



BSI Standards Publication

Industrial-process measurement and control — Programmable controllers

Part 8: Guidelines for the application and implementation of programming languages

National foreword

This Published Document is the UK implementation of IEC TR 61131-8:2017. It supersedes PD IEC TR 61131-8:2003, which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee GEL/65/2, Elements of systems.

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

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

© The British Standards Institution 2018
Published by BSI Standards Limited 2018

ISBN 978 0 580 95881 6

ICS 25.040.40; 35.240.50

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

This Published Document was published under the authority of the Standards Policy and Strategy Committee on 28 February 2018.

Amendments/corrigenda issued since publication

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

TECHNICAL REPORT



Industrial-process measurement and control – Programmable controllers – Part 8: Guidelines for the application and implementation of programming languages

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

ICS 25.040.40; 25.240.50

ISBN 978-2-8322-4898-0

Warning! Make sure that you obtained this publication from an authorized distributor.

CONTENTS

FOREWORD	6
INTRODUCTION	8
1 Scope	9
2 Normative references	9
3 Terms and definitions	9
4 Abbreviated terms	9
5 Overview	10
6 Introduction to IEC 61131-3	10
6.1 General considerations	10
6.2 Overcoming historical limitations	13
6.3 Basic features in IEC 61131-3	13
6.4 Language items overview	14
6.5 Changes from IEC 61131-3:2003 (edition 2) to IEC 61131-3:2013 (edition 3)	16
6.6 Software engineering considerations	17
6.6.1 Application of software engineering principles	17
6.6.2 Portability	19
7 Application guidelines	20
7.1 Use of data types	20
7.1.1 Type selection	20
7.1.2 Type versus variable initialization	20
7.1.3 Use of enumerated and subrange types	21
7.1.4 Use of BCD data	22
7.1.5 Use of REAL data types	22
7.1.6 Use of character string data types	23
7.1.7 Use of character data types	24
7.1.8 Use of time data types	24
7.1.9 Declaration and use of multi-element variables	26
7.1.10 Use of bit-string variables	26
7.1.11 Use of partial accessing of bitstring variables	27
7.1.12 Type assignment	27
7.2 Data passing over POU	27
7.2.1 General	27
7.2.2 External variables	29
7.2.3 In-out (VAR_IN_OUT) variables	29
7.2.4 Formal and non-formal invocations and argument lists	32
7.2.5 Assignment of input, output, and in-out variables of programs	34
7.3 Use of function blocks	35
7.3.1 Function block types and instances	35
7.3.2 Scope of data within function blocks	35
7.3.3 Function block access and invocation	36
7.4 Differences between function block instances and functions	37
7.5 Use of indirectly referenced function block instances	37
7.5.1 General	37
7.5.2 Establishing an indirect function block instance reference	38
7.5.3 Access to indirectly referenced function block instances	40

7.5.4	Invocation of indirectly referenced function block instances	41
7.5.5	Recursion of indirectly referenced function block instances	43
7.5.6	Execution control of indirectly referenced function block instances	44
7.5.7	Use of indirectly referenced function block instances in functions	44
7.6	Use of programs	44
7.6.1	Difference to function block	44
7.6.2	Communication with other programs	44
7.7	Object orientation	45
7.7.1	General introduction	45
7.7.2	Usage of methods.....	45
7.7.3	Usage of class variable.....	49
7.7.4	Usage of inheritance.....	50
7.7.5	Usage of override	53
7.7.6	Usage of interfaces.....	53
7.8	Recursion within programmable controller programming languages	54
7.9	Multiple invocations of a function block instance	54
7.10	Language specific features	55
7.10.1	Edge-triggered functionality	55
7.10.2	Edge-triggering in LD language	55
7.10.3	Use of edge-triggered function blocks.....	56
7.10.4	Use of EN/ENO in functions and function blocks.....	56
7.10.5	Language selection.....	57
7.11	Namespaces	58
7.11.1	General	58
7.11.2	Usage of global namespace.....	58
7.11.3	Usage of INTERNAL	58
7.12	Use of SFC elements	59
7.12.1	General	59
7.12.2	Action control	59
7.12.3	Boolean actions	60
7.12.4	Non-SFC actions	64
7.12.5	SFC actions.....	65
7.12.6	SFC function blocks.....	66
7.13	Scheduling, concurrency and synchronization mechanisms.....	67
7.13.1	Operating system issues.....	67
7.13.2	Task scheduling.....	68
7.13.3	Semaphores	69
7.13.4	Messaging	70
7.13.5	Time stamping	70
7.14	Communication facilities in ISO/IEC 9506-5 and IEC 61131-5.....	71
7.14.1	Overview	71
7.14.2	Data representation	71
7.14.3	Communication channels.....	73
7.14.4	Reading and writing variables.....	73
7.14.5	Communication function blocks.....	74
7.15	Deprecated programming practices.....	75
7.15.1	General	75
7.15.2	Global variables.....	75
7.15.3	Jumps in FBD/LD language	75

7.15.4	Dynamic modification of task properties	75
7.15.5	Execution control of function block instances by tasks	76
7.15.6	WHILE and REPEAT constructs for interprocess synchronisation	76
7.15.7	Expecting programs associated with a task to be executed sequentially	77
7.16	REAL_TO_INT conversion functions	78
7.17	Implementation dependant parameters	78
8	Implementation guidelines	80
8.1	General.....	80
8.2	Resource allocation	80
8.3	Implementation of data types	80
8.3.1	REAL and LREAL data types	80
8.3.2	Bit strings	81
8.3.3	Character strings	81
8.3.4	Time data types	81
8.3.5	Multi-element variables	82
8.4	Execution of functions and function blocks	83
8.4.1	General	83
8.4.2	Functions.....	83
8.4.3	Function blocks	83
8.5	Object oriented features.....	84
8.5.1	Classes	84
8.5.2	Methods	84
8.5.3	Dynamic binding and virtual method table.....	85
8.5.4	Interfaces	85
8.6	Implementation of SFCs.....	85
8.6.1	General considerations	85
8.6.2	SFC evolution.....	85
8.6.3	SFC analysis	86
8.7	Task scheduling.....	87
8.7.1	General	87
8.7.2	Classification of tasks	88
8.7.3	Task priorities.....	88
8.8	Error handling	88
8.8.1	Error-handling mechanisms	88
8.8.2	Run-time error-handling procedures.....	90
8.9	System interface	92
8.10	Compliance.....	92
8.10.1	General	92
8.10.2	Compliance statement	92
8.10.3	Compliance testing	92
Annex A (informative)	Relationships to other standards	93
INDEX		94
Bibliography		101
Figure 1 – A distributed application		11
Figure 2 – Stand-alone applications		12
Figure 3 – Cyclic or periodic scanning of a program		13
Figure 4 – Programming Modelm		14

Figure 5 – Software Modelm	16
Figure 6 – ST example of time data type usage	26
Figure 7 – Example of declaration and use of array types	26
Figure 8 – Examples of VAR_IN_OUT usage	32
Figure 9 – Hiding of function block instances	36
Figure 10 – Graphical use of a function block name	40
Figure 11 – Access to an indirectly referenced function block instance	40
Figure 12 – Invocation of an indirectly referenced function block instance.....	43
Figure 13 – Standard FB CTUD according to IEC 61131-3	46
Figure 14 – Functionblock CTUD object oriented version	47
Figure 15 – Call of standard and OO-Functionblock in ST	48
Figure 16 – Call of standard function block in FBD.....	48
Figure 17 – All of a method in FBD	48
Figure 18 – Synthesis: a traditional function block derived from an OO-function block	49
Figure 19 – Use of function blocks derived from ETrig-Base, the base function block interface is highlighted	53
Figure 20 – Timing of edge triggered functionality.....	56
Figure 21 – Execution control example	57
Figure 22 – Timing of Boolean actions	64
Figure 23 – Example of a programmed non-Boolean action.....	64
Figure 24 – Use of the pulse (P) qualifier	65
Figure 25 – An SFC function block.....	66
Figure 26 – Example of incorrect and allowed programming constructs.....	77
Figure 27 – Reduction steps	86
Figure 28 – Reduction of SFCs	87
Table 1 – Available data passing of variables to POU	28
Table 2 – Examples of textual invocations of functions and function blocks.....	33
Table 3 – Characteristics of the languages	58
Table 4 – Differences between multi-user and real-time systems	68
Table 5 – Implementation-dependent parameters.....	78
Table 6 – Recommended run-time error-handling mechanisms	89

INTERNATIONAL ELECTROTECHNICAL COMMISSION

**INDUSTRIAL-PROCESS MEASUREMENT AND CONTROL –
PROGRAMMABLE CONTROLLERS –****Part 8: Guidelines for the application
and implementation of programming languages**

FOREWORD

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

The main task of IEC technical committees is to prepare International Standards. However, a technical committee may propose the publication of a technical report when it has collected data of a different kind from that which is normally published as an International Standard, for example "state of the art".

