



BSI Standards Publication

Programmable controllers

Part 3: Programming languages

National foreword

This British Standard is the UK implementation of EN IEC 61131-3:2025. It is identical to IEC 61131-3:2025. It supersedes BS EN 61131-3:2013, which will be withdrawn on 31 July 2028.

The UK participation in its preparation was entrusted to Technical Committee GEL/65, Measurement and control.

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

Contractual and legal considerations

This publication has been prepared in good faith, however no representation, warranty, assurance or undertaking (express or implied) is or will be made, and no responsibility or liability is or will be accepted by BSI in relation to the adequacy, accuracy, completeness or reasonableness of this publication. All and any such responsibility and liability is expressly disclaimed to the full extent permitted by the law.

This publication is provided as is, and is to be used at the recipient's own risk.

The recipient is advised to consider seeking professional guidance with respect to its use of this publication.

This publication is not intended to constitute a contract. Users are responsible for its correct application.

© The British Standards Institution 2025
Published by BSI Standards Limited 2025

ISBN 978 0 539 17026 9

ICS 25.040.40; 35.060; 35.240.50

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

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 July 2025.

Amendments/corrigenda issued since publication

Date	Text affected
------	---------------

English Version

Programmable controllers - Part 3: Programming languages (IEC 61131-3:2025)

Automates programmables - Partie 3: Langues de
programmation
(IEC 61131-3:2025)

Speicherprogrammierbare Steuerungen - Teil 3:
Programmiersprachen
(IEC 61131-3:2025)

This European Standard was approved by CENELEC on 2025-06-26. CENELEC members are bound to comply with the CEN/CENELEC Internal Regulations which stipulate the conditions for giving this European Standard the status of a national standard without any alteration.

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

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

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



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

CEN-CENELEC Management Centre: Rue de la Science 23, B-1040 Brussels

European foreword

The text of document 65B/1281/FDIS, future edition 4 of IEC 61131-3, prepared by SC 65B "Measurement and control devices" of IEC/TC 65 "Industrial-process measurement, control and automation" was submitted to the IEC-CENELEC parallel vote and approved by CENELEC as EN IEC 61131-3:2025.

The following dates are fixed:

- latest date by which the document has to be implemented at national (dop) 2026-07-31 level by publication of an identical national standard or by endorsement
- latest date by which the national standards conflicting with the (dow) 2028-07-31 document have to be withdrawn

This document supersedes EN 61131-3:2013 and all of its amendments and corrigenda (if any).

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

Any feedback and questions on this document should be directed to the users' national committee. A complete listing of these bodies can be found on the CENELEC website.

Endorsement notice

The text of the International Standard IEC 61131-3:2025 was approved by CENELEC as a European Standard without any modification.

In the official version, for Bibliography, the following note has to be added for the standard indicated:

IEC 60848 NOTE Approved as EN 60848

Annex ZA (normative)

Normative references to international publications with their corresponding European publications

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements 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.

NOTE 1 Where an International Publication has been modified by common modifications, indicated by (mod), the relevant EN/HD applies.

NOTE 2 Up-to-date information on the latest versions of the European Standards listed in this annex is available here: www.cencenelec.eu.

<u>Publication</u>	<u>Year</u>	<u>Title</u>	<u>EN/HD</u>	<u>Year</u>
IEC 61131-1	-	Programmable controllers - Part 1: General information	EN 61131-1	-
IEC 61131-5	-	Programmable controllers - Part 5: Communications	EN 61131-5	-
ISO/IEC 10646	-	Information technology - Universal coded character set (UCS)	-	-
ISO/IEC 60559	-	Information technology - Microprocessor Systems - Floating-Point arithmetic	-	-

CONTENTS

FOREWORD.....	7
1 Scope.....	9
2 Normative references.....	9
3 Terms and definitions.....	9
4 Architectural models.....	17
4.1 Software model.....	17
4.2 Communication model.....	19
4.3 Programming model.....	20
5 Compliance.....	22
5.1 General.....	22
5.2 Feature tables.....	22
5.3 Implementer's compliance statement.....	22
6 Common elements.....	24
6.1 Use of printed characters.....	24
6.1.1 Character set.....	24
6.1.2 Identifiers.....	24
6.1.3 Keywords.....	25
6.1.4 Use of white space.....	25
6.1.5 Comments.....	25
6.2 Pragma.....	26
6.3 Literals – External representation of data.....	26
6.3.1 General.....	26
6.3.2 Numeric literals and string literals.....	26
6.3.3 Character string literals.....	28
6.3.4 Duration literal.....	31
6.3.5 Date and time of day literal.....	31
6.4 Data types.....	32
6.4.1 General.....	32
6.4.2 Elementary data types (BOOL, INT, REAL, STRING, etc.).....	32
6.4.3 Generic data types.....	36
6.4.4 User-defined data types.....	37
6.5 Variables.....	51
6.5.1 Declaration and initialization of variables.....	51
6.5.2 Variable sections.....	54
6.5.3 Variable length ARRAY variables.....	57
6.5.4 Constant variables.....	59
6.5.5 Directly represented variables (%).....	59
6.5.6 Retentive variables (RETAIN, NON_RETAIN).....	62
6.6 Program organization units (POUs).....	63
6.6.1 Common features for POU's.....	63
6.6.2 Functions.....	77
6.6.3 Function blocks.....	113
6.6.4 Programs.....	132
6.6.5 Classes.....	134
6.6.6 Interface.....	155
6.6.7 Object-oriented features for function block types.....	165

6.6.8	Polymorphism.....	171
6.7	Sequential function chart (SFC) elements.....	174
6.7.1	General.....	174
6.7.2	Steps.....	175
6.7.3	Transitions	176
6.7.4	Actions	179
6.7.5	Rules of evolution	187
6.8	Configuration elements	195
6.8.1	General.....	195
6.8.2	Tasks	200
6.9	Synchronization of concurrent execution	205
6.9.1	General.....	205
6.9.2	Mutex	206
6.9.3	Object-oriented mutex	208
6.9.4	Semaphore.....	209
6.9.5	Object-oriented semaphore	211
6.10	Namespaces	211
6.10.1	General.....	211
6.10.2	Declaration.....	212
6.10.3	Usage.....	217
6.10.4	Namespace directive <code>USING</code>	218
7	Textual languages	221
7.1	Common elements.....	221
7.2	Structured text (ST)	221
7.2.1	General.....	221
7.2.2	Expressions	221
7.2.3	Statements.....	223
8	Graphical languages	229
8.1	Common elements.....	229
8.1.1	General.....	229
8.1.2	Representation of variables and instances.....	229
8.1.3	Representation of lines and blocks	231
8.1.4	Direction of flow in networks	232
8.1.5	Evaluation of networks	233
8.1.6	Execution control elements	234
8.2	Ladder diagram (LD).....	235
8.2.1	General.....	235
8.2.2	Power rails	236
8.2.3	Link elements and states	236
8.2.4	Contacts	236
8.2.5	Coils.....	238
8.2.6	Functions and function blocks	239
8.2.7	Order of network evaluation	239
8.3	Function block diagram (FBD).....	240
8.3.1	General.....	240
8.3.2	Combination of elements	240
8.3.3	Order of network evaluation	240
Annex A (normative)	Formal specification of the language elements	241

