



Programmable controllers

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



This Australian Standard® was prepared by Committee IT-006, Industrial Process Measurement, Control and Automation. It was approved on behalf of the Council of Standards Australia on 17 November 2014.
This Standard was published on 19 December 2014.

The following are represented on Committee IT-006:

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

This Standard was issued in draft form for comment as DR AS IEC 61131.8:2014.

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

Keeping Standards up-to-date

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

Standards may also be withdrawn.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

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

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

Australian Standard[®]

Programmable controllers

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

Originated as AS IEC 61131.8—2004.
Second edition 2014.

COPYRIGHT

© Standards Australia Limited

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

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

ISBN 978 1 74342 933 4

PREFACE

This Standard was prepared by the Standards Australia Committee IT-006, Industrial Process Measurement, Control and Automation, to supersede AS IEC 61131.8—2004.

The objective of this Standard is to provide guidelines for the implementation of the programming languages defined in AS IEC 61131.3, in programmable controller systems and their programming support environments (PSEs).

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

This Standard is identical with, and has been reproduced from, IEC/TR 61131-8, Ed. 2.0 (2003), *Programmable controllers—Part 8: Guidelines for the application and implementation of programming languages*.

The principal differences between this and the previous edition have been made to align this edition with the current edition of AS IEC 61131.3.

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

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

References to International Standards should be replaced by references to Australian Standards, as follows:

<i>Reference to International Standard</i>	<i>Australian Standard</i>
IEC	AS IEC
61131 Programmable controllers	61131 Programmable controllers
61131-1 Part 1: General information	61131.1 Part 1: General information
61131-2 Part 2: Equipment requirements and tests	61131.2 Part 2: Equipment requirements and tests
61131-3 Part 3: Programming languages	61131.3 Part 3: Programming languages
61131-5 Part 5: Communications	61131.5 Part 5: Communications

The term ‘informative’ has been used in this Standard to define the application of the annex to which it applies. An ‘informative’ annex is only for information and guidance.

CONTENTS

1	General	9
1.1	Scope	9
1.2	Normative references	9
1.3	Abbreviated terms	9
1.4	Overview	10
2	Introduction to IEC 61131-3	10
2.1	General considerations	10
2.2	Overcoming historical limitations	12
2.3	Basic features in IEC 61131-3	13
2.4	New features in the second edition of IEC 61131-3	14
2.5	Software engineering considerations	14
2.5.1	Application of software engineering principles	14
2.5.2	Portability	17
3	Application guidelines	17
3.1	Use of data types	17
3.1.1	Type versus variable initialization	18
3.1.2	Use of enumerated and subrange types	18
3.1.3	Use of BCD data	19
3.1.4	Use of REAL data types	21
3.1.5	Use of character string data types	21
3.1.6	Use of time data types	22
3.1.7	Declaration and use of multi-element variables	22
3.1.8	Use of bit-string functions	23
3.1.9	Strongly typed assignment	24
3.2	Data passing	25
3.2.1	Global and external variables	25
3.2.2	In-out (VAR_IN_OUT) variables	26
3.2.3	Formal and non-formal invocations and argument lists	28
3.3	Use of function blocks	30
3.3.1	Function block types and instances	30
3.3.2	Scope of data within function blocks	31
3.3.3	Function block access and invocation	32
3.4	Differences between function block instances and functions	33
3.5	Use of indirectly referenced function block instances	33
3.5.1	Establishing an indirect function block instance reference	34
3.5.2	Access to indirectly referenced function block instances	35
3.5.3	Invocation of indirectly referenced function block instances	35
3.5.4	Recursion of indirectly referenced function block instances	38
3.5.5	Execution control of indirectly referenced function block instances	38
3.5.6	Use of indirectly referenced function block instances in functions	38
3.6	Recursion within programmable controller programming languages	39
3.7	Single and multiple invocation	39

3.8	Language specific features	40
3.8.1	Edge-triggered functionality	40
3.8.2	Use of EN/ENO in functions and function blocks	41
3.8.3	Use of non-IEC 61131-3 languages	42
3.9	Use of SFC elements	42
3.9.1	Action control	42
3.9.2	Boolean actions	44
3.9.3	Non-SFC actions	47
3.9.4	SFC actions	48
3.9.5	SFC function blocks	48
3.9.6	“Indicator” variables	49
3.10	Scheduling, concurrency, and synchronization mechanisms	50
3.10.1	Operating system issues	50
3.10.2	Task scheduling	51
3.10.3	Semaphores	52
3.10.4	Messaging	53
3.10.5	Time stamping	53
3.11	Communication facilities in ISO/IEC 9506/5 and IEC 61131-5	54
3.11.1	Communication channels	54
3.11.2	Reading and writing variables	54
3.11.3	Communication function blocks	55
3.12	Deprecated programming practices	56
3.12.1	Global variables	56
3.12.2	Jumps in FBD language	56
3.12.3	Multiple invocations of function block instances in FBD	56
3.12.4	Coupling of SFC networks	56
3.12.5	Dynamic modification of task priorities	57
3.12.6	Execution control of function block instances by tasks	57
3.12.7	Incorrect use of WHILE and REPEAT constructs	57
3.13	Use of TRUNC and REAL_TO_INT functions	58
4	Implementation guidelines	58
4.1	Resource allocation	59
4.2	Implementation of data types	59
4.2.1	REAL and LREAL data types	59
4.2.2	Bit strings	59
4.2.3	Character strings	59
4.2.4	Time data types	60
4.2.5	Multi-element variables	60
4.3	Execution of functions and function blocks	60
4.3.1	Functions	60
4.3.2	Function blocks	61
4.4	Implementation of SFCs	62
4.4.1	General considerations	62
4.4.2	SFC evolution	62
4.5	Task scheduling	63
4.5.1	Classification of tasks	63
4.5.2	Task priorities	63