IEC 61131-8, which is a technical report, has been prepared by subcommittee 65B: Measurement and control devices, of IEC technical committee 65: Industrial-process measurement, control and automation.

This third edition cancels and replaces the second edition published in 2003. This edition constitutes a technical revision.

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

This third edition is a compatible extension of the second edition. The main extensions are new data types and conversion functions, references, name spaces and the object oriented features of classes and function blocks (see listing in Annex B of IEC 61131-3:2013).

The text of this technical report is based on the following documents:

DTR	Report on voting
65B/1058/DTR	65B/1073/RVDTR

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

This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

A list of all parts in the IEC 61131 series, published under the general title *Industrial-process measurement and control – Programmable controllers*, can be found on the IEC website.

Future standards in this series will carry the new general title as cited above. Titles of existing standards in this series will be updated at the time of the next edition.

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

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

A bilingual version of this publication may be issued at a later date.

IMPORTANT – The 'colour inside' logo on the cover page of this publication indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.

INTRODUCTION

This part of IEC 61131 is being issued as a technical report in order to provide guidelines for the implementation and application of the programming languages defined in IEC 61131-3:2013.

The content of this document answers a number of frequently asked questions about the intended application and implementation of the normative provisions of IEC 61131-3.

INDUSTRIAL-PROCESS MEASUREMENT AND CONTROL – PROGRAMMABLE CONTROLLERS –

Part 8: Guidelines for the application and implementation of programming languages

1 Scope

This part of IEC 61131, which is a technical report, applies to the programming of programmable controller systems using the programming languages defined in IEC 61131-3. The scope of IEC 61131-3 is applicable to this part.

This document provides

- a) guidelines for the application of IEC 61131-3,
- b) guidelines for the implementation of IEC 61131-3 languages for programmable controller systems,
- c) programming and debugging tool (PADT) recommendations.

For further information IEC 61131-4 describes other aspects of the application of programmable controller systems, e.g. electromagnetic compatibility or functional safety.

NOTE Neither IEC 61131-3 nor this document explicitly addresses safety issues of programmable controller systems or their associated software. The various parts of IEC 61508 can be consulted for such considerations.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC 61131-3:2013, *Programmable controllers – Part 3: Programming languages*

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

3 Terms and definitions

No terms and definitions are listed in this document.

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

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

4 Abbreviated terms

ED	Early Detection
ETrig	Edge triggered function
EW	Early Warning

FB	Function Block
FBD	Function Block Diagram
FC	Function
IL	Instruction List
LD	Ladder Diagram
MMS	Manufacturing Message Specification
NaN	Not a number
OO	Object Orientation
OOP	Object Oriented Programming
PLC	Programmable Logic Controller
POU	Program Organization Unit
PADT	Programming And Debugging Tool
RT	RunTime
SFC	Sequential Function Chart
ST	Structured Text
VMD	Virtual Manufacturing Device

5 Overview

The intended audience for this document consists of

- users of programmable controller systems as defined in IEC 61131-3, who shall program, configure, install and maintain programmable controllers as part of industrial-process measurement and control systems; and
- implementers of programming and debugging tools (PADT) as defined in IEC 61131-3, for programmable controller systems. This can include vendors of software and hardware for the preparation and maintenance of programs for these systems, as well as vendors of the programmable controller systems themselves.

IEC 61131-3 is mainly oriented toward the implementers of programming languages for programmable controllers. Users who wish a general introduction to these languages and their application should consult any of several generally available textbooks on this subject.

Clause 6 of this document provides a general introduction to IEC 61131-3, while Clause 7 provides complementary information about the application of some of the programming language elements specified in IEC 61131-3. Clause 8 provides information about the intended implementation of some of these programming language elements.

Hence, it is expected that users of programmable controllers will find Clauses 6 and 7 of this part most useful, while programming language implementers will find Clause 8 more useful, referring to the background material in Clauses 6 and 7 as necessary.

6 Introduction to IEC 61131-3

6.1 General considerations

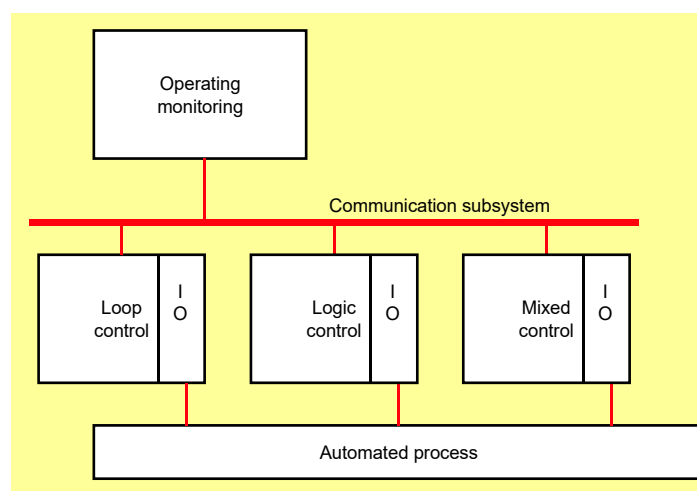
In the time before the IEC 61131-3 was existing, the limited capabilities of expensive hardware components imposed severe constraints on the design process for industrial-process control, measurement and automation systems. Software design and implementation were tightly tailored to the selected hardware. This required specialists who were highly skilled, both in solving process automation problems and in dealing with complicated, often hardware-specific computer programming constructs.

With the rapid innovation in microelectronics and related technologies, the cost/performance ratio of system hardware has decreased dramatically. At present, a programmable controller can cost many times less than the cost of programming it.

Driven by rapidly decreasing hardware cost, a trend has become established of replacing large, centrally installed process computers or other comparatively large, isolated controllers by systems with spatially and functionally distributed parts.

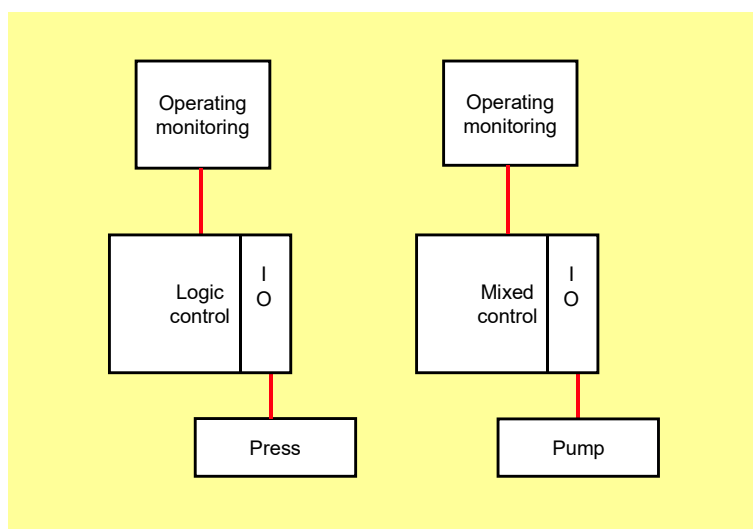
As illustrated in Figure 1, the essential backbone of such systems is the communication subsystem, which provides the mechanism for information exchange between the distributed automating devices. Connected to this backbone are the devices, such as programmable controllers, which deliver the distributed processing power of the system. Each device, under the control of its own software, performs a dedicated subtask to achieve the required overall system functionality. Each device is chosen with the size and performance required to meet the demands of its particular subtask.

