IEC TR 61131-8

Edition 3.0 2017-11

TECHNICAL REPORT

colour inside

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

IEC TR 61131-8:2017-11(en)

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland

Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00
info@iec.ch
www.iec.ch

About the IEC
The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications
The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

IEC Catalogue - webstore.iec.ch/catalogue
The stand-alone application for consulting the entire bibliographical information on IEC International Standards, Technical Specifications, Technical Reports and other documents. Available for PC, Mac OS, Android Tablets and iPad.

IEC publications search - www.iec.ch/searchpub
The advanced search enables to find IEC publications by a variety of criteria (reference number, text, technical committee,…). It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished
Stay up to date on all new IEC publications. Just Published details all new publications released. Available online and also once a month by email.

Electropedia - www.electropedia.org
The world's leading online dictionary of electronic and electrical terms containing 20 000 terms and definitions in English and French, with equivalent terms in 16 additional languages. Also known as the International Electrotechnical Vocabulary (IEV) online.

IEC Glossary - std.iec.ch/glossary
65 000 electrotechnical terminology entries in English and French extracted from the Terms and Definitions clause of IEC publications issued since 2002. Some entries have been collected from earlier publications of IEC TC 37, 77, 86 and CISPR.

IEC Customer Service Centre - webstore.iec.ch/csc
If you wish to give us your feedback on this publication or need further assistance, please contact the Customer Service Centre: csc@iec.ch.

IEC TR 61131-8

Edition 3.0 2017-11

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

CONTENTS

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.

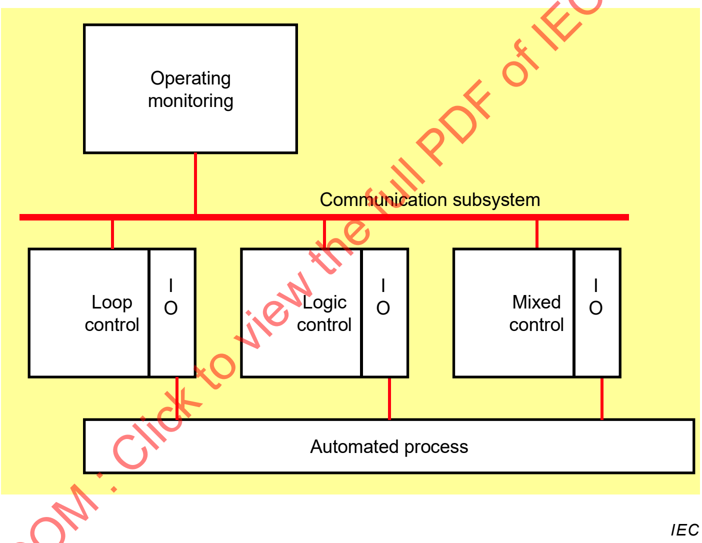


Figure 1 – A distributed application

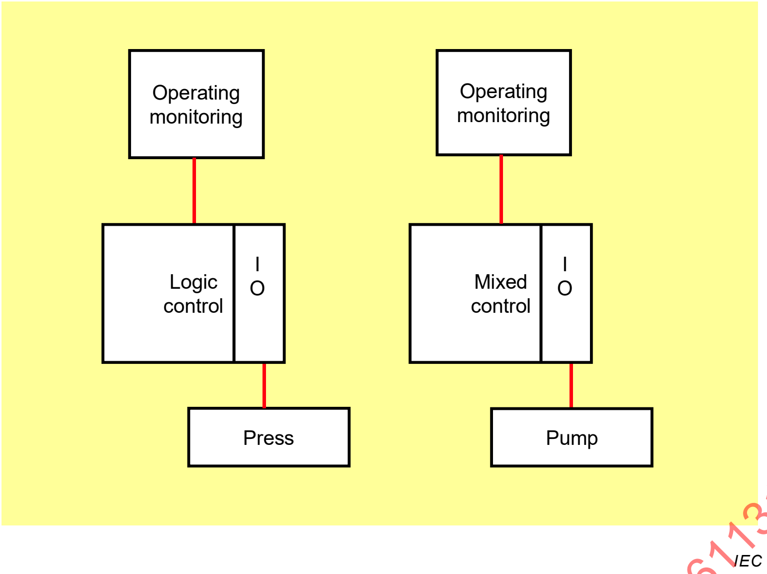


Figure 2 – Stand-alone applications

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 inter-faces, 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.

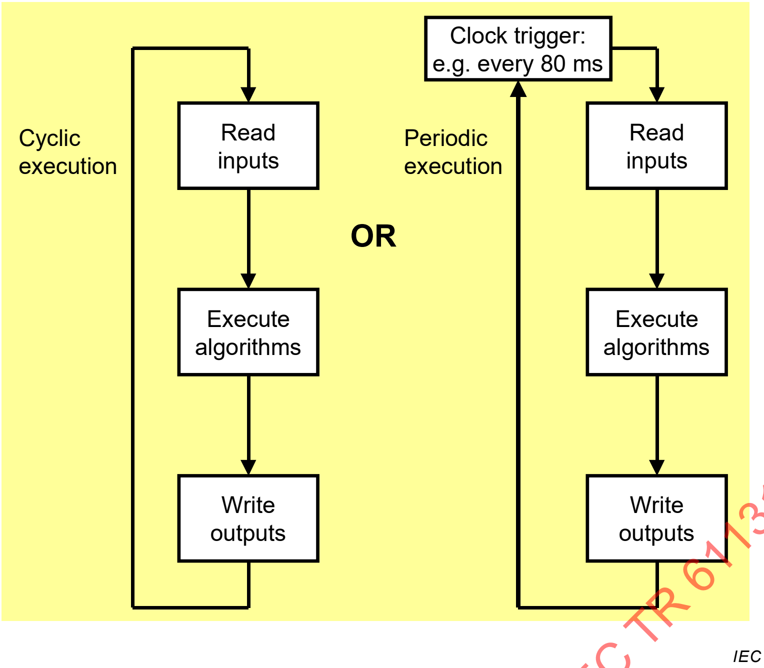


Figure 3 – Cyclic or periodic scanning of a program

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.

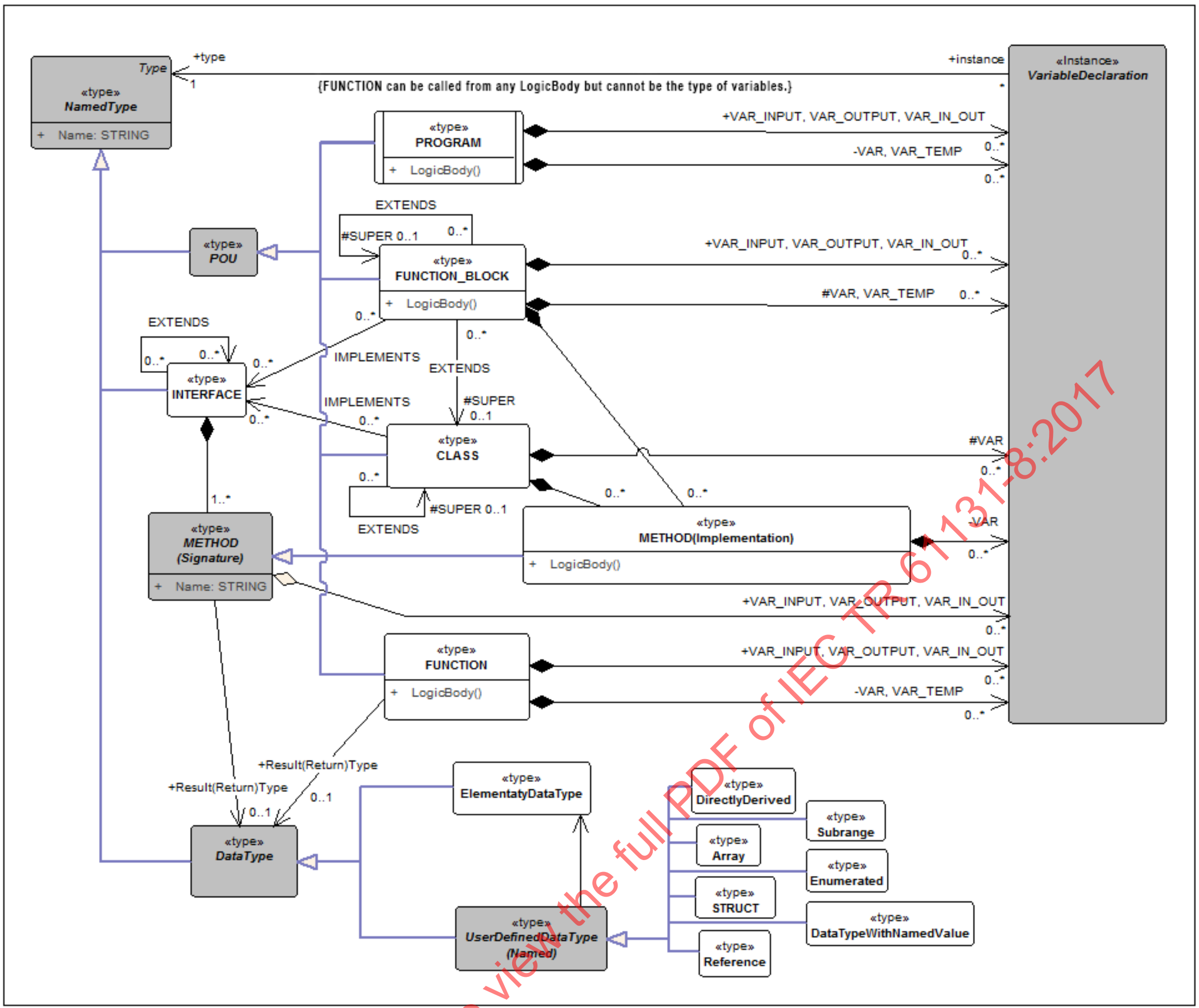
f) Flexibility in the selection of languages suited for programming application-specific functionalities will increase the reuse of software solutions to process control problems.

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.

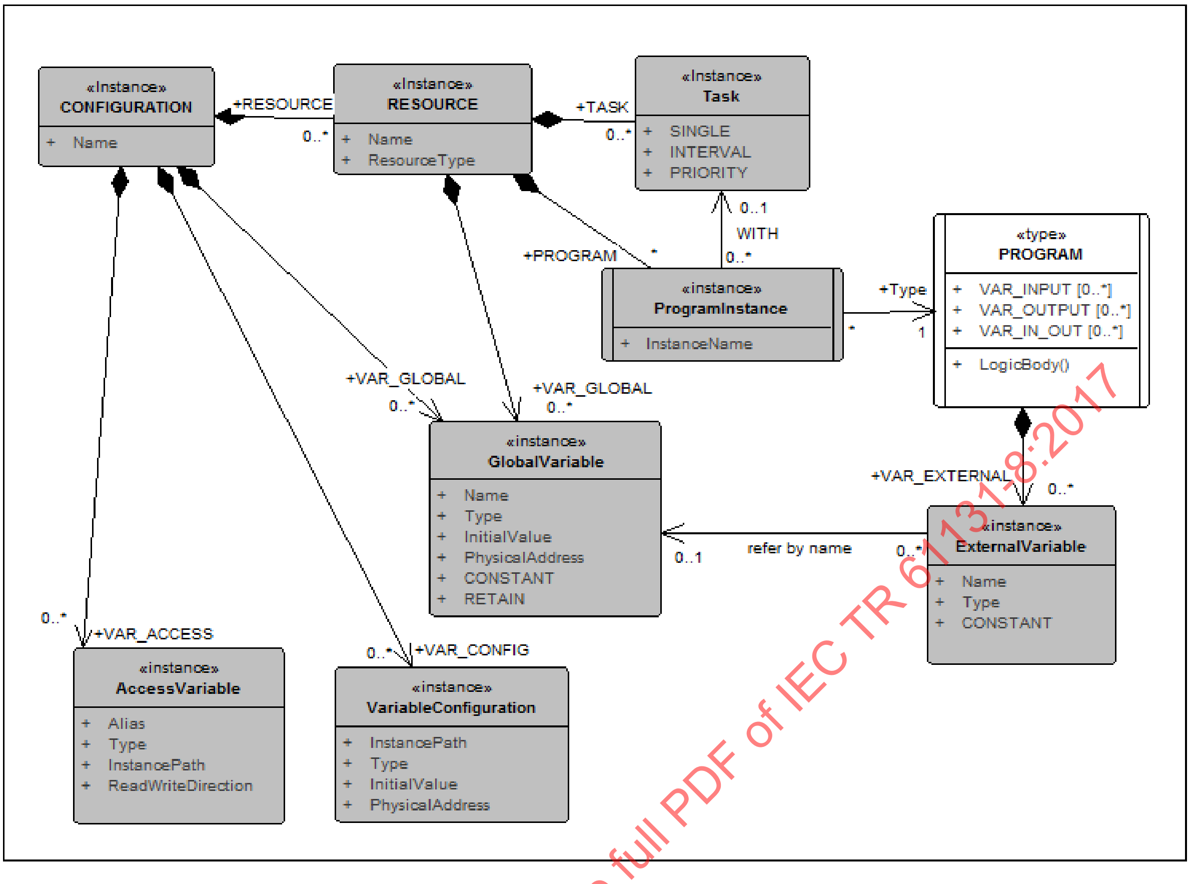
6.4 Language items overview

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 POUs, 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 POUs 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 POUs 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, $L_{\text{di}} \leq L_{\text{mi}}$
- Declared maximum length, $L_{\text{md}} \leq L_{\text{mi}}$
  - $L_{\text{md}} = L_{\text{di}}$ 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).

```
FUNCTION_BLOCK ACCUM

VAR_IN_OUT A: INT; END_VAR
VAR_INPUT X: INT; END_VAR
```

```
        +---+
    A---| + |---A
    X---|   |
        +---+
```

```
A:= A + X;
```

```
END_FUNCTION_BLOCK
```

IEC

a) Declaration of a FB type using VAR_IN_OUT

```
          +----------+
          | SUM_PROD |
    INT---|ACC----ACC|---INT
    INT---|X1        |
    INT---|X2        |
          +----------+

                 ACC1
                +-------+
                | ACCUM |
  ACC-----------|A-----A|---ACC
        +---+   |       |
  X1---| * |---|X       |
  X2---|   |   +-------+
        +---+
```

```
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
```

IEC

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

```
      +------------+
      | SUM_2_PROD |
INT---|ACC------ACC|---INT
INT---|X1          |
INT---|X2          |
INT---|X3          |
INT---|X4          |
      +------------+

               ACC1
             +-------+
             | ACCUM |
ACC----------|A-----A|--+
     +---+   |       |  |
X1---| * |---|X      |  |
X2---|   |   +-------+  |
     +---+              |
+-----------------------+
|              ACC2
|            +-------+
|            | ACCUM |
+------------|A-----A|---ACC
     +---+   |       |
X3---| * |---|X      |
X4---|   |   +-------+
     +---+
```

```
FUNCTION_BLOCK SUM_2_PROD

VAR_INPUT X1,X2,X3,X4: INT;
END_VAR

VAR_IN_OUT ACC: INT; END_VAR
VAR ACC1, ACC2: ACCUM; END_VAR

ACC1(A:= ACC, X:= X1 * X2);
ACC2(A:= ACC, X:= X3 * X4);

END_FUNCTION_BLOCK
```

IEC

c) Usage of two instances of a FB type using VAR_IN_OUT

```
            +-------+
            | ASUM  |
            |       |--INT
INT[1..10]--|A-----A|--INT[1..10]
            +-------+
```

```
FUNCTION ASUM: INT
 VAR_IN_OUT
  A: INT[1..10];
 END_VAR
 VAR
  I1: INT;
  Sum: INT:= 0;
 END_VAR
 FOR I1:= 1 TO 10 DO
  Sum:= A[I1] + Sum;
  A[I1]:= 0;
 END_FOR;
 ASUM:= Sum;
END_FUNCTION
```

IEC

d) Declaration of another FB type using VAR_IN_OUT

```
              ACC1
     +---+   +-------+
X1---| * |   | ACCUM |
X2---|   |---|A-----A|---ACC
     +---+   |       |
X3-----------|X      |
             +-------+
```

ILLEGAL USAGE:
In-out A is not a variable
or function block name

IEC

e) Example of illegal usage

<table>
<tr>
<td><pre>
       ACC1
    +--------+
    | ACCUM |
2.0---|A-----A|---2.0
    | |
 ---|X |
    +--------+
</pre></td>
<td>ILLEGAL USAGE:
In-out A is not a variable
or function block name</td>
</tr>
</table>

IEC

f) Example of illegal usage

Figure 8 – Examples of VAR_IN_OUT usage

7.2.4 Formal and non-formal invocations and argument lists

The content of this 7.2.4 refers to invocations in textual languages only.

a) According to IEC 61131-3, functions are able to yield not only a single result – passed to the invoking expression by the function name – but they can – like function blocks – pass values through output variables and/or in-out variables. Functions still contain no internal state information. The introduction of output variables now allows for example to pass `ENO` additionally to the function result in textual invocations of standard functions. `ENO` is the only additional output variable provided by standard functions; user-defined functions may declare an arbitrary number of additional output or in-out variables.

b) According to IEC 61131-3, function block invocations use the same rules and features as function calls. This means especially that the argument list of a function block invocation may also have the form of a parenthesized list of the complete set of actual input arguments.

c) IEC 61131-3 introduces the terms "formal call" and "non-formal call" for the two invocation variants with formal argument assignment and without formal argument assignment. These two variants are introduced as features for the invocation of functions and methods, and through the reference for the invocation of function blocks, too. If the programming tool offers both then the usage of formal invocation is recommended.

d) The usage of the operator "=>" for program configurations is extended to formal argument lists. This allows the assignment of output variables inside the formal argument list of a function or method call or a function block invocation. For function and method calls, this mechanism of output variable assignment is the only possible one, since there is no construct to access output variables of functions or methods from the calling program organization unit, as it exists for output variables of function blocks. The typical usage of this operator is the assignment of `ENO` in a function call to a variable in the calling POU. An output variable may be prefixed by a `NOT` operator, to allow the representation of a negated output.

e) A formal argument list has the form of a set of assignments of actual argument values to the input and in-out variables using the ":=" operator, and of the values of output variables to variables located in the calling POU using the "=>" operator. It is not necessary to provide an assignment for every declared variable of the function, method or function block: any variable not assigned a value in the list shall have the default value, if any, assigned in the function, method or function block specification, or the default value for the associated data type. The ordering of argument assignments in the list is not significant.

f) In contrast to the rules for formal argument lists, for non-formal argument lists of function and method calls and function block invocations, the actual arguments inside the list have to be ordered as prescribed by the order of variables defined in the function, method or function block declaration. Actual arguments have to be provided for the complete set of declared variables, not including the execution control variables `EN` and `ENO` – these variables cannot be handled by non-formal argument lists. Concerning the completeness of the set, there is one exception for textual invocations of functions in IL: in this textual language the first argument of the function call is expected as the current result, and the non-formal argument list starts with the second argument.