4.6	Error handling	64
4.6.1	Error-handling mechanisms.....	64
4.6.2	Run-time error-handling procedures	65
4.7	System interface	67
4.8	Compliance	67
4.8.1	Compliance statement	67
4.8.2	Controller instruction sets	68
4.8.3	Compliance testing	68
5	PSE requirements	68
5.1	User interface.....	68
5.2	Programming of programs, functions and function blocks	69
5.3	Application design and configuration.....	70
5.4	Separate compilation	70
5.5	Separation of interface and body	71
5.5.1	Invocation of a function from a programming unit.....	71
5.5.2	Declaration and invocation of a function block instance	72
5.6	Linking of configuration elements with programs	73
5.7	Library management.....	75
5.8	Analysis tools	75
5.8.1	Simulation and debugging	75
5.8.2	Performance estimation	76
5.8.3	Feedback loop analysis.....	76
5.8.4	SFC analysis	76
5.9	Documentation requirements	79
5.10	Security of data and programs	79
5.11	On-line facilities	79
	Annex A (informative) Changes to IEC 61131-3, Second edition	80
	Annex B (informative) Software quality measures.....	90
	Annex C (informative) Relationships to other standards.....	92
	INDEX	93
	Bibliography	105
	Figure 1 – A distributed application	11
	Figure 2 – Stand-alone applications	11
	Figure 3 – Cyclic or periodic scanning of a program	12
	Figure 4 – Function block BCD_DIFF	20
	Figure 5 – Function block SBCD_DIFFF	20
	Figure 6 – ST example of time data type usage.....	22
	Figure 7 – Example of declaration and use of “anonymous array types”	23
	Figure 8 – Examples of VAR_IN_OUT usage.....	28
	Figure 9 – Hiding of function block instances	32
	Figure 10 – Graphical use of a function block name	35
	Figure 11 – Access to an indirectly referenced function block instance	35

Figure 12 – Invocation of an indirectly referenced function block instance	37
Figure 13 – Timing of edge triggered functionality	40
Figure 14 – Execution control example.....	42
Figure 15 – Timing of Boolean actions	47
Figure 16 – Example of a programmed non-Boolean action	47
Figure 17 – Use of the pulse (P) qualifier	48
Figure 18 – An SFC function block	49
Figure 19 – Example of incorrect and allowed programming constructs	58
Figure 20 – Essential phases of POU creation	69
Figure 21 – Essential phases of application creation	70
Figure 22 – Separate compilation of functions and function blocks	70
Figure 23 – Compiling a program accessing external or directly represented variables	71
Figure 24 – Compiling a function that invokes another function.....	71
Figure 25 – Compiling a program containing local instances of function blocks	72
Figure 26 – Separate compilation example.....	73
Figure 27 – The configuration process	74
Figure 28 – Reduction steps	77
Figure 29 – Reduction of SFCs	78
 Table 1 – IEC 61131-3 elements supporting encapsulation and hiding.....	 15
Table 2 – Examples of textual invocations of functions and function blocks	29
Table 3 – Differences between multi-user and real-time systems.....	51
Table 4 – Recommended run-time error-handling mechanisms.....	64
Table A.1 – Changes in usage to achieve program compliance	89

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: 2003, second edition.

Its contents answer a number of frequently asked questions about the intended application and implementation of the normative provisions of IEC 61131-3, second edition and about its differences from IEC 61131-3:1993, first edition.

NOTES

AUSTRALIAN STANDARD

Programmable controllers**Part 8:**
Guidelines for the application and implementation of programming languages**1 General****1.1 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. It also provides guidelines for the implementation of these languages in programmable controller systems and their programming support environments (PSEs).

IEC 61131-4 should be consulted for other aspects of the application of programmable controller systems.

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

1.2 Normative references

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

IEC 61131-1:1992, *Programmable controllers – Part 1: General information*

IEC 61131-2:2003, *Programmable controllers – Part 2: Equipment requirements and tests*

IEC 61131-3:2003, *Programmable controllers – Part 3: Programming languages*

IEC 61131-5:2000, *Programmable controllers – Part 5: Communications*

1.3 Abbreviated terms

FB	Function Block
FBD	Function Block Diagram
LD	Ladder Diagram
IL	Instruction List
POU	Program Organization Unit
PSE	Programming Support Environment
SFC	Sequential Function Chart
ST	Structured Text

1.4 Overview

The intended audience for this technical report consists of

- *users of programmable controller systems* as defined in IEC 61131-3, who must program, configure, install and maintain programmable controllers as part of industrial-process measurement and control systems; and
- *implementors of programming languages*, as defined in IEC 61131-3, for programmable controller systems. This may 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 *implementors* 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. Subclause 1.4 of IEC 61131-3 should be consulted by those who wish a “top-down” overview of the contents of that standard.

Clause 2 of this technical report provides a general introduction to IEC 61131-3, while Clause 3 provides complementary information about the application of some of the programming language elements specified in IEC 61131-3. Clause 4 provides information about the intended implementation of some of these programming language elements, while Clause 5 provides general information about requirements for hardware and software for program development and maintenance. Hence, it is expected that users of programmable controllers will find Clauses 2 and 3 of this technical report most useful, while programming language implementors will find Clauses 4 and 5 more useful, referring to the background material in Clauses 2 and 3 as necessary.

2 Introduction to IEC 61131-3

2.1 General considerations

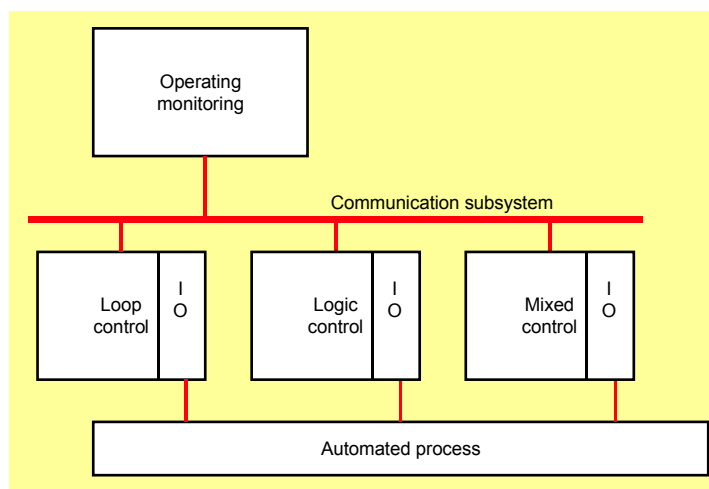
In the past, the limited capabilities of expensive hardware components imposed severe constraints on the design process for industrial-process control, measurement and automation systems. Software design and implementation were tightly tailored to the selected hardware. This required specialists who 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 small programmable controller may cost many times less than the cost of programming it.

