



Programmable controllers

Part 3: Programming languages



This Australian Standard® was prepared by Committee IT-006, Industrial Process Measurement, Control and Automation. It was approved on behalf of the Council of Standards Australia on 14 November 2014.
This Standard was published on 19 December 2014.

The following are represented on Committee IT-006:

- Australian Computer Society
 - Australian Industry Group
 - Australian Petroleum Production and Exploration Association
 - Australian Safety Critical Systems Association
 - Consult Australia
 - Engineers Australia
 - Institute of Chemical Engineers Australia
 - Institute of Instrumentation, Control and Automation Australia
 - ISACA
 - Process Control Society
 - University of Queensland
 - Workplace Health and Safety Queensland
-

This Standard was issued in draft form for comment as DR AS IEC 61131.3:2014.

Standards Australia wishes to acknowledge the participation of the expert individuals that contributed to the development of this Standard through their representation on the Committee and through the public comment period.

Keeping Standards up-to-date

Australian Standards® are living documents that reflect progress in science, technology and systems. To maintain their currency, all Standards are periodically reviewed, and new editions are published. Between editions, amendments may be issued.

Standards may also be withdrawn. It is important that readers assure themselves they are using a current Standard, which should include any amendments that may have been published since the Standard was published.

Detailed information about Australian Standards, drafts, amendments and new projects can be found by visiting www.standards.org.au

Standards Australia welcomes suggestions for improvements, and encourages readers to notify us immediately of any apparent inaccuracies or ambiguities. Contact us via email at mail@standards.org.au, or write to Standards Australia, GPO Box 476, Sydney, NSW 2001.

Australian Standard[®]

Programmable controllers

Part 3: Programming languages

Originated as AS 4168.3—1994.
Previous edition AS IEC 61131.3—2004.
Second edition 2014.

COPYRIGHT

© Standards Australia Limited

All rights are reserved. No part of this work may be reproduced or copied in any form or by any means, electronic or mechanical, including photocopying, without the written permission of the publisher, unless otherwise permitted under the Copyright Act 1968.

Published by SAI Global Limited under licence from Standards Australia Limited, GPO Box 476, Sydney, NSW 2001, Australia

ISBN 978 1 74342 931 0

PREFACE

This Standard was prepared by the Standards Australia Committee IT-006, Industrial Process Measurement, Control and Automation, to supersede AS IEC 61131.3—2004.

The objective of this Standard is to specify syntax and semantics of programming languages for programmable controllers as defined in AS IEC 61131.1.

This Standard should be read in conjunction with the other parts of the AS IEC 61131 series.

This Standard is identical with, and has been reproduced from, IEC 61131-3, Ed. 3.0 (2013), *Programmable controllers—Part 3: Programming languages*.

This edition is a fully compatible extension of the previous (2004) edition. The main extensions are new data types and conversion functions, references, name spaces and the object oriented features of classes and function blocks. For further details, see Annex B.

As this Standard is reproduced from an International Standard, the following applies:

- (a) In the source text ‘this part of IEC 61131’ should read ‘this Australian Standard’.
- (b) A full point substitutes for a comma when referring to a decimal marker.

References to International Standards should be replaced by references to Australian or Australian/New Zealand Standards, as follows:

<i>Reference to International Standard</i>	<i>Australian Standard</i>
IEC	AS IEC
61131 Programmable controllers	61131 Programmable controllers
61131-1 Part 1: General information	61131.1 Part 1: General information
61131-5 Part 5: Communications	61131.5 Part 5: Communications

Only normative references that have been adopted as Australian or Australian/New Zealand Standards have been listed.

The terms ‘normative’ and ‘informative’ are used to define the application of the annex to which they apply. A normative annex is an integral part of a standard, whereas an informative annex is only for information and guidance.

CONTENTS

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

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

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

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

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

Table 10 – Elementary data types	31
Table 11 – Declaration of user-defined data types and initialization	35
Table 12 – Reference operations	46
Table 13 – Declaration of variables	48
Table 14 – Initialization of variables	49
Table 15 – Variable-length <code>ARRAY</code> variables	52
Table 16 – Directly represented variables	54
Table 17 – Partial access of <code>ANY_BIT</code> variables	60
Table 18 – Execution control graphically using <code>EN</code> and <code>ENO</code>	65
Table 19 – Function declaration	72
Table 20 – Function call	74
Table 21 – Typed and overloaded functions	76
Table 22 – Data type conversion function	78
Table 23 – Data type conversion of numeric data types	80
Table 24 – Data type conversion of bit data types	82
Table 25 – Data type conversion of bit and numeric types	83
Table 26 – Data type conversion of date and time types	85
Table 27 – Data type conversion of character types	86
Table 28 – Numerical and arithmetic functions	87
Table 29 – Arithmetic functions	88
Table 30 – Bit shift functions	89
Table 31 – Bitwise Boolean functions	89
Table 32 – Selection functions ^d	90
Table 33 – Comparison functions	91
Table 34 – Character string functions	92
Table 35 – Numerical functions of time and duration data types	93
Table 36 – Additional functions of time data types <code>CONCAT</code> and <code>SPLIT</code>	94
Table 37 – Function for endianness conversion	98
Table 38 – Functions of enumerated data types	98
Table 39 – Validate functions	99
Table 40 – Function block type declaration	100
Table 41 – Function block instance declaration	104
Table 42 – Function block call	105
Table 43 – Standard bistable function blocks ^a	112
Table 44 – Standard edge detection function blocks	113
Table 45 – Standard counter function blocks	113
Table 46 – Standard timer function blocks	115
Table 47 – Program declaration	117
Table 48 – Class	120
Table 49 – Class instance declaration	122
Table 50 – Textual call of methods – Formal and non-formal parameter list	125
Table 51 – Interface	137
Table 52 – Assignment attempt	140

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

NOTES

AUSTRALIAN STANDARD

Programmable controllers**Part 3:
Programming languages****1 Scope**

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

The functions of program entry, testing, monitoring, operating system, etc., are specified in Part 1 of IEC 61131.

This part of IEC 61131 specifies the syntax and semantics of a unified suite of programming languages for programmable controllers (PCs). This suite consists of two textual languages, Instruction List (IL) and Structured Text (ST), and two graphical languages, Ladder Diagram (LD) and Function Block Diagram (FBD).

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

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

2 Normative references

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

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

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

ISO/IEC 10646:2012, *Information technology – Universal Coded Character Set (UCS)*

ISO/IEC/IEEE 60559, *Information technology – Microprocessor Systems – Floating-Point arithmetic*

3 Terms and definitions

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

3.1**absolute time**

combination of time of day and date information

3.2**access path**

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

3.3**action**

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

3.4**action block**

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

3.5**aggregate**

structured collection of data objects forming a data type

[SOURCE: ISO/AFNOR:1989]

3.6**array**

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

[SOURCE: ISO/AFNOR:1989]

3.7**assignment**

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

[SOURCE: ISO/AFNOR:1989]

3.8**base type**

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

3.9**based number**

number represented in a specified base other than ten

3.10**binary coded decimal**

BCD

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

3.11**bistable function block**

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

3.12**bit string**

data element consisting of one or more bits

3.13**bit string literal**

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

3.14**body**

set of operations of the program organization unit

3.15**call**

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

3.16**character string**

aggregate that consists of an ordered sequence of characters

3.17**character string literal**

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

3.18**class**

program organization unit consisting of:

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

3.19**comment**

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

[SOURCE: ISO/AFNOR:1989]

3.20**configuration**

language element corresponding to a programmable controller system

3.21**constant**

language element which declares a data element with a fixed value

3.22**counter function block**

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

3.23**data type**

set of values together with a set of permitted operations

[SOURCE: ISO/AFNOR:1989]

3.24**date and time**

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

3.25**declaration**

mechanism for establishing the definition of a language element

3.26**delimiter**

character or combination of characters used to separate program language elements

3.27**derived class**

class created by inheritance from another class

Note 1 to entry: Derived class is also named extended class or child class.