Table 2 illustrates function calls and function block invocations with non-formal and formal argument lists in the two textual languages ST and IL.

Table 2 – Examples of textual invocations of functions and function blocks

DESCRIPTION	**EXAMPLE** (Note 8)
a) Formal function call in ST (Note 1)	A:= LIMIT(EN:= COND, IN:= B, MX:= 5, ENO => TEMP);
b) Non-formal function call in ST	A:= LIMIT(1, B, 5);
c) Formal function call in ST (Note 2)	A:= LIMIT(EN:= TRUE, MN:= 1, IN:= B, MX:= 5);
d) Non-formal function block invocation in ST	CMD_TMR(START, T#300ms, OUT, ELAPSED);
e) Formal function block invocation in ST (Note 3)	CMD_TMR(IN:= START, PT:= T#300ms, Q => OUT, ET => ELAPSED, ENO => ERR);
f) Formal function block invocation in ST (Note 4)	MYTON(EN:= NOT(X <> Y), IN:= START, NOT Q => OUT);
g) Non-formal function call in IL (Note 5)	LD 1 LIMIT B,5 ST A
h) Formal function call in IL (Note 6)	LIMIT(EN:= COND, IN:= B, MN:= 0, MX:= 5, ENO => TEMP) ST A
i) Non-formal function block invocation in IL	CAL C10(START, FALSE, A, OUT, B)
j) Formal function block invocation in IL (Note 7)	CAL C10(CU:= START, Q => OUT)

NOTE 1 Illustrates assignments of EN and ENO; uses default value MN:= 0.

NOTE 2 Yields the same result as example b).

NOTE 3 Exhibits the same behaviour as example d); additionally assigns ENO to ERR.

NOTE 4 Illustrates negated input EN and negated output Q.

NOTE 5 Yields the same result as example b); loads value of MN into current result.

NOTE 6 Exhibits the same behaviour as example a).

NOTE 7 Counts like example i); uses PV:= A from a previous call, but does not assign to B.

NOTE 8 A declaration such as

```
VAR
 CMD_TMR, MYTON: TON;
 C10: CTU;
 A, B: INT;
 X, Y: REAL;
 ELAPSED: TIME;
 START, COND, TEMP, OUT, ERR, START: BOOL;
END_VAR
```

is assumed in the above examples.

7.2.5 Assignment of input, output, and in-out variables of programs

Input, output, and in-out variables can be declared in PROGRAMs and they can be assigned only in the instance declaration of the PROGRAM within resource declaration. Global variables and directly represented variables can be passed into or out of the programs.

```
CONFIGURATION CELL_1

    VAR_GLOBAL  w: UINT;  END_VAR

    RESOURCE STATION_1 ON PROCESSOR_TYPE_1

      VAR_GLOBAL  z1: BYTE;  END_VAR

      TASK SLOW_1(INTERVAL:= t#20ms, PRIORITY:= 2);

      TASK FAST_1(INTERVAL:= t#10ms, PRIORITY:= 1);

      PROGRAM PINST1 WITH SLOW_1: PROG1(x1:= %IX1.1);

           // Assigning directly represented variable to program input

      PROGRAM PINST2 WITH SLOW_1: PROG2(OUT1 => w);

          // Assigning program output to a global variable

    END_RESOURCE

  END_CONFIGURATION
```

This is a way for PROGRAM to read/write global variables or directly represented variables. The differences from reading/writing via external variables or reading/writing directly represented variables within the PROGRAM are listed as follows.

a) PROGRAM declaration itself does not depend on the global variable's name or the address. Different global variables or address can be assigned for each PROGRAM instance.

b) Which global variables the PROGRAM instance reads and writes are explicitly declared. Data-flow among PROGRAM instances becomes clear using a declaration. In the case of external variable or directly represented variable, read/write direction is implicit and the logic inside the PROGRAM needs to be examined to know it.

c) The values of the global variables or directly represented variables are not reflected instantly. As the natural behaviour of input / output variables, the values of global variables or directly represented variables are copied to the input variables before the execution of the PROGRAM, and the value of the output variables are copied to the global variables or directly represented variables after the execution of the PROGRAM. The values of input variables are never changed during the execution of the PROGRAM even if assigned global / directly represented variables are overwritten by something outside of the PROGRAM. Also output variables are never reflected to the assigned global / directly represented variables until the execution of the PROGRAM finished. This is an important behaviour in the case that global variables are used for data passing over the programs related with different tasks.

7.3 Use of function blocks

7.3.1 Function block types and instances

A function block type is a POU. This POU describes the input and output variables and the local data area for instances of the function block. It further contains the rules for processing this data when an instance of the function block is invoked. Variables cannot be read from, or written to, the function block type itself, nor can the function block type itself be invoked; these operations are reserved for instances of the function block.

Multiple function block instances based on this type of declaration can be used. The individual instances are independent from each other. Each function block instance has a unique identifier (the instance name) and its own data area used for input, output, and internal variables of this function block instance. A function block instance can be accessed and invoked via its instance name.

These qualities of function blocks are related to object oriented programming (OOP) and known as instantiation. IEC 61131-3:2013 extends this idea of OOP by other language constructs and is described in a further subclause.

Typical function block instances are timers or counters, which keep their values from one invocation to the next and determine whether a final value has been reached or not. Another field of interest for using function blocks is the access to a shared device in a controlled manner. Here, a single function block instance gets the exclusive control of that device and acts like a semaphore, where the device can be accessed only if the corresponding function block instance is invoked.

The advantage of using function block instances is that the functionality associated with a defined data structure only has to be declared once, and can then be used independently in multiple instances within a programmable controller program. This "prototype" is kept in the function block type and it can be reused as many times as necessary by declaring instances of this type. Thus, the user is assured that there are no errors in any function block instance as long as there are no errors in the associated function block type.

Function block instances can also be helpful for testing and debugging, since the entire set of current state data for the instance is easily accessible for monitoring and on-line modification.

Function block instances are a special feature of function blocks as defined in IEC 61131-3. There is no equivalent in procedural programming languages such as Pascal or C.

7.3.2 Scope of data within function blocks

The principle of data encapsulation and hiding of variable names discussed in 6.6.1.1 can be unfamiliar to users of traditional programmable controller systems. This principle also applies to instances of function blocks, which are declared in the same manner as variables in a `VAR..END_VAR` construct. Hence, function block instances that appear inside another function block are not visible outside the containing function block, contrary to the practice in some traditional programmable controller systems. The scope of an instance of a function block shall be local to the program organization unit in which it is instantiated, unless it is declared to be global in a `VAR_GLOBAL` block.

This requirement might be considered to contradict the assertion that any function block which has already been declared can be used in the declaration of another function block or program. However, it is clear that this latter assertion refers to function block types, not function block instances; hence, the two requirements are not contradictory.

Figure 9 illustrates the application of this principle. Here, an instance named FB1 of function block type FBy will occur in each instance of function block type FBx. When function block type FBx is instantiated twice in an instance of program type A, two separate and distinct instances of function block type FBy are created. As illustrated in Figure 9c), each such instance forms part of the private data area of an associated instance of FBx (FB1 and FB2, respectively), and is hence invisible outside of this instance.

```
FUNCTION_BLOCK FBx
 VAR FB1: FBy; END_VAR; (* FBy is a function block type *)
...
 FB1(...); (* Invoke instance FB1 *)
...
END_FUNCTION_BLOCK
```

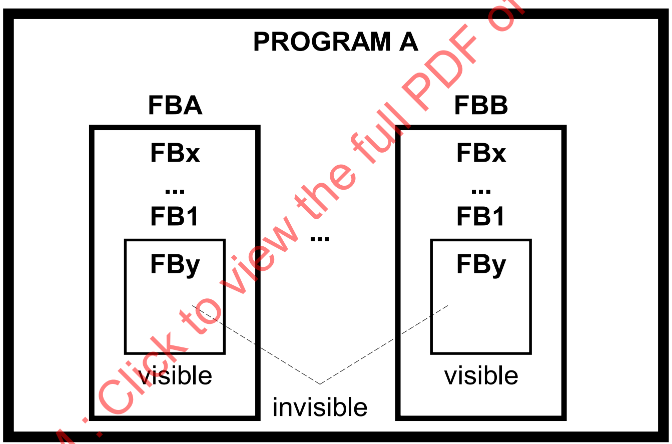
IEC

a) Declaration of contained function block type

```
PROGRAM A
 VAR FBA: FBx; (* Two instances of type FBx *)
   FBB: FBx; (* Each contains an instance FB1 of type FBy *)
 END_VAR;
...
 FBA(...); (* Invoke instance FBA *)
 FBB(...); (* Invoke instance FBB *)
...
END_PROGRAM
```

IEC

b) Declaration of containing program type



IEC

c) Visibility of function block instances

NOTE Suppression of irrelevant detail is shown by the ellipsis (...).

Figure 9 – Hiding of function block instances

7.3.3 Function block access and invocation

There is a difference between read access to a variable of a function block instance and the invocation of the function block instance itself. Reading an output variable of a function block instance is equivalent to reading the value of an element of a structured variable. However, in contrast to structured variables, an explicit assignment to the function block output variables is not allowed. The assignment of values to output variables of a function block instance is allowed only within the body of the function block instance.

In general, it is possible to invoke a single instance of a function block several times ("multiple assignment") within a POU. However, depending on the programmable controller implementation, this possibility may be restricted to a single invocation of each function block instance within a POU. A POU that uses multiple invocations of a single function block instance may be non-portable to such implementations. See 7.9 for more details.

All the values of the private data area of a function block instance persist from one invocation to the next. Thus, the private data area of this instance can be seen to be a "memory" recording a current state. For this reason, different invocations of the same function block instance with the same input variable values may yield different values of the data area of this instance.

The assignment of values to input variables is allowed only as part of the invocation of the function block instance. However, not all the input variables of a function block instance have to be set explicitly in order to enable an invocation. This is true because no undefined values of variables are possible for the following reasons:

– standard default values are defined for initialization of variables of all possible data types;
– the initial value of any variable can be specified in its declaration;
– all variables keep their values from one invocation of a function block instance to the next.

The formal invocation of a function block has to be used, if not all input variables of a function block instance are assigned. A non-formal invocation of a function block requires the complete set of actual arguments in its argument list.

In addition to the inputs and outputs, variables can be declared using the keyword PUBLIC at the variable declarations of a POU. Other POUs using the previous declared POU can access these public variables even outside of the invocation.

7.4 Differences between function block instances and functions

Although both function blocks and functions are POUs, there are significant differences between function blocks and functions.

a) The invocation of a function may have one or no result. Additionally a function can deliver output variables like a function block.

   NOTE In IEC 61131-3 the invocation of a function can affect the values of its associated output variables or in-out variables as well as its result.

b) The result of invoking a function can be used as a value in an expression or an assignment statement but cannot be used as the target of an assignment operation.
c) A function does not possess a private memory (i.e., internal state information) which is affected by the history of previous invocations. Thus each call of a function with identical arguments yields the same result.
d) The scope of a function name, like that of a function block type, is global, as opposed to the scope of a function block instance name.

7.5 Use of indirectly referenced function block instances

7.5.1 General

A function block instance is referenced for the purpose of reading via its output variables or for invoking the referenced instance. These operations can also be performed on a function block instance whose instance name is passed as an argument to a POU. In this case, the function block instance is not referenced directly by using a fixed name for the function block instance; rather, the reference is indirect via an input or in-out variable of the POU. The instance name of the referenced function block is assigned to an appropriate variable from “outside” of the POU in which it is referenced.

This mechanism allows access to, or invocation of, different instances of a specified function block type within the body of another program or function block.

The technique of using function block instance names as arguments offers the user many new possibilities in programming. These features make it possible to access or invoke the instance of a function block type without defining the particular instance of the referenced function block need in the declaration section of the referencing POU. Furthermore, the referenced instance can change from one invocation of the referencing POU to another.

Useful applications for this mechanism can be found in connection with problems handling several machines with the same behaviour, each represented by a single function block instance.

7.5.2 Establishing an indirect function block instance reference

IEC 61131-3 defines mechanisms for establishing an indirect reference to an instance of a function block. As illustrated in Figure 10 below, the function block reference may be established as

a) a variable declared in a VAR_INPUT declaration;

b) a variable declared in a VAR_IN_OUT declaration;

c) an external variable.

In each example in Figure 10, the interface definition of a block that references a function block instance is shown, followed by an example of the passing of the referenced function block instance as part of the invocation of the referencing function block.

```
        +---------------+    (* Referencing block type *)
        |   INSIDE_A    |
 TON---|I_TMR EXPIRED |---BOOL
        +---------------+
```

```
FUNCTION_BLOCK                  (* Reference-passing block *)
         +---------------+    (* External interface *)
         | EXAMPLE_A     |
 BOOL---|GO     DONE     |---BOOL
         +---------------+

                   (* Function Block body *)
    (* Referenced Block *)
             E_TMR          (* Referencing Block *)
            +------+                I_BLK
            | TON  |              +---------------+
       GO---|IN  Q |              |   INSIDE_A    |
 t#100ms---|PT  ET|    E_TMR---|I_TMR EXPIRED |---DONE
            +------+              +---------------+
END_FUNCTION_BLOCK
```

IEC

a) Function block name used as an input variable

```
        +-----------------+    (* Referencing block type *)
        |     INSIDE_B    |
 TON---|IO_TMR----IO_TMR|---TON
 BOOL--|TMR_GO    EXPIRED|---BOOL
        +-----------------+
```

```
FUNCTION_BLOCK              (* Reference-passing block *)
         +---------------+    (* External interface *)
         | EXAMPLE_B     |
 BOOL---|GO     DONE     |---BOOL
         +---------------+

                   (* Function Block body *)
    (* Referenced Block *)
             E_TMR          (* Referencing Block *)
            +------+                I_BLK
            | TON  |              +------------------+
            |IN  Q |              |    INSIDE_B      |
 t#100ms---|PT  ET|    E_TMR---|IO_TMR-----IO_TMR|--- E_TMR
            +------+    GO------|TMR_GO    EXPIRED |---DONE
                                  +------------------+
END_FUNCTION_BLOCK
```

IEC

b) Function block name used as an in-out variable

```
FUNCTION_BLOCK            (* Referencing block type *)
       +---------------+     (* External interface *)
       |   INSIDE_C     |
 BOOL--|TMR_GO EXPIRED|---BOOL
       +---------------+
 VAR_EXTERNAL X_TMR: TON; END_VAR
 END_FUNCTION_BLOCK
```

```
 PROGRAM                (* Reference-passing program *)
         +---------------+    (* External interface *)
         | EXAMPLE_C     |
 BOOL---|GO     DONE     |---BOOL
         +---------------+
 VAR_GLOBAL X_TMR: TON; END_VAR (* Referenced block *)
                    (* Program body *)
                I_BLK          (* Referencing block *)
          +----------------+
          |   INSIDE_C      |
 GO------|TMR_GO EXPIRED |---DONE
          +----------------+
 END_PROGRAM
```

IEC

c) Function block name used as an external variable

Figure 10 – Graphical use of a function block name

7.5.3 Access to indirectly referenced function block instances

Implicit or explicit access to an indirectly referenced function block instance means reading (i.e., using without modifying) its output variables. The private data area of the function block instance remains unchanged.