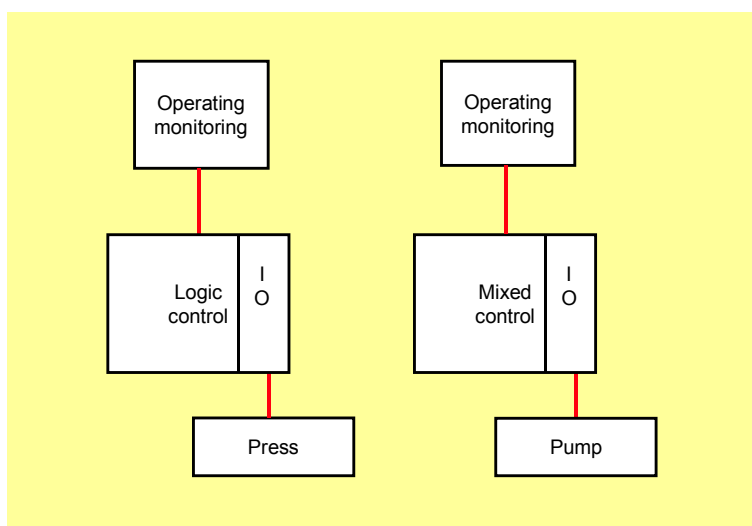
Driven by rapidly decreasing hardware cost, a trend has become established of replacing large, centrally installed process computers or other comparatively large, isolated controllers by systems with spatially and functionally distributed parts.

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

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



IEC 2060/03

Figure 1 – A distributed application

IEC 2061/03

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. Periodic execution of a program is triggered by a clock mechanism at equidistant points in time.

These principles are well known and applied in the operation of digital signal processing systems to simulate the operation of continuously operating analog or electromechanical systems. Process values are read into the device and written out to the process as discrete samples at random or equidistant points in time, depending on the control task that has to be fulfilled.

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

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

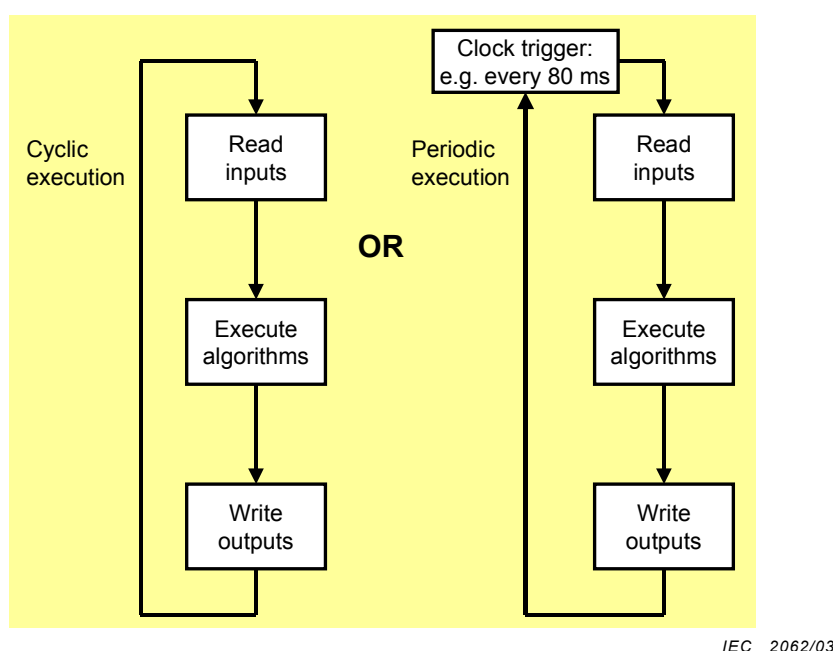


Figure 3 – Cyclic or periodic scanning of a program

2.2 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. In the past, this has resulted in significant differences in the elements and methods of programming the software as well. This 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 2.1, 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.

2.3 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 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 necessary functionalities for designing the communication between application programs are 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) Two graphical languages and two 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.
 - 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
 - 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 project.

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.

2.4 New features in the second edition of IEC 61131-3

Since 1993, the publication date of the first edition of IEC 61131-3, its environment has changed greatly. During the first phase, a large amount of experience with the practical application of the standard was gained. A number of inconsistencies, contradictions and unresolved questions as well as features, which were unnecessarily difficult to implement, were discovered. The industrial end-users, often represented by software companies, supplied many proposals for changes and amendments.

To maintain the value of investment by the former IEC 61131 users and by today's users of IEC 61131-3 control software as far as possible for the future, the IEC has decided to use the review of existing standards, which is due every five years, for a two-step revision.

Step 1: Elimination of inconsistencies within IEC 61131-3 (corrigendum)

Step 2: Amelioration of specific items in need of improvement within IEC 61131-3 and the integration of features regarded as particularly relevant in practice (amendment).

For every individual item to be altered, changes were kept upward-compatible, i.e. a user program compliant with the first edition is also compliant with the new edition, with the exceptions noted in Clause A.4.

A summary of the corrections and amendments is given in Annex A of this technical report.

2.5 Software engineering considerations

2.5.1 Application of software engineering principles

2.5.1.1 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 B of this technical report for descriptions of the software engineering quality measures referenced in this subclause, 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.

Associated with encapsulation is the notion of *hiding* of procedures and data, in which the only knowledge that the user has of a software entity is its external interface and specified functionality. Details of internal data structure and procedure implementation are intentionally hidden. Hiding contributes to software maintainability, integrity, usability, portability and reusability.

The elements of IEC 61131-3 that support encapsulation and hiding, and their main subclauses in IEC 61131-3, are listed in Table 1.

Table 1 – IEC 61131-3 elements supporting encapsulation and hiding

Element (subclause)	Encapsulation		Hiding	
	Data	Procedures	Data	Procedures
Structure (2.3.3)	Yes	No	No	No
Function (2.5.1)	No	Yes	Yes	Yes
Function block (2.5.2)	Yes	Yes	Yes	Yes
Program (2.5.3)	Yes	Yes	Yes	Yes
Action (2.6.4)	No	Yes	No	No
Access path (2.7.1)	Yes	No	Yes	No

2.5.1.2 Explicit representation of state

The SFC elements defined in 2.6 of 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 must 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.

2.5.1.3 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 2.1 and 2.3 of this technical report. This characteristic contributes to software maintainability, usability and adaptability.

2.5.1.4 Mapping of design to implementation

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.

