

Programmable controllers —

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

ICS 25.040.40; 35.240.50

NO COPYING WITHOUT BSI PERMISSION EXCEPT AS PERMITTED BY COPYRIGHT LAW

BSI
Documents



Document Sharing Hub

National foreword

This Published Document (PD) reproduces verbatim IEC TR 61131-8:2000 (Technical Report).

This PD is not to be treated as a British Standard.

The UK participation in its preparation was entrusted by Technical Committee GEL/65, Measurement and Control, to Subcommittee GEL/65/2, Elements of Systems, which has the responsibility to:

- aid enquirers to understand the text;
- present to the responsible international/European committee any enquiries on the interpretation, or proposals for change, and keep the UK interests informed;
- monitor related international and European developments and promulgate them in the UK.

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

From 1 January 1997, all IEC publications have the number 60000 added to the old number. For instance, IEC 27-1 has been renumbered as IEC 60027-1. For a period of time during the change over from one numbering system to the other, publications may contain identifiers from both systems.

Cross-references

The British Standards which implement international or European publications referred to in this document may be found in the BSI Standards Catalogue under the section entitled “International Standards Correspondence Index”, or by using the “Find” facility of the BSI Standards Electronic Catalogue.

A British Standard does not purport to include all the necessary provisions of a contract. Users of British Standards are responsible for their correct application.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

Summary of pages

This document comprises a front cover, an inside front cover, the IEC title page, pages 2 to 87 and a back cover.

The BSI copyright notice displayed in this document indicates when the document was last issued.

This British Standard, having been prepared under the direction of the Electrotechnical Sector Committee, was published under the authority of the Standards Committee and comes into effect on 15 June 2000

© BSI 06-2000

ISBN 0 580 34664 1

Amendments issued since publication

Amd. No.	Date	Comments

**RAPPORT
TECHNIQUE
TECHNICAL
REPORT**

**CEI
IEC**

TR 61131-8

Première édition
First edition
2000-01

Automates programmables –

**Partie 8:
Lignes directrices pour l'application et la mise
en oeuvre des langages de programmation**

Programmable controllers –

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



Commission Electrotechnique Internationale
International Electrotechnical Commission
Международная Электротехническая Комиссия

CONTENTS

	Page
FOREWORD.....	6
 Clause	
1 General	7
1.1 Scope.....	7
1.2 Reference documents	7
1.3 Overview	8
2 Introduction to IEC 61131-3	9
2.1 General considerations	9
2.2 Historical limitations.....	11
2.3 New features in IEC 61131-3	12
2.4 Software engineering considerations.....	13
2.4.1 Software quality measures	13
2.4.2 Application of software engineering principles	14
2.4.3 Portability	16
3 Application guidelines	17
3.1 Use of data types.....	17
3.1.1 Type vs. 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.....	21
3.1.7 Use of multi-element variables	22
3.2 Data import and export	23
3.2.1 Global and external variables	23
3.2.2 Input/output (VAR_IN_OUT) variables	24
3.3 Use of function blocks.....	26
3.3.1 Function block types and instances	26
3.3.2 Scope of data within function blocks	27
3.3.3 Function block access and invocation.....	28
3.4 Differences between function block instances and functions	29
3.5 Use of indirectly referenced function block instances.....	29
3.5.1 Establishing an indirect function block instance reference.....	29
3.5.2 Access to indirectly referenced function block instances	31
3.5.3 Invocation of indirectly referenced function block instances	32
3.5.4 Recursion of indirectly referenced function block instances.....	35
3.5.5 Execution control of indirectly referenced function block instances.....	35
3.5.6 Use of indirectly referenced function block instances in functions	35
3.6 Recursion within programmable controller programming languages	36
3.7 Single and multiple invocation.....	36

