

Programmable controllers —

Part 3: Programming languages

The European Standard EN 61161-3:2003 has the status of a British Standard

ICS 25.040; 35.240.50

National foreword

This British Standard is the official English language version of EN 61131-3:2003. It is identical with IEC 61131-3:2003. It supersedes BS EN 61131-3:1993 which is withdrawn.

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

- aid enquirers to understand the text;
- present to the responsible international/European committee any enquiries on the interpretation, or proposals for change, and keep the UK interests informed;
- monitor related international and European developments and promulgate them in the UK.

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

Cross-references

The British Standards which implement international or European publications referred to in this document may be found in the *BSI Catalogue* under the section entitled “International Standards Correspondence Index”, or by using the “Search” facility of the *BSI Electronic Catalogue* or of British Standards Online.

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

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

Summary of pages

This document comprises a front cover, an inside front cover, the EN title page, pages 2 to 219, an inside back cover and a back cover.

The BSI copyright date displayed in this document indicates when the document was last issued.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 16 June 2003

© BSI 16 June 2003

ISBN 0 580 42021 3

Amendments issued since publication

Amd. No.	Date	Comments

EUROPEAN STANDARD

EN 61131-3

NORME EUROPÉENNE

EUROPÄISCHE NORM

March 2003

ICS 25.040; 35.240.50

Supersedes EN 61131-3:1993

English version

Programmable controllers
Part 3: Programming languages
(IEC 61131-3:2003)

Automates programmables
Partie 3: Langues de programmation
(CEI 61131-3:2003)

Speicherprogrammierbare Steuerungen
Teil 3: Programmiersprachen
(IEC 61131-3:2003)

This European Standard was approved by CENELEC on 2002-12-01. 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, Czech Republic, Denmark, Finland, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Luxembourg, Malta, Netherlands, Norway, Portugal, Slovakia, 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

© 2003 CENELEC - All rights of exploitation in any form and by any means reserved worldwide for CENELEC members.

Ref. No. EN 61131-3:2003 E



Documents
Document Sharing Hub

Foreword

The text of document 65B/456/FDIS, future edition 2 of IEC 61131-3, prepared by SC 65B, Devices, of IEC TC 65, Industrial-process measurement and control, was submitted to the IEC-CENELEC parallel vote and was approved by CENELEC as EN 61131-3 on 2002-12-01.

This European Standard supersedes EN 61131-3:1993.

The following dates were fixed:

- latest date by which the EN has to be implemented
at national level by publication of an identical
national standard or by endorsement (dop) 2003-10-01
- latest date by which the national standards conflicting
with the EN have to be withdrawn (dow) 2005-12-01

Annexes designated "normative" are part of the body of the standard.

Annexes designated "informative" are given for information only.

In this standard, annexes A, B, C, D, E and ZA are normative and annexes F and G are informative.

Annex ZA has been added by CENELEC.

Endorsement notice

The text of the International Standard IEC 61131-3:2003 was approved by CENELEC as a European Standard without any modification.

CONTENTS

1 General	9
1.1 Scope	9
1.2 Normative references	9
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	19
1.5.1 System compliance	20
1.5.2 Program compliance	22
2 Common elements	22
2.1 Use of printed characters	22
2.1.1 Character set	22
2.1.2 Identifiers	23
2.1.3 Keywords	23
2.1.4 Use of white space	23
2.1.5 Comments	24
2.1.6 Pragmas	24
2.2 External representation of data	24
2.2.1 Numeric literals	24
2.2.2 Character string literals	25
2.2.3 Time literals	27
2.2.3.1 Duration	27
2.2.3.2 Time of day and date	28
2.3 Data types	28
2.3.1 Elementary data types	29
2.3.2 Generic data types	30
2.3.3 Derived data types	31
2.3.3.1 Declaration	31
2.3.3.2 Initialization	32
2.3.3.3 Usage	34
2.4 Variables	35
2.4.1 Representation	35
2.4.1.1 Single-element variables	35
2.4.1.2 Multi-element variables	37
2.4.2 Initialization	37
2.4.3 Declaration	38
2.4.3.1 Type assignment	40
2.4.3.2 Initial value assignment	41
2.5 Program organization units	44
2.5.1 Functions	44
2.5.1.1 Representation	45
2.5.1.2 Execution control	48
2.5.1.3 Declaration	49
2.5.1.4 Typing, overloading, and type conversion	51
2.5.1.5 Standard functions	54
2.5.1.5.1 Type conversion functions	54
2.5.1.5.2 Numerical functions	55
2.5.1.5.3 Bit string functions	58
2.5.1.5.4 Selection and comparison functions	58
2.5.1.5.5 Character string functions	61
2.5.1.5.6 Functions of time data types	63
2.5.1.5.7 Functions of enumerated data types	63



2.5.2 Function blocks.....	65
2.5.2.1 Representation	66
2.5.2.1a) Use of EN and ENO in function blocks	67
2.5.2.2 Declaration	68
2.5.2.3 Standard function blocks	76
2.5.2.3.1 Bistable elements	76
2.5.2.3.2 Edge detection.....	77
2.5.2.3.3 Counters	77
2.5.2.3.4 Timers.....	80
2.5.2.3.5 Communication function blocks.....	82
2.5.3 Programs	82
2.6 Sequential Function Chart (SFC) elements.....	83
2.6.1 General.....	83
2.6.2 Steps	83
2.6.3 Transitions	85
2.6.4 Actions	88
2.6.4.1 Declaration	88
2.6.4.2 Association with steps	90
2.6.4.3 Action blocks	91
2.6.4.4 Action qualifiers	92
2.6.4.5 Action control.....	93
2.6.5 Rules of evolution	98
2.6.6 Compatibility of SFC elements	106
2.6.7 SFC Compliance requirements	107
2.7 Configuration elements.....	107
2.7.1 Configurations, resources, and access paths	109
2.7.2 Tasks	113
3 Textual languages	122
3.1 Common elements	122
3.2 Instruction list (IL)	122
3.2.1 Instructions	122
3.2.2 Operators, modifiers and operands.....	123
3.2.3 Functions and function blocks.....	125
3.3 Structured Text (ST).....	128
3.3.1 Expressions	128
3.3.2 Statements	130
3.3.2.1 Assignment statements	132
3.3.2.2 Function and function block control statements	132
3.3.2.3 Selection statements	132
3.3.2.4 Iteration statements	133
4 Graphic languages	134
4.1 Common elements	134
4.1.1 Representation of lines and blocks	134
4.1.2 Direction of flow in networks.....	134
4.1.3 Evaluation of networks	135
4.1.4 Execution control elements	137
4.2 Ladder diagram (LD)	138
4.2.1 Power rails.....	138
4.2.2 Link elements and states.....	138
4.2.3 Contacts	139
4.2.4 Coils.....	139
4.2.5 Functions and function blocks.....	139
4.2.6 Order of network evaluation	140
4.3 Function Block Diagram (FBD).....	142
4.3.1 General.....	142
4.3.2 Combination of elements.....	142
4.3.3 Order of network evaluation	142