3.28**derived data type**

data type created by using another data type

3.29**derived function block type**

function block type created by inheritance from another function block type

3.30**direct representation**

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

3.31**double word**

data element containing 32 bits

3.32**dynamic binding**

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

3.33**evaluation**

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

3.34**execution control element**

language element which controls the flow of program execution

3.35**falling edge**

change from 1 to 0 of a Boolean variable

3.36**function**

language element which, when executed, typically yields one data element result and possibly additional output variables

3.37**function block instance**

instance of a function block type

3.38**function block type**

language element consisting of:

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

3.39**function block diagram**

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

3.40**generic data type**

data type which represents more than one type of data

3.41**global variable**

variable whose scope is global

3.42**hierarchical addressing**

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

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

3.43**identifier**

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

3.44**implementation**

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

3.45**Implementer**

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

3.46**inheritance**

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

3.47**initial value**

value assigned to a variable at system start-up

3.48**in-out variable**

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

3.49**input variable**

variable which is used to supply a value to a program organization unit except for class

3.50**instance**

individual, named copy of the data structure associated with a function block type, class, or program type, which keeps its values from one call of the associated operations to the next

3.51**instance name**

identifier associated with a specific instance

3.52**instantiation**

creation of an instance

3.53**integer**

integer number which may contain positive, null, and negative values

3.54**integer literal**

literal which directly represents an integer value

3.55**interface**

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

3.56**keyword**

lexical unit that characterizes a language element

3.57**label**

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

3.58**language element**

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

3.59**literal**

lexical unit that directly represents a value

[SOURCE: ISO/AFNOR:1989]

3.60**logical location**

location of a hierarchically addressed variable in a schema which may or may not bear any relation to the physical structure of the programmable controller's inputs, outputs, and memory

3.61**long real**

real number represented in a long word

3.62**long word**

64-bit data element

3.63**method**

language element similar to a function that can only be defined in the scope of a function block type and with implicit access to static variables of the function block instance or class instance

3.64**method prototype**

language element containing only the signature of a method

3.65**named element**

element of a structure which is named by its associated identifier

3.66**network**

arrangement of nodes and interconnecting branches

3.67**numeric literal**

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

3.68**operation**

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

3.69**operand**

language element on which an operation is performed

3.70**operator**

symbol that represents the action to be performed in an operation

3.71**override**

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

3.72**output variable**

variable which is used to return a value from the program organization unit except for classes

3.73**parameter**

variable which is used to provide a value to a program organization unit (as input or in-out parameter) or a variable which is used to return a value from a program organization unit (as output or in-out parameter)

3.74**reference**

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

3.75**power flow**

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

3.76**pragma**

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

3.77**program**

to design, write, and test user programs

3.78**program organization unit**

function, function block, class, or program

3.79**real literal**

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

3.80**resource**

language element corresponding to a “signal processing function” and its “man-machine interface” and “sensor and actuator interface functions”, if any

3.81**result**

value which is returned as an outcome of a program organization unit

3.82**return**

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

3.83**rising edge**

change from 0 to 1 of a Boolean variable

3.84**scope**

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

3.85**semantics**

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

3.86**semigraphic representation**

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

3.87**signature**

set of information defining unambiguously the identity of the parameter interface of a `METHOD` consisting of its name and the names, types, and order of all its parameters (i.e. inputs, outputs, in-out variables, and result type)

3.88**single-element variable**

variable which represents a single data element

3.89**static variable**

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

3.90**step**

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

3.91**structured data type**

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

3.92**subscripting**

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

3.93**task**

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

3.94**time literal**

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

3.95**transition**

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

3.96**unsigned integer**

integer number which may contain positive and null values

3.97**unsigned integer literal**

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

3.98**user-defined data type**

data type defined by the user

EXAMPLE Enumeration, array or structure.