Clause	Page
3.8 Language specific features	37
3.8.1 Edge triggered functionality	37
3.8.2 Use of EN/ENO in functions and function blocks	39
3.8.3 Use of non-IEC 61131-3 languages	40
3.9 Use of SFC elements	40
3.9.1 Action control	41
3.9.2 Boolean actions	42
3.9.3 Non-SFC actions	47
3.9.4 SFC actions	48
3.9.5 SFC function blocks	50
3.9.6 "Indicator" variables	51
3.10 Scheduling, concurrency and synchronization mechanisms	52
3.10.1 Operating system issues	52
3.10.2 Task scheduling	53
3.10.3 Semaphores	54
3.10.4 Messaging	56
3.10.5 Time stamping	57
3.11 Communication facilities in ISO/IEC 9506-5 and IEC 61131-5	58
3.11.1 Communication channels	58
3.11.2 Reading and writing variables	58
3.11.3 Communication function blocks	59
3.12 Recommended programming practices	60
3.12.1 Global variables	60
3.12.2 Jumps in function block diagram (FBD) language	60
3.12.3 Multiple invocations of function block instances in FBD	60
3.12.4 Coupling of sequential function chart (SFC) networks	60
3.12.5 Dynamic modification of task priorities	61
3.12.6 Execution control of function block instances by tasks	61
3.12.7 Use of RTC (real time clock) function blocks	62
4 Implementation guidelines	62
4.1 Resource allocation	62
4.2 Implementation of data types	63
4.2.1 REAL and LREAL data types	63
4.2.2 Character strings	63
4.2.3 Time data types	63
4.2.4 Multi-element variables	64
4.3 Execution of functions and function blocks	64
4.3.1 Functions	64
4.3.2 Function blocks	65
4.4 Implementation of sequential function charts (SFCs)	66
4.5 Task scheduling	67
4.5.1 Classification of tasks	67
4.5.2 Task priorities	68

Clause	Page
4.6 Error handling.....	68
4.6.1 Error handling mechanisms	68
4.6.2 Run-time error handling procedures	69
4.7 System interface.....	72
4.8 Compliance	72
4.8.1 Compliance statement.....	72
4.8.2 Controller instruction sets.....	73
4.8.3 Compliance testing.....	73
4.9 Compatibility with IEC 60617-12, IEC 60617-13 and IEC 60848.....	73
5 Programming support environment (PSE) requirements.....	73
5.1 User interface.....	73
5.2 Programming of programs, functions and function blocks	74
5.3 Application design and configuration	75
5.4 Separate compilation	75
5.5 Interface/body separation	77
5.5.1 Invocation of a function from a programming unit.....	77
5.5.2 Declaration and invocation of a function block instance	77
5.6 Linking of configuration elements with programs	79
5.7 Library management	81
5.8 Analysis tools	82
5.8.1 Simulation and debugging	82
5.8.2 Performance estimation.....	82
5.8.3 Feedback loop analysis.....	83
5.8.4 SFC analysis.....	83
5.9 Documentation requirements	86
5.10 Security of data and programs	86
5.11 On-line facilities.....	86
Bibliography.....	87
Figure 1 – Distributed application	9
Figure 2 – Stand-alone applications	10
Figure 3 – Scanning a cyclic program.....	11
Figure 4 – Function block BCD_DIFF a) External interface b) Body.....	20
Figure 5 – Function block SBCD_DIFF a) Definition of structured data type SBCD_BYTE b) External interface c) Body	20
Figure 6 – Structured text (ST) example of time data type usage	22
Figure 7 – Example of array usage.....	22

Figure 8 – Examples of VAR_IN_OUT usage	25
Figure 9 – Hiding of function block instances a) Declaration of contained function block type b) Declaration of containing program type c) Visibility of function block instances.....	28
Figure 10 – Establishing an indirectly referenced function block instance a) Establishment as input variable b) Establishment as input/output variable c) Establishment as external variable	31
Figure 11 – Access to an indirectly referenced function block instance	31
Figure 12 – Invocation of an indirectly referenced function block instance a) Textual declaration and invocation b) Graphical invocation c) Textual passing of instance name d) Graphical passing of instance name	34
Figure 13 – Timing of edge triggered functionality a) Example network b) Worst-case timing	38
Figure 14 – Execution control example (see clause F.4 of IEC 61131-3) a) ST language without EN/ENO b) FBD language with EN/ENO.....	40
Figure 15 – Example of an action block	41
Figure 16 – Timing of Boolean actions a) N (non-stored) action b) P (pulse) action c) S/R (set/reset) action	43
Figure 16 – Timing of Boolean actions d) L (time limited) action e) D (time delayed) action .	44
Figure 16 – Timing of Boolean actions f) SD (stored and time delayed) g) DS (time delayed and stored)	45
Figure 16 – Timing of Boolean actions h) SL (stored and time limited) action	46
Figure 17 – Example of a programmed non-Boolean action	47
Figure 18 – Use of pulse (P) qualifier a) SFC fragment b) Incorrect body for action A15 (ST language) c) Correct body for action A15 (ST language) d) Correct body for action A15 (FBD language)	48
Figure 19 – Example of an SFC action enabled by an SFC	49
Figure 20 – An SFC-function block a) External interface, b) Body	51
Figure 21 – Essential phases of program organization unit (POU) creation	74
Figure 22 – Essential phases of application creation	75
Figure 23 – Separate compilation of functions and function blocks without external reference	76
Figure 24 – Compilation of a program which access to external or directly represented variables.....	76
Figure 25 – Compilation of a function whose body contains an invocation of another function a) Declaration of example function b) Compilation of example function	77
Figure 26 – Compilation of a program containing local instances of function blocks a) Declaration of example program b) Compilation of example program	78
Figure 27 – Separate compilation example a) Sketch of programming units to be compiled b) Separate compilation	80
Figure 28 –Configuration process a) Declaration of the configuration b) Production of the linked application	81
Figure 29 – Reduction steps.....	83
Figure 30 – Reduction of SFCs a) No reduction required as per termination rule 1 b) No reduction required as per termination rule 2 c) An irreducible (unsafe) SFC d) An irreducible (unreachable) SFC	85
Table 1 – IEC 61131-3 elements supporting encapsulation and hiding	14
Table 2 – Differences between multi-user and real-time systems	53
Table 3 – Recommended error handling mechanisms	69

