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.

1.4.1 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 part, that is, *programs* and *function blocks*; and *configuration elements*, namely, *configurations*, *resources*, *tasks*, *global variables*, and *access paths*, which support the installation of programmable controller *programs* into programmable controller systems.



NOTES

- 1 This figure is illustrative only. The graphical representation is not normative.
- 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 1131-1. 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 1131-1. 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 blocks* 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 1131-1. The starting of a *configuration* shall cause the initialization of its *global variables* according to the rules given in 2.4.2, 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 2.7.2, while mechanisms for the starting and stopping of *configurations* and *resources* via communication functions are defined in IEC 1131-5.

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 1131-1. 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 2.7.1.

The mapping of the language elements defined in this subclause on to communication objects is defined in IEC 1131-5.

1.4.2 *Communication model*

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

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

Variable values can be communicated between programs in the same configuration via *global variables* such as the variable *x* illustrated in figure 2b. These variables shall be declared as GLOBAL in the configuration, and as EXTERNAL in the programs, as specified in 2.4.3.

As illustrated in figure 2c, the values of variables can be communicated between different parts of a program, between programs in the same or different configurations, or between a programmable controller program and a non-programmable controller system, using the communication function blocks defined in IEC 1131-5 and described in 2.5.2.3.5. In addition, programmable controllers or non-programmable controller systems can transfer data which is made available by *access paths*, as illustrated in figure 2d, using the mechanisms defined in IEC 1131-5.