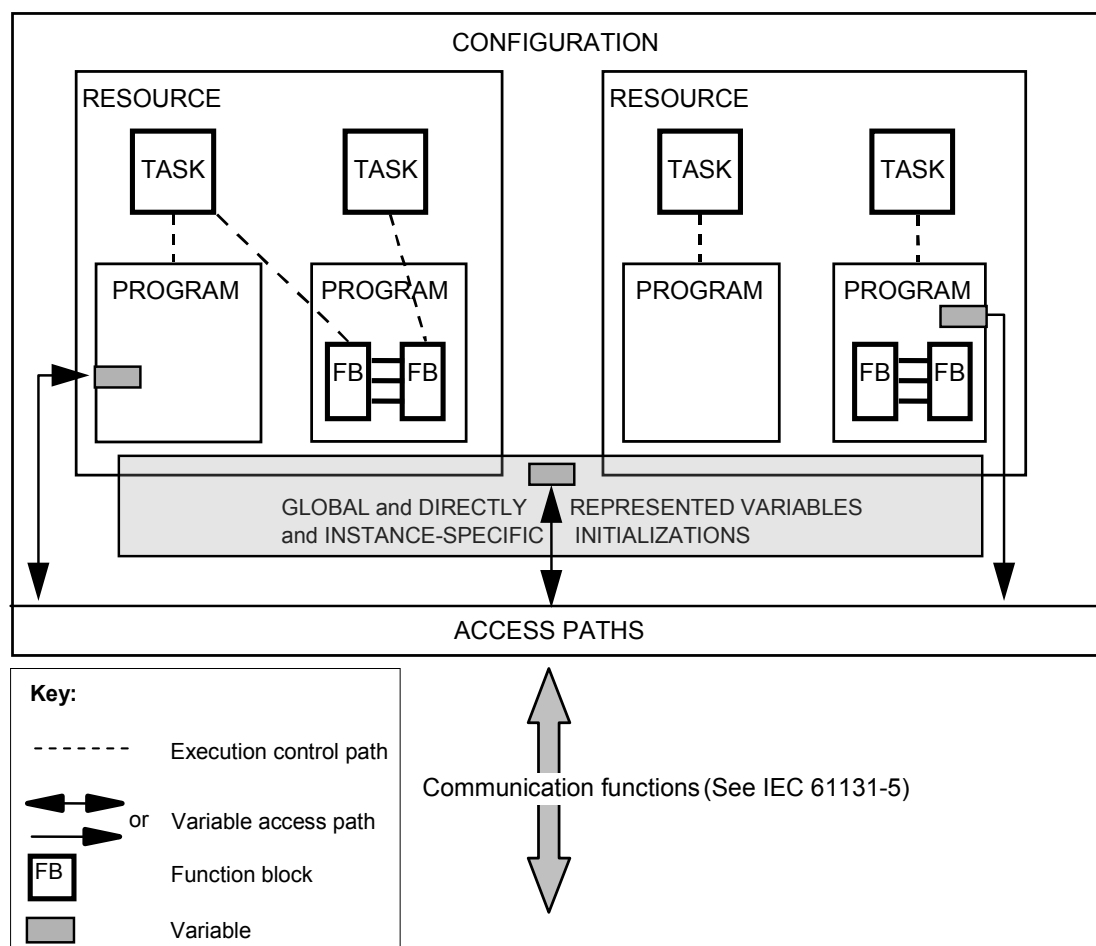
3.99**variable**

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

4 Architectural models**4.1 Software model**

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

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



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

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

Figure 1 – Software model

A configuration is the language element which corresponds to a programmable controller system as defined in IEC 61131-1. A resource corresponds to a “signal processing function” and its “man-machine interface” and “sensor and actuator interface” functions (if any) as defined in IEC 61131-1.

A configuration contains one or more resources, each of which contains one or more programs executed under the control of zero or more tasks.

A program may contain zero or more function block instances or other language elements as defined in this part of IEC 61131.

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

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

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

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

Access paths and their corresponding access privileges are defined in this standard.

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

4.2 Communication model

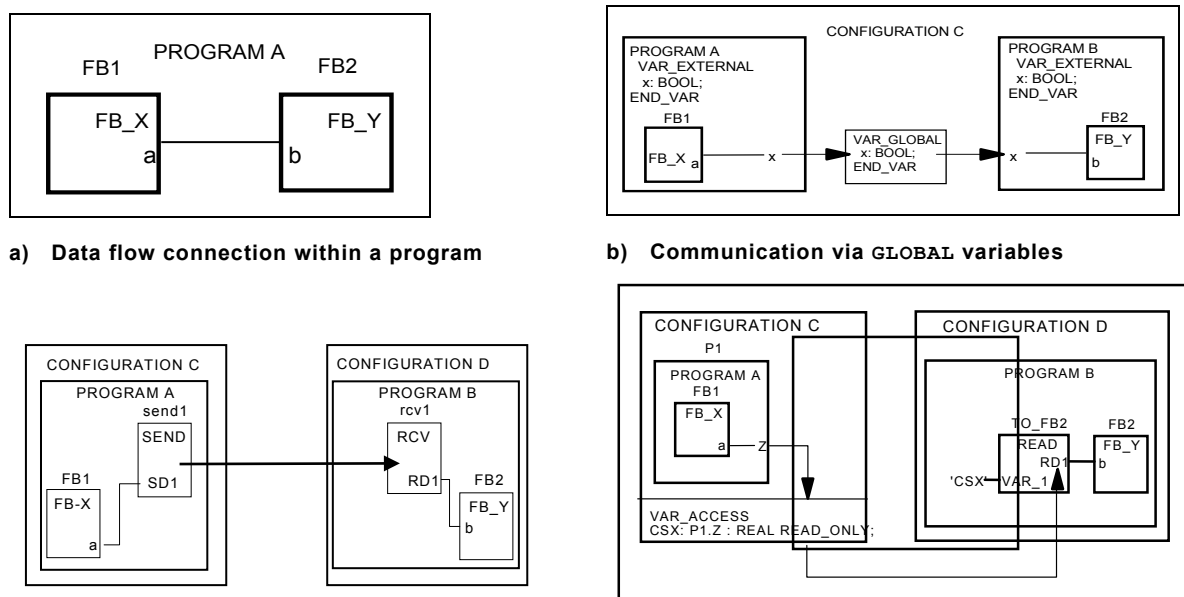
Figure 2 illustrates the ways that values of variables can be communicated among software elements.

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

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

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

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



a) Data flow connection within a program

b) Communication via GLOBAL variables

c) Communication function blocks

d) Communication via access paths

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

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

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

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

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

Figure 2 – Communication model

4.3 Programming model

In Figure 3 are the PLC Languages elements summarized. The combination of these elements shall obey the following rules:

1. Data types shall be declared, using the standard data types and any previously defined data types.
2. Functions can be declared using standard or user-defined data types, the standard functions and any previously defined functions.

This declaration shall use the mechanisms defined for the IL, ST, LD or FBD language.

3. Function block types can be declared using standard and user-defined data types, functions, standard function block types and any previously defined function block types.

These declarations shall use the mechanisms defined for the IL, ST, LD, or FBD language, and can include Sequential Function Chart (SFC) elements.

Optionally, one may define object oriented function block types or classes which use methods and interfaces.

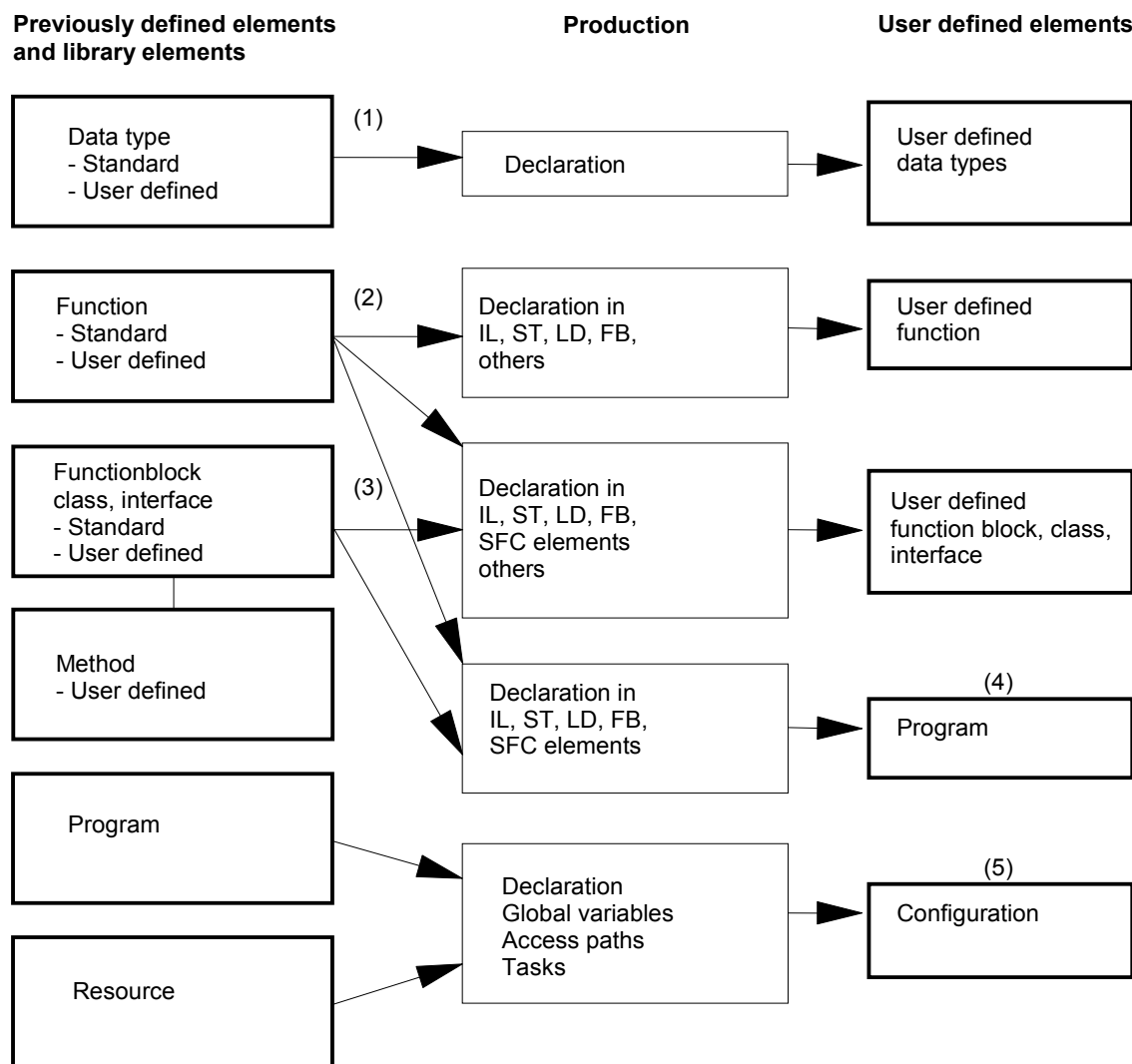
4. A program shall be declared using standard or user-defined data types, functions, function blocks and classes.