INTERNATIONAL ELECTROTECHNICAL COMMISSION
PROGRAMMABLE CONTROLLERS –

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

FOREWORD

- 1) The IEC (International Electrotechnical Commission) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of the 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, the IEC publishes International Standards. 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. The 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 the 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 National Committees.
- 3) The documents produced have the form of recommendations for international use and are published in the form of standards, technical specifications, technical reports or guides and they are accepted by the National Committees in that sense.
- 4) In order to promote international unification, IEC National Committees undertake to apply IEC International Standards transparently to the maximum extent possible in their national and regional standards. Any divergence between the IEC Standard and the corresponding national or regional standard shall be clearly indicated in the latter.
- 5) The IEC provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with one of its standards.
- 6) Attention is drawn to the possibility that some of the elements of this technical report may be the subject of patent rights. The 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".

Technical reports do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful by the maintenance team.

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

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

Enquiry draft	Report on voting
65B/284/CDV	65B/336/RVC

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 document which is purely informative is not to be regarded as an International Standard.

This publication has been drafted in accordance with the ISO/IEC Directives, Part 3.

PROGRAMMABLE CONTROLLERS –

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

1 General

1.1 Scope

This 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. IEC 61508-1 and IEC 61508-7 should be consulted for such considerations.

1.2 Reference documents

IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems*

IEC 60617-12:1991, *Graphical symbols for diagrams – Part 12: Binary logic elements*

IEC 60617-13:1993, *Graphical symbols for diagrams – Part 13: Analogue elements*

IEC 60848:1988, *Preparation of functional charts for control systems*

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

IEC 61131-2:1992, *Programmable controllers – Part 2: Equipment requirements and tests*

IEC 61131-3:1993, *Programmable controllers – Part 3: Programming languages*

IEC 61131-4:1995, *Programmable controllers – Part 4: User guidelines*

IEC 61131-5:1998, *Programmable controllers – Part 5: Communications*

IEC 61508-1:1998, *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 1: General requirements*

IEC 61508-2, — *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems* ¹⁾

IEC 61508-3:1998, *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements*

IEC 61508-4:1998, *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 4: Definitions and abbreviations of terms*

IEC 61508-5:1998, *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 5: Examples of methods for the determination of safety integrity levels*

IEC 61508-6, — *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3* ¹⁾

IEC 61508-7, — *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 7: Overview of techniques and measures* ¹⁾

ISO/IEC 646:1991, *Information technology – ISO 7-bit coded character set for information interchange*

ISO/IEC 9506-5:1999, *Industrial automation systems – Manufacturing message specification – Part 5: Companion standard for programmable controllers*

ANSI/IEEE Std 754:1985, *Binary floating-point arithmetic*

NOTE The technical and normative provisions of IEC 60559 and ANSI/IEEE Std 754 are equivalent.

1.3 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;
- *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.

Clause 2 of this technical report provides a general introduction to IEC 61131-3, whilst 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, whilst 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, whilst programming language implementors will find clauses 4 and 5 more useful, referring to the background material in clauses 2 and 3 as necessary.

¹⁾ To be published.

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.

Historically, programmable controllers have been 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 of the new relatively small automation tasks can be solved profitably and flexibly by programmable controllers.