- 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 2.7 of IEC 61131-3 can be used in this step.
- A first decomposition of the system design is made by allocating the required functionality into one or more elements, typically *programs* (2.5.3 of IEC 61131-3). 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.
- 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* (2.6 of IEC 61131-3) expressing the sequence of operations to be performed and the conditions for repeating the cycle of

operations. Each *action* (2.6.4 of IEC 61131-3) of the SFC is then further decomposed, typically into interconnected *function blocks* (2.5.2 of IEC 61131-3), i.e., an FBD (4.3 of IEC 61131-3). 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 (see 1.4.3 of IEC 61131-3), or can be expressed algorithmically in one of the textual or graphic languages in IEC 61131-3, i.e. IL (3.2 of IEC 61131-3), ST (3.3 of IEC 61131-3), LD (4.2 of IEC 61131-3), or FBD (4.3 of 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 2.7 of IEC 61131-3.

2.5.1.5 Structured programming

The contribution of structured programming techniques to software reliability, maintainability and adaptability is well known. The ST language defined in 3.3 of IEC 61131-3 provides a full set of constructs to support this style of programming, while retaining full compatibility with the other graphical and textual languages and elements in IEC 61131-3. Whereas the first edition of IEC 61131-3 still contained deficiencies concerning this compatibility (for example, EN/ENO usage), the second edition introduces the necessary extensions and adaptations in the syntax and semantics of function calls and function block invocations, to ensure mutual language interchange.

2.5.1.6 Software reuse

The programming model described in 1.4.3 of IEC 61131-3 and shown in Figure 3 of IEC 61131-3 strongly supports the reuse of software elements. These may be developed by the user in the “bottom-up” implementation process described in 2.5.2.4 above or may be supplied as “libraries” by software vendors. This method of system construction may be unfamiliar to users who have previously developed automation system applications as a single, large ladder diagram. Hence, some training may 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 1.4.3 of IEC 61131-3 and Clause B.0 of IEC 61131-3, the software elements that may be placed in libraries for reuse include, in order of increasing complexity and functionality:

- data types;
- functions;
- function blocks;
- programs;
- configurations.

2.5.2 Portability

2.5.2.1 Inter-language portability

As noted in Annex B of this technical report, *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 support environment (PSE) to another.

As noted in 2.5.2.1 of this technical report, 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 item 5) of 2.3 of this technical report, 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.

The problems in the first edition of IEC 61131-3 with the different support of EN/ENO in graphical languages and textual languages are solved in the second edition, as already mentioned in 2.5.1.5.

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.

2.5.2.2 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 in its Annex B for the common elements and for the two textual languages ST and IL. The minimally required encoding for the export and import of library elements written in this textual format is also defined in item j) of 1.5.1 of IEC 61131-3. This is a new requirement in the second edition of IEC 61131-3, which will improve inter-system portability.

Consequently, compliant POUs, as defined in 1.5.2 of IEC 61131-3, may 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 may also not be portable if the set of implementation-dependent parameters of the two systems differ in important values. A typical example is the support of a different number of characters used in distinguishing two identifiers.

3 Application guidelines

3.1 Use of data types

Subclause 2.3.2 of IEC 61131-3 offers many elementary data types. The user can also define new data types, as described in 2.3.3 of IEC 61131-3, 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 as described in 1.4.3 of IEC 61131-3. 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.

3.1.1 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 standard. 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
```

3.1.2 Use of enumerated and subrange types

IEC 61131-3 provides mechanisms for the definition of *enumerated* and *subrange* types. These types can make a program more readable and thus act as documentation aids. In addition, these types can contribute to program reliability by helping to avoid the use of unintended values of variables as well as by explicitly expressing the intended semantics of the values of enumerated variables.

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, Green, or Blue. IEC 61131-3 does not define numerical values to which these enumerated values may correspond. 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 colour values is used. All other values are flagged as errors.

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

IEC 61131-3, 2nd edition, also provides for the *typing* of enumerated values to distinguish, for instance, between the values `brickColor#Red` and `paintColor#Red`. The second edition also allows the use of an enumerated value as a selector in a CASE statement.

The syntax given in B.1.4 and B.1.3.3 of IEC 61131-3 allows the creation of “anonymous subrange data types” and “anonymous enumerated data types” in the declaration of variables and of elements of structured data types. An “anonymous subrange data type” is characterized by its base type and subrange. Similarly, an “anonymous enumerated data type” is characterized by the number, order, and identifiers of its enumerated values.

EXAMPLE 1

Given the type and variable declarations below, the variable `Y` is considered to be of the same anonymous 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 not anonymous.

```

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

```

EXAMPLE 2

See 3.1.9 below for an example of the definition and use of an “anonymous subrange type”.

3.1.3 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 must 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 2.5.1.5.2 of 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.

Users should be aware of the potential errors that may be caused in the encoding of BCD data.

- Since no standard BCD encoding is defined for the “minus” sign, the use of a negative number as an input to a `**_TO_BCD` conversion may cause a conversion error.
- The range of an integer variable is larger than the range of a BCD-encoded bit string variable with the same number of bits. For instance, the range of a variable of type `SINT`

is (−128..127), while the range of numbers that can be encoded in a bit string of type BYTE is only (0..99).

The example shown in Figure 4 illustrates precautions that may be necessary to avoid these errors. In this example, the function block BCD_DIFF takes the difference between two two-digit BCD-encoded inputs THUMB1 and THUMB2 and outputs the result as a two-digit BCD encoded bit string plus a Boolean sign, which could be used for example to drive a “two-digit plus sign” BCD display.

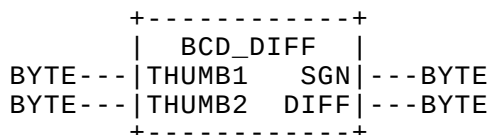


Figure 4a – External interface

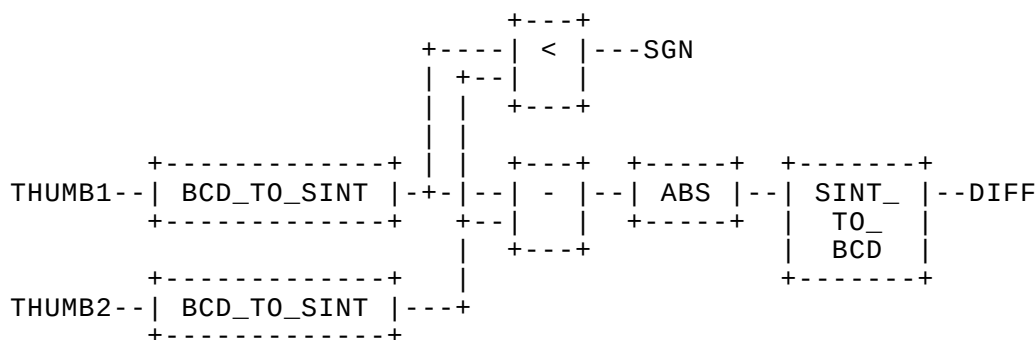


Figure 4b – Body

Figure 4 – Function block BCD_DIFF

A different solution to this problem would be to define a new structured data type containing both the sign and BCD-encoded data, such as type SBCD_BYTE shown in Figure 5a. Functions such as SBCD_DIFF, as shown in Figure 5b and c, could then be defined which produce values of this new type as the result of their execution.