Examples of this usage are given in Figures 11a) through 11c), corresponding to the interface and variable declarations given in Figures 10a) through 10c), respectively. In each case, the `Q` output of the referenced `TON` function block is passed to the `EXPIRED` output of the referencing function block.

```
EXPIRED:= I_TMR.Q;
```

IEC

a) Referenced input block (ST)

```
   IO_TMR.Q              Expired------
II-------    …    ----()
```

IEC

b) Referenced in-out block (LD)

```
              +----------+
              |   MOVE   |
              |          |
IO_TMR.Q ---|  IN   OUT |--- Expired
              +----------+
```

IEC

c) Referenced external block (FBD)

NOTE See Figure 10 for corresponding variable and interface definitions.

Figure 11 – Access to an indirectly referenced function block instance

7.5.4 Invocation of indirectly referenced function block instances

As described in IEC 61131-3, the invocation of an indirectly referenced function block instance means the invocation of the function block instance from inside another function block. Using this capability, the private data area (and therefore also the output variables) of this function block instance can be modified.

The general rules for an invocation of a function block instance given in IEC 61131-3:2013, Table 42 are valid in this case.

– The assignment of a value to an input variable of the indirectly referenced function block instance is allowed only as part of the invocation of this function block.

– The output variables of this function block instance cannot be modified from outside.

As defined in IEC 61131-3, the reference to the invoked function block instance can be established via

– a variable declared in a VAR_IN_OUT declaration, as illustrated in Figure 12;

– an external variable.

Examples of this usage are shown in Figures 12b and 12c, respectively.

An invocation of an indirectly referenced function block instance established via an input variable is not allowed, since an invocation of this function block instance may change the values of the private data area. The modified values may have effects outside the invoked function block. This behaviour is prohibited for input variables. As noted in 7.5.3, this precludes the graphical representation (which implies invocation) of indirectly referenced function block instances established via input variables.

```
FUNCTION_BLOCK B
 VAR_IN_OUT COUNTER_FB: CTU; END_VAR
 VAR REACHED: BOOL; END_VAR
 ...
 (* Invocation of the variable COUNTER_FB *)
 COUNTER_FB (...);
 (* Access to output Q of the variable COUNTER_FB *)
 REACHED:= COUNTER_FB.Q;
 ...
END_FUNCTION_BLOCK
```

IEC

a) Textual declaration and invocation

```
          COUNTER_FB
        +-----------+
        |  CTU      |
        |           |
     --->CU       Q |---REACHED
        |           |
     ---|R          |
        |           |
     ---|PV      CV |
        +-----------+
```

IEC

b) Graphical invocation

```
PROGRAM P2
...
 VAR REACHED: BOOL;
  B1: B;
  COUNTER1: CTU;
  COUNTER2: CTU;
 END_VAR
 ...
 (* Access value of COUNTER1.Q before invocation *)
 REACHED:= COUNTER1.Q;
 ...
 (* Invocation of B1 causes invocation of COUNTER1 *)
 B1 (COUNTER_FB:= COUNTER1);
 ...
 (* Access value of COUNTER1.Q after invocation *)
 REACHED:= COUNTER1.Q;
 ...
 (* Invocation of B1 causes invocation of COUNTER2 *)
 B1 (COUNTER_FB:= COUNTER2);
 ...
END_PROGRAM
```

IEC

c) Textual passing of instance name

```
COUNTER1.Q---REACHED
...
                 B1
            +-----------------------+
            |      B                |
COUNTER1---|COUNTER_FB---COUNTER_FB|--- COUNTER1
            |                       |
            |                       |
            +-----------------------+
...
COUNTER1.Q---REACHED
...
            +-----------------------+
            |      B                |
COUNTER2---|COUNTER_FB---COUNTER_FB|--- COUNTER2
            |                       |
            |                       |
            +-----------------------+
```

IEC

d) Graphical passing of instance name

Figure 12 – Invocation of an indirectly referenced function block instance

In the example shown in Figure 12, different instances of CTU function blocks can be invoked within the body of function block B. An indirect reference to a CTU function block instance is established as a variable declared in a VAR_IN_OUT declaration to function block B. In Figure 12c) and d), several invocations of instance B1 of function block B with different instance names of function block COUNTER_FB as a variable result in invocations of different CTU instances corresponding to the variable instance names.

As shown in IEC 61131-3, a function block instance can be declared as a global variable in a program, and the function block instance name can then be used as an external variable in function blocks of other types. Although the particular instances of such function blocks may vary between programs, the referenced function block instance does not vary from one invocation to the next of the referencing function block (the block containing the external reference). Such a function block instance may be used, for example, to provide means for externally setting the values of variables of all instances of a function block type that contains the function block name as an external reference.

Thus, the use of external variables for function block instance names differs from the use of VAR_IN_OUT variables for function block instance names as described above. While VAR_IN_OUT variables can be changed from one invocation of a function block instance to the next, external function block instance references will not change, although their variable values may change.

7.5.5 Recursion of indirectly referenced function block instances

According to IEC 61131-3, invocations of POUs shall not be recursive. This is also true for invocations of function blocks.

Since IEC 61131-3 defines the POU as the function block type and not the function block instance, the check for recursive invocations of function blocks can be made by checking the body of a function block type. This is true even when the mechanism of using function block instance names as arguments is supported, because the instance name but not the type of an invoked function block can vary from one invocation to the next. The type of the invoked function block is determined by the declaration in the body of the invoking POU.

7.5.6 Execution control of indirectly referenced function block instances

The following measures are recommended to avoid ambiguity in determining the execution control of indirectly referenced function block instances.

a) Direct association of tasks to such function blocks, as described in IEC 61131-3, should be avoided.

b) Use of such function blocks should be restricted if possible to algorithms written in the ST language defined in IEC 61131-3, which has explicit function block invocation statements.

c) Where an application makes graphic use of such function blocks unavoidable, the PADT should provide tools for establishing unambiguous execution control.

7.5.7 Use of indirectly referenced function block instances in functions

As mentioned in IEC 61131-3, a function block instance name can be used as an input variable of a function. Within the body of this function, the output variables of the indirectly referenced function block instance can be read.

At a first view this seems to be contrary to the definition of a function in IEC 61131-3. A call of a function with the same input variables always has to yield the same output value. This may not be true, if the same function block instance name is passed to a function and this function block instance has been invoked in the meantime. A more detailed investigation of this fact, however, shows that the read access to a function block instance is similar to a read access of a structured variable. To get the same function value, it is therefore not enough to ensure that the name of a function block instance is identical from one function call to the next but also that the values of all output variables of the referenced function block instance are identical.

7.6 Use of programs

7.6.1 Difference to function block

Basically a program is almost the same as a conventional function block of IEC 61131-3:2003, e.g. without object-oriented feature of IEC 61131-3:2013. Each program instance should always be an independently executable POU that can only be assigned to a task.

– Program instance are executed by their task and cannot be invoked by other POUs.

– Each program should be designed to be executed independently of other programs. Dependencies between programs reduce their reusability and should be minimized.

– As for input, output, and in-out variables of the program, they can be assigned from and to the global variables or directly represented variables, before and after the execution of the program instance. Such assignment can be defined in the declaration of the program instance in the RESOURCE construct.

7.6.2 Communication with other programs

As a program is an asynchronously executable POU which can be assigned to a task, communications among program instances have to be assumed to be asynchronous. The way of communication among program instances can be considered to have one or more of the following implementation options.

a) By the use of access variables and function blocks defined by IEC 61131-5

If access variables (VAR_ACCESS) and IEC 61131-5 function blocks are supported by the PADT and runtime system, this option can be selected. Refer to 7.14 in detail.

b) By global variables via input, output variables of programs

If the PADT and runtime system have some solution for concurrent access to the global variables, this option can be selected. As for example, refer to 7.2.5.

c) By global variables via external variables of programs

If the PADT and runtime system have some solution for concurrent access to the global variables, this option can be selected. However, this option is not recommended since it still may have problem with concurrency, and data exchange among programs is not represented explicitly.

7.7 Object orientation

7.7.1 General introduction

Object orientation is a well-established concept for programming languages. In the IEC 61131-3 it has been extended in the 2013 edition. New language elements CLASS, METHOD and INTERFACE have been introduced as well as the new concept of inheritance.

Every feature that is suitable for classes is also suitable for function block types. Therefore all statements for classes are valid for function blocks, too, if not explicitly stated otherwise.

A class is a collection of methods combined with a set of variables. The accessibility to the variables and to the methods is defined by using the keywords PUBLIC, PRIVATE, PROTECTED and INTERNAL.

A class has to be instantiated in order to be used, similar to function blocks. A class instance has to be declared in order to use its methods. A class instance is often called an object.

Though the data of a class instance may be accessible for read and write operations from outside the class, the preferred means to manipulate a class object is invoking the methods on the class object.

In contrast to a function block instance, there is no call of the class object. A class does not define a body.

A class may be derived from another class, and a class may implement interfaces.

In the following paragraphs the term “class” will be used as a synonym for CLASS and FUNCTION_BLOCK as well.

7.7.2 Usage of methods

In comparison to a classic function block without methods, a class offers more possibilities to structure the code. A classic function block is limited to one body, which can be seen as one method, whereas a class may contain an arbitrary number of methods.

A method can be described as a function with special access to the variables of the class. The method itself contains temporary inputs, outputs and local variables.

The most important reason to use methods is to reduce the complexity of the code. There are different approaches to measure the complexity of code (e.g.: the number of lines or networks). These different approaches will not be discussed here. But it is good practice to measure the complexity of code and to restrict the code to some upper level of complexity.

NOTE Important software metrics have been developed for instance by McCabe (cyclomatic complexity), Halstead, Prather, and others. See Bibliography (see the last three references of the bibliography).

If the body of a function block exceeds a certain complexity, it is a sign to divide the code into methods and use object oriented function blocks or classes instead of a classic function block. However, finding a convincing choice of methods is nothing but a simple task.

Typically not all code of a function block is executed at each cycle. There are situations were only a fracture of the code is executed. There might be for example an initialization phase and a cyclic phase, and distinct code for the two phases. In this case, we could easily introduce two different methods for these two phases.

The CTUD-Function block for example as defined in IEC 61131-3 has code for counting up and counting down in the same body. Both are mutual exclusive (Figure 13). An equivalent object oriented implementation could introduce methods for Reset, Load, Count_Down, Count_Up (Figure 14).

```
FUNCTION_BLOCK CTUD
VAR_INPUT
    CU, CD: BOOL R_EDGE;
    Reset, LOAD: BOOL;
    PV: INT;
END_VAR
VAR_OUTPUT
    QU, QD: BOOL;
    CV: INT;
END_VAR

IF Reset THEN
    CV:= 0;
ELSIF LD THEN
    CV:= PV;
ELSE
    IF NOT (CU AND CD) THEN
        IF CU AND (CV < PVmax) THEN
            CV:= CV+1;
        ELSIF CD AND (CV > PVmin) THEN
            CV:= CV-1;
        END_IF;
    END_IF;
END_IF;
QU:= (CV >= PV);
QD:= (CV <= 0);
END_FUNCTION_BLOCK
```

IEC

NOTE PVmax resp. PVmin refer to the implementation dependent limits of a counter.

Figure 13 – Standard FB CTUD according to IEC 61131-3

```
CLASS CTUD_OO
VAR
   m_PV: INT;
   m_CV: INT;
END_VAR

METHOD Reset
    m_CV:= 0;
END_METHOD

(*Load and Reset could be combined, since most time, we want to do
both*)
METHOD Load
VAR_INPUT
    PVIn: INT;
END_VAR
    m_PV:= PVIn;
END_METHOD

METHOD Count_Up: BOOL
VAR_INPUT
    CU: BOOL;
END_VAR

IF CU AND (m_CV < PVmax)THEN
   m_CV:= m_CV+1;
END_IF
Count_Up:= (m_CV >= m_PV);
END_METHOD

METHOD Count_Down: BOOL
VAR_INPUT
    CD: BOOL R_EDGE;
END_VAR

IF CD AND (m_CV > 0)THEN
   m_CV:= m_CV-1;
END_IF
Count_Down:= (m_CV <= 0);
END_METHOD
```

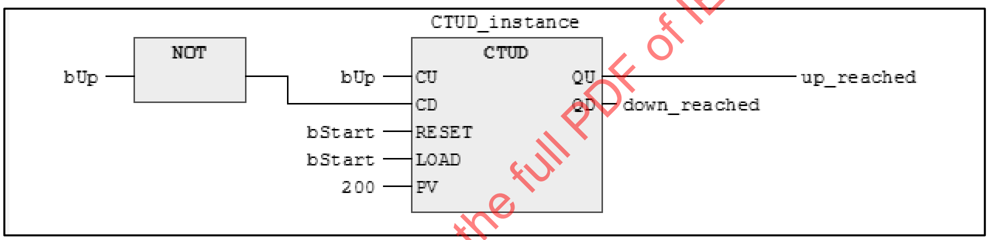
IEC

Figure 14 – Functionblock CTUD object oriented version

```
(*Call of a traditional instance*)

CTUD_instance(Load:= bStart, Reset:= bStart, PV:= 200, CU:= bUp, CD:=
NOT(bUp), QU => up_reached, QD => down_reached);


(*Call of an OO-instance*)

IF bStart THEN

   CTUD_OO_Instance.Reset();

   CTUD_OO_Instance.Load(200);

ELSE

    IF bUp THEN

        up_reached:= CTUD_OO_Instance.Count_Up();

    ELSE

        down_reached:= CTUD_OO_Instance.Count_Down();

    END_IF

END_IF
```

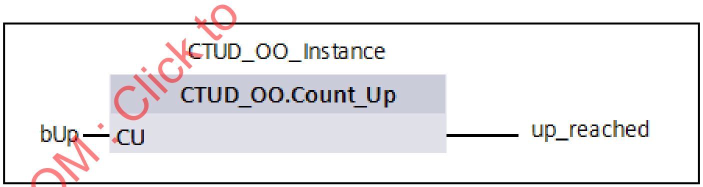
IEC

Figure 15 – Call of standard and OO-Functionblock in ST



IEC

Figure 16 – Call of standard function block in FBD



IEC

Figure 17 – All of a method in FBD

```
FUNCTION_BLOCK CTUD EXTENDS CTUD_OO
VAR_INPUT
    CU, CD: BOOL R_EDGE;
    Reset, LOAD: BOOL;
    PV: INT;
END_VAR
VAR_OUTPUT
    QU, QD: BOOL;
    CV: INT;
END_VAR

IF Reset THEN
    Reset();
ELSIF LD THEN
    Load(PVin: = PV);
ELSE
    IF NOT (CU AND CD) THEN
        QU: = Count_Up(CU:= CU);
        QD: = Count_Down(CD:= CD);
    END_IF;
END_IF;
PV:= m_PV;
CV:= m_CV;
END_FUNCTION_BLOCK
```

IEC

Figure 18 – Synthesis: a traditional function block derived from an OO-function block

The object oriented approach has several advantages. First of all, the object oriented version is less complex. One source of confusion is avoided in the OO implementation: it is not possible to call the function block with both inputs CU and CD being TRUE. Figure 15 shows both a call of a standard FB and the call of an OO FB in the language ST. Note that the OO-Function block needs also a less number of variables.

When used however, the traditional function block still shows some advantages. Especially in graphical languages, the call of one instance of a function block (Figure 16) is easier to handle and easier to read as the call of several methods (Figure 17).

Both approaches are possible solutions. So the traditional function block still is useful. Developers should therefore consider offering a traditional as well as an object oriented variant of the same function block type. It would then be useful to use the object oriented version as a basis for implementing the standard function block as shown in Figure 18.

7.7.3 Usage of class variable

Typically, all access to the variables in a class from outside the class is done via the methods of the class. Direct access to variables in a class should be avoided. However, for convenience or for performance reasons, it is possible to grant direct access to the data in a class. The access to the variable is defined with one of the following keywords: PUBLIC, PROTECTED, PRIVATE and INTERNAL.

- Private variables: variables which are accessible only from within the class itself.
- Protected variables: variables which are accessible from within the class and its derived classes.

- Public variables: variables which are accessible from everywhere.
- Internal variables: variables which are accessible from the local namespace.

It is recommended

- not to use PUBLIC,
- to limit the usage of PROTECTED,
- to use methods for variable access.

Example:

The class below scales a value up to a preset scaling factor and offset (linear equation). The scaling factor is PRIVATE.