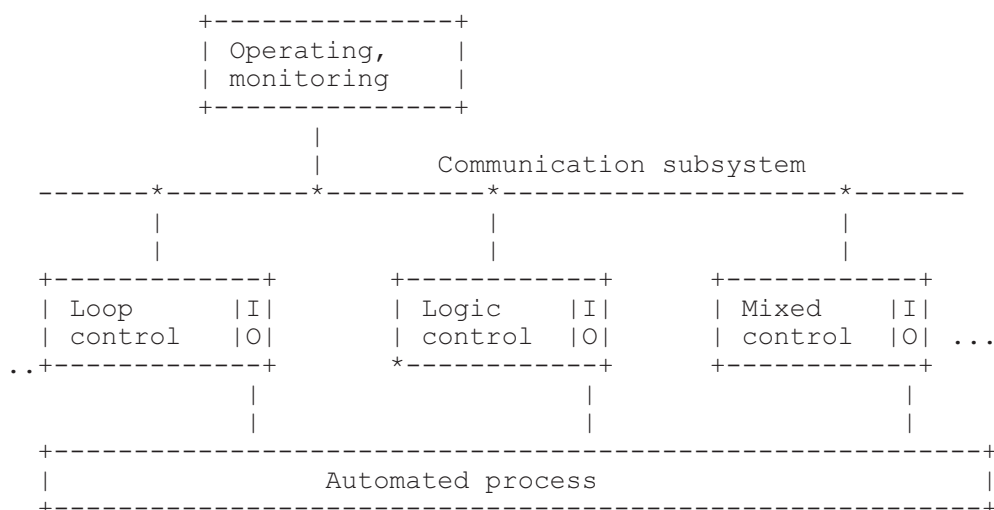


Figure 1 – Distributed application

IEC 1614/99



IEC 1615/99

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 principle of cyclic program execution illustrated in figure 3. This principle is 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 of time, depending on the control task that has to be fulfilled.

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

A particular advantage of graphical programming languages is that the representation they offer can be used not only for program input and documentation, but 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.

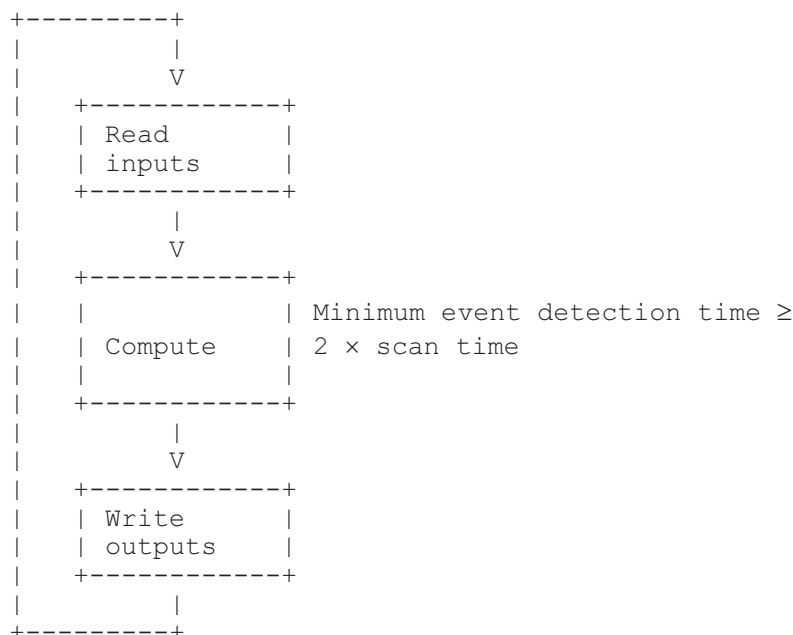


Figure 3 – Scanning a cyclic program

IEC 1616/99

2.2 Historical limitations

Automation system designers are often required to use programmable controllers from various manufacturers in different systems or even in the same system. However, the hardware of programmable controllers from different manufacturers may have very little in common. Historically, 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 often carry very specialized software for programming, testing and maintaining particular controller “families”.

Changing from one controller family to another often requires the designer to read large manuals for both the hardware and software of the new family. Often, the manual must 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 do it. For this reason, many people regard 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 is concentrated in one or a few specialists and cannot be transferred effectively to those responsible for system operation, maintenance and upgrade.

A major goal of IEC 61131-3 is to remove such barriers to the understanding and application of programmable controllers. Thus, IEC 61131-3 introduces numerous facilities to support the advantages of programmable controllers described in 2.1, even if controllers of different vendors are concerned. It is hoped that the resulting expansion of the application domains of programmable controllers will offset the costs that vendors will incur in making their systems compliant to the standard.