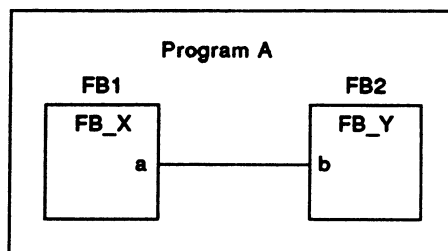


Figure 2a – Data flow connection within a program

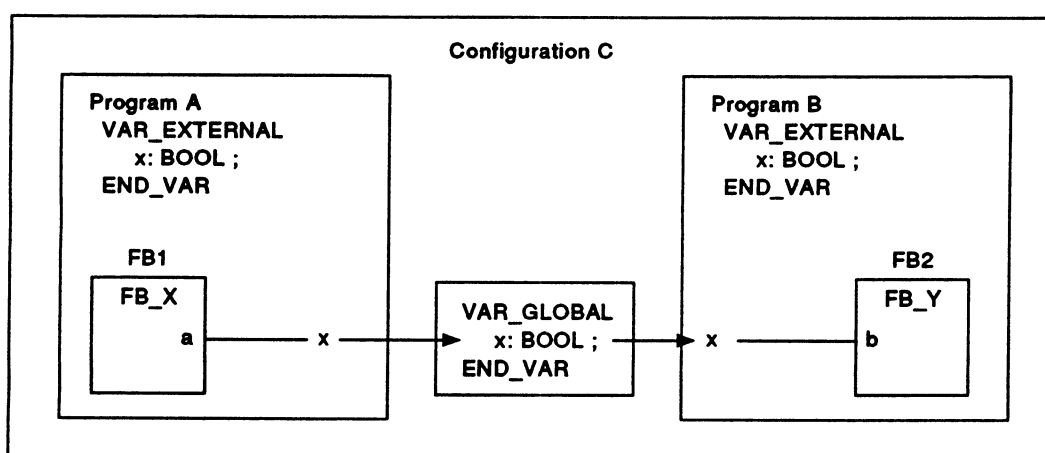


Figure 2b – Communication via GLOBAL variables

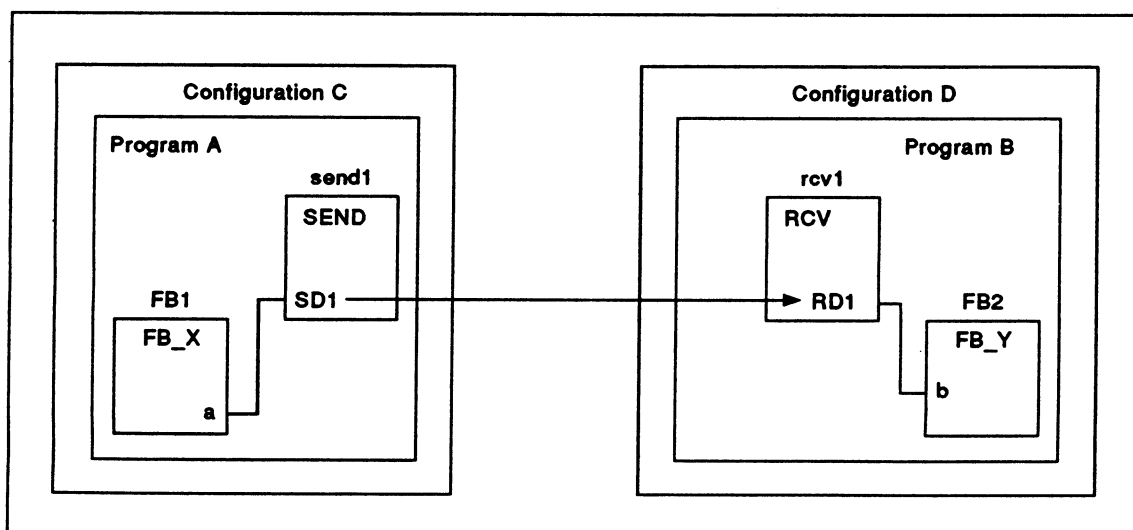


Figure 2c – Communication function blocks

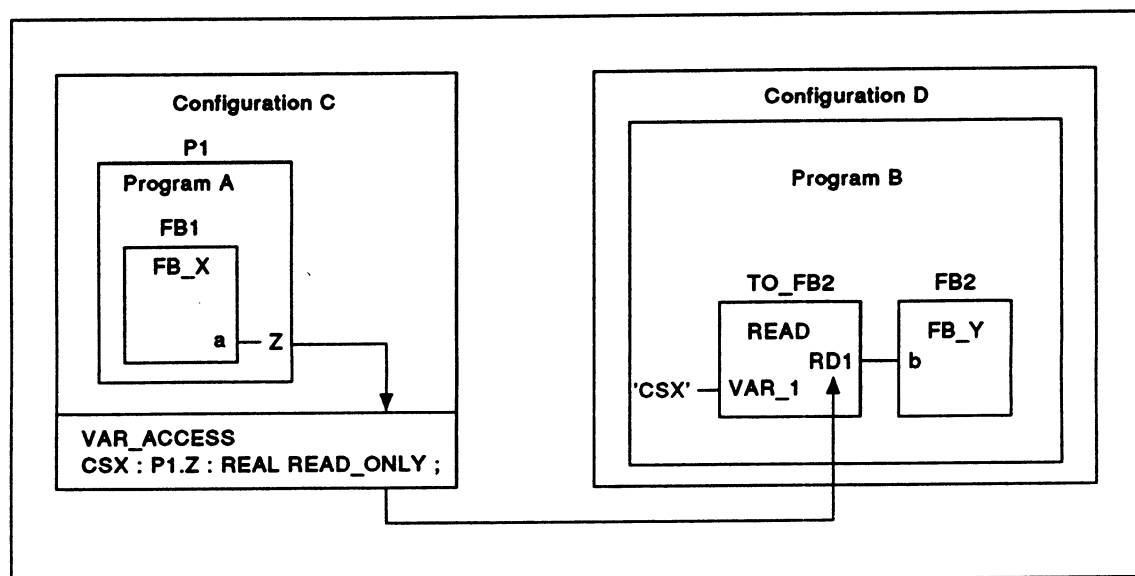


Figure 2d – Communication via access paths

NOTES

- 1 This figure is illustrative only. The graphical representation is not normative.
- 2 In these examples, configurations c and d are each considered to have a single resource.
- 3 The details of the communication function blocks are not shown in this figure. See 2.5.2.3.5 and IEC 1131-5.
- 4 As specified in 2.7, access paths can be declared on directly represented variables, global variables, or program input, output, or internal variables.
- 5 IEC 1131-5 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

1.4.3 Programming model

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

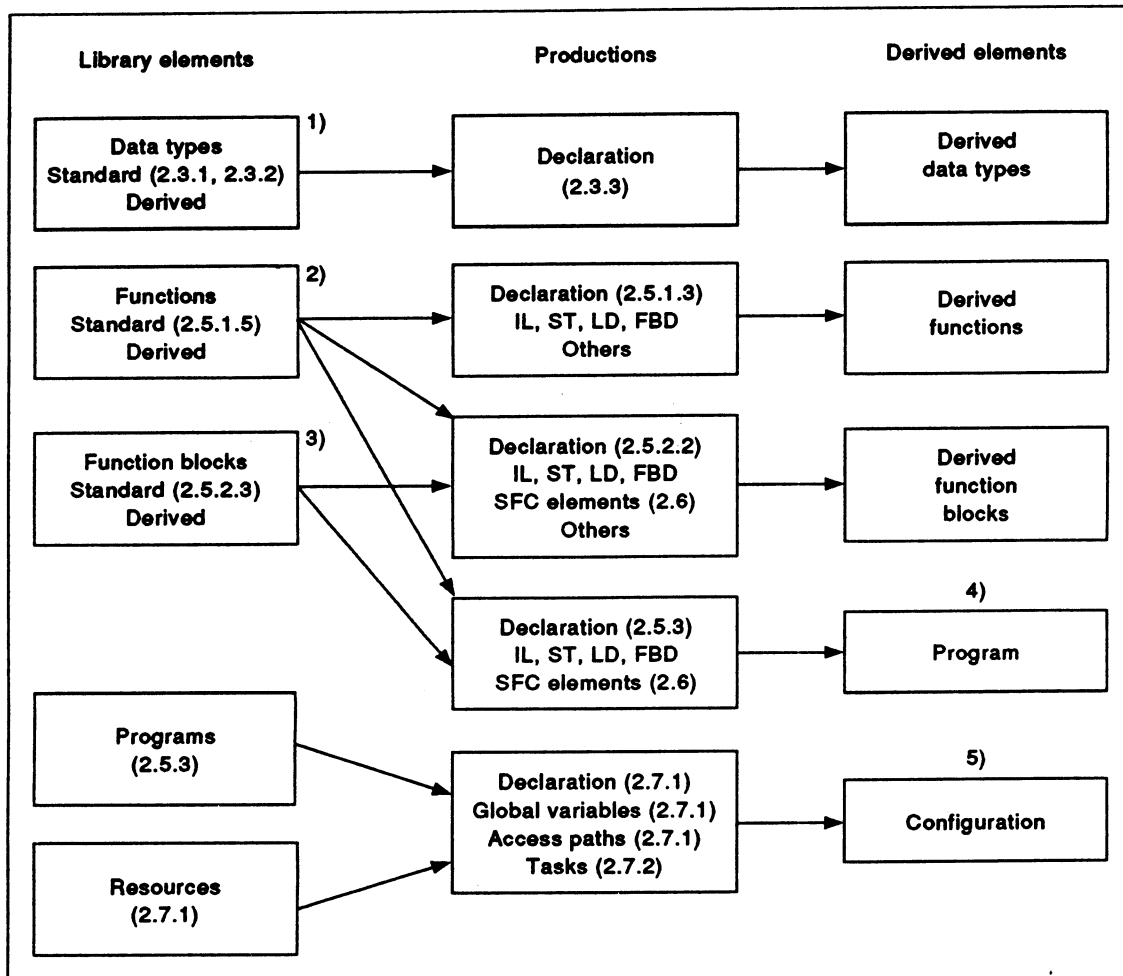
- Data types (2.3)
- Program organization units (2.5)
 - Functions (2.5.1)
 - Function blocks (2.5.2)
 - Programs (2.5.3)
- Sequential Function Chart (SFC) elements (2.6)
- Configuration elements (2.7)
 - Global variables (2.7.1)
 - Resources (2.7.1)
 - Tasks (2.7.2)
 - Access paths (2.7.1)

