TECHNICAL REPORT

IEC TR 61131-8

Second edition
2003-09

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

Programmable controllers –

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

Automates programmables –

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

Reference number
IEC/TR 61131-8:2003(E)

Publication numbering

As from 1 January 1997 all IEC publications are issued with a designation in the 60000 series. For example, IEC 34-1 is now referred to as IEC 60034-1.

Consolidated editions

The IEC is now publishing consolidated versions of its publications. For example, edition numbers 1.0, 1.1 and 1.2 refer, respectively, to the base publication, the base publication incorporating amendment 1 and the base publication incorporating amendments 1 and 2.

Further information on IEC publications

The technical content of IEC publications is kept under constant review by the IEC, thus ensuring that the content reflects current technology. Information relating to this publication, including its validity, is available in the IEC Catalogue of publications (see below) in addition to new editions, amendments and corrigenda. Information on the subjects under consideration and work in progress undertaken by the technical committee which has prepared this publication, as well as the list of publications issued, is also available from the following:

- **IEC Web Site (www.iec.ch)**

- **Catalogue of IEC publications**

  The on-line catalogue on the IEC web site (www.iec.ch/searchpub) enables you to search by a variety of criteria including text searches, technical committees and date of publication. On-line information is also available on recently issued publications, withdrawn and replaced publications, as well as corrigenda.

- **IEC Just Published**

  This summary of recently issued publications (www.iec.ch/online_news/ justpub) is also available by email. Please contact the Customer Service Centre (see below) for further information.

- **Customer Service Centre**

  If you have any questions regarding this publication or need further assistance, please contact the Customer Service Centre:

  Email: custserv@iec.ch
  Tel: +41 22 919 02 11
  Fax: +41 22 919 03 00

IECNORM.COM : Click to view the full PDF of IEC TR 61158-5:2003
Withdrawn

TECHNICAL REPORT

IEC TR 61131-8

Second edition
2003-09

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

Programmable controllers –

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

Automates programmables –

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

© IEC 2003 — Copyright - all rights reserved

No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

International Electrotechnical Commission, 3, rue de Varembé, PO Box 131, CH-1211 Geneva 20, Switzerland
Telephone: +41 22 919 02 11 Telefax: +41 22 919 03 00 E-mail: inmail@iec.ch Web: www.iec.ch

Commission Electrotechnique Internationale
International Electrotechnical Commission
Международная Электротехническая Комиссия

PRICE CODE XD

For price, see current catalogue

CONTENTS

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

INTERNATIONAL ELECTROTECHNICAL COMMISSION

PROGRAMMABLE CONTROLLERS –

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

FOREWORD

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

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

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

This second edition cancels and replaces the first edition, published in 2000, and constitutes a technical revision.

The main changes with respect to the previous edition are to make IEC 61131-8 consistent with IEC 61131-3, 2nd edition.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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

Enquiry draft	Report on voting
65B/478/DTR	65B/492/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 publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

The committee has decided that the contents of this publication will remain unchanged until 2008. At this date, the publication will be

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

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

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003

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.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003

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

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003

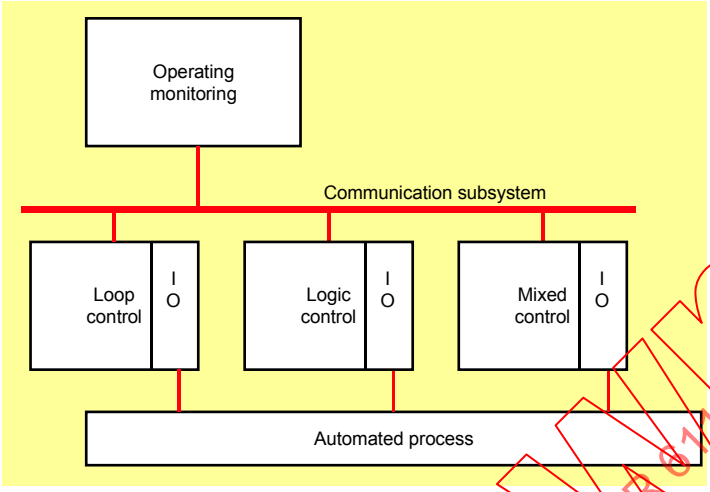


Figure 1 – A distributed application

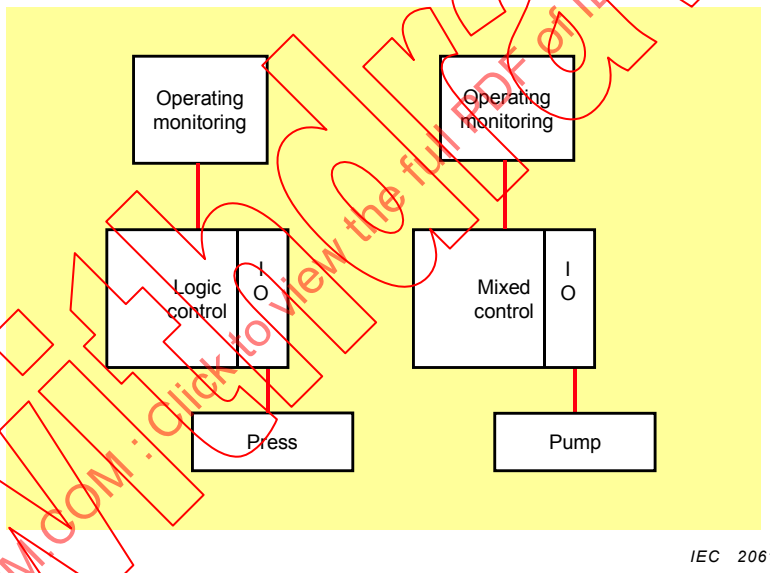


Figure 2 – Stand-alone applications

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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.

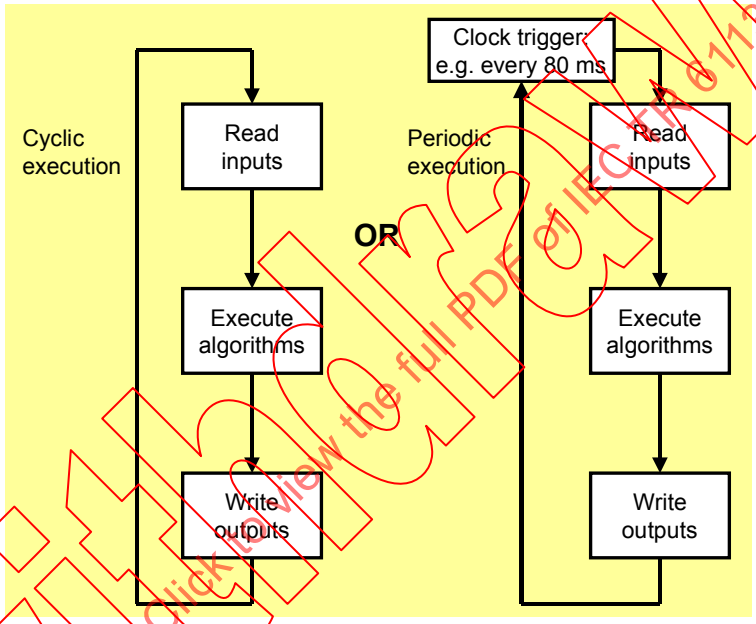


Figure 3 – Cyclic or periodic scanning of a program

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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 application

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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.

a) The desired functionality and external interface of the system are specified. For instance, the basic functionality of a machining cell might be to accept a rough part from a material handling system, produce a finished part from the rough blank, measure the finished part, pass it back to the material handling system, and report the results of the operation to a manufacturing information system. External interfaces in this cell would include the material handling and information system interfaces. The *configuration elements* described in 2.7 of IEC 61131-3 can be used in this step.

b) A first decomposition of the system design is made by allocating the required functionality into one or more elements, typically *programs* (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.

c) Each element defined in the preceding step is further decomposed into more basic functional units. If the functionality of the element is essentially sequential, the first step in the decomposition may be the formulation of an *SFC* (2.6 of IEC 61131-3) expressing the sequence of operations to be performed and the conditions for repeating the cycle of

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003

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.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003

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.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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.

a) 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.

b) 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

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003

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.

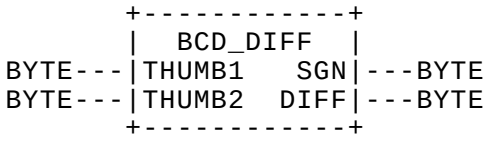


Figure 4a – External interface

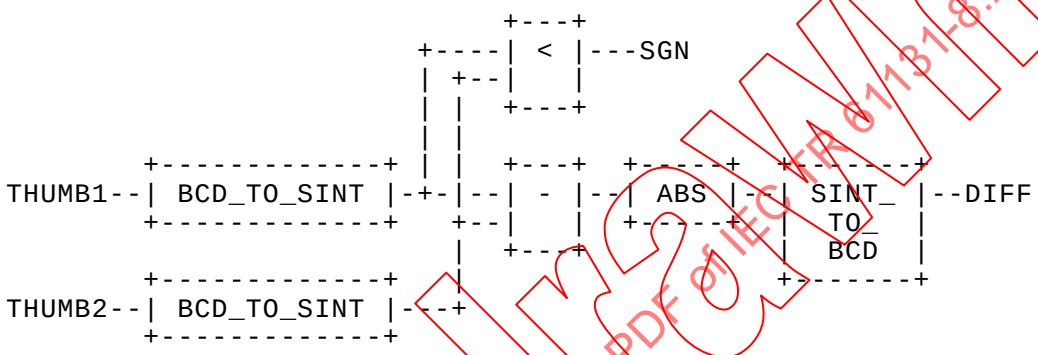


Figure 4b – Body

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

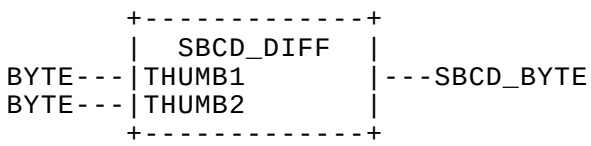


Figure 5b – External interface

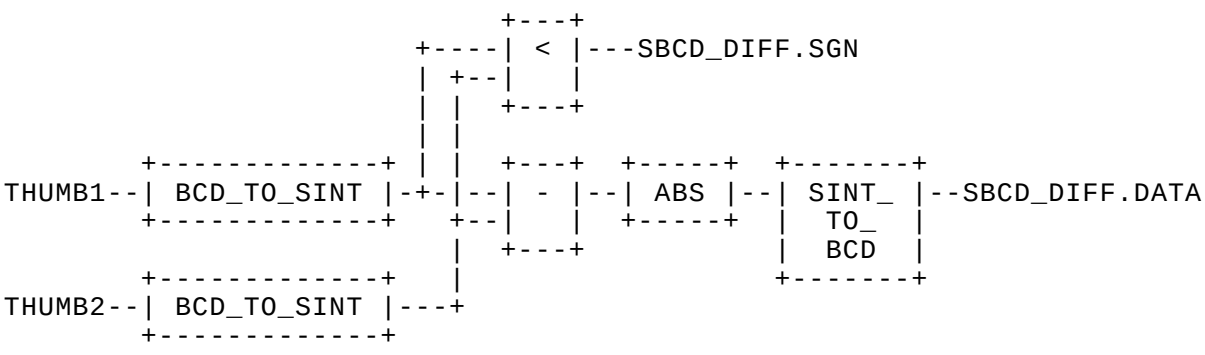


Figure 5c – Body

IEC 2064/03

Figure 5 – Function block SBCD_DIFFF

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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, secon 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, `speeds` is an

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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 lineState: INT; END_VAR;
```

```
VAR_OUT lineSpeed: REAL; END_VAR;
```

```
VAR
  speeds: ARRAY[0..3] OF REAL:= (0.0, 1.0, 3.0, 9.0);
  accelerations: ARRAY[0..3] OF REAL;
  positions: ARRAY[1..4] OF REAL;
  I: INT;
END_VAR;
```

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

IEC 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 .
```

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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 EC 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.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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 POUs 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

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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 POUs, 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*X2 at each invocation of an instance of the function block SUM_PROD. There is no need to copy the output value from ACC1.A back into the variable ACC. Similarly, in Figure 8c, the variable ACC will be augmented by the sum of products X1*X2 + X3*X4 at each invocation of an instance of the function block SUM_2_PROD.

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

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.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

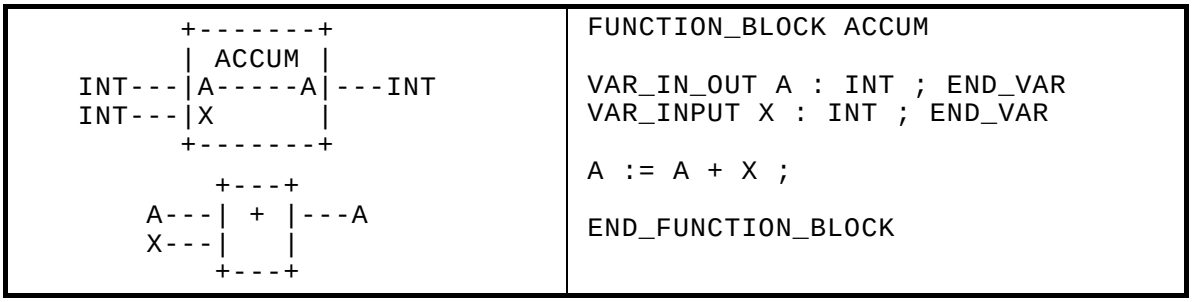


Figure 8a – Declaration of a FB type using VAR_IN_OUT

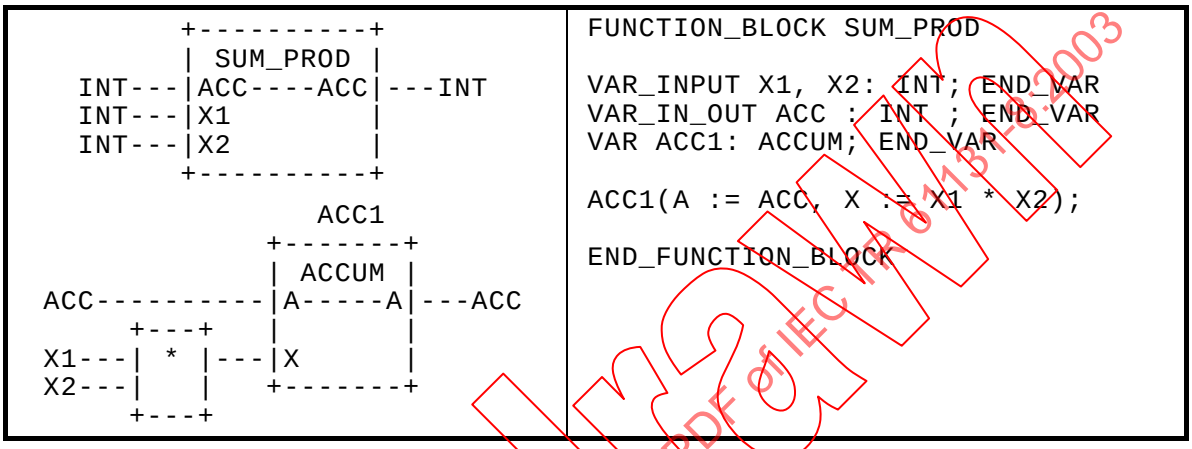


Figure 8b – Usage of an instance of a FB type using VAR_IN_OUT

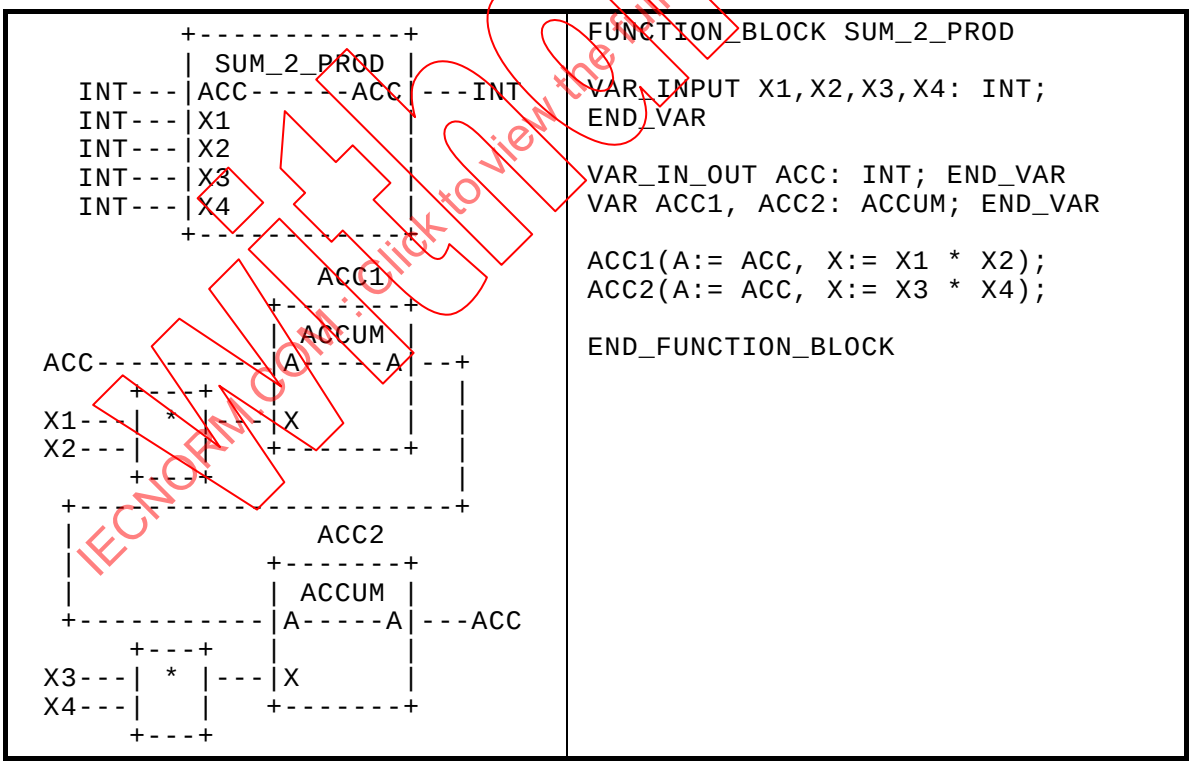


Figure 8c – Usage of two instances of a FB type using VAR_IN_OUT

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

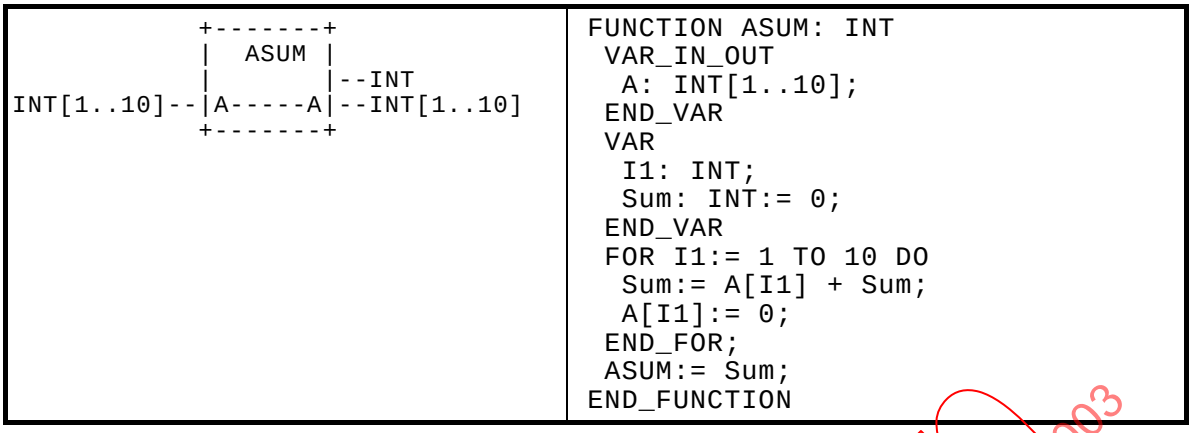


Figure 8d – Declaration of another FB type using VAR_IN_OUT

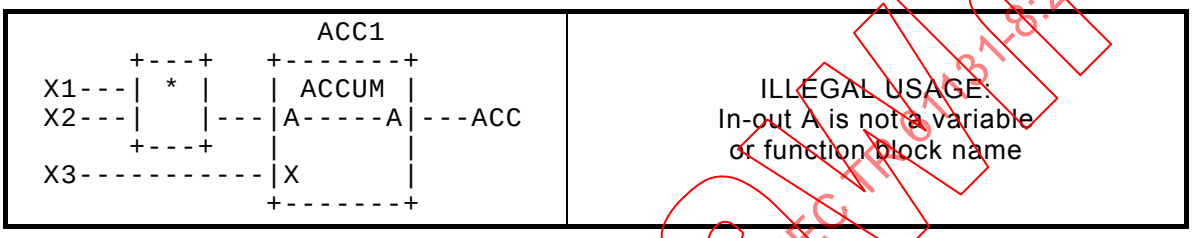


Figure 8e – Example of illegal usage

Figure 8f – Example of illegal usage

IEC 2067/03

Figure 8 – Examples of VAR_IN_OUT usage

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.