In a different environment, programmable controllers are used in stand-alone applications as illustrated in Figure 2. Users of these applications also stand to gain by the evolution outlined above. Due to the present low cost of hardware components, many new, relatively small, automation tasks can be solved profitably and flexibly by programmable controllers.



IEC

Figure 1 – A distributed application



IEC

Figure 2 – Stand-alone applications

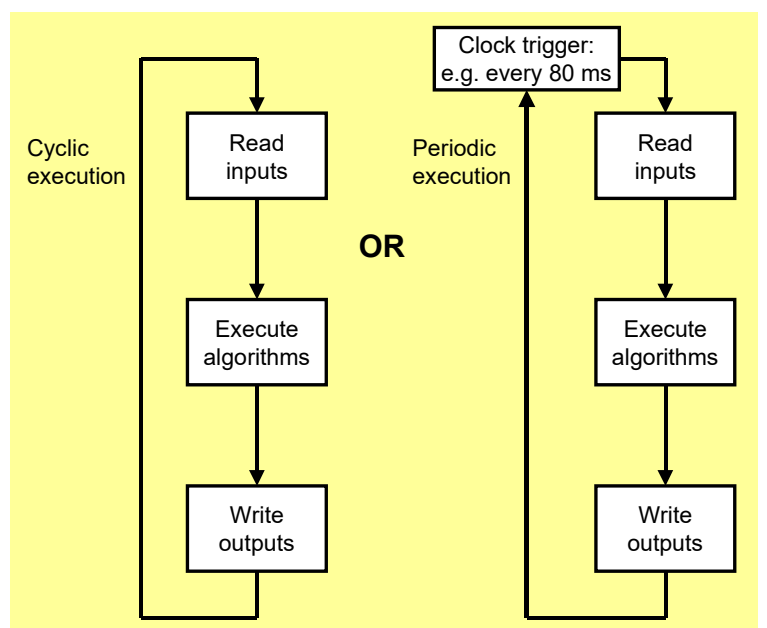
In addition to their low hardware price, the intensive use of programmable controllers in solving automation tasks is also advanced by their straightforward operating and programming principles, which are easily understood and applied by the shop-floor personnel involved in programming, operation and maintenance.

Programmable controllers typically employ the principles of cyclic or periodic program execution illustrated in Figure 3. Cyclically running programs restart execution as fast as possible after they have terminated execution, in IEC 6113-3 such programs are not assigned to a task and executed with the lowest priority. Periodic execution of a program is triggered by a clock mechanism at equidistant points in time. The same controller may execute such programs quasi-simultaneous with different periodic times. Another kind (defined by IEC 61131-3 but not shown in Figure 3) is the non-periodic execution, which is based on events and will be executed upon each occurrence of such events. These principles are well known and applied in the operation of digital signal processing systems to simulate the operation of continuously operating analogue or electromechanical systems. Process values are read into the device and written out to the process as discrete samples at random or equidistant points in time, depending on the control task that has to be fulfilled.

The advantage of these operating principles is that they allow the construction of programs for programmable controllers using elements closely related to the principles of hard-wired logic or continuous control circuits previously used for the same purpose.

The operating principles of programmable controllers thus enable the provision of application-specific, graphical programming languages. Combined with appropriate man-machine interfaces, these languages enable the control engineer to concentrate on solving the problems of the application, without extensive training in software engineering. The control engineer's technological specifications can be mapped directly to the corresponding language elements.

Another particular advantage of such programming languages is that the representation they offer can be used not only for program input and documentation, but also for on-line test and diagnosis as well. Thus, programming and debugging tools (PADT) for programmable controllers are able to provide the graphically oriented representation and documentation that are already familiar to the application engineer and shop-floor personnel.



IEC

Figure 3 – Cyclic or periodic scanning of a program

6.2 Overcoming historical limitations

Automation system designers are often required to use programmable controllers from various manufacturers in different automation systems, or even in the same system. However, the hardware of programmable controllers from different manufacturers may have very little in common. This has resulted in significant differences in the elements and methods of programming the software as well and has led to the development of manufacturer-specific programming and debugging tools, which generally carried very specialized software for programming, testing and maintaining particular controller “families”.

Changing from one controller family to another often required the designer to read large manuals for both the hardware and software of the new family. Often, the manual had to be reviewed several times in order to understand the exact meaning and to use the new controller family in an appropriate way. Due to the concentrated, tedious work necessary to read and understand the new, vendor-specific material, few people did it. For this reason, many people regarded the design and the programming of such controllers as some black magic to be avoided. Thus, the knowledge of how to use such systems effectively was concentrated in one or a few specialists and could not be transferred effectively to those responsible for system operation, maintenance, and upgrade.

A major goal of IEC 61131-3 was to remove such barriers to the understanding and application of programmable controllers. Thus, IEC 61131-3 introduced numerous facilities to support the advantages of programmable controllers described in 6.1, even if controllers of different vendors are concerned. It has turned out that the resulting expansion of the application domains of programmable controllers, and the increasing demand of customers fed through this expansion, stimulated a lot of vendors to make their programming systems compliant to the standard.

Vendor and user organizations like PLCopen accelerated this process by promoting the benefits and advantages of standardizing PLC programming to a large extent.

6.3 Basic features in IEC 61131-3

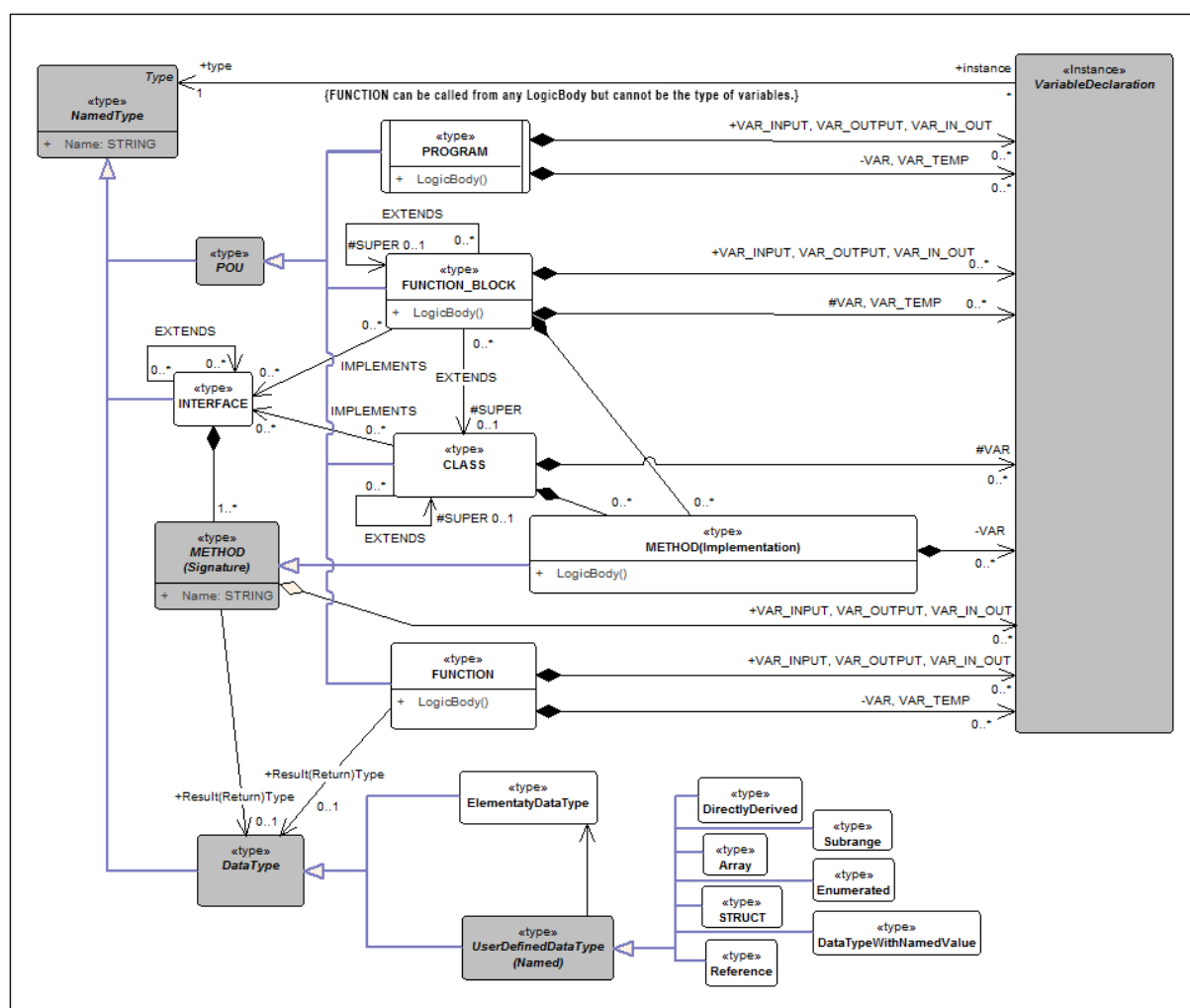
From the point of view of the application engineer and the control systems configurator, the most important features introduced by IEC 61131-3 can be summarized as follows.