2.3 New features in IEC 61131-3

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

- 1) 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.
- 2) Strong data typing is not only supported, but inherently required, thus eliminating a major source of programming errors.
- 3) 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.
- 4) 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.
- 5) 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:
 - a) the graphical ladder diagram (LD) language models networks of simultaneously functioning electromechanical elements such as relay contacts and coils, timers, counters, etc.;
 - b) the graphical function block diagram (FBD) language models networks of simultaneously functioning electronic elements such as adders, multipliers, shift registers, gates, etc.;
 - c) 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;
 - d) the instruction list (IL) language models the low-level programming of control systems in assembly language;
 - e) 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:
 - f) sequential function charts (SFCs), which model time- and event-driven sequential control devices and algorithms.
- 6) Flexibility in selection of languages suited for programming application-specific functionalities will increase the reuse of software solutions to process control problems.
- 7) 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.

2.4 Software engineering considerations

2.4.1 Software quality measures

The quality of software-driven systems is measured by a number of attributes [1] ¹⁾ Among the most important quality measures of industrial process measurement, control and automation systems are;

- 1) *capability*: the extent to which the system performs its intended function. Typical measures of capability include:
 - a) *responsiveness*: the time required for the system to produce appropriate responses to specified combinations of external events;
 - b) *processing capacity*: the extent to which the system can meet scheduling deadlines under specified sets of conditions;
 - c) *storage capacity*: The extent to which the system can retain in memory all the required programs and data under specified sets of conditions.
- 2) *availability*: the proportion of the total process operating time the system is capable of performing its intended function. Availability is affected by factors such as:
 - a) *reliability*: the ability of the system to continue to perform all its intended functions over a specified period of time and range of conditions. An inverse measure of reliability is mean time between failures (MTBF);
 - b) *maintainability*: the ease with which the system can be restored to full capability after the occurrence of one or more faults from a specified set. An inverse measure of maintainability is mean time to repair (MTTR); availability is often defined by the expression $MTBF/(MTBF+MTTR)$;
 - c) *integrity*: the degree to which the system can continue to perform all its intended functions over a specified range of threats, including both unintended user actions, intentionally hostile actions and potentially hazardous application and system events.
- 3) *usability*: The ease with which a specified set of users can acquire and exercise the ability to interact with the system in order to perform its intended functions. Usability is affected by factors such as:
 - a) *entry requirements*: the level of formal and informal training required before the user can learn to interact with the system (e.g. educational level, training in the use of operating systems, windowing systems, etc.);
 - b) *learning requirements*: the training required for a user meeting a specified set of entry requirements to learn to interact with the system to perform a specified set of system functions;
 - c) *user productivity*: the number of system-related operations per unit time which can be performed by a user with a specified level of training and experience;
 - d) *congeniality*: the extent to which a user prefers to utilize the system software to perform the intended system functions, with respect to alternative activities that may or may not be system-related.
- 4) *adaptability*: the ease with which the functionality of the system may be changed in various ways such as:
 - a) *improvability*: the ease with which the existing system *capability*, *availability* and/or *usability* can be upgraded without basic changes in the system's functionality;
 - b) *extensibility*: the ease with which functionality can be added to the system;

¹⁾ Figures in square brackets refer to the bibliography.

- c) *portability*: the ease with which the system functionality can be moved from one system to another;
- d) *reusability*: the ease with which the functional capabilities of an existing software element can be used to add capability to a new or existing system.

2.4.2 Application of software engineering principles

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 contributors to software quality and their embodiment in IEC 61131-3 are discussed below.

2.4.2.1 Encapsulation and hiding

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

Elements (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

2.4.2.2 Explicit representation of state

The sequential function chart (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.

2.4.2.3 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. This characteristic contributes to software maintainability, usability and adaptability.

2.4.2.4 Mapping of design to implementation

IEC 61131-3 supports a style of system realization known as “top-down” (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.