Annex B (informative) List of major changes and extensions of Edition 4 of IEC 61131-3	249
B.1 General	249
B.2 New features	249
B.3 Significant changes	249
B.4 Deletions	250
B.5 Deprecations	250
Annex C (informative) Relating strings, characters, and their literals to ISO/IEC 10646	251
Bibliography	253
Figure 1 – Software model	18
Figure 2 – Communication model	20
Figure 3 – Combination of programmable controller language elements	21
Figure 4 – Implementer's compliance statement (Example)	23
Figure 5 – Hierarchy of the generic data types	37
Figure 6 – Initialization by literals and constant expressions (Rules)	38
Figure 7 – Variable declaration keywords (Summary)	54
Figure 8 – Usage of <code>VAR_GLOBAL</code> , <code>VAR_EXTERNAL</code> and <code>CONSTANT</code> (Rules)	55
Figure 9 – Conditions for the initial value of a variable (Rules)	62
Figure 10 – Formal and non-formal representation of call (Examples)	68
Figure 11 – Data type conversion rules – implicit and explicit (Summary)	72
Figure 12 – Implicit type conversions overview	73
Figure 13 – Data type conversion from string types to array of bytes	96
Figure 14 – Data type conversion from array of bytes to string types	97
Figure 15 – Usage of function block input and output parameters (Rules)	123
Figure 16 – Usage of function block input and output parameters (Illustration of rules)	124
Figure 17 – Standard timer function blocks – timing diagrams (Rules)	130
Figure 18 – Recommended implementation in pseudo code for timer	132
Figure 19 – Overview of inheritance and interface implementation	135
Figure 20 – Inheritance of classes (Illustration)	144
Figure 21 – Interface with derived classes (Illustration)	157
Figure 22 – Inheritance of interface and class (Illustration)	162
Figure 23 – Function block types with optional body and methods (Illustration)	168
Figure 24 – Inheritance of function block body with <code>SUPER()</code> (Example)	170
Figure 25 – <code>ACTION_CONTROL</code> function block – External interface (Summary)	184
Figure 26 – <code>ACTION_CONTROL</code> function block body (Summary)	185
Figure 27 – Action control (Example)	186
Figure 28 – SFC evolution (Rules)	193
Figure 29 – SFC errors (Example)	194
Figure 30 – Configuration (Example)	196
Figure 31 – <code>CONFIGURATION</code> and <code>RESOURCE</code> declaration (Example)	199
Figure 32 – The reference implementation of mutex functions	208
Figure 33 – The reference implementation of object-oriented mutex	209
Figure 34 – The reference implementation of semaphore functions	210

Figure 35 – The reference implementation of object-oriented semaphore	211
Figure 36 – Accessibility using namespaces (Rules)	214
Figure 37 – Common textual elements (Summary)	221
Figure C.1 – Relation between ISO/IEC 10646 and IEC 61131-3	251
Figure C.2 – Example of encoded strings in memory	252
Table 1 – Character set.....	24
Table 2 – Identifiers.....	24
Table 3 – Comments.....	26
Table 4 – Pragma.....	26
Table 5 – Numeric literals.....	28
Table 6 – Character string literals	29
Table 7 – Two-character combinations in character strings	30
Table 8 – Duration literals	31
Table 9 – Date and time of day literals.....	32
Table 10 – Elementary data types	33
Table 11 – Declaration of user-defined data types and initialization	38
Table 12 – Reference operations	51
Table 13 – Declaration of variables	53
Table 14 – Initialization of variables	53
Table 15 – Variable-length <code>ARRAY</code> variables	57
Table 16 – Directly represented variables	60
Table 17 – Partial access of <code>ANY_BIT</code> variables (except <code>BOOL</code>)	65
Table 18 – Execution control graphically using <code>EN</code> and <code>ENO</code>	70
Table 19 – Function declaration	79
Table 20 – Function call	81
Table 21 – Typed and overloaded functions	84
Table 22 – Data type conversion function.....	86
Table 23 – Data type conversion of numeric data types	88
Table 24 – Data type conversion of bit data types.....	91
Table 25 – Data type conversion of bit and numeric types.....	92
Table 26 – Data type conversion of date and time types	94
Table 27 – Data type conversion of character types	95
Table 28 – Numerical and arithmetic functions.....	98
Table 29 – Arithmetic functions.....	99
Table 30 – Bit shift functions.....	100
Table 31 – Bitwise Boolean functions	100
Table 32 – Selection functions	101
Table 33 – Comparison functions	102
Table 34 – Character string functions	103
Table 35 – Numerical functions of time and duration data types	105
Table 36 – Additional functions of time data types <code>CONCAT</code> and <code>SPLIT</code>	107
Table 37 – Function for endianness conversion	

Table 38 – Functions of enumerated and named values data types.....	111
Table 39 – Validate functions.....	112
Table 40 – Function ASSERT	112
Table 41 – Function block type declaration.....	114
Table 42 – Function block instance declaration.....	118
Table 43 – Function block call.....	120
Table 44 – Standard bistable function blocks ^a	126
Table 45 – Standard edge detection function blocks.....	127
Table 46 – Standard counter function blocks	128
Table 47 – Standard timer function blocks.....	129
Table 48 – Program declaration	133
Table 49 – Class	136
Table 50 – Class instance declaration	138
Table 51 – Textual call of methods – Formal and non-formal parameter list.....	142
Table 52 – Interface.....	156
Table 53 – Assignment attempt.....	164
Table 54 – Object-oriented function block type	166
Table 55 – SFC step.....	176
Table 56 – SFC transition and transition condition	178
Table 57 – SFC declaration of actions	180
Table 58 – Association between step and action.....	181
Table 59 – Action block.....	182
Table 60 – Action qualifiers	182
Table 61 – Action control features	187
Table 62 – Sequence evolution – graphical	188
Table 63 – Configuration and resource declaration	198
Table 64 – Task	201
Table 65 – Mutex type and operations	207
Table 66 – Semaphore type and operations	209
Table 67 – Namespace	216
Table 68 – Nested namespace declaration options.....	217
Table 69 – Namespace directive <code>USING</code>	220
Table 70 – Operators of the ST language.....	223
Table 71 – ST language statements	224
Table 72 – Graphical execution control elements	235
Table 73 – Power rails and link elements	236
Table 74 – Contacts.....	237
Table 75 – Coils	239

INTERNATIONAL ELECTROTECHNICAL COMMISSION

PROGRAMMABLE CONTROLLERS –

Part 3: Programming languages

FOREWORD

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

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

This fourth edition cancels and replaces the third edition published in 2013. This edition constitutes a technical revision.

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

- a) inclusion of UTF-8 strings and their associated functions;
- b) Annex B contains a comprehensive list of features that have been added, removed or deprecated in comparison to IEC 61131-3:2013.

The text of this International Standard is based on the following documents:

Draft	Report on voting
65B/1281/FDIS	65B/1291/RVD

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

The language used for the development of this International Standard is English.

This document was drafted in accordance with ISO/IEC Directives, Part 2, and developed in accordance with ISO/IEC Directives, Part 1 and ISO/IEC Directives, IEC Supplement, available at www.iec.ch/members_experts/refdocs. The main document types developed by IEC are described in greater detail at www.iec.ch/publications.

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

The committee has decided that the contents of this document will remain unchanged until the stability date indicated on the IEC website under webstore.iec.ch in the data related to the specific document. At this date, the document will be

- reconfirmed,
- withdrawn, or
- revised.

PROGRAMMABLE CONTROLLERS –

Part 3: Programming languages

1 Scope

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

This document specifies the syntax and semantics of a unified suite of programming languages for programmable controllers (PCs). This suite consists of the textual language structured text (ST), and the graphical languages, ladder diagram (LD) and function block diagram (FBD).

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

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

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements 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 61131-1, *Programmable controllers – Part 1: General information*

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

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

ISO/IEC 60559, *Floating-point arithmetic*

3 Terms and definitions

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

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

- IEC Electropedia: available at <https://www.electropedia.org/>
- ISO Online browsing platform: available at <https://www.iso.org/obp>

3.1

access path

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

3.2

action

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

3.3

action block

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

3.4

aggregate

structured collection of data objects forming a data type

3.5

array

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

3.6

assignment

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

3.7

base type

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

3.8

based number

number represented in a specified base other than ten

3.9

binary coded decimal

BCD

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

3.10

bistable function block

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

3.11

bit string

data element consisting of one or more bits

3.12

bit string literal

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

3.13

body

set of operations of a program organization unit

3.14

call

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

3.15**character string**

aggregate that consists of an ordered sequence of characters

3.16**character string literal**

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

3.17**class**

program organization unit consisting of:

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

3.18**comment**

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

3.19**configuration**

language element corresponding to one or multiple programmable controller system

3.20**constant**

language element which declares a data element with a fixed value

3.21**counter function block**

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

3.22**data type**

language element which allows a set of values together with a set of permitted operations

3.23**date and time**

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

3.24**declaration**

mechanism for establishing the definition of a language element

3.25**delimiter**

character or combination of characters used to separate program language elements

3.26**derived class****extended class****child class**

class that extends another class by inheritance

3.27**directly derived data type**

data type that is declared as an alias for another data type

3.28

derived function block type

function block type created by inheritance from another function block type or class type

3.29

direct representation

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

3.30

double word

data element containing 32 bits

3.31

dynamic binding

situation in which the implementation of a method is retrieved during runtime according to the actual type of the instance