Due to that the setting of the scaling is performed by a method. In this method the scaling factor is checked and only valid factor is accepted. Therefore the scaling method SCALE_VALUE does not need to check the factor on each invocation. The current scaling can be retrieved via GET_* methods.

```
CLASS SCALER
  VAR PRIVATE
    FACTOR: INT:= 1;
    OFFSET: INT;
  END_VAR

// Perform scaling according to a linear equation
  METHOD PUBLIC SCALE_VALUE: INT
    VAR_INPUT  VALUE: INT;  END_VAR
      SCALE_VALUE:= FACTOR * VALUE + OFFSET;
  END_METHOD

  METHOD PUBLIC SET_SCALING: BOOL
    VAR_INPUT
      SET_FACTOR: INT;
      SET_OFFSET: INT;
    END_VAR
    IF (0 = SET_FACTOR) THEN            // invalid scaling
      SET_SCALING:= FALSE;        // do not accept
    ELSE          // valid scaling
      FACTOR:= SET_FACTOR;
      OFFSET:= SET_OFFSET;
      SET_SCALING:= TRUE;         // accepted
    END_IF
  END_METHOD

  METHOD PUBLIC GET_FACTOR: INT
    GET_FACTOR:= FACTOR;
  END_METHOD

  METHOD PUBLIC GET_OFFSET: INT
    GET_OFFSET:= OFFSET;
  END_METHOD
END_CLASS
```

7.7.4 Usage of inheritance

Inheritance is a possibility to reuse a code without copying the code. A class is derived from another class by using the keyword EXTENDS. The derived class contains all variables and methods from the base class, and may add new variables and methods, as well as override some methods of the base class.

Although inheritance is historically one of the characteristic features of object oriented programming, the use of inheritance should be limited. Since the derived class has more information about the base class than other classes, inheritance is conflicting with the principle of information hiding. Nowadays composition is typically favored over inheritance. That means a class that wants to make use of the code of another class should better use a local instance of the class instead of inheriting from that class.

Inheritance allows a class to be derived from only a single base class. Usage of composition or interface allows a class to be designed from multiple origins.

However, in some cases, the use of inheritance may be justified, especially the use of inheritance from an abstract base class.

For example, see the following base class function block ETrig (Edge Triggered function block).

```
FUNCTION_BLOCK ABSTRACT ETrig
VAR_INPUT
  xExecute: BOOL;
END_VAR
VAR_OUTPUT
  xDone: BOOL;
  xBusy: BOOL;
  xError: BOOL;
END_VAR
VAR
  xPreviousExecute: BOOL;
  xPreviousAbortInProgress: BOOL;
  xAbortInProgress: BOOL;
END_VAR
METHOD PROTECTED ABSTRACT CyclicAction
  // implementation has to set the variables xDone and xError
appropriately
END_METHOD
METHOD PROTECTED ABSTRACT ResetOutputs
END_METHOD
METHOD PROTECTED ABSTRACT Start
END_METHOD
METHOD PROTECTED ABSTRACT Abort
END_METHOD

IF NOT xAbortInProgress THEN
  IF NOT xBusy THEN
    IF NOT xExecute AND xPreviousExecute THEN
      ResetOutputs();
    ELSIF xExecute AND NOT xPreviousExecute THEN
      xBusy:= TRUE;
      xPreviousExecute:= TRUE;
      Start();
    END_IF
  END_IF

  IF xBusy THEN
    CyclicAction();
    IF xDone THEN
      xBusy:= FALSE;
    END_IF
  END_IF

  IF xBusy THEN
    IF xError AND NOT xPreviousAbortInProgress THEN
      xAbortInProgress:= TRUE;
```

```
    END_IF
  END_IF
END_IF

IF xAbortInProgress THEN
  Abort();
    IF NOT xAbortInProgress THEN
      xBusy:= FALSE;
    END_IF
END_IF
xPreviousAbortInProgress:= xAbortInProgress;

END_FUNCTION_BLOCK
```

This function block is designated to work as an abstract base for all kinds of function blocks with a similar behavior: a function block that is started on a rising edge event on an input, runs for one or more cycles and produces a rising edge on an output when done.

A function block derived from this base class has to implement the abstract methods CyclicAction, ResetOutputs, Start and Abort, and it has to trigger the variables xDone (if the action is successfully completed) and xError (if the action cannot be completed).

In the main body the derived function block has to call the main body of the base function block.

The derived function block does not have to care about the state machine, since the implementation of the state machine is implemented in the base class.

Therefore, the behavior of ETrig-derived function blocks is always the same.

For example, if the cyclic action can be completed in one cycle, then the function block will completely execute in one cycle, and produce the done-Signal in the same cycle, without losing a cycle being in busy state.

Multiple ETrig-derived function blocks connected may be executed within the same cycle.

It is easy to create function blocks, which make use of this base implementation. All you have to do is to override and implement the protected methods of the base implementation. The behavior model will work in the same way for all of them.

One example for function blocks, which could be derived from this model, is file accessing.

The blocks below are derived from the block above and get all the properties.

The following code piece shows how to use these function blocks:

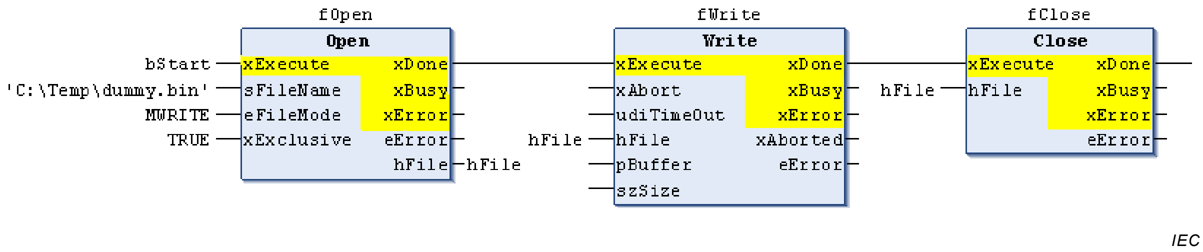


Figure 19 – Use of function blocks derived from ETrig-Base, the base function block interface is highlighted

Thus, the implementer of an ETrig-function block is guided in the implementation. The user of an ETrig-function block can be certain about its behavior.

7.7.5 Usage of override

In addition to simply add new methods, a derived class may also change the implementation of base class methods, by overriding the base class method.

As stated above, composition is mostly described as the favorable choice compared to inheritance. This also affects the usage of override. Since inheritance should best be restricted for abstract classes, override should best be restricted for abstract methods.

For example, the ETrig-Function Block as defined in 7.7.4 offers four abstract methods to override. These may be implemented with an empty implementation, but have to be implemented.

7.7.6 Usage of interfaces

Interfaces provide the possibility to handle different function blocks or classes with common properties in a common way.

An interface is simply a set of method prototypes. A method prototype defines the external interface of a method containing its inputs, outputs and inouts, but not the code, and not the temporary variables.

Example for a very simple interface, for an error providing object:

```
INTERFACE IErrorObject
METHOD getErrorDescription: STRING END_METHOD
METHOD getLastErrorTimeStamp: DATE_AND_TIME END_METHOD
METHOD getTypeOfError: UINT END_METHOD
END_INTERFACE

CLASS A IMPLEMENTS IErrorObject
…
END_CLASS
```

An interface can be seen as a contract between two classes. Class A implements the interface IErrorObject and class B makes use of this interface. The concrete implementation of the interface and the internal representation of the two classes is irrelevant for the use of the interface.

Class B may for example implement the following interface which uses the interface IErrorObject:

```
CLASS B
```

```
METHOD AddErrorObject
      VAR_IN_OUT
            newObject: IErrorObject;
      END_VAR
END_METHOD
// will check if any registered object is in error state,
// and will handle according to the state
// returns TRUE if one or more objects are in state Error
// or FatalError

…
END_CLASS
```

Note that Class B does not have to know the concrete implementation of the interface IErrorObject, but works with every class that implements the IErrorObject-Interface. This approach makes it much simpler to change the concrete implementation of one class without touching the concrete implementation of other classes.

7.8 Recursion within programmable controller programming languages

As stated in IEC 61131-3, recursive calls of POUs are vendor-specific. Though recursion of functions is not recommended, it may be useful for handling recursive data types, like linked lists.

Because of run-time performance, it is recommended that the program shall be checked for possible recursion during commissioning time. This requires the ability to detect possible recursion by traversing and checking the static call hierarchy tree.

With object oriented features, and especially with the introduction of dynamic binding (TBD. reference), it is no longer possible to determine a static call hierarchy. In this case a dynamic check of the deepest possible call stack is recommended (stack check).

7.9 Multiple invocations of a function block instance

Some programmable controllers allow multiple invocations which includes multiple assignments to any input of a function block instance.

This complication can lead to unexpected effects. For example the following errors can occur:

- Edge detection is not working properly;
- The second call overwrites instance data unintentionally.

On the other hand, there is no reason why a function block instance implemented in a digital computing entity could not be invoked from different locations, with different input values in each invocation.

An example of multiple invocations in ST is as follows:

```
VAR aFB: FUNCTION_BLOCK_TYPE; END_VAR
...
IF aBooleanExpression
THEN aFB( IN:= 1, ...);
ELSE aFB( IN:= 0, ...);
END_IF;
...
```

Here a function block instance named `aFB` is expected to have a Boolean input variable named “IN”. Depending on the current value of some Boolean expression this function block instance is invoked with either a value of `0` or a value of `1` for this input. There are two invocations of the function block instance `aFB` with different values assigned to an input but both never called in one program cycle. In a system that allows multiple assignments this would not cause any problems; but in a strict single assigning system the IF statement would have to be reformulated to

```
aFB(IN:= aBooleanExpression, ... );
```

Depending on the complexity of a function block and the number of inputs that depend on calculated values, the invocation of function blocks becomes more and more complicated and hides the intended purpose of the function block. If in the example above the input IN would not have the type BOOL but the type INT, a binary selection function SEL would have been required.

```
aFB(IN1:= SEL(G:= aBooleanExpression, IN0:= 0, IN1:= 1), IN2:=...);
```

Strict single assignment might also increase the number of intermediate variables necessary to decouple the assignment of values to the inputs of function blocks.

Similar examples of multiple assignments may be found in SFCs where an instance of a function block may be invoked in more than one action.

An appropriate example for the usage of multiple invocations of function block instances is the implementation of code synchronization means, such as semaphores, monitors or appointments.

NOTE These function blocks have to be manufacturer- or user-defined since there are no standard function blocks for code-synchronization provided in IEC 61131-3.

IEC 61131-3 does not stipulate whether a system uses strict single invocation or allows multiple invocations. Both have pros and cons.

With a single invocation rule there is one, and only one, place where the input variables of a function block instance will be assigned a value, which increases the reliability and maintainability of the software.

With multiple invocation of a function block instance, programmable controller programs may be maintained more easily because of a lower level of nesting of functions, and may have better responsiveness and processing capacity by avoiding superfluous calculations.

7.10 Language specific features

7.10.1 Edge-triggered functionality

Several mechanisms for rising and falling edge-triggered functionality are defined in IEC 61131-3:

– as function block inputs ;
– as R_TRIG and F_TRIG function blocks;
– as positive (P) and negative (N) transition-sensing contacts and coils for the LD language.

7.10.2 Edge-triggering in LD language

Many programmable controller systems use cyclic execution to implement sampled-data control. In the IEC 61131-3 notation, this is accomplished by associating the program that implements the control algorithm with a periodic TASK. Users should be aware that in such systems, in worst-case conditions, the effect of a change of state in an edge-triggered physical input may not appear at the system outputs until two scan cycles later. This can be illustrated by the simple LD network and timing diagram shown in Figure 20.

```
 |     IN1       OUT1   |
 +----|P|------(  )---+
 |                    |
```

IEC

a) Example network

```
           +----------------------------...
 IN1       |
----------+

 Input  ++        ++        ++        ++
 Scan   ||        ||        ||        ||
--------++--------++--------++--------++--
...
 Program +----+     +----+     +----+
 Scan    |    |     |    |     |    |
----------+    +-----+    +-----+    +---...

 Output         ++        ++        ++
 Scan           ||        ||        ||
--------------++--------++--------++--...

                          +---------+
 OUT1                     |         |
----------*---------------+         +--...
          |<--- Delay --->|
```

IEC

b) Worst-case timing

Figure 20 – Timing of edge triggered functionality

7.10.3 Use of edge-triggered function blocks

The `R_TRIG` and `F_TRIG` function blocks exhibit different behaviour following “cold restart". The `Q` output of an `R_TRIG` instance can be set to `1` on the first invocation but the `Q` output of an `F_TRIG` instance always requires at least two invocations before being set to `1`. This behaviour can be explained as follows.

– Since the default value of Boolean variables is `0`, then if the `CLK` input is `1` on the first invocation, it means that the input has changed its value from `0` to `1` i.e. a rising edge has been detected and thus the `Q` of an `R_TRIG` instance is set to `1`.

– In the case of an `F_TRIG` instance, the `CLK` input shall first be detected as being a `1` before a change of state from `1` to `0` can be detected. Thus, at least two invocations are needed.

An interesting use of `R_TRIG` as a “first-cycle detection mechanism” follows from the above description. If the `CLK` input to an `R_TRIG` instance is the constant `1` (or `TRUE`), the output `Q` will be true only on the first invocation since no subsequent change of state from `0` to `1` will be possible. This may be used as a first-cycle detect mechanism, for example,

```
VAR firstCycle: R_TRIG; END_VAR
firstCycle( CLK:= TRUE );
IF firstCycle.Q THEN (* first cycle only *)
  ....
ELSE
  ...
END_IF;
....
```

7.10.4 Use of EN/ENO in functions and function blocks

IEC 61131-3 defines the Boolean input `EN` and output `ENO`, which can be used to control the execution of functions and function blocks. Typically, these variables are used to provide

Boolean "power flow" through the functions in the LD language. However, they can also be used in the other languages and are especially useful in eliminating ambiguities that might be caused by the use of the graphical execution control.

Figure 21 shows an application of this principle to portions of the FBD body of the STACK_INT function block example. Combinations of the Boolean inputs PUSH, POP, and R1 are used in conjunction with the EN and ENO variables to achieve unambiguous execution control without jumps and labels.

According to IEC 61131-3, the ENO output of a function is to be reset to FALSE (0) upon the occurrence of an error condition in the execution of the function. This feature can be used to prevent the propagation of erroneous values through subsequent functional evaluations. However, the use of ENO for extensive run-time error reporting and diagnosis is not recommended; instead, the procedures described in 8.7.2 of this document should be employed.

```
IF R1 THEN OUT:= 0; PTR:= -1; ...etc.
ELSIF (POP & NOT EMPTY) THEN PTR:= PTR - 1; EMPTY:= (PTR < 0);
     ...etc.
ELSIF (PUSH & NOT OFLO) THEN PTR:= PTR + 1; OFLO:= (PTR = NI);
     ...etc.
END_IF;
```

IEC

a) ST language without EN/ENO

```
      +-------+             +-------+
      |:=     |             |:=     |
R1---|EN  ENO|------------|EN  ENO|--------...etc.
 0---|       |--OUT    -1--|       |--PTR
      +-------+             +-------+

                         +-------+        +-------+
         +---+           |   -   |        |   <   |
R1-----O|  & |---------|EN  ENO|-------|EN  ENO|---------...
POP-----|    |    PTR--|       |--PTR--|       |--EMPTY--...etc.
EMPTY--O|    |      1--|       |    0--|       |
         +---+           +-------+        +-------+

                         +-------+        +-------+
         +---+           |   +   |        |   =   |
R1-----O|  & |---------|EN  ENO|-------|EN  ENO|---------...
PUSH----|    |    PTR--|       |--PTR--|       |--OFLO---...etc.
OFLO---O|    |      1--|       |   NI--|       |
         +---+           +-------+        +-------+
```

IEC

b) FBD language with EN/ENO

Figure 21 – Execution control example

7.10.5 Language selection

Implementation language should be selected depending on the characteristic of each language shown in Table 3.

Table 3 – Characteristics of the languages

Language	Representation	Characteristic
SFC (Sequential Function Chart)	Graphical representation	High-level representation of activity flow chart where concrete conditions of transitions or actions are implemented by other languages. It is suitable for POUs which mainly controls supervisory processing sequence or machine state.
LD (Ladder Diagram)	Graphical language	Traditional graphical language which simulates physical relay circuit. All of instructions and function block instances are basically invoked in every cycle of execution unless JUMP or RETURN instruction is used. It is suitable for POUs which mainly implements logical operations of boolean signals.
FBD (Function Block Diagram)	Graphical language	A kind of data flow diagram, where input / output of function block are inter-connected with each other. All of instructions and function block instances are basically invoked in every cycle of execution unless JUMP or RETURN instruction is used. It is suitable for a POU which mainly implements continuous data processing.
ST (Structured Text)	Textual language	Pascal like textual language which support what are called “Structured statements” that means sequence, selection (IF.. THEN, CASE), and loop (FOR, WHILE). Execution of instructions and function block instances itself is controlled by selection and loop. It is suitable for a POU which implements algorithmic operations.
IL (Instruction List)	Textual language	Low level textual language like assembler. The use of this language has been marked deprecated in IEC 61131-3:2013. Not recommended for new designs.

7.11 Namespaces

7.11.1 General

Namespace has been introduced in IEC 61131-3:2013 to be used to avoid identifier ambiguities and to provide a syntactical means for a library or module concept.

A library may be written by the author of the project himself, or by another author of the same company, by the vendor of the PADT, by the vendor of the PLC, or even by a third party. In some projects all kinds of libraries are combined. To put an order into that, namespaces should be used.

Namespaces may be hierarchical. That means, a namespace may contain other namespaces.