a) 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.

b) 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.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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

	DESCRIPTION	EXAMPLE (Note 8)

a)	Formal function call in ST (Note 1)	`A:= LIMIT(EN:= COND,`
		`IN:= B,`
		`MX:= 5,`
		`ENO => TEMP);`

b)	Non-formal function call in ST	`A:= LIMIT(1, B, 5);`

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

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

is assumed in the above examples.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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

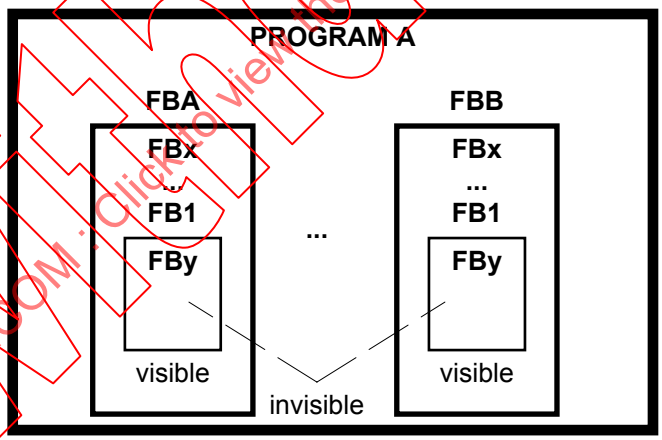


Figure 9c – Visibility of function block instances

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

IEC 2068/03

Figure 9 – Hiding of function block instances

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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 3.7 of this technical report for more details.

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

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

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

NOTE 2 The formal invocation of a function block has to be used, if not all input variables of a function block instance are assigned. A non-formal invocation of a function block requires the complete set of actual arguments in its argument list.

NOTE 3 IEC 61131-3, second edition, has introduced specific initializations of inputs for single instances of a function block (see IEC 61131-3, second edition, Table 18, feature 10). See A.3 of this technical report for the rationale for this change.

3.4 Differences between function block instances and functions

Although both function blocks and functions are POUs, there are significant differences between function blocks and functions (see 2.5.1 of IEC 61131-3).

a) The invocation of a function has one result.

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

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

3.5 Use of indirectly referenced function block instances

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

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

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

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

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

3.5.1 Establishing an indirect function block instance reference

Items 6) and 7) of 2.5.2.2, Table 33, B.1.4.3 and B.1.5.2 of IEC 61131-3 define the mechanisms for establishing an indirect reference to an instance of a function block. As illustrated in Figure 10 below, the function block reference may be established as

a) a variable declared in a VAR_INPUT declaration;

b) a variable declared in a VAR_IN_OUT declaration;

c) an external variable.

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

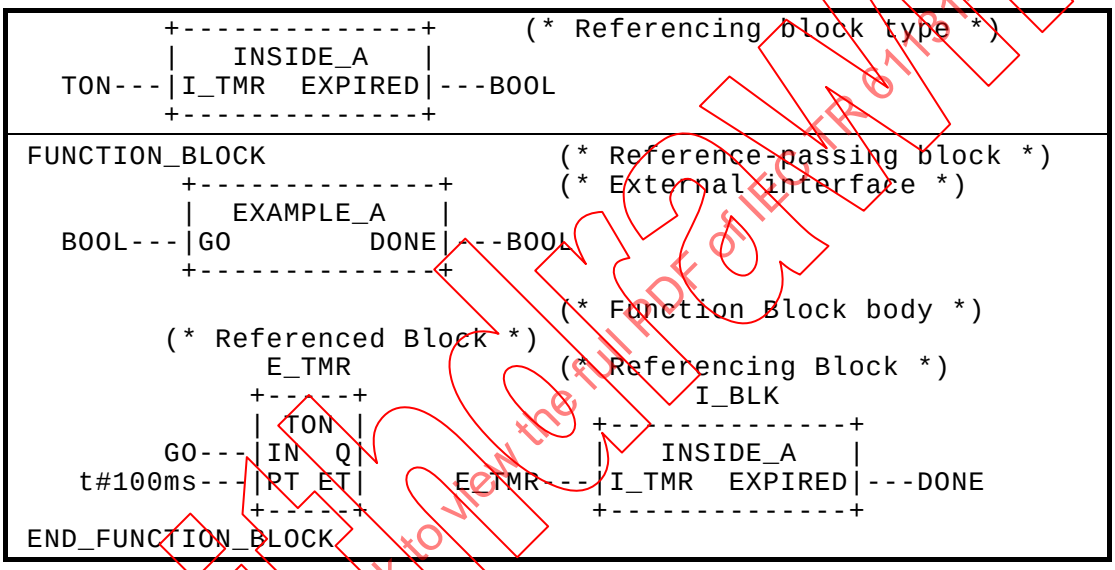


Figure 10a – Function block name used as an input variable

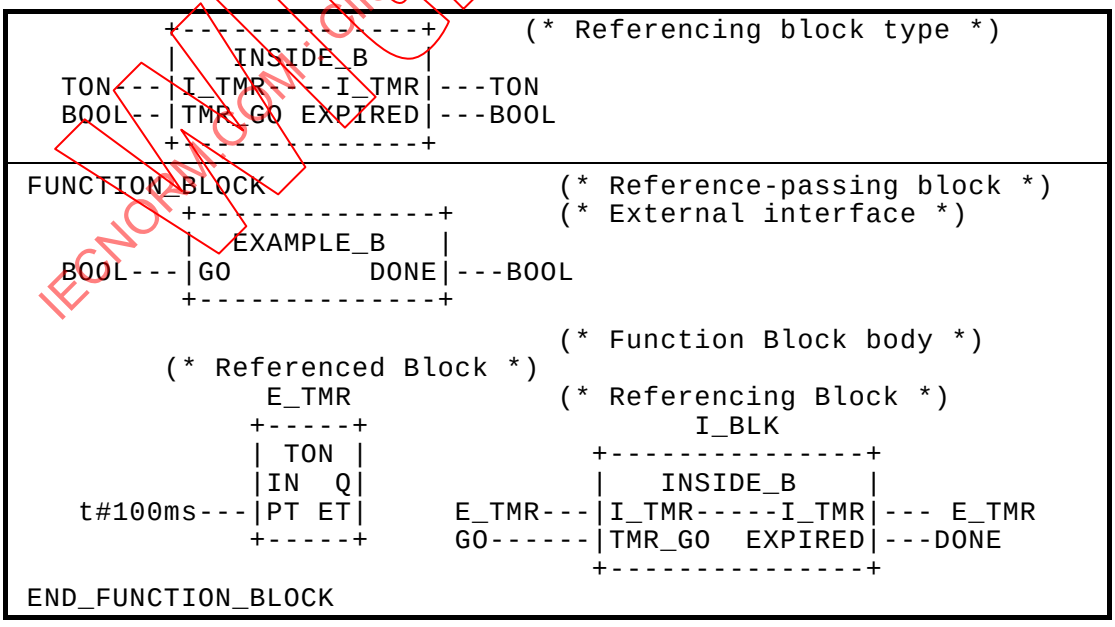


Figure 10b – Function block name used as an in-out variable

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

```
FUNCTION_BLOCK                    (* Referencing block type *)
        +--------------+          (* External interface *)
        |   INSIDE_C   |
  BOOL--|TMR_GO EXPIRED|---BOOL
        +--------------+

VAR_EXTERNAL X_TMR: TON; END_VAR
END_FUNCTION_BLOCK
```

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

Figure 10c – Function block name used as an external variable

IEC 2069/03

Figure 10 – Graphical use of a function block name

3.5.2 Access to indirectly referenced function block instances

Access to an indirectly referenced function block instance means *reading* (i.e., using without modifying) its output variables. The private data area of the function block instance remains unchanged (see Table 32 and 2.5.2.2, item 4, of IEC 61131-3).

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

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

Figure 11a –Referenced input block

```
          I_TMR
        +-----+
        | TON |
TMR_GO--|IN  Q|---EXPIRED
        |PT ET|
        +-----+
```

Figure 11b – Referenced in-out block

```
          X_TMR
        +-----+
        | TON |
TMR_GO--|IN  Q|---EXPIRED
        |PT ET|
        +-----+
```

Figure 11c – Referenced external block

IEC 2070/03

NOTE See Figure 10 for corresponding variable and interface definitions.

Figure 11 – Access to an indirectly referenced function block instance

3.5.3 Invocation of indirectly referenced function block instances

As described in item 5 of 2.5.2.2 of IEC 61131-3, the *invocation* of an indirectly referenced function block instance means the invocation of the function block instance from inside

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

another function block. Using this capability, the private data area (and therefore also the output variables) of this function block instance can be modified.

The general rules for an invocation of a function block instance given in IEC 61131-3, Table 32 are valid in this case.

– The assignment of a value to an input variable of the indirectly referenced function block instance is allowed only as part of the invocation of this function block.
– The output variables of this function block instance cannot be modified from outside.

As defined in B.1.4.3 and B.1.5.2 of IEC 61131-3, the reference to the invoked function block instance can be established via

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

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

An invocation of an indirectly referenced function block instance established via an input variable is not allowed, since an invocation of this function block instance may change the values of the private data area. The modified values may have effects outside the invoked function block. This behaviour is prohibited for input variables. As noted in 3.5.2, this precludes the graphical representation (which implies invocation) of indirectly referenced function block instances established via input variables.

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

Figure 12a – Textual declaration and invocation

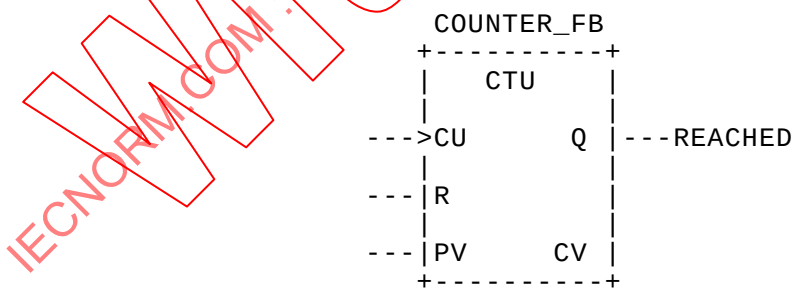


Figure 12b – Graphical invocation

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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

Figure 12c – Textual passing of instance name

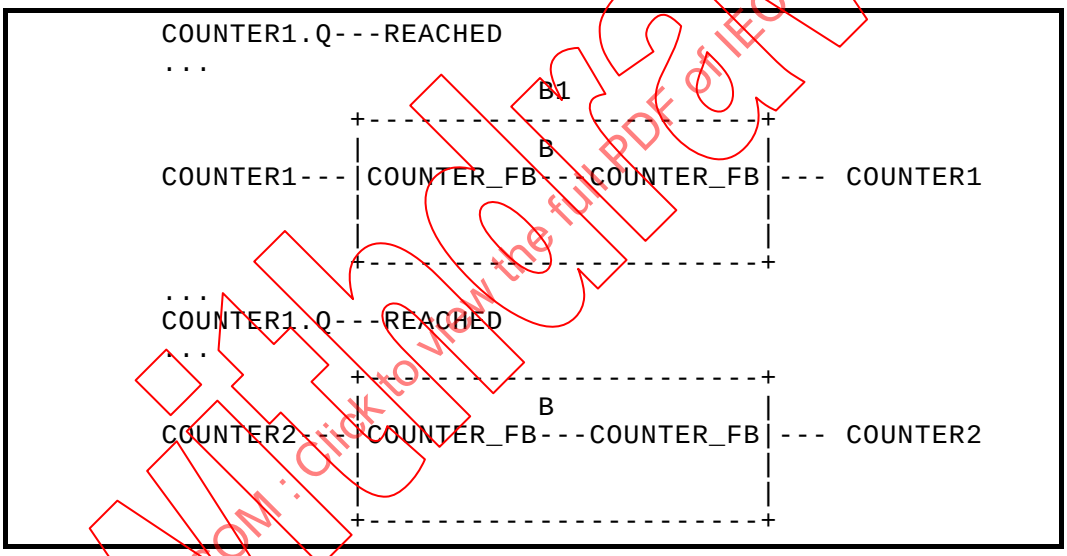


Figure 12d – Graphical passing of instance name

IEC 2071/03

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

Figure 12 – Invocation of an indirectly referenced function block instance

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

As shown in IEC 61131-3, Figure 11c), a function block instance can be declared as a *global variable* in a *program*, and the function block instance name can then be used as an *external* variable in function blocks of other types. Although the particular instances of such function blocks may vary between programs, the referenced function block instance does not vary from one invocation to the next of the *referencing* function block (the block containing the external reference). Such a function block instance may be used, for example, to provide means for externally setting the values of variables of all instances of a function block type that contains the function block name as an external reference.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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

This fact has implications for the validity checking provided by the PSE for the scope of a function block instance. It is strongly recommended that this scope check should be done during edit time, so that there is no need for the user to wait until run time for a scope check. The consequence of this requirement is that the function block instance names used as arguments are fixed and not kept within a variable. Also, there is neither a chance to concatenate this string at run time nor to access a string array keeping this function block instance name.

3.5.4 Recursion of indirectly referenced function block instances

According to 2.5 of IEC 61131-3, invocations of POUs shall not be recursive. This is also true for invocations of function blocks.

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

Recursion within programmable controller programming languages is discussed in more detail in 3.6 of this technical report.

3.5.5 Execution control of indirectly referenced function block instances

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

a) Direct association of tasks to such function blocks, as described in 2.7.2 of IEC 61131-3, should be avoided.

b) Use of such function blocks should be restricted if possible to algorithms written in the ST language defined in 3.3 of IEC 61131-3, which has explicit function block invocation statements.

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

3.5.6 Use of indirectly referenced function block instances in functions

As mentioned in 2.5.2 of IEC 61131-3, a function block instance name can be used as an input variable of a function. Within the body of this function, the output variables of the indirectly referenced function block instance can be read.

At a first view this seems to be contrary to the definition of a function in 2.5.1 of IEC 61131-3. A call of a function with the same input variables always has to yield the same output value. This may not be true, if the same function block instance name is passed to a function and this function block instance has been invoked in the meantime. A more detailed investigation of this fact, however, shows that the read access to a function block instance is similar to a read access of a structured variable. To get the same function value, it is therefore not enough to ensure that the name of a function block instance is identical from one function call to the next but also that the values of all output variables of the referenced function block instance are identical.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

3.6 Recursion within programmable controller programming languages

As stated in 2.5 of IEC 61131-3, recursive calls of POUs are not allowed. That means that a POU shall not contain or invoke another POU of the same type. This is true also if the recursion is not direct but indirect, for example, if a POU A calls a POU B and this POU B calls POU A again.

Because of run-time performance, it is recommended that the program be checked for possible recursion by the PSE during edit time. This requires the ability to detect possible recursion by traversing and checking the static call hierarchy tree. In every possible combination of the path from the PROGRAMS associated with a TASK to the deepest level call of a POU, no name of a POU may occur more than once.

The edit time check for recursion is possible also in connection with the use of function block instance names as arguments as the recursion is bound to function block types rather than to function block *instances*.

3.7 Single and multiple invocation

Some programmable controllers allow assignment of only one value to any input of a function or function block, while others allow multiple assignments to any input. If a programmable controller system is regarded as a replacement for electrical circuitry, no multiple assignments may exist as these would introduce "short circuits" of outputs, equivalent to the "wired or" of outputs in FBDs. On the other hand, there is no reason why a function block implemented in a digital computing entity could not be invoked from different locations, with different input values in each invocation.

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

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

An example of multiple invocation in ST is as follows:

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

Here a function block named `aFB` is expected to have a Boolean input variable named "IN". Depending on the current value of some Boolean expression this function block is invoked with either a value of `0` or a value of `1` for this input. There are two invocations of the function block `aFB` with different values assigned to an input. In a system that allows multiple assignment this would not cause any problems; but in a strict single assigning system the `IF` statement would have to be reformulated to

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

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

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

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

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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

IEC 61131-3 does not stipulate whether a system uses strict single invocation or allows multiple invocations. Both have advantages and shortcomings. With a single invocation rule there is one, and only one, place where the input variable of a function block will be assigned a value, which increases the reliability and maintainability of the software. With multiple invocation, programmable controller programs may be maintained more easily because of a lower level of nesting of functions, and may have better responsiveness and processing capacity by avoiding superfluous calculations.

NOTE See 3.12.3 for a description of additional situations in which multiple invocations should not be used.

3.8 Language specific features

3.8.1 Edge-triggered functionality

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

– as function block inputs in 2.5.2.2;

– as `R_TRIG` and `F_TRIG` function blocks in 2.5.2.3.2;

– as positive (`P`) and negative (`N`) transition-sensing contacts and coils for the LD language in 4.2.3, Table 61, and 4.2.4, Table 62, respectively.

3.8.1.1 Edge-triggering in LD language

As noted in 2.1 of this technical report, many programmable controller systems use cyclic execution to implement sampled-data control. In the IEC 61131-3 notation, this is accomplished by associating the program that implements the control algorithm with a periodic TASK as described in 2.7.2 of IEC 61131-3. Users should be aware that in such systems, in worst-case conditions, the effect of a change of state in an edge-triggered input may not appear at the system outputs until two scan cycles later, as noted in Figure 3 above. This can be illustrated by the simple LD network and timing diagram shown in Figure 13.