This declaration shall use the mechanisms defined for the IL, ST, LD, or FBD language, and can include Sequential Function Chart (SFC) elements.

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

Reference to “previously defined” data types, functions, and function blocks in the above rules is intended to imply that once such a previously defined element has been declared, its definition is available, for example, in a “library” of previously defined elements, for use in further definitions.

A programming language other than one of those defined in this standard may be used for programming of a function, function block type and methods.



LD: Ladder Diagram
 FBD: Function Block Diagram
 IL: Instruction List
 ST: Structured Text
 Others: Other programming languages

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

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

Figure 3 – Combination of programmable controller language elements

5 Compliance

5.1 General

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

- a) shall provide a subset of the features and provide the corresponding Implementer's compliance statement as defined below.
- b) shall not require the inclusion of substitute or additional language elements in order to accomplish any of the features.
- c) shall provide a document that specifies all Implementer specific extensions. These are any features accepted by the system that are prohibited or not specified.
- d) shall provide a document that specifies all Implementer specific dependencies. This includes the implementation dependencies explicitly designated in this part of IEC 61131 and the limiting parameters like maximum length, number, size and range of value which are not explicitly here.
- e) shall provide a document that specifies all errors that are detectable and reported by the implementation. This includes the errors explicitly designated in this part and the errors detectable during preparation of the program for execution and during execution of the program.

NOTE Errors occurring during execution of the program are only partially specified in this part of IEC 61131.

- f) shall not use any of the standard names of data types, function or function block names defined in this standard for implementation-defined features whose functionality differs from that described in this part of IEC 61131.

5.2 Feature tables

All tables in this part of IEC 61131 are used for a special purpose in a common way. The first column contains the "feature number", the second column gives the "feature description", the following columns may contain examples or further information. This table structure is used in the Implementer's compliance statement.

5.3 Implementer's compliance statement

The Implementer may define any consistent subset of the features listed in the feature tables and shall declare the provided subset in the "Implementer's compliant statement".

The Implementer's compliance statement shall be included in the documentation accompanying the system, or shall be produced by the system itself.

The format of the Implementer's compliance statement shall provide the following information. Figure 4 shows an example.

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

Optional is the title and subtitle of the feature table, the feature description, examples, Implementer's note etc.

Not implemented tables and features may be omitted.

IEC 61131-3 "PLC Programming Languages"						
Implementer: Company name, address, etc.						
Product: Product name, version, etc. Controller type specific subset, etc.						
Date: 2012-05-01						
This Product complies with the requirements of the standard for the following language features:						
Feature No.	Table number and title / Feature description	Compliantly implemented in the language (✓)				Implementer's note
		LD	FB D	ST	IL	
	Table 1 – Character set					
1	ISO/IEC 10646:2012, Information technology – Universal Coded Character Set (UCS)	✓	✓	✓	✓	
2a	Lower case characters a: a, b, c, ...	✓	✓	✓		No "ß, ü, ä, ö"
2b	Number sign: # See Table 5	✓				
2c	Dollar sign: \$ See Table 6		✓			
	Table 2 – Identifiers					
1	Upper case letters and numbers: IW215					
2	Upper and lower case letters, numbers, embedded under-score					
3	Upper and lower case, numbers, leading or embedded under-score					
	Table 3 – Comments					
1	Single-line comment //...					
2a	Multi-line comment (* ... *)					
2b	Multi-line comment /* ... */					
3a	Nested comment (* ..(* .. *) ..*)					
3b	Nested comment /* .. /* .. */ .. */					
	Table 4 – Pragma					
1	Pragma with curly brackets { ... }					
	Table 5 – Numeric literals					
1	Integer literal: -12					
2	Real literal: -12.0					
3	Real literals with exponent: -1.34E-12					
4	Binary literal: 2#1111_1111					
5	Octal literal: 8#377					
6	Hexadecimal literal: 16#FF					
7	Boolean zero and one					
8	Boolean FALSE and TRUE					
9	Typed literal: INT#-123					
	Etc.					

Figure 4 – Implementer's compliance statement (Example)

6 Common elements

6.1 Use of printed characters

6.1.1 Character set

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

Table 1 – Character set

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

6.1.2 Identifiers

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

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

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

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

Identifier features and examples are shown in Table 2.

Table 2 – Identifiers

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

6.1.3 Keywords

Keywords are unique combinations of characters utilized as individual syntactic elements. Keywords shall not contain embedded spaces. The case of characters shall not be significant.

in keywords; for instance, the keywords `FOR` and `for` are syntactically equivalent. They shall not be used for any other purpose, for example, variable names or extensions.

6.1.4 Use of white space

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

6.1.5 Comments

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

1. Single line comments start with the character combination `//` and end at the next following line feed, new line, form feed (page), or carriage return.

In single-line comments the special character combinations `( * and *)` or `/* and */` have no special meaning.

2. Multi-line comments shall be delimited at the beginning and end by the special character combinations `( * and *)`, respectively.

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

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

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

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

Nested comments use corresponding

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

Table 3 – Comments

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

6.2 Pragma

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

Table 4 – Pragma

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

6.3 Literals – External representation of data

6.3.1 General

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

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

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

6.3.2 Numeric literals and string literals

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

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

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

Literals can also be represented in base 2, 8, or 16. The base shall be in decimal notation. For base 16, an extended set of digits consisting of the letters A through F shall be used, with the conventional significance of decimal 10 through 15, respectively. Based numbers shall not contain a leading sign “+” or “-”. They are interpreted as bit string literals.

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

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

Numeric literal features and examples are shown in Table 5.