There should be a logical relation between the functions, function blocks or classes in a namespace. For example the namespace FILE_ACCESS could contain function blocks for File-Open, File-Close, File-Read, File-Write etc.

7.11.2 Usage of global namespace

Global namespace is defined as the root namespace which has a global scope, and has no specific name. All of standard functions and function blocks defined in the IEC 61131-3 belong to the global namespace. User should avoid defining their functions, function blocks, and data types in the global namespace.

7.11.3 Usage of INTERNAL

The keyword INTERNAL provides a possibility to hide local information within a namespace from outside access.

7.12 Use of SFC elements

7.12.1 General

SFC elements allow a clear graphical and logical representation of the sequential structure of control programs and parts of control programs.

An SFC consists of one or more networks of steps and transitions. Associated with each step is a set of actions. An action represents the operations associated with one or more steps, and may consist of

– a collection of instructions in the IL language;
– a collection of statements in the ST language;
– a collection of rungs in the LD language;
– a collection of networks in the FBD language;
– a SFC elements; or
– a Boolean variable.

7.12.2 Action control

The execution of an action is affected by the states of its associated steps combined with the effects of their action qualifiers.

As defined in IEC 61131-3, the action control block contains a logic description of the effects and interactions of action qualifiers. In order to gain a better understanding of the action control, a copy (instance) of the action control block is assigned to each action. Therefore, the action control block is a constituent part of an action: it defines the execution of the action (one could also say that it “controls” the execution).

In this way, the execution of an action can be enabled for the duration of a step, executed just once (“pulsed”), enabled for an indefinite time (“set” or “stored”), disabled (“reset”), enabled after a time delay, or enabled for a limited time.

IEC 61131-3 requires that the functionality of action control blocks, but not necessarily the blocks themselves, shall be implemented in an SFC-compliant programmable controller system. This functionality can be described informally by the following rules.

a) The behaviour (function) of an action in parts of the program is mainly determined by the following features:
   – status of step flag;
   – action qualifier;
   – action;
   – action control block.
b) The action control block describes (logically) the operating method (function) and interplay of action qualifiers. (An action can be defined with different action qualifiers within one program or part of a program.) Action qualifiers are
   N non-stored;
   S set/stored;
   R overriding reset;
   P pulse;
   L time-limited;
   D time-delayed;
   DS delayed and stored;

SD stored and time-delayed;

SL stored and time-limited;

P1 rising edge pulse (when entering step);

P0 falling edge pulse (when leaving step).

Figures 23a) and 23b) of IEC 61131-3:2013 represent the logical flow of the activation as a wiring diagram. Figure 23 of IEC 61131-3:2013 defines two action control features: 1) control with "final scan" as shown in Figure 23a) and 2) control without "final scan" as shown in Figure 23b).

A typical use of the P0 qualifier is to substitute for a "final scan" where needed in systems which do not implement feature 1).

The existence of two types of permissible action control ("final scan" versus no "final scan") limits program portability. Users should check that the action control features supported are identical between two different programmable controller systems before attempting to port an SFC from one system to another.

The influence of the P0 and P1 actions on the "Q" output of the action control block is inconsistent between the "final scan" and "no final scan", respectively. That is, the Q output is false (0) during the action of P1 or P0 with final scan, and true (1) during the action of P1 and P0 without final scan. Users should be especially careful of this difference before attempting to port an SFC between systems providing different treatment of "final scan".

Any section of an SFC network usually comprises a succession of action blocks, which often only differ in the choice of the attribute. Furthermore, several different actions can be connected to one step, as illustrated in IEC 61131-3. Therefore, the following general observations can be made.

– Each step can be connected to several action blocks.
– However, each action block is connected to one step.
– Each action is connected with exactly one action control block.
– Each action control block is connected with one or more action blocks.

Figure 23b) of IEC 61131-3:2013 illustrates the transformation of the SFC net in Figure 23a) into a block wiring diagram, corresponding to the realization of the action blocks of the SFC as an (optimized) wiring net. The actual executable code generated need not be identical to this figure, as long as its behaviour is functionally equivalent.

c) There may only occur one activation of a time-dependent action qualifier (D, L, SD, DS, SL) per action block.

d) The activation of an SL qualifier in one action block does not allow the processing of an SD qualifier in the same block, and vice versa.

Timing diagrams of the functional characteristics of the various qualifiers are given in the following subclause.

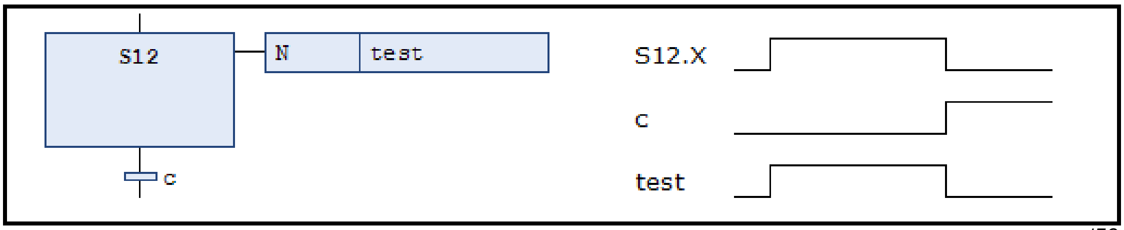
7.12.3 Boolean actions

An action that operates on a Boolean variable only is called a Boolean action. In this special case, the behaviour of an action is established by the direct assignment of the status of output Q to the Boolean variable.

Figure 22 shows all the qualifier functions for Boolean variables, based on the description of action qualifiers in IEC 60848. In these figures, it is assumed that `test` has been declared as a variable of type BOOL.

If a combination of various qualifiers is required, it is recommended to refer to the detailed description of the action control block in order to gain a better understanding of the resulting operations.

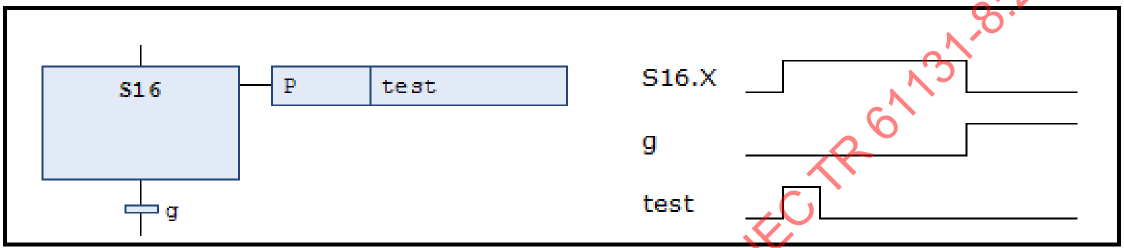
The use of P1 and P0 qualifiers will always result in a false value of an associated Boolean action when “final scan” is implemented. When “no final scan” is implemented, the associated Boolean action will be true for the single scan when the step is entered or exited, respectively. Therefore, the use of these qualifiers for Boolean actions is not recommended.



IEC

The Boolean variable “test” is set as long as the step S12 is active.

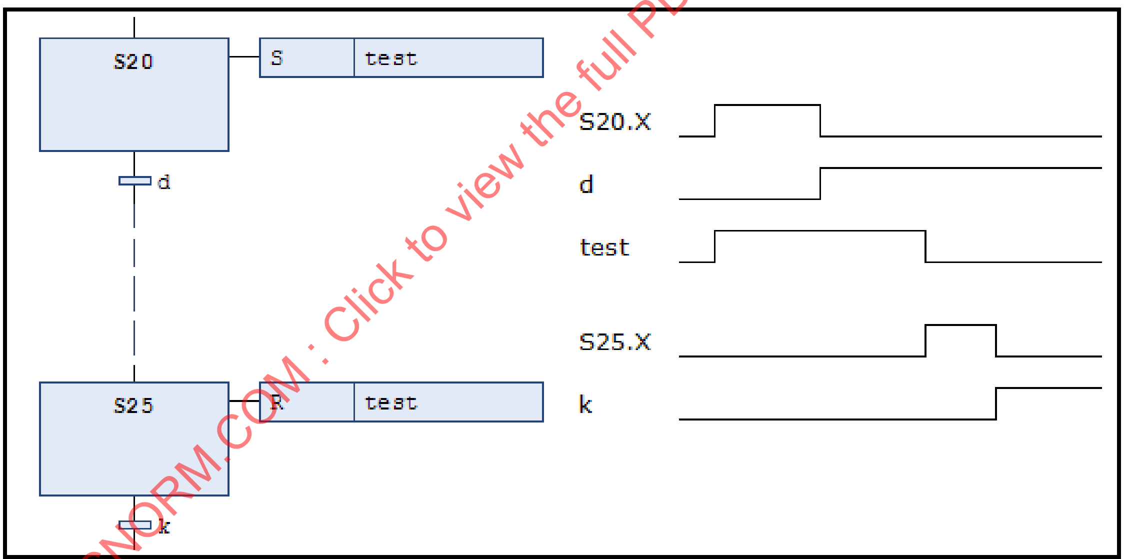
a) N (non-stored) action



IEC

As soon as the step is active, the Boolean variable is set for one operating cycle.

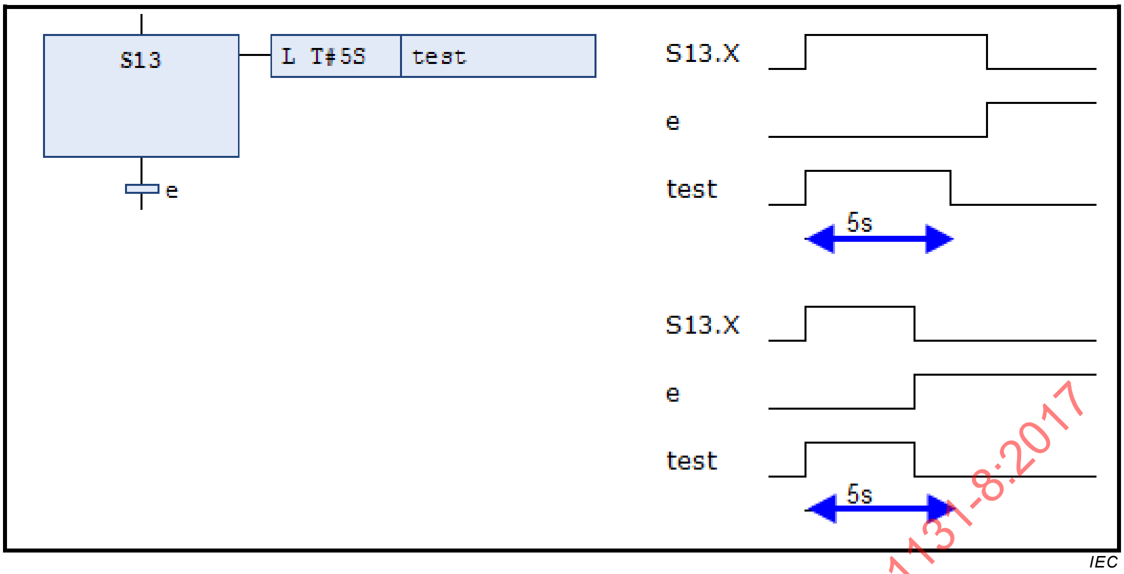
b) P (pulse) action



IEC

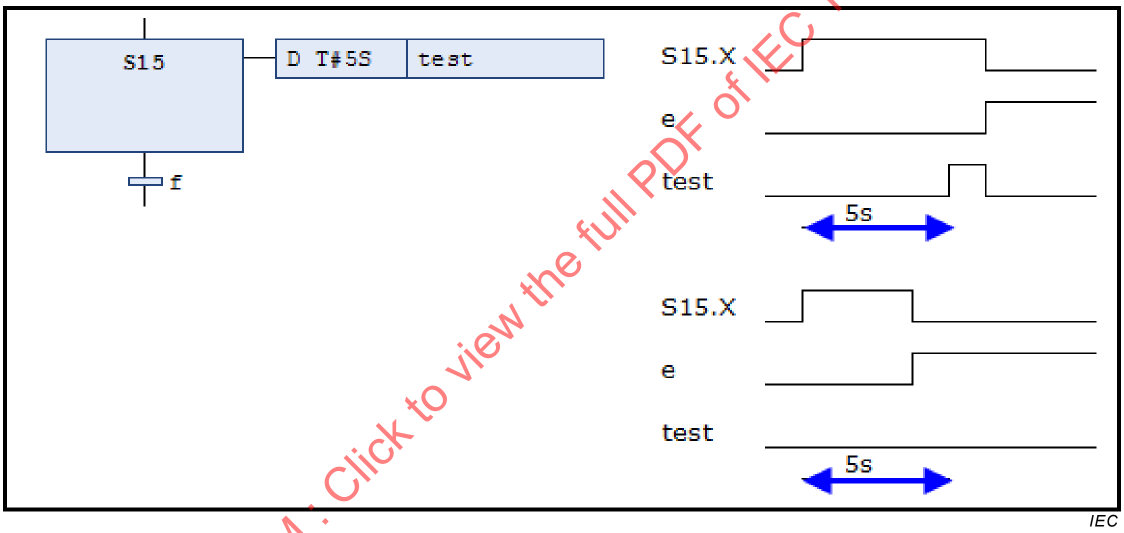
The Boolean variable is set and stored as soon as the step is active.
It can only be reset by means of the R-qualifier.

c) S/R (set/reset) action



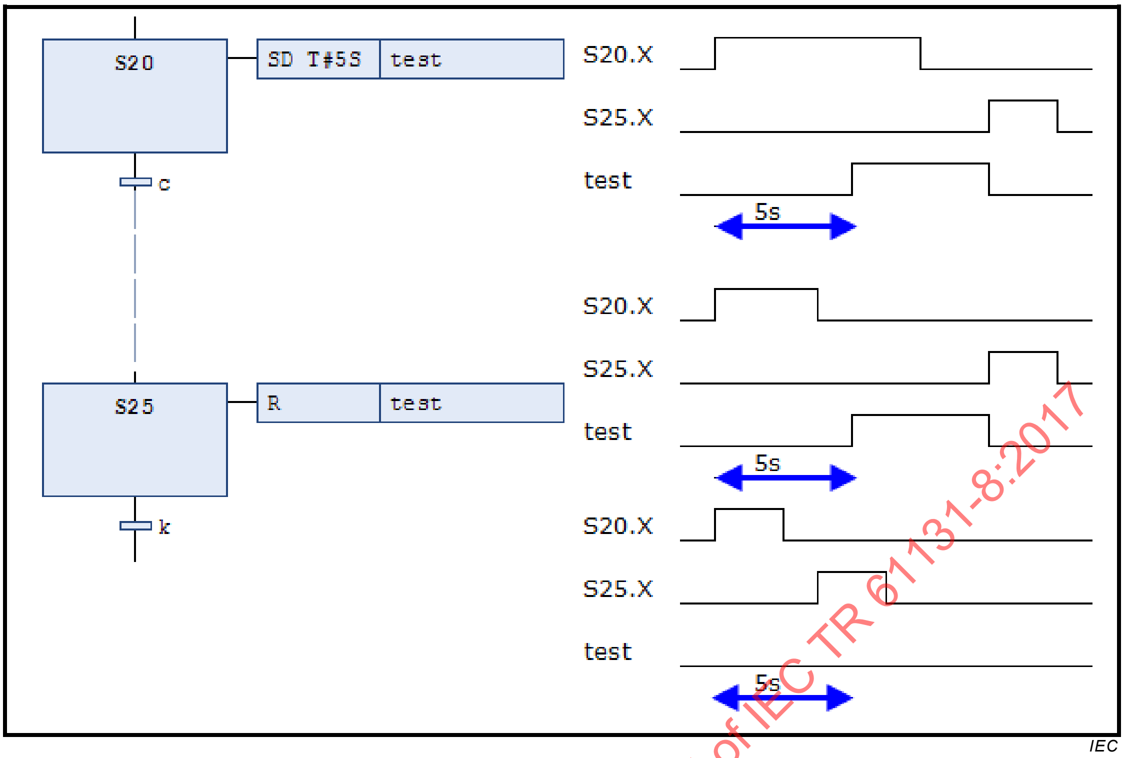
The Boolean variable is set for a preset length of time, as long as the corresponding step is active.