Figure 13a – Example network

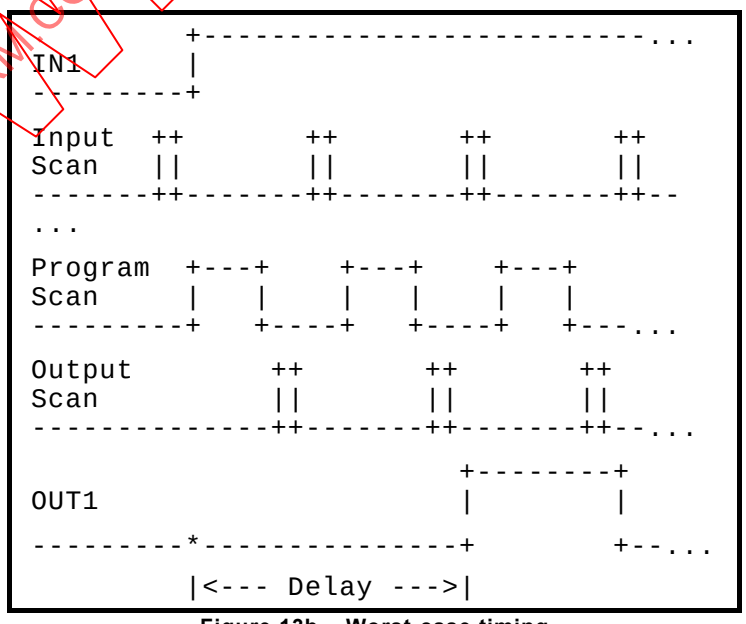


Figure 13b – Worst-case timing

Figure 13 – Timing of edge triggered functionality

IEC 2072/03

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

3.8.1.2 Use of edge-triggered function blocks

The R_TRIG and F_TRIG function blocks defined in 2.5.2.3.2 of IEC 61131-3 exhibit different behaviour following “cold restart” as described in 2.4.2 of that document. The Q output of an R_TRIG instance can be set to 1 on the first invocation but the Q output of an F_TRIG instance always requires at least two invocations before being set to 1. This behaviour can be explained as follows.

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

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

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

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

3.8.2 Use of EN/ENO in functions and function blocks

Subclause 2.5.1.2 of IEC 61131-3 defines the Boolean input EN and output ENO, which can be used to control the execution of functions. Typically, these variables are used to provide Boolean “power flow” through the functions in the LD language. However, they can also be used in the FBD language and are especially useful in eliminating ambiguities that might be caused by the use of the graphical execution control elements described in 4.1.4 of IEC 61131-3.

NOTE IEC 61131-3, second edition, now allows the use of EN and ENO in textual languages as well as graphical languages and also in the declaration of function blocks types as well as function types.

Figure 14 shows an application of this principle to portions of the FBD body of the STACK_INT function block example given in IEC 61131-3, Clause F.4. Combinations of the Boolean inputs PUSH, POP, and R1 are used in conjunction with the EN and ENO variables to achieve unambiguous execution control without jumps and labels.

According to rule 3 of 2.5.2.1 of IEC 61131-3, the ENO output of a function is to be reset to FALSE (0) upon the occurrence of an error condition in the execution of the function. This feature can be used to prevent the propagation of erroneous values through subsequent functional evaluations. However, the use of ENO for extensive run-time error reporting and diagnosis is not recommended; instead, the procedures described in 4.6.2 of this technical report should be employed.

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

Figure 14a – ST language without EN/ENO

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

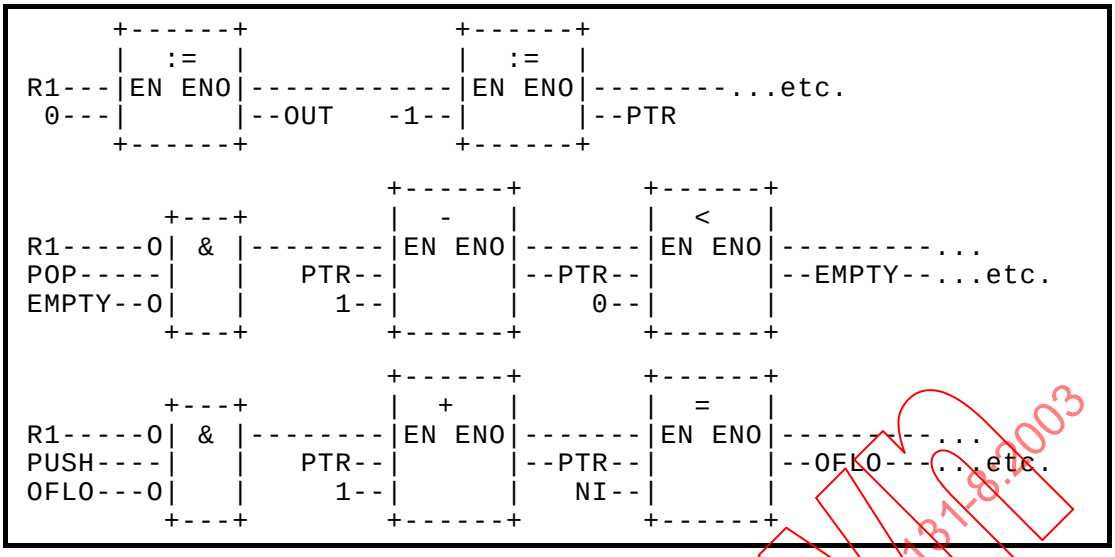


Figure 14b – FBD language with EN/ENO

Figure 14 – Execution control example

3.8.3 Use of non-IEC 61131-3 languages

Subclause 1.4.3 of IEC 61131-3 states that other programming languages such as Pascal and C may be used, in addition to those defined by IEC 61131-3, in the programming of *functions* and *function blocks*. Regardless of which language is used for such programming, interface information as discussed in 5.5 of this technical report must be provided in order to permit the use of these program organization unit types in the PSE.

3.9 Use of SFC elements

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

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

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

3.9.1 Action control

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

As defined in 2.6.4.5 of IEC 61131-3, the action control block contains a logic description of the effects and interactions of action qualifiers. In order to gain a better understanding of the action control, a copy (instance) of the action control block is assigned to each action. Therefore, the action control block is a constituent part of an action: it defines the execution of the action (one could also say that it “controls” the execution).

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

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

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

a) The behaviour (function) of an action in parts of the program is mainly determined by the following features:
   - status of step flag;
   - action qualifier;
   - action;
   - action control block.

b) The action control block describes (logically) the operating method (function) and interplay of action qualifiers. (An action can be defined with different action qualifiers within one program or part of a program.) Action qualifiers are

   N non-stored;
   S set/stored;
   R overriding reset;
   P pulse;
   L time-limited;
   D time-delayed;
   DS delayed and stored;
   SD stored and time-delayed;
   SL stored and time-limited;
   P1 rising edge pulse (when entering step);
   P0 falling edge pulse (when leaving step).

   Figures 15a) and 15b) of IEC 61131-3 represent the logical flow of the activation as a wiring diagram. Table 45a of IEC 61131-3 defines two action control features: 1) with “final scan” as shown in Figure 15a) and 2) without “final scan” as shown in Figure 15b).

   NOTE 1 A typical use of the P0 qualifier is to substitute for a “final scan” where needed in systems which do not implement feature (1) of Table 45a.

   NOTE 2 The existence of two types of permissible action control (“final scan” versus no “final scan”) limits program portability. Users should check that the action control features supported are identical between two different programmable controller systems before attempting to port an SFC from one system to another.

   NOTE 3 The influence of the P0 and P1 actions on the “Q” output of the action control block is inconsistent between the “final scan” and “no final scan” implementations in Figures 15a) and 15b) of IEC 61131-3, second edition, respectively. That is, the Q output is false (0) during the action of P1 or P0 in Figure 15a), and true (1) during the action of P1 and P0 in Figure 15b). Users should be especially careful of this difference before attempting to port an SFC between systems providing different treatment of “final scan”.

   Any section of an SFC network usually comprises a succession of action blocks, which often only differ in the choice of the attribute. Furthermore, several different actions can be connected to one step, as illustrated in Figure 16a) of IEC 61131-3. Therefore, the following general observations can be made.

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

   Figure 16b) of IEC 61131-3 illustrates the transformation of the SFC net in Figure 16a) into a block wiring diagram, corresponding to the realization of the action blocks of the

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

SFC as an (optimized) wiring net. The actual executable code generated need not be identical to this figure, as long as its behaviour is functionally equivalent.

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

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

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

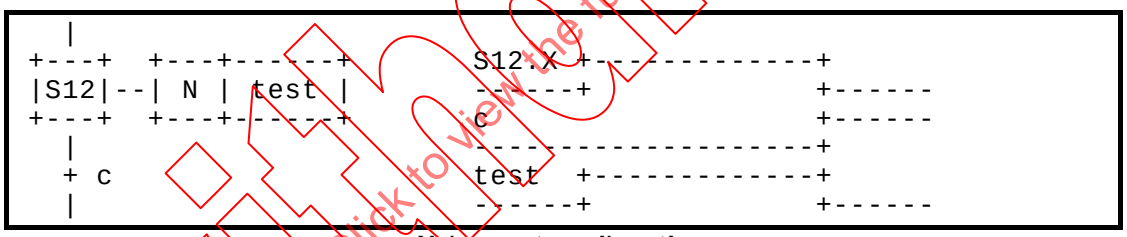
3.9.2 Boolean actions

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

Figure 15 shows all the qualifier functions for Boolean variables, based on the description of action qualifiers in IEC 60848. In these figures, it is assumed that `test` has been declared as a variable of type BOOL.

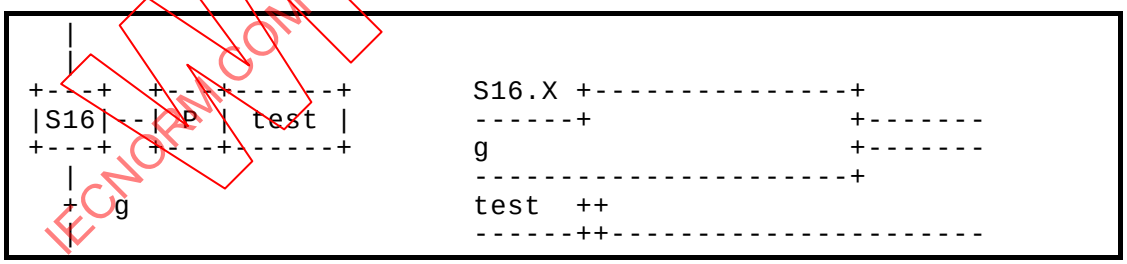
NOTE 1 If a combination of various qualifiers is required, it is recommended to refer to the detailed description of the action control block in 2.6.4.5 of IEC 61131-3 in order to gain a better understanding of the resulting operations.

NOTE 2 The use of P1 and P0 qualifiers will always result in a false value of an associated Boolean action when "final scan" (feature 1 of Table 45a of IEC 61131-3) is implemented. When "no final scan" (feature 2 of Table 45a of IEC 61131-3) is implemented, the associated Boolean action will be true for the single scan when the step is entered or exited, respectively. Therefore, the use of these qualifiers for Boolean actions is not recommended.



N (non-stored) action

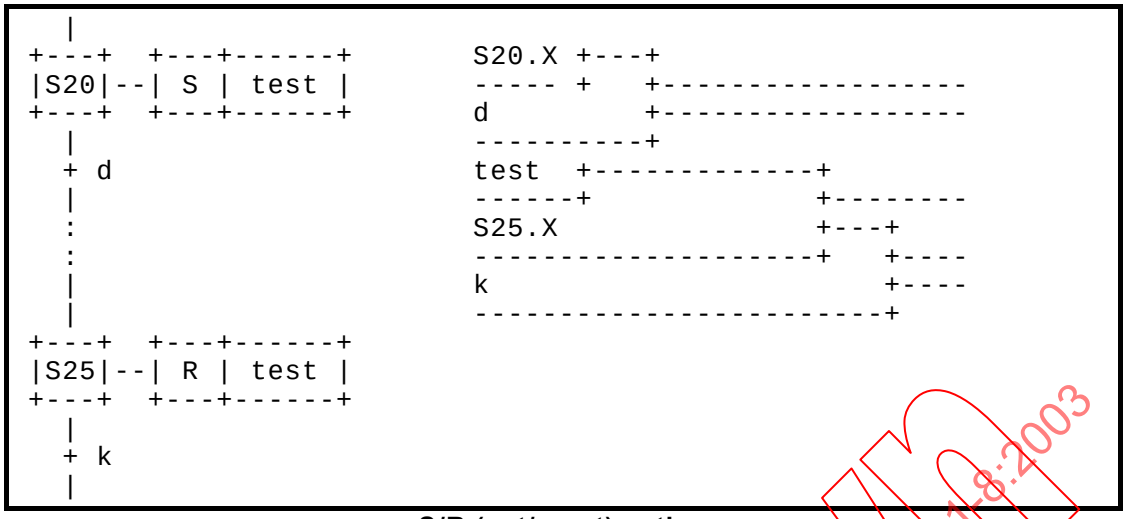
The Boolean variable "test" is set as long as the step S12 is active.



P (pulse) action

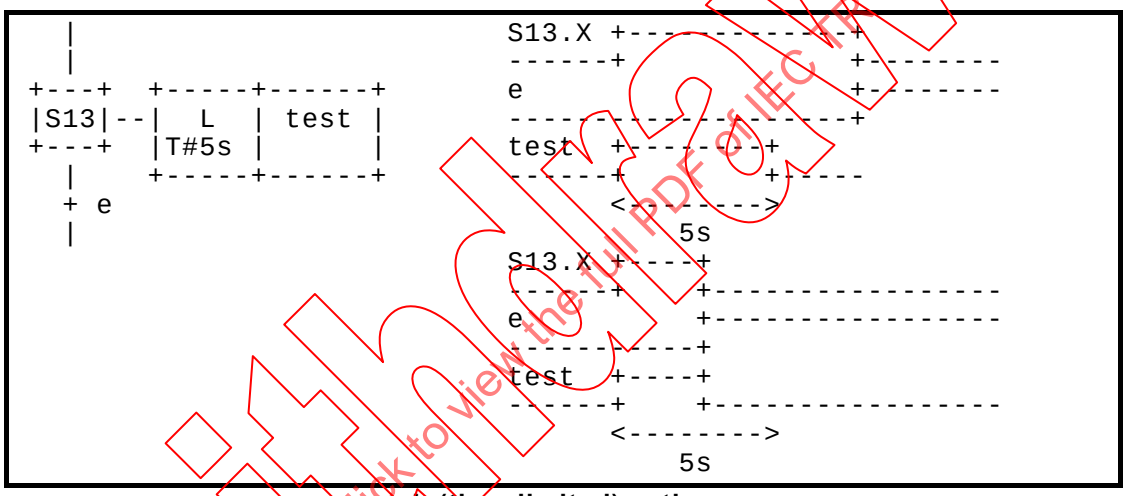
As soon as the step is active, the Boolean variable is set for one operating cycle.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003



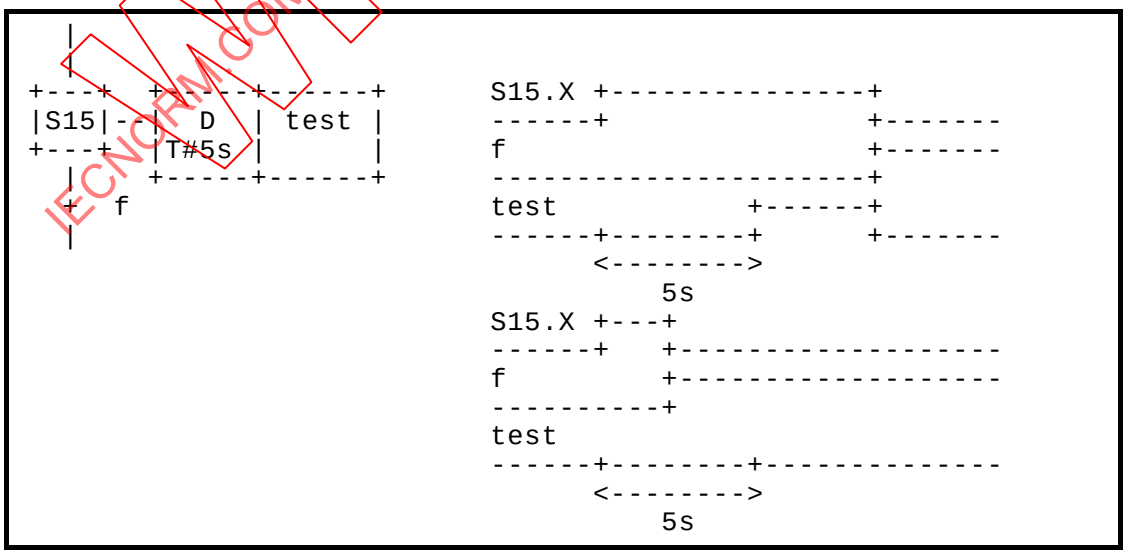
S/R (set/reset) action

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



L (time-limited) action

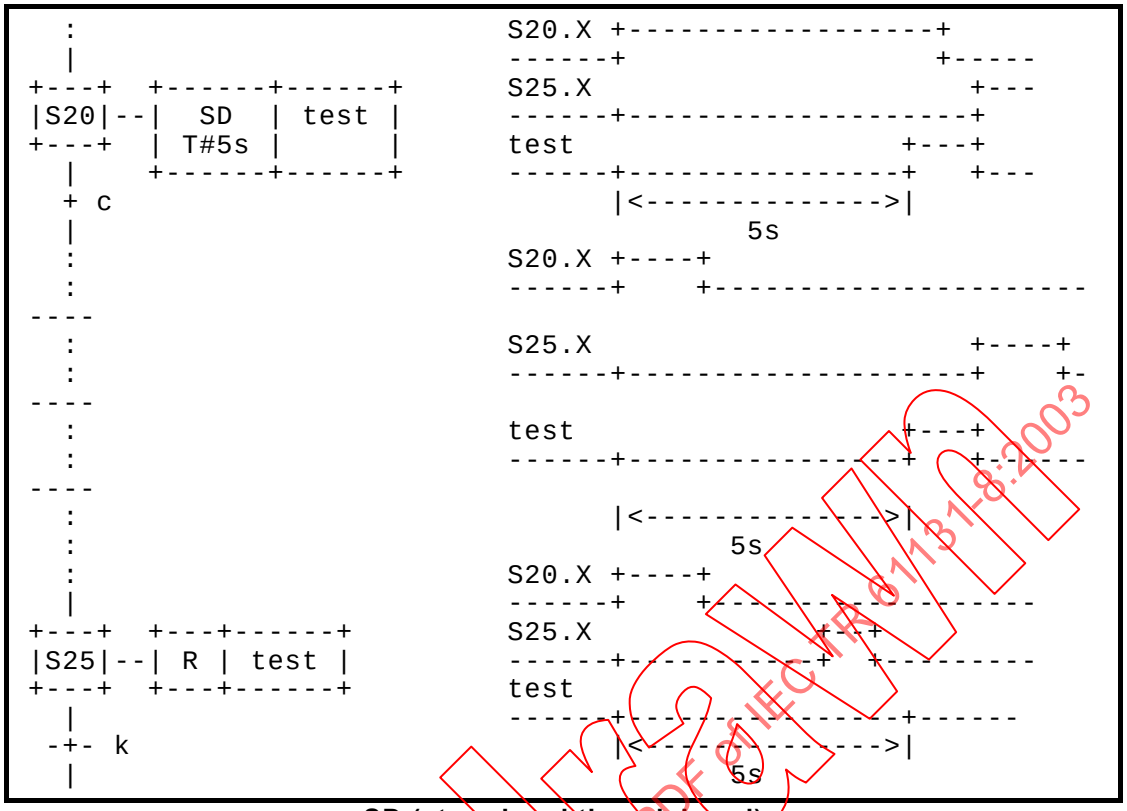
The Boolean variable is set for a preset length of time,
as long as the corresponding step is active.