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 2.3.3, using the standard data types specified in 2.3.1 and 2.3.2 and any previously derived data types.
- 2) Derived *functions* can be declared as specified in 2.5.1.3, using standard or derived data types, the standard functions defined in 2.5.1.5, and any previously derived functions. This declaration shall use the mechanisms defined for the IL, ST, LD or FBD language.
- 3) Derived *function blocks* can be declared as specified in 2.5.2.2, using standard or derived data types and functions, the standard function blocks defined in 2.5.2.3, and any previously derived 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 2.6.
- 4) A *program* shall be declared as specified in 2.5.3, 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 2.6.
- 5) *Programs* can be combined into *configurations* using the elements defined in 2.7, 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, e.g., 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*. The means by which a user program written in one of the languages defined in this standard invokes the execution of, and accesses the data associated with, such a derived function or function block shall be as defined in this standard.



NOTES

- 1 The parenthesized numbers 1) to 5) refer to corresponding paragraphs in 1.4.3.
- 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
 LD-Ladder Diagram (4.2)
 FBD-Function Block Diagram (4.3)
 IL-Instruction List (3.2)
 ST-Structured Text (3.3)
 Others: Other programming languages (1.4.3)

1.5 Compliance

This subclause defines the requirements which shall be met by programmable controller systems and programs which claim compliance with this part of IEC 1131.

1.5.1 Programmable controller systems

A programmable controller system, as defined in IEC 1131-1, which claims to comply, wholly or partially, with the requirements of this part of IEC 1131 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 1131-3, 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 1131. Table titles are to be taken from the following table.

Table title	For features in:
Common elements	Clause 2
Common textual elements	Subclause 3.1
IL language elements	Subclauses 3.2.1 to 3.2.3
ST language elements	Subclauses 3.3.1 to 3.3.2.4
Common graphical elements	Subclauses 4.1 to 4.1.4
LD language elements	Subclauses 4.2 to 4.2.6
FBD language elements	Subclauses 4.3 to 4.3.3

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

- shall not require the inclusion of substitute or additional language elements in order to accomplish any of the features specified in this part;
- shall be accompanied by a document that specifies the values of all implementation-dependent parameters as listed in annex D;
- shall be able to determine whether or not a user's language element violates any requirement of this part, where such a violation is not designated 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 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 part. Such features shall be described as being "extensions to the <language> language as defined in IEC 1131-3";

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 the implementation-dependent features specified in annex D;

h) shall not use any of the standard data type, function or function block names defined in this part for manufacturer-defined features whose functionality differs from that described in this part;

i) shall be accompanied by a document defining, in the form specified in annex A, the formal syntax of all textual language elements supported by the system.

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.

1.5.2 Programs

A programmable controller program complying with the requirements of IEC 1131-3:

- a) shall use only those features specified in this part 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-dependent features.

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, except as these results are influenced by program execution timing, the use of implementation-dependent features (as listed in annex D) in the program, and the execution of error handling procedures.

2 Common elements

This clause defines textual and graphic elements which are common to all the programmable controller programming languages specified in this part of IEC 1131.

2.1 Use of printed characters

2.1.1 Character set

Textual languages and textual elements of graphic languages shall be represented in terms of the "Basic code table" of the ISO/IEC 646 character set.