ANNEX A (normative) Specification method for textual languages	143
A.1 Syntax	143
A.1.1 Terminal symbols	143
A.1.2 Non-terminal symbols	143
A.1.3 Production rules	144
A.2 Semantics	144
ANNEX B (normative) Formal specifications of language elements	145
B.0 Programming model	145
B.1 Common elements	145
B.1.1 Letters, digits and identifiers	145
B.1.2 Constants	146
B.1.2.1 Numeric literals	146
B.1.2.2 Character strings	146
B.1.2.3 Time literals	147
B.1.2.3.1 Duration	147
B.1.2.3.2 Time of day and date	147
B.1.3 Data types	148
B.1.3.1 Elementary data types	148
B.1.3.2 Generic data types	148
B.1.3.3 Derived data types	148
B.1.4 Variables	150
B.1.4.1 Directly represented variables	150
B.1.4.2 Multi-element variables	150
B.1.4.3 Declaration and initialization	151
B.1.5 Program organization units	153
B.1.5.1 Functions	153
B.1.5.2 Function blocks	154
B.1.5.3 Programs	155
B.1.6 Sequential function chart elements	155
B.1.7 Configuration elements	156
B.2 Language IL (Instruction List)	158
B.2.1 Instructions and operands	158
B.2.2 Operators	158
B.3 Language ST (Structured Text)	159
B.3.1 Expressions	159
B.3.2 Statements	159
B.3.2.1 Assignment statements	160
B.3.2.2 Subprogram control statements	160
B.3.2.3 Selection statements	160
B.3.2.4 Iteration statements	160
ANNEX C (normative) Delimiters and keywords	161
ANNEX D (normative) Implementation-dependent parameters	164
ANNEX E (normative) Error conditions	166
ANNEX F (informative) Examples	168
F.1 Function WEIGH	168
F.2 Function block CMD_MONITOR	169
F.3 Function block FWD_REV_MON	172
F.4 Function block STACK_INT	177
F.5 Function block MIX_2_BRIX	182
F.6 Analog signal processing	185
F.6.1 Function block LAG1	186
F.6.2 Function block DELAY	186

F.6.3 Function block AVERAGE	187
F.6.4 Function block INTEGRAL.....	187
F.6.5 Function block DERIVATIVE	188
F.6.6 Function block HYSTERESIS	188
F.6.7 Function block LIMITS_ALARM.....	189
F.6.8 Structure ANALOG_LIMITS.....	189
F.6.9 Function block ANALOG_MONITOR.....	190
F.6.10 Function block PID.....	191
F.6.11 Function block DIFFEQ	192
F.6.12 Function block RAMP	193
F.6.13 Function block TRANSFER.....	194
F.7 Program GRAVEL.....	194
F.8 Program AGV	202
F.9 Use of enumerated data types	205
F.10 Function block RTC (Real Time Clock).....	205
F.11 Function block ALRM_INT.....	205
ANNEX G (informative) Reference character set.....	207
Annex ZA (normative) Normative references to international publications with their corresponding European publications	209
Index	210

Table 1 - Character set features.....	22
Table 2 - Identifier features.....	23
Table 3 - Comment feature.....	24
Table 3a - Pragma feature.....	24
Table 4 - Numeric literals.....	25
Table 5 - Character string literal features	26
Table 6 - Two-character combinations in character strings	27
Table 7 - Duration literal features	28
Table 8 - Date and time of day literals.....	28
Table 9 - Examples of date and time of day literals	28
Table 10 - Elementary data types	29
Table 11 - Hierarchy of generic data types	31
Table 12 - Data type declaration features	32
Table 13 - Default initial values of elementary data types.....	33
Table 14 - Data type initial value declaration features	34
Table 15 - Location and size prefix features for directly represented variables.....	36
Table 16a - Variable declaration keywords	38
Table 16b - Usages of VAR_GLOBAL, VAR_EXTERNAL and CONSTANT declarations	40
Table 17 - Variable type assignment features.....	40
Table 18 - Variable initial value assignment features.....	42
Table 19 - Graphical negation of Boolean signals	46
Table 19a - Textual invocation of functions for formal and non-formal argument list	48
Table 20 - Use of EN input and ENO output	49
Table 20a - Function features	50
Table 21 - Typed and overloaded functions	52
Table 22 - Type conversion function features	54

Table 23 - Standard functions of one numeric variable.....	56
Table 24 - Standard arithmetic functions.....	57
Table 25 - Standard bit shift functions.....	58
Table 26 - Standard bitwise Boolean functions.....	59
Table 27 - Standard selection functions ^d	60
Table 28 - Standard comparison functions.....	61
Table 29 - Standard character string functions.....	62
Table 30 - Functions of time data types.....	63
Table 31 - Functions of enumerated data types.....	65
Table 32 - Examples of function block I/O variable usage.....	67
Table 33 - Function block declaration and usage features.....	70
Table 34 - Standard bistable function blocks ^a	76
Table 35 - Standard edge detection function blocks.....	77
Table 36 - Standard counter function blocks.....	78
Table 37 - Standard timer function blocks.....	80
Table 38 - Standard timer function blocks - timing diagrams.....	81
Table 39 - Program declaration features.....	82
Table 40 - Step features.....	84
Table 41 - Transitions and transition conditions.....	86
Table 42 - Declaration of actions ^{a,b}	89
Table 43 - Step/action association.....	91
Table 44 - Action block features.....	92
Table 45 - Action qualifiers.....	93
Table 45a - Action control features.....	97
Table 46 - Sequence evolution.....	100
Table 47 - Compatible SFC features.....	107
Table 48 - SFC minimal compliance requirements.....	107
Table 49 - Configuration and resource declaration features.....	111
Table 50 - Task features.....	115
Table 51a - Examples of instruction fields.....	123
Table 51b - Parenthesized expression features for IL language.....	124
Table 52 - Instruction List operators.....	124
Table 53 - Function Block invocation and Function invocation features for IL language.....	126
Table 54 - Standard Function Block input operators for IL language.....	128
Table 55 - Operators of the ST language.....	130
Table 56 - ST language statements.....	131
Table 57 - Representation of lines and blocks.....	135
Table 58 - Graphic execution control elements.....	137
Table 59 - Power rails.....	138
Table 60 - Link elements.....	139
Table 61 - Contacts ^a	140
Table 62 - Coils.....	141
Table C.1 - Delimiters.....	161
Table C.2 - Keywords.....	162
Table D.1 - Implementation-dependent parameters.....	164
Table E.1 - Error conditions.....	166