D (time-delayed) action

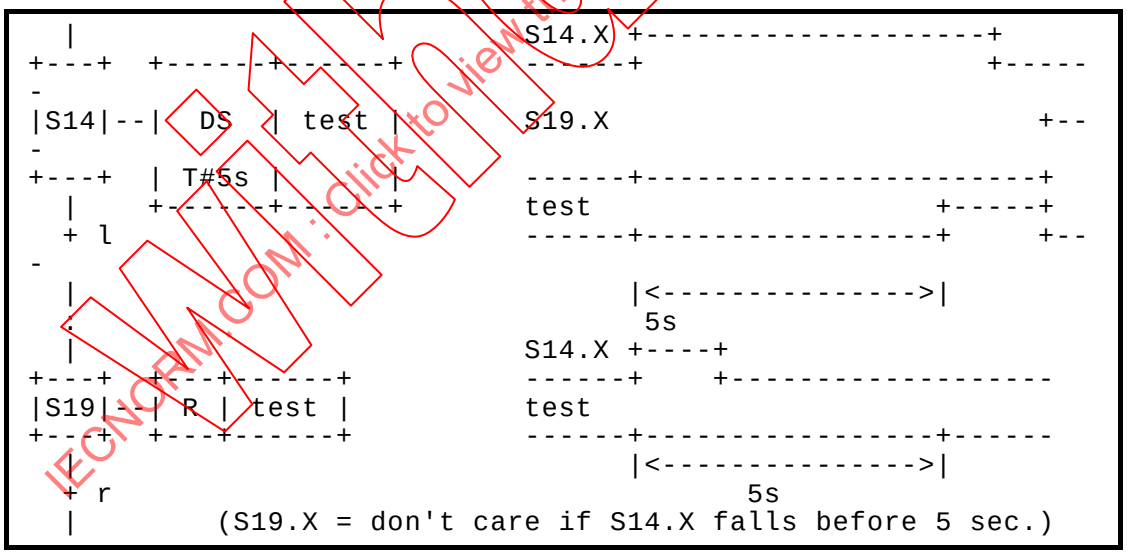
The Boolean variable is set after a preset time has elapsed and remains set for
as long as the corresponding step is active.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn



SD (stored and time-delayed)

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

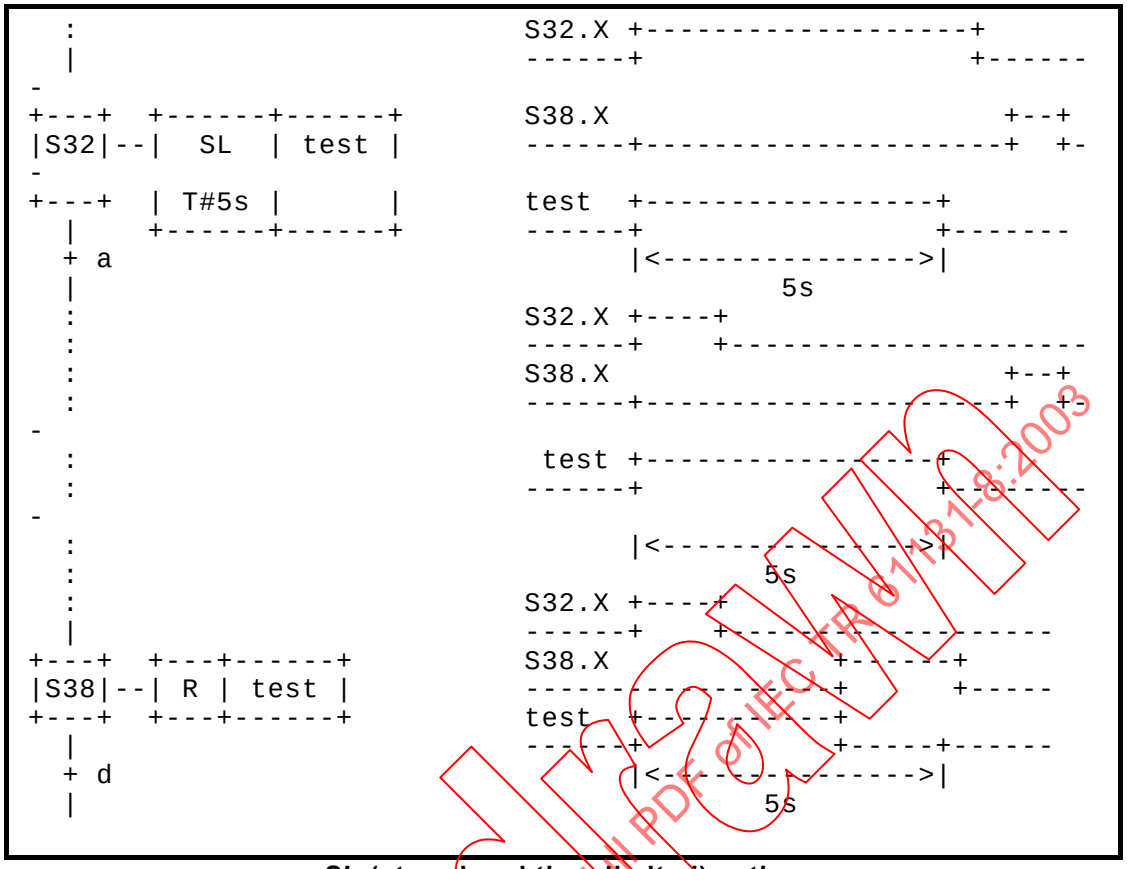


DS (time-delayed and stored)

In this case, as for the SD qualifier, the Boolean variable is set with a time delay.

It differs from the SD qualifier in that the corresponding step must remain active during the time delay.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn



SL (stored and time-limited) action

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

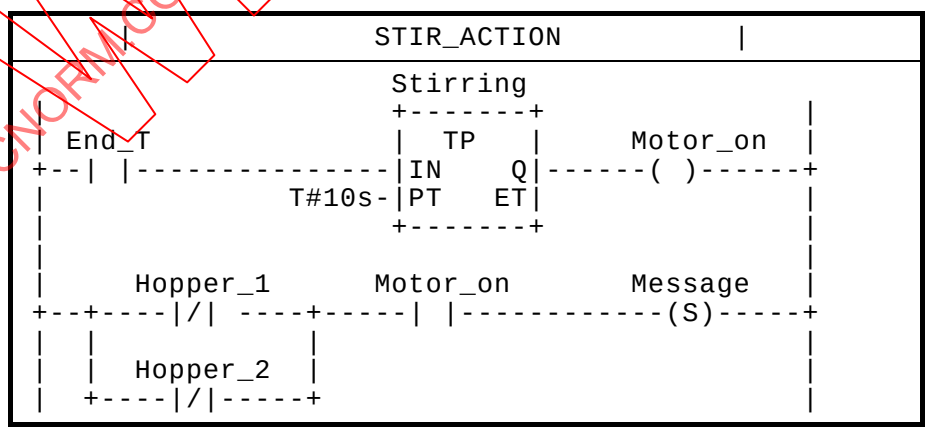
IEC 2074/03

Figure 15 – Timing of Boolean actions

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

3.9.3 Non-SFC actions

In addition to Boolean variables, more complex algorithms can also be programmed within actions, as illustrated in Figure 16.



IEC 2075/03

Figure 16 – Example of a programmed non-Boolean action

The processing of such actions is dependent on the status of the Q output of the corresponding action control block. While output Q is TRUE, the code of the action is continuously enabled to be executed, under the control of the enclosing program organization unit such as a program or function block. Upon the transition of Q from TRUE to FALSE, the

code is executed one final time. During this final execution, the corresponding step flag is already set to `0`.

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

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

In order to avoid unexpected results, programmers must bear in mind that each action is evaluated at least twice (once with `Q = 1` and once with `Q = 0`). Figure 17 illustrates this point in the context of the `P` (pulse) qualifier. In this example, the purpose is to maintain an integer count, `S15_CT`, of the number of times step `S15` is activated. As shown in Figure 17a), this is to be accomplished by action `A15`. Figure 17b) illustrates an incorrect body for this action, which will result in `S15_CT` being equal to twice the number of activations, since `A15` will be invoked once upon activation of `S15` and again in the next scan. Figures 18c) and d) illustrate possible solutions to this problem in the ST and FBD languages, respectively.

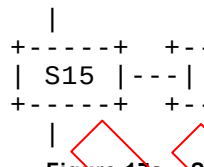


Figure 17a – SFC fragment

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

Figure 17b – Incorrect body for action A15 (ST language)

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

Figure 17c – Correct body for action A15 (ST language)

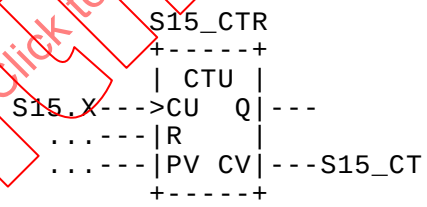


Figure 17d – Correct body for action A15 (FBD language)

Figure 17 – Use of the pulse (P) qualifier IEC 2076/03

3.9.4 SFC actions

An *SFC action* is an action that contains one or more complete SFC networks. Just as with other actions, an SFC action can be associated with several steps (see 2.6.4 of IEC 61131-3).

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

3.9.5 SFC function blocks

The body of a function block can be defined by an SFC; this will be denoted as an *SFC function block*. The use of such function blocks follows the rules common to all function blocks. The user can declare several instances of the same function block type; each will have a private data area, which also contains information about the current state of the SFC.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

Each instance then can work independently, using the same SFC algorithm but with separate, hidden internal state variables.

A declaration of a simple SFC function block is shown in Figure 18. It will be noted that the SFC contained in an SFC function block is defined in the same (textual or graphical) way and under the same restrictions as for ordinary SFCs. In particular, each SFC network of an SFC function block must contain an initial step.

Instances of SFC function blocks can be invoked in the same manner as all other function blocks. Input variables that control the evolution of the SFC can be declared and used. For example, a Boolean transition condition for starting the evolution can be passed via input variables such as T11 in Figure 18. An action declared locally in the body of the SFC function block can also access these function block variables.

Execution of an instance of an SFC function block is invoked in the same way as for any function block, depending on the invocation mechanisms available in the language used for programming the instance of the function block. The body of the function block is then executed using an SFC scan algorithm such as the one outlined in 3.9.4.

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

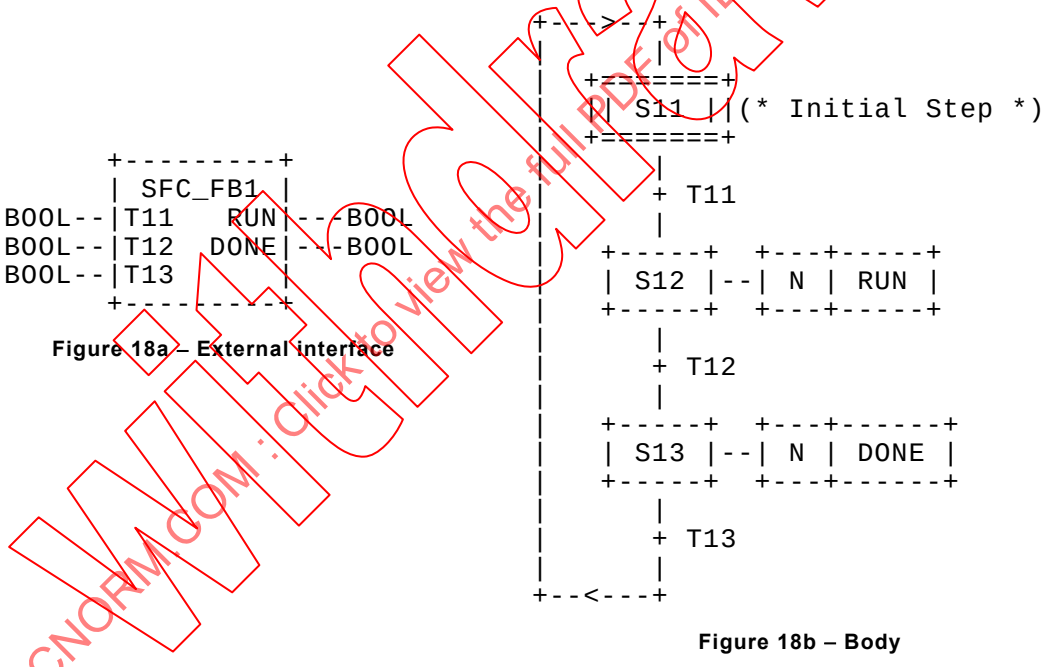


Figure 18a – External interface

Figure 18b – Body

Figure 18 – An SFC function block

IEC 2077/03

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

3.9.6 "Indicator" variables

As shown in 2.6.4.3 of IEC 61131-3, an "indicator" variable can be specified in an action block in order to indicate the correct or incorrect execution of an action and to pass on the information for further processing, for instance, as a transition enabling condition.

The main purpose of the representation of "indicator" variables in an action block is to improve the clarity and the documentation of the program.

The programming of an "indicator" variable is done in the content of an action by the user.

As the "indicator" variables are freely programmable, they can be used in a very flexible way. For instance, the variable may be associated with the first occurrence of a defined status. It

can also be associated with the fulfilment of a condition over a longer period of time or a time-delayed available status. This flexibility would be lost if the “indicator” variable’s functionality were fixed as part of the run-time library of a programmable controller.

3.10 Scheduling, concurrency, and synchronization mechanisms

3.10.1 Operating system issues

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

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

In this discussion, the term *task* refers to a set of *programs* or *function blocks* that runs independently of and (quasi-)parallel to other tasks.

The term TASK is used as a key word in IEC 61131-3 in order to define the temporal activation of tasks, i.e., a TASK statement specifies the run-time features of the associated tasks. The following short example is adapted from Figure 20 of IEC 61131-3:

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

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

Thus, program `P1` and function block `FB1` in program `P2` are associated with a task that is running every 20 ms with priority of 2, defined as `SLOW_1`. The faster task `FAST_1` only consists of function block `FB2`. Program `P2` belongs to a task that is always active when no other task is executable (see 2.7.2 of IEC 61131-3). Program `P4` is associated with task `INT_2`, triggered on the rising edge of variable `z2`, while function block `FB1` in `P4` is associated with the periodically executing task `PER_2`.

The strategy by which tasks are selected for execution determines the responsiveness of the system and the efficiency with which its processing capacity is utilized. Responsiveness and efficiency are usually improved when a running task can be interrupted by another one (“preemptive scheduling”). These aspects are addressed in 3.10.2.

Other issues in multi-processing are the control of exclusive access to operational equipment and the provision of data transfer between tasks. These aspects are addressed in 3.10.3 and 3.10.4, respectively.

As shown in Table 3, multi-user/multitasking systems and real-time responsiveness and efficiency are usually mutually exclusive.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

Table 3 – Differences between multi-user and real-time systems

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

3.10.2 Task scheduling

Subclause 2.7.2 of IEC 61131-3 introduces two methods to schedule the execution of tasks, called *preemptive* and *non-preemptive* scheduling. Both methods are employed in many real-time systems. Preemptive scheduling is mostly used by large programmable controllers and process computers, while non-preemptive scheduling can be found in smaller programmable controllers.

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

3.10.2.1 Performance effects

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

When a non-preemptive programmable controller switches from one task to another, there is a negligible amount of data in the processor that has to be saved. The context switch itself would be short, probably shorter than in the case of a preemptive context switch; but this

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

changeover from one task to another will never be done while a POU is still executing. Hence, with non-preemptive scheduling, the worst-case reaction time is at least as long as the time the longest running function block or program might need for a complete execution. As this not only holds for standard FBs but also for all user-defined FBs and programs, it may be difficult to determine a maximum limit for the reaction time of a non-preemptive scheduling system.

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

Usually, preemptive task switching causes a small performance degradation (as the context switches consume time) but improves the reaction time significantly. This can be seen by comparing the second column of Example 1 in Table 50 of IEC 61131-3 with the corresponding column in Example 2. Here it can be seen that with preemptive scheduling (Example 2), the high-priority function block FB2 in program P2 (P2.FB2@1) never has to wait, while with non-preemptive scheduling (Example 1) it has to wait for 4 ms in two out of three activations (at t = 10 ms and t = 20 ms).

3.10.2.2 Concurrency effects

Satisfying the rules for data concurrency given in item (7) of 2.7.2 of IEC 61131-3 may be substantially more complex in a programmable controller system with preemptive task scheduling than in a system with non-preemptive scheduling. In some configurations and programs, a preemptive system may not be able to guarantee that these rules are satisfied. The manufacturer of a programmable controller should provide sufficient information to enable a user to determine the means and extent to which these rules are satisfied.

3.10.3 Semaphores

NOTE 1 The semaphore (SEMA) function block has been deleted from IEC 61131-3, 2nd edition, due to the difficulty of addressing this construct in this standard. This subclause addresses the nature of these difficulties.

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

3.10.3.1 General

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

According to 3.3.2.4 of IEC 61131-3, “the WHILE and REPEAT statements shall not be used to achieve inter-process synchronization, for example, as a ‘wait loop’ with an externally determined termination condition. The SFC elements defined in 2.6 shall be used for this purpose”. This interdiction applies to semaphores, which are also an “inter-process synchronization” mechanism. Violation of this rule can lead to severe degradation in software reliability, portability and reusability; in the case of semaphores, it may lead to unanticipated and untraceable suspension of the execution of a program organization unit.

3.10.3.2 Deadlocks

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

Four conditions are necessary for a deadlock.

a) The deadlocked tasks have exclusive access to the corresponding operational equipment.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

b) The deadlocked tasks already have operational equipment assigned to them while they wait for further operational equipment.

c) It is not possible to remove the assignment of operational equipment to any task until the task releases it.

d) There is a closed chain of tasks, in which each task is assigned operational equipment which is required by the next task in the chain.

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

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

3.10.4 Messaging

Messaging (inter-process communication) describes methods for tasks to exchange data with each other and (possibly) synchronize their operations. In general there are three methods for such data transfer, namely, *global storage*, *mailboxes* and *queues*. Semaphores can also be used for communication; however, this can be regarded as a special case of mailboxes.

3.10.4.1 Global storage

Two tasks can exchange data via a storage area that is available for both. The protocol of the accesses has to be included explicitly in the program. Apart from this, there is great liberty for the implementation of a data transfer. This facility is implemented by the VAR_GLOBAL and VAR_EXTERNAL constructs defined in 2.4.3 and 2.7.1 of IEC 61131-3.

3.10.4.2 Mailboxes and queues

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

The characteristic of *queues* is that data elements are read in the same order as they were entered. An example is the character buffer of a terminal, which contains the characters in the same order than the user entered them.

Either mailboxes or queues, or both, may be used by the communication function blocks described in 3.11 of this technical report. However, the particular mechanism employed is completely invisible to the user, who need only be concerned with the externally specified interface and functionality of these function blocks.