3.32

evaluation

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

3.33

execution control element

language element which controls the flow of program execution

3.34

falling edge

change from TRUE (1) to FALSE (0) of a Boolean variable

3.35

function

program organization unit that does not retain state and which, when executed, typically yields one result and possibly additional output variables

3.36

function block instance

instance of a function block type

3.37

function block type

language element consisting of:

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

3.38

function block diagram

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

3.39

generic data type

data type which represents more than one type of data

3.40**global variable**

variable whose scope is global

3.41**hierarchical addressing**

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

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

3.42**identifier**

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

3.43**implementation**

product version of a PLC or the programming and debugging tool provided by the implementer

3.44**implementer**

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

3.45**inheritance**

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

3.46**initial value**

value assigned to a variable at the beginning of its lifetime

3.47**in-out variable**

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

3.48**input variable**

variable which is used to supply a value to a program organization unit or method

3.49**instance**

individual, named copy of the data structure associated with a function block type, class, or program type

3.50**instance name**

identifier associated with a specific instance

3.51**instantiation**

creation of an instance

3.52**integer literal**

literal which directly represents an integer value

3.53

interface

language element in the context of object-oriented programming containing a set of method and property prototypes

3.54

keyword

lexical unit reserved for use in a programming language

3.55

label

identifier associated with a network in the LD or FBD language

3.56

language element

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

3.57

literal

lexical unit that directly represents a value

3.58

logical location

location of a hierarchically addressed variable in a schema which can bear some relation to the physical structure of the programmable controller's inputs, outputs, and memory

3.59

method

language element of a function block type or class type, which is equal to a function except that it is called on an instance and its operations can access this instance

3.60

method prototype

language element consisting only of the name and the signature of a method

3.61

named element

element of a structure which is named by its associated identifier

3.62

network

arrangement of nodes and interconnecting branches

3.63

numeric literal

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

3.64

operation

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

3.65

operand

language element on which an operation is performed

3.66**operator**

symbol that represents the action to be performed in an operation

3.67**override**

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

3.68**output variable**

variable which is used to return a value from a function, function block, program or method

3.69**parameter**

variable which is used to provide a value to a function, function block, program or method (as input or in-out parameter) or a variable which is used to return a value from a function, function block, program or method (as output or in-out parameter)

3.70**reference**

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

3.71**power flow**

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

3.72**pragma**

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

3.73**program**

language construct to design, write, and test user programs

3.74**program organization unit**

function, function block, class, or program

3.75**property**

language construct that typically can be used like a member variable and is defined by special methods to get or set its value

3.76**property prototype**

language construct consisting only of the name and the type of a property

3.77**real literal**

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

3.78

resource

language element corresponding to a signal processing function and its human-machine interface as well as its sensor and actuator interface functions

3.79

result

value which is returned as an outcome of a function or method

3.80

return

language construct within a program organization unit designating an end to the execution sequences in the unit

3.81

rising edge

change from `FALSE` (0) to `TRUE` (1) of a Boolean variable

3.82

scope

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

3.83

semantics

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

3.84

semigraphic representation

representation of graphical information by the use of a limited set of characters

3.85

signature

set of information defining a `FUNCTION`, `FUNCTION BLOCK`, or `METHOD` interface, consisting of an ordered list of parameters and the return type (if any), where each parameter is defined by its name, its type, and its variable-kind (input, output or in-out)

3.86

single-element variable

variable which represents a single data element

3.87

static variable

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

3.88

step

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

3.89

structured data type

aggregate data type which has been declared using a `STRUCT` declaration

3.90

subscripting

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

3.91**task**

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

3.92**time literal**

literal representing data of type `TIME`, `LTIME`, `DATE`, `LDATE`, `TIME_OF_DAY`, `LTIME_OF_DAY`, `DATE_AND_TIME` or `LDATE_AND_TIME`

3.93**transition**

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

3.94**unsigned integer**

integer which can only have a value greater than or equal to zero

3.95**unsigned integer literal**

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

3.96**user-defined data type**

data type defined by the user

EXAMPLE Enumeration, array or structure.

3.97**variable**

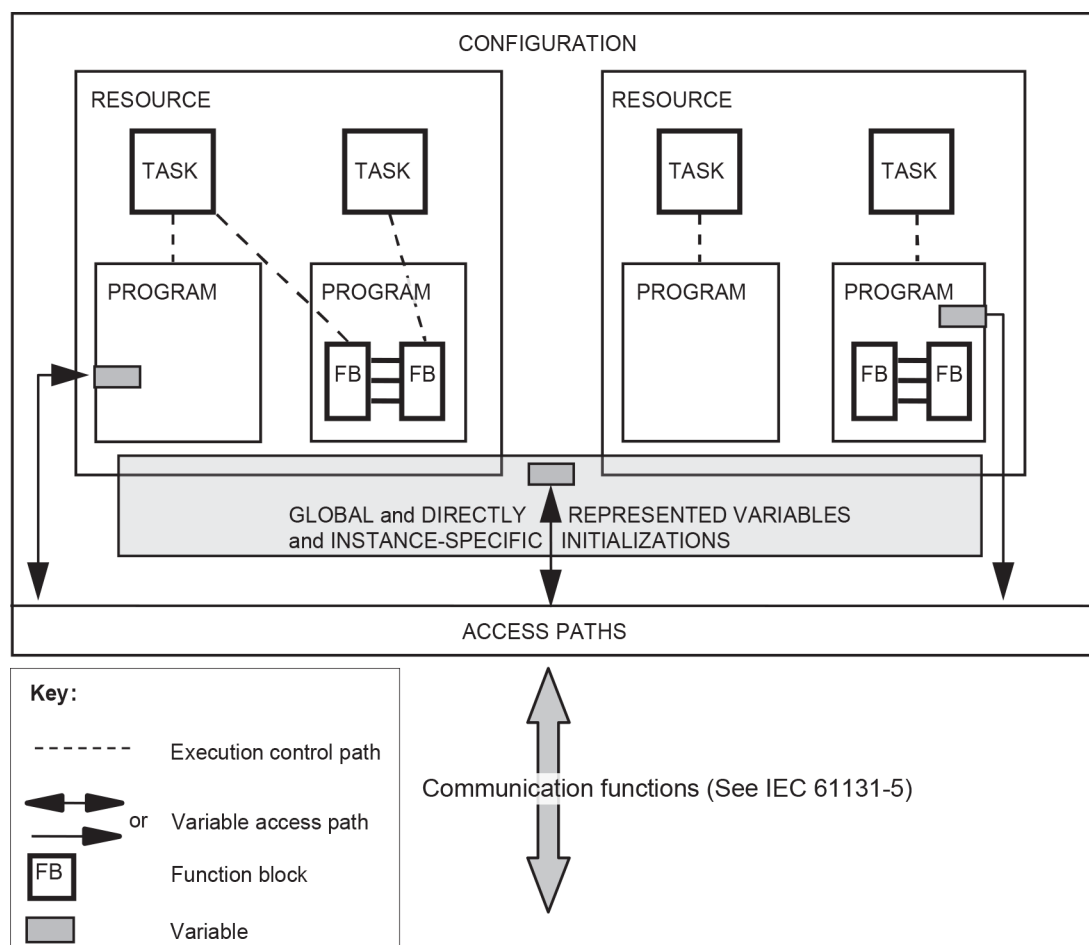
software entity that can take different values, one at a time

4 Architectural models

4.1 Software model

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

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



IEC

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

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

NOTE 3 The use of access paths to make data available for transfer is deprecated.

NOTE 4 The association of tasks directly to function block instances is deprecated.

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 human-machine interface (HMI) 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 can contain zero or more function block instances or other language elements as defined in this document.