Table G.1 - Character representations	207
Table G.2 - Character encodings	208
Figure 1 - Software model	15
Figure 2 a) - Data flow connection within a program	16
Figure 2 b) - Communication via GLOBAL variables	16
Figure 2 c) - Communication function blocks	17
Figure 2 d) - Communication via access paths	17
Figure 3 - Combination of programmable controller language elements	19
Figure 4 - Examples of function usage	44
Figure 5 - Use of formal argument names	47
Figure 6 - Examples of function declarations and usage	51
Figure 7 - Examples of explicit type conversion with overloaded functions	53
Figure 8 - Examples of explicit type conversion with typed functions	53
Figure 9 - Function block instantiation examples	66
Figure 10 - Examples of function block declarations	69
Figure 11 a) - Graphical use of a function block name as an input variable	72
Figure 11 b) - Graphical use of a function block name as an in-out variable	73
Figure 11 c) - Graphical use of a function block name as an external variable	74
Figure 12 - Declaration and usage of in-out variables in function blocks	75
Figure 14 - ACTION_CONTROL function block - External interface (Not visible to the user)	94
Figure 15 a) - ACTION_CONTROL function block body with "final scan" logic	95
Figure 15 b) - ACTION_CONTROL function block body without "final scan" logic	96
Figure 16 a) - Action control example - SFC representation	97
Figure 16 b) - Action control example - functional equivalent	98
Figure 17 - Examples of SFC evolution rules	104
Figure 18 a) - Examples of SFC errors: an "unsafe" SFC	105
Figure 18 b) - Examples of SFC errors: an "unreachable" SFC	106
Figure 19 a) - Graphical example of a configuration	108
Figure 19 b) - Skeleton function block and program declarations for configuration example	109
Figure 20 - Examples of CONFIGURATION and RESOURCE declaration features	112
Figure 21 a) - Synchronization of function blocks with explicit task associations	119
Figure 21 b) - Synchronization of function blocks with implicit task associations	120
Figure 21 c) - Explicit task associations equivalent to figure 21 b)	121
Figure 22 - EXIT statement example	133
Figure 23 - Feedback path example	136
Figure 24 - Boolean OR examples	142

PROGRAMMABLE CONTROLLERS –

Part 3: Programming languages

1 General

1.1 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.

1.2 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*

1.3 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.

- 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:** 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:** 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:** structured collection of data objects forming a *data type*. (ISO)
- 1.3.6 argument:** synonymous with *input variable*, *output variable* or *in-out variable*.
- 1.3.7 array:** *aggregate* that consists of data objects, with identical attributes, each of which may be uniquely referenced by *subscripting*. (ISO)
- 1.3.8 assignment:** mechanism to give a value to a variable or to an *aggregate*. (ISO)
- 1.3.9 based number:** number represented in a specified base other than ten.
- 1.3.10 bistable function block:** *function block* with two stable states controlled by one or more inputs.
- 1.3.11 bit string:** 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:** language construct for *invoking* the execution of a *function* or *function block*.
- 1.3.14 character string:** *aggregate* that consists of an ordered sequence of characters.
- 1.3.15 comment:** 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:** language element corresponding to a *programmable controller system* as defined in IEC 61131-1.
- 1.3.18 counter function block:** *function block* which accumulates a value for the number of changes sensed at one or more specified *inputs*.
- 1.3.19 data type:** 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 as a single language element.
- 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: character or combination of characters used to separate program *language elements*.

1.3.23 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.

1.3.24 double word: 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 (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 *invocation* can be used in textual languages as an *operand* in an expression.

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

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

1.3.32 generic data type: *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: 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, for example, a point within a module which is contained in a rack, which in turn is contained in a cubicle, etc.

1.3.36 identifier: 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 in-out variable: *variable* that is declared in a VAR_IN_OUT . . . END_VAR block.

1.3.38 initial value: the value assigned to a variable at system start-up.

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

1.3.40 instance: 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.

1.3.41 instance name: *identifier* associated with a specific *instance*.

1.3.42 instantiation: the creation of an *instance*.

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

1.3.44 invocation: the process of initiating the execution of the operations specified in a *program organization unit*.

1.3.45 keyword: lexical unit that characterizes a *language element*, for example, "IF".

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

1.3.47 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.

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

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

1.3.50 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.

1.3.51 long real: real number represented in a *long word*.

1.3.52 long word: 64-bit data element.

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

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

1.3.55 network: arrangement of nodes and interconnecting branches.

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

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

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

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

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

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

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

1.3.63 program (verb): to design, write, and test user programs.

1.3.64 program organization unit: *function*, *function block*, or *program*.

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

1.3.65 real literal: *literal* representing data of type `REAL` or `LREAL`.

1.3.66 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.

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

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

1.3.69 rising edge: the change from 0 to 1 of a Boolean variable.

1.3.70 scope: that portion of a *language element* within which a *declaration* or *label* applies.

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

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

1.3.73 single data element: data element consisting of a single value.

1.3.74 single-element variable: *variable* which represents a *single data element*.

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

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

1.3.77 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.

1.3.78 symbolic representation: the use of *identifiers* to name variables.

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

1.3.80 time literal: *literal* representing data of type `TIME`, `DATE`, `TIME_OF_DAY`, or `DATE_AND_TIME`.

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

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

1.3.83 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.

1.4 Overview and general requirements

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.