The encoding of characters from national or extended (8-bit) character sets shall be consistent with ISO/IEC 646.

The *required character set* shown as feature 1 in table 1 consists of all the characters in columns 3 to 7 of the "Basic code table" given as table 1 in ISO/IEC 646, except for lower-case letters and those character positions which are reserved or optionally available for use in national character sets.

The manufacturer shall support one option (a or b) for each of features (3a,b) to (6a,b) of table 1, according to the following rules:

- The "pound sign" (£) shall be used in place of the "number sign" (#) when the former occupies character position 2/3 of a national implementation of the ISO/IEC 646 character set.
- The "currency sign" shall be used in place of the "dollar sign" (\$) when the former occupies character position 2/4 of a national implementation of the ISO/IEC 646 character set.
- When the 7/12 character position in the ISO/IEC 646 character set is used by another character in a national set, the "exclamation mark" (!) at position 2/1 shall be used to represent vertical lines.
- For delimitation of subscripts, the left and right parentheses "()" shall be used in place of the left and right brackets "[]" when the latter occupy character positions of a national implementation of the ISO/IEC 646 character set.

NOTE - The use of characters from national character sets is a typical extension of this standard.

Table 1 – Character set features

No	Description
1	Required character set (see 2.1.1)
2	Lower case characters
3a 3b	Number sign (#) or Pound sign (£)
4a 4b	Dollar sign (\$) or Currency sign
5a 5b	Vertical bar () or Exclamation mark (!)
6a 6b	Subscript delimiters: Left and right brackets "[]" or Left and right parentheses "()"
NOTE - When lower-case letters (feature 2) are supported, the case of letters shall not be significant in language elements (except within terminal symbols as defined in annexes A and B, comments as defined in 2.1.5, string literals as defined in 2.2.2, and variables of type STRING as defined in 2.3.1), e.g., the identifiers "abcd", "ABCD" and "aBCd" shall be interpreted identically.	

2.1.2 Identifiers

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

Underlines shall be significant in identifiers, e.g., "A_BCD" and "AB_CD" shall be interpreted as different identifiers. Multiple leading or multiple embedded underlines are not allowed.

Identifiers shall not contain imbedded space (SP) characters.

At least six characters of uniqueness shall be supported in all systems which support the use of identifiers, e.g., "ABCDE1" shall be interpreted as different from "ABCDE2" in all such systems.

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

2.1.3 Keywords

Keywords are unique combinations of characters utilized as individual syntactic elements as defined in annex B. All keywords used in this part are listed in annex C. Keywords shall not contain imbedded spaces. The keywords listed in annex C shall not be used for any other purpose, e.g., variable names or extensions as defined in 1.5.1.

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

2.1.4 Use of spaces

The user shall be allowed to insert one or more spaces (code position 2/0 in the ISO/IEC 646 character set) anywhere in the text of programmable controller programs except within keywords, literals, identifiers, or delimiter combinations (e.g., for comments as defined below).

2.1.5 Comments

User comments shall be delimited at the beginning and end by the special character combinations "(*" and "*)", respectively, as shown in table 3. Except in the IL language as defined in 3.2, comments shall be permitted anywhere in the program where spaces are allowed, except within character string literals as defined in 2.2.2. Comments shall have no syntactic or semantic significance in any of the languages defined in this part.

Nested comments are not allowed, e.g., (* (* NESTED *) *).

Table 3 – Comment feature

No.	Feature description	Examples
1	Comments	<pre>(*****)</pre> <pre>(* A framed comment *)</pre> <pre>(*****)</pre>

2.2 External representation of data

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

2.2.1 Numeric 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 -).

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 4.

Table 4 – 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 (240 decimal)
5	Base 8 literals	8#377 (255 decimal) 8#340 (240 decimal)
6	Base 16 literals	16#FF or 16#ff (255 decimal) 16#E0 or 16#e0 (240 decimal)
7	Boolean zero and one	0 1
8	Boolean FALSE and TRUE	FALSE TRUE
NOTE - The keywords FALSE and TRUE correspond to Boolean values of 0 and 1, respectively.		

2.2.2 Character string literals

A character string literal is a sequence of zero or more characters prefixed and terminated by the single quote character ('). In 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 table 5. Additionally, two-character combinations beginning with the dollar sign shall be interpreted as shown in table 6 when they occur in character strings.

Table 5 – Character string literal feature

No.	Example	Explanation
1	''	Empty string (length zero)
	'A'	String of length one containing the single character A
	' '	String of length one containing the "space" character
	'\$'	String of length one containing the "single quote" character
	'\$R\$L' '\$0D\$0A'	Strings of length two containing CR and LF characters
	'\$\$1.00'	String of length five which would print as "\$1.00"

Table 6 – Two-character combinations in character strings

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

2.2.3 Time literals

The need to provide external representations for two distinct types of time-related data is recognized: *duration* data for measuring or controlling the elapsed time of a control event, and *time of day* data (which may also include date information) for synchronizing the beginning or end of a control event to an absolute time reference.

Duration and time of day literals shall be delimited on the left by the keywords defined in 2.2.3.1 and 2.2.3.2.