- 1) The system’s desired functionality and external interface are specified. For instance, the basic functionality of a machining cell might be to accept a rough part from a material handling system, procure 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.
- 2) A first decomposition of the system design is made by allocating the required functionality into one or more elements, typically *programs* (see 2.5.3 of IEC 61131-3). The interfaces among the programs and between the programs and the system’s external interfaces, are defined and the functionality allocated to each program is defined. Such a decomposition will often follow the physical portioning 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 cell’s material handling robot, if any.
- 3) 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 a *sequential function chart* (SFC) (see 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* (see 2.6.4 of IEC 61131-3) of the SFC is then further decomposed, typically into interconnected *function blocks* (see 2.5.2 of IEC 61131-3), i.e. a *function block diagram* (FBD) (see 4.3 of IEC 61131-3). For instance, the main program for the machining station in the cell described above may be a SFC describing the sequence of machining operations to be performed, whilst the actions might contain function blocks performing the required motion control functions.
- 4) 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. *structured text* (ST) (see 3.2 of IEC 61131-3), *instruction list* (IL) (see 3.1 of IEC 61131-3), *ladder diagram* (LD) (see 4.2 of IEC 61131-3) or *function block diagram* (FBD) (see 4.3 of IEC 61131-3).

- 5) 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.
- 6) 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.

2.4.2.5 Structured programming

The contribution of structured programming techniques to software reliability, maintainability and adaptability is well known. The structured text (ST) language defined in 3.3 of IEC 61131-3 provides a full set of constructs to support this style of programming, whilst retaining full compatibility with the other graphical and textual languages and elements in IEC 61131-3.

2.4.2.6 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 2.4.2.4 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 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.

2.4.3 Portability

As noted in 2.4.1 above, *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.

2.4.3.1 Inter-language portability

As noted in 2.4.2.1, 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 item 5) of 2.3, 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;
- the EN input and ENO output of functions can be represented directly in the LD and FBD languages, but not in the ST or IL language;
- textual expressions can only be used in the ST language, not the LD or FBD language.

Therefore, 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.

2.4.3.2 Inter-system portability

IEC 61131-3 neither defines nor requires a common exchange format for interchange or program organization unit (POU) type definitions. Consequently, compliant POU, as defined in 1.5.2 of IEC 61131-3, will only be easily portable if they are written in a textual language (ST or IL). Also, compliant POU will not be portable unless the set of features supported in the target system is equal to or is a superset of the features supported in the source system.

NOTE A common exchange format is under consideration as an extension to IEC 61131-3.

3 Application guidelines

3.1 Use of data types

Subclause 2.3.1 of IEC 61131-3 offers twenty 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 zero to one thousand, 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 vs. 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 mechanism 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, whilst 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 setup 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

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, Green, Blue); 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. The standard does not state which numerical values these enumerated values should have. There is also 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.

IEC 61131-3 provides mechanisms for the definition of *enumerated* and *subrange* types. These types do not add any new functionality to a program, but their use will at least make a program more readable and thus is a documentation aid. On some systems there might also be some kind of value checking for variables of these types throughout the lifetime of a program. This behaviour is usually referred to as “range checking.”

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 2.5.1.5.1 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.

- 1) 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.
- 2) 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), whilst 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 2-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.

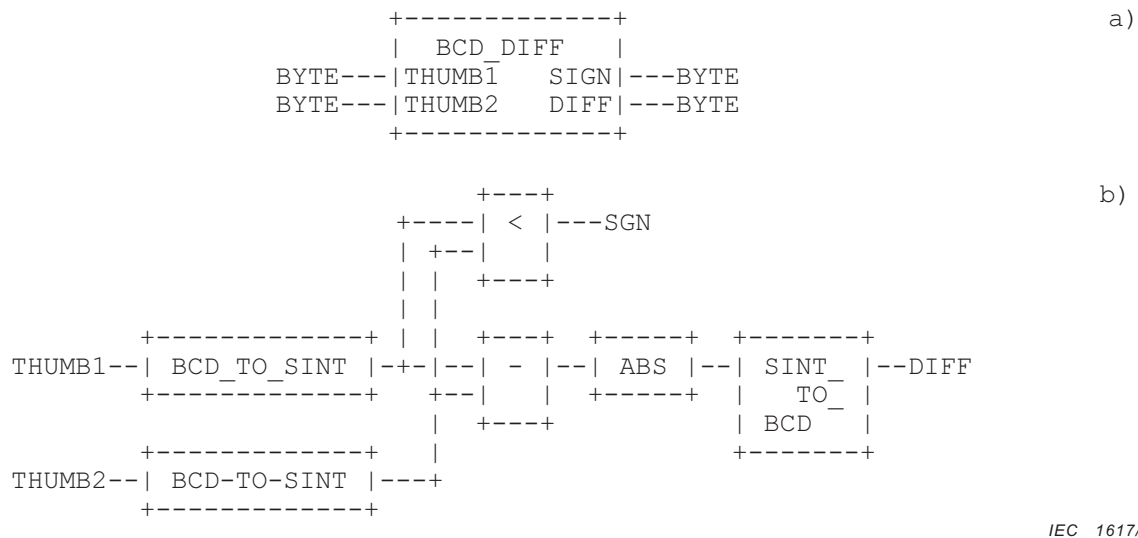


Figure 4 – Function block BCD_DIFF
a) External interface,
b) Body

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 Figures 5b) and 5c), could then be defined which produce values of this new type as the result of their execution.