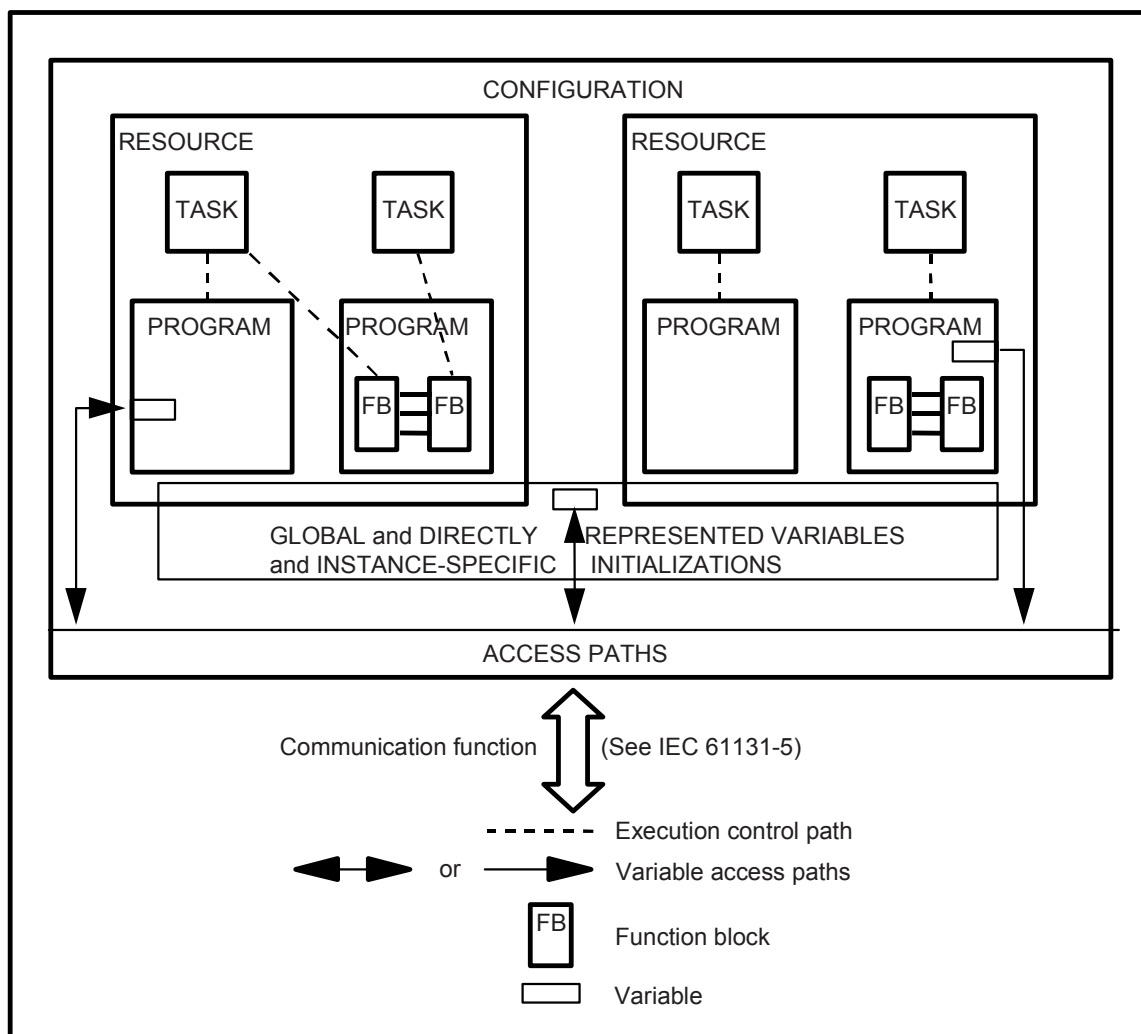
In addition, features are defined which facilitate communication among programmable controllers and other components of automated systems.

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.

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 standard, that is, *programs* and *function blocks*; 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 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 61131-1. 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 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 2.7.1.

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

1.4.2 Communication model

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

As shown in figure 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 2.4.3.

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 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 2 d), using the mechanisms defined in IEC 61131-5.