- a) Well-structured, “top-down” or “bottom-up” program development is facilitated by language constructs for the definition of typed functional objects (program organization units) such as functions, function blocks, classes and programs.
- b) Strong data typing is not only supported but inherently required, thus eliminating a major source of programming errors.
- c) A sufficient set of features for the execution control of program organization units is included; those features associated with steps, transitions and action blocks offer excellent means to represent complicated sequential control solutions in a concise form.
- d) The functionality for designing the communication using the VAR_ACCESS keyword between application programs is provided. Independent of the mapping of programs onto a single device or different devices, identical communication features can be used between two programs. This facilitates the reuse of software in different environments.
- e) Graphical or textual languages may be chosen by the designer, according to the requirements of the application. These languages, plus a set of textual and graphical common elements, support software design methodologies based on well-understood models.
 - 1) The graphical Ladder Diagram (LD) language models networks of simultaneously functioning electromechanical elements such as relay contacts and coils, timers, counters, etc.
 - 2) The graphical Function Block Diagram (FBD) language models networks of simultaneously functioning electronic elements such as adders, multipliers, shift registers, gates, etc.
 - 3) The Structured Text (ST) language models typical information processing tasks such as numerical algorithms using constructs found in general-purpose high level languages such as Pascal.
 - 4) The Instruction List (IL) language models the low-level programming of control systems in assembly language.
In IEC 61131-3:2013, IL is marked as deprecated for the next release, because an assembler like language is not up-to-date in modern development environments.
 - 5) A set of graphical and textual common elements provides rules for defining values and variables, features for software configuration and object declaration. The common elements include graphical and textual elements for the construction of program organization units.
 - 6) Sequential Function Charts (SFCs), which model time- and event-driven sequential control devices and algorithms.
- f) Flexibility in the selection of languages suited for programming application-specific functionalities will increase the reuse of software solutions to process control problems.
- g) Each application specialist on a project team can use a programming style and language suited for the particular functionality in question, with the assurance that the results of the work of the individual specialists will integrate smoothly together.

In summary, the principal goal of IEC 61131-3 is to introduce all the necessary standardized language concepts and constructs to solve the technological problems of each application and to provide principles for the construction of vendor-independent software elements. This facilitates the reusability of control software designs for different controller types, even though some effort will almost always be required in order to move control programs from one controller family to another.

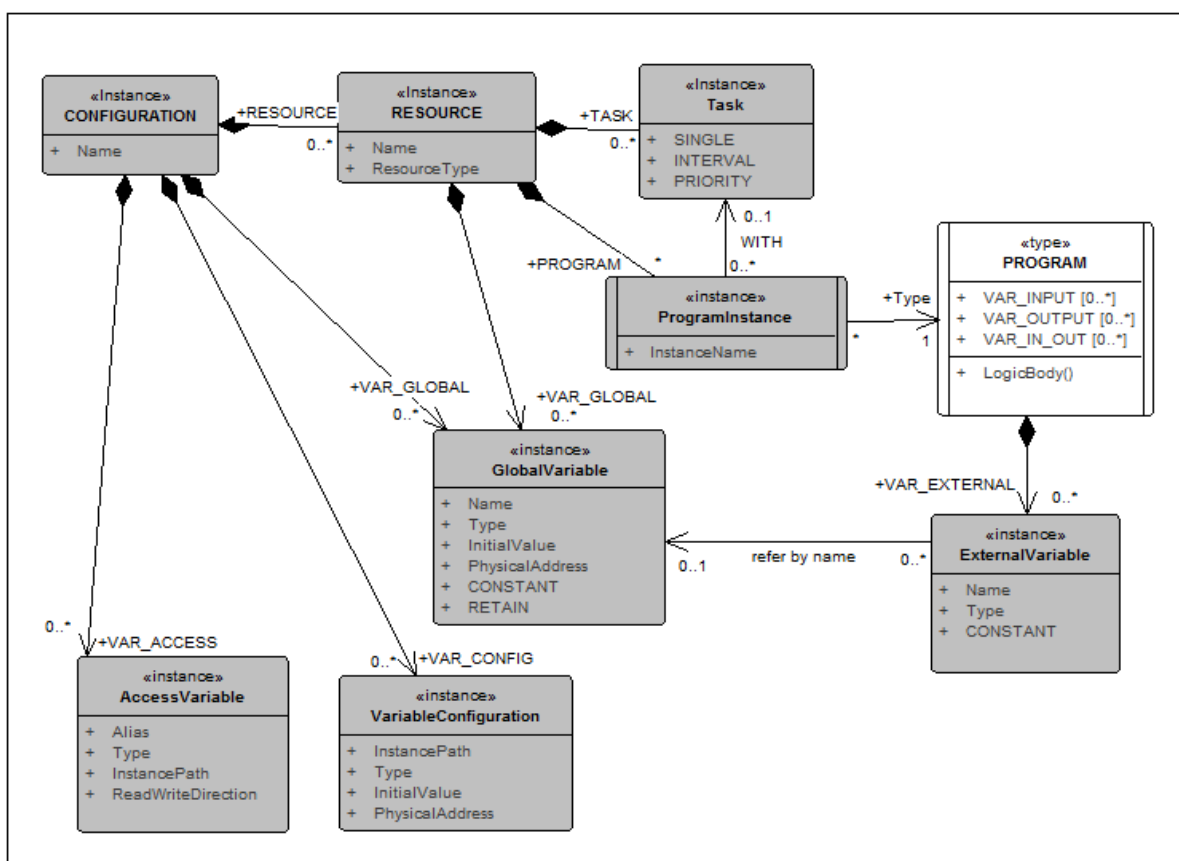
6.4 Language items overview

The basic high-level language elements for programming and their relationship are shown in the programming model shown in Figure 4 and Figure 5.



IEC

Figure 4 – Programming Modelm



IEC

Figure 5 – Software Modelm

6.5 Changes from IEC 61131-3:2003 (edition 2) to IEC 61131-3:2013 (edition 3)

One of the major changes is the extension of the basic concept of the programming language by supporting object oriented features beyond FB instantiation. Major object oriented benefits in IEC 61131-3 are:

- Better structuring of the program
- Extension by reusing existing FBs and Classes
- Maintenance and extensions

The changes made to IEC 61131-3:2003 are described in IEC 61131-3:2013, Annex B.