3.10.5 Time stamping

The term “time stamping” comes originally from the field of data bases. With time stamping, each data element of the data base contains not just the value of the data, but also information on the time of input. If the data changes, the old element is not removed from the data base; instead, the old element is marked invalid and the new element is entered along with its time stamp. The advantage of this approach is that changes within the data base can be traced relatively easily by following the time stamps; also, new values that are in error can be deleted and older, correct values restored.

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

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

Time stamping of data is easily supported by including the value of the data in a variable of a structured data type that also includes its time stamp. For instance, a time-stamped data type that can contain REAL values could be declared according to the rules given in 2.3.3 of IEC 61131-3 as

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

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

NOTE The term *variable* is used throughout this clause in place of the term *parameter* used in IEC 61131-5, for the reasons given in Note 1 of 3.2 .

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

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

3.11.1 Communication channels

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

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

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

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

3.11.2 Reading and writing variables

A programmable controller can read from, and write to, variables within other remote programmable controllers supporting IEC 61131-5, providing they are either directly represented variables as defined in 2.4.1.1 of IEC 61131-3 or variables declared to be accessible using the VAR_ACCESS ... END_VAR construction defined in 2.7.1 of IEC 61131-3. This reading and writing are performed using the READ and WRITE FBs, respectively, as described below.

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

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

3.11.3 Communication function blocks

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

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

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

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

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

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

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

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

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

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

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

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

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

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

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

3.12 Deprecated programming practices

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

3.12.1 Global variables

Excessive use of global variables contradicts the principles of encapsulation and hiding discussed in 2.4.2.1 and can greatly reduce software reliability, maintainability and reusability. In particular, the writing of global variables from more than one program location should be avoided. It is recommended that global variables should be used (if at all) only for

– defining access paths for open communication;

– supplying values of “global” interest to other program organization units.

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

3.12.2 Jumps in FBD language

As noted in 2.3, the FBD language is modelled on connected hardware circuits (ICs), i.e., function block instances in networks are modelled as working in parallel. In such a context, jumps over or between networks can be confusing to the user who may be thinking in hardware terms, thus reducing software reliability, maintainability and adaptability. It is recommended that conditional execution of FBD networks should be controlled by placing such networks in SFC actions as described in 2.6.4 of IEC 61131-3, or by using the EN (enable) input and ENO output of functions as described in 2.5.1.2 of IEC 61131-3. This corresponds more closely to the concept of conditionally disabling or enabling portions of hardware circuits; proper operation can then be verified more easily than locating and checking the state of jump conditions.

3.12.3 Multiple invocations of function block instances in FBD

For the same reasons as given above, the body of a program organization unit written in the FBD language should not contain multiple copies of the same function block instance. Hardware-oriented users will not understand “circuit diagrams” containing several copies of the same “chip”, and it may be impossible to determine the sequence of execution of multiple copies of the same function block instance, thus reducing software reliability and maintainability.

3.12.4 Coupling of SFC networks

When multiple SFC networks exist in the body of a program or function block, they are typically interlocked together in some fashion. An example of this is given in program `AGV` shown in Clause F.8 of IEC 61131-3, where the interlocking is performed via the Boolean variable `CYCLE` and the step flags `READY.X` and `DONE.X`.

Unexpected effects can arise among coupled state machines of any kind, including SFCs. One of the most common is the *deadly embrace,* analogous to the *deadlock* condition described in 3.10.3.2 above, where each SFC is waiting for a transition clearing condition from another in a deadlocked chain. The following measures are recommended to enhance the avoidance and diagnosis of such conditions, thus increasing software reliability and maintainability.

a) Avoid the use of step flags in transition conditions to detect the completion of actions associated with other steps, especially if action qualifiers other than “N” (Non-stored) are used.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

b) Whenever possible, avoid the use of *SFC actions* as described in 3.9.4 above; this introduces coupling via the action control mechanism in addition to possible coupling through transition conditions.

c) Whenever possible, encapsulate individual SFC networks into *SFC function blocks* as described in 3.9.5 above, and use an FBD to express the coupling among the SFC function blocks. This explicit coupling greatly enhances the readability and hence the maintainability of the software, as well as providing the potential for reuse of the encapsulated SFCs via the function block instantiation mechanism.

d) Ensure that each SFC network is "reachable", i.e., a closed path exists from the initial step to any other step in the network and back to the initial step, and is "safe", i.e., uncontrolled generation of tokens is not possible (see 2.6.5 of IEC 61131-3 for further discussion of this point).

3.12.5 Dynamic modification of task priorities

It is well known from the theory of operating systems that the dynamic modification of task priorities may have unpredictable effects on the execution of parallel programs, including the generation of deadlocks. Since 2.7 of IEC 61131-3 TASKS are not necessarily mapped direct to operating system tasks, additional implementation dependent effects may occur. Therefore, it is highly recommended that the application programmer not use dynamic task priority modification in an IEC 61131-3 compliant system.

3.12.6 Execution control of function block instances by tasks

The association of tasks with function block instances and its effects on data concurrency are described in 2.7.2 of IEC 61131-3. The programmer should be aware of the fact that use of this feature may produce data consistency errors during program run time. The guidelines provided by the IEC 61131-3 implementor should be consulted to determine the mechanisms provided to assure data consistency. Since these mechanisms are implementation-dependent, programs using this feature may not be portable between different IEC 61131-3 compliant systems.

3.12.7 Incorrect use of WHILE and REPEAT constructs

According to 3.3.2.4 of IEC 61131-3, "the WHILE and REPEAT statements shall not be used to achieve interprocess synchronization, for example, as a 'wait loop' with an externally determined termination condition. The SFC elements defined in 2.6 shall be used for this purpose". In order to avoid unanticipated and potentially dangerous effects, users are strongly cautioned to extend this prohibition to any case of inter-code synchronization. For instance, suppose it is desired to perform a scaling calculation at the rising edge of a SAMPLE input of the function block shown in Figure 19a) and provide a rising edge at a DONE output when the calculation is complete. Then the programming construct shown in Figure 19b) should not be used. To avoid this usage, the SAMPLE input can be changed to a rising-edge trigger as shown in Figure 19c) and the function block body shown in Figure 19d) can be used, or the SFC shown in Figure 19e) can be used with the interface shown in Figure 19a).

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

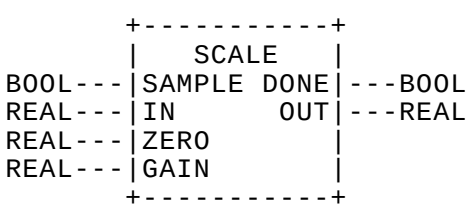


Figure 19a – Example interface

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

Figure 19 – Incorrect construct

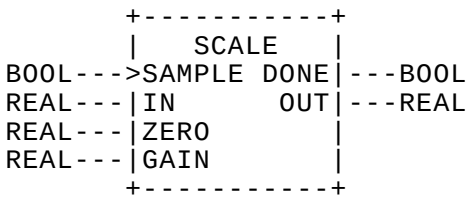


Figure 19c – Alternative interface

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

Figure 19d – FB body for interface c)

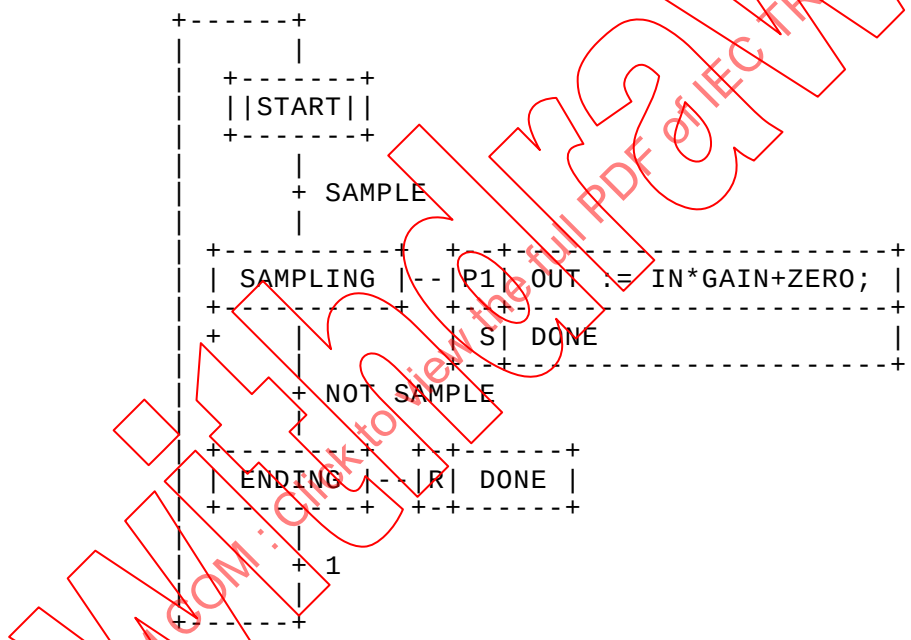


Figure 19e – Allowed SFC function block body for interface a)

Figure 19 – Example of incorrect and allowed programming constructs

IEC 2078/03

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

3.13 Use of TRUNC and REAL_TO_INT functions

Footnote (a) of Table 22 of IEC 61131-3 states that “Conversion from type `REAL` or `LREAL` to `SINT`, `INT`, `DINT` or `LINT` shall round according to the convention of IEC 60559, according to which, if the two nearest integers are equally near, the result shall be the nearest even integer....” It should be noted that the rounding procedure specified by IEC 60559 may not operate in the same manner as the rounding function defined in other programming languages. The examples given in Table 22 of IEC 61131-3 should be consulted for clarification.

4 Implementation guidelines

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

4.1 Resource allocation

The resource, as described in 1.4 and 2.7.1 of IEC 61131, is the basic unit of IEC 61131-3 for storage of data and program code, the scheduling of code for execution, and the execution of the appropriate code when a POU, i.e., a *program*, *function* or *function block*, is invoked.

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

4.2 Implementation of data types

4.2.1 REAL and LREAL data types

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

There is a wide range of microprocessors, particularly digital signal processors (DSPs), that have their own internal floating-point formats. In such cases, the implementor must limit the range of values supported by the REAL and/or LREAL implementation to those specified in 2.3.1 of IEC 61131-3 or must specify new implementation-dependent data types, say DSP_REAL and DSP_LREAL for the native floating-point formats in addition to supporting the standard REAL and LREAL types, in order to meet the compliance requirements of items a) and h) of 1.5.1 of IEC 61131-3.

4.2.2 Bit strings

Implementation of bit-string comparison operations should correspond to the explanation given in 3.1.8.

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

- Let $\mathbf{M}$ be a set of bit-string data $\{b_1, b_2, b_3, \ldots b_n\}$.
- $(B = MAX(b_1, b_2, b_3, \ldots b_n)) \equiv ((B \in \mathbf{M}) \ \& \ \forall\ 1 \le j \le n,\ b_j \in \mathbf{M}\text{:}\ (B \ge b_j))$; that is, saying that B is the maximum value of the set is equivalent to saying that it is a member of the set and it is greater than, or equal to, the value of any member of the set.
- $(B = MIN(b_1, b_2, b_3, \ldots b_n)) \equiv ((B \in \mathbf{M}) \ \& \ \forall\ 1 \le j \le n,\ b_j \in \mathbf{M}\text{:}\ (B \le b_j))$; that is, saying that B is the minimum value of the set is equivalent to saying that it is a member of the set and it is less than, or equal to, the value of any member of the set.

See the examples given in 3.1.8.

4.2.3 Character strings

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

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003 Withdrawn

There are two main techniques used for defining the character string length: 1) a null character terminator is used, or 2) the length is stored at the head of the string. The null character technique, as used in the 'C' language, may not be convenient if there is also a requirement to have null characters embedded within the string but does have the advantage that indefinitely long strings can be stored. Storing the length in a byte or word limits the maximum string length to 255 or 65535 respectively, but does allow strings to hold null characters.

NOTE See also the discussion of character string data types in 3.1.5 of this technical report.

4.2.4 Time data types

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

For example, the TIME (duration) data type might be represented by a 32-bit unsigned double integer storing the duration as a count of milliseconds. This allows a TIME data type to define accurately any duration from 1 ms to 49 days. However, this would not allow the computation of small time differences (less than 1 ms) between events, or to manipulate time differences that might be negative.

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

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

4.2.5 Multi-element variables

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

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

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

4.3 Execution of functions and function blocks

This clause provides general guidelines for the execution of functions and function blocks. In addition, when the bodies of user-defined functions or function blocks are programmed in one of the graphical languages (LD or FBD) defined in Clause 4 of IEC 61131-3, the implementation should assure that execution obeys the rules for evaluation of networks in 4.1.3, 4.2.6 and 4.3.3 of IEC 61131-3 as well as the rules for evaluation of LD elements given in 4.2.2 through 4.2.5 of IEC 61131-3.

4.3.1 Functions

A *function* is defined as a program organization unit which, when executed, yields exactly one data element (which may be multi-valued). A function's internal data is dynamically initialized

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

for each activation, i.e., invocation of a function with the same arguments (input variables) shall always yield the same value (output).

NOTE 1 Certain standard functions defined in 2.5.1.5 of IEC 61131-3 are allowed to be *extensible,* i.e., to have a variable number of inputs.

A function *invocation* establishes a list of actual variables corresponding to the list of formal variables specified in the function's type declaration and causes execution of the program code corresponding to the function body. Depending on the implementation, the list of variables may consist of the actual variable values, addresses at which to locate the actual variable values, or a combination of the two, and may be passed to the executing code on a stack or by some other means. Typically, the result of function execution will also be returned on the stack.

Function execution may be made conditional on the EN input described in 2.5.1.2 of IEC 61131-3. By rules (1) and (2) of 2.5.1.2 of IEC 61131-3, the initial system action upon invocation of a function is to copy the EN input to the ENO output. This variable acts as a "power flow" through the function.

If an error occurs during the processing of a function, the minimum required system action is that the ENO output of the function be reset to FALSE. Depending on the implementation, an error condition may also trigger the execution of a system- or user-defined *error task* at the end of function execution. The user may associate a special error-processing program with this task.

NOTE 2 The effect of ENO at the end of function execution should be the same for all languages in a given implementation.

NOTE 3 The reading of the EN input within the body of a function will (in effect) always deliver the Boolean value TRUE, since the function will not be evaluated when EN is FALSE.

4.3.2 Function blocks

A *function block* is defined in IEC 61131-3 as a program organization unit which, when executed, yields one or more output values; moreover, a function block can have multiple instances, each with its own private data. A function block's internal data persists from one execution to the next; therefore, successive invocations of a function block may yield different results, even with the same arguments (input variables).

A number of standard function blocks are defined in IEC 61131-3. Typically, at least counter and timer function blocks are implemented in the controller firmware.

A function block *invocation* establishes values for the function block's input variables and causes execution of the program code corresponding to the function block body. These values may be established graphically by connecting variables or the outputs of other functions or function blocks to the corresponding inputs, or textually by listing the value assignments to input variables. If no value is established for a variable in the function block invocation, a default value is used. Depending on the implementation, input variables may consist of the actual variable values, addresses at which to locate the actual variable values, or a combination of the two. These values are always passed to the executing code in the data structure associated with the function block instance. The results of function block execution are also returned in this data structure. Hence, if the function block invocation is implemented as a procedure call, only a single argument – the address of the instance data structure – need be passed to the procedure for execution.

NOTE 1 When a function block instance in a program is associated with a separate task (features 3b and 4b of IEC 61131-3, Table 50), invocation of the function block from the program should establish values for the function block's input variables, but should not cause execution of the program code associated with the function block body. Execution of this code should be under the exclusive control of the associated task as required by Rule (5) of 2.7.2 of IEC 61131-3.

NOTE 2 When a function block instance is used to interface to high-speed hardware, such as counters or flash A/D converters, or when the function block instance is executed preemptively by a high-speed periodic or interrupt-driven task, the actual output values of the function block may change while computations involving those outputs

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

are proceeding in the program containing the function block instance. In such cases, the implementation must provide means of effectively “freezing” such outputs while such computations are taking place, for example, by providing a temporary buffer for the actual output value.

Additional data-type qualifiers R_EDGE and F_EDGE are available for function block input variables only. These data types provide rising (0-->1) or falling (1-->0) edge detection, respectively, of Boolean inputs, upon invocation of the function block. The variable condition is only TRUE when the specified edge is detected and is FALSE otherwise.

NOTE 3 See 2.5.2.2 of IEC 61131-3 for additional descriptions of this feature.

4.4 Implementation of SFCs

4.4.1 General considerations

A number of points relevant to the implementation of SFCs have already been discussed in previous portions of this technical report. These points are recapitulated below:

a) Action control may be implemented either as an action control block or its functional equivalent in optimized code and data structures; see 3.9.1 above and 2.6.4.5 of IEC 61131-3.

b) A consistent method, such as the four-step algorithm described in 3.9.4 above, should be utilized for execution of SFC evolution, whether the SFC occurs in programs, FBs, or SFC actions.

c) It is recommended that the “indicator” variables described in 2.6.4.3 of IEC 61131-3 be supported as a notational convenience, and not with any specific implementation-dependent functionality, in order to maximize flexibility and portability in the use of this feature, as discussed in 3.9.6 above.

d) It may be useful for an implementation to enforce the SFC programming disciplines described in 3.12.4 above.

e) As noted in 4.1 above, when allocating storage for a *program* or *function block* containing SFCs, it is necessary to consider the requirements for storage of SFC state information as well as for the variables used by the program or function block. Such SFC state information includes the step flags and action control block states (if any), step elapsed times (if supported), and other state information as required by the implementation.

4.4.2 SFC evolution

There is no defined termination to the evolution of an SFC. This is different from programs using languages that can be scanned in a defined order from start to end. Thus, a definition is required for the meaning of one “scan” of an SFC, whether this SFC constitutes the “top level” of a program organization unit or an SFC action. One possible algorithm for scanning an SFC is

a) determine the currently active set of steps. This is the set of initial steps for the first scan following system initialization. Otherwise, this set is determined by deactivating the steps preceding and activating the steps following the current list of transitions to be cleared;

b) determine the state of all action control block `Q` outputs and perform the final scan of any actions associated with a falling edge `Q`;

c) scan all actions with action control block `Q` values of Boolean `1`;

d) determine the transitions to be cleared (if any) on the next scan.

This definition of scans recognizes the fact that there is no implicit looping to the initial step whenever a final step has been reached. All loops have to be programmed explicitly, for example, the loop from ENDING to START in Figure 19e). In this way, there is no chance to lose the token of an SFC network as all transitions have to be succeeded by a step.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

4.5 Task scheduling

The TASK construct is defined in 2.7 of IEC 61131-3 to enable the user to specify the requirements for scheduling the execution of programs and FBs, without having to develop by hand a detailed cyclic executive. These requirements can be combined with modern scheduling techniques for real-time systems to determine in advance whether the system can be scheduled to meet the expressed user requirements, especially in systems where preemptive scheduling is supported (see 3.10.2 above for a discussion of preemptive versus non-preemptive scheduling). Implementors should be aware of these techniques and consider their implementation and support in IEC 61131-3 compliant systems.

4.5.1 Classification of tasks

When a task is triggered, it schedules the execution of the associated programs and FBs. Hence, IEC 61131-3 tasks can be characterized by their triggering mechanisms.

– A periodic task is triggered regularly at a determined time interval. This is configured by the user by connecting the SINGLE input of the task block to Boolean FALSE (or just leaving it disconnected), and setting the INTERVAL input to a non-zero value of TIME type, representing the periodic triggering interval.