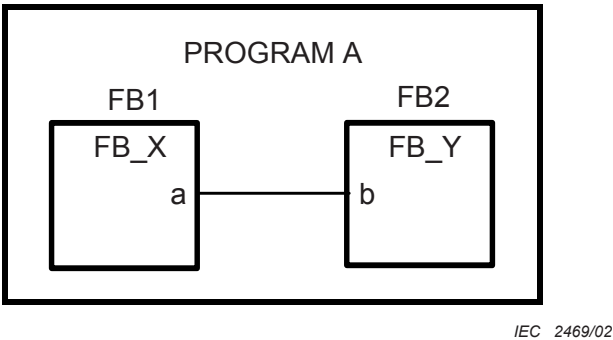


Figure 2 a) - Data flow connection within a program

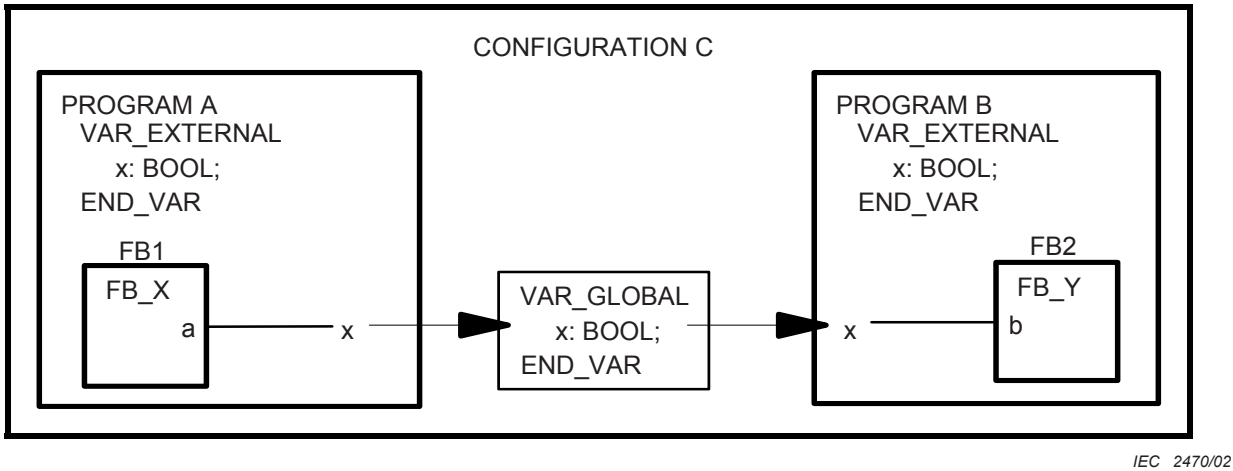
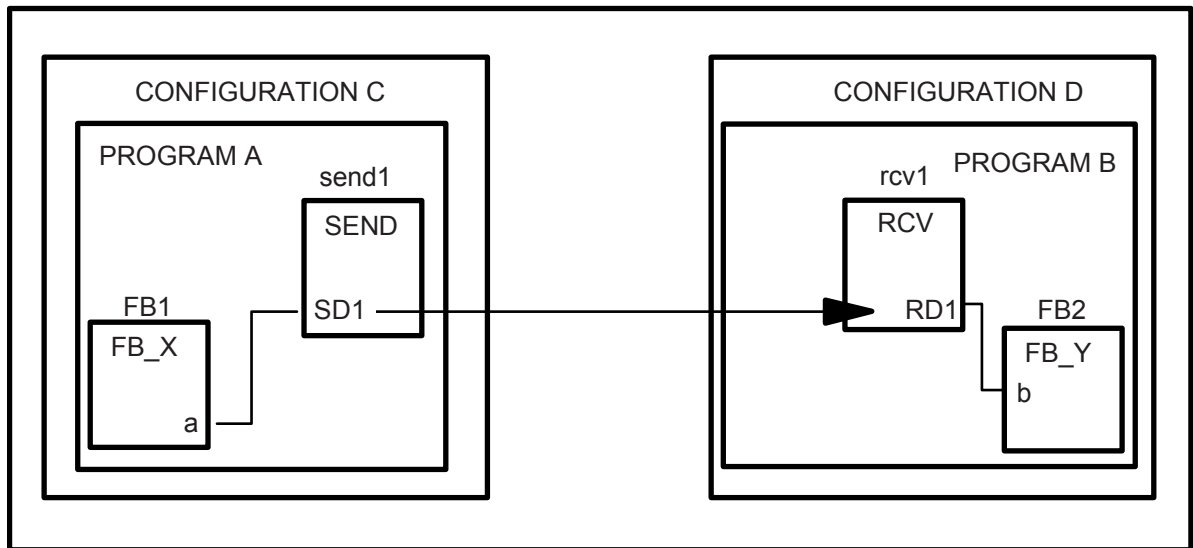
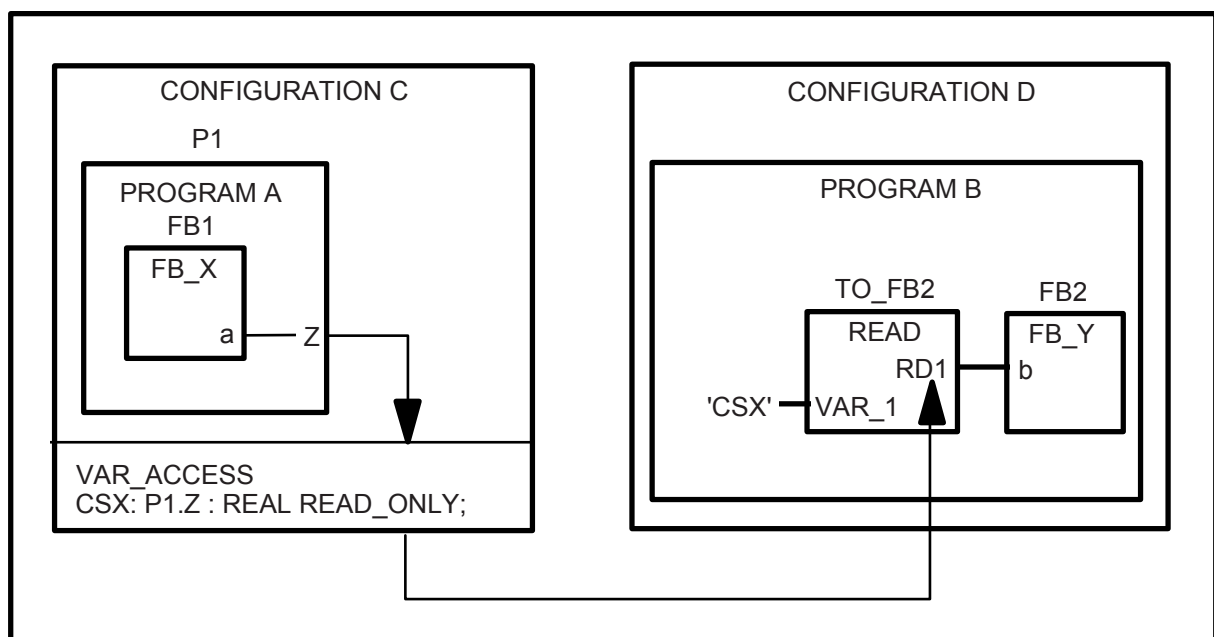


Figure 2 b) - Communication via GLOBAL variables



IEC 2471/02

Figure 2 c) - Communication function blocks



IEC 2472/02

Figure 2 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 2.5.2.3.5 and IEC 61131-5.
- NOTE 4 As specified in 2.7, 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.

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:

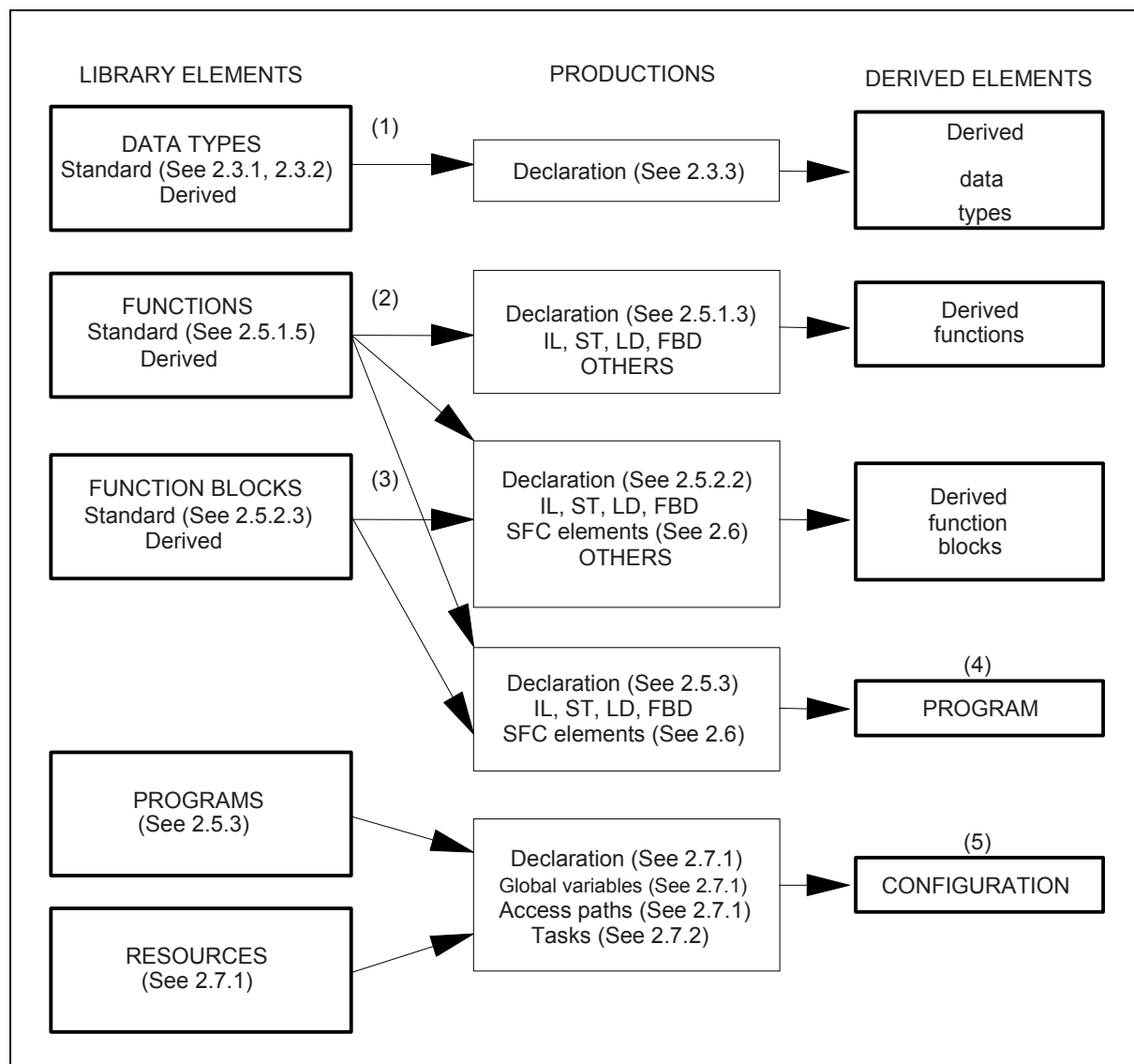
- Data types (2.3)
- Variables (2.4)
- Program organization units (2.5)
 - Functions (2.5.1)
 - Function blocks (2.5.2)
 - Programs (2.5.3)
- Sequential Function Chart (SFC) elements (2.6)
- Configuration elements (2.7)
 - Global variables (2.7.1)
 - Resources (2.7.1)
 - Access paths (2.7.1)
 - Tasks (2.7.2)