d) L (time-limited) action



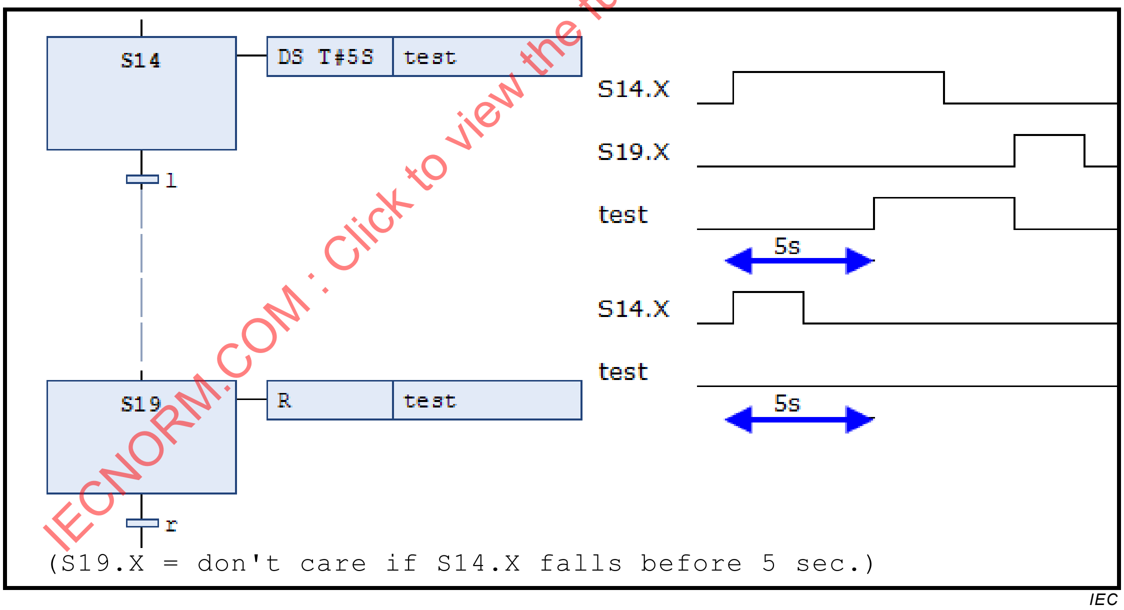
The Boolean variable is set after a preset time has elapsed and remains set for as long as the corresponding step is active.

e) D (time-delayed) action



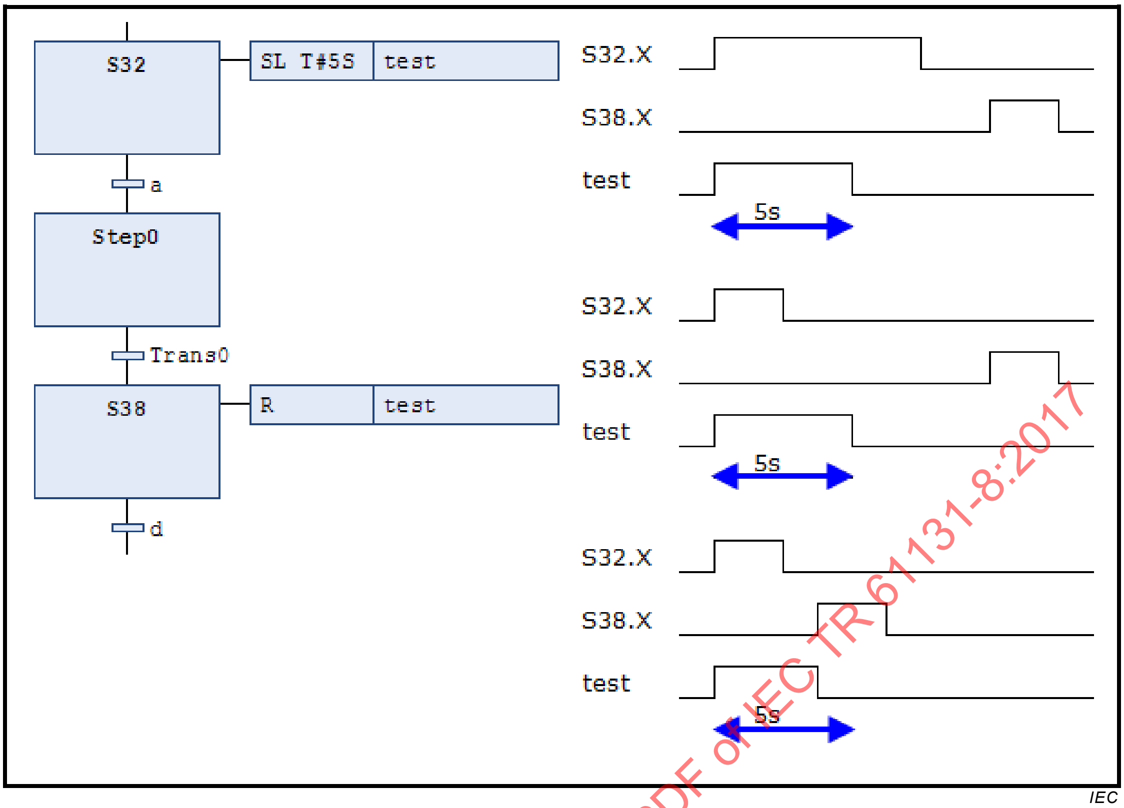
The action is stored, and the Boolean variable is set when a preset period of time has elapsed after step activation, even if the step becomes inactive.
This condition persists until the action is reset.

f) SD (stored and time-delayed)



In this case, as for the SD qualifier, the Boolean variable is set with a time delay.
It differs from the SD qualifier in that the corresponding step shall remain active during the time delay.

g) DS (time-delayed and stored)



The Boolean variable is set and stored for a preset length of time as soon as the corresponding step is active.

h) SL (stored and time-limited) action

Figure 22 – Timing of Boolean actions

7.12.4 Non-SFC actions

In addition to Boolean variables, more complex algorithms can also be programmed within actions, as illustrated in Figure 23.

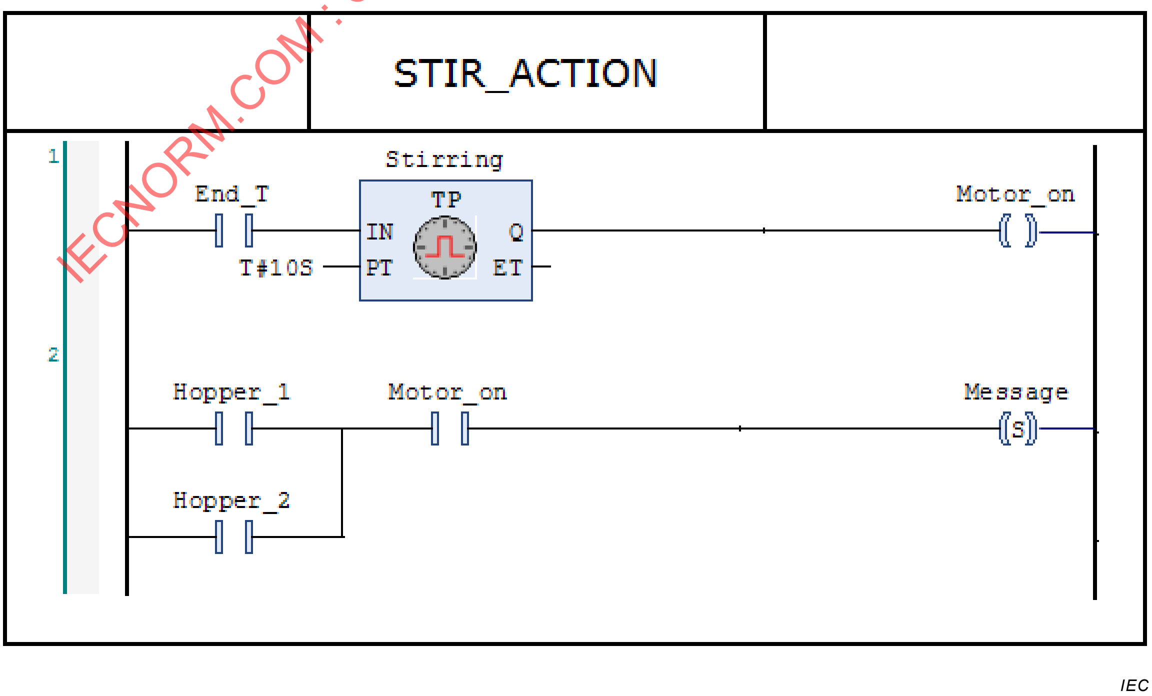


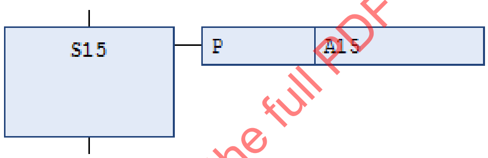
Figure 23 – Example of a programmed non-Boolean action

The processing of such actions is dependent on the status of the `Q` output of the corresponding action control block. While output `Q` is `TRUE`, the code of the action is continuously enabled to be executed, under the control of the enclosing program organization unit such as a program or function block. Upon the transition of `Q` from `TRUE` to `FALSE`, the code is executed one final time. During this final execution, the corresponding step flag is already set to `0`.

In the development of IEC 61131-3, the above specification for programming of actions was formulated in order to maximize user control of the following aspects:

– maintaining the consistency of computed data by assuring that the programmed action cannot be interrupted prematurely;
– assuring that a computed output bit can be forced to the correct value, for example, by evaluating a step flag, when the action is terminated.

In order to avoid unexpected results, programmers shall bear in mind that each action is evaluated at least twice (once with `Q = 1` and once with `Q = 0`). Figure 24 illustrates this point in the context of the `P` (pulse) qualifier. In this example, the purpose is to maintain an integer count, `S15_CT`, of the number of times step `S15` is activated. As shown in Figure 24a), this is to be accomplished by action `A15`. Figure 24b) illustrates an incorrect body for this action, which will result in `S15_CT` being equal to twice the number of activations, since `A15` will be invoked once upon activation of `S15` and again in the next scan. Figures 24c) and d) illustrate possible solutions to this problem in the ST and FBD languages, respectively.



IEC

a) SFC fragment

```
S15_CT:= S15_CT + 1;
```

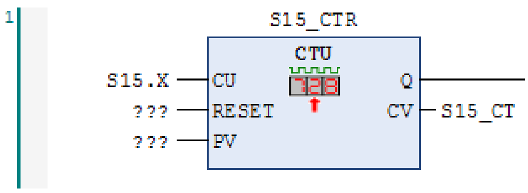
IEC

b) Incorrect body for action A15 (ST language)

```
S15_CTR(CU:= S15.X);
S15_CT:= S15_CTR.CV;
```

IEC

c) Correct body for action A15 (ST language)



IEC

d) Correct body for action A15 (FBD language)

Figure 24 – Use of the pulse (P) qualifier

7.12.5 SFC actions

An SFC action is an action that contains one SFC network. Just as with other actions, an SFC action can be associated with several steps.

The use of SFC actions is NOT RECOMMENDED due to the possibility of independent operation of “child” SFC actions even when execution of the “parent” has been suspended. Deletion of this feature from future editions of IEC 61131-3 is anticipated.

7.12.6 SFC function blocks

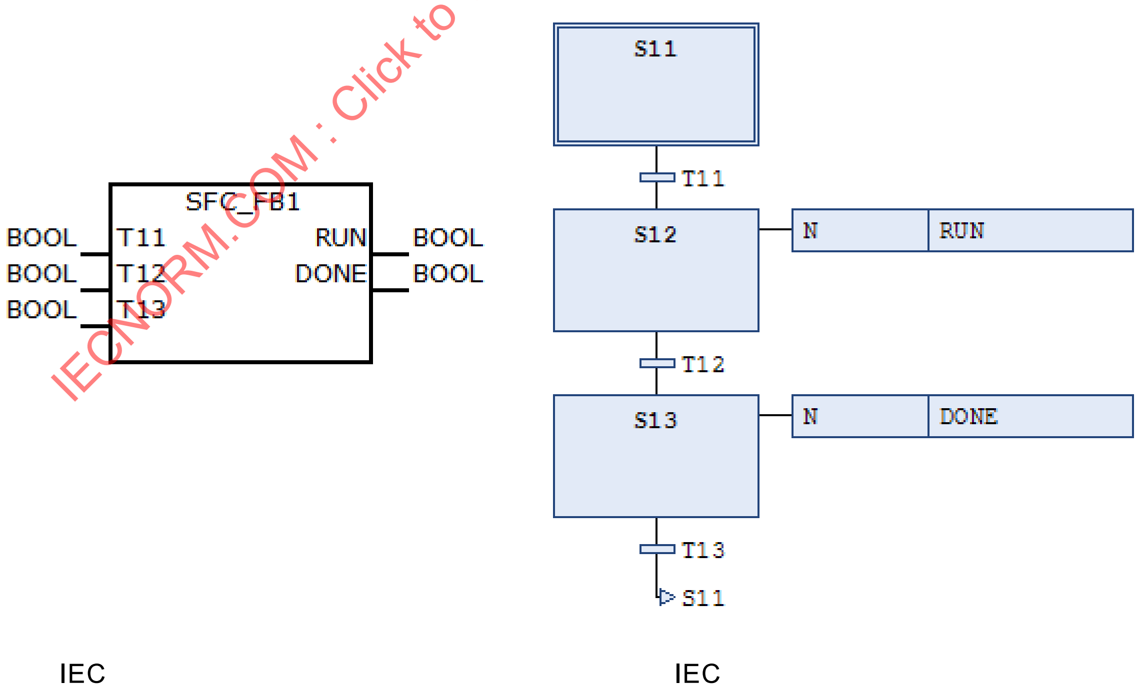
The body of a function block can be defined by an SFC; this will be denoted as an SFC function block. The use of such function blocks follows the rules common to all function blocks. The user can declare several instances of the same function block type; each will have a private data area, which also contains information about the current state of the SFC. Each instance then can work independently, using the same SFC algorithm but with separate, hidden internal state variables.

A declaration of a simple SFC function block is shown in Figure 25. It will be noted that the SFC contained in an SFC function block is defined in the same (textual or graphical) way and under the same restrictions as for ordinary SFCs. In particular, each SFC network of an SFC function block shall contain an initial step.

Instances of SFC function blocks can be invoked in the same manner as all other function blocks. Input variables that control the evolution of the SFC can be declared and used. For example, a Boolean transition condition for starting the evolution can be passed via input variables such as T11 in Figure 25. An action declared locally in the body of the SFC function block can also access these function block variables.

Execution of an instance of an SFC function block is invoked in the same way as for any function block, depending on the invocation mechanisms available in the language used for programming the instance of the function block. The body of the function block is then executed using an SFC scan algorithm such as the one outlined in 7.12.2.

Due to the restriction that no recursion of program organization units is allowed, an SFC function block may not contain an invocation of any function block instance of the same type.



a) **External interface** b) **Body**

Figure 25 – An SFC function block

7.13 Scheduling, concurrency and synchronization mechanisms

7.13.1 Operating system issues

Programmable controllers have always served the purpose of controlling a multitude of parallel industrial processes in real time. Prior program execution strategies assumed that the controller can execute its entire program as quickly as necessary to meet the real-time constraints of the fastest process being controlled. This approach is easy to implement and use, and is appropriate for many control systems. However, a single controller may no longer be able to meet ever-increasing requirements for speed and complexity with a straightforward, single program scan.

IEC 61131-3 offers additional language elements to aid the user in controlling program execution to meet these new requirements. In order to assist the programmer in making most effective use of these features, the most important terms are briefly explained below.

In this discussion, the term TASK refers to a set of PROGRAM or FUNCTION_BLOCK blocks that runs independently of and (quasi-)parallel to other tasks.

The term TASK is used as a keyword in IEC 61131-3 in order to define the temporal activation of tasks, i.e., a TASK statement specifies the run-time features of the associated tasks. The following short example is adapted from IEC 61131-3:

```
TASK SLOW_1 (INTERVAL:=t#20ms, PRIORITY:= 2);
TASK FAST_1 (INTERVAL:=t#10ms, PRIORITY:= 1);
TASK PER_2 (INTERVAL:=t#50ms, PRIORITY:= 3);
TASK INT_2(SINGLE:= z2, PRIORITY:= 1);
PROGRAM P1 WITH SLOW_1: F(x1:= %IX1.1,...);
PROGRAM P2: G(FB1 WITH SLOW_1, FB2 WITH FAST_1);
PROGRAM P4 WITH INT_2: H(FB1 WITH PER_2);
```

The characteristics of four tasks are defined explicitly by the first four lines. The first three define the characteristics of tasks whose execution is to be scheduled periodically at the specified intervals, while the fourth specifies a task triggered by the rising edge of a Boolean variable `z2`.

Thus, program `P1` and function block `FB1` in program `P2` are associated with a task that is running every 20 ms with priority of 2, defined as `SLOW_1`. The faster task `FAST_1` only consists of function block `FB2`. Program instance `P2` is assigned to a task that is always active when no other task is executable. Program `P4` is associated with task `INT_2`, triggered on the rising edge of variable `z2`, while function block `FB1` in `P4` is associated with the periodically executing task `PER_2`.

The strategy by which tasks are selected for execution determines the responsiveness of the system and the efficiency with which its processing capacity is utilized. Responsiveness and efficiency are usually improved when a running task can be interrupted by another one (“preemptive scheduling”). These aspects are addressed in 7.13.2.

Other issues in multi-processing are the control of exclusive access to operational equipment and the provision of data transfer between tasks. These aspects are addressed in 7.13.3 and 7.13.4 respectively.

As shown in Table 4, multi-user/multitasking systems and real-time responsiveness and efficiency are usually mutually exclusive.