- Object oriented extensions
 - Introducing CLASS and METHOD.
 - Introducing INTERFACE element and IMPLEMENTS declaration by POU.
 - Overriding body and method of the base POU by EXTENDS and OVERRIDE keyword
 - Introducing access specifier PUBLIC / INTERNAL / PROTECTED / PRIVATE / FINAL for internal variables of CLASS and FUNCTION_BLOCKS
 - Extending FUNCTION_BLOCK to support these OO extensions.
- Introducing NAMESPACE
 - Declaring NAMESPACE in type definitions.
 - USING direction
 - Access specifier INTERNAL for the contents of the namespace.
- Addition of some standard data types.

- 64 bit time types (LTIME, LTOD, LDT),
- Character type (CHAR, WCHAR)
- Addition of some standard functions and function blocks
 - Functions to operate newly added data types above
 - Typed TRUNC, endian conversion, validation, ATAN2, etc.
- Extension of user-defined data types
 - Reference type, i.e. typed pointer without arithmetic
 - Variable-length array for the type of input / in_out variables of POU
 - Data type with named values
 - Data type with explicit layout
 - Partial access to ANY_BIT variables
- Miscellaneous
 - Addition of some commenting notations. (//, /* */, Nesting comment permitted.)
 - Addition of compare contact in ladder diagram
 - Addition of CONTINUE statement in structured text (ST) language
 - Function without result become permitted
 - Recursive call of POU become implementer specific
 - Implicit / explicit type conversion rules are clarified

6.6 Software engineering considerations

6.6.1 Application of software engineering principles

6.6.1.1 Encapsulation and hiding

A number of software engineering principles were employed in the development of IEC 61131-3 in order to promote increased software quality. A few of the more important principles, their contributions to software quality, and their embodiment in IEC 61131-3 are discussed below.

NOTE See Annex A of this document for descriptions of the software engineering quality measures referenced in 6.6.1, for example, reliability, maintainability, etc.

Encapsulation is the “packaging” of functionally related data and/or procedures in a single software entity. Encapsulation contributes to software reliability, maintainability, usability, and adaptability.

The notion of hiding of procedures and data is associated with encapsulation. “Hiding” means that the user does not get information about the internal data structure and procedure implementation of a software entity. He only gets information about its external interface and specified functionality. Hiding contributes to software maintainability, integrity, usability, portability and reusability.

IEC 61131-3 supports encapsulation and hiding by the following elements:

- Program;
- Functionblock;
- Class;
- Interface;
- Function;
- Method;
- Datatypes.

NAMESPACE defined in IEC 61131-3 can be used to avoid naming conflicts among type declarations. Elements of a NAMESPACE can be hidden against outside access with access specifier INTERNAL.

6.6.1.2 Explicit representation of state

The SFC elements defined in IEC 61131-3 enable the state of the control system to be determined at any point in time as the set of active steps and actions. Without this representation, the state of the system shall be inferred from data such as system inputs, outputs, and some set of “state” (Boolean) variables. The use of SFC elements thus contributes to software maintainability, usability and portability. Furthermore, system responsiveness and processing capacity are enhanced by performing only those portions of the software algorithm relevant to the current state.

6.6.1.3 Mapping to the application domain

The direct mapping of the elements of IEC 61131-3 to well-understood concepts in industrial-process measurement, automation and control has already been noted in 6.1 and 6.3. This characteristic contributes to software maintainability, usability and adaptability.

6.6.1.4 Mapping of design to application

IEC 61131-3 supports a style of system realization known as “top-down design” (or “design by functional decomposition”) followed by “bottom-up implementation” (or “implementation by functional composition”). This contributes to software reliability, maintainability, usability and adaptability.

This style of system design and implementation is characterized by the following sequence of steps.

- a) The desired functionality and external interface of the system are specified. For instance, the basic functionality of a machining cell might be to accept a rough part from a material handling system, produce a finished part from the rough blank, measure the finished part, pass it back to the material handling system, and report the results of the operation to a manufacturing information system. External interfaces in this cell would include the material handling and information system interfaces. The configuration elements described in IEC 61131-3 can be used in this step.
- b) A first decomposition of the system design is made by allocating the required functionality into one or more elements, typically programs. The interfaces among the programs, and between the programs and the external interfaces of the system, are defined, and the functionality allocated to each program is defined. Such a decomposition will often follow the physical partitioning of the system; for instance, in the machining cell described above, separate programs might be defined for the machining station, the measuring station, and for the material handling robot of the cell, if any.
- c) Each element defined in the preceding step is further decomposed into more basic functional units. If the functionality of the element is essentially sequential, the first step in the decomposition may be the formulation of an SFC expressing the sequence of operations to be performed and the conditions for repeating the cycle of operations. Each action of the SFC is then further decomposed, typically into interconnected function blocks, i.e., an FBD. For instance, the main program for the machining station in the cell described above may be an SFC describing the sequence of machining operations to be performed, while the actions might contain function blocks performing the required motion control functions.
- d) This functional decomposition process is performed recursively until all functionality can be identified as belonging to existing library elements, or can be expressed algorithmically in one of the textual or graphic languages in IEC 61131-3.
- e) The system is then implemented by “bottom-up” functional composition, i.e., by compiling and adding to the library the newly defined elements in the reverse order in which they

have been defined in the preceding steps. With some attention to design for reusability, many of the new library elements may be usable in future system designs.

- f) Finally, the allocation of programs to resources, and resources to configurations is completed, program execution tasks are set up, and access paths are established for communication with information systems, using the configuration elements defined in IEC 61131-3.

6.6.1.5 Software reuse

The programming model described in IEC 61131-3 strongly supports the reuse of software elements. These can be developed by the user in the “bottom-up” implementation process or can be supplied as “libraries” by software vendors. This method of system construction can be unfamiliar to users who have previously developed automation system applications as a single, large ladder diagram. Hence, some training can be required in order to realize the large potential gains in software quality and productivity presented by the new approach to software reuse presented in IEC 61131-3.

As described in IEC 61131-3, the software elements that can be placed in “libraries” for reuse include, in order of increasing complexity and functionality:

- data types;
- functions;
- function blocks;
- classes;
- interfaces;
- programs;
- configurations,
- resources.

6.6.2 Portability

6.6.2.1 Inter-language portability

As noted in Annex A of this document, portability is defined as the ease with which system functionality can be moved from one system to another. This may be considered from the point of view of

- inter-language portability, i.e. the ease of converting a program organization unit type specification from one language to another; or
- inter-system portability, i.e. the ease of converting a program organization unit type specification from one programming and debugging tool (PADT) to another.

As noted in 6.6.1.1, the encapsulation and hiding facilities of IEC 61131-3 provide a high degree of reusability of functions, function blocks, and data types among all the defined languages. However, as noted in 6.3 of this document, each of the IEC 61131-3 languages is specialized to some extent for a particular model of the problem domain. This limits the ease with which an algorithm written in one of the IEC languages can be translated into another programming language. For instance,

- the selection and iteration constructs of the ST language are difficult to translate efficiently into FBD or LD;
- textual expressions can only be used in the ST language, not in the LD or FBD language.

Due to the remaining limitations, the user should choose the language most appropriate to the type of algorithm to be developed in the body of a function or function block.

6.6.2.2 Inter-system portability

IEC 61131-3 neither defines nor requires a common exchange format for interchange of program organization unit (POU) type definitions written in graphical languages; but it does specify a textual syntax for the common elements and for the two textual languages ST and IL. Based on this textual syntax, an intersystem portability may be possible.