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, 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*. 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.



IEC 2473/02

NOTE 1 The parenthesized numbers (1) to (5) refer to the corresponding paragraphs in 1.4.3.

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

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 61131.

1.5.1 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 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	Subclause 4.1
LD language elements	Subclause 4.2
FBD language elements	Subclause 4.3

For the purposes of determining compliance, tables 9, 11, 13, 16a, 16b, 32, 38, 47, 48 and 51 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:

- 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;
- 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 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 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 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 formal syntax 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 B.0, 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.

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 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-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, such 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 61131.

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 “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
2	Lower case characters ^a
3a 3b	Number sign (#) OR Pound sign (£)
4a 4b	Dollar sign (\$) OR Currency sign (¤)
5a 5b	Vertical bar () OR Exclamation mark (!)
NOTE The feature numbering in this table is such as to maintain consistency with the first edition of IEC 61131-3,.	
^a When lower-case letters (feature 2) are supported, the case of letters shall not be significant in language elements except within comments as defined in 2.1.5, string literals as defined in 2.2.2, and variables of type <code>STRING</code> and <code>WSTRING</code> as defined in 2.3.1.	

2.1.2 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-dependent** parameter.

Identifier features and examples are shown in table 2.

Table 2 - Identifier features

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

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 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 1.5.1.

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

2.1.4 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 subclause 2.4.1.1, or delimiter combinations (for example, for comments as defined in 2.1.5). “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.

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. 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 standard.

The use of nested comments, for example, (* (* NESTED *) *) , shall be treated as an **error** according to the provisions of 1.5.1 d).

The maximum number of characters allowed in a comment is an **implementation-dependent** parameter.

Table 3 - Comment feature

No.	Feature description	Example
1	Comments	<pre>(***** (* A framed comment *) *****)</pre>
NOTE The example given above represents three separate comments.		

2.1.6 Pragmas

As illustrated in table 3 a), pragmas shall be delimited at the beginning and end by curly brackets "{ " and "}", respectively. The syntax and semantics of particular pragma constructions are implementation dependent. Pragmas shall be permitted anywhere in the program where spaces are allowed, except within character string literals as defined in 2.2.2.

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 3 a) - Pragma feature

No.	Feature description	Examples
1	Pragmas	<pre>{VERSION 3.1} {AUTHOR JHC} {x := 256, y := 384}</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.

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 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 (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	DINT#5 (DINT representation of 5) UINT#16#9AF (UINT representation of the hexadecimal value 9AF) BOOL#0 BOOL#1 BOOL#TRUE BOOL#FALSE	
NOTE The keywords FALSE and TRUE correspond to Boolean values of 0 and 1, respectively.			

2.2.2 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 5.

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 5.

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 features

No.	Example	Explanation
1	Single-byte character strings	
	' '	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
	'"'	String of length one containing the "double quote" character
	'\$R\$L'	String of length two containing CR and LF characters
	'\$0A'	String of length one containing the LF character
	'\$\$1.00'	String of length five which would print as "\$1.00"
2	Double-byte character strings	
	""	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
	"\$"	String of length one 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 string literals	
	STRING# 'OK'	String of length two containing two single-byte characters
4	Double-byte typed string literals	
	WSTRING# 'OK'	String of length two containing two double-byte characters
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 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
9	\$ "	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.		

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#` 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 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 lower- case letters.

As illustrated in table 7, both positive and negative values are allowed for durations.

Table 7 - Duration literal features

No.	Feature description	Examples
1a	Duration literals without underlines: 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
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 by the syntax given in B.1.2.3.2.

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	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}
17	BYTE	Bit string of length 8	8 ^{i,g}
18	WORD	Bit string of length 16	16 ^{i,g}
19	DWORD	Bit string of length 32	32 ^{i,g}
20	LWORD	Bit string of length 64	64 ^{i,g}
21	WSTRING	Variable-length double-byte character string	16 ^{i,g}

Table 10 - Elementary data types

^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.

2.3.2 Generic data types

In addition to the data types shown in table 10, the hierarchy of generic data types shown in table 11 can be used in the specification of inputs and outputs of standard functions and function blocks (see subclause 2.5.1.4). Generic data types are identified by the prefix “ANY”. The use of generic data types is subject to the following rules:

- 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 *subrange* derived type (feature 3 of table 12) shall be **ANY_INT**.
- 3) The generic type of a *directly 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_DERIVED**.

Table 11 - Hierarchy of generic data types

ANY
ANY_DERIVED (Derived data types - see preceding text)
ANY_ELEMENTARY
ANY_MAGNITUDE
ANY_NUM
ANY_REAL
LREAL
REAL
ANY_INT
LINT, DINT, INT, SINT
ULINT, UDINT, UINT, USINT
TIME
ANY_BIT
LWORD, DWORD, WORD, BYTE, BOOL
ANY_STRING
STRING
WSTRING
ANY_DATE
DATE_AND_TIME
DATE, TIME_OF_DAY

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. 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-dependent** parameter.

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 2.2.1. 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.

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 value of a subrange type falls outside the specified range of values.

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-dependent** parameters.

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. The maximum number of array subscripts, maximum array size and maximum range of subscript values are **implementation-dependent** parameters.

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 table 12, No.2, and table 14, No.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.

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_DATA2` 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` 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. 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. The maximum allowed length of `STRING` and `WSTRING` variables is an **implementation-dependent** parameter.

Table 12 - Data type declaration features

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	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

Table 12 - Data type declaration features

No.	Feature/textual example
5	Structured data types, e.g.: <pre> 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 </pre>
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 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)

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 := 4000); 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, for example variables of the types `RU_REAL` and `FREQ` 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, 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.

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.