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

Table 5 – Numeric literals

No.	Description	Examples	Explanation
1	Integer literal	-12, 0, 123_4, +986	
2	Real literal	0.0, 0.4560, 3.14159_26	
3	Real literals with exponent	-1.34E-12, -1.34e-12 1.0E+6, 1.0e+6 1.234E6, 1.234e6	
4	Binary literal	2#1111_1111 2#1110_0000	Base 2 literal 255 decimal 224 decimal
5	Octal literals	8#377 8#340	Base 8 literal 255 decimal 224 decimal
6	Hexadecimal literal	16#FF or 16#ff 16#E0 or 16#e0	Base 16 literal 255 decimal 224 decimal
7	Boolean zero and one	0 or 1	
8	Boolean FALSE and TRUE	FALSE TRUE	
9	Typed literal	INT#-123	INT representation of the decimal value -123
		INT#16#7FFF	INT representation of the decimal value 32767
		WORD#16#AFF	WORD representation of the hexadecimal value 0AFF
		WORD#1234	WORD representation of the decimal value 1234=16#4D2
		UINT#16#89AF	UINT representation of the hexadecimal value 89AF
		CHAR#16#41	CHAR representation of the 'A'
		BOOL#0	
		BOOL#1	
		BOOL#FALSE	
		BOOL#TRUE	

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

NOTE 2 The feature 5 'Octal literals' is deprecated and may not be included in the next edition of this part of IEC 61131.

6.3.3 Character string literals

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

- A single-byte character string literal is a sequence of zero or more characters prefixed and terminated by the single quote character ('). In single-byte character strings, the three-character combination of the dollar sign (\$) followed by two hexadecimal digits shall be interpreted as the hexadecimal representation of the eight-bit character code, as shown in feature 1 of Table 6.
- A double-byte character string literal is a sequence of zero or more characters from the ISO/IEC 10646 character set prefixed and terminated by the double quote character ("). In double-byte character strings, the five-character combination of the dollar sign "\$" followed by four hexadecimal digits shall be interpreted as the hexadecimal representation of the sixteen-bit character code, as shown in feature 2 of Table 6.

NOTE Relation of ISO/IEC 10646 and Unicode:

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

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

Table 6 – Character string literals

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

No.	Description	Examples
NOTE If a particular implementation supports feature 4 but not feature 2, the Implementer may specify Implementer specific syntax and semantics for the use of the double-quote character.		

Table 7 – Two-character combinations in character strings

No.	Description	Combinations
1	Dollar sign	\$\$
2	Single quote	\$'
3	Line feed	\$L or \$l
4	Newline	\$N or \$n
5	Form feed (page)	\$P or \$p
6	Carriage return	\$R or \$r
7	Tabulator	\$T or \$t
8	Double quote	\$"
NOTE 1 The "newline" character provides an implementation-independent means of defining the end of a line of data; for printing, the effect is that of ending a line of data and resuming printing at the beginning of the next line.		
NOTE 2 The \$' combination is only valid inside single quoted string literals.		
NOTE 3 The \$" combination is only valid inside double quoted string literals.		

6.3.4 Duration literal

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

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

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

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

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

Table 8 – Duration literals

No.	Description	Examples
	Duration abbreviations	
1a	d	Day
1b	h	Hour
1c	m	Minute
1d	s	Second
1e	ms	Millisecond
1f	us (no μ available)	Microsecond
1g	ns	Nanoseconds

No.	Description	Examples
Duration literals without underscore		
2a	short prefix	T#14ms T#-14ms LT#14.7s T#14.7m T#14.7h t#14.7d t#25h15m lt#5d14h12m18s3.5ms t#12h4m34ms230us400ns
2b	long prefix	TIME#14ms TIME#-14ms time#14.7s
Duration literals with underscore		
3a	short prefix	t#25h_15m t#5d_14h_12m_18s_3.5ms LTIME#5m_30s_500ms_100.1us
3b	long prefix	TIME#25h_15m ltime#5d_14h_12m_18s_3.5ms LTIME#34s_345ns

6.3.5 Date and time of day literal

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

Table 9 – Date and time of day literals

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

6.4 Data types

6.4.1 General

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

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

6.4.2.1 Specification of elementary data types

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

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

Table 10 – Elementary data types

No.	Description	Keyword	Default initial value	N (bits) ^a
1	Boolean	BOOL	0, FALSE	1 ^h
2	Short integer	SINT	0	8 ^c
3	Integer	INT	0	16 ^c
4	Double integer	DINT	0	32 ^c
5	Long integer	LINT	0	64 ^c
6	Unsigned short integer	USINT	0	8 ^d
7	Unsigned integer	UINT	0	16 ^d
8	Unsigned double integer	UDINT	0	32 ^d
9	Unsigned long integer	ULINT	0	64 ^d
10	Real numbers	REAL	0.0	32 ^e
11	Long reals	LREAL	0.0	64 ^f
12a	Duration	TIME	T#0s	-- ^b
12b	Duration	LTIME	LTIME#0s	64 ^{m, q}
13a	Date (only)	DATE	NOTE	-- ^b
13b	Long Date (only)	LDATE	LDATE#1970-01-01	64 ⁿ
14a	Time of day (only)	TIME_OF_DAY or TOD	TOD#00:00:00	-- ^b
14b	Time of day (only)	LTIME_OF_DAY or LTOD	LTOD#00:00:00	64 ^{o, q}
15a	Date and time of Day	DATE_AND_TIME or DT	NOTE	-- ^b
15b	Date and time of Day	LDATE_AND_TIME or LDT	LDT#1970-01-01-00:00:00	64 ^{p, q}
16a	Variable-length single-byte character string	STRING	' ' (empty)	8 ^{i, g, k, l}
16b	Variable-length double-byte character string	WSTRING	" " (empty)	16 ^{i, g, k, l}
17a	Single-byte character	CHAR	'\$00'	8 ^{g, l}
17b	Double-byte character	WCHAR	"\$0000"	16 ^{g, l}
18	Bit string of length 8	BYTE	16#00	8 ^{j, g}
19	Bit string of length 16	WORD	16#0000	16 ^{j, g}
20	Bit string of length 32	DWORD	16#0000_0000	32 ^{j, g}
21	Bit string of length 64	LWORD	16#0000_0000_0000_0000	64 ^{j, g}
NOTE Implementer specific because of special starting date different than 0001-01-01.				

No.	Description	Keyword	Default initial value	N (bits) ^a
a	Entries in this column shall be interpreted as specified in the table footnotes.			
b	The range of values and precision of representation in these data types is Implementer specific.			
c	The range of values for variables of this data type is from $-(2^{N-1})$ to $(2^{N-1}) - 1$.			
d	The range of values for variables of this data type is from 0 to $(2^N) - 1$.			
e	The range of values for variables of this data type shall be as defined in IEC 60559 for the basic single width floating-point format. Results of arithmetic instructions with denormalized values, infinity, or not-a-number values are Implementer specific.			
f	The range of values for variables of this data type shall be as defined in IEC 60559 for the basic double width floating-point format. Results of arithmetic instructions with denormalized values, infinity, or not-a-number values are Implementer specific.			
g	A numeric range of values does not apply to this data type.			
h	The possible values of variables of this data type shall be 0 and 1, corresponding to the keywords <code>FALSE</code> and <code>TRUE</code> , respectively.			
i	The value of N indicates the number of bits/character for this data type.			
j	The value of N indicates the number of bits in the bit string for this data type.			
k	The maximum allowed length of <code>STRING</code> and <code>WSTRING</code> variables is Implementer specific.			
l	The character encoding used for <code>CHAR</code> , <code>STRING</code> , <code>WCHAR</code> , and <code>WSTRING</code> is ISO/IEC 10646 (see 6.3.3).			
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 date 1970-01-01.			
o	The data type <code>LDT</code> is a signed 64-bit integer with unit of nanoseconds with starting date 1970-01-01-00:00:00.			
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.			