2.2.3.1 Duration

Duration data shall be delimited on the left by the keyword T#, TIME#, 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 7. The least significant time unit can be written in real notation without exponent.

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

"Overflow" of the most significant unit of a duration literal is permitted, e.g., the notation T#25h_15m is permitted.

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

Table 7 – Duration literal features

No.	Feature description	Examples
1a	Duration literals without underlines: Short prefix	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#14.7s
2a	Duration literals with underlines: 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

2.2.3.2 Time of day and date

Prefix keywords for time of day and date literals shall be as shown in table 8. As illustrated in table 9, representation of time-of-day and date information shall be as specified in ISO 8601.

Table 8 – 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 9 – 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

2.3 Data types

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 (see 2.5.1.4). A mechanism for the user or manufacturer to specify additional data types is also defined.

2.3.1 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 10.

Table 10 – Elementary data types

No.	Keyword	Data type	Bits	Range
1	BOOL	Boolean	1	Note 8
2	SINT	Short integer	8	Note 2
3	INT	Integer	16	Note 2
4	DINT	Double integer	32	Note 2
5	LINT	Long integer	64	Note 2
6	USINT	Unsigned short integer	8	Note 3
7	UINT	Unsigned integer	16	Note 3
8	UDINT	Unsigned double integer	32	Note 3
9	ULINT	Unsigned long integer	64	Note 3
10	REAL	Real numbers	32	Note 4
11	LREAL	Long reals	64	Note 5
12	TIME	Duration	Note 1	Note 6
13	DATE	Date (only)	Note 1	Note 6
14	TIME_OF_DAY or TOD	Time of day (only)	Note 1	Note 6
15	DATE_AND_TIME or DT	Date and time of day	Note 1	Note 6
16	STRING	Variable-length character string	Note 1	Note 7
17	BYTE	Bit string of length 8	8	Note 7
18	WORD	Bit string of length 16	16	Note 7
19	DWORD	Bit string of length 32	32	Note 7
20	LWORD	Bit string of length 64	64	Note 7

NOTES

1 The length of these data elements is implementation-dependent.

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

3 The range of values for variables of this data type is from 0 to $(2^{Bits})-1$.

4 The range of values for variables of this data type shall be as defined in IEC 559 for the basic single width floating-point format.

5 The range of values for variables of this data type shall be as defined in IEC 559 for the basic double width floating-point format.

6 The range of values for variables of this data type is implementation-dependent.

7 A numeric range of values does not apply to this data type.

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

2.3.2 Generic data types

In addition to the data types in table 10, the hierarchy of generic data types shown in table 11 shall be used as defined in 2.5.1.4 in the specification of overloaded inputs and outputs of standard functions and function blocks. Generic data types are identified by the prefix "ANY".

Table 11 – Hierarchy of generic data types

ANY ANY_NUM ANY_REAL LREAL REAL ANY_INT LINT, DINT, INT, SINT ULINT, UDINT, UINT, USINT ANY_BIT LWORD, DWORD, WORD, BYTE, BOOL STRING ANY_DATE DATE_AND_TIME DATE TIME_OF_DAY TIME Derived (see notes)	NOTES 1 Generic data types shall not be used in user-declared program organization units as defined in 2.5. 2 The generic type of a <i>subrange</i> derived type (feature 3 of table 12) shall be ANY_INT. 3 The generic type of a <i>directly</i> derived type (feature 1 of table 12) shall be the same as the generic type of the elementary type from which it is derived. 4 The generic type of all other derived types defined in table 12 shall be ANY.
---	---

2.3.3 Derived data types

2.3.3.1 Declaration

Derived (i.e., user- or manufacturer-specified) 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 2.3.1, in variable declarations as defined in 2.4.3.

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.

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.

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.

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 2.4.1.2.

2.3.3.2 Initialization

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 tables 12 and 14, the default initial values of elements of data types **ANALOG_SIGNAL_TYPE** and **ANALOG_SIGNAL_RANGE** are **SINGLE_ENDED** and **UNIPOLAR_1_5V**, respectively.

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.

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 2.4.2.

The default maximum length of elements of type **STRING** 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. For example, if type **STR10** is declared by

```
TYPE STR10 : STRING(10) := 'ABCDEF' ; END_TYPE
```

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

Table 12 – Data type declaration feature

No.	Feature/textual example
1	Direct derivation from elementary types, e.g.: TYPE R : REAL ; END_TYPE
2	Enumerated data types, e.g.: TYPE ANALOG_SIGNAL_TYPE : (SINGLE_ENDED, DIFFERENTIAL) ; END_TYPE
3	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
NOTE - For examples of the use of these types in variable declarations, see 2.3.3.3, 2.4.1.2, and table 17.	