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, 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 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 “%” (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 (.).

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* as defined in 2.4.3.1, 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 2.7.1 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.

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 2.5.2, *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	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 49 and vice versa.			

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. In the ST language defined in subclause 3.3, a subscript shall be an expression yielding a value corresponding to one of the sub-types of generic type *ANY_INT* as defined in table 11. The form of subscripts in the IL language defined in 3.2, and the graphic languages defined in clause 4, is restricted to *single-element variables* or *integer literals*.

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 ;
```

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;
```

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 a), 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, 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 61131-1 shall be initialized in an **implementation-dependent** manner.

2.4.3 Declaration

Each declaration of a program organization unit type (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 a), 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 16 a) - Variable declaration keywords

Keyword	Variable usage
<code>VAR</code>	Internal to organization unit
<code>VAR_INPUT</code>	Externally supplied, not modifiable within organization unit
<code>VAR_OUTPUT</code>	Supplied by organization unit to external entities
<code>VAR_IN_OUT</code>	Supplied by external entities - can be modified within organization unit
<code>VAR_EXTERNAL</code>	Supplied by configuration via <code>VAR_GLOBAL</code> (2.7.1) Can be modified within organization unit
<code>VAR_GLOBAL</code>	Global variable declaration (2.7.1)
<code>VAR_ACCESS</code>	Access path declaration (2.7.1)
<code>VAR_TEMP</code>	Temporary storage for variables in function blocks and programs (2.4.3)
<code>VAR_CONFIG</code>	Instance-specific initialization and location assignment.
<code>RETAIN^{b, c, d, e}</code>	Retentive variables (see preceding text)
<code>NON_RETAIN^{b, c, d, e}</code>	Non-retentive variables (see preceding text)
<code>CONSTANT^a</code>	Constant (variable cannot be modified)
<code>AT</code>	Location assignment (2.4.3.1)
NOTE 1 The usage of these keywords is a feature of the program organization unit or configuration element in which they are used. Normative requirements for the use of these keywords are given in 2.4.3.1, 2.4.3.2, 2.5 and 2.7. NOTE 2 Examples of the use of <code>VAR_IN_OUT</code> variables are given in figures 11b and 12.	

Table 16 a) - Variable declaration keywords

- ^a The `CONSTANT` qualifier shall not be used in the declaration of *function block instances* as described in 2.5.2.1.
- ^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 *invocation* of an *instance* of the program organization unit, and do not persist between invocations.

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 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*.

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;
- 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-dependent** parameter.

Table 16 b) – 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

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* 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 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.

The asterisk notation (feature No. 10 in table 15) can be used in address assignments inside programs and function block types to denote not yet fully specified locations for directly represented variables.

Table 17 - Variable type assignment features

No.	Feature/examples	
1 ^a	Declaration of directly represented variables	
	VAR AT %IW6.2 : WORD; AT %MW6 : INT ; END_VAR	16-bit string (note 2) 16-bit integer, initial value = 0
2 ^a	Declaration of directly represented retentive variables	
	VAR RETAIN AT %QW5 : WORD ; END_VAR	At cold restart, will be initialized to a 16-bit string with value 16#0000
3 ^a	Declaration of locations of symbolic variables	
	VAR_GLOBAL LIM_SW_S5 AT %IX27 : BOOL;	Assigns input bit 27 to the Boolean variable LIM_SW_5 (note 2)
	CONV_START AT %QX25 : BOOL;	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

Table 17 - Variable type assignment features

No.	Feature/examples	
4 ^a	Array location assignment	
	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)
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)
NOTE 1 Initialization of system inputs is implementation-dependent ; see 2.4.2.		
NOTE 2 The notes to table 16 a) also apply to this table.		
^a If directly represented variables are explicitly located, 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. 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, as defined in 2.5.2 and 2.5.3, respectively.		

2.4.3.2 Initial value assignment

The VAR...END_VAR construction can be used as shown in table 18 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 2.7.1 (table 49, 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 subclause 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, for example, `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.

When a variable is declared to be a *function block instance*, as defined in 2.5.2.2, initial values for the inputs and any accessible 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 18. Elements for which initial values are not listed shall have the default initial values declared for those elements in the function block declaration.

Table 18 - Variable initial value assignment features

No.	Feature/examples	
1 ^a	Initialization of directly represented variables	
	<pre>VAR AT %QX5.1 : BOOL :=1; AT %MW6 : INT := 8 ; END_VAR</pre>	Boolean type, initial value = 1 Initializes a memory word to integer 8
2 ^a	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 ^a	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 <code>VALVE_POS</code> , with an initial value of 100
4 ^a	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 <code>%QW6</code> , each with an initial value of 1

Table 18 - Variable initial value assignment features

No.	Feature/examples	
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 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	
	<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 := [4((RANGE := UNIPOLAR_1_5V)), (RANGE:= BIPOLAR_10_V, MIN_SCALE := 0, MAX_SCALE := 500)]); END_VAR</pre>	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	
	<pre>VAR CONSTANT PI : REAL := 3.141592 ; END_VAR</pre>	
10	Initialization of function block instances	
	<pre>VAR TempLoop : PID := (PropBand := 2.5, Integral := T#5s); END_VAR</pre>	Allocates initial values to inputs and outputs of a function block instance
^a 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

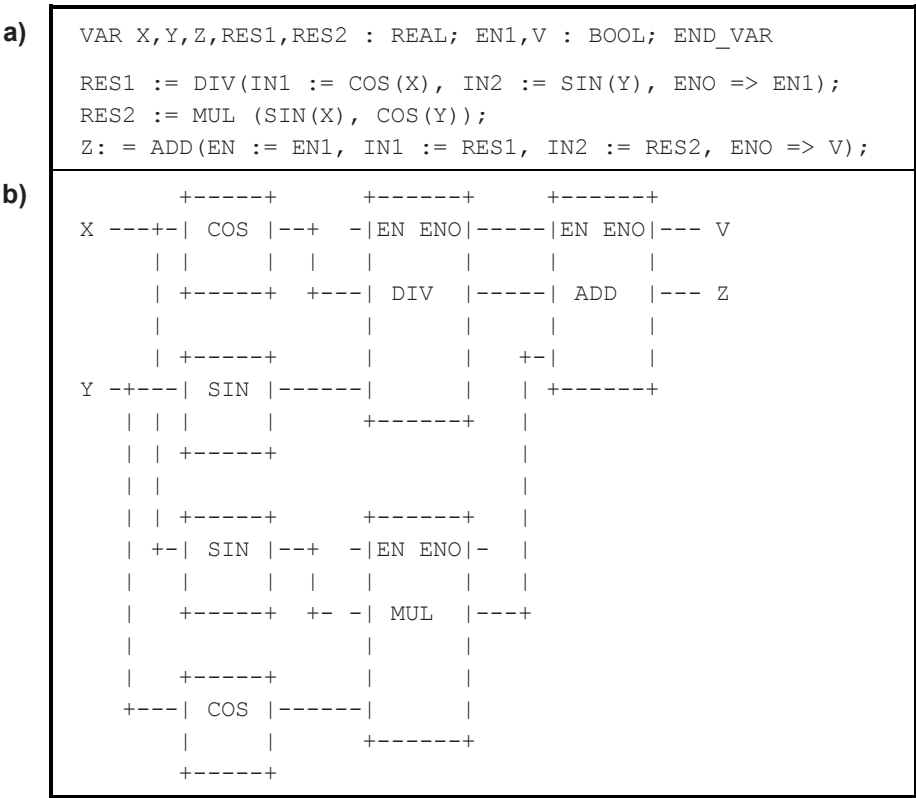
The program organization units defined in this part of IEC 61131 are the *function*, *function block*, and *program*. These program organization units can be delivered by the manufacturer, or programmed by the user by the means defined in this part of the standard.