Consequently, compliant POU's, as defined in IEC 61131-3, can only be portable if they are written in a textual language (ST or IL). Even if written in textual language, compliant POU's will not be portable unless the set of features supported in the target system is equal to or a superset of the features supported in the source system. Compliant POU's can also not be portable if the set of implementation-dependent parameters of the two systems differ in any values. A typical example is the support of a different number of characters used in distinguishing two identifiers.

Definitions from the IEC 61131-10 can also support an intersystem portability for the other languages too.

7 Application guidelines

7.1 Use of data types

7.1.1 Type selection

IEC 61131-3 offers many elementary data types. The user can also define new data types as necessary to meet the data representation needs of the application. All data types, including user-defined types, are made available for use in a "library" of data types. The user then declares the data type to be used for each variable.

The selection of a type for a variable should be appropriate to the range of values and operations to be performed on the variable. For instance,

- if a variable can only hold the values 0 or 1, and is only to be operated on by Boolean operations, then the elementary type `BOOL` should be chosen;
- if a programmable controller program has to count something and the counts are expected to be in the range from 0 to 1000, a variable of type `SINT` or `USINT` cannot be used, since their value ranges only extend from -128 to +127 for `SINT` and from 0 to 255 for `USINT`. A reasonable data type for this purpose would be `UINT`. This has a sufficient value range and the usage of an unsigned integer type also makes it clear that negative values are not expected.

7.1.2 Type versus variable initialization

In a program that complies with IEC 61131-3, each variable has to be initialized, either explicitly by programming or implicitly by the default mechanisms defined in the standard. Uninitialized values should never occur. To ease the declaration of variables, all elementary types have default initial values specified in the document. If no initialization of a variable is specified by the user, then that variable will have the default initial value. Most default initial values are defined as the representation of the value of zero for the type.

IEC 61131-3 also allows the user to specify default initial values for user-defined types. For instance, consider a type declared by

```
TYPE TempLimit: REAL:= 250.0; END_TYPE
```

Any declared variable of this new type `TempLimit` is initialized with the default value of 250.0 instead of 0.0 as would be the normal case for all `REAL` data. Thus, in the following declaration, the variable `BoilerMaxTemperature` is initialized to 250.0, while the variable `PipeMaxTemperature` is initialized to 0.0. If the value of zero is not a reasonable maximum

temperature for the pipeline, its correct value has to be set before the first usage of the variable. Forgetting this will cause problems. In the present example, the maximum temperature for the boiler is initialized with a proper default initial value. There is no need for a set-up before the first usage, which greatly simplifies a programmable controller program and increases software reliability.

```
VAR_GLOBAL
  BoilerMaxTemperature: TempLimit;
  PipeMaxTemperature: REAL;
END_VAR
```

7.1.3 Use of enumerated and subrange types

IEC 61131-3 provides mechanisms for the definition of enumerated and subrange types as well as data types with named values. These types can make a program more readable and thus act as documentation aids by explicitly expressing the intended semantics of the values of enumerated variables. In addition, enumerated and subrange types can contribute to program reliability by avoiding the use of unintended values of variables. Whereas the usage of enumerated types can mostly be checked statically (e.g. during compile time), the checking for valid subrange values often requires dynamic checks during runtime, resulting in implementer-specific errors. Alternately, the usage of data types with named values does not result in additional checks except the usage of literals in initialization and assignments.

An enumerated data type restricts the values of variables of the type to a user-defined set of identifiers. As an example consider

```
TYPE Color: ( Red, Yellow, Green ); END_TYPE
...
VAR_GLOBAL brickColor: Color; END_VAR
```

Here a new type `Color` is defined. It may only have three values – Red, Yellow or Green. There also is no conversion function to and from enumerated types to integral types. The values only have to be distinct and reproducible. An assignment of a value to the variable `brickColor` is possible only if one of the defined `Color` values is used. All other values are flagged as errors.

A data type with named values improves the usage of literals in initializations and assignments. As an example consider

```
TYPE PaintColor: WORD ( Blue:=1, Red:=2, Yellow:=4,
                        Green:=5 );
END_TYPE
...
VAR_GLOBAL myPaintColor: PaintColor:= Blue OR Yellow;
END_VAR
```

Here a new type `PaintColor` is defined. It may have the same values as `WORD` and can be used wherever `WORD` can be used. However the literals `Blue`, `Red`, `Yellow` or `Green` can be used as initial values or in assignments to a variable of `PaintColor` Type.

Or another example

```
TYPE BrowseResultMask: WORD
  ( IsReference:= 0x0001,
    IsForword:= 0x0010,
    NodeClass:= 0x0100,
    DisplayName:= 0x1000 );
END_TYPE
...
VAR_GLOBAL Result: BrowseResultMask; END_VAR
```

```
IF (Result AND IsReference) = TRUE THEN ...
```

Can be used for symbolic masking of Statuswords of a machine.

IEC 61131-3 provides standard functions for multiplexing, selection, and comparison (equality and inequality) of enumerated data types.

IEC 61131-3 also provides for the typing of enumerated values to distinguish, for instance, between the values `Color#Red` and `PaintColor#Red`. Usage of an enumerated value or a named value also is allowed as a selector in a `CASE` statement.

EXAMPLE

Given the type and variable declarations below, the variable `Y` is of the same enumerated type as the `CURRENT_COLOR` component of the variable `X` and the assignment statement shown is valid. However, an assignment of the value of the variable `brickColor` shown above to either `Y` or `X.CURRENT_COLOR` is not allowed because the type `Color` is a different type although the (sub-) set of identifiers are the same as in `TRAFFIC_COLOR`.

```
TYPE
  TRAFFIC_COLOR: (Red, Yellow, Green);
  TRAFFIC_LIGHT:
    STRUCT
      POWER_STATE: BOOL;
      CURRENT_COLOR: TRAFFIC_COLOR;
    END_STRUCT
END_TYPE
VAR X: TRAFFIC_LIGHT;
    Y: TRAFFIC_COLOR;
END_VAR
...
Y:= X.CURRENT_COLOR;
```

7.1.4 Use of BCD data

Users should be aware of the fact that “BCD” is not a data type in IEC 61131-3. Rather, it represents an encoding option for the bit-string types `BYTE`, `WORD`, `DWORD`, and `LWORD`, where data of these types might be encoded as 2, 4, 8, or 16 BCD digits, respectively. This is in recognition of the fact that BCD is rarely used in modern systems except for transfer of data in bit-string form to and from external devices such as multi-segment displays and thumbwheel switches.

Since compliant systems are not required to support BCD arithmetic, BCD encoded data shall be converted to one of the integer types (`SINT`, `INT`, `DINT`, `LINT`, `USINT`, `UINT`, `UDINT`, or `ULINT`) using one of the `BCD_TO_*` conversion functions defined in IEC 61131-3, in order to be manipulated arithmetically. Similarly, the `**_TO_BCD` functions are provided to convert integer data to BCD encoded form for transfer to external devices.

The function “`IS_VALID_BCD`” checks whether a variable represents a valid BCD coding.

7.1.5 Use of REAL data types

The `REAL` data type described in IEC 61131-3 can be used for holding the majority of decimal values such as control loop set-points, and process values. The data format of `REAL` data is defined according to ISO/IEC/IEEE 60559 i.e. IEEE 754.

NOTE 1 ISO/IEC/IEEE 60559 and IEEE 754 have the same content.

The 32-bit `REAL` data type consists of

- 1 sign bit
- 8 exponent bits which corresponds to a range of $\pm 10^{\pm 38}$.
- 24 significant precision bits (23 bits are explicitly stored) which corresponds to a precision of approximately 7 decimal digits.

The 64-bit `LREAL` data type consists of

- 1 sign bit
- 11 exponent bits which corresponds to a range of $\pm 10^{\pm 308}$.
- 53 significant precision bits (52 bits are explicitly stored) which corresponds to a precision of approximately 15 decimal digits.

In some algorithms, rounding errors can be magnified by the calculations performed. Data types with higher precision than initially apparent can be required in order to avoid such errors.