Table 4 – Differences between multi-user and real-time systems

Multi-user systems	Real-time systems
Fair distribution of the processor time on the running tasks yields high production rate of the system	The task execution has to keep up with the external process. Prompt reaction to external events and the execution of the responding task before a new occurrence is necessary
Program running times cannot be calculated	Bounds on program running times (for example, response time) have to be guaranteed
Storage administration can swap out a task onto secondary storage (dynamic allocation of resources)	Allocation of resources is fixed (the reloading of a swapped-out task into the primary storage takes time)
The task priority changes dynamically, for example, according to the CPU time usage	Priorities are fixed. The task with the highest priority gets the processor for as long as it requires. The task can only be interrupted by another task of higher priority or when it explicitly relinquishes control
Data files are usually in directories as tree structures. Large data files may be distributed on non-contiguous disc sectors	High disc I/O speed requirements typically dictate the storage of data on contiguous sectors
Bus throughput is likely to be a bottleneck	Bus interrupt response is likely to be a bottleneck

7.13.2 Task scheduling

7.13.2.1 General

IEC 61131-3 introduces two methods to schedule the execution of tasks, called preemptive and non-preemptive scheduling. Both methods are employed in many real-time systems. Preemptive scheduling is mostly used by large programmable controllers and process computers, while non-preemptive scheduling can be found in smaller programmable controllers.

Non-preemptive scheduling expects that a programmable controller processor can change from one task to another (usually called a context switch) only after a complete execution of all POUs associated with the current task. After complete execution of all such POUs, the next task that will be activated is the one that currently has the highest priority or has waited the longest time if several tasks are waiting at the same priority level. Preemptive scheduling allows a context switch at any time. Whenever a task of higher priority than the current task is enabled, depending on the inputs of the associated TASK block, the entire state of the currently executing task will be saved by the processor and the task will be suspended. Then the other task of higher priority will execute. This procedure is called a “preemptive context switch” as the processor (or at least all its registers) has to be emptied in favour of the task with higher priority. At a later time the new task is expected to relinquish the processor. Thereafter, the suspended task will regain its context (all register values as they had been at the preemption point) and will resume execution. Preemptive scheduling may even have nested contexts: at a given point in time there can be several lower priority tasks that have been suspended.

7.13.2.2 Performance effects

Both methods of scheduling (preemptive and non-preemptive) try to execute pending tasks with higher priority before tasks with lower priority levels. The main difference is the amount of time a higher priority task has to wait when it requests execution. With preemptive scheduling this time is usually very short. Depending on the underlying operating system and processor speed, the necessary housekeeping time for a context switch ranges from several microseconds to milliseconds. This implies that the reaction time (i.e., the time from setting a task ready to execute by means of its TASK block to the beginning of execution) of a system using preemptive scheduling is short, at the expense of some additional processing time to save and restore context data.

When a non-preemptive programmable controller switches from one task to another, there is a negligible amount of data in the processor that has to be saved. The context switch itself would be short, probably shorter than in the case of a preemptive context switch; but this changeover from one task to another will never be done while a POU is still executing. Hence, with non-preemptive scheduling, the worst-case reaction time is at least as long as the time the longest running function block or program might need for a complete execution. As this not only holds for standard FBs but also for all user-defined FBs and programs, it may be difficult to determine a maximum limit for the reaction time of a non-preemptive scheduling system.

NOTE The use of scheduling mechanisms other than those defined in IEC 61131-3 can be needed in some applications which require a guaranteed maximum reaction time.

Usually, preemptive task switching causes a small performance degradation (as the context switches consume time) but improves the reaction time significantly. This can be seen by comparing the second column of Example 1 in Table 63 of IEC 61131-3:2013 with the corresponding Example 2. Here it can be seen that with preemptive scheduling (Example 2), the high-priority function block `FB2` in program `P2` (`P2.FB2@1`) never has to wait, while with non-preemptive scheduling (Example 1) it has to wait for 4 ms in two out of three activations (at t = 10 ms and t = 20 ms).

7.13.2.3 Concurrency effects

Satisfying the rules for data concurrency given in IEC 61131-3 may be substantially more complex in a programmable controller system with preemptive task scheduling than in a system with non-preemptive scheduling. In some configurations and programs, a preemptive system may not be able to guarantee that these rules are satisfied. The manufacturer of a programmable controller should provide sufficient information to enable a user to determine the means and extent to which these rules are satisfied.

7.13.3 Semaphores

7.13.3.1 General

Semaphores serve the purpose of controlling the access to certain commonly used equipment (printer, disc, etc.). Only one task at a time can have access to such a resource. Other accesses are rejected or delayed according to the principle that first the program tests if the semaphore is already assigned; if not, the program requests it. With a successful assignment of a semaphore to a task, this task will be given permission to have access to the associated data or equipment. The semaphore is then released after conclusion of the operations.

NOTE 1 A semaphore function block has not been defined in IEC 61131-3:2013 due to the difficulty of addressing this construct in that standard. This 7.13.3 addresses the nature of these difficulties.

NOTE 2 Manufacturers are encouraged to encapsulate means (such as semaphores) to control shared access to operational equipment within function blocks if possible and to submit such implementations through their National Committees for inclusion in future editions of IEC 61131-3.

According to IEC 61131-3, “the WHILE and REPEAT statements shall not be used to achieve inter-process synchronization, for example, as a ‘wait loop’ with an externally determined termination condition. The SFC elements defined in 7.12 shall be used for this purpose”. This interdiction applies to semaphores, which are also an “inter-process synchronization” mechanism. Violation of this rule can lead to severe degradation in software reliability, portability and reusability; in the case of semaphores, it may lead to unanticipated and untraceable suspension of the execution of a program organization unit.

Another mechanism to implement “inter process synchronisation” may be done by using OOP methods like shown in 7.7.4 by the example of ETRIG.

7.13.3.2 Deadlocks

Incorrect use of semaphores may result in a task waiting for operational equipment that another task is using at that moment and vice versa. As a consequence, both tasks will in principle wait forever, which is called a deadlock.

Four conditions may lead to a deadlock.

a) The deadlocked tasks have exclusive access to the corresponding operational equipment.
b) The deadlocked tasks already have operational equipment assigned to them while they wait for further operational equipment.
c) It is not possible to remove the assignment of operational equipment to any task until the task releases it.
d) There is a closed chain of tasks, in which each task is assigned operational equipment which is required by the next task in the chain.

Programmable controller PADTs may eventually support methods for the avoidance, prevention, detection and/or removal of deadlocks. In the meantime, the programmer and system designer should be aware of the potential existence of deadlocks.

NOTE Encapsulation of semaphores in function blocks can simplify the search for deadlocks when they do occur and will facilitate the future application of deadlock prevention algorithms.

7.13.4 Messaging

7.13.4.1 General

Messaging (inter-process communication) describes methods for tasks to exchange data with each other and (possibly) synchronize their operations. In general there are three methods for such data transfer: global storage, mailboxes and queues.

7.13.4.2 Global storage

Two tasks can exchange data via a storage area that is available for both. The protocol of the accesses has to be included explicitly in the program. Apart from this, there is great liberty for the implementation of a data transfer. This facility can be implemented by the VAR_GLOBAL and VAR_EXTERNAL constructs. A better recommendation is to assign global variables to the input, output or inout parameters of PROGRAM instance in the construct of the resource configuration as mentioned in 7.2.5 and 7.6.2.

7.13.4.3 Mailboxes and queues

A mailbox is a kind of a "letter-box" for data to be transferred. In contrast to data transfer via global storage, there is a protocol for transferring the data to the mailbox and making it available to other partners. The transfer of the data can be carried out via various media, for example, common storage areas, backplanes, or communication networks.

A queue is a kind of data storage, where the data elements are read in the same order as they were entered. An example is the character buffer of a terminal, which contains the characters in the same order than the user entered them.

Either mailboxes or queues, or both, may be used by the communication function blocks described in 7.13.5 of this document. However, the particular mechanism employed is completely invisible to the user, who need only be concerned with the externally specified interface and functionality of these function blocks.

7.13.5 Time stamping

The term "time stamping" comes originally from the field of data bases. With time stamping, each data element of the data base contains not just the value of the data, but also information on the time of input.

Time stamping of data is easily supported by including the value of the data in a variable of a structured data type that also includes its time stamp. For instance, a time-stamped data type that can contain one REAL value could be declared according to the rules given in IEC 61131-3 as follows:

```
TYPE STAMPED_REAL:
 STRUCT
  VALUE: REAL;
  STAMP: DATE_AND_TIME;
 END_STRUCT;
END_TYPE
```

If the data changes, the old element is not removed from the data base; instead, the old element is marked invalid and the new element is entered along with its time stamp. The advantage of this approach is that changes within the data base can be traced relatively easily by following the time stamps; also, new values that are in error can be deleted and older, correct values restored.

Time stamps have also been used extensively in distributed control systems for continuous industrial processes for similar reasons, namely, for establishing historical process log data; for determining the validity of data by its age; and for restoring erroneous data to a previously valid value.

7.14 Communication facilities in ISO/IEC 9506-5 and IEC 61131-5

7.14.1 Overview

NOTE The term variable is used throughout 7.14.1 in place of the term parameter used in IEC 61131-5.

IEC 61131-5 defines a set of standard function blocks that can be used in the IEC 61131-3 languages for communication among programmable controllers as well as for communication with host devices. This will enable the user to program network-independent communication functionality in any programmable controller system complying to IEC 61131-3 and IEC 61131-5.

ISO/IEC 9506-5 specifies the requirements for a programmable controller serving as a Virtual Manufacturing Device (VMD) in the Manufacturing Message Specification (MMS) context, including the mapping of certain IEC 61131-3 elements into the MMS context.

7.14.2 Data representation

7.14.2.1 Structure with explicit layout

Data exchange between different systems often requires a different memory layout of the data, e.g. they are packed or always starting at an even byte address.

To enable symbolic addressing of variables in that case too, the standard offers the possibility to define datatypes with a defined memory layout of its elements.

This definition also allows a memory overlapping with different datatypes. A simple use case for using overlapping may be an explicit reading access to the sign of a number.

The locations (addresses) of the elements of a structured type can be defined relative to the beginning of the structure by using the keyword AT and a relative byte location. The declaration may include gaps in the memory layout.

Example: Explicit layout without Overlapping

```
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
```

If overlapping is required the keyword OVERLAP has been given in the declaration.

Example: Explicit layout with Overlapping

```
TYPE
 Values_and_sign:
  STRUCT OVERLAP
        one AT %B0: INT;
        sign1 AT %X0.0: BOOL;
        two AT %B2: DINT;
        sign2 AT %X2.0: BOOL;
  END_STRUCT;
END_TYPE
```

With that definition the sign of the 2 values is direct readable.

Overlapping of strings is beyond the scope of the document, because the internal representation of the string is not defined by the document.

There may be some disadvantages concerning the performance because of less than ideal data access of the processor, but on the other hand there is no need to convert the data by an own program.

7.14.2.2 Usage of endian conversion functions

Varying kinds of processors are addressing its data in a different way.

The terms "big endian" and "little endian" indicate the internal storage of a variable in the memory that is needed by the processor for optimal addressing of variables.

Endianess is the ordering of the bytes within a longer data type or variable.

- The data values of data in big endian are placed in the memory locations beginning with the leftmost byte first and the rightmost byte last.
- The data values of data in little endian are placed in the memory locations beginning with the rightmost byte first and the leftmost byte last.

Endianess functions can be used for data exchange between different systems to convert to the appropriate storage format.

The endianess conversion functions convert a variable to and from the implementer specific, internally used endianess of the PLC from and to the requested endianess.

Using the partial access with the lower number returns the lower value part independently of the specified endianess, i.e. independently of the endianess the bit offset 0 addresses the rightmost bit of a data type.

There are 4 endianess functions:

- Conversion to big endian data format
  Example:
  ```
  A:= TO_BIG_ENDIAN(B);
  ```
- Conversion to little endian data format
  Example:
  ```
  B:= TO_LITTLE_ENDIAN(A);
  ```
- Conversion from big endian data format
  Example:
  ```
  A:= FROM_BIG_ENDIAN(B);
  ```
- Conversion from little endian data format
  Example:
  ```
  A:= FROM_LITTLE_ENDIAN(B);
  ```

7.14.2.3 Usage of validate functions

Arithmetical operations with REAL or LREAL numbers can lead to improper results like Not-a-Numer (NaN) or infinite (+Inf, -Inf).

The validate functions check if the given input parameter contains a valid value. So it is possible to recognize errors and react directly in the program.

The overloaded function IS_VALID is defined for the data types REAL and LREAL. In the case the real number is not a valid real bit pattern the result of the validate function is FALSE.

Some implementers may support additional data types with the validate function IS_VALID. The result of these extensions is implementer specific.

7.14.3 Communication channels

A programmable controller can establish communication with a remote programmable controller device using IEC 61131-5 by opening a communication channel.

Lower layers within the network protocol software stack ensure that data is transferred to and from the remote node without error if possible. For example, if a communication error occurs and data is corrupted, it may be re-transmitted automatically.

All of the details concerning the low-level control of the network such as messaging queuing, framing, etc. are hidden by the IEC 61131-5 function blocks. The applications programmer need only be concerned with the application-specific details of the channel.

The specifications in ISO/IEC 9506-5 and IEC 61131-5 allow only a one-level hierarchy for addressing loadable objects like IEC 61131-3 programs at a remote controller. Therefore, unique access variable names are required inside a configuration to guarantee unique addressing of remote programs through a given channel, although resource scope allows the use of equal names in different resources.

7.14.4 Reading and writing variables

A programmable controller can read from, and write to, variables within other remote programmable controllers supporting IEC 61131-5, providing they are either directly represented variables as defined in IEC 61131-3 or variables declared to be accessible using the VAR_ACCESS ... END_VAR construction defined in IEC 61131-3. This reading and writing are performed using the READ and WRITE FBs, respectively, as described below.

It is recommended that the VAR_ACCESS method always be used when accessing variables in remote programmable controllers because it is then possible to use meaningful names for variables. There is always the likelihood that I/O physical addresses will be changed if the remote programmable controller program is modified. It may be convenient to fix VAR_ACCESS names for a number of different programmable controller programs.

7.14.5 Communication function blocks

IEC 61131-5 defines a set of function blocks that can be controlled and accessed within an IEC 61131-3 program in exactly the same manner as other blocks. These function blocks provide the following functionality.

CONNECT – provides a local “channel ID” (not an “identifier” in the IEC 61131-3 sense) for communicating with a remote device. The remote device should have a unique name. The channel ID provided by this function block can be used by other communication function blocks to identify remote devices.

STATUS – polls a remote device for device verification information. It is important that a programmable controller check the status of a remote device periodically to ensure that the remote programmable controller is behaving correctly.

USTATUS – allows a programmable controller to receive the remote device’s verification information, including its physical and logical status. The remote device shall have the capability to send its device verification information whenever it changes.

READ – polls a remote device for the values of one or more variables. A list of variables can be given as inputs to the block. After a short delay due to the time to transfer the request and response across the network, the values of the remote variables are presented on the function block outputs.

The READ block does not provide an input variable to control the poll rate. The application program should re-trigger the block to initiate a new poll.

WRITE – writes one or more values to one or more variables in a remote device. A list of variable names can be specified to identify variables in a remote device. The remote device is selected by the R_ID variable obtained from the CONNECT block.

USEND – sends the values of one or more variables to a URCV block in a remote application program. The remote application program can use the values transferred to the URCV function block in the normal way. An R_ ID variable ensures that the local USEND block sends values to the correct URCV block in the remote device.

URCV – receives the values of one or more variables from an associated USEND block.

SEND – provides an interlocked data exchange with an RCV block in a remote device. The SEND block sends the values of one or more variables to a remote RCV block corresponding to the channel ID from a CONNECT block and the R_ID variable. The remote programmable controller application program, on receiving the values, loads a set of values as a response, which is then returned to the SEND block. The SEND and RCV blocks are provided for applications where there is a requirement for interlocking as well as data exchange between the local and remote programs.

RCV – receives the values of one or more variables from an associated SEND block.

The time between the New Data Ready (NDR) indication from the RCV block and the request to transmit the response is determined by the application program. Since, in many communication systems, the communication channel can block if too many responses are pending, programming constructs that respond as quickly as possible to the NDR signal are recommended.

ALARM – sends values of one or more variables to a remote device identified by the channel ID and an event identifier when an event condition is detected. The alarm can be characterized by severity level. This block expects the remote device to acknowledge the reception of the alarm.

NOTIFY – this is the same as the ALARM block except an acknowledgement from the remote device is not expected.

7.15 Deprecated programming practices

7.15.1 General

The effects of programming technique on software quality should be considered when choosing among the options made available in IEC 61131-3. This 7.15 indicates some of the more significant of these effects and recommends programming practices to achieve higher software quality. These practises should be used and of course the deprecated features of IEC 61131-3 should be avoided.

7.15.2 Global variables

Excessive use of global variables contradicts the principles of encapsulation and hiding discussed in 6.6.1.1 and can greatly reduce software reliability, maintainability and reusability. In particular, the writing of global variables from more than one program location should be avoided. It is recommended that global variables should be used (if at all) only for

– supplying values of “global” interest like
  - to or between program organization units;
  - defining access paths for open communication;
  - physical I/O.

Global variables should never be used for communicating data between asynchronously running programs if data concurrency is an important issue. SEND/RCV or USEND/URCV FBs should be used in such cases to assure concurrency.