– An aperiodic task is triggered by an external or internal event that does not necessarily occur at a regular interval. This is configured by the user by connecting the SINGLE input of the task block to a Boolean variable whose rising edge represents the triggering event, and setting the INTERVAL input to `t#0s` (or just leaving it disconnected).

– The default task is automatically associated with programs that have no explicit task association and schedules its associated programs in round-robin fashion at the lowest system priority. This task may also handle the scanning of inputs and outputs as discussed in 2.1 above.

Aperiodic tasks can be classified according to the nature of the triggering event. Such events may include

– “cold restart” or “warm restart” as discussed in 2.4.2 of IEC 61131-3;

– run-time error conditions as discussed in item d) 4) of 1.5.1 of IEC 61131-3 and listed in Annex E of IEC 61131-3;

– events detected or generated by implementation-dependent hardware or software mechanisms (sometimes called “interrupt events”), such as the rising edge of an electrical signal or the terminal count of a high-speed hardware counter.

Implementors should consider specifying Boolean variable names or directly represented Boolean variables (as described in 2.4.1.1 of IEC 61131-3) whose rising edges represent events such as those described above for a particular *resource* type or instance as defined in 2.7.1 of IEC 61131-3. This will facilitate the association of these events with tasks which can schedule programs to respond to the event occurrence. Implementors should also consider providing default programs to process the more frequently occurring types of events such as restarts and run-time errors.

4.5.2 Task priorities

The interaction between the assignment of task priorities, the scheduling intervals of periodic tasks, the arrival rates of aperiodic events, and task execution times can have profound effects on the responsiveness and schedulability of real-time systems. Implementors of IEC 61131-3 compliant systems should provide adequate scheduling tools and facilities for the tasking features supported by the implementation.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

4.6 Error handling

4.6.1 Error-handling mechanisms

Items c) and d) of 1.5.1 of IEC 61131-3 specify the treatment of errors in compliant systems. Item c) typically applies to syntax or configuration errors in the source program. These errors may be detected at the time they are entered into the PSE by the user, during the parsing of the program in the compilation phase (if any), during the linking of program organization units into a configuration, or during the loading of the configured software into the controller for execution.

Item d) of 1.5.1 of IEC 61131-3 refers to the errors listed in Annex E of IEC 61131-3, which are for the most part errors that may occur during execution of the user program, i.e., "run-time errors". Item d) lists four possibilities for dealing with these errors:

" 1) there shall be a statement in an accompanying document that the error is not reported;

2) the system shall report during preparation of the program for execution that an occurrence of that error is possible;

3) the system shall report the error during preparation of the program for execution;

4) the system shall report the error during execution of the program and initiate appropriate system- or user-defined error handling procedures."

NOTE 1 The use of option 1) is not recommended for run-time errors.

NOTE 2 Option 3) is typically mutually exclusive with options 2) and 4). However, options 2) and 4) are not mutually exclusive and should be used in combination whenever possible. That is, if the error cannot be detected before run time per option 3), the user should be warned that the error may occur and the error should also be detected at run time.

NOTE 3 Items (f) and (g) of 1.5.1 of IEC 61131-3 require that extensions and implementation-dependent features be treated by the system hardware and/or software in the same manner as errors as listed above. However, the implementor may supply a software switch by means of which the user can disable such processing.

Table 4 recommends the error-handling mechanisms that should be applied to the error conditions listed in Table E.1 of IEC 61131-3.

Table 4 – Recommended run-time error-handling mechanisms

Subclause	Error conditions (Notes 1 and 2)	Mechanisms (Note 3)
2.3.3.1	Value of a variable exceeds the specified subrange	RT
2.4.2	Length of initialization list does not match number of array entries	ED
2.4.3	Attempt by a program organization unit to modify a variable which has been declared CONSTANT	ED
2.5.1.5.1	Type conversion errors	ED
2.5.1.5.2	Numerical result exceeds range for data type Division by zero	RT ED, RT
2.5.1.5.3	N input is less than zero in a bit-shift function	EW,RT
2.5.1.5.4	Mixed input data types to a selection function Selector (K) out of range for MUX function	ED RT
2.5.1.5.5	Invalid character position specified Result exceeds maximum string length ANY_INT input is less than zero in a string function	EW, RT
2.5.1.5.6	Result exceeds range for data type	RT
2.5.2.2	No value specified for a function block instance used as input variable	ED
2.5.2.2	No value specified for a VAR_IN_OUT variable	ED
2.6.2	Zero or more than one initial steps in SFC network User program attempts to modify step state or time	ED
2.6.3	Side-effects in evaluation of transition condition	ED

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

Table 4 – Recommended run-time error-handling mechanisms

Subclause	Error conditions (Notes 1 and 2)	Mechanisms (Note 3)
2.6.4.5	Action control contention error	EW, RT
2.6.5	Simultaneously true, non-prioritized transitions in selection divergence Unsafe or unreachable SFC	EW,RT ED
2.7.1	Data type conflict in VAR_ACCESS	ED
2.7.2	Tasks require too many processor resources Execution deadline not met Other task scheduling conflicts	ED, RT ED, RT EW, RT
3.2.2	Numerical result exceeds range for data type	EW, RT
3.3.1	Division by zero Numerical result exceeds range for data type Invalid data type for operation	ED, EW, RT EW, RT ED
3.3.2.1	Return from function without value assigned	ED
3.3.2.4	Iteration fails to terminate	ED, EW, RT
4.1.1	Same identifier used as connector label and element name	ED
4.1.3	Uninitialized feedback variable	ED

NOTE 1 This table does not include all entries from Table E.1 of IEC 61131-3, but only those entries that may be identified as run-time errors.

NOTE 2 This table is not an exhaustive listing of all possible run-time errors. Implementors may extend this table and the corresponding error-handling facilities.

NOTE 3 ED = Early detection per item d) 3) of 1.5.1 of IEC 61131-3;

EW = Early warning per item d) 2) of 1.5.1 of IEC 61131-3;

RT = Run-time detection per item d) 4) of 1.5.1 of IEC 61131-3.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

4.6.2 Run-time error-handling procedures

This subclause contains recommendations for the implementation of rule (d)(4) in subclause 1.5.1 of IEC 61131-3:2003, "the system shall report the error during execution of the program and initiate system- or user-defined error-handling procedures...."

NOTE The scope of this provision in IEC 61131-3 is limited to errors in user programs; however, implementors may consider providing similar procedures for the handling of errors from other sources such as I/O or communication subsystems.

4.6.2.1 Reporting of errors

The information to be reported upon occurrence of an error should include

– notification of the fact that an error has occurred;

– classification of the type of error (for example, "division by zero");

– identification of the source of the error (for example, a program organization unit).

In order to provide a uniform style of error reporting, implementors should consider the encoding of this information into variables of a single type such as the following.

```
TYPE ERROR_REPORT:
 STRUCT FLAG: BOOL;
     CLASS: STRING(...);
     SOURCE: STRING(...);
 END_STRUCT;
END_TYPE
```

NOTE 1 The length of the STRING elements above will be implementation-dependent.

NOTE 2 The new WSTRING data type defined in IEC 61131-3, 2nd edition, should be considered for the reporting of errors in various national languages.

NOTE 3 The use of integer or enumerated types for the error class element may be considered if higher efficiency is required in subsequent error processing.

Such error-reporting variables would typically be made available as implementation-specific default declarations, for example, as global variable declarations for a particular resource type, or as outputs of programs or tasks. For instance, a set of default declarations for a resource might be

```
VAR_GLOBAL
  MATH_ERROR: ERROR_REPORT; (* Table 23 *)
  ARITHMETIC_ERROR: ERROR_REPORT; (* Table 24 *)
  SFC_ERROR: ERROR_REPORT; (* 2.6.4.5(4) *)
  ...
END_VAR
```

NOTE 1 In this example, the comments refer to the locations in IEC 61131-3 where the features that may give rise to the particular error are described.

NOTE 2 A single error-reporting variable may be used; however, higher performance of error-handling procedures may be achieved by using a larger number of error-reporting variables to provide higher resolution of error classification.

4.6.2.2 System-defined error-handling procedures

The handling of run-time errors generally consists of the following steps.

a) The normal flow of program execution is suspended.

b) Action appropriate to the error type is taken, for instance,

   – the error may be corrected if possible;

   – if correction is not possible, default values may be substituted for the erroneous variables;

   – the occurrence of the error and any corrective actions taken may be reported to an operator or logged to a file for future reference.

c) Program execution is resumed at an appropriate point. Depending on the error type and the possibility of corrective action, such resumption may be immediate or contingent upon a system event such as “warm restart”, “cold restart”, or a command from the communication network or from an operator.

The implementor should specify the corrective and reporting actions taken by the system, and the procedure for program resumption, for each type of run-time error processed by the system. If the reporting of errors is modelled by global variables as discussed in the preceding subclause, the handling of errors could be modelled by preemptive scheduling of error processing tasks as defined in 2.7.2 of IEC 61131-3. For the example given in the preceding subclause, the resource-specific specification of the error processing tasks could have the form:

```
TASK MATH_ERROR_TASK
     (SINGLE := MATH_ERROR.FLAG, PRIORITY :=...);
TASK ARITHMETIC_ERROR_TASK
     (SINGLE := ARITHMETIC_ERROR.FLAG, PRIORITY :=...);
TASK SFC_ERROR_TASK
     (SINGLE := SFC_ERROR.FLAG, PRIORITY :=...);
...
```

NOTE 1 The implementor should specify the priority levels at which various errors are to be processed. These priorities will usually be higher than those of user program tasks in order to assure that user programs will be interrupted for error processing.

NOTE 2 The occurrence and processing of errors may have severe impacts on the ability of the system to meet its task scheduling deadlines.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003

With the above model, error correction and reporting procedures could be specified by associating the error processing tasks with appropriate system-defined error processing programs. In the above example, the declarations of such associations could be

```
PROGRAM PROCESS_MATH_ERROR WITH MATH_ERROR_TASK:
    SYSTEM_MATH_ERROR_PROCEDURE;
PROGRAM PROCESS_ARITHMETIC_ERROR WITH ARITHMETIC_ERROR_TASK:
    SYSTEM_ARITHMETIC_ERROR_PROCEDURE;
PROGRAM PROCESS_SFC_ERROR WITH SFC_ERROR_TASK:
    SYSTEM_SFC_ERROR_PROCEDURE;
...
```

The implementor could then specify the actions to be taken by the specified error-handling program for each possible error classification in the associated global `ERROR_REPORT` variable, for example, the actions to be taken by the `SYSTEM_MATH_ERROR_PROCEDURE` program for each of the possible values of `MATH_ERROR.CLASS`, etc.

4.6.2.3 User-defined error-handling procedures

An IEC 61131-3 implementation may provide for user-defined error-handling procedures. In the examples given in the preceding subclause, this could be accomplished by the substitution or augmentation of system-defined error handling programs by user-defined programs associated with the appropriate error handling tasks.

If user-defined error handling procedures are supported, the implementor should provide facilities for such procedures to specify the same types of error handling and reporting mechanisms, as well as user program resumption options, as employed by the system-defined procedures. Such facilities could take the form, for instance, of global variables that could be set, or function blocks that could be invoked by the user-defined error handling procedure.

4.7 System interface

Implementors should consider the provision of global variables (which may include function block instances) within resources for the purpose of interface to system functions. For instance, as described in 4.5.1 above, global Boolean variables may be used to represent system-specified status or events, for example, `BATTERY_LOW` or `POWER_ABOUT_ TO_FAIL`. Alternatively, system entities may be represented as instances of system-specific function blocks, for example, `BATTERY` or `POWER_SUPPLY`, with defined input and output variables representing their status or control interfaces.

4.8 Compliance

IEC 61131-3 contains numerous requirements in addition to the general compliance requirements enumerated in 1.5.1 of IEC 61131-3. Implementors should pay attention to any occurrence of the word “shall” in IEC 61131-3, since each such occurrence indicates a requirement.

The following subclauses deal with a number of the more important provisions that implementors should bear in mind in the development of IEC 61131-3 compliant systems.

4.8.1 Compliance statement

The first requirement enumerated in 1.5.1 of IEC 61131-3 is that a compliance statement be included in the documentation or produced by the system itself, for example, as a readable and printable file included with the system. This consists of a statement of compliance and a series of tables enumerating the features supported by the system; the exact form of these tables is prescribed in 1.5.1 of IEC 61131-3.

NOTE The compliance statement is not the only documentation requirement imposed by IEC 61131-3; see, for instance, items b), d)1), e) and i) of 1.5.1 of IEC 61131-3.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

This format for statement of compliance was considered to be more practical than the enumeration of compliance classes, given the wide range of application of programmable controllers. Accumulation of experience in the use of the elements of IEC 61131-3 may make it possible for compliance classes to be defined in future revisions.

4.8.2 Controller instruction sets

Subclause 1.1 of IEC 61131-3 specifically limits its scope to “the printed and displayed representation...of the programming languages to be used for programmable controllers....”. In particular, it is not required that the IL language defined in 3.2 of IEC 61131-3 be considered an instruction set for any real or virtual machine. Rather, the IL language is considered a way to express most of the functionality available in the other IEC 61131-3 languages in an assembly-language format familiar to a large number of users of existing systems.

In principle, the IEC 61131-3 language elements can be compiled to a large number of machine instruction sets. PSEs must be capable of presenting the programming of the programmable controller system to the user in the formats prescribed by IEC 61131-3 and should hide the details of machine instruction sets from the user. This will allow the evolution of programmable controller architectures while protecting the user's investment in software and training.

It is a natural consequence of this flexibility in instruction sets that compliant programs, as defined in 1.5.2 of IEC 61131-3, will only be portable at the source code level.

4.8.3 Compliance testing

The development and execution of compliance test suites for the IEC 61131-3 languages will depend on the accumulation of practical experience in their application; hence, this is a topic for future standardization.

5 PSE requirements

5.1 User interface

The user interface is the principal means of making the features and functionality of a programmable control system visible to the user, and is often the primary basis for the user's opinion of the system. Therefore, the development of the user interface is a major issue for creators of programmable controller software and hardware.

There will obviously be different means of implementation and design of user functions even though the same programming language standard is used. However, an understanding of how the language elements in the standard are intended to be used should lead to more satisfactory PSE design and implementation.

For any software product, it is critical to know which information is useful, when it is useful, and how it must be displayed. The design of the PSE must respect fundamental ergonomic principles, for instance:

– the more complex the PSE is, the more the user has to be guided and informed about its use;

– the PSE should guide the user through the steps of a systematic software development methodology. However, it should be possible for the user to perform these steps in a user-determined order;

– the PSE can guide the user with appropriate cues on a screen and window organization and with a set of dialogues appropriate to the step of the software methodology currently being performed;

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003

– when available, contextual on-line help should be given in response to user requests. Help information that is presented when not requested will typically not be read;

– recent studies show that the user does not want time-consuming and exhaustive training. On-line tutorial material should be available that will enable the user to perform the most commonly used functions within a short learning period, and to learn advanced methods as necessary in the normal course of productive work;

– the use of paper documents is uneconomical for both the PSE vendor and user; therefore, they should only be used for reference material as a supplement to on-line documentation.

When a multi-windowing user interface system is used, the ergonomic guidelines of the supplier of the windowing system should be followed in order to assure high user productivity and low error rates in the shortest possible training time. Such guidelines typically include

– correspondences between pointing device (mouse, etc.) and keyboard operations;

– conventions for selecting and deselecting objects;

– rules for using special windows such as message boxes, list boxes, dialogue boxes, etc.;

– the order of items in menus.

5.2 Programming of programs, functions and function blocks

The user must perform a similar set of tasks in programming any POU, whether it be a function, function block, or program. Programming is the structuring of POUs, using either a top-down or a bottom-up approach, or both. As shown in Figure 20, a POU contains declarations of variables and a body programmed in any language. The means of declaring variables can be independent of the language in which they are used. IEC 61131-3 standardizes the textual syntax of data declarations but does not require that it be presented or entered by the user in this form while programming.

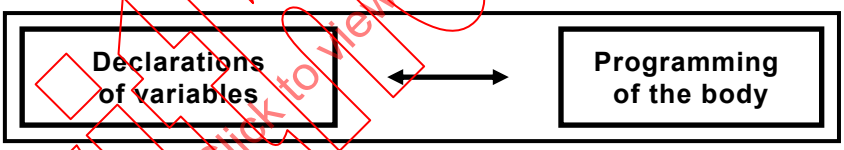


Figure 20 – Essential phases of POU creation

Serious consideration should be given to the provision of features, such as syntax-directed editors, for the immediate detection and (possibly) correction of syntax errors during the declaration and programming of POUs. Such features improve the productivity and quality of programming in any language. However, this advantage will be especially pronounced in PSEs for the IEC 61131-3 languages for the following reasons.

– Each POU can be programmed in a different textual or graphic language. Immediate detection and correction of errors can assist the user in learning the syntax of a less familiar language.

– When a POU is programmed in a graphical language, it is easier to make errors such as connection of variables of incompatible types. Immediate detection and correction of such invalid connections can prevent the generation of a large number of confusing error messages at compile time.

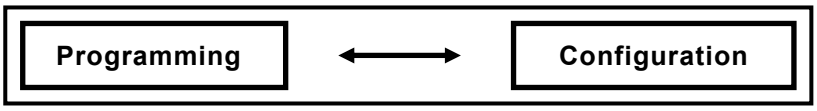
– The tasks of declaration, editing and compilation of a POU may be performed at different times, and even by different users. It is thus imperative that syntactic errors be detected and prevented as early in the programming activity as possible.

It should be possible for the user to turn the automatic detection of syntax errors on and off as desired.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

5.3 Application design and configuration

As illustrated in Figure 21, IEC 61131-3 separates the *configuration* of an application from its *programming*. Therefore, the PSE must also distinctly separate these two phases without necessarily ordering them. That is, it must be possible to program up to a certain level before making a configuration.



IEC 2080/03

Figure 21 – Essential phases of application creation

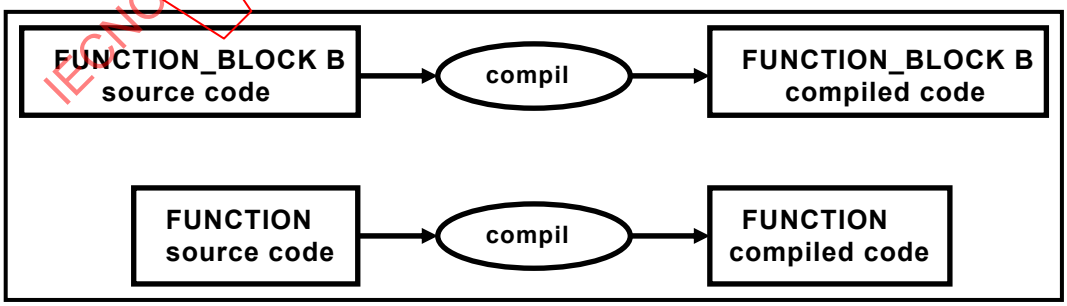
The PSE must assist the user in performing system configuration as described in IEC 61131-3. This includes such tasks as the description of resources, declaration of global variables, assigning programs to be called by tasks, managing program libraries, defining communications between the application and external entities, etc. For instance, to assist the user in the configuration of access paths as described in 2.7 of IEC 61131-3, a graphical editor could be provided to enable the user to specify access paths in the format of Figure 19 of IEC 61131-3. Such an editor could automatically generate the required VAR_ACCESS...END_VAR statements with syntactically and semantically correct type declarations.