Internal coding of a real number is based on binary fractions. Not every decimal fraction, even with fewer digits than stated above results in an exact binary fraction. e.g. 18,4 does not have an exact representation in the real format.

Addition/subtraction of 2 real values of different magnitude can lead to unexpected results, e.g. `REAL#7654321.0 + REAL#0.1` can result in `REAL#7654321.0`.

Because some numbers cannot be expressed in the floating point notation, comparison operation equal and not equal should not be used, except compared to 0.0.

NOTE 2 Additional considerations in the implementation of REAL types are provided in 8.3.1 of this document.

7.1.6 Use of character string data types

7.1.6.1 General

The `STRING` data type provides storage for variable length textual data consisting of 8-bit characters, which is required in the majority of application programs, for example, for holding process batch identifiers, recipe names, operator security codes.

IEC 61131-3 also provides the `WSTRING` data type for strings of 16-bit characters of the ISO/IEC 10646 character set (UCS-2 or the corresponding Unicode format UTF-16). Implementation dependent is the endianness (little endian or big endian) or whether or whether not a Byte Order Mark (BOM) is inserted implicitly.

IEC 61131-3 provides also explicit conversion functions between `STRING` and `WSTRING`. For `STRING` and `WSTRING`, IEC 61131-3 provides means for defining non-printable characters within a character string. This is often required when constructing messages for external devices. For example, in order to format a report it may be necessary to embed “form feed” and similar control characters in messages sent to a printer.

7.1.6.2 Length of character strings

According to IEC 61131-3, a character string is characterized by its maximum length and its current length. The maximum length is determined by the declaration of a string variable or the usage of a string constant.

- Implementation-dependent maximum length, L_{mi}
- Implementation-dependent default length, $L_{di} \leq L_{mi}$
- Declared maximum length, $L_{md} \leq L_{mi}$
 - $L_{md} = L_{di}$ when length is not explicitly declared for string variables

- L_{md} = Actual string length for string constants

The current length of a string constant does not change, but the current length of a string variable may be changed by an assignment of a new value to the string.

- Current length, L_c
 - $L_c \leq L_{md}$ for string variables
 - $L_c = L_{md}$ for string constants

IEC 61131-3 does not specify what can happen when an assignment operation tries to assign a new value of current length $L_{c, new}$ to a string variable with declared maximum length $L_{md} < L_{c, new}$. Users should be aware that implementation dependencies can cause an error or can truncate the assigned value to the length L_{md} in this case.

Truncation rather than error is the recommended option for implementers.

7.1.7 Use of character data types

IEC 61131-3 provides the character data types CHAR or WCHAR which allow access to single characters of a STRING or WSTRING

- using square brackets and the position of the character in the string, starting with position 1
- using conversion functions between STRING and CHAR and between WSTRING and WCHAR.

IEC 61131-3 provides also conversion functions between CHAR and WCHAR.

Like for WSTRING the endianness of WCHAR is also implementation dependent.

The IEC 61131-3 defines functions for endianness conversion which can be applied also to WCHAR.

7.1.8 Use of time data types

IEC 61131-3 provides a number of data types for holding time of day, date and duration. Recording the times activities occur, measuring process durations, and triggering actions at prescribed times of day or on particular dates are typical and, in some cases, essential features of most process, production and manufacturing application programs.

Typical usage of time includes

- a) accurate definition of the duration of a process phase, for example, in heat treatment where the annealing time of some materials is critical;
- b) recording the date and time of alarm conditions for process audit and maintenance purposes;
- c) controlled switch-on of a process according to the time of day, for example, to initiate pre-heating of a reactor vessel before the first shift of the week;
- d) recording the calibration date of critical analog inputs so that the system can warn when re-calibration is required;
- e) recording the times of power failure and power resumption, to calculate the down-time duration. This can be used with an application to define a power-fail strategy. For example, if power fails for a few minutes, the application may be able to continue because the process vessels are still warm; however, after a long power failure, the application should abort the process and take whatever action is necessary to put the plant into a safe state;

NOTE This example assumes that the programmable controller is able to retain the date and time of power failure in non-volatile memory.

- f) defining time-outs for certain operations to complete. For example, if a communications transaction with a serial device is not complete by a certain time, the operation is assumed to have failed.

For TIME, TIME_OF_DAY, DATE and DATE_AND_TIME the data format, the resolution, the span, the starting date are implementation dependent.

For LTIME, LTIME_OF_DAY, LDATE and LDATE_AND_TIME, the following features are specified in IEC 61131-3:2013:

- The data type LTIME is a signed 64-bit integer with unit of nanoseconds.
- The data type LDATE is a signed 64-bit integer with unit of nanoseconds with starting date 1970-01-01.
- The data type LDT is a signed 64-bit integer with unit of nanoseconds with starting date 1970-01-01-00:00:00.
- The data type LTOD is a signed 64-bit integer with unit of nanoseconds with starting time midnight with TOD#00:00:00.
- 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.

For LDATE the developer has to pay attention to a unit value that doesn't match a correct beginning of the day.

For LDT it is not specified which kind of time it represents. In most cases it will be based on the Coordinated Universal Time (UTC) but can also be based e.g. on atomic time (TAI: Temps Atomique International).

LDT does not count the leap seconds i.e. a day always consists of 86 400 s, a non-leap year consists of 365 days, a leap year consists of 366 days. This behaviour is exactly like defined in the UNIX time format.

Since LDT counts the nanoseconds from 1970-01-01-00:00:00 in a 64-bit signed integer this means a valid time span from 1677-09-21-00:12:44 to 2262-04-11-23:47:16. The handling of the over- or underflow when these dates are exceeded is not specified.

Also not specified is the handling in the case of inserting leap seconds, in a discontinuous way by decreasing the value by 1 s which can lead to ambiguous time values or negative time differences or in a continuous way by smoothly inserting small portions of seconds.

The handling of time zones or of daylight saving or summer times to get the correct local times is implementer specific. This information can be provided implicitly in the LDT value or handled in additional data fields.

The standard specifies useful operations between the time and date data types as well as ANY_NUM.

An example of such usage is given in Figure 6.

```
processDuration:= phaseDuration * phaseCount;
endTime       := startTime + processDuration;
endDateAndTime:= CONCAT_DATE_TOD(todayDate, endTime);
```

IEC

Figure 6 – ST example of time data type usage

There are also type conversion functions to support all the required manipulation between dates, time of day and duration. For example, it is possible to extract the `TIME_OF_DAY` from a `DATE_AND_TIME` variable.

7.1.9 Declaration and use of multi-element variables

There is provision within IEC 61131-3 for multi-element (“aggregate”) variables, including arrays and structures. Arrays are useful in a wide range of programs. Their use can avoid repetition of code and in many cases can make the program easier to understand. An example of the usage of an array variable is given in Figure 7. In this example, `speeds` is an array of permitted line speeds, and `lineState` is an integer (`INT`) giving state of the line such as 0 for stopped, 1 to 3 for graded increases in speed.

```
TYPE
  SPEED_ARRAY: ARRAY[0..3] OF REAL;
  POSITION_ARRAY: ARRAY[1..4] OF REAL;
END_TYPE

VAR_INPUT lineState: INT; END_VAR;
VAR_OUTPUT lineSpeed: REAL; END_VAR;

VAR
  speeds: SPEED_ARRAY:= (0.0, 1.0, 3.0, 9.0);
  accelerations: SPEED_ARRAY;
  positions: POSITION_ARRAY;
  I: INT;
END_VAR;

lineSpeed:= speeds[ lineState ];
accelerations:= speeds; (* accelerations[0]:= speeds[0], etc. *)
FOR I:= 0 TO 3
  positions[I+1]:= speeds[I]; (* element-wise assignment *)
END_FOR
```

IEC

Figure 7 – Example of declaration and use of array types

7.1.10 Use of bit-string variables

IEC 61131-3 provides as elementary data types the bit strings `BYTE`, `WORD`, `DWORD` and `LWORD`. These are composed of 8, 16, 32 and 64 Boolean values. Bitwise Boolean operations `AND`, `OR`, `XOR` and `NOT` are provided. In addition, shifting and rotating the bits is possible, where the direction to the right (left) means towards the most (least) significant bit. Arithmetic operations are not intended for these data types. Comparison functions are provided, which compare at first the least significant bit and continue the comparison toward the most significant bit. This continuation stops when the comparison result becomes settled.