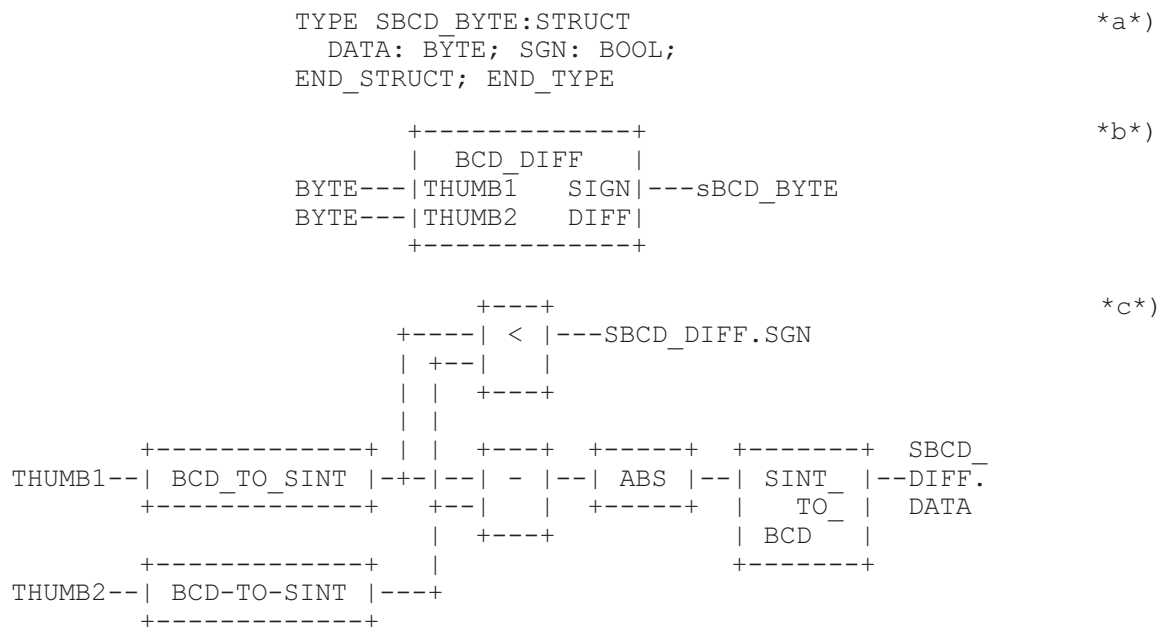


Figure 5 – Function block SBCD_DIFF
a) Definition of structured data type SBCD_BYTE
b) External interface
c) Body

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 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 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, which is required in the majority of application programs, for example for holding processes batch identifiers, recipe names, operator security codes.

IEC 61131-3 also has provision 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.

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 the following.

- 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, e.g. to initiate pre-heating of a reactor vessel before the first shift of the week.
- d) Recording the calibration date of critical analogue inputs so that the system can warn when recalibration 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, whereas, 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 completed 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 (todayDate, endTime);
```

IEC 1619/99

Figure 6 – Structured text (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 DATE_AND_TIME variable.

3.1.7 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 integer (INT) giving state of the line such as 0 for stopped, 1 to 3 for graded increases in speed.