5.4 Separate compilation

The separate compilation of POUs can be analysed in terms of dependency. A POU that declares a variable becomes dependent on the type declaration of the variable. To simplify the explanations and figures given below, dependencies on data types are assumed but not explicitly described in each case.

The simplest case to be considered, as illustrated in Figure 22, consists of a POU that is not dependent on other POUs. For instance, a function that does not call any other function can be compiled alone. Similarly, a function block whose declared variables are not instances of function blocks, and whose body does not contain a call to any other function, can be compiled alone. Compilation can proceed directly from a textual or graphical source of the POU. In this case, the benefits of separate compilation are obvious.

– Scanning, parsing, semantic analysis, and independent testing are sufficient to verify and validate an independent POU.
– Re-use of POUs is simplified if the relocatable generated code can be associated with the source.



IEC 2081/03

Figure 22 – Separate compilation of functions and function blocks

The separate compilation of function blocks or of programs that use either VAR_EXTERNAL or directly represented variables inside their body is similarly straightforward, as shown in Figure 23. It is possible to compile these POUs and to resolve the unknown links upon configuration of the application.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

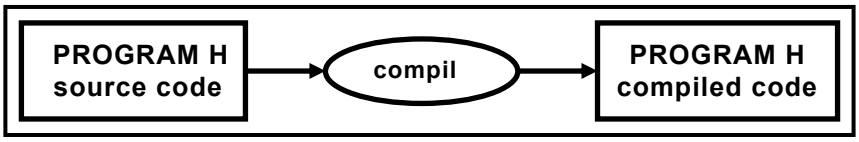


Figure 23 – Compiling a program accessing external or directly represented variables

In this first level of separate compilation, the compiler works from a source that has no links with other POUs. In more complex cases, a POU may invoke another function, or an FB or program may contain the declaration of an instance of another FB. These cases require the introduction of the concept of *interfaces.*

5.5 Separation of interface and body

5.5.1 Invocation of a function from a programming unit

To program the invocation of function B in the example shown in Figure 24a), it is necessary to know the name and the type of this function, and also the named and typed list of its input variables (VAR_INPUT). If also present, the output variables (VAR_OUTPUT) or the in-out variables (VAR_IN_OUT) of the function have to be known. This set of information constitutes the function's *external interface*. Function B in the example has a single input variable only. To compile function A, it is sufficient to know the external interface of function B, as shown in Figure 24b), even if the source or compiled code of function B is not known.

```
FUNCTION A: REAL
 VAR_INPUT C,D: REAL; (* External interface *)
 A:= B(C) + D;
 END_FUNCTION
```

Figure 24a – Declaration of example function

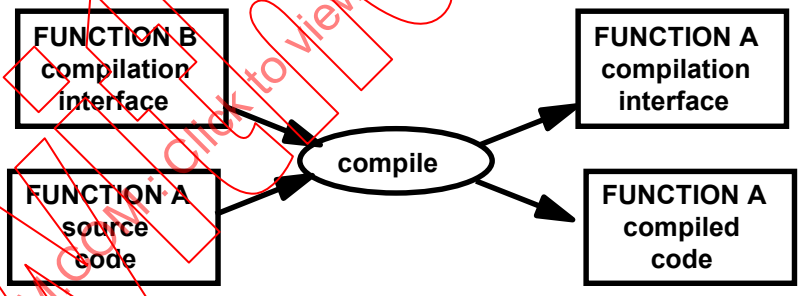


Figure 24b – Compilation of example function

IEC 2083/03

Figure 24 – Compiling a function that invokes another function

This example illustrates the importance of a clear separation of the POU interface from its body. However, if the use of separate compilation is too difficult, the user (who is not necessarily a computer scientist) could reject it. Therefore, the PSE should assist the user by constructing the interface automatically to the maximum extent possible.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

5.5.2 Declaration and invocation of a function block instance

To program the invocation of function block instances, it is necessary to know the name of this function block and the names and the types of its VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT variables. This set of information constitutes the function block's *external interface*. For compilation, the total memory size required for each instance must be determined. This is calculated from the function block's external interface and from its local (VAR) and external (VAR_EXTERNAL) declarations. This set of information constitutes the *compilation interface*. In the following discussion, the term "interface" will refer to the compilation interface.

To compile the program F illustrated in Figure 25a), the interface of the function blocks FB1 and FB2 must be known in order to allocate memory for each of their declared instances in program F. In this example, there are two instances of FB1 and one instance of FB2. However, as shown in Figure 25b), the bodies of these function blocks are not required in order to compile program F.

Compared to the compilation *interface* for a function, the compilation *interface* for a *function block* additionally consists of the names and the types of the declarations of its internal and external variables (VAR and VAR_EXTERNAL).

```
PROGRAM F
...
(* local declarations *)
VAR X1: FB1;
    X2: FB1;
    X3: FB2;
    ...
```

Figure 25a – Declaration of example program

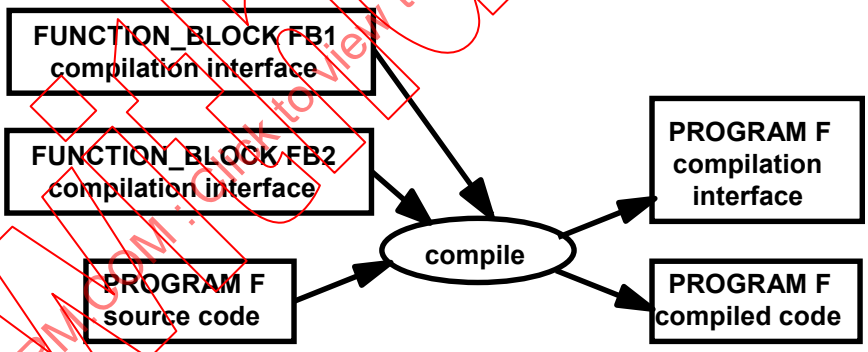


Figure 25b – Compilation of example program

IEC 2084/03

Figure 25 – Compiling a program containing local instances of function blocks

Figure 26 provides an example of the application of the principles developed above. In this example, it is assumed that

– the functions F1 and F2 do not invoke any other function;
– function block FB1 invokes functions F1 and F2;
– program G invokes function F1 and uses an instance of the function block FB1;
– program G does not use any global or directly represented variables, and does not contain a VAR_ACCESS declaration.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

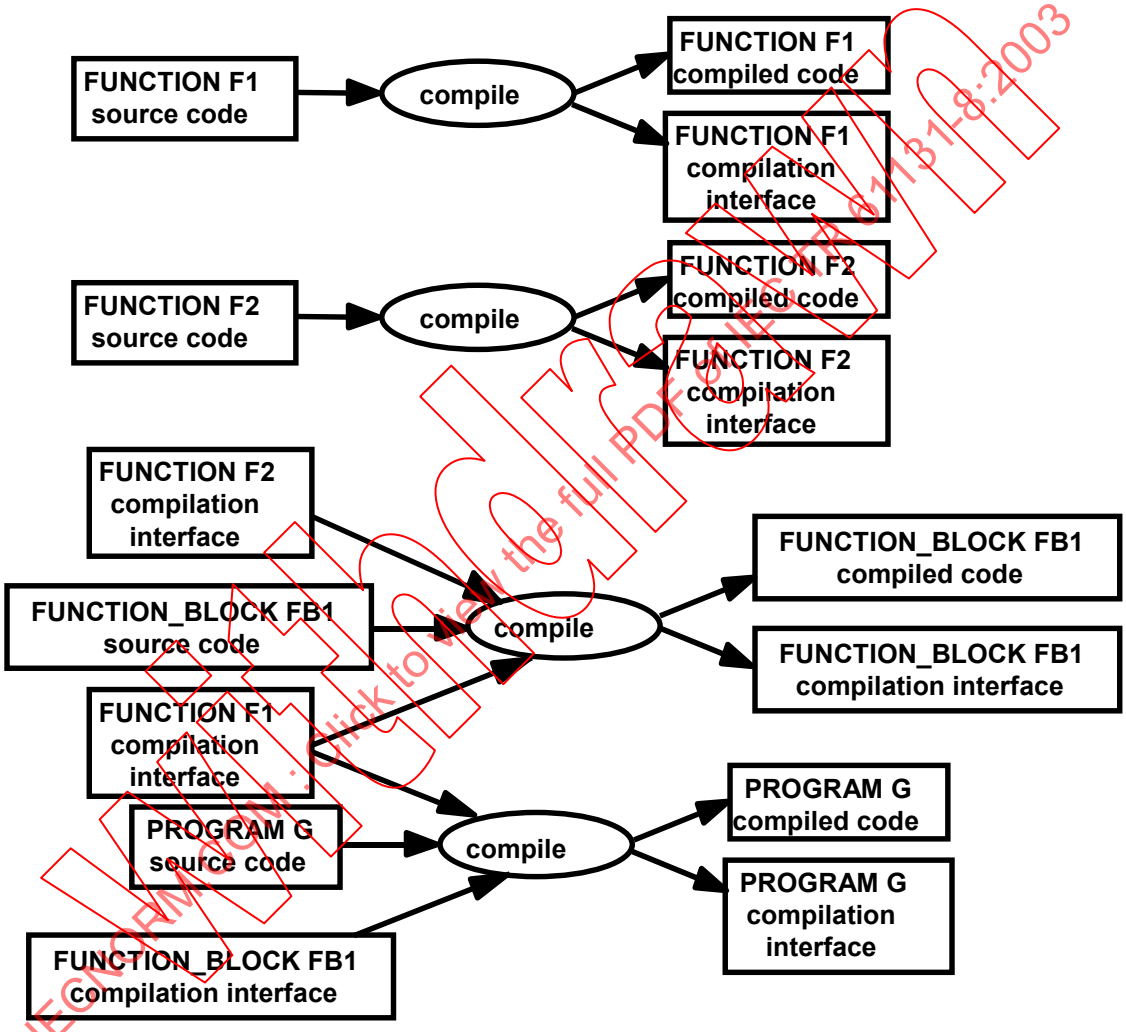
```
PROGRAM G
(* external
interface *)
...VAR toto: FB1;
...END_VAR
...(* body *)
...(* invoke F1 *)
 X:= F1(Y,Z);
...(* use of toto *)
 toto(d:= X);
...END_PROGRAM
```

```
FUNCTION F2: REAL
...
END_FUNCTION

FUNCTION F1: INT
...
END_FUNCTION
```

```
FUNCTION_BLOCK FB1
...(* external
 interface *)
...(* body *)
...(* invoke F1 *)
 A:= F1(B,C);
...(* invoke F2 *)
 Q:= F2(R);
...END_FUNCTION_BLOCK
```

Figure 26a – Sketch of programming units to be compiled



IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

Figure 26b – Separate compilation

IEC 2085/03

Figure 26 – Separate compilation example

5.6 Linking of configuration elements with programs

In an application configuration, a program is a particular named use (an instance) of a POU of program type. For example, given a program G as illustrated in 5.5, an application can be composed of two programs, P1 and P2, which are both instances of the same program G. As illustrated in Figure 27, it is only necessary to compile program G once, and to resolve the unknown links when P1 and P2 are mapped onto the configuration.

```
CONFIGURATION V
  RESOURCE R1 ON PROCESSOR_TYPE_1
    PROGRAM P1: G(...) ;
  END_RESOURCE
  RESOURCE R2 ON PROCESSOR_TYPE_2
    PROGRAM P2: G(...) ;
  END_RESOURCE
END_CONFIGURATION
```

Figure 27a – Declaration of the configuration

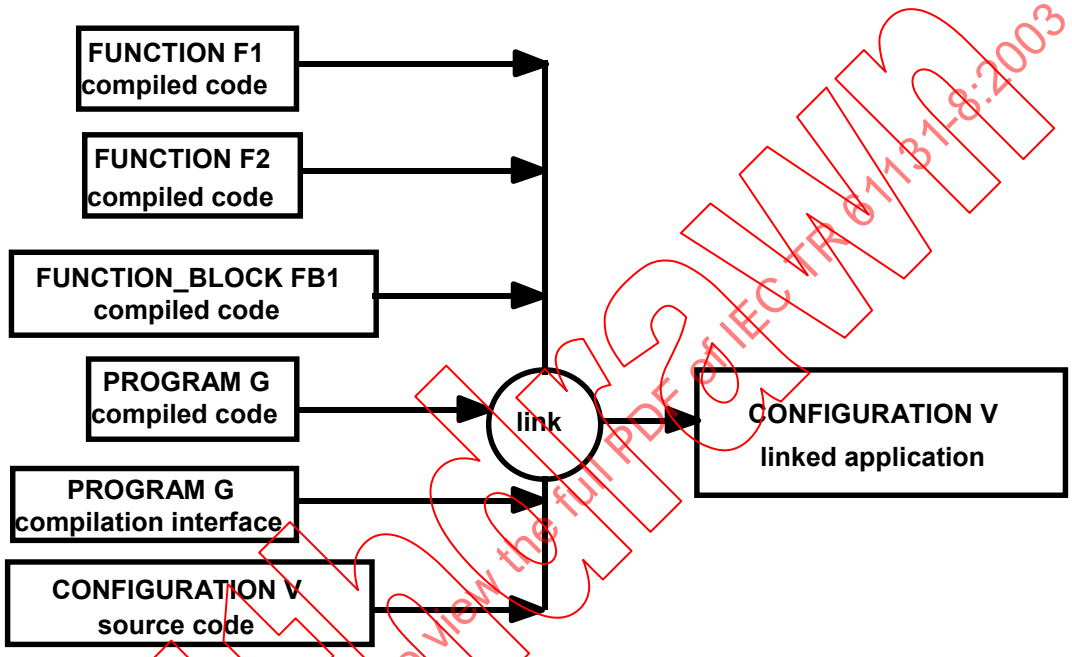


Figure 27b – Production of the linked application

IEC 2086/03

Figure 27 – The configuration process

In order to map a program into a resource, its compilation interface must be known. Similarly, to map a program that uses a global instance of a function block (declared with VAR_EXTERNAL) into a resource, the compilation interface of the function block must be known.

When the configuration description is complete, it can be merged with the POUs in the library to produce the application program for the configuration. This merging can be done in various ways. For instance, if all POUs are already separately compiled, the production of the application code can be generated as illustrated in Figure 27b). This figure represents a monolithic phase of production of the application code, which is similar to a classic link resolution. In this phase unresolved function or function block calls are resolved if possible, interface errors are detected, and links between the different POUs are established.

Many implementation-dependent options are possible in the production and use of the application program. For example,

– the application program may contain the actual address assignments to be used upon loading, or it may be relocatable, with the calculations of real addresses taking place during the transfer of the application program to the PLC;
– the application program may contain single or multiple copies of the code for different types of POUs;
– the production of the application code may be an incremental operation that takes place either off line or during the loading onto the PLC.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

Whichever option is selected, it must not affect the functional characteristics of the program as seen by the user. However, depending on the technical choices of the manufacturer, the execution time of POUs may vary, and some functions may be impossible or incomplete, for example, the management of a distributed application.

The loading of the application program into the programmable controller is dependent on the programmable controller's hardware and software implementation.

5.7 Library management

As illustrated in Figure 3 of IEC 61131-3, the user must accomplish two major tasks in the programming of programmable controller systems.

– *Re-use management*, i.e., the creation, modification, and deletion of reusable *program elements*. These consist of derived *POUs*, i.e., *functions*, *function blocks,* and (sometimes) *programs* and the associated derived *data types*. Program elements may be stored in a library that is available to several applications, or in a library of derived program elements to be used only in a specific application.

– *Application management*, i.e., the creation, modification, and deletion of one or more *configurations* to be used in each application. A *program* that is used in a *resource* of a particular *configuration* may not be reusable, particularly if it utilizes global and/or directly represented variables.

The user must be able to switch easily between these tasks, and the PSE must assist the user in keeping track of the dependencies between library elements. Also, the PSE must support the updating of libraries with minimal disruption to existing elements and dependencies in the user library.

The PSE should support the software engineering processes of top-down design by functional decomposition and bottom-up implementation by functional composition described in 2.5.2.4 of this technical report. In particular, it should be easy for the user to define the *interfaces* to functions and function blocks in an *ad hoc* manner while performing functional decomposition; to store and reuse these interface definitions in a library; and to defer the definition of the *bodies* of these program organization units until a subsequent stage in the design or implementation process.

The PSE should support the management of the separate compilation process described in 5.4 and 5.5 of this technical report by maintaining the relationships among the source code, compiled code, external interfaces, and compilation interfaces of POUs and configurations that use them.

5.8 Analysis tools

5.8.1 Simulation and debugging

User productivity and software quality can be significantly enhanced if the PSE provides facilities for simulation and debugging of software without having to be physically connected to the actual programmable controller system. Such facilities may be provided for

– individual program organization units (functions, function blocks, and programs);

– system configurations (partial or complete).

Further enhancements in productivity can be realized if the PSE provides facilities for debugging such that those portions of a configuration that are already debugged can be loaded into a controller system and operated in conjunction with simulation of those portions of the configuration that are still being developed and debugged.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

5.8.2 Performance estimation

Information to determine execution times of POUs is identified as an implementation dependency in Annex D of IEC 61131-3; additionally, item 3)b) of 2.7.2 of IEC 61131-3 requires that “the manufacturer shall provide information to enable the user to determine whether all deadlines will be met in a proposed configuration.” Since the rules for determining this information may be quite complex and difficult to apply systematically, it is to the benefit of both the implementer and the user if algorithms for making such determinations are provided as an integral part of the PSE.

5.8.3 Feedback loop analysis

When the FBD language defined in 4.3 of IEC 61131-3 is supported, the PSE should provide tools for the detection and resolution of feedback loops as discussed in 4.1.3 of IEC 61131-3.

5.8.4 SFC analysis

Subclause 2.6.5 of IEC 61131-3 mentions the possibility of unsafe SFCs and unreachable branches in SFCs. Nesting of AND (simultaneous) branches and mixing of AND and OR (selection) branches can cause such SFCs. The PSE can assist the user by providing early detection of possible unsafe and unreachable SFCs. One possible algorithm for such detection is described below.

NOTE ”Safety” in this context does not refer to the safety of the programmable controller application. However, an “unsafe” SFC can be expected to degrade the safety of the application.

The reduction algorithm is based on the provable assertion that an SFC is executable (in the sense of safe and reachable) if one derives after a number of reduction steps at an SFC consisting of one step and one transition. The possible reduction steps are illustrated in Figure 28.

The reductions are repetitively applied starting from the initial step, even if this step is not at the top of the SFC. The algorithm concludes that the SFC is safe and reachable if either of the following termination conditions are detected.

1) The SFC consists only of OR branches; or
2) the SFC contains only AND branches and the initial step is not located in one of the branches.