7.1.11 Use of partial accessing of bitstring variables

Often there is more than one information coded in one variable of a bitstring datatype. To get access to this information either it can be masked out using the logical bitstring functions AND, OR,... or the partial access function, introduced with IEC 61131-3:2013, can be used.

The partial access feature offers the possibility to get access to a specific bit, byte, word or dword of a variable of a larger size.

The counting of the desired part starts by 0 at the least significant end of the variable.

EXAMPLE Partial access to bitstring variable

```
VAR
  Bo: BOOL;
  By: BYTE;
  Wo: WORD;
  Do: DWORD;
  Lo: LWORD;
END_VAR;

Bo:= By.%X0;           // access to bit 0 of variable By
Bo:= By.7;             // access to bit 7 of By; %X is the default and may be omitted.
By:= Lo.%B3           // access to byte 3 of Lo;
```

7.1.12 Type assignment

Although not explicitly stated in IEC 61131-3, it was the intention of the document to specify that assignment of the result of evaluating an expression to a variable be “strongly typed”; that is, that assignment should only take place when the result is of the same type as the variable.

The document extends this rule by implicit type conversion. This means that assignments are also legal if no information gets lost, i.e. the precision and range of the source data type can be expressed by the destination data type.

7.2 Data passing over POU

7.2.1 General

There are several means for passing data into and out of POUs including functions, function blocks, class and programs (see Table 1 below).

Table 1 – Available data passing of variables to POU

	Function	Method of Class / Function block	Class	Function block	Program
Immediate assignment to input, output, in-out variables as arguments of invocation	Yes	Yes	N/A	Yes	N/A
Separate assignment to input variables as static variable	No	No	N/A	Yes	Yes
Separate reading of output variables as static variable	No	No	N/A	Yes	Yes
Separate reading of or assignment to in-out variables as static variable	No	No	N/A	No	No
Internal variable with PUBLIC, INTERNAL specifier	N/A	N/A	Yes	Yes	No
Internal variables assigned to I/O address with "AT" keyword	No	No	Yes	Yes	Yes
External variables	Yes	Yes	Yes	Yes	Yes

Input, output, and in-out variables can be passed as arguments of invocation of function, method of class or function block, and function block body. The code of the invoked function, method or body is permitted to read input variables and to read or write output variables. Similarly in-out variables can be read or written.

In addition, separate assignment to input variables without invocation are permitted for function block and program since they are kind of static variables. As for programs, assignment is allowed only in the declaration of program instance in the RESOURCE construct.

As a new feature of IEC 61131-3:2013, class and function block can have static internal variables, which are defined by VAR construct, with access specifier "PUBLIC" or "INTERNAL". These can be accessed outside of the class or function block. There are no restrictions on reading or writing direction for these variables unlike static input or output variables of function block.

Also, static internal variables defined by VAR construct within a class, function block, or program can be assigned to not yet fully the I/O addresses specified by "AT %I*" or "AT %Q*" construct. These can be considered as a kind of data passing to or from a POU instance, too.

All .2 variables that were defined within a configuration, resource, or program by use of the VAR_GLOBAL keyword can be accessed from the inside of the function block, if these global variables are redeclared as external variables in the function block definition by use of the VAR_EXTERNAL keyword.

Using access paths a variable can be disclosed to a different resource and read or written by a communication function block of that resource.

7.2.2 External variables

Global variables can be defined and initialized within a configuration, resource, or program by use of the `VAR_GLOBAL` keyword. Any function, class, function block, or program can access global variables by declaring external variables with the same name as global variables by using the `VAR_EXTERNAL` keyword.

The following example shows how to define a variable `MY_TEMPERATURE` outside a function block and access it within a function block.

```
(* Within a CONFIGURATION, RESOURCE or PROGRAM *)
  VAR_GLOBAL MY_TEMPERATURE: INT; END_VAR
(* Access from inside a FUNCTION_BLOCK *)
  VAR_EXTERNAL MY_TEMPERATURE: INT; END_VAR
  ...
  MY_TEMPERATURE:= ...;
```

As global variable declarations include directly represented variables, aliases may be declared. These aliases could be referenced by external variables in functions, classes, function blocks and programs. At the first glance this construct may be only recognized as “alias” but in addition it also enables the use of the same POU on a different resource, where this variable is located on a different address. So this greatly eases porting POU to a different resource/hardware.

In the following example, a global variable `MY_TEMPERATURE` is declared to be of type `INT` located at input word `%IW22`. Within a function block, the variable name `MY_TEMPERATURE` is used as an alias for this input word.

A function block may not directly read the input word `%IW22` but may indirectly use the alias instead.

```
(* Within a CONFIGURATION, RESOURCE or PROGRAM *)
  VAR_GLOBAL MY_TEMPERATURE AT %IW22: INT; END_VAR
(* Access from inside a FUNCTION_BLOCK *)
  VAR_EXTERNAL MY_TEMPERATURE: INT; END_VAR
  ...
  ...:= MY_TEMPERATURE;
```

7.2.3 In-out (VAR_IN_OUT) variables

In-out variables are a special kind of variables used with functions, methods, function blocks and programs. They do not represent any data directly but reference other data of the appropriate type. They are declared by use of the `VAR_IN_OUT` keyword. In-out variables may be read or written to.

Inside a POU, in-out variables allow access to the original instance of a variable instead of a local copy of the value contained in the variable. Consider, for instance, the function block `ACCUM` illustrated in Figure 8a). Upon each invocation of an instance of the function block, the current value of the input `x` is added to the in-out variable `A`. If an instance of this function block is declared and invoked as shown in Figure 8b), the variable `ACC` itself will be augmented by the product `x1*x2` at each invocation of an instance of the function block `SUM_PROD`. There is no need to copy the output value from `ACC1.A` back into the variable `ACC`. Similarly, in Figure 8c), the variable `ACC` will be augmented by the sum of products `x1*x2 + x3*x4` at each invocation of an instance of the function block `SUM_2_PROD`.

Figure 8d) illustrates the use of a `VAR_IN_OUT` variable in a function which sums the elements of an array and then resets the array values to zero.

Since the output of a function does not have permanent storage associated with it, the output of a function cannot be used as a `VAR_IN_OUT` variable, as illustrated in Figure 8e). Since an in-out variable may be written to, it follows that literals, for example, 2.0, or constants that have been allocated in `VAR CONSTANT` declarations, cannot be assigned as in-out variables, as illustrated in Figure 8f).

<pre>FUNCTION_BLOCK ACCUM VAR_IN_OUT A: INT; END_VAR VAR_INPUT X: INT; END_VAR</pre>	
<pre> +----+ A--- + ---A X--- +----+</pre>	<pre>A:= A + X;</pre>
<pre>END_FUNCTION_BLOCK</pre>	

IEC

a) Declaration of a FB type using `VAR_IN_OUT`

<pre> +-----+ SUM_PROD INT--- ACC---ACC ---INT INT--- X1 INT--- X2 +-----+ ACC1 +-----+ ACCUM ACC----- A-----A ---ACC +----+ X1--- * --- X X2--- +-----+ +----+</pre>	
<pre>FUNCTION_BLOCK SUM_PROD VAR_INPUT X1, X2: INT; END_VAR VAR_IN_OUT ACC: INT; END_VAR VAR ACC1: ACCUM; END_VAR ACC1(A:= ACC, X:= X1 * X2); END_FUNCTION_BLOCK</pre>	

IEC

b) Usage of an instance of a FB type using `VAR_IN_OUT`