Table 13 – Default initial values

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)

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 ; END_TYPE
4	Initialization of array data types, e.g.: TYPE ANALOG_16_INPUT_DATAI : ARRAY [1..16] OF ANALOG_DATA := 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 ; MAX_SCALE : ANALOG_DATA := 4095 ; 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 := 9999) ; END_TYPE

2.3.3.3 Usage

The usage of variables which are declared (as defined in 2.4.3.1) to be of derived data types shall conform to the following rules:

- 1) A single-element variable, as defined in 2.4.1.1, of a derived type, can be used anywhere that a variable of its "parent's" type can be used, e.g. variables of the types R and PI as shown in tables 12 and 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
```

2) An element of a multi-element variable, as defined in 2.4.1.2, can be used anywhere the "parent" type can be used, e.g., 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.

This rule can also be applied recursively, e.g., 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.

2.4 Variables

In contrast to the external representations of data described in 2.2, *variables* provide a means of identifying data objects whose contents may change, e.g., 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 2.3.1, or one of the derived types which are declared as defined in 2.3.3.1.

2.4.1 Representation

2.4.1.1 Single-element variables

A *single-element variable* is defined as a variable which represents a single data element of one of the elementary types defined in 2.3.1; a derived enumeration or subrange type as defined in 2.3.3.1; or a derived type whose "parentage", as defined recursively in 2.3.3.3, is traceable to an elementary, enumeration or subrange type. This subclause defines 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.

Identifiers, as defined in 2.1.2, 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 "%" (position 2/5 in the ISO/IEC 646 code table), a *location prefix* and a *size prefix* from table 15, and one or more unsigned integers, separated by periods (.)

Examples of directly represented variables are:

%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

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.

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 only permitted in *programs*, as defined in 2.5.3, and in *configurations* and *resources* as defined in 2.7.1. The maximum number of levels of hierarchical addressing is an implementation-dependent parameter.

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

No.	Prefix	Meaning
1	I	Input location
2	Q	Output location
3	M	Memory location
4	X	Single bit size
5	None	Single bit size
6	B	Byte (8 bits) size
7	W	Word (16 bits) size
8	D	Double word (32 bits) size
9	L	Long (quad) word (64 bits) size
NOTES 1 Unless otherwise declared, the data type of a directly addressed variable of "single bit" size shall be BOOL. 2 National standards organizations can publish tables of translations of these prefixes.		

2.4.1.2 Multi-element variables

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

An *array* is a collection of data elements of the same data type referenced by one or more *subscripts* enclosed in brackets and separated by commas. A subscript shall be an expression yielding a value corresponding to one of the sub-types of generic type ANY_INT as defined in table 11.

An example of the use of array variables in the ST language as defined in 3.3 is:

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

The maximum number of subscripts, and the maximum range of subscript values, which may be used to access array variables is an implementation-dependent parameter.

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.

For instance, 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 3.3 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
```

The maximum number of levels of structure element addressing is an implementation-dependent parameter.

2.4.2 Initialization

When a configuration element (*resource* or *configuration*) is "started" as defined in 1.4.1, 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 16, 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 1131-1, 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 1131-1, the initial values of retentive variables shall be the user-specified initial values, or the default value, as defined in 2.3.3.2, 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, as defined in 2.3.3.2, 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 1131-1 shall be initialized in an implementation-dependent manner.

2.4.3 Declaration

Each programmable controller program organization unit type declaration (i.e., each declaration of a *program*, *function*, or *function block*, as defined in 2.5) 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 16, followed in the case of VAR by zero or one occurrence of the qualifier RETAIN or the qualifier CONSTANT, and in the case of VAR_OUTPUT by zero or one occurrence of the qualifier 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.

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 parameter 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 2.7.1. 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*.

Table 16 – 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 NOTE - Examples of the use of these variables are given in figures 11b and 12
VAR_EXTERNAL	Supplied by configuration via VAR_GLOBAL (2.7.1) Can be modified within organization unit
VAR_GLOBAL	Global variable declaration (2.7.1)
VAR_ACCESS	Access path declaration (2.7.1)
RETAIN	Retentive variables (see preceding text)
CONSTANT	Constant (variable cannot be modified)
AT	Location assignment (2.4.3.1)
NOTE - The usage of these keywords is a feature of the program organization unit or configuration element in which they are used; see 2.5 and 2.7.	

2.4.3.1 Type assignment

As shown in table 17, 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* 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 is defined in 2.5.

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.

Table 17 – Variable type assignment features

No.	Feature/examples	
1	Declaration of directly represented, non-retentive variable	
	VAR AT %IW6.2 : WORD ; AT %MW6 : INT ; END_VAR	16-bit string (note 2) 16-bit integer, initial value = 0
2	Declaration of directly represented retentive variables	
	VAR RETAIN AT %QW5 : WORD ; END_VAR	At cold restart, %QW5 will be initialized to a 16-bit string with value 0
3	Declaration of locations of symbolic variables	
	VAR_GLOBAL LIM_SW_5 AT %IX27 : BOOL ; CONV_START AT %QX25 : BOOL ; TEMPERATURE AT %IW28 : INT ; END_VAR	Assigns input bit 27 to the Boolean variable LIM_SW_5 (note 2) Assigns output bit 25 to the Boolean variable CONV_START Assigns input word 28 to the integer variable TEMPERATURE (note 2)
4	Array location assignment	
	VAR INARRAY 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)
5	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".
6	Array declaration	
	VAR THREE : ARRAY[1..5,1..10,1..8] OF INT ; END_VAR	Allocates 400 memory words for a three-dimensional array of integers
7	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
8	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)
NOTES: 1 Features 1 to 4 can only be used in PROGRAM and VAR_GLOBAL declarations, as defined in 2.5.3 and 2.7.1 respectively. 2 Initialization of system inputs is implementation-dependent; see 2.4.2.		