```

TYPE SBCD_BYTE: STRUCT
  DATA: BYTE; SGN: BOOL;
END_STRUCT; END_TYPE
  
```

Figure 5a – Definition of structured data type SBCD_BYTE

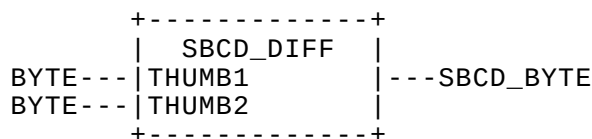


Figure 5b – External interface

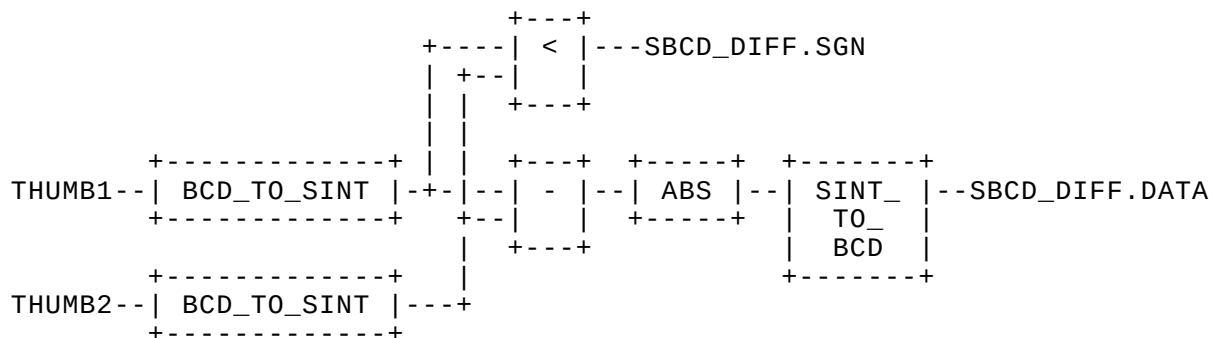


Figure 5c – Body

Figure 5 – Function block SBCD_DIFF

3.1.4 Use of REAL data types

The 32-bit REAL data type described in 2.3.1 of IEC 61131-3 can be used for holding the majority of decimal values such as control loop set-points, and process values. The REAL data type supports a wide range of values within the range $\pm 10^{\pm 38}$, with a precision of 1 part in 2^{23} , i.e., 1 part in 8388608.

Where a higher value range or higher precision is required, the 64-bit (long) format LREAL can be used with a range of $\pm 10^{\pm 308}$ and precision of 1 part in 2^{52} .

NOTE 1 In some algorithms, rounding errors may be magnified by the calculations performed. Data types with higher precision than initially apparent may be required in order to avoid such errors.

NOTE 2 See 4.2.1 of this technical report for additional considerations in the implementation of REAL types.

3.1.5 Use of character string data types

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, second edition, also provides the WSTRING data type for strings of 16-bit ("Unicode") characters.

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.

Length of character strings

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

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

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

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

IEC 61131-3 does not specify what may 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 may cause an error or may truncate the assigned value to the length L_{md} in this case.

NOTE Truncation rather than error is the recommended option for implementers.

3.1.6 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 1 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.

The time data types `TIME`, `TIME_OF_DAY`, `DATE` and `DATE_AND_TIME`, can be used in expressions with the numeric operators `ADD (+)`, `SUB (-)`, `MUL (*)`, `DIV (/)`, and also with the concatenation function `CONCAT`. An example of such usage is given in Figure 6.

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

IEC 2065/03

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.

NOTE 2 IEC 61131-3, second edition, modified the naming conventions for several functions of time data types in Table 30 of IEC 61131-1, in order to make them consistent with the definition of overloaded functions given in 2.5.1.4 of IEC 61131-3. Previous function names not consistent with this definition are deprecated, i.e. they will not be included in IEC 61131-3, third edition.

3.1.7 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, s

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.

According to the IEC 61131-3 definition of *aggregate*, each definition of an array in a variable or structure declaration creates a *data type*. Such “anonymous array types” are characterized by their element types and subscript range(s). For instance, the aggregate accelerations as declared in Figure 7 would be considered to be of the same type as the aggregate speeds, while the aggregate positions would not be considered to be of the same type. Hence the values of all elements of the aggregate accelerations can be assigned the values of all elements of the aggregate speeds in a single assignment, while the assignment of values from speeds to the elements of positions has to be done by component-wise assignment as shown in Figure 7.

VAR_IN <code>lineState: INT; END_VAR;</code>
VAR_OUT <code>lineSpeed: REAL; END_VAR;</code>
VAR <code>speeds: ARRAY[0..3] OF REAL := (0.0, 1.0, 3.0, 9.0);</code> <code>accelerations: ARRAY[0..3] OF REAL;</code> <code>positions: ARRAY[1..4] OF REAL;</code> <code>I: INT;</code> END_VAR;
<code>lineSpeed := speeds[lineState];</code> <code>accelerations := speeds; (* accelerations[0] := speeds[0], etc. *)</code> FOR <code>I := 0 TO 3</code> <code>positions[I+1] := speeds[I]; (* element-wise assignment *)</code> END_FOR

IEC 2066/03

Figure 7 – Example of declaration and use of “anonymous array types”

3.1.8 Use of bit-string functions

The selection and comparison functions in Tables 27 and 28 of IEC 61131-3 are defined to operate on data of the bit-string data types `BOOL`, `BYTE`, `WORD`, `DWORD` and `LWORD` as well as on numeric data types such as `INT`. Comparisons of bit-string data are made bitwise from the most significant to the least significant bit, and shorter bit strings are considered to be filled on the left with zeros when compared to longer bit strings; that is, comparison of bit-string variables has the same result as comparison of unsigned integer variables.

EXAMPLES – The following expressions all evaluate to TRUE:

```
GT(TRUE, FALSE)
GT(TRUE, DWORD#0)
GT(WORD#1FF, BYTE#FF)
```

The operation of MIN and MAX functions on bit-string types can be considered as an application of the comparison functions. For mathematical details, see 4.2.2.

EXAMPLES