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

The supported maximum length of elements of type `STRING` and `WSTRING` shall be Implementer specific values and define the maximum length of a `STRING` and `WSTRING` which is supported by the programming and debugging tool.

The explicit maximum length is specified by a parenthesized maximum length (which shall not exceed the Implementer specific supported maximum value) in the associated declaration.

Access to single characters of a string using elements of the data type `CHAR` or `WCHAR` shall be supported using square brackets and the position of the character in the string, starting with position 1.

It shall be an error if double byte character strings are accessed using single byte characters or if single byte character strings are accessed using double byte characters.

EXAMPLE 1 STRING, WSTRING and CHAR, WCHAR**a) Declaration**

VAR

```

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

```

END_VAR

b) Usage of STRING and CHAR

```

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

```

c) Usage of WSTRING and WCHAR

```

WChar1:= aWStrings[1][2];      //is equivalent to WChar1:= '6';
aWStrings[1][3]:=WChar1;       //results in aWStrings[1]:= "5668"
aWStrings[1][4]:= "6";         //results in aWStrings[1]:= "5666"
aWStrings[1][1]:= aWStrings[1][4]; //results in String1:= "6666"
aWStrings[0]:= aWStrings[1][4]; (* results in aWStrings[0]:= "6";
                               if implicit conversion WCHAR_TO_WSTRING has been implemented *)

```

d) Equivalent functions (see 6.6.2.5.11)

```

Char1:= String1[2];
is equivalent to
Char1:= STRING_TO_CHAR(Mid(IN:= String1, L:= 1, P:= 2));
aWStrings[1][3]:= WChar1;
is equivalent to
REPLACE(IN1:= aWStrings[1], IN2:= WChar1, L:= 1, P:=3 );

```

e) Error cases

```

Char1:= String1[2]; //mixing WCHAR, STRING
String1[2]:= String2;
//requires implicit conversion STRING_TO_CHAR which is not allowed

```

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

EXAMPLE 2

If type STR10 is declared by

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

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

6.4.3 Generic data types

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

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

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

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

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

Figure 5 – Hierarchy of the generic data types

6.4.4 User-defined data types

6.4.4.1 Declaration (TYPE)

6.4.4.1.1 General

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

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

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

A type declaration consists of

- the name of the type
- a ':' (colon)
- the declaration of the type itself as defined in the following clauses.

EXAMPLE Type declaration

```

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

```

6.4.4.1.2 Initialization

User-defined data types can be initialized with user-defined values. This initialization has priority over the default initial value.

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

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

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

Generic Data Type	Initialized by literal	Result
ANY_UNSIGNED	Non-negative integer literal or non-negative constant expression	Non-negative integer value
ANY_SIGNED	Integer literal or constant expression	Integer value
ANY_REAL	Numeric literal or constant expression	Numeric value
ANY_BIT	Unsigned integer literal or unsigned constant expression	Unsigned integer value
ANY_STRING	String literal	String value
ANY_DATE	Date and Time of Day literal	Date and Time of Day value
ANY_DURATION	Duration literal	Duration value

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

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

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

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

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

6.4.4.2 Enumerated data type

6.4.4.2.1 General

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

The enumeration list defines an ordered set of enumerated values, starting with the first identifier of the list, and ending with the last one.

Different enumerated data types may use the same identifiers for enumerated values. The maximum allowed number of enumerated values is Implementer specific.

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

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

EXAMPLE Enumerated data type

```

TYPE
  Traffic_light: (Red, Amber, Green);
  Painting_colors: (Red, Yellow, Green, Blue) := Blue;
END_TYPE

VAR
  My_Traffic_light: Traffic_light := Red;
END_VAR

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

```

6.4.4.2.2 Initialization

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

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

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

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

6.4.4.3 Data type with named values

6.4.4.3.1 General

Related to the enumeration data type – where the values of enumerated identifiers are not known by the user – is an enumerated data type with named values. The declaration specifies the data type and assigns the values of the named values, as illustrated in Table 11.

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

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

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

EXAMPLE Data type with named values

```

TYPE
  Traffic_light:  INT (Red:= 1, Amber := 2, Green:= 3):= Green;
  Painting_colors: INT (Red:= 1, Yellow:= 2, Green:= 3, Blue:= 4):= Blue;
END_TYPE

VAR
  My_Traffic_light: Traffic_light;
END_VAR

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

IF My_Traffic_light = 123 THEN ...           // OK
IF My_Traffic_light = Traffic_light#Amber THEN ... // OK
IF My_Traffic_light = Traffic_light#Red THEN ... // OK
IF My_Traffic_light = Amber THEN ...         // OK because Amber is unique
IF My_Traffic_light = Red THEN ...           // Error because Red is not unique

```

6.4.4.3.2 Initialization

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

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

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

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

6.4.4.4 Subrange data type

6.4.4.4.1 General

A subrange declaration specifies that the value of any data element of that type can only take on values between and including the specified upper and lower limits, as illustrated in Table 11.

The limits of a subrange shall be literals or constant expressions.

EXAMPLE

```

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

```

6.4.4.4.2 Initialization

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

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

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

6.4.4.5 Array data type

6.4.4.5.1 General

The declaration of an array data type specifies that a sufficient amount of data storage shall be allocated for each element of that type to store all the data which can be indexed by the specified index subrange(s), as illustrated in Table 11.

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

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

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

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

In the ST language a subscript shall be an expression yielding a value corresponding to one of the sub-types of generic type `ANY_INT`.

The form of subscripts in the IL language and the graphic languages defined in Clause 8 is restricted to single-element variables or integer literals.

EXAMPLE

a) Declaration of an array

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

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

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

6.4.4.5.2 Initialization

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

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

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

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

EXAMPLE Initialization of an array

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

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

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

6.4.4.6 Structured data type**6.4.4.6.1 General**

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

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

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

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

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

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

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

b) Declaration of a structured variable

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

c) Usage of structured variables in the ST language:

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

6.4.4.6.2 Initialization

The default values of the components of a structure are given by their individual data types.

The user can initialize the components of the structure with user-defined values. This initialization has priority.

The user can also initialize a previously defined structure using a list of assignments to the components of the structure. This initialization has a higher priority than the default initialization and the initialization of the components.