2.4.3.2 Initial value assignment

The VAR...END_VAR construction shown in table 18 shall be used to specify initial values of directly represented variables. This construction shall also be used to assign initial values of symbolically represented single- or multi-element variables (the usage of the VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT constructions is defined in 2.5).

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 variables.

Parentheses can be used as a repetition factor in array initialization lists, e.g., "2(1,2,3)" is equivalent to the initialization sequence "1,2,3,1,2,3".

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

When a variable is declared to be of a derived, structured data type as defined in 2.3.3.1, initial values for the elements of the variable can be declared in a parenthesized list following the data type identifier, as shown in table 18. 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.

Table 18 – Variable initial value assignment features

No.	Feature/examples	
1	Initialization of directly represented, non-retentive variables	
	<pre>VAR AT %QX 5.1 : BOOL := 1 ; AT %MW6 : INT := 8 ; END_VAR</pre>	Boolean type, initial value =1 Initializes a memory word to integer 8
2	Initialization of directly represented retentive variables	
	<pre>VAR RETAIN AT %QW5 : WORD := 16#FF00 ; END_VAR</pre>	At cold restart, the 8 most significant bits of the 16-bit string at output word 5 are to be initialized to 1 and the 8 least significant bits to 0
3	Location and initial value assignment to symbolic variables	
	<pre>VAR VALVE_POS AT %QW28 : INT := 100 ; END_VAR</pre>	Assigns output word 28 to the integer variable VALVE_POS, with an initial value of 100
4	Array location assignment and initialization	
	<pre>VAR OUTARY AT %QW6: ARRAY [0..9] OF INT := 10(1) ; END_VAR</pre>	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	
	<pre>VAR MYBIT : BOOL := 1 ; OKAY : STRING(10) := 'OK' ; END_VAR</pre>	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 length 2 and contains the two-byte sequence of characters 'OK' in the ISO/IEC 646 character set, in an order appropriate for printing as a character string.
6	Array initialization	
	<pre>VAR BITS : ARRAY [0..7] OF BOOL := 1,1,0,0,0,1,0,0 ; TBT : ARRAY[1..2,1..3] OF INT := 1,2,3(4),6 ; END_VAR</pre>	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] := 1, TBT[1,2] := 2, TBT[1,3] := 4, TBT[2,1] := 4, TBT[2,2] := 4, TBT[2,3] := 6,
7	Retentive array declaration and initialization	
	<pre>VAR RETAIN RTBT : ARRAY [1..2, 1..3] OF INT := 1,2,3(4) ; END_VAR</pre>	Declares retentive array RTBT with "cold restart" initial values of: RTBT[1,1] := 1, RTBT[1,2] := 2, RTBT[1,3] := 4, RTBT[2,1] := 4, RTBT[2,2] := 4, RTBT[2,3] := 0
8	Initialization of structured variables	
	<pre>VAR MODULE_8_CONFIG : ANALOG_16_INPUT_CONFIGURATION (SIGNAL_TYPE := DIFFERENTIAL, CHANNEL[5].RANGE := BIPOLAR_10_V, CHANNEL[5].MIN_SCALE := 0, CHANNEL[5].MAX_SCALE := 500) ; END_VAR</pre>	Initialization of a variable of derived data type (see table 12)
9	Initialization of constants	
	<pre>VAR CONSTANT PI : REAL := 3.141592 ; END_VAR</pre>	
	NOTE - Features 1 to 4 can only be used in PROGRAM and VAR_GLOBAL declarations, as defined in 2.5.3 and 2.7.1 respectively.	

2.5 Program organization units

The program organization units defined in this part of IEC 1311 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 invocation of a program organization unit shall not cause the invocation of another program organization unit of the same type.

2.5.1 Functions

For the purposes of programmable controller programming languages, a *function* is defined as a program organization unit which, when executed, yields exactly one data element (which can be multi-valued, e.g., an array or structure), and whose invocation can be used in textual languages as an operand in an expression. For example, the SIN and COS functions could be used as shown in figure 4.