```
MAX(BYTE#5, BYTE#F0, BYTE#EF) = BYTE#F0 because
BYTE#F0 ≥ BYTE#5 and BYTE#F0 ≥ BYTE#EF;
similarly,
MIN(BYTE#5, BYTE#F0, BYTE#EF) = BYTE#5 because
BYTE#5 ≤ BYTE#F0 and BYTE#5 ≤ BYTE#EF .
```

3.1.9 Strongly typed assignment

Although not explicitly stated in IEC 61131-3, it was the intention of the standard 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. This is implied, for instance, by the discussion of the assignment statement in 3.3.2.1 of IEC 61131-3. In this interpretation, a result is considered to be of the same type as a variable when

- its type name is the same as the type name of the variable to which it is to be assigned; or
- both the result and the variable to which it is to be assigned are “anonymous array types” as discussed in 3.1.7, whose element type and subscript range(s) exactly match each other; or
- both the result and the variable to which it is to be assigned are “anonymous enumerated types” as discussed in 3.1.2, whose enumerated value sets exactly match each other in number, order and identifiers of the enumeration elements.

In contrast, the assignment of a result to the value of a variable is allowed as long as the type of the result is the same as the “parent” type of the subrange type of the variable; however, a run-time check must be made to determine that the value is within the specified subrange limits of the variable.

These conditions apply not only to the assignment of values to variables in assignment statements but also to the use of the ST (store) operator in the IL language defined in 3.2.2 of IEC 61131-3, data flow connections in the LD and FBD languages defined in Clause 4 of IEC 61131-3, and data passing as discussed in 3.2.

EXAMPLE 1

Consider the variables defined in the statements

```
VAR X : ARRAY[1..16] OF ANALOG_CHANNEL_CONFIGURATION;
      (* A variable of an anonymous array type *)
  Y : ANALOG_16_INPUT_CONFIGURATION;
  Z : SINT(5..95) ; (* A variable of an anonymous subrange type *)
END_VAR
```

Where the applicable data types are defined as in Table 12 of IEC 61131-3:

```
TYPE
  ANALOG_CHANNEL_CONFIGURATION :
    STRUCT
      RANGE : ANALOG_SIGNAL_RANGE ;
      MIN_SCALE : ANALOG_DATA ;
      MAX_SCALE : ANALOG_DATA ;
    END_STRUCT ;

  ANALOG_16_INPUT_CONFIGURATION :
    STRUCT
      SIGNAL_TYPE : ANALOG_SIGNAL_TYPE ;
      FILTER_PARAMETER : SINT (0..99) ;
      (* An anonymous subrange data type *)
      CHANNEL : ARRAY [1..16] OF ANALOG_CHANNEL_CONFIGURATION;
      (* An anonymous array type *)
    END_STRUCT ;
END_TYPE
```

Then an assignment statement of the form

```
X := Y.CHANNEL;
```

is valid and would cause an assignment of the values Y.CHANNEL[1] through Y.CHANNEL[16] to the variable elements X[1] through X[16], respectively.

Similarly, an assignment statement of the form

```
Z := Y.FILTER_PARAMETER;
```

is valid and could cause an assignment of the value of the `FILTER_PARAMETER` element of the structured variable `Y` to the variable `Z`. However, a run-time error would occur if the value of the `FILTER_PARAMETER` element were less than 5 or greater than 95.

EXAMPLE 2

See 3.1.2 for an example and use of an “anonymous enumerated type”.

3.2 Data passing

There are several methods for passing data into and out of POU's including functions, function blocks, and programs.

NOTE 1 The term “parameter passing” is not used in this subclause, because the term *parameter* is not used in IEC 61131-3 except in the narrow sense of “A *variable* that is given a constant value for a specified *application* and that may denote the application” as defined in ISO/IEC 2382-02 and cited in IEC 61499-1. Therefore, the term *parameter* is only used in IEC 61131-3 in the context of “implementation-dependent parameters” and “configuration parameters”. The term *variable* is used instead in all other contexts to mean “A *software entity* that may take different values, one at a time” as defined in IEC 61499-1.

The most restricted methods to access data are defined for functions. Inside functions, only reading of input variables is allowed and a function has to return a value which may be of an aggregate type (for example, array, string or structure). Functions do not have any access to globally defined variables, nor may they access directly represented variables. As another design goal the standard requires functions to have no static variables. This means a function cannot save any of its computed or input values from one invocation to the next. Each invocation will receive a set of freshly initialized local variables, possibly with default values if not stated differently by the function definition. These restrictions ensure that operation of the function is independent of any previous execution, depending only on the set of argument values from its current invocation.

NOTE 2 IEC 61131-3, second edition, now allows functions to access in-out variables (see 3.2.2), and output variables in addition to input and internal variables and the single variable denoting the function output.

NOTE 3 IEC 61131-3, second edition, has also introduced additional flexibility in the means for textual invocations of functions; see 2.5.1.1 of IEC 61131-3 and 3.2.3 .

Within a function block body, the reading of input variables and reading or writing of output variables is permitted. Function blocks also may have in-out variables, which can be read or written. All global 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 in the function block definition by use of the `VAR_EXTERNAL` keyword.

NOTE 4 IEC 61131-3, second edition, now allows *function blocks* in addition to *programs*, *configurations* and *resources* to access *directly represented variables*.

The values of directly represented variables can be passed to a function block as input variables from the program outside that function block. The same holds for output values from a function block, which can be copied to directly represented variables in a program. A function block may declare static storage that it can use to save any data from one invocation to the next. Function block execution can therefore have many effects upon data, including external data, which may be essential to the operation of the function block.

Programs have access to data as function blocks have and can contain declarations of global variables and access paths; see 2.5.3 of IEC 61131-3. These have to be declared within the program and thereafter can be freely read or written.

3.2.1 Global and external variables

Global variables can be defined and initialized within a configuration, resource, or program by use of the `VAR_GLOBAL` keyword. Each program or function block that needs access to one or

more of these global variables has to redeclare the variables by use of the VAR_EXTERNAL keyword.

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

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

As global variable declarations include directly represented variables, aliases may be declared. These aliases could be referenced by external variable declarations in function blocks and programs.

In the following example a global variable TEMP is declared to be of type INT located at input word %IW22. Within a function block the variable name TEMP 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 TEMP AT %IW22: INT; END_VAR
(* Access from inside a FUNCTION_BLOCK *)
  VAR_EXTERNAL TEMP: INT; END_VAR
  ...
  ... := TEMP;
```

3.2.2 In-out (VAR_IN_OUT) variables

In-out variables are a special kind of variable used with POU's, i.e., *functions*, *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.