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

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

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

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

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

4.2 Communication model

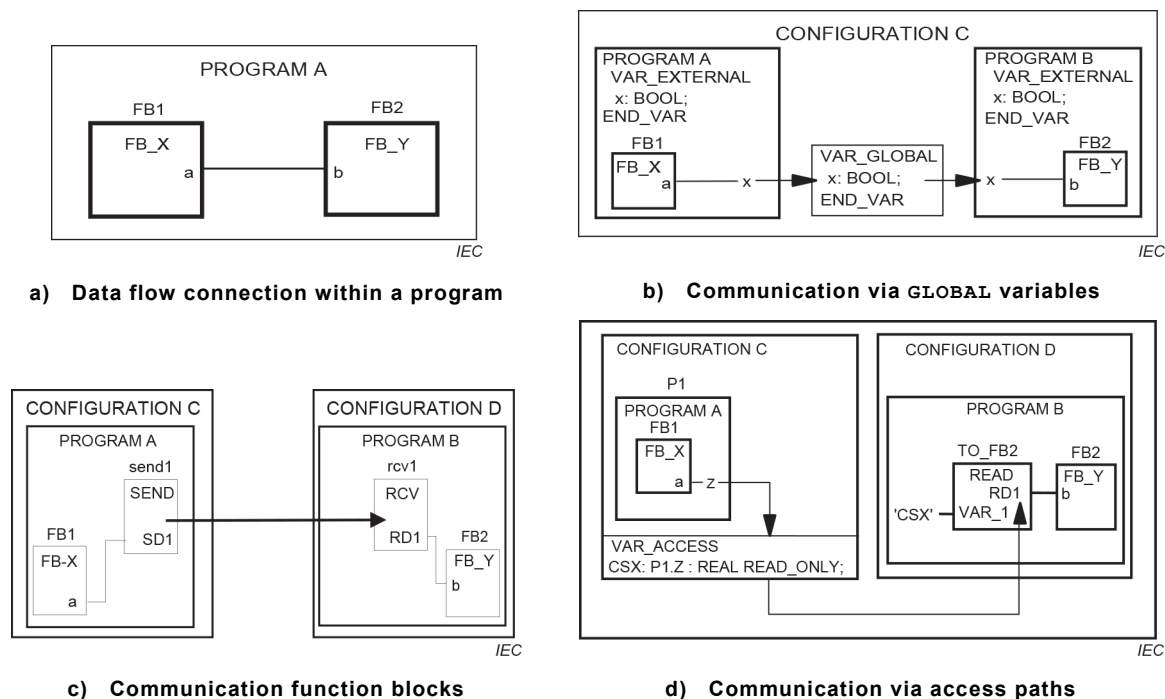
Figure 2 illustrates the ways that values of variables can be communicated among software elements.

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

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

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

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



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

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

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

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

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

NOTE 6 The feature 9 VAR_ACCESS in Table 48 is deprecated and will not be included in the next edition of IEC 61131-3.

Figure 2 – Communication model

4.3 Programming model

Figure 3 illustrates the essential relationships between the most common PLC language elements. The combination of these elements shall obey the following rules:

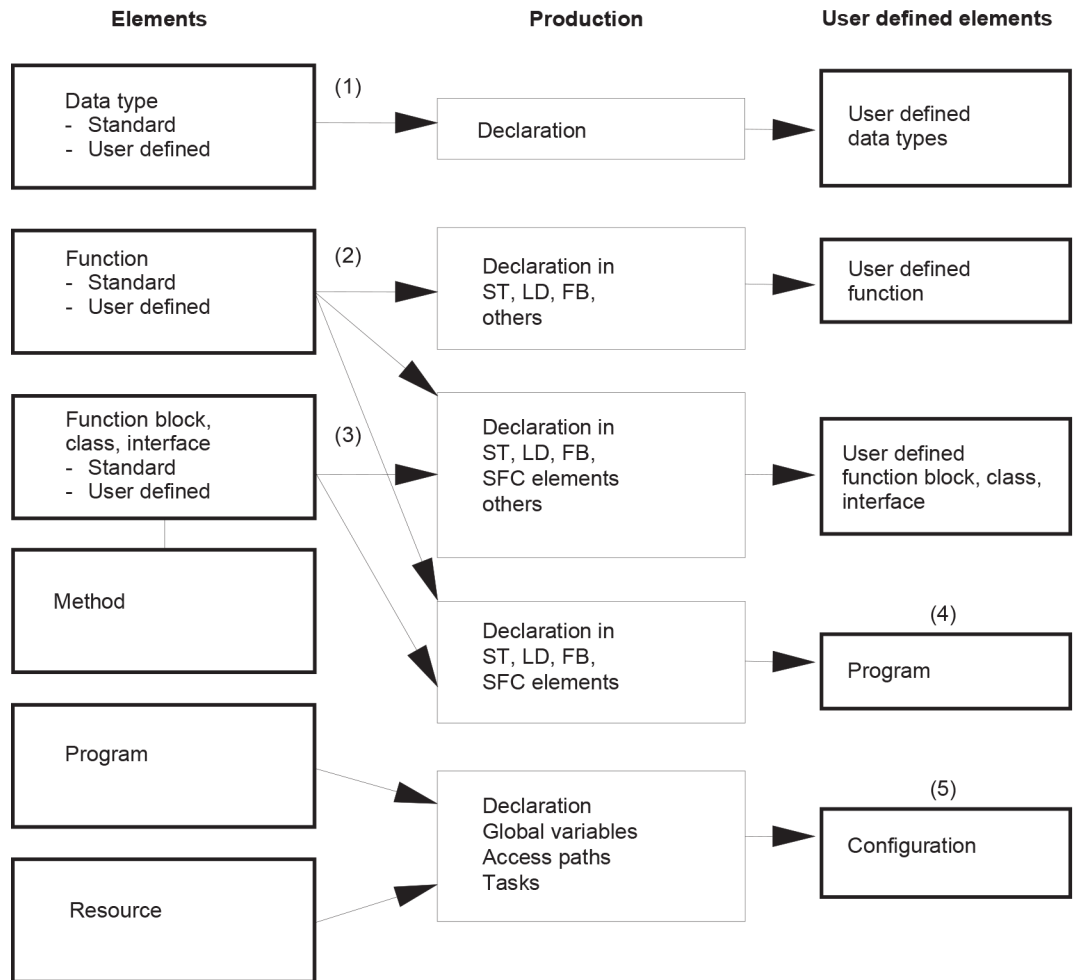
- 1) Data types shall be declared using standard or user-defined data types.
- 2) Functions can be declared using standard or user-defined data types and functions.
 This declaration shall use the mechanisms defined for the ST, LD or FBD language.
- 3) Function block types can be declared using standard or user-defined data types, functions, standard function block types and any previously defined function block types.
 These declarations shall use the mechanisms defined for the ST, LD, or FBD language, and can include sequential function chart (SFC) elements.
 Object-oriented function block types or classes can be declared and can contain methods. Object-oriented function block types and classes can also be derived and can implement interfaces (not shown).
- 4) A program shall be declared using standard or user-defined data types, functions, function blocks and classes.
 This declaration shall use the mechanisms defined for the ST, LD, or FBD language, and can include sequential function chart (SFC) elements.

- 5) Programs can be combined into configurations using the configuration elements, that is, global variables, resources, tasks, and access paths.

A programming language other than one of those defined in this document may be used for programming of a function, function block type, program or method.

It shall be an error if two declared elements have the same fully qualified name (see 6.9). An element can be a data type, function, function block, class, global variable or a namespace.

NOTE The fully qualified name is equal to the name if namespaces are not supported.



IEC

LD: ladder diagram

FBD: function block diagram

ST: structured text

others: other programming languages

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

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

Figure 3 – Combination of programmable controller language elements

5 Compliance

5.1 General

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

- a) It shall provide a subset of the features and provide the corresponding “Implementer’s compliance statement” as defined below.
- b) It shall not require the inclusion of substitute or additional language elements to accomplish any of the features.
- c) It shall provide a document that specifies all implementer specific extensions. These are any features accepted by the system that are prohibited or not specified.
- d) It shall provide a document that specifies all implementer specific dependencies. This includes the implementation dependencies explicitly designated in this document and the limiting parameters like maximum length, number, size, and range of value which are not explicitly specified here.
- e) It shall provide a document that specifies all errors that are detectable and reported by the implementation. This includes the errors explicitly designated in this part and the errors detectable during preparation of the program for execution and during execution of the program.

NOTE Errors occurring during execution of the program are only partially specified in this document.

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

5.2 Feature tables

The tables in this document list the features defined in this document and share the same basic structure. The first column contains the “feature number”, the second column gives the “feature description”, the following columns, if present, contain examples or further information. The same table structure should be used in the implementer’s compliance statement.

5.3 Implementer’s compliance statement

The implementer may define any consistent subset of the features listed in the feature tables and shall declare the provided subset in the “Implementer’s compliance statement”.

The “Implementer’s compliance statement” shall be included in the documentation accompanying the system or shall be produced by the system itself.

The “Implementer’s compliance statement” shall provide the following information. Figure 4 shows an example.

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

Optionally, the title and subtitle of the feature table, the feature description, examples, implementer’s note and other relevant information may be provided.