```
VAR_IN lineState: INT ; END_VAR;
VBAR_OUT LineSpeed: REAL ; END_VAR;
VAR speeds: REAL [0..3] := (0.0, 1.0, 3.0, 9.0); END_VAR;
lineSpeed := speeds [lineState];
```

IEC 1620/99

Figure 7 – Example of array usage

3.2 Data import and export

There are several methods for the transfer of data to and from functions, function blocks and programs.

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 single value, which may be of an aggregate type (e.g. array, string or structure). Functions do not have any access to globally defined variables, nor can they use input/output variables (see 3.2.2), 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 fresh initialized local variables, possibly with values of zero if not stated differently by the function definition.

These restrictions ensure that a function will never have any side effects. It cannot modify any other variable and it only depends on the set of input parameters from its current invocation.

Within a function block body, the reading of input variables and reading or writing of output variables are permitted. Function blocks also may have input/output 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. However, a function block is not allowed to access any directly represented variables, except by means of a globally defined aliasing name that can be referenced in a "VAR_EXTERNAL" declaration. Nevertheless, the values of directly represented variables can be passed to a function block as input parameters 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, but additionally may use directly represented variables 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.

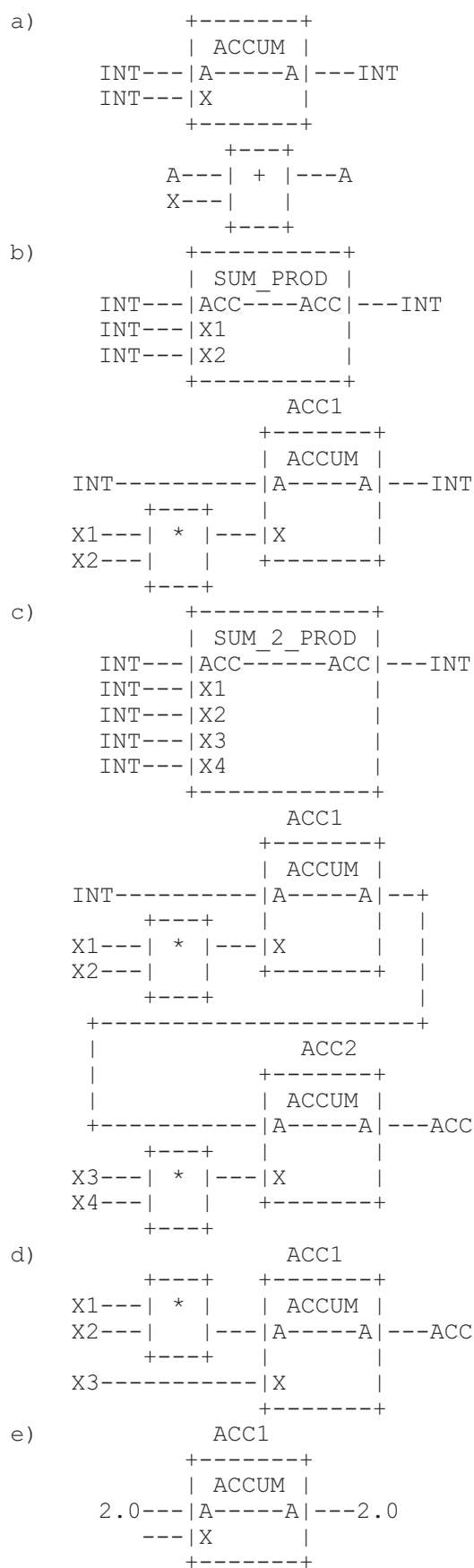
```
(* Within a CONFIGURATION, RESOURCES OR PROGRAM *)
    VAR_GLOBAL TEMP AT %IW22: INT; END_VAR
(* Access from inside a FUNCTION_BLOCK *)
    VAR_EXTERNAL TEMP: INT; END_VAR
...
... := TEMP;
```

3.2.2 Input/output (VAR_IN_OUT) variables

Input/output variables are a special kind of variable used only with 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. Input/output variables may be read or written to.

Inside a function block, input/output variables allow access to the original instance of a variable instead of a 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 input/output 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”.

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 parameter, as illustrated in figure 8d). Since an input/output variable may be written to, it follows that integrals, e.g. “2.0” or constants that have been allocated in VAR CONSTANT declarations, cannot be assigned as input/output variables, as illustrated in figure 8e).



FUNCTION_BLOCK ACCUM

```

VAR_IN_OUT A : INT ; END_VAR
VAR_INPUT X : INT ; END_VAR
A := A + X ; END_FUNCTION_BLOCK

```

FUNCTION_BLOCK SUM_PROD

```

VAR_INPUT X : INT ; END_VAR
VAR_IN_OUT A: INT ; END_VAR
VAR ACC1: ACCUM; END_VAR

```

```

ACC1(A:= ACC, X:= X1 * X2);
END_FUNCTION_BLOCK

```

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

```

ILLEGAL USAGE:

Input/output A is not a variable
or function block name

ILLEGAL USAGE:

Input/output A is not a variable
or function block name

IEC 1621/99

Figure 8 – Examples of VAR_IN_OUT usage