NOTE IEC 61131-3, second edition, now permits the use of in-out variables with functions.

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 $x1 \times x2$ 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 $x1 \times x2 + x3 \times x4$ 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.

NOTE This is a new feature introduced in IEC 61131-3, second edition.

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.

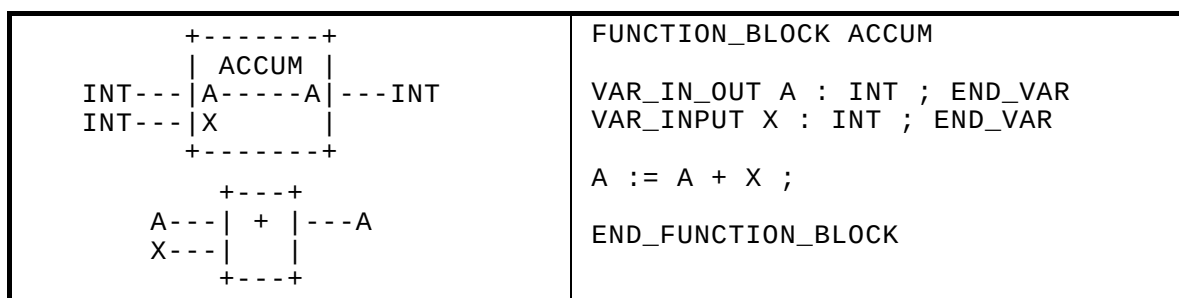


Figure 8a – Declaration of a FB type using VAR_IN_OUT

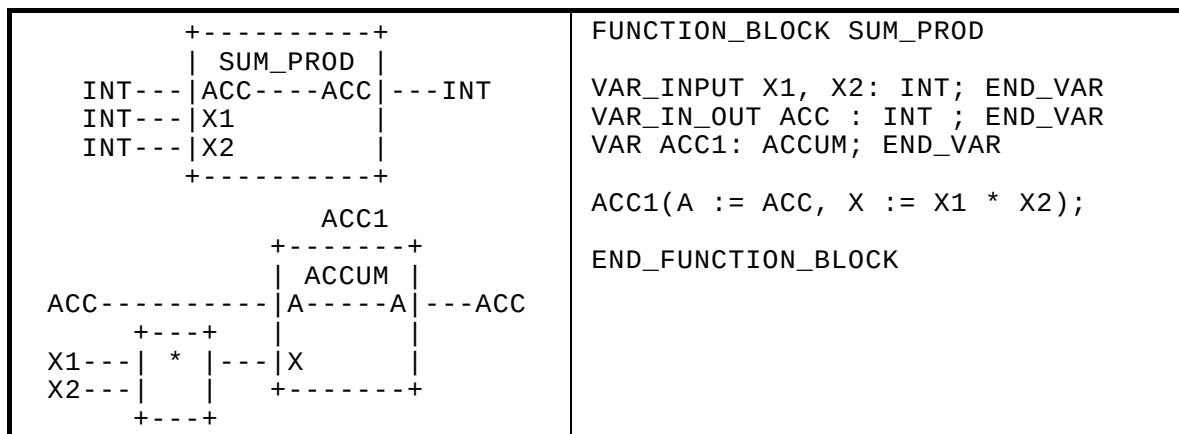


Figure 8b – Usage of an instance of a FB type using VAR_IN_OUT

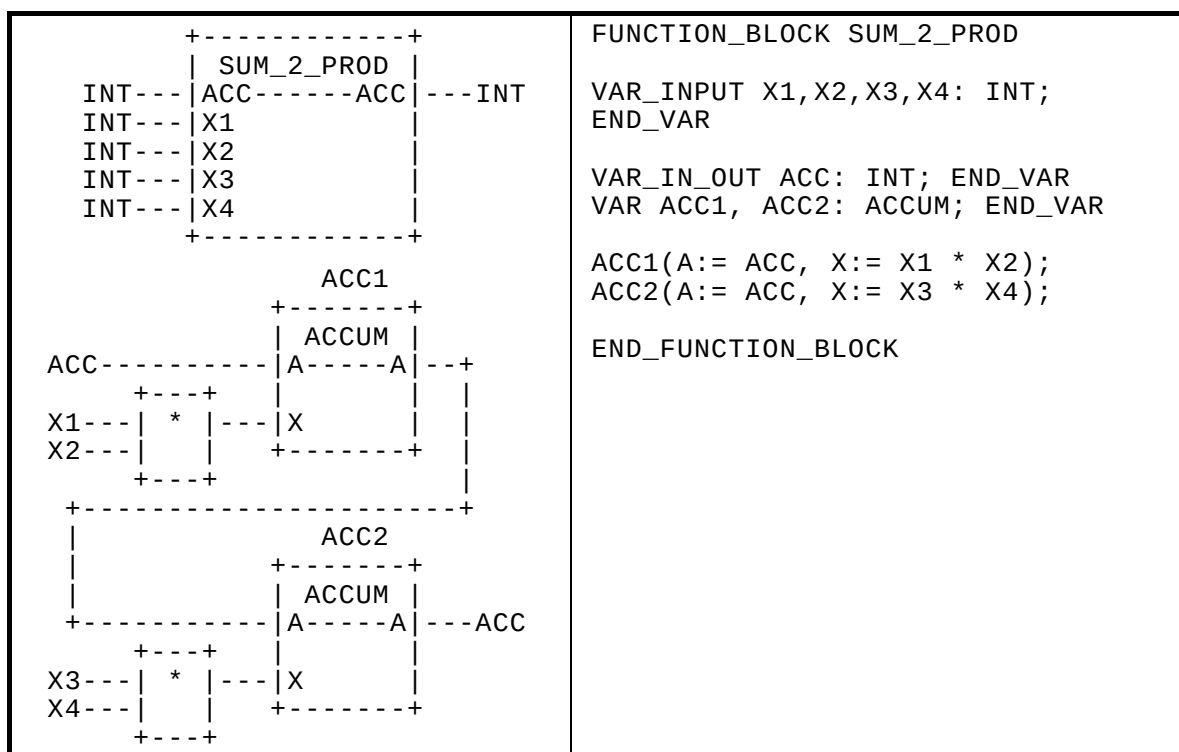


Figure 8c – Usage of two instances of a FB type using VAR_IN_OUT

<pre> +-----+ ASUM +-----+ INT[1..10]-- A-----A --INT +-----+ </pre>	<pre> 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 </pre>
--	--

Figure 8d – Declaration of another FB type using VAR_IN_OUT

<pre> +-----+ ACC1 +-----+ X1--- * X2--- X3--- +-----+ X----- X +-----+ </pre>	<pre> +-----+ ACCUM +-----+ A-----A---ACC +-----+ </pre>	ILLEGAL USAGE: In-out A is not a variable or function block name
--	--	--

Figure 8e – Example of illegal usage

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

Figure 8f – Example of illegal usage

Figure 8 – Examples of VAR_IN_OUT usage

IEC 2067/03

3.2.3 Formal and non-formal invocations and argument lists

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

IEC 61131-3, second edition, introduces extensive improvements in the specifications of functions and function blocks as described below.