Not implemented tables and features may be omitted from the “Implementer’s compliance statement”.

IEC 61131-3 “PLC Programming Languages”						
Implementer: Company name, address, etc.						
Product: Product name, version, etc. Controller type specific subset, etc.						
Date: 2022-05-01						
This Product complies with the requirements of the standard for the following language features:						
Feature No.	Table number and title / Feature description	Compliantly implemented in the language (✓)				Implementer's note
		LD	FB D	ST		
	Table 1 – Character set					
1	ISO/IEC 10646, Information technology – Universal Coded Character Set (UCS)	✓	✓	✓		
2a	Lower case characters a: a, b, c, ...	✓	✓	✓		No “ß, ü, ä, ö”
2b	Number sign: # See Table 5	✓				
2c	Dollar sign: \$ See Table 6		✓			
	Table 2 – Identifiers					
1	Upper case letters and numbers: IW215					
2	Upper and lower case letters, numbers, embedded underscore					
3	Upper and lower case, numbers, leading or embedded underscore					
	Table 3 – Comments					
1	Single-line comment //...					
2a	Multi-line comment (* ... *)					
2b	Multi-line comment /* ... */					
3a	Nested comment (* ..(* .. *) ..*)					
3b	Nested comment /* .. /* .. */ .. */					
	Table 4 – Pragma					
1	Pragma with curly brackets { ... }					
	Table 5 – Numeric literals					
1	Integer literal: -12					
2	Real literal: -12.0					
3	Real literals with exponent: -1.34E-12					
4	Binary literal: 2#1111_1111					
5	Hexadecimal literal: 16#FF					
6	Boolean zero and one					
7	Boolean FALSE and TRUE					
8	Typed literal: INT#-123					
	Etc.					

Figure 4 – Implementer's compliance statement (Example)

6 Common elements

6.1 Use of printed characters

6.1.1 Character set

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

Table 1 – Character set

No.	Description
1	ISO/IEC 10646
2a	Lower case characters ^a : a, b, c
2b	Number sign: # See Table 5
2c	Dollar sign: \$ See Table 6
^a When lower-case letters are supported, the case of letters shall not be significant in language elements except within comments as defined in 6.1.5, string literals as defined in 6.3.3, and variables of type <code>STRING</code> , <code>WSTRING</code> and <code>USTRING</code> as defined in 6.3.3.	

6.1.2 Identifiers

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

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

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

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

Identifier features and examples are shown in Table 2.

Table 2 – Identifiers

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

6.1.3 Keywords

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

6.1.4 Use of white space

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

6.1.5 Comments

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

- 1) Single line comments start with the character combination `//` and end at the next following line feed, new line, form feed (page), or carriage return.
- 2) In single-line comments the character combinations `( * and * )` or `/* and */` have no special meaning.
- 3) Multi-line comments shall be delimited at the beginning and end by the special character combinations `( * and * )`, respectively.

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

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

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

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

Nested comments use corresponding

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

Table 3 – Comments

No.	Description	Examples
1	Single-line comment with // ...	X:= 13; // comment for one line // a single line comments can start at // the first character position.
2a	Multi-line comment with (* ... *)	(* comment *) (* ***** A framed comment on three line *****)
2b	Multi-line comment with /* ... */	/* comment in one or more lines */
3a	Nested comment with (* .. (* .. *) .. *)	(* (* NESTED *) *)
3b	Nested comment with /* .. /* .. */ .. */	/* /* NESTED */ /*

6.2 Pragma

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

Table 4 – Pragma

No.	Description	Examples
1	Pragma with { ... } curly brackets	{VERSION 2.0} {AUTHOR JHC} {x:= 256, y:= 384}

6.3 Literals – External representation of data

6.3.1 General

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

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

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

6.3.2 Numeric literals and string literals

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

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

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

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

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

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

Numeric literal features and examples are shown in Table 5.

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

Table 5 – Numeric literals

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

6.3.3 Character string literals

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

- A single-byte character string literal is a sequence of zero or more characters from the Basic Latin and Latin-1 Supplement (code point 0 to 255) of the ISO/IEC 10646 character set. It is 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 1g of Table 6.

- A double-byte character string literal is a sequence of zero or more characters from the Basic Multilingual Plane (code point 0 to 65535) of the ISO/IEC 10646 character set. It is 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 2h of Table 6.
- A Unicode character string literal is a sequence of zero or more characters from the entire ISO/IEC 10646 character set. It is prefixed by the characters U# and the single quote character (') and terminated by the single quote character ('). Additionally, the prefix characters USTRING# can be used in place of U# by the implementer. In these literals, the seven-character combination of the dollar sign "\$" followed by six hexadecimal digits shall be interpreted as the hexadecimal representation of the 24-bit character code, as shown in feature 5e of Table 6.

NOTE 1 Relation of ISO/IEC 10646 and Unicode:

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

In any string literal the character combinations starting with '\${' followed by hexadecimal digits and '}' shall be interpreted as the hexadecimal representation of the character code as shown in feature 6 of Table 6. It is an error if the escaped character is not representable by the string literal's type.

Table 6 – Character string literals

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

No.	Description	Examples
4d	Typed double-byte character (using single quote character)	WCHAR# 'X'
String literals supporting entire Unicode character set		
5a	Unicode character (using U# prefix and single quotes)	U# 'X'
5b	Unicode string (using U# prefix and single quotes)	U# 'OK'
5c	Unicode character (using UCHAR# prefix and single quotes)	UCHAR# 'X'
5d	Unicode string (using USTRING# prefix and single quotes)	USTRING# 'OK'
5e	Support of a character representation with '\$' and six hexadecimal digits in a literal according to 5a to 5d (see NOTE 2)	U# 'OK\$01F600'
6	Support of a character representation with '\$' and hexadecimal digits between '{' and '}'	'\${9}' → '\$t' "\${100}" → "Ä" U# '\${1F600}' → '👉'

NOTE 1 If a particular implementation supports feature 4 but not feature 2, the implementer can specify implementer specific syntax and semantics for the use of the double-quote character.

NOTE 2 Any valid ISO/IEC 10646 code point can be represented using 6 hexadecimal digits, although its UTF-8 encoding can occupy up to 4 bytes (see Annex C).

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

Table 7 – Two-character combinations in character strings

No.	Description	Combinations
1	Dollar sign	\$\$
2	Single quote	\$'
3	Line feed	\$L or \$l
4	Newline	\$N or \$n
5	Form feed (page)	\$P or \$p
6	Carriage return	\$R or \$r
7	Tabulator	\$T or \$t
8	Double quote	\$"

NOTE 1 The newline character provides an implementation-independent means of defining the end of a line of data; for printing, the effect is that of ending a line of data and resuming printing at the beginning of the next line.

NOTE 2 The '\$' combination is only valid inside single quoted string literals.

NOTE 3 The '\$' combination is only valid inside double quoted string literals.

NOTE 2 An implementer can specify additional implementer specific literals (e.g. additional UTF-8 literals) that yield single-byte character strings that can be assigned to `STRING` variables. A direct relation between a character of the implementer specific literal and a character of the single-byte-character string is not guaranteed.

EXAMPLE Assigning an implementer specific literal with UTF-8 encoding to a `STRING` variable. The variables `name1` and `name2` are equal.

```
VAR
    name1: STRING:= UTF8#'ノウサギ';
    name2: STRING:= '$E3$83$8E$E3$82$A6$E3$82$B5$E3$82$AE';
END_VAR
```

6.3.4 Duration literal

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

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

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

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

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

Table 8 – Duration literals

No.	Description	Examples
Duration abbreviations		
1a	d	Day
1b	h	Hour
1c	m	Minute
1d	s	Second
1e	ms	Millisecond
1f	us (no μ available)	Microsecond
1g	ns	Nanoseconds
Duration literals without underscore		
2a	short prefix	T#14ms T#-14ms LT#14.7s T#14.7m T#14.7h t#14.7d t#25h15m lt#5d14h12m18s3.5ms t#12h4m34ms230us400ns
2b	long prefix	TIME#14ms TIME#-14ms time#14.7s
Duration literals with underscore		
3a	short prefix	t#25h_15m t#5d_14h_12m_18s_3.5ms LTIME#5m_30s_500ms_100.1us
3b	long prefix	TIME#25h_15m ltime#5d_14h_12m_18s_3.5ms LTIME#34s_345ns