7.15.3 Jumps in FBD/LD language

As noted in 6.3, the FBD/LD language is modelled on connected hardware circuits (ICs), i.e., function block instances in networks are modelled as working in parallel. In such a context, jumps over or between networks can be confusing to the user thus reducing software reliability, maintainability and adaptability. It is recommended that conditional execution of FBD/LD networks should be controlled

– by placing such networks in actions, or
– by using the EN (enable) input and ENO output, or
– by using methods.

Proper operation can then be verified more easily than locating and checking the state of jump conditions.

7.15.4 Dynamic modification of task properties

Every task is defined by some properties e.g its priority. It is well known from the theory of operating systems that the dynamic modification of task priorities may have unpredictable effects on the execution of parallel programs, including the generation of deadlocks. Since IEC 61131-3 TASKS are not necessarily mapped direct to operating system tasks, additional implementation dependent effects may occur. Therefore, it is highly recommended that the application programmer does neither use dynamic task priority modification nor modification of another property in an IEC 61131-3 compliant system.

7.15.5 Execution control of function block instances by tasks

The association of tasks with function block instances and its effects on data concurrency are described in IEC 61131-3. The programmer should be aware of the fact that use of this feature may produce data consistency errors during program run time. The guidelines provided by the IEC 61131-3 implementer should be consulted to determine the mechanisms provided to assure data consistency. Since these mechanisms are implementation-dependent, programs using this feature may not be portable between different IEC 61131-3 compliant systems.

7.15.6 WHILE and REPEAT constructs for interprocess synchronisation

WHILE and REPEAT statements shall not be used to achieve interprocess synchronization, for example, as a ‘wait loop’ with an externally determined termination condition. SFC elements shall be used for this purpose.

In order to avoid unanticipated and potentially dangerous effects, users are strongly cautioned to extend this prohibition to any case of inter-code synchronization. For instance, suppose it is desired to perform a scaling calculation at the rising edge of a `SAMPLE` input of the function block shown in Figure 26a) and provide a rising edge at a `DONE` output when the calculation is complete. Then the programming construct shown in Figure 26b) should not be used. To avoid this usage, the `SAMPLE` input can be changed to a rising-edge trigger as shown in Figure 26c) and the function block body shown in Figure 26d) can be used, or the SFC shown in Figure 26e) can be used with the interface shown in Figure 26a).

```
    +-----------+
    |  SCALE    |
BOOL---|SAMPLE  DONE|---BOOL
REAL---|IN        OUT|---REAL
REAL---|ZERO        |
REAL---|GAIN        |
    +-----------+
```

IEC

a) **Example interface**

```
(* DEPRECATED!!! *)
WHILE TRUE DO(* DO FOREVER *)
 WHILE NOT SAMPLE DO(* WAITING *)
 END_WHILE;
 OUT:= IN*GAIN+ZERO;
 DONE:= TRUE;
 WHILE SAMPLE DO(* WAIT AGAIN *)
 END_WHILE;
DONE:= FALSE;
END_WHILE;
```

IEC

b) **Incorrect construct**

```
       +------------+
       |  SCALE     |
BOOL--->SAMPLE DONE|---BOOL
REAL---|IN    OUT   |---REAL
REAL---|ZERO        |
REAL---|GAIN        |
       +------------+
```

IEC

c) **Alternative interface**

```
IF SAMPLE THEN
 OUT:= IN*GAIN+ZERO;
 DONE:= TRUE;
ELSE DONE:= FALSE;
END_IF;
```

IEC

d) **FB body for interface c)**

```
+-------+
|       |
|     +--------+
|     ||START||
|     +--------+
|       |
|       + SAMPLE
|       |
|     +----------+  +--+----------------------+
|     | SAMPLING |--|P1| OUT:= IN*GAIN+ZERO; |
|     +----------+  +--+----------------------+
|       +   |       | S| DONE                  |
|       |           +--+----------------------+
|       + NOT SAMPLE
|       |
|     +---------+  +-+-------+
|     | ENDING |--|R| DONE  |
|     +---------+  +-+-------+
|       |
|       + 1
|       |
+-------+
```

IEC

e) **Allowed SFC function block body for interface a)**

Figure 26 – Example of incorrect and allowed programming constructs

7.15.7 Expecting programs associated with a task to be executed sequentially

In the case that multiple program instances are associated with the same task, it is not specified that they shall be executed sequentially in IEC 61131-3. User should not expect them to be executed sequentially, even if some venders implement them to be executed sequentially and provide means for user to specify their execution order. It contradicts the principle that programs should be asynchronously executable POU which can be assigned to tasks independently. Also it impedes program portability between systems of different venders.

For instance, logic of a program should not be divided into multiple programs if their each logic is sensitive to their execution order within a cycle. When the logic of the program is huge and need to be divided into some portions. An option is dividing them into function blocks and invoking them from a single program.

7.16 REAL_TO_INT conversion functions

IEC 61131-3 states that “Conversion from type REAL or LREAL to SINT, INT, DINT or LINT shall round according to the convention of ISO/IEC/IEEE 60559, according to which, if the two nearest integers are equally near, the result shall be the nearest even integer....” It should be noted that the rounding procedure specified by ISO/IEC/IEEE 60559 may not operate in the same manner as the rounding functions defined in other programming languages. It may be possible that some microprocessors do not support this kind of rounding. Therefore the documentation of the manufacturer has to be considered.

7.17 Implementation dependant parameters

Implementation-dependent parameters mentioned in IEC 61131-3:2013, and the primary reference clause for each, are listed in Table 5 below.

Table 5 – Implementation-dependent parameters

Subclause of IEC 61131-3:2013	Parameters
6.1.2	Maximum length of identifiers
6.2	Syntax and semantics of pragmas
6.3.3	Syntax and semantics for the use of the double-quote character when a particular implementation supports Feature #4 but not Feature #2 of Table 6 of IEC 61131-3:2013.
6.4.2.1	Range of values and precision of representation for variables of type TIME, DATE, TIME_OF_DAY and DATE_AND_TIME
	Precision of representation of seconds in types TIME, TIME_OF_DAY and DATE_AND_TIME Initialization of Date datatypes Floating point according to ISO/IEC/IEEE 60559 Maximum length of strings Update / accuracy of time
6.4.2.1	Default maximum length of STRING and WSTRING variables Maximum allowed length of STRING and WSTRING variables
6.4.4.2 6.4.4.5 6.4.4.6	Maximum number of enumerated values Maximum number of array subscripts Maximum array size Maximum number of structure elements Maximum structure size Maximum range of subscript values Maximum number of levels of nested structures
6.5.2.2	Maximum number of variables per declaration Effect of using AT qualifier in declaration of function block instances
6.5.5.2	Number of hierarchical addressing of directly represented variables
6.5.5.3	Initialization of system inputs
6.5.6	Warm start behaviour (initial values) if variable is declared as neither RETAIN nor NON_RETAIN
6.5.6.2	Initialisation of variables on startup
6.6.1.1	Support of recursive calls of POUs and methods Maximum number of POU, methods, instances
6.6.1.2.2	Assignment string to shorter string Additional assignment rules for arrays

Subclause of IEC 61131-3:2013	Parameters
6.6.1.5	Values of outputs when `ENO` is `FALSE`
6.6.1.5	Function block input variable assignment when `EN` is `FALSE`
6.6.1.6	Additional conversion rules
6.6.1.7.2	Handling on overloaded type conversion
6.6.2.5	Effects of type conversions between time data types and other data types
6.6.2.5.1	Maximum number of inputs of extensible functions
6.6.2.5.2	Effects of type conversions on accuracy Error conditions during type conversions
6.6.2.5.8	Accuracy of numerical functions
6.6.2.5.13	Endianess
6.6.2.5.15	Validate function with other datatypes
6.6.3.4.2	Side effects with “not allowed” usages of input output parameters
6.6.3.5.4	`Pvmin`, `Pvmax` of counters
6.6.3.5.5	Effect of a change in the value of a `PT` input during a timing operation
6.7	Precision of step elapsed time Maximum number of steps per SFC
6.7	Maximum number of transitions per SFC and per step
6.7.4.3	Maximum number of action blocks per step
6.7.4.6	Access to the functional equivalent of the `Q` or `A` outputs
6.7.5	Transition clearing time Maximum width of diverge/converge constructs
6.8.1	Contents of `RESOURCE` libraries
6.8.1	Effect of using `READ_WRITE` access to function block outputs
6.8.2	Maximum number of tasks Task interval resolution
7.3.2	Maximum length of ST expressions
7.3.3.1	Maximum length of ST statements
7.3.3.3.3	Maximum number of `CASE` selections
7.3.3.4.2	Value of control variable upon termination of `FOR` loop
8.2.3	Restrictions on network topology
8.2.5.2	Evaluation order of feedback loops

Other implementation-dependent parameters such as the accuracy, precision and repeatability of timing and execution control features may have significant effects on the portability of programs too but are beyond the scope of the document.

So the developer has to pay attention to these parameters if the program code shall be portable between different PADT’s.

At least any program with such dependencies has to be compiled and tested within the target system.

8 Implementation guidelines

8.1 General

This clause describes implementation techniques consistent with the intent of IEC 61131-3. Other implementations are possible and allowed, as long as the functional requirements imposed by IEC 61131-3 are met.

8.2 Resource allocation

The resource, as described in IEC 61131, is the basic unit of IEC 61131-3 for storage of data and program code, the scheduling of code for execution, and the execution of the appropriate code when a POU, i.e., a program, function or function block, is invoked.

It is possible for an implementation to store in a resource just one copy of the executable code associated with each type of POU currently resident in the resource. On the other hand, a separate data area shall be provided for each instance of a function block or program. This includes sufficient data for all the variables and (possibly) SFC state information associated with the program or function block. In contrast, storage of variables for a function is temporary and only lasts for the duration of the function's execution; hence, storage for such data is typically dynamically allocated from a “stack” (first-in, last-out queue) or a “heap” (memory reserved for temporary allocation).

8.3 Implementation of data types

8.3.1 REAL and LREAL data types

In order to reduce the loss in precision that can occur with floating-point calculations, it may be necessary to convert all floating-point values stored in REAL data types into the double-width (LREAL) format before initiating a sequence of arithmetic operations, particularly if such calculations could involve the computation of relatively small differences between floating-point numbers. The results are then converted back to REAL format for storage.

There is a wide range of microprocessors, particularly digital signal processors (DSPs) that have their own internal floating-point formats. In such cases, the implementer shall limit the range of values supported by the REAL and/or LREAL implementation to those specified in IEC 61131-3 or shall specify new implementation-dependent data types, say DSP_REAL and DSP_LREAL for the native floating-point formats in addition to supporting the standard REAL and LREAL types.

Since the ISO/IEC/IEEE 60559 does not completely define all aspects of floating-point operations the following features are can either be language-defined or implementation-defined:

– The precision with which intermediate results are stored.

– The "rounding direction" applied when values have to fit into a smaller destination format.

– How floating-point exceptions are handled.

– Whether subnormal numbers are supported or not.

– Whether value-changing optimizations may be applied during translation of the programming language. Value-changing optimizations are optimizations that change the result of the operation in contrast to the result that would be computed when interpreting the source code literally.

  For example, applying the associative and distributive law to floating-point arithmetic expressions is a value-changing optimization (due to rounding losses).

Of these, IEC 61131-3 only specifies the rounding direction in case of conversions to integer values, which is to round to the nearest integer, and in case of a tie, to the nearest even integer. Even though IEC 61131-3 leaves the rounding direction open in all other cases, it is reasonable to assume that IEC 61131-3 intends the rounding direction to be the same in all situations.

Note that even though IEC 61131-3 states that this rounding direction is the "convention of ISO/IEC/IEEE 60559", in ISO/IEC/IEEE 60559, that rounding direction is only special insofar as all compliant implementations are required to provide it.

Furthermore, IEC 61131-3 makes all the other listed aspects language-defined behaviour, sometimes explicitly, sometimes by omission. As a result, program organization unit type definitions containing any number of floating-point operations may yield different results for different IEC 61131-3 compliant programming systems. Even worse, value-changing optimizations may be even compiler dependant.

8.3.2 Bit strings

Implementation of bit-string comparison operations should correspond to the explanation given in 7.1.10.

To eliminate ambiguity in the interpretation of the MIN and MAX functions, it is considered that their semantics can be derived from the comparison functions through the application of the following definitions.

- Let $\mathbf{M}$ be a set of bit-string data {b1, b2, b3, ..., bn}.
- (B = MAX(b1, b2, b3, ...bn)) $\equiv$ ((B $\in \mathbf{M}$) & $\forall$ 1 $\leq$ j $\leq$ n, $b_j \in \mathbf{M}$: (B $\geq$ b_j)); that is, saying that B is the maximum value of the set is equivalent to saying that it is a member of the set and it is greater than, or equal to, the value of any member of the set.
- (B = MIN(b1, b2, b3, ...bn)) $\equiv$ ((B $\in \mathbf{M}$) & $\forall$ 1 $\leq$ j $\leq$ n, $b_j \in \mathbf{M}$: (B $\leq$ b_j)); that is, saying that B is the minimum value of the set is equivalent to saying that it is a member of the set and it is less than, or equal to, the value of any member of the set.

8.3.3 Character strings

The maximum length and format of variable length strings is implementation-dependent. This implies that function blocks and algorithms using character-string data types may not be portable between programmable controllers, particularly if the maximum string length is significantly different.

There are two main techniques used for defining the character string length:

a) null character terminator is used, or
b) the length is stored at the head of the string.

The null character technique, as used in the ‘C’ language, may not be convenient if there is also a requirement to have null characters embedded within the string but does have the advantage that indefinitely long strings can be stored. The disadvantages of this concept are the long calculation time for finding this termination character. Storing the length in a byte or word limits the maximum string length to 255 or 65535 respectively, but does allow strings to hold null characters.

NOTE Information on character string data types is provided in 7.1.6 of this document.

8.3.4 Time data types

The storage and length of the time data types are implementation-dependent. This poses the possibility that functions and function blocks that use time-data types may not be portable. It is therefore required by IEC 61131-3 that the range and precision of values of time-data types be clearly specified by the implementer.

For example, the TIME (duration) data type might be represented by a 32-bit unsigned integer storing the duration as a count of milliseconds. This allows a TIME data type to define accurately any duration from 0 ms to approximately 49 days. However, this would not allow the computation of small time differences (less than 1 ms) between events, or to represent negative durations. Negative durations may be denoted as time-literals or result from operations on time-related values.

The use of floating-point representation for time data is not recommended because of the lack of precision in the fractional part of the value. For example, using a 32-bit unsigned integer, the duration of 30 days, 10 min, and 200 ms, i.e., `T#30d10m300ms` can be represented accurately; this is not possible with a 32-bit floating-point value.

It may be convenient to store DATE, TIME_OF_DAY and DATE_AND_TIME in the UNIX format in which the DATE value is held in a 32-bit unsigned integer as the number of seconds from midnight, 1 January 1970, and the TIME_OF_DAY value is held as the number of seconds from midnight. This gives dates and time of days up to the year 2106. However, this format does not allow events to be time-stamped with a precision better than 1 s.

Note, that issues may arise, if the implementation uses different resolutions among time-related types. Consider the implementation of DATE and TIME as given above and the operations on time-related values (IEC 61131-3:2013, 6.6.2.5.12) in the following.

First, (intermediate) results may overflow in an unexpected way. For instance, the strict evaluation of the expression "DATE#2001-01-01 – DATE#2000-01-01 + DATE#2000-01-01" yields an undefined result, since the intermediate result of the subtraction cannot be represented by type TIME.

Second, certain arithmetic equivalencies do no longer hold. For instance, the expressions (DATE#2000-01-01 – DATE#2000-01-01 + TIME#1ms) and (DATE#2000-01-01 + TIME#1ms – DATE#2000-01-01) appear to be equivalent, but will actually yield different results due to an intermediate loss in latter expression. The long time types (LTIME, LDATE, LTOD, LDT) do not raise such issues, as they all use the same resolution, which is 1 ns.

If a higher precision is necessary the values may be stored with the datatypes LDATE, LTIME_OF_DAY, LDATE_AND_TIME that are defined in IEC 61131-3:2013.

These datatypes allow the storage as 64-bit integer with the unit of nanoseconds. For LDATE and LDATE_AND_TIME the number of nanoseconds are counted from midnight, 1 January 1970 , resp. the number of nanoseconds from midnight for LTIME_OF_DAY. This gives a maximal accuracy of nanoseconds for at least 584 years.

But the accuracy may be implementer specific and updated every microsecond or millisecond.

8.3.5 Multi-element variables

Implementations of IEC 61131-3 may need to limit the number and size of array dimensions to accommodate performance and memory limitations of the programmable controller. For example a reasonable limitation for most applications is to limit array variables to no more than three dimensions.

Deeply nested array indexing may also need to be limited due to the increased complexity involved in resolving memory addresses within the programmable controller, especially if the variables use complex derived data types. An example of such deeply nested indexing is

```
loop.sp:= spList[loopParams[phase[recipe[job1]]]];
```