NOTE See A.3 of this technical report for a description of the rationale for these changes with respect to IEC 61131-3, first edition.

- According to 2.5.1 of 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, i.e., invocation of a function with the same arguments (input variable values and in-out variable values) will always yield the same output values (output variables, in-out variables and function result). 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.
- According to 3.3.2.2 of IEC 61131-3, which refers to 2.5.1.1 and Table 19a, function block invocations now 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 – like in the above-mentioned invocation variant for extensible standard functions in IEC 61131-3, first edition.

- c) Subclause 3.6.2.2 of IEC 61131-3 introduces the terms “formal argument list” and “non-formal argument list” for the two invocation variants with formal argument assignment and without formal argument assignment. In 2.5.1.1 of IEC 61131-3 (Table 19a), these two variants are introduced as features for the invocation of functions, and through the reference in 3.3.2.2 of IEC 61131-3 for the invocation of function blocks, too. The usage of feature 1 (formal invocation) or feature 2 (non-formal invocation) is left to the choice of the programmer, who may, however, have to satisfy the requirements of the actual application (for example, usage of ENO necessary).
- d) Many standard functions did not explicitly define input variable names inside their declarations in IEC 61131-3, first edition. To allow the usage of formal argument lists for calls of these functions, corresponding rules for formal input variable naming are now introduced for all declarations of standard functions in 2.5.1.5 of IEC 61131-3.
- e) The usage of the operator “=>” introduced in 2.7.1 of IEC 61131-3 for program configurations is extended to formal argument lists. This allows the assignment of output variables inside the formal argument list of a function call or a function block invocation. For function calls, this mechanism of output variable assignment is the only possible one, since there is no construct to access output variables of functions 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, as introduced in 2.5.1.1 of IEC 61131-3 (Table 19) for graphical languages, in textual invocations.
- f) 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 or function block: any variable not assigned a value in the list shall have the default value, if any, assigned in the function or function block specification, or the default value for the associated data type. The ordering of argument assignments in the list is not significant.
- NOTE 1 All rules in the foregoing paragraph – except for the introduction of the operator “=>” – are unmodified compared to the provisions of IEC 61131-3, first edition.
- g) In contrast to the rules for formal argument lists, for non-formal argument lists of function 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 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 (see 3.2.3 of IEC 61131-3).
- NOTE 2 All rules in the foregoing paragraph are unmodified compared to the provisions of IEC 61131-3, first edition.

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)	<pre>A := LIMIT(EN:= COND, IN:= B, MX:= 5, ENO => TEMP);</pre>
b)	Non-formal function call in ST	<pre>A := LIMIT(1, B, 5);</pre>

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

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(%IX5, T#300ms, OUT, ELAPSED);
e)	Formal function block invocation in ST (Note 3)	CMD_TMR(IN := %IX5, 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(%IX10, FALSE, A, OUT, B)
j)	Formal function block invocation in IL (Note 7)	CAL C10(CU := %IX10, 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 <pre> VAR CMD_TMR, MYTON: TON; C10: CTU; A, B: INT; X, Y: REAL; ELAPSED: TIME; COND, TEMP, OUT, ERR, START: BOOL; END_VAR </pre> is assumed in the above examples.		

3.3 Use of function blocks

3.3.1 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 a private 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). The function block type is similar to a class, which defines the data structure and computational method within the body of the function block. Individual objects are represented by the private data areas of the individual function block instances. This data can only be modified from outside the function block body in a controlled manner. This enforces the software engineering principles of encapsulation and information hiding, a key element of OOP.

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.

Instantiation of function blocks is an extension to the capabilities of today’s programmable controllers. In many current implementations, there is only a fixed number of instances of each function block type available, and additional instances cannot be created. User-defined function blocks that can be instantiated in an unlimited way are also an extension to most existing systems.

3.3.2 Scope of data within function blocks

The principle of data encapsulation and hiding of variable names discussed in the preceding subclause may 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. This principle is expressed in 2.5.2 of IEC 61131-3 in the statement:

“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 as defined in 2.7.1.”

This requirement might be considered to contradict the assertion in 2.5.2 of IEC 61131-3 that “any function block which has already been declared can be used in the declaration of another function block or program as shown in Figure 3”. However, it is clear by reference to

Figure 3 of IEC 61131-3 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
```

Figure 9a – 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
```

Figure 9b – Declaration of containing program type

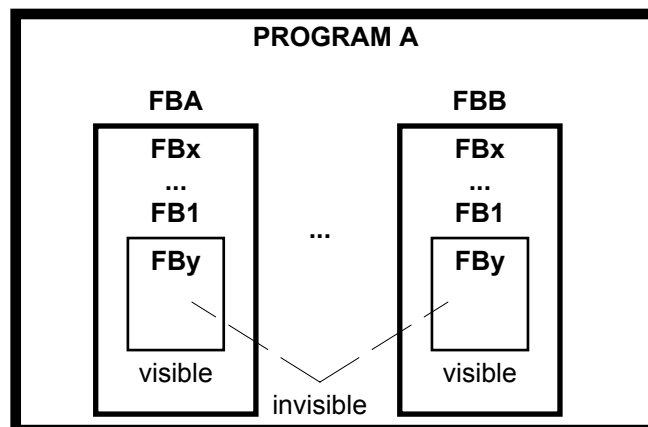


Figure 9c – Visibility of function block instances

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

IEC 2068/03

Figure 9 – Hiding of function block instances

3.3.3 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 (see IEC 61131-3, Table 32).

NOTE 1 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 2.7 of this technical report for more details.