6.3.5 Date and time of day literal

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

Table 9 – Date and time of day literals

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

6.4 Data types

6.4.1 General

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

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

6.4.2.1 Specification of elementary data types

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

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

Table 10 – Elementary data types

No.	Description	Keyword	Default initial value	N (bits) ^a
1	Boolean	BOOL	0, FALSE	1 ^h
2	Short integer	SINT	0	8 ^c
3	Integer	INT	0	16 ^c
4	Double integer	DINT	0	32 ^c
5	Long integer	LINT	0	64 ^c
6	Unsigned short integer	USINT	0	8 ^d
7	Unsigned integer	UINT	0	16 ^d
8	Unsigned double integer	UDINT	0	32 ^d
9	Unsigned long integer	ULINT	0	64 ^d
10	Real numbers	REAL	0.0	32 ^e
11	Long reals	LREAL	0.0	64 ^f
12a	Duration	TIME	T#0s	-- ^b
12b	Long duration	LTIME	LTIME#0s	64 ^{m, q, s}
13a	Date (only)	DATE		-- ^{b, r}
13b	Long date (only)	LDATE	LDATE#1970-01-01	64 ^{n, s}
14a	Time of day (only)	TIME_OF_DAY or TOD	TOD#00:00:00	-- ^b
14b	Long time of day (only)	LTIME_OF_DAY or LTOD	LTOD#00:00:00	64 ^{p, q, s}
15a	Date and time of day	DATE_AND_TIME or DT		-- ^{b, r}
15b	Long date and time of day	LDATE_AND_TIME or LDT	LDT#1970-01-01-00:00:00	64 ^{o, q, s}
16a	Variable-length single-byte character string	STRING	' ' (empty)	8 ^{i, g, k, l, u}
16b	Variable-length double-byte character string	WSTRING	" " (empty)	16 ^{i, g, k, l, u}
16c	Variable-length UTF-8 encoded character string	USTRING	U# ' ' (empty)	-- ^{g, k, l, t}
17a	Single-byte character	CHAR	'\$00'	8 ^{g, l, u}
17b	Double-byte character	WCHAR	"\$0000"	16 ^{g, l, u}
17c	Four-byte character	UCHAR	U# '\$000000'	32 ^{g, l, u}
18	Bit string of length 8	BYTE	16#00	8 ^{j, g}
19	Bit string of length 16	WORD	16#0000	16 ^{j, g}
20	Bit string of length 32	DWORD	16#0000_0000	32 ^{j, g}
21	Bit string of length 64	LWORD	16#0000_0000_0000_0000	64 ^{j, g}

NOTE A single character occupies 1 to 4 code-units, depending on the character's code-point. Code-points in the range 16#00–16#7F are represented by a single code-unit. Code-points in the range 16#80–16#7FF are represented by two code-units. Code-points in the range 16#800–16#FFFF are represented by three code-units and code-points in the range 16#10000–16#10FFFF are represented by four code-units. A code-unit has a size of one byte.

a	Entries in this column shall be interpreted as specified in the table footnotes.
b	The range of values and precision of representation in these data types is implementer specific.
c	The range of values for variables of this data type is from $-(2^{N-1})$ to $(2^{N-1}) - 1$.
d	The range of values for variables of this data type is from 0 to $(2^N) - 1$.
e	The range of values for variables of this data type shall be as defined in ISO/IEC 60559 for the basic single width floating-point format. Results of arithmetic instructions with denormalized values, infinity, or not-a-number values are implementer specific.
f	The range of values for variables of this data type shall be as defined in ISO/IEC 60559 for the basic double width floating-point format. Results of arithmetic instructions with denormalized values, infinity, or not-a-number values are implementer specific.
g	A numeric range of values does not apply to this data type.
h	The possible values of variables of this data type shall be 0 and 1, corresponding to the keywords <code>FALSE</code> and <code>TRUE</code> , respectively.
i	The value of N indicates the number of bits per character for this data type.
j	The value of N indicates the number of bits in the bit string for this data type.
k	The maximum allowed length of <code>STRING</code> , <code>WSTRING</code> and <code>USTRING</code> variables is implementer specific.
l	For <code>CHAR</code> , <code>STRING</code> , <code>WCHAR</code> , <code>WSTRING</code> , <code>UCHAR</code> , and <code>USTRING</code> , characters are assigned code-points (character encoding) according to ISO/IEC 10646 (see 6.4.2.2 and Annex C).
m	The data type <code>LTIME</code> is a signed 64-bit integer with unit of nanoseconds.
n	The data type <code>LDATE</code> is a signed 64-bit integer with unit of nanoseconds with starting time at midnight 1970-01-01.
o	The data type <code>LDT</code> is a signed 64-bit integer with unit of nanoseconds with starting time at midnight 1970-01-01.
p	The data type <code>LTOD</code> is a signed 64-bit integer with unit of nanoseconds with starting time midnight with <code>TOD#00:00:00</code> .
q	The update accuracy of the values of this time format is implementer specific, i.e. the value is given in nanoseconds, but it may be updated every microsecond or millisecond.
r	The default initial value of this data type is implementer specific.
s	Long time formats do not count the leap seconds i.e. a day always consists of 86 400 s, a non-leap year consists of 365 days, a leap year consists of 366 days. Gregorian calendar is used.
t	Each character in a <code>USTRING</code> is binary represented according to UTF-8 as defined in ISO/IEC 10646 (see NOTE and Annex C).
u	The code-point of characters is binary represented as unsigned integer (see Annex C).

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

The definition of string datatypes relies on ISO/IEC 10646. A basic overview of ISO/IEC 10646 is given in Annex C.

The supported maximum length of values of type `STRING`, `WSTRING` and `USTRING` shall be implementer specific.

The explicit maximum length `n` is specified in square brackets “[`n`]” in the associated declaration. The explicit maximum length `n` shall not exceed the implementer specific supported maximum value.

For `STRING` and `WSTRING` the explicit maximum length is the number of characters the string can contain. For `USTRING` the explicit maximum length is the number of code-units the string can contain.

NOTE 1 For instance, this is the number of ‘a’ characters that can be represented by any of the three string types.

Access to single characters of a string using elements of the data type CHAR, WCHAR or UCHAR shall be supported using square brackets and the position of the character in the string, starting with position 1. Let U be a variable of type USTRING[n] that contains the sequence of characters $C_1, C_2, \dots, C_i, \dots, C_m$. The assignment $U[i] := X$ denotes the replacement of C_i by X . If the resulting character sequence does not fit in U , a minimal number of characters at the end of the string is truncated.

NOTE 2 For USTRING this means that if the number of code-units of C_i does not match that of X , code-units of the following characters are shifted accordingly.

EXAMPLE 1 STRING, WSTRING, USTRING and CHAR, WCHAR, UCHAR

a) Declaration

```
VAR
  String1:   STRING[10] := 'ABCD';
  String2:   STRING[10] := '';
  aWStrings: ARRAY [0..1] OF WSTRING := ["1234", "5678"];
  Ustring1:  USTRING[10] := U#'A+Ä-☹'; //5 UCHAR with 9 code-units:'A+Ä-☹'
  Char1:     CHAR;
  Wchar1:    WCHAR;
  Uchar1:    UCHAR := U#'$01F600';      // Unicode character '👤'
END_VAR
```

b) Usage of STRING and CHAR

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

c) Usage of WSTRING and WCHAR

```
Wchar1:= aWStrings[1][2]; //is equivalent to Wchar1:= "6";
aWStrings[1][3]:= Wchar1; //results in aWStrings[1]:= "5668"
aWStrings[1][4]:= "6";    //results in aWStrings[1]:= "5666"
aWStrings[1][1]:= aWStrings[1][4]; //results in String1:= "6666"
aWStrings[0]:= aWStrings[1][4]; (*results in aWStrings[0]:= "6";
                                if implicit conversion WCHAR_TO_WSTRING
                                has been implemented *)
```

d) Usage of USTRING and UCHAR

```
Uchar1:= Ustring1[4];    //results in Uchar1 = U#'-`
Ustring1[2]:= Uchar1;    //results in Ustring1 = U#'A-Ä-☹'
Ustring1[5]:= U#'$01F600'; //results in Ustring1 = U#'A-Ä-☹'
Ustring1[5]:= USTRING#'$01F600'; //equivalent to previous line
Ustring1[1]:= USTRING#'$01F68D'; (* results in Ustring1 = U#'☹-Ä-`
                                To fit in USTRING[10], the string is
                                truncated from 12 code-units to 8 code-units
                                by dropping the last character *)

Ustring1:= "ABCÄ"; // results in Ustring1 = U#'ABCÄ' (implicit conversion)
```

e) Equivalent functions (see 6.6.2.5.11)

```
Char1:= String1[2];
```

is equivalent to

```
Char1:= STRING_TO_CHAR(Mid(IN:= String1, L:= 1, P:= 2));
aWStrings[1][3]:= Wchar1;
```

is equivalent to

```
REPLACE(IN1:= aWStrings[1], IN2:= Wchar1, L:= 1, P:=3 );
```

f) Error cases

```

String1[2] := Wchar1;           //invalid implicit conversion
String1[2] := String2;         //requires implicit conversion STRING_TO_CHAR
                                //which is not allowed

Char1 := Uchar1;               // invalid implicit conversion
Ustring1 := U#'ä'; Uchar1 := Ustring1[2]; // access beyond string
Byte1 := Ustring1[1];          // invalid conversion
Ustring1 := '${1F600}';        // error, since the character 🐼 is not part of
                                // the character set of a STRING-literal