EXAMPLE Initialization of a structure

a) Declaration with initialization of a structured data type

```
TYPE
    ANALOG_SIGNAL_RANGE:
        (BIPOLAR_10V,
         UNIPOLAR_10V) := UNIPOLAR_10V;
    ANALOG_DATA: INT (-4095 .. 4095);
    ANALOG_CHANNEL_CONFIGURATION:
        STRUCT
            RANGE:      ANALOG_SIGNAL_RANGE;
            MIN_SCALE: ANALOG_DATA := -4095;
            MAX_SCALE: ANALOG_DATA := 4096;
        END_STRUCT;
    ANALOG_8BI_CONFIGURATION:
        ARRAY [1..8] OF ANALOG_CHANNEL_CONFIGURATION
            := [8((RANGE:= BIPOLAR_10V))];
END_TYPE
```

b) Declaration with initialization of a structured variable

```
VAR
    MODULE_CONFIG: ANALOG_CHANNEL_CONFIGURATION
        := (RANGE:= BIPOLAR_10V, MIN_SCALE:= -1023);
    MODULE_8_SMALL: ANALOG_8BI_CONFIGURATION
        := [8((MIN_SCALE:= -2047, MAX_SCALE:= 2048))];
END_VAR
```

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

6.4.4.7.1 General

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

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

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

The components of the structure shall not overlap in their memory layout, except if the keyword **OVERLAP** has been given in the declaration.

Overlapping of strings is beyond the scope of this standard.

NOTE Counting of bit offsets starts with 0 at the rightmost bit. Counting of byte offsets starts at the beginning of the structure with byte offset 0.

EXAMPLE Relative location and overlapping in a structure

TYPE

```
Com1_data: STRUCT
  head      AT %B0:      INT;           // at location 0
  length    AT %B2:      USINT:= 26;    // at location 2
  flag1     AT %X3.0:    BOOL;          // at location 3.0
  end       AT %B25:     BYTE;          // at 25, leaving a gap
END_STRUCT;

Com2_data: STRUCT OVERLAP
  head      AT %B0:      INT;           // at location 0
  length    AT %B2:      USINT;         // at location 2
  flag2     AT %X3.3:    BOOL;          // at location 3.3
  data1     AT %B5:      BYTE;          // at locations 5, overlapped
  data2     AT %B5:      REAL;          // at locations 5 to 8
  end       AT %B19:     BYTE;          // at 19, leaving a gap
END_STRUCT;

Com_data: STRUCT OVERLAP // C1 and C2 overlap
  C1 at %B0: Com1_data;
  C2 at %B0: Com2_data;
END_STRUCT;
END_TYPE
```

6.4.4.7.2 Initialization

Overlapped structures cannot be initialized explicitly.

6.4.4.8 Directly represented components of a structure – partly specified using “*”

The asterisk notation “*” in Table 11 can be used to denote not yet fully specified locations for directly represented components of a structure.

EXAMPLE Assigning of the components of a structure to not yet located inputs and outputs.

TYPE

```
HW_COMP: STRUCT;
  IN      AT %I*: BOOL;
  VAL     AT %I*: DWORD;
  OUT     AT %Q*: BOOL;
  OUT_VAR AT %Q*: WORD;
  ITNL_VAR: REAL; // not located
END_STRUCT;
```

END_TYPE

In the case that a directly represented component of a structure is used in a location assignment in the declaration part of a program, a function block type, or a class, an asterisk “*” shall be used in place of the size prefix and the unsigned integer(s) in the concatenation to indicate that the direct representation is not yet fully specified.

The use of this feature requires that the location of the structured variable so declared shall be fully specified inside the VAR_CONFIG...END_VAR construction of the configuration for every instance of the containing type.

Variables of this type shall not be used in a VAR_INPUT, VAR_IN_OUT, or VAR_TEMP section.

It is an error if any of the full specifications in the VAR_CONFIG...END_VAR construction is missing for any incomplete address specification expressed by the asterisk notation “*” in any instance of programs or function block types which contain such incomplete specifications.

6.4.4.9 Directly derived data type

6.4.4.9.1 General

A user-defined data type may be directly derived from an elementary data type or a previously user-defined data type.

This may be used to define new type-specific initial values.

EXAMPLE Directly derived data type

```

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

END_TYPE

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

```

6.4.4.9.2 Initialization

The default initial value is the initial value of the data type the new data type is derived from.

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

The user-defined initial value of the elements of structure can be declared in a parenthesized list following the data type identifier. Elements for which initial values are not listed in the initial value list shall have the default initial values declared for those elements in the original data type declaration.

EXAMPLE 1 User-defined data types - usage

Given the declaration of ANALOG_16_INPUT_DATA in Table 11
and the declaration VAR INS: ANALOG_16_INPUT_DATA; END_VAR
the variables INS[1] through INS[16] can be used anywhere a variable of type INT could be used.

EXAMPLE 2

Similarly, given the definition of Com_data in Table 11
and additionally the declaration VAR telegram: Com_data; END_VAR
the variable telegram.length can be used anywhere a variable of type USINT could be used.

EXAMPLE 3

This rule can also be applied recursively:
Given the declarations of ANALOG_16_INPUT_CONFIGURATION, ANALOG_CHANNEL_CONFIGURATION and ANALOG_DATA in Table 11
and the declaration VAR CONF: ANALOG_16_INPUT_CONFIGURATION; END_VAR
the variable CONF.CHANNEL[2].MIN_SCALE can be used anywhere that a variable of type INT could be used.

6.4.4.10 References

6.4.4.10.1 Reference declaration

A reference is a variable that shall only contain a reference to a variable or to an instance of a function block. A reference may have the value NULL, i.e. it refers to nothing.

References shall be declared to a defined data type using the keyword `REF_TO` and a data type – the reference data type. The reference data type shall already be defined. It may be an elementary data type or a user defined data type.

NOTE References without binding to a data type are beyond the scope of this part of IEC 61131.

EXAMPLE 1

```

TYPE
  myArrayType:      ARRAY[0..999] OF INT;
  myRefArrType:     REF_TO myArrayType;           // Definition of a reference
  myArrOfRefType:   ARRAY [0..12] OF myRefArrType; // Definition of an array of refer-
  ences
END_TYPE

VAR
  myArray1:         myArrayType;
  myRefArr1:        myRefArrType;                 // Declararion of a reference
  myArrOfRef:       myArrOfRefType;               // Declararion of an array of refer-
  ences
END_VAR

```

The reference shall reference only variables of the given reference data type. References to data types which are directly derived are treated as aliases to references to the base data type. The direct derivation may be applied several times.

EXAMPLE 2

```

TYPE
  myArrType1:  ARRAY[0..999] OF INT;
  myArrType2:  myArrType1;
  myRefType1:  REF_TO myArrType1;
  myRefType2:  REF_TO myArrType2;
END_TYPE
myRefType1 and myRefType2 can reference variables of type ARRAY[0..999] OF INT and of the derived data
types.

```

The reference data type of a reference can also be a function block type or a class. A reference of a base type can also refer to instances derived from this base type.

EXAMPLE 3

```

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

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

```

References of type `myRefF1` can reference instances of class `F1` and `F2` and derivations of both. Where references of `myRefF2` cannot reference instances of `F1`, only instances of `F2` and derivations of it, because `F1` may not support methods and variables of the extended class `F2`.

6.4.4.10.2 Initialization of references

References can be initialized using the value `NULL` (default) or the address of an already declared variable, instance of a function block or class.