Program organization units shall not be *recursive*; that is, the invocation of a program organization unit shall not cause the invocation 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-dependent** parameters.

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 is considered to be the function result, and arbitrarily many additional output elements (VAR_OUTPUT and VAR_IN_OUT). As for any data element, the function result can be multi-valued, for example, an array or structure. The invocation of a function 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.



IEC 2474/02

a) Structured Text (ST) language - see subclause 3.3
b) Function Block Diagram (FBD) language - see subclause 4.3
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 4 - Examples of function usage

Functions shall contain no internal state information, i.e., invocation of a function with the same arguments (input variables `VAR_INPUT` and in-out variables `VAR_IN_OUT`) shall always yield the same values (output variables `VAR_OUTPUT`, in-out variables `VAR_IN_OUT` and function result). It shall be an **error** if external variables as defined in 2.4.3 cause the violation of this rule.

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 textual languages defined in clause 3 of this standard, the invocation of functions shall be according to the following rules:

- 1) Input argument assignment shall follow the rules given in table 19 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 (feature 1 of table 19 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 4 of this standard, functions shall be represented as graphic blocks according to the following rules:

- 5) The form of the block shall be rectangular or square.
- 6) The size and proportions of the block may vary depending on the number of inputs and other information to be displayed.
- 7) The direction of processing through the block shall be from left to right (input variables on the left and output variables on the right).
- 8) The function name or symbol, as specified below, shall be located inside the block.
- 9) 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 2.5.1.5, when the given graphical form includes the variable names; or
 - any additional function declared as specified in 2.5.1.3.

This usage is subject to the following provisions:

- a) Where no names are given for input variables in standard functions, the default names `IN1`, `IN2`, . . . shall apply in top-to-bottom order.
- b) When a standard function has a single unnamed input, the default name `IN` shall apply.
- c) The default names described above may, but need not appear at the inside left-hand side of the graphic representation.

- 10) An additional input `EN` and/or output `ENO` as specified in 2.5.1.2 may be used. If present, they shall be shown at the uppermost positions at the left and right side of the block, respectively.
- 11) 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 2.5.1.3, no output variable name shall be shown at the right side of the block.
- 12) Argument connections (including function result) shall be shown by signal flow lines.
- 13) 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 2.1.1, this shall be represented by the upper case alphabetic "O", as shown in table 19.
- 14) 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.
- 15) Function results and function outputs (`VAR_OUTPUT`) can be connected to a variable, used as input to other function blocks or functions, or can be left unconnected.
- 16) It shall be an **error** if any `VAR_IN_OUT` variable of any function block invocation or function invocation 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 invocation, 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 invocation.
- 17) A "properly mapped" (see rule 12 above) `VAR_IN_OUT` variable of a function block instance or a function invocation 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 19 - Graphical negation of Boolean signals

No.	Feature ^{a, b}	Representation
1	Negated input	<pre> +----+ ---O --- +----+</pre>
2	Negated output	<pre> +----+ ---- O--- +----+</pre>
^a 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. ^b The use of these constructs is forbidden for in-out variables.		

Figure 5 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.

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 variable 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 argument names)
<pre> A := SHL(IN := B,N := C); </pre>	Textual use of SHL function (ST language; see 3.3)
<pre> +-----+ SHL ENABLE--- EN ENO O--NO_ERR B--- IN ---A C--- N +-----+ </pre>	Graphical use of SHL function (See 2.5.1.5.3) (FBD language; see 4.3) (Formal argument names; use of EN input and negated ENO output)
<pre> A := SHL(EN:=ENABLE, IN:=B, N:=C, NOT ENO => NO_ERR); </pre>	Textual use of SHL function (ST language; see 3.3)
<pre> +-----+ INC ---A X--- V---V ---X +-----+ </pre>	Graphical use of user-defined INC function (FBD language, see 4.3) (Formal argument names for VAR_IN_OUT)
<pre> A := INC(V := X); </pre>	Textual use of INC function (ST language, see 3.3)

Figure 5 - Use of formal argument names

IEC 2475/02

Features for the textual invocation of functions are defined in table 19 a). The textual invocation of a function shall consist of the function name followed by a list of arguments. In the ST language defined in subclause 3.3, 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 19 a) (formal invocation), the argument list has the form of a set of assignments of actual values to the formal argument names (formal argument list), that is:

- 1) assignments of values to input and in-out variables using the " := " operator, and
- 2) assignments of the values of output variables to variables using the " => " operator.

The ordering of arguments in the list shall be insignificant. In feature 1 of table 19 a), 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 19 a) (non-formal invocation), 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 19 a) - Textual invocation of functions for formal and non-formal argument list

No.	Feature				Example In Structured Text (ST) language - see 3.3
	Invocation type	Variable assignment	Variable order	Number of variables	
1	formal	yes	any	any	<code>A := LIMIT(EN:=COND, IN:=B, MX:=5, ENO=>TEMPL);</code>
2^a	non-formal	no	fixed	fixed	<code>A := LIMIT(1, B, 5);</code>
^a Feature #2 is required for invocation of any of the standard functions defined in subclause 2.5.1.5 without formal names for one or more input variables, but feature #1 shall be used if <code>EN/ENO</code> is necessary in function invocations.					
NOTE 1 In the example given in feature #1, the <code>MN</code> variable will have the default value 0 (zero). NOTE 2 The example given in feature #2 is semantically equivalent to the following invocation with formal variable assignments (feature #1): <code>A := LIMIT(EN := TRUE, MN := 1, IN := B, MX := 5);</code>					

2.5.1.2 Execution control

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 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 any of the errors defined in table E.1 for subclauses of 2.5.1.5 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, or the manufacturer shall specify other disposition of such an **error** according to the provisions of 1.5.1.
- 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**.