```

NOTE 3 The data types for single characters (CHAR and WCHAR) can only contain one character. Strings can contain several characters; therefore, strings can require additional management information which is not needed for single characters.

EXAMPLE 2

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

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

```

VAR
    UString10: USTRING[10] := U#'A+Ä-🐼1F68D'; //5 UCHAR with 9 code-units:'A+Ä-🐼'
END_VAR

```

Here the variable UString10 is defined by the maximum length of 10 code-units, default initial value is 'A+Ä-🐼' containing 5 Unicode characters and 9 code-units.

6.4.3 Generic data types

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

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

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

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

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

Figure 5 – Hierarchy of the generic data types

6.4.4 User-defined data types

6.4.4.1 Declaration (TYPE)

6.4.4.1.1 General

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

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

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

A user-defined type declaration consists of

- the name of the type,
- the character “:” (colon), and
- the declaration of the type itself as defined in the following clauses.

EXAMPLE Type declaration

```

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

```

6.4.4.1.2 Initialization

The initial value is given by the default initial value, unless a user-defined value is specified.

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

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

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

Generic data type	Initialized by literal	Result
ANY_UNSIGNED	Non-negative integer literal or non-negative constant expression	Non-negative integer value
ANY_SIGNED	Integer literal or constant expression	Integer value
ANY_REAL	Numeric literal or constant expression	Numeric value
ANY_BIT	Unsigned integer literal or unsigned constant expression	Unsigned integer value
ANY_STRING	String literal	String value
ANY_DATE	Date and time of day literal	Date and time of day value
ANY_DURATION	Duration literal	Duration value

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

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

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

No.	Description	Example	Explanation
1a 1b	Enumerated data types	<pre> TYPE ANALOG_SIGNAL_RANGE: (BIPOLAR_10V, UNIPOLAR_10V, UNIPOLAR_1_5V, UNIPOLAR_0_5V, UNIPOLAR_4_20_MA, UNIPOLAR_0_20_MA) := UNIPOLAR_1_5V; END_TYPE </pre>	Initialization
2a 2b	Named values data types	<pre> TYPE Colors: DWORD (Red:= 16#00FF0000, Green:= 16#0000FF00, Blue:= 16#000000FF, White:= Red OR Green OR Blue, Black:= Red AND Green AND Blue) := Green; END_TYPE </pre>	Initialization
3a 3b	Subrange data types	<pre> TYPE ANALOG_DATA: INT(-4095 .. 4095) := 0; END_TYPE </pre>	
4a 4b	Array data types	<pre> TYPE ANALOG_16_INPUT_DATA: ARRAY [1..16] OF ANALOG_DATA := [8(-4095), 8(4095)]; END_TYPE </pre>	ANALOG_DATA see above. Initialization
5a 5b	FB types and classes as array elements	<pre> TYPE TONs: ARRAY[1..50] OF TON := [50((PT:=T#100ms))]; END_TYPE </pre>	FB TON as array element Initialization

No.	Description	Example	Explanation
6a 6b	Structured data type	<pre> TYPE ANALOG_CHANNEL_CONFIGURATION: STRUCT RANGE: ANALOG_SIGNAL_RANGE; MIN_SCALE: ANALOG_DATA:= -4095; MAX_SCALE: ANALOG_DATA:= 4095; END_STRUCT; END_TYPE </pre>	ANALOG_SIGNAL_RANGE see above
7a 7b	FB types and classes as structure elements	<pre> TYPE Cooler: STRUCT Temp: INT; Cooling: TOF:= (PT:= T#100ms); END_STRUCT; END_TYPE </pre>	FB TOF as structure element
8a 8b	Structured data type with relative addressing AT	<pre> TYPE Com1_data: STRUCT head AT %B0: INT; length AT %B2: USINT:= 26; flag1 AT %X3.0: BOOL; end AT %B25: BYTE; END_STRUCT; END_TYPE </pre>	Explicit layout without overlapping
9a	Structured data type with relative addressing AT and OVERLAP	<pre> TYPE Com2_data: STRUCT OVERLAP head AT %B0: INT; length AT %B2: USINT; flag2 AT %X3.3: BOOL; data1 AT %B5: BYTE; data2 AT %B5: REAL; end AT %B19: BYTE; END_STRUCT; END_TYPE </pre>	Explicit layout with overlapping
10a 10b	Directly represented elements of a structure – partly specified using “*”	<pre> TYPE HW_COMP: STRUCT; IN AT %I*: BOOL; OUT_VAR AT %Q*: WORD:= 200; ITNL_VAR: REAL:= 123.0; // not located END_STRUCT; END_TYPE </pre>	Assigns the components of a structure to not yet located inputs and outputs
11a 11b	Directly derived data types	<pre> TYPE CNT: UINT; FREQ: REAL:= 50.0; ANALOG_CHANNEL_CONFIG: ANALOG_CHANNEL_CONFIGURATION := (MIN_SCALE:= 0, MAX_SCALE:= 4000); END_TYPE </pre>	Initialization new initialization
12	Initialization using constant expressions	<pre> TYPE PIx2: REAL:= 2 * 3.1416; END_TYPE </pre>	Uses a constant expression

No.	Description	Example	Explanation
13a 13b	Reference data type	<pre> TYPE ref1: REF_TO INT; ref2: REF_TO INT:= NULL; st: STRUCT a: INT; ref3: REF_TO INT:= REF(a); END_STRUCT; END_TYPE VAR_GLOBAL g: ARRAY [0..7] OF INT; END_VAR PROGRAM A VAR_EXTERNAL g: ARRAY [0..7] OF INT; END_VAR VAR var1: REF_TO INT:= REF(g[3]); var2: INT; var3: REF_TO INT:= REF(var2); END_VAR ; END_PROGRAM </pre>	Initialization Initialization Initialization Initialization
The declaration of data type is possible without initialization (feature a) or with (feature b) initialization. If only feature (a) is supported, the data type is initialized with the default initial value. If feature (b) is supported, the data type shall be initialized with the given value or default initial value, if no initial value is given. Variables with directly represented elements of a structure – partly specified using “ * ” shall not be used in the VAR_INPUT, VAR_IN_OUT or VAR_TEMP sections.			

6.4.4.2 Enumerated data type

6.4.4.2.1 General

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

The enumeration list defines an ordered set of enumerated identifiers, starting with the first identifier of the list, and ending with the last one. Different enumerated data types can use the same identifiers. The maximum allowed number of enumerated identifiers is implementer specific. It shall be an error if an enumerated identifier is assigned a value.

NOTE 1 The user has no means to determine the value that has been internally assigned to an enumeration identifier.

An enumerated data type T is not related to any other type, except for directly derived datatypes of (transitive) base type T as defined in 6.4.4.9.

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

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

NOTE 2 An expression of the enumeration's type can only evaluate to values of the enumeration.

EXAMPLE Enumerated data type

```

TYPE
  Traffic_light: (Red, Amber, Green);
  Painting_colors: (Red, Yellow, Green, Blue) := Blue;
  Painting_colors2: Painting_colors := Green;    // Painting_colors2 is directly derived
                                                // data type, hence it is compatible to
                                                // Painting_colors.
END_TYPE

VAR
  My_Traffic_light: Traffic_light := Amber;
END_VAR

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

```

6.4.4.2.2 Initialization

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

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

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

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

6.4.4.3 Named values data type**6.4.4.3.1 General**

The named values data type is similar to the enumerated data types, but also specifies a base type and the encoding of its values, as illustrated in Table 11.