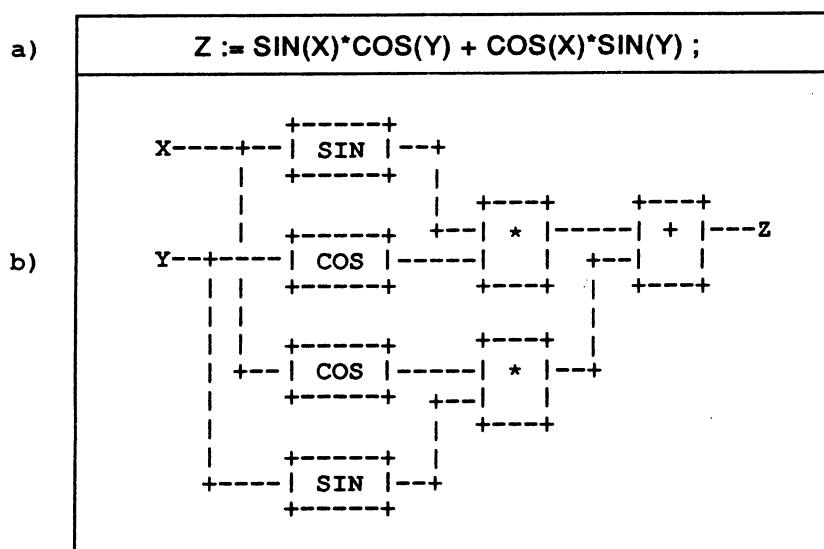


Figure 4 – Examples of function usage

- a) Structured Text (ST) language; see subclause 3.3
- b) Function Block Diagram (FBD) language; see subclause 4.3

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

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

2.5.1.1 Representation

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

In the graphic languages defined in clause 4 of this part, functions shall be represented as graphic blocks according to the following rules:

- 1) The form of the block shall be rectangular or square.
- 2) The size and proportions of the block may vary depending on the number of inputs and other information to be displayed.
- 3) The direction of processing through the block shall be from left to right (input parameters on the left and output parameter on the right).
- 4) The function name or symbol, as specified below, shall be located inside the block.
- 5) Provision shall be made for formal input parameter names appearing at the inside left of the block when the block represents:
 - one of the standard functions defined in subclause 2.5.1.5, when the given graphical form includes the formal parameter names; or
 - any additional function declared as specified in subclause 2.5.1.3.
- 6) Since the name of the function is used for the assignment of its output value as specified in 2.5.1.3, no formal output parameter name shall be shown at the right side of the block.
- 7) Actual parameter connections shall be shown by signal flow lines.
- 8) 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 ISO/IEC 646 character set, this shall be represented by the upper case alphabetic "O", as shown in table 19.
- 9) The output of a graphically represented function shall be represented by a single line at the right side of the block, even though the output may be a multi-element variable.

Table 19 – Graphical negation of Boolean signals

No.	Feature	Representation
1	Negated input	
2	Negated output	
NOTE - If either of these features is supported for functions, it shall also be supported for function blocks as defined in 2.5.2, and vice versa.		

As shown in figure 5, where a formal parameter name is present in the definition of a standard function in 2.5.1.5, the formal parameter name shall also be used in the textual invocation of the function. In the latter case, the formal parameter names and associated actual values can be given in any order.

The representation of functions in textual languages shall be as specified in clause 3 of this part.

Example	Explanation
<pre> +-----+ ADD B---- ----A C---- D---- +-----+ </pre>	Graphical use of "ADD" function (See 2.5.1.5.2) (FBD language; see 4.3) (No formal parameter names)
<pre>A := ADD (B, C, D) ;</pre>	Textual use of "ADD" function (ST language; see 3.3)
<pre> +-----+ SHL B---- IN ----A C---- N +-----+ </pre>	Graphical use of "SHL" function (See 2.5.1.5.3) (FBD language; see 4.3) (Formal parameter names)
<pre>A := SHL (IN := B, N := C) ;</pre>	Textual use of "SHL" function (ST language; see 3.3)

Figure 5 – Use of formal parameter names

2.5.1.2 Execution control

As shown in table 20, an additional Boolean "EN" (Enable) input and "ENO" (Enable Out) output shall be used with functions in the LD language defined in 4.2, and their use shall also be possible in the FBD language defined in this part. These variables are considered to be available in every function according to the implicit 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 invoked, 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 one of the errors defined in annex E occurs during the execution of one of the standard functions defined in 2.5.1.5, the ENO output of that function shall be reset to FALSE (0) by the programmable controller system.

NOTE - The use of the ENO output is an allowable exception to the rule that the execution of a function yields exactly one output.

Table 20 – Use of EN input and ENO output

No.	Feature	Example
1	Use of "EN" and "ENO" Required for LD (Ladder Diagram) language (see 4.2)	<pre> +-----+ ADD_EN + ADD_OK +---+ ---+ EN ENO ---()---+ A--- ---C B--- +-----+ </pre>
2	Use of "EN" and "ENO" Optional for FBD (Function Block Diagram) language (see 4.3)	<pre> +-----+ + ADD_EN-- EN ENO ---ADD_OK A--- ---C B--- +-----+ </pre>
3	FBD without "EN" and "ENO"	<pre> +-----+ A--- + ---C B--- +-----+ </pre>

2.5.1.3 Declaration

A function shall be declared textually or graphically.

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, a colon (:), and the data type of the value to be returned by the function;
- 2) A VAR_INPUT...END_VAR construct as defined in 2.4.2, specifying the names and types of the function's input parameters;
- 3) A VAR...END_VAR construct, if required, specifying the names and types of the function's internal variables;
- 4) A *function body*, written in one of the languages defined in this part, or another programming language as defined in 1.4.3, which specifies the operations to be performed upon the input parameter(s) in order to assign one or more values to a variable with the same name as the function, which represents the value to be returned by the function;
- 5) The terminating keyword END_FUNCTION.