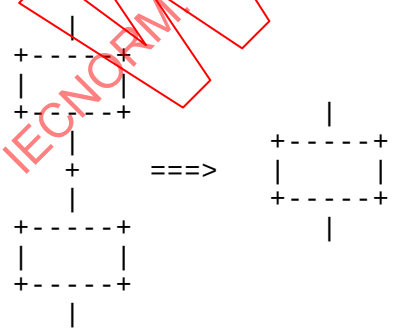


Figure 28a – A sequence consisting of a step, a transition and a step is replaced by one step

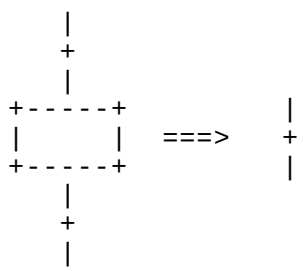


Figure 28b – A sequence consisting of a transition, a step and a transition is replaced by one transition

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

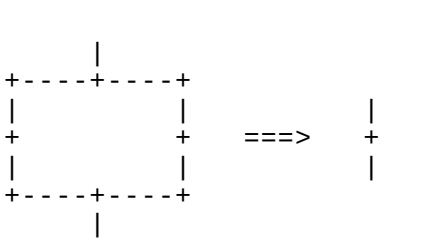


Figure 28c – A selection sequence (OR branch) is replaced by one transition

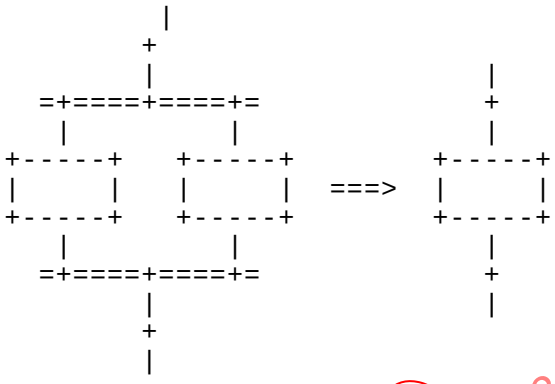


Figure 28d – A simultaneous sequence (AND branch) is replaced by one step

Figure 28e – A loop is replaced by one transition

IEC 2087/03

Figure 28 – Reduction steps

Figures 29a and 29b illustrate SFCs that are *a priori* safe and reachable by application of the termination rules (1) and (2), respectively. Figure 29c illustrates an irreducible and unsafe SFC derived from Figure 18a of IEC 61131-3 through the application of rules (a) and (b). Figure 29d illustrates an irreducible and unreachable SFC similarly derived from Figure 18b) of IEC 61131-3. In both cases, the algorithm will produce a warning since no further reduction is possible but neither termination condition applies.

NOTE This algorithm can detect all possible unsafe and unreachable SFCs; however, there are cases where this algorithm rejects complex but still valid SFCs. That is, this algorithm provides a sufficient but not necessary condition for safe and reachable SFCs.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

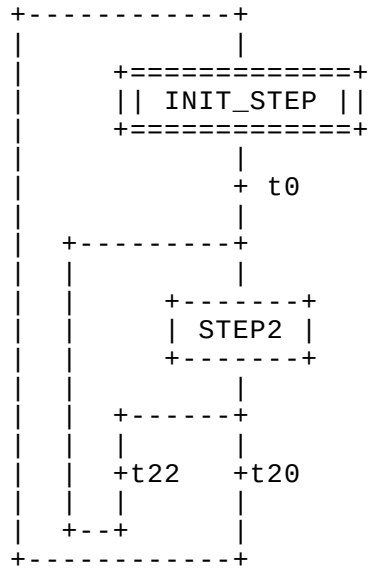


Figure 29a – No reduction required per termination rule 1

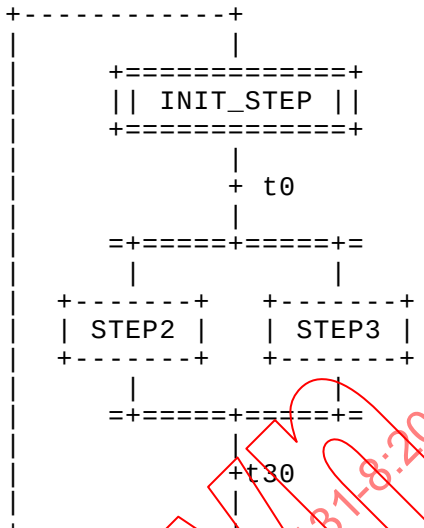


Figure 29b – No reduction required per termination rule 2

Figure 29c – An irreducible (unsafe) SFC

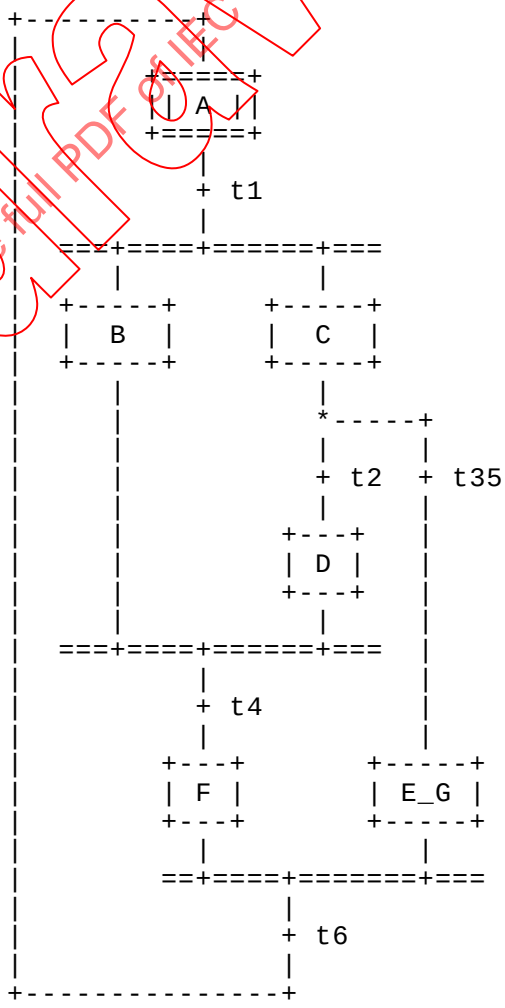


Figure 29d – An irreducible (unreachable) SFC

IEC 2088/03

Figure 29 – Reduction of SFCs

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

5.9 Documentation requirements

In providing facilities for the production of software documentation by the PSE, the implementer should consider the following requirements.

1) Listings of the declarations and bodies of POUs should be produced in the same form in which they were entered by and displayed to the user on the PSE.
2) Listings should include comments in the same form and location in which they were entered by and displayed to the user.
3) The PSE should be capable of producing an index and cross-reference of usage of global and directly represented variables and access paths (if any).

The PSE should be capable of producing a mapping between the symbolic representations of variables and their physical locations (for example, in the I/O subsystem).

As an implementation-dependent feature, the PSE may also be capable of handling the mapping between the directly represented variables of a configuration and reference designations for signals and tags, as defined in IEC 61346-1.

6) If supported, version control information should be provided with the listings.

5.10 Security of data and programs

The implementer should consider

– which items in the PSE and the programmable controller system should be capable of protection from user access; and
– which classes of users should have access to the various items in the system.

For instance, the following types of access protection (among others) may be specified.

– Permission to use the contents of a specified library.
– Permission to use the interface to a specified POU.
– Permission to view the body of a specified POU.
– Permission to modify the body of a specified POU.
– Permission to modify the interface of a specified POU.

5.11 On-line facilities

Facilities which may be provided by a PSE when connected to a programmable controller system are described in the following subclauses of IEC 61131-1.

– Loading of application programs (4.6.2.1)
– Memory access (4.6.2.2)
– Adapting the programmable controller system (4.6.2.3)
– Indicating the automated system status (4.6.2.4)
– Testing the application program (4.6.2.5)
– Modifying the application program (4.6.2.6)
– Archiving the application program (4.6.4)

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

Annex A
(informative)

Changes to IEC 61131-3, Second edition

A.1 Reasons for the second edition of IEC 61131-3

Since the publication of the first edition of IEC 61131-3 in 1993, the environment of the standard has changed greatly. During this time, 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 IEC 61131-3 users and to extend the usefulness of existing IEC 61131-3 compliant control software as far as possible for the future, IEC SC65B 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 the standard (corrigendum)

Step 2) Amelioration of specific items in need of improvement within the standard and the integration of features regarded as particularly relevant in practice (amendment)

For every individual item to be altered, changes must be upward-compatible, i.e. as a rule, a program compatible with the current standard must also be in accordance with the new one.

A.2 Corrigendum

The first step was intended to eliminate real errors within IEC 61131-3. In addition to simple misprints, this includes especially semantic contradictions in the main part and inconsistencies between the main part and the annexes, in particular the syntax definition in Annex B. During the first phase, the changes to be incorporated were collected and evaluated by the IEC task force in a corrigendum. This was achieved with the active help of many experts.

In the following list, the most important corrections are briefly described giving the clause number of the clauses which are almost identical in the two editions of IEC 61131-3.

1.3 Some terms which are written in italics in the text but not previously defined are now included in this “Definitions” subclause.

1.4.1 Correction of the figure which explains the software model

2.3.3.2 Corrections in the syntax of examples, especially concerning the use of parentheses (Tables 14, 17, 18, 22, 50, Figure 20).

2.3.3.2, 2.3.3.3 Semantic corrections of examples (Tables 14, 17, 18).

2.4.1.2 Language-specific treatment of array subscripts according to the language definitions for IL, ST, LD and FBD.

2.5.1.1 Correction of the general variable assignment description of function calls in ST (Table 27).

2.5.1.4, 2.5.2.2 More precise definition of the applicability of overloading to standard functions, function block types, operators and instructions (Table 21).

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

2.5.1.5.1 Precise description of the operation of the type conversion function features according to IEC 60559 (rounding to the next integer value, Table 22).

2.5.1.5.3 Correction of the variable assignment description of standard functions (Tables 25, 29, 30, 31).

2.5.1.5.7 The symbol for “is equal to” is “=” and not “–” (Table 31).

2.5.2. Correction of description of the function block AND_EDGE (rising edge input, Table 33).

2.5.2.3.1 Deleting of the unsafe semaphore example (Figure 13, Table 34).

2.5.2.3.3 Correction of the CTUD example (Table 36).

2.5.2.3.4 Correction of the RTC example (Table 37).

2.6.3 Syntax correction of the SFC description (Tables 41 and 42).

2.6.4.5 With an active Q output of the action control block, SFC actions are executed only with every invocation of the POU and not permanently; thus, the contradiction to rule 6 in 2.7.2 is resolved.

4.3.3 Elimination of the possible deadlock situation during mutual reading and writing of variables in different networks (Annex D).

Annex B Correction of BNF productions.

Annex C Correction of delimiters and keywords.

Annex F Correction of the examples.

None of these corrections leads to an incompatibility of existing control software in accordance with the current standard with the “corrected” IEC 61131-3.

A.3 Amendment

A.3.1 Background

As a second step for the evaluation of possible standard improvements, a Type 2 technical report was prepared soon after the official release of IEC 61131-3. This technical report proposed trial-use extensions to the programming languages with the goal of arriving at a consensus on a common set of features to be defined in a future edition of IEC 61131-3. These proposals could conflict with each other, with the normative provisions of IEC 61131-3, or both.

Using this technical report as a starting point, the improvements which appeared most urgent were included in an amendment. These amendments were to be integrated in the standard on the basis of the IEC 61131-3 corrections in step 1. They only refer to improvements of the utmost importance.

The committee draft of the amendment was divided into two main groups of alterations and supplements and a collection of minor changes.

The general goals of all changes in the amendment were

- an increased acceptance of the IEC 61131-3 languages by the application programmers due to harmonized language possibilities which are better adapted to practice, for example, new function concept and function block instance-specific initialization of local variables;
- a more efficient utilization of the control hardware by means of application programs written in the standard languages, for example, temporary variables in function blocks;
- increased actual semantic portability of control software due to the elimination of possible misunderstandings concerning the effects of certain language constructs, for example, EN/ENO behaviour, retentive/non-retentive behaviour of variables and I/Os;

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

- elimination of error sources due to the mixed use of different control languages, for example, identical treatment of EN/ENO by different languages.

NOTE The parenthesized references in the headings of the following subclauses refer to the subclause of IEC 61131-3, 2nd edition, that is mainly affected by the changes described; for instance, A.3 below describes changes mainly affecting 2.2.1 of IEC 61131-3, 2nd edition.

A.3.2 Numeric literals (2.2.1) – typed literals

In the first edition, there are situations where the type of data represented by literal values such as 12.43, 73, 2#1001 is ambiguous. This is a particular problem in the IL language; for example, when a literal value is used with a load (LD) instruction, the type of data loaded into the IL register cannot be specified.

The amendment proposed that all literals can be prefixed with their data type in the form '<data type> #'. This feature is added to 2.2.1 of IEC 61131-3, 2nd edition.

A.3.3 Elementary data types (2.3.1) – double-byte strings

The amendment proposes an additional data type WSTRING to allow strings to be defined to hold double-byte characters in accordance with ISO/IEC 10646. These are required for handling messages in languages with complex character sets, such as Japanese. The string handling functions will be overloaded so that they can be used with double byte strings. This feature is added to 2.3.1, 2.3.2 and 2.5.1.5.5 of IEC 61131-3, 2nd edition.

A.3.4 Derived data types (2.3.3) – enumerated data types

There are several deficiencies with the use of enumerated data types in the first edition of IEC 61131-3. As with literals, the data type of a particular enumerated value is ambiguous. For example, there may be several enumerated type definitions using the enumeration string 'ON'. It was therefore proposed in the amendment and included in 2.3.3.1 of IEC 61131-3, 2nd edition, that the data type of a particular enumeration literal can be specified using a prefix in the form <data type>#.

Examples are:

```
TYPE
 VALVE_MODE: (OPEN, SHUT, FAULT);
 PUMP_MODE: (RUNNING,OFF,FAULT);
END_TYPE;
...
IF AX100 = PUMP_MODE#FAULT THEN
        XV23 = VALVE_MODE#OPEN;
```

The syntax of structured text in Annex B of IEC 61131-3, 2nd edition is extended to allow enumerated variables to be used in CASE statements, for example:

```
TYPE
 BATCH_TYPE: (SMALL, LARGE, CUSTOM);
END_TYPE;
VAR
 NEW_BATCH: BATCH_TYPE;
END_VAR;
...
CASE NEW_BATCH OF
 SMALL: ... (* Small batch *)
 LARGE: ... (* Large batch *)
 CUSTOM: ... (* Custom size batch *)
ELSE
...
```

A.3.5 Single element variables (2.4.1.1) – "wild-card" direct addresses

The definition of direct addresses for some variables, i.e. addresses used with the AT construct, is another aspect of a configuration which may require redefinition when a configuration is modified to work on a different physical system.

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

For example, a sensor connected to a programmable controller input may be at a particular physical input channel and I/O rack position in one system but be connected to completely different physical input channel and rack in another. In order to use the same software, i.e. the same IEC 61131-3 configuration in the two systems, it is necessary to redefine the direct address used for the input. In the first edition of IEC 61131-3, changing direct addresses of variables can only be achieved by modifying various variable declarations throughout the configuration and then recompiling the configuration.

The amendment proposed that each variable for which the direct address is to be redefined is declared using a 'wild-card' address specified as an asterisk ('*'). Such variables are considered to be 'not located'. This feature is now included in 2.4.1.1 of IEC 61131-3, 2nd edition.

Examples are:

```
VAR INPUT1 AT %IX*:BOOL;(*Boolean input not located *)
VAR VALV1 AT %QW*: INT;(*Integer output not located *)
```

The location of such variables can be specified in the VAR_CONFIG construct. For example, the following statements will define the location of these variables; assume that INPUT1 and VALV1 are declared within resource RES1.

```
VAR_CONFIG
 RES1.INPUT1 AT %IX100: BOOL;(* Locate input1 *)
 RES1.VALV1 AT %QW210: INT;(* Locate valve 1 *)
END_VAR
```

It is assumed that initial values for the variables specified in the VAR_CONFIG will be applied as the last process before creating the program object data that is downloaded into the PLC.

NOTE An error will be reported when the configuration is built if any variable declared with an unlocated direct address is not given a valid direct address.

A.3.6 Declaration (2.4.3) – Temporary variables

In the first edition of IEC 61131-3, there was no provision to create variables within programs and function blocks to hold temporary values. Values held in variables declared using the VAR construct within POUs always persist between POU invocations. Using such variables for temporary values can result in an inefficient use of memory.

The amendment proposed, and 2.4.3 of IEC 61131-3, 2nd edition provides, that temporary variables can be declared using a VAR_TEMP construct. Such variables will be placed in a temporary memory area, such as on a stack, which is cleared when the POU invocation terminates.

For example:

```
VAR_TEMP
 RESULT: REAL;
END_VAR;

RESULT:= AF18 * XV23 * XV767 + 54.2;
OUT1:= SQRT(RESULT);
```

A.3.7 RETAIN and NON_RETAIN Variable attributes (2.4.3.1)

The use of the RETAIN attribute with multi-element variables such as structures and function block instances is modified in IEC 61131-3, 2nd edition. When RETAIN is used with a multi-element variable it applies to all contained variables except those that are declared with a NON_RETAIN attribute.

NON_RETAIN is a new attribute and indicates that the variable's value is not retained during a powerfail and its default initial value is to be used after a warm restart. If NON_RETAIN is

IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003
Withdrawn

applied to a multi-element variable, then it applies to all contained variables that are not declared with a RETAIN attribute.

A.3.8 Invocations and argument lists of functions (2.5.1)

IEC 61131-3, 1st edition, had introduced two variants for function calls and one variant for function block invocations in textual languages.

For function calls, the variant to be used depended on the declaration of the respective function: if the declaration of a function does not explicitly specify input variable names, the function had to be called by providing the full set of actual input arguments in the parenthesized argument list. This was true, for example, for all extensible standard functions defined in 2.5.1.5 of IEC 61131-3, but also for some of the non-extensible functions. If the declaration of a function specifies input variable names, the function had to be called with an argument list, which contained assignments of actual input arguments to the declared input variables. Since user-defined functions always declare input variable names, they had to be generally invoked according to the latter variant.

This was also true for function block invocations. Their argument list generally had to contain assignments of actual input arguments to the declared input variables.

The rules in IEC 61131-3, 1st edition, had two implications.

a) There was a different handling of function calls and function block invocations from the user's point of view.

b) Some features available in the graphical languages were not available in the textual languages: extensible functions (for example ADD, MUL) could be used with EN/ENO in the graphical languages, but not in the textual languages – no assignment of EN or ENO was possible in ST or IL.

IEC 61131-3, 2nd edition, overcomes these deficiencies by introducing identical conventions for invocations of functions and function blocks. These changes occur mainly in subclause 2.5 of IEC 61131-3, 2nd edition and its subclauses. A detailed description of the additional features and their application is given in 3.2.3 of this technical report.

A.3.9 Type conversion functions (2.5.1.5.1)

In the first edition of IEC 61131-3, there are a number of BCD (binary coded decimal) data type conversion functions where the name of the function is not consistent with the data type of the initial or converted variable. Subclause 2.5.1.5.1 of IEC 61131-3, 2nd edition now provides BCD conversion functions that include the data type of the BCD value in their name.

Examples of some new BCD type conversion functions are:

WORD_BCD_TO_UINT()	Converts a word bit string containing a BCD value to an unsigned integer.
UINT_TO_BCD_DWORD()	Converts a unsigned integer to a BCD value in a double-word bit string.

Data type conversion functions are provided for BCD values held in BYTE, WORD, DWORD and LWORD variables.

A.3.10 Functions of time data types (2.5.1.5.6)

Subclause 2.5.1.5.6 now defines functions of time data types that resolve conflicts that existed in the previous edition with the rules for overloading functions for time and date calculations. Except as noted in A.4, the changes have been made 'backward compatible' and do not impact existing programming systems.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC TR 61131-8:2003