The definition of a base type is mandatory. The base type shall be one of the `ANY_INT` or `ANY_BIT` types without `BOOL`. In contrast to the enumerated data type, a value of a named values data type can be supplied where a value of its base type is expected and vice versa.

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

In contrast to the enumerated data type, any operations on named values shall be available like on the base type of the named values.

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

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

EXAMPLE Definition and usage of named values data types

```

TYPE
  Traffic_light:   INT (Red:= 1, Amber:= 2, Green:= 3):= Green;
  Painting_colors: INT (Red:= 1, Yellow:= 2, Green:= 3, Blue:= 4):= Blue;
  BaseColors:     DWORD (Red:= 16#00FF0000, Green:= 16#0000FF00, Blue:= 16#000000FF);
  Colors:         DWORD (White:=BaseColors#Red OR BaseColors#Green OR BaseColors#Blue,
                        Black:=BaseColors#Red AND BaseColors#Green AND BaseColors#Blue);
END_TYPE

VAR
  My_Traffic_light: Traffic_light;
  My_Painting_Color: Painting_colors;
END_VAR

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

My_Traffic_light:= Traffic_light#Red + 1;

IF My_Traffic_light = 123 THEN ...           // OK
IF My_Traffic_light = Traffic_light#Amber THEN ... // OK
IF My_Traffic_light = Traffic_light#Red THEN ... // OK
IF My_Traffic_light = Amber THEN ...         // OK, because Amber is unique
IF My_Painting_Color = Amber THEN ...         // OK, it has the same base data type
IF My_Traffic_light = Red THEN ...            // Error, because Red is not unique
My_Traffic_light:= Painting_colors#Red;       // OK, since it has the same base data type
My_Traffic_light:= Painting_colors#Blue-1;    // OK
My_Traffic_light:= My_Painting_colors;        // OK

```

6.4.4.3.2 Initialization

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

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

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

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

6.4.4.4 Subrange data type**6.4.4.4.1 General**

A subrange is a subtype of its base type, which is limited to a user-defined range of values.

An expression of the base type can be given, where a value of the subrange type is expected. It is an error if a data element of a subrange type is assigned a value that is not a value of the subrange type, i.e. does not lie in the subrange which includes the specified upper and lower limit.

It is not an error if an intermediate value (e.g. during expression evaluation) exceeds the bounds of the subrange, as long as the bounds of its base type are not exceeded. The limits of a subrange shall be literals or constant expressions and shall be values of the base type.

EXAMPLE:

```

TYPE
    ANALOG_DATA: INT (-4096..4095) := 0;
END_TYPE

PROGRAM prog
VAR
    i: INT;
    a: ANALOG_DATA;
    b AT %I*: ANALOG_DATA;
END_VAR
[... ]
a:= i;           // is an error, if i exceeds the range -4095..4095;
a:= (a*2)/3;     // no error. a*2 can exceed the subrange, but the assigned value
                // does not exceed the subrange
a:= (a*20)/30;   // is an error, if a*20 exceeds the range -32768..32767 of INT
i:= a;          // no error

// externally supplied values of b can exceed the range without leading to an
// error
IF NOT LE(-4096, b, 4095) THEN //LE(...): less than or equal
    (*error handling*)
END_IF;
END_PROGRAM

```

6.4.4.4.2 Initialization

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

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

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

6.4.4.5 Array data type**6.4.4.5.1 General**

The declaration of an array data type specifies the datatype of array-elements and an ordered sequence of index-ranges. The sequence shall not be empty. Each range consists of a lower and an upper bound. Each range corresponds to one dimension of the array.

It shall be an error if a dimension is indexed with an integer value that is smaller than the lower bound or larger than the upper bound. It shall be an error if a dimension is indexed with a non-integer value. It shall be an error if the result of subtracting the lower bound from the upper bound is smaller than -1. Support for a result of -1 (i.e. an array with no element) is implementer specific.

NOTE 1 The ARRAY[1..1] contains one element, ARRAY[1..0] contains no element.

Two array data types that differ in their element datatype or in the ordered sequence of index-ranges are different data types.

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

NOTE 2 This error can be detected only at runtime for a computed index.

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

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

In the ST language a subscript shall be an expression yielding a value of generic type `ANY_INT`.

In graphical languages the nominal form of subscripts defined in 8.1.2 is integer literals or single-element variables of generic type `ANY_INT`.

EXAMPLE

a) Declaration of an array

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

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

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

6.4.4.5.2 Initialization

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

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

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

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

EXAMPLE Initialization of an array

```
A: ARRAY [0..5] OF INT:= [2(1, 2, 3)]
is equivalent to the initialization sequence 1, 2, 3, 1, 2, 3.
```

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

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

6.4.4.6 Structured data type

6.4.4.6.1 General

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

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

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

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

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

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

a) Declaration of a structured data type

```
TYPE
  ANALOG_SIGNAL_RANGE: (BIPOLAR_10V, UNIPOLAR_10V);
  ANALOG_DATA: INT (-4095 .. 4095);
  ANALOG_CHANNEL_CONFIGURATION:
    STRUCT
      RANGE:          ANALOG_SIGNAL_RANGE;
      MIN_SCALE, MAX_SCALE: ANALOG_DATA;
    END_STRUCT;
END_TYPE
```

b) Declaration of structured variables

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

c) Usage of structured variables in the ST language:

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

6.4.4.6.2 Initialization

The default initial value of a structured data type is constituted of the initial values of its elements.

The default initial value of such an element is the initial value of its type. A user-defined initial value for an element can be defined, which takes precedence over the default initial value as initial value.

A user-defined initial value can be specified for a structured data type. The user-defined initial value is given by specifying values for the elements of the data type, using the syntax defined in Annex A.

A user-defined initial value for a structured data type does not need to define a value for each element of that structure.

If no user-defined initial value is specified for a structured data type, then the initial value of that data type is the default initial value.

If a user-defined initial value does not define a value for an element, the value for that element in the structure's default initial value is used as initial value.

These rules apply recursively if the type of an element of a structured data type is itself a structured data type.

Formally, let *IVD* be the default initial value of a structured data type *S* and *IVU* the user-defined initial value of *S*. If *C* is a component of *S* that is included in the definition of *IVU*, the initial value of *C* is defined to be the value of *C* in *IVU*. Otherwise, the initial value of *C* defined by *IVD*.

NOTE Informally, the initial value for a structured data type is the default initial value of the data type, unless specifically overridden for some or all of its elements.

If no user-defined initial value *IVU* is defined for a structured datatype *S*, the default initial value *IVD* is used for initialization. If a user-defined initial value *IVU* is defined that covers all components *C* of *S*, that value is used for initialization. If a user-defined initial value *IVU* is defined that covers only a part of the components *C* of *S*, it is used to initialize these components. The default initial value *IVD* is used to initialize the remaining components. This means that the user-defined initial value *IVU* overrides only a part of the default initial value *IVD* in this case.

EXAMPLE Definition of a structure and initial values

```

TYPE
  T: INT:= 1;
  S: STRUCT
    I: T;           // initial value is 1
    J: T:= 20;      // initial value is 20
    K: T:= 30;      // initial value is 30
    L: T:= 40;      // initial value is 40
  END_STRUCT;

  U: S:= (K:= 300); // initial values are (I=1,J=20,K=300,L=40)

  V: U:= (L:= 4000); // initial values are (I=1,J=20,K=300,L=4000)
END_TYPE

```

The initial value 1 of the element *I* is its default initial value, which is the initial value of *T*.

The initial value 20 of the element *J* is its user-defined initial value.

The initial value (*I*:=1, *J*:=20, *K*:=30, *L*:=40) of the structural type *S* is its default initial value, which is constituted by the initial values of its elements.

The initial value (*I*:=1, *J*:=20, *K*:=300, *L*:=40) of *U* is composed of its default initial value, which is the initial value of *S* and of *U*'s user-defined initial value, which is (*K*:=300). The latter takes precedence.

The initial value (*I*:=1, *J*:=20, *K*:=300, *L*:=4000) of *V* is composed of its default initial value, which is the initial value of *U* and of *V*'s user-defined initial value, which is (*L*:=4000). The latter takes precedence.

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

6.4.4.7.1 General

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

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

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