EXAMPLE

```

FUNCTION_BLOCK F1 ... END_FUNCTION_BLOCK;

VAR
  myInt:  INT;
  myRefInt: REF_TO INT:= REF(myInt);
  myF1:  F1;
  myRefF1: REF_TO F1:= REF(myF1);
END_VAR

```

6.4.4.10.3 Operations on references

The `REF()` operator returns a reference to the given variable or instance. The reference data type of the returned reference is the data type of the given variable. Applying the `REF()` operator to a temporary variable (e.g. variables of any `VAR_TEMP` section and any variables inside functions) is not permitted.

A reference can be assigned to another reference if the reference data type is equal to the base type or is a base type of the reference data type of the assigned reference.

References can be assigned to parameters of functions, function blocks and methods in a call if the reference data type of the parameter is equal to the base type or is a base type of the reference data type. References shall not be used as in-out variables.

If a reference is assigned to a reference of the same data type, then the latter references the same variable. In this context, a directly derived data type is treated like its base data type.

If a reference is assigned to a reference of the same function block type or of a base function block type, then this reference references the same instance, but is still bound to its function block type; i.e. can only use the variables and methods of its reference data type.

Dereferencing shall be done explicitly.

A reference can be dereferenced using a succeeding '^' (caret).

A dereferenced reference can be used in the same way as using a variable directly.

Dereferencing a `NULL` reference is an error.

NOTE 1 Possible checks of `NULL` references can be done at compile time, by the runtime system, or by the application program.

The construct `REF()` and the dereferencing operator '^' shall be used in the graphical languages in the definition of the operands.

NOTE 2 Reference arithmetic is not recommended and is beyond the scope of this part of IEC 61131.

EXAMPLE 1

```

TYPE
S1: STRUCT
  SC1: INT;
  SC2: REAL;
END_STRUCT;
A1: ARRAY[1..99] OF INT;
END_TYPE

VAR
  myS1: S1;
  myA1: A1;
  myRefS1: REF_TO S1:= REF(myS1);
  myRefA1: REF_TO A1:= REF(myA1);
  myRefInt: REF_TO INT:= REF(myA1[1]);
END_VAR

myRefS1^.SC1:= myRefA1^[12];    // in this case, equivalent to S1.SC1:= A1[12];
myRefInt:= REF(A1[11]);

S1.SC1:= myRefInt^;              // assigns the value of A1[11] to S1.SC1

```

EXAMPLE 2

Graphical representation of the statements of Example 1

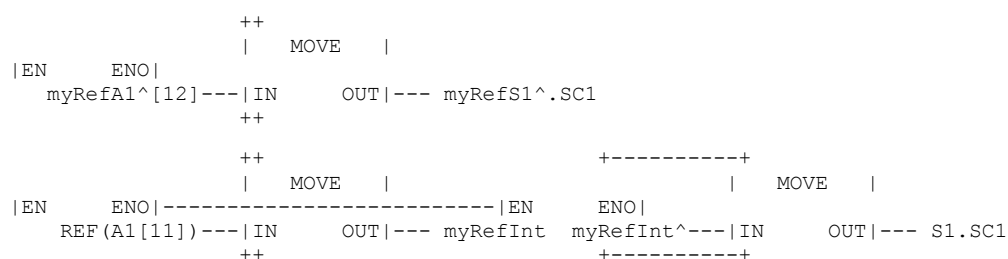


Table 12 defines the features for reference operations.

Table 12 – Reference operations

No	Description	Example
	Declaration	
1	Declaration of a reference type	<pre> TYPE myRefType: REF_TO INT; END_TYPE </pre>
	Assignment and comparison	
2a	Assignment reference to reference	<pre> <reference>:= <reference> myRefType1:= myRefType2; </pre>
2b	Assignment reference to parameter of function, function block and method	<pre> myFB (a:= myRefS1); </pre> The types shall be equal!
2c	Comparison with NULL	<pre> IF myInt = NULL THEN ... </pre>
	Referencing	
3a	<pre>REF(<variable>)</pre> Provides of the typed reference to the variable	<pre> myRefA1:= REF (A1); </pre>

No	Description	Example
3b	REF(<function block instance>) Provides the typed reference to the function block or class instance	myRefFB1:= REF(myFB1)
	Dereferencing	
4	<reference>^ Provides the content of the variable or the content of the instance to which the reference variable contains the reference	myInt:= myA1Ref^[12];

6.5 Variables

6.5.1 Declaration and initialization of variables

6.5.1.1 General

The variables provide a means of identifying data objects whose contents may change, for example, data associated with the inputs, outputs, or memory of the programmable controller.

In contrast to the literals which are the external representations of data, variables may change their value over time.

6.5.1.2 Declaration

Variables are declared inside of one of the variable sections.

A variable can be declared using

- an elementary data type or
- a previously user-defined type or
- a reference type or
- an instantly user-defined type within the variable declaration.

A variable can be

- a single-element variable, i.e. a variable whose type is either
 - an elementary type or
 - a user-defined enumeration or subrange type or
 - a user-defined type whose “parentage”, defined recursively, is traceable to an elementary, enumeration or subrange type.
- a multi-element variable, i.e. a variable which represents an ARRAY or a STRUCT
- a reference, i.e. a variable that refers to another variable or function block instance.

A variable declaration consists of

- a list of variable names which are declared
- a “:” (colon) and
- a data type with an optional variable-specific initialization.

EXAMPLE

```

TYPE
  myType: ARRAY [1..9] OF INT;    // previously user-defined data type
END_TYPE

```

```

VAR
  myVar1, myVar1a: INT;           // two variables using an elementary type
  myVar2: myType;                 // using a previously user-defined type
  myVar3: ARRAY [1..8] OF REAL;   // using an instantly user-defined type
END_VAR

```

6.5.1.3 Initialization of variables

The default initial value(s) of a variable shall be

1. the default initial value(s) of the underlying elementary data types as defined in Table 10,
2. NULL, if the variable is a reference,
3. the user-defined value(s) of the assigned data type;
this value is optionally specified by using the assignment operator “:=” in the TYPE declaration defined in Table 11,
4. the user-defined value(s) of the variable;
this value is optionally specified by using the assignment operator “:=” in the VAR declaration (Table 14).

This user-defined value may be a literal (e.g. -123, 1.55, “abc”) or a constant expression (e.g. 12*24).

Initial values cannot be given in VAR_EXTERNAL declarations.

Initial values can also be specified by using the instance-specific initialization feature provided by the VAR_CONFIG...END_VAR construct. Instance-specific initial values always override type-specific initial values.

Table 13 – Declaration of variables

No.	Description	Example	Explanation
1	Variable with elementary data type	<pre> VAR MYBIT: BOOL; OKAY: STRING[10]; VALVE_POS AT %QW28: INT; END_VAR </pre>	Allocates a memory bit to the Boolean variable MYBIT. Allocates memory to contain a string with a maximum length of 10 characters.
2	Variable with user-defined data type	<pre> VAR myVAR: myType; END_VAR </pre>	Declaration of a variable with a user data type.
3	Array	<pre> VAR BITS: ARRAY[0..7] OF BOOL; TBT: ARRAY [1..2, 1..3] OF INT; OUTA AT %QW6: ARRAY[0..9] OF INT; END_VAR </pre>	
4	Reference	<pre> VAR myRefInt: REF_TO INT; END_VAR </pre>	Declaration of